IEMS5722
Mobile Network Programming and Distributed Server Architecture

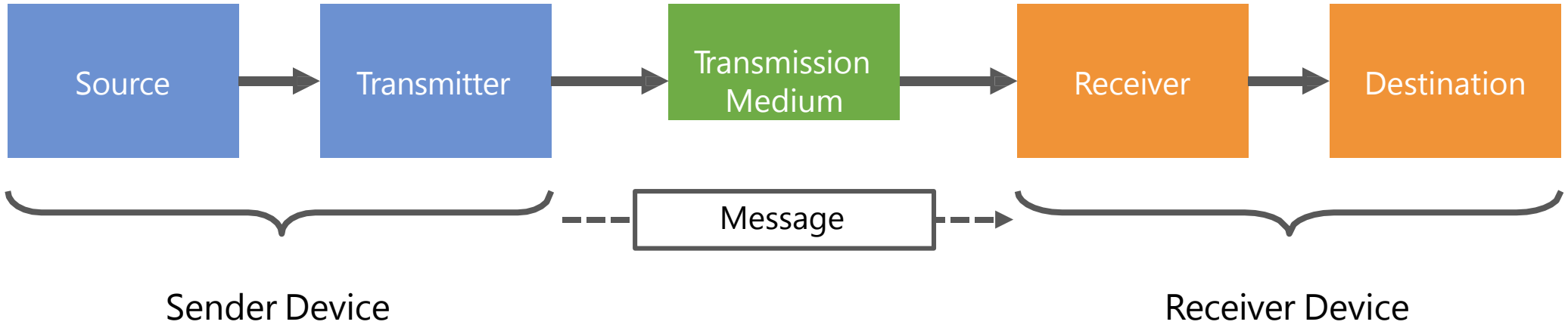
Lecture 3

Data Communication and Client-Server Architecture

Data Communication & Network Protocols

Data Communication

- Exchange of data between two devices using some form of transmission medium
- A simplified communication model:

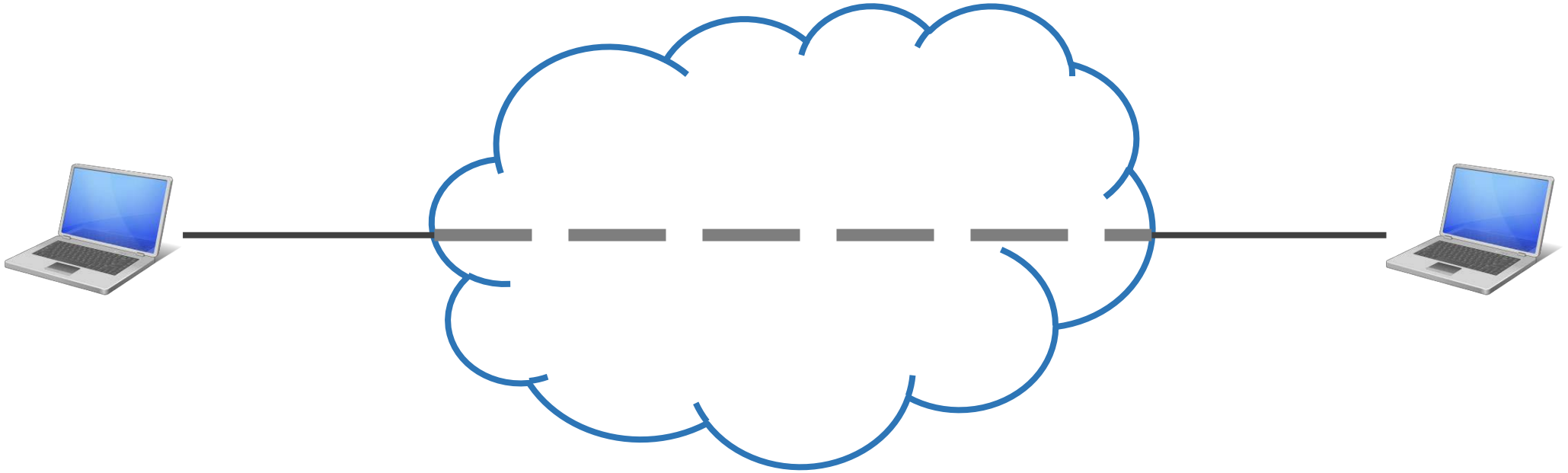


- Protocols: rules that govern how data is transmitted in this system

Method of Communication

Switching

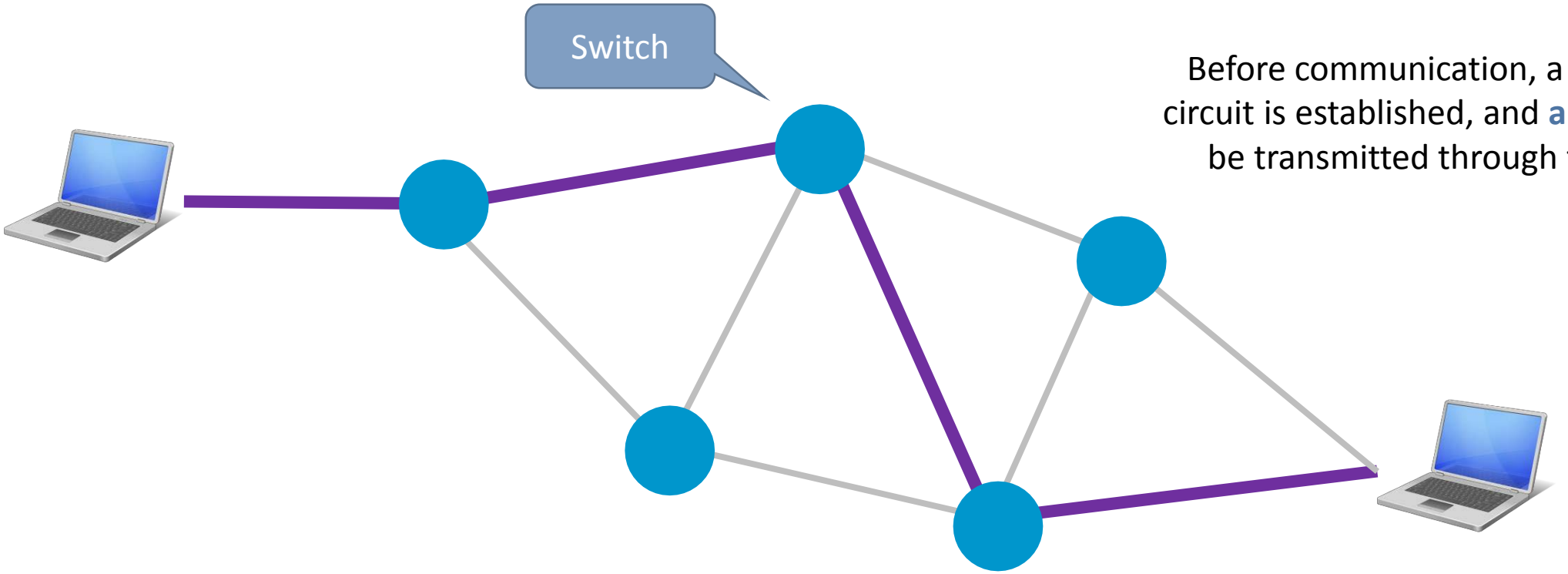
- When two computers need to communication over a network, we need to know how to connect them to each other



Method of Communication

Circuit Switching

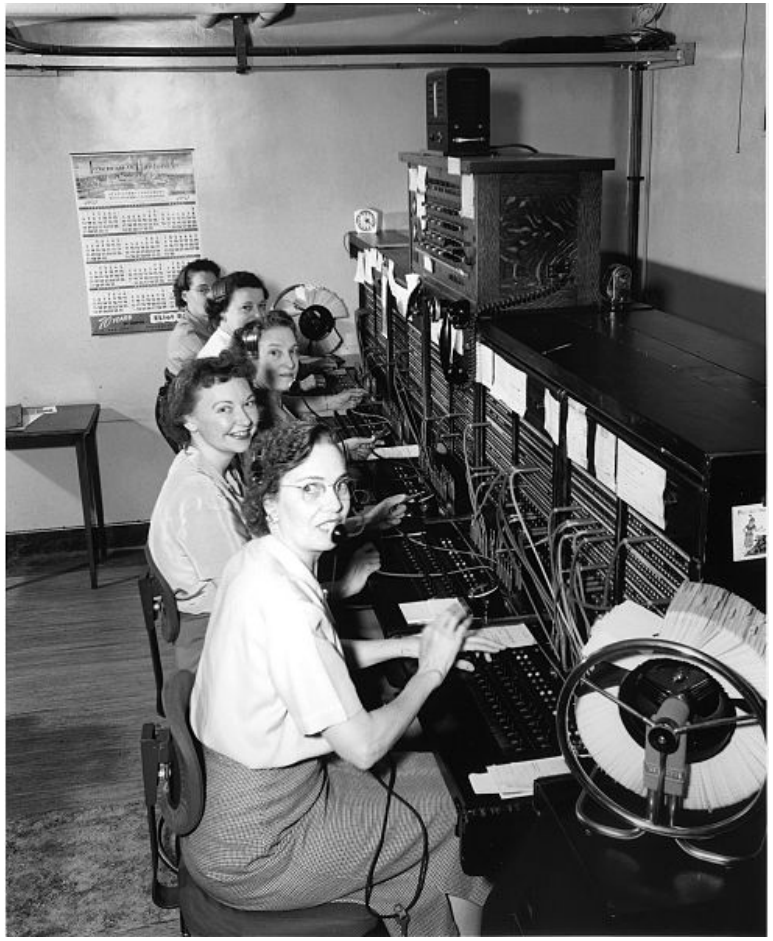
- To establish a dedicated communication link (a circuit) between two computers when they need to talk to each other



Before communication, a dedicated circuit is established, and **all data** must be transmitted through this **path**

Method of Communication

The Analog Telephone Network as a Classic Circuit Switching Example

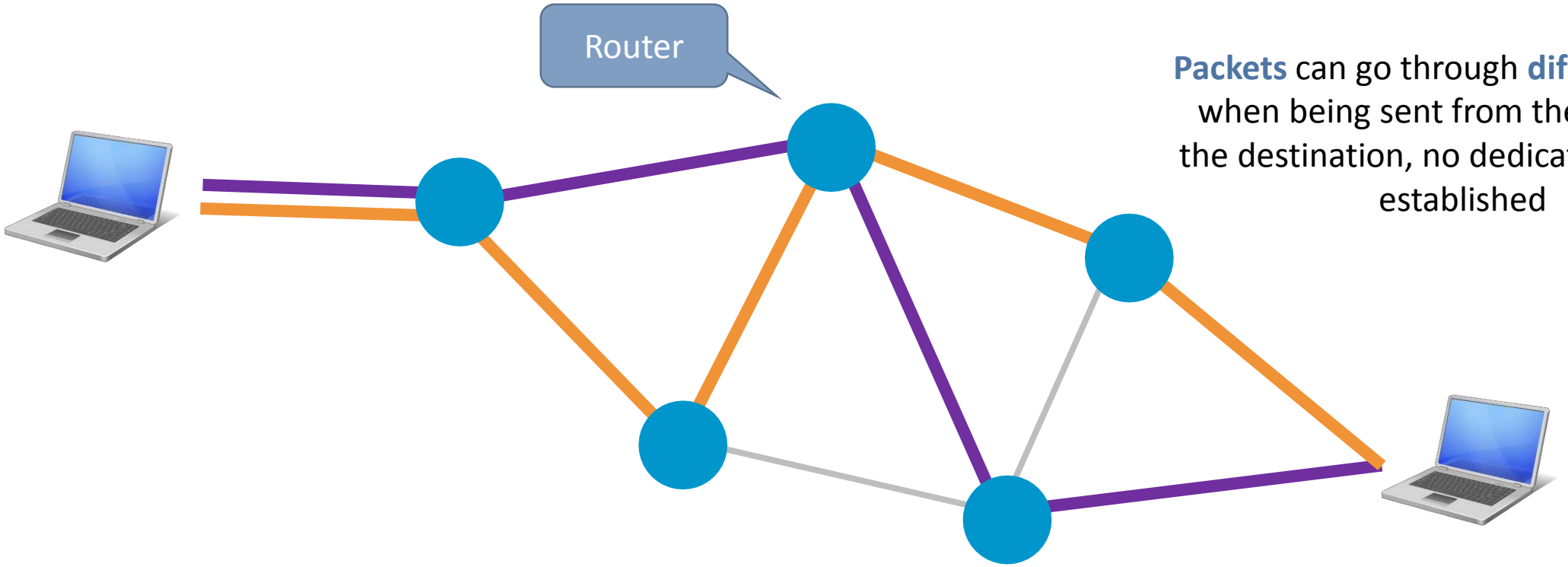


Switchboard Operators
Reference: https://en.wikipedia.org/wiki/Switchboard_operator

Method of Communication

Packet Switching

- Data is broken down into small pieces (packets), and are sent to the destination through the network through all possible paths



Packets can go through **different paths** when being sent from the source to the destination, no dedicated circuit is established

Packet Switching

Advantages of **Packet Switching**

- The network can be used in a more **efficient** way
  (The same link can be shared by many different connections)
- More **fault tolerant**
  (Consider when a switch is broken in the middle of the communication)

Protocols

What are **protocols**?

- A set of **rules** that govern how communication parties **interact** with each other
- An **agreement** between the communicating entities
- Two devices need to agree on **common protocols** when they communicate

Some of the issues that a protocol should cover

- The format of the **addressing** scheme
- How do we specify the **start** and **end** of a data stream?
- How do we handle **errors** or **data loss**?
- How to handle **problems** in data transfer?

Layered Architecture

Beethoven is writing a letter to Einstein…

Write a letter, put the letter in an envelope with a stamp, drop into the post box

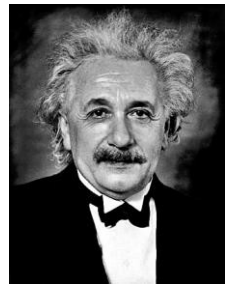
The letter is collected and brought to the post office

Letter is delivered to the carrier to be shipped

The letter is received by the post office from the carrier

The letter is delivered to Einstein's mail box by the postman

The letter is picked up and removed from the envelope and read



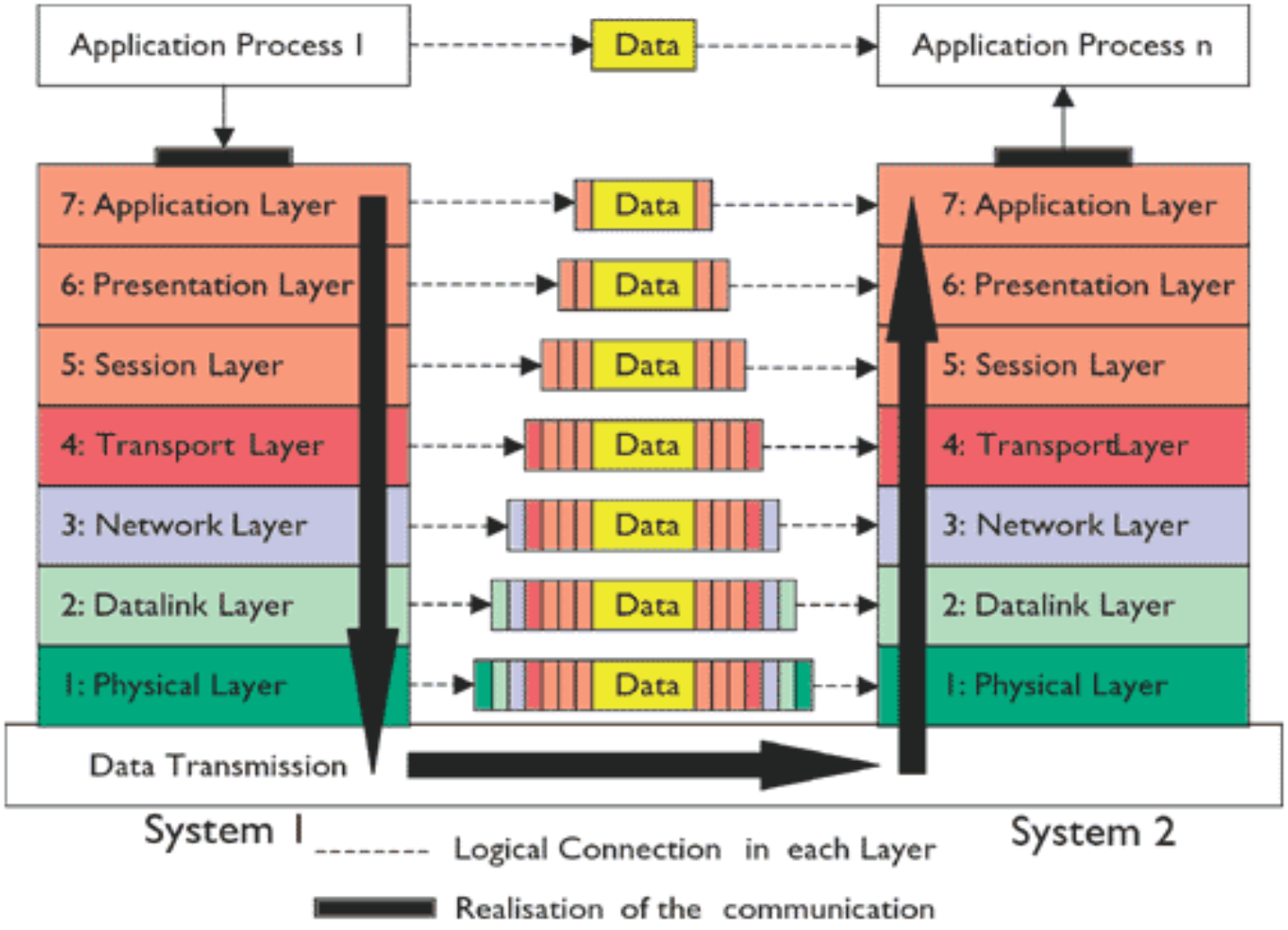
Layered Architecture

The ISO OSI (Open System Interconnection) 7-Layer Model

- A **theoretical** model of how a computer network should work
- It organizes different functions of a network into **7 different layers**
- It specifies the **interfaces for communication** between different layers and different endpoints
- **Note**:
    1. It is a theoretical model
    2. It is not a program or software
    3. Practical networks may be implemented in a different way

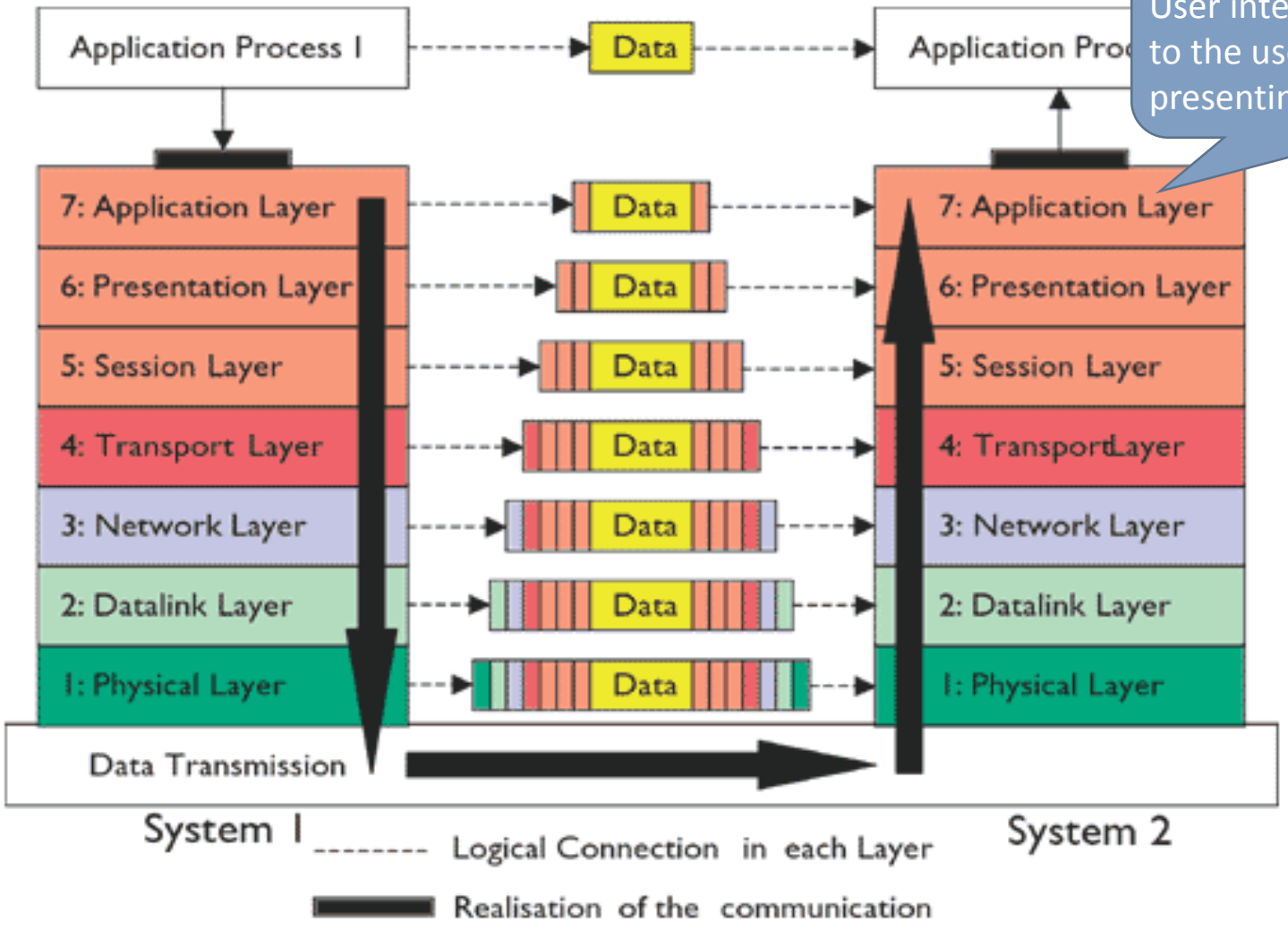
Layered Architecture

ISO OSI 7-Layer Architecture



Layered Architecture

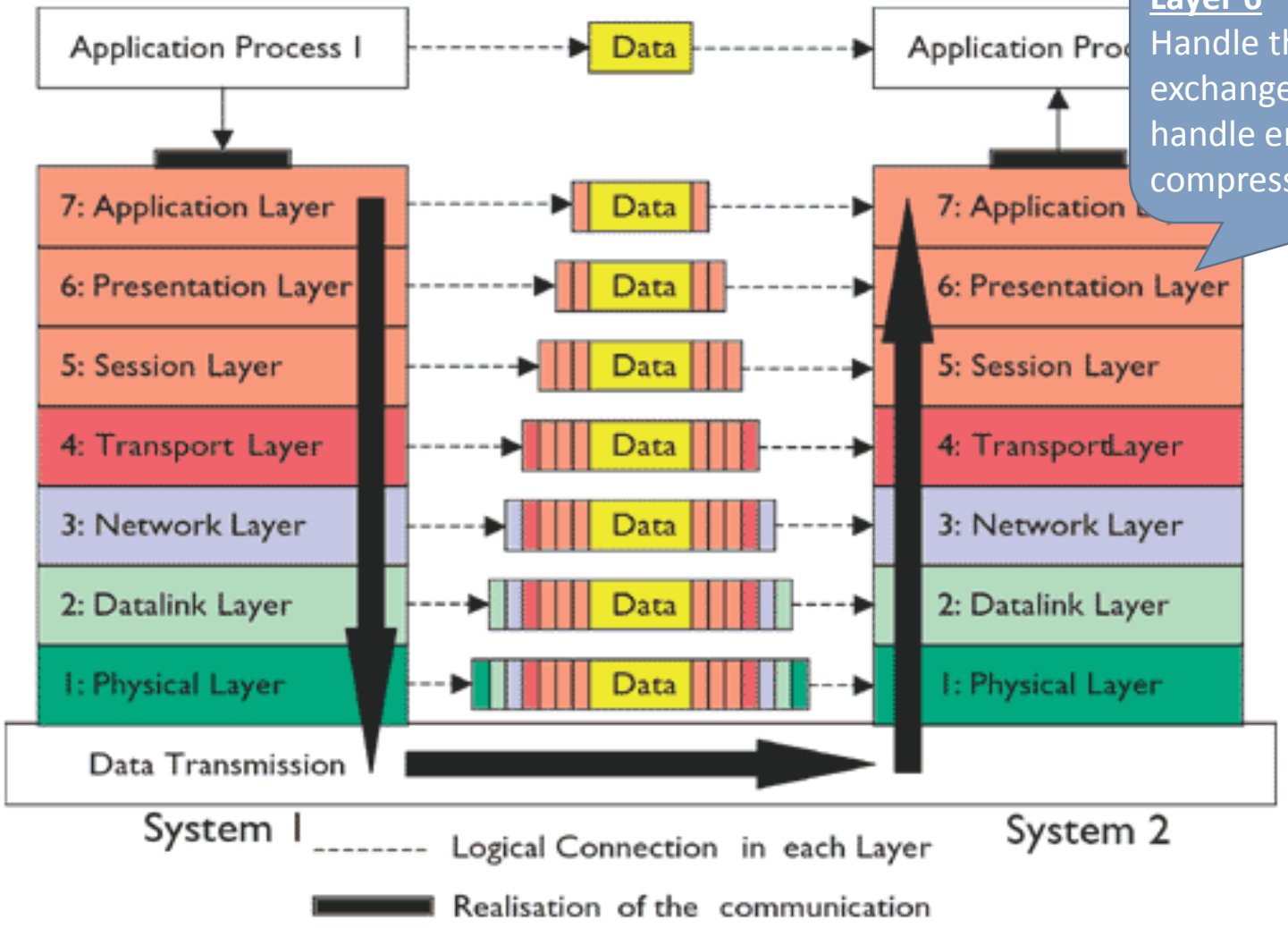
ISO OSI 7-Layer Architecture



Layer 7
User interface, providing services to the user, collecting user input, presenting information, etc.

Layered Architecture

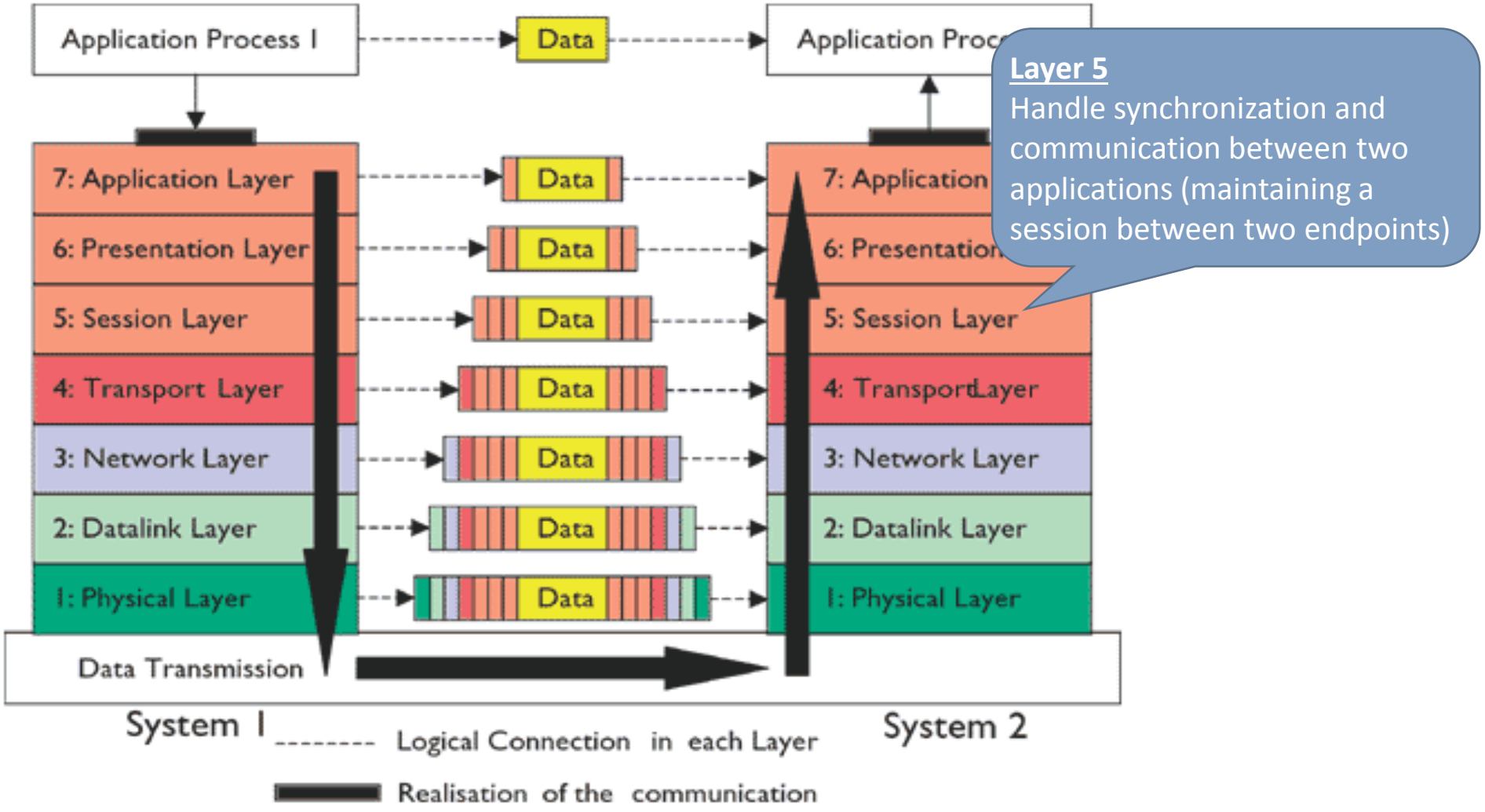
ISO OSI 7-Layer Architecture



Layer 6
Handle the semantics of the data exchanged by the applications, handle encryptions, compressions, translation, etc.

Layered Architecture

ISO OSI 7-Layer Architecture

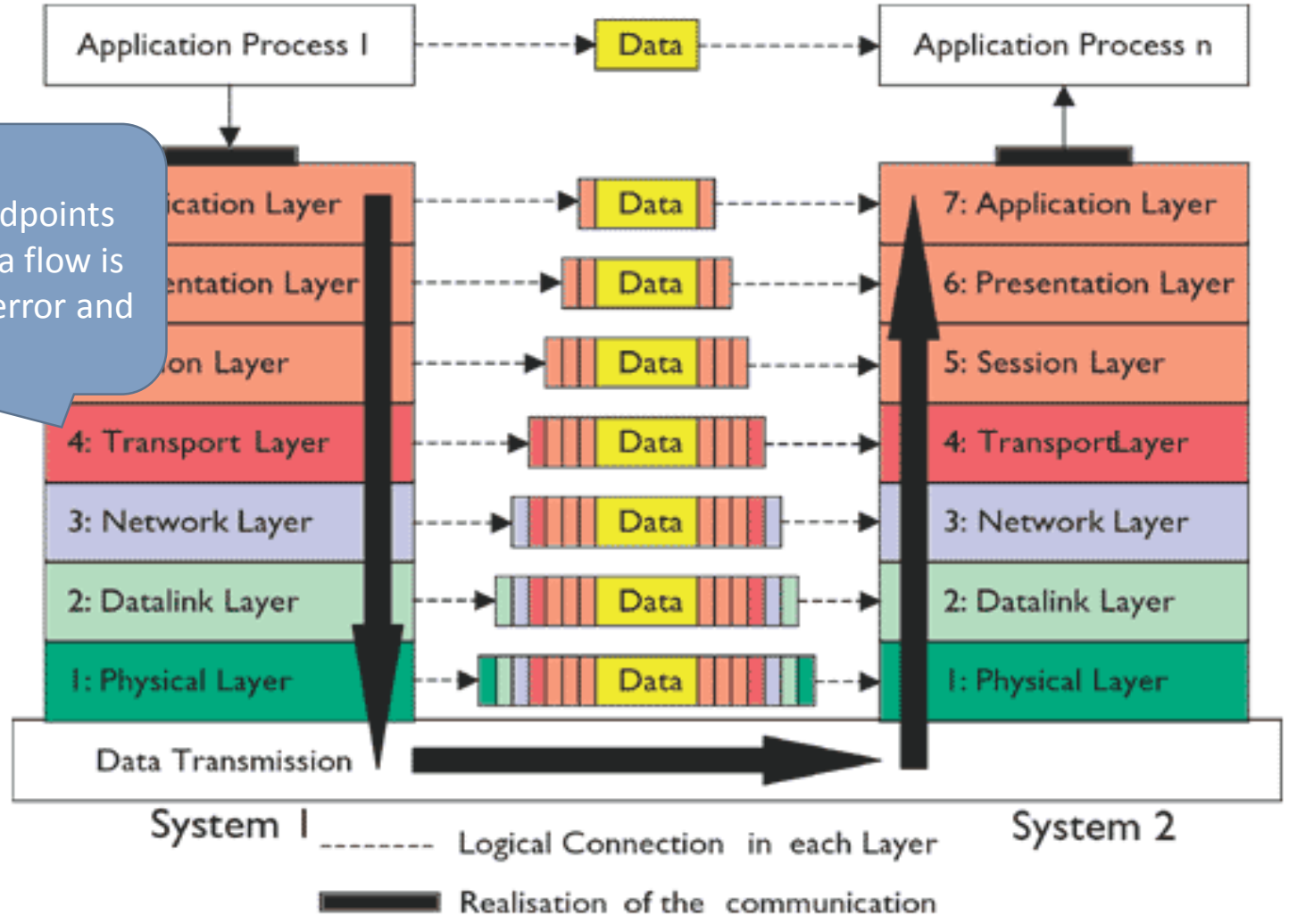


Layer 5
Handle synchronization and communication between two applications (maintaining a session between two endpoints)

Layered Architecture

ISO OSI 7-Layer Architecture

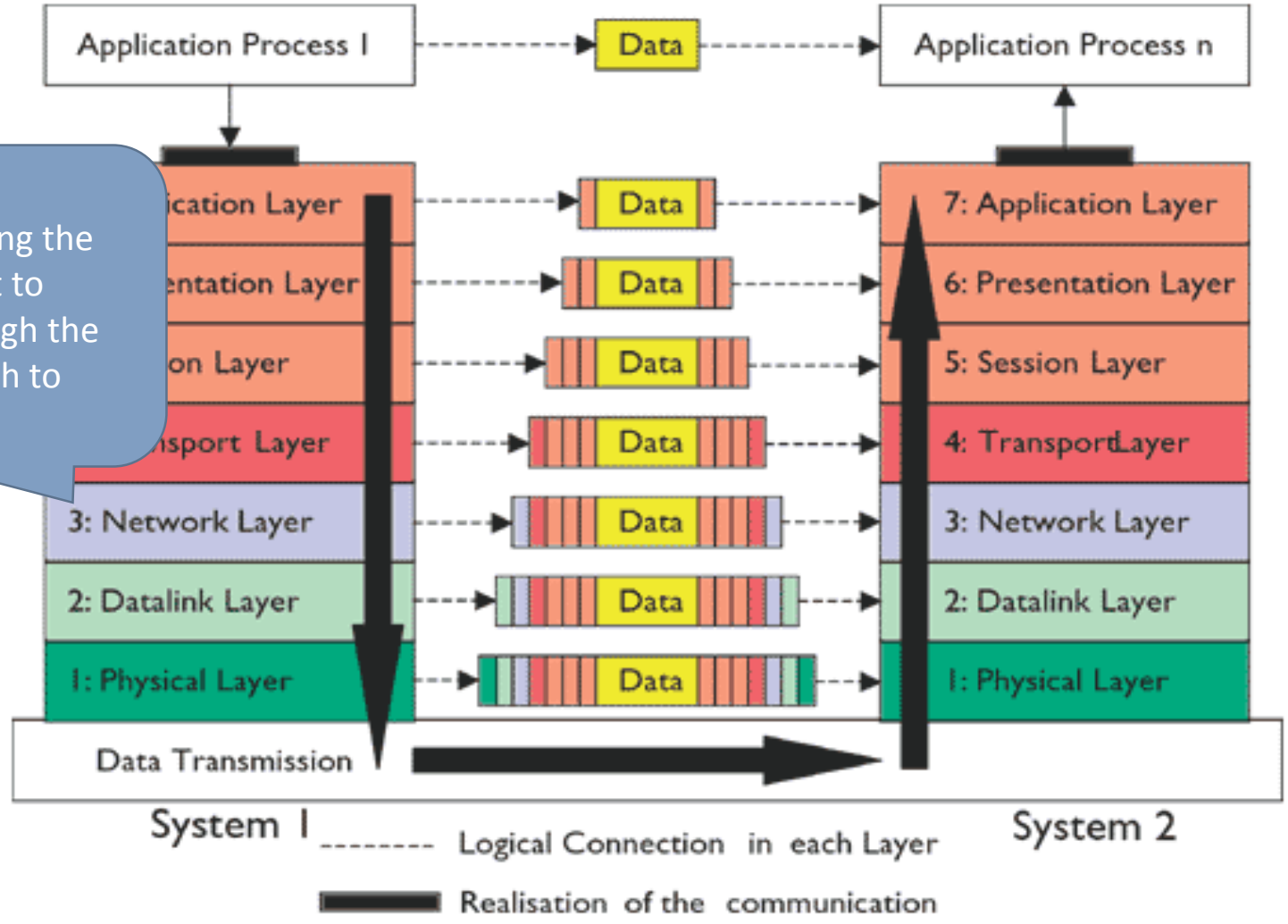
Layer 4
Control how the two endpoints are connected, how data flow is controlled, and handle error and missing data



Layered Architecture

ISO OSI 7-Layer Architecture

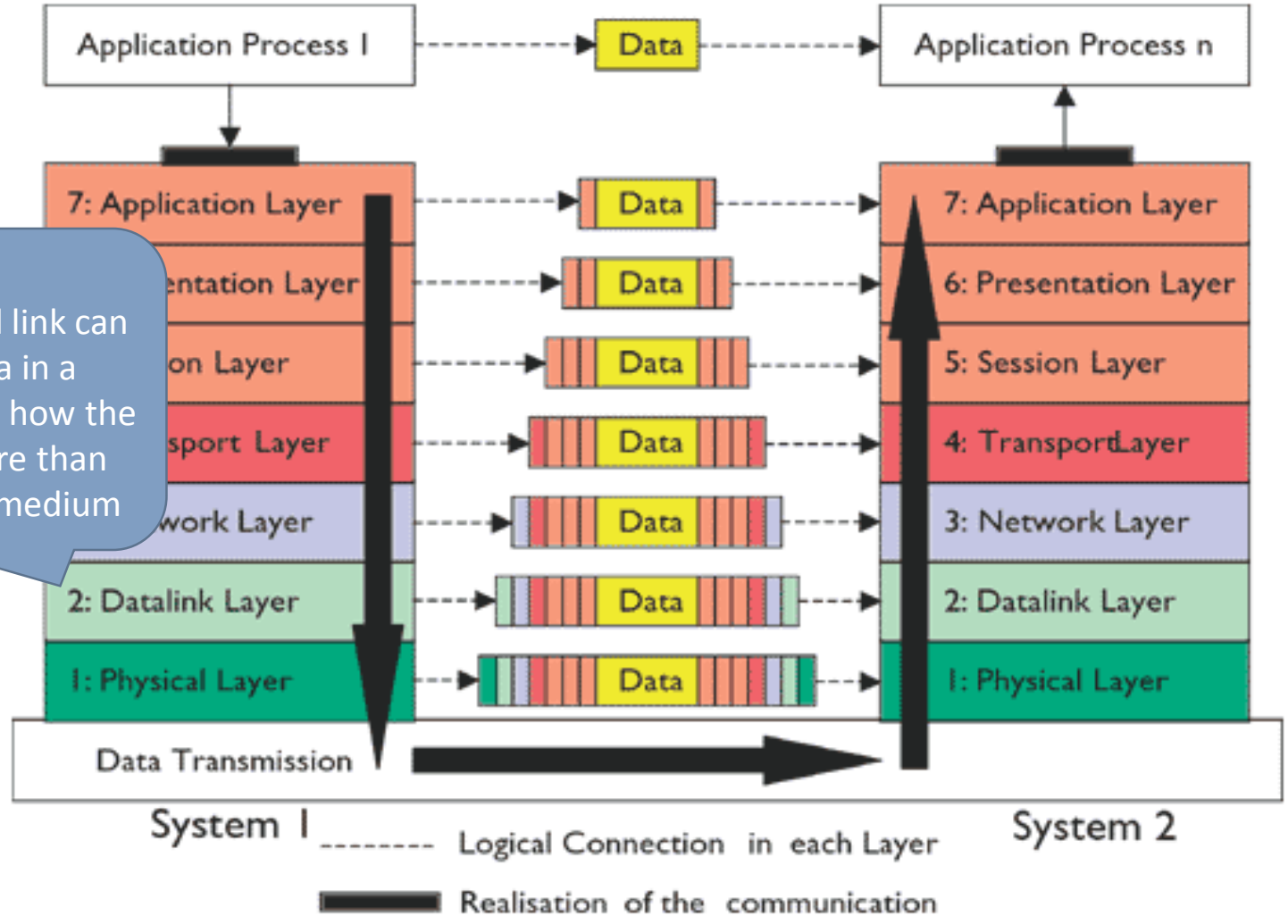
Layer 3
Responsible for delivering the data from one endpoint to another endpoint through the network (e.g. which path to take?)



Layered Architecture

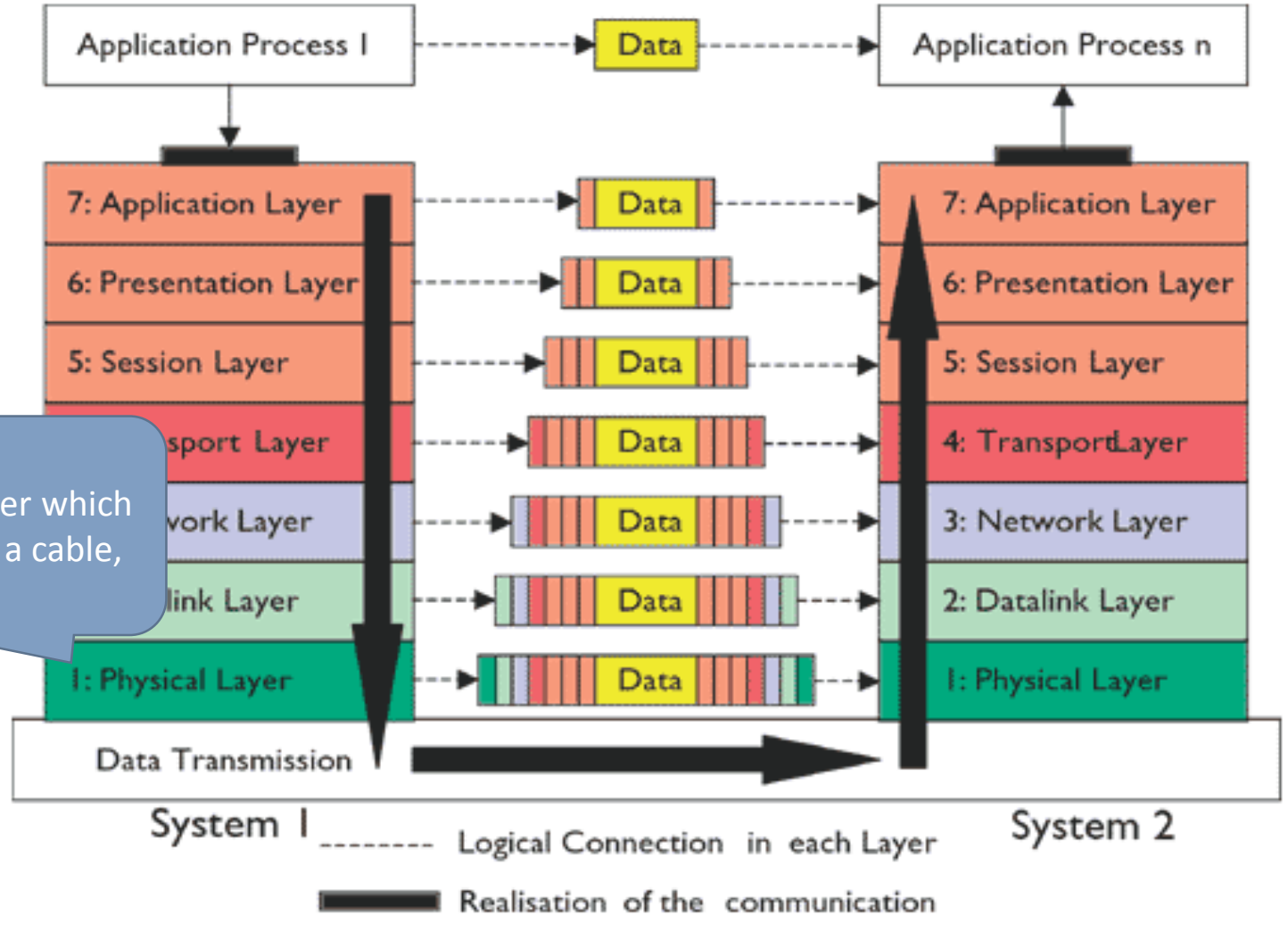
ISO OSI 7-Layer Architecture

Layer 2
Ensure that the physical link can be used to transmit data in a reliable way, controlling how the link is utilized when more than one device is using the medium



Layered Architecture

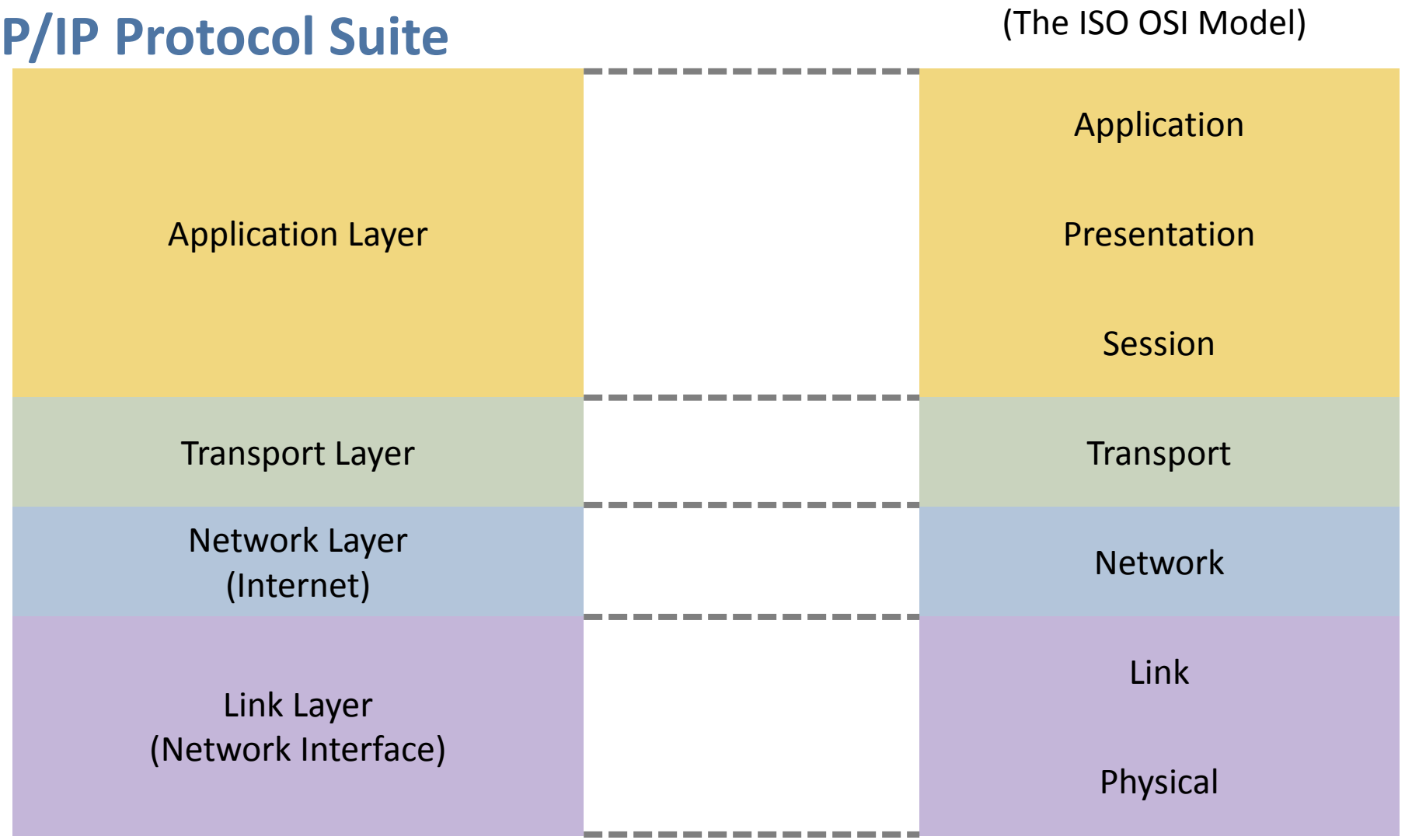
ISO OSI 7-Layer Architecture



Layer 1
The physical medium over which data is transmitted (e.g. a cable, an optical fiber)

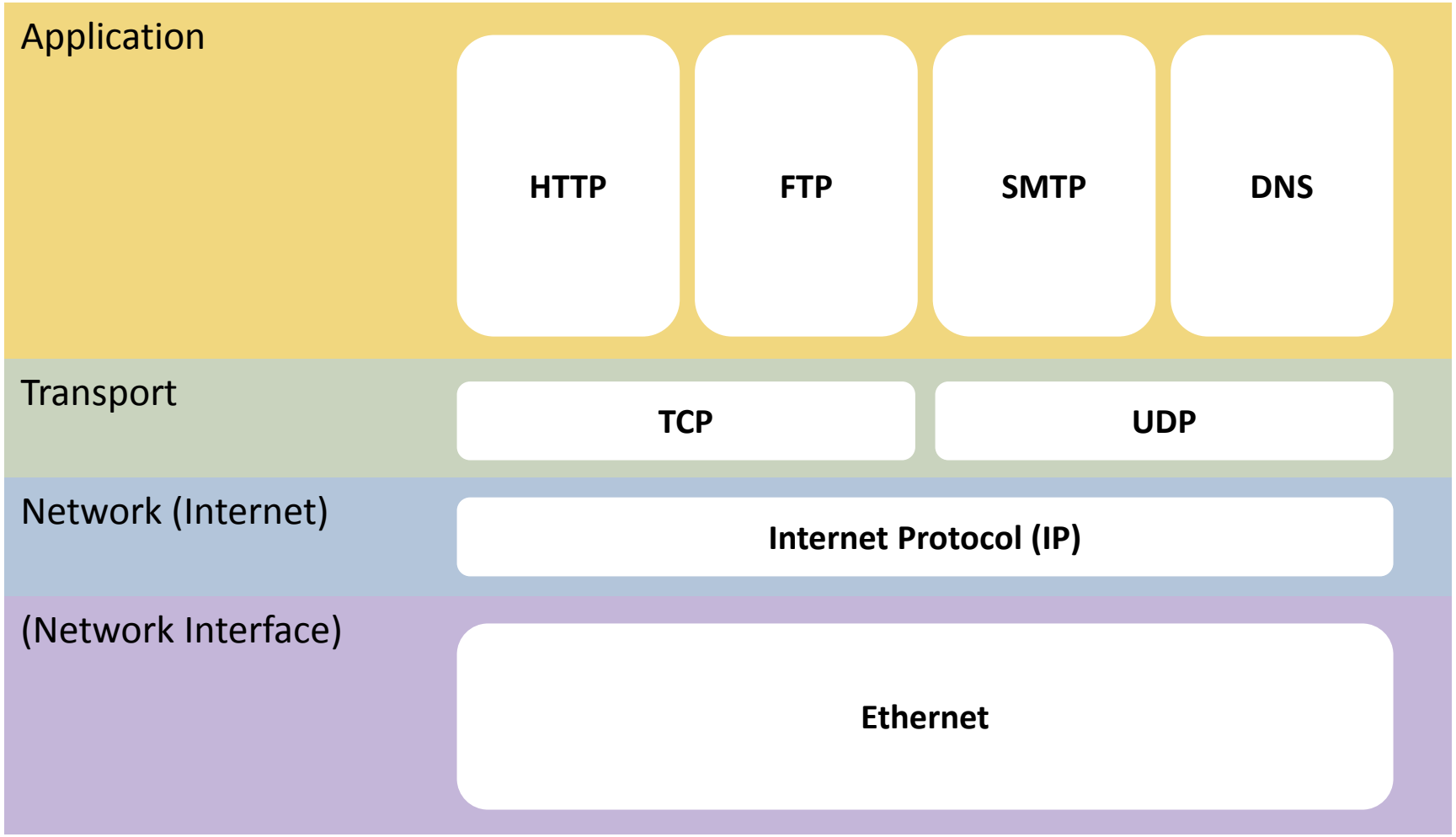
Layered Architecture

The TCP/IP Protocol Suite



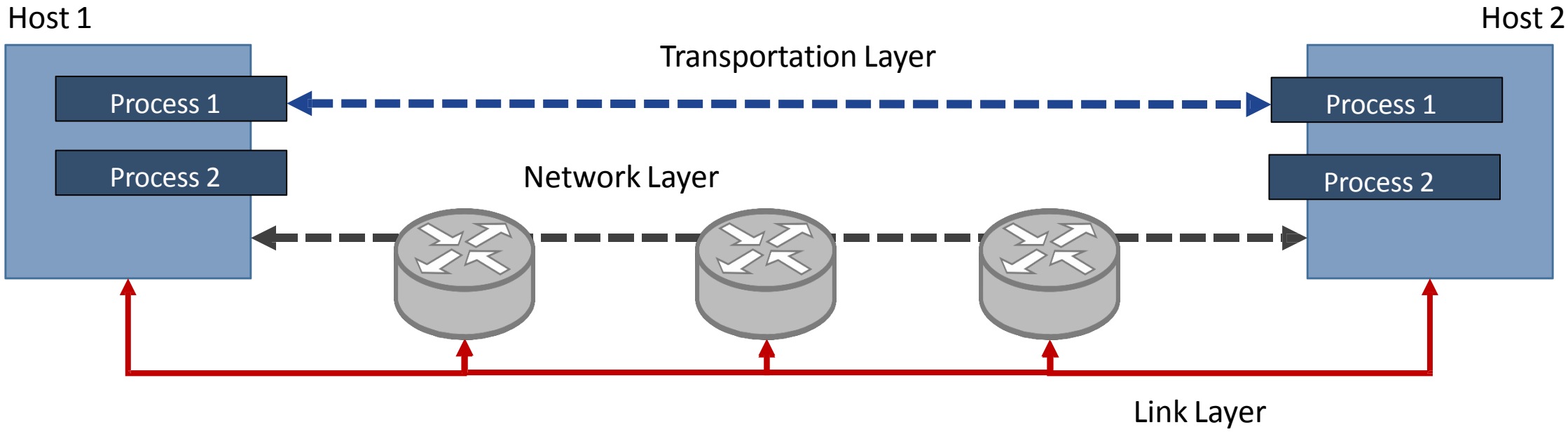
Layered Architecture

The TCP/IP Protocol Suite



TCP/IP

- **Transport Layer**: responsible for process-to-process delivery
- **Network Layer**: host-to-host delivery
- **Link Layer**: node-to-node delivery (hop-by-hop)
- A process is an application program running on a host



TCP/IP

Client/Server Paradigm

- A process, called a client, requests services from a process on another host, called a server
- The following must be defined
  - Local host (Source IP address)
  - Local process (Source port number)
  - Remote host (Destination IP address)
  - Remote process (Destination port number)
- In client-server model, if we regard the client as the local host, then the server is the remote host, and vice versa

TCP/IP

There are **three transport layer protocols** defined in the TCP/IP Protocol Suite

- **User Datagram Protocol (UDP)**
- **Transmission Control Protocol (TCP)**
- Stream Control Transmission Protocol (SCTP)
  - New reliable and message-oriented protocol combines the best features of UDP and TCP
  - For streaming applications (e.g. video streaming)

TCP/IP – The Transport Layer

Connectionless vs. Connection-oriented Protocol

Connectionless

- No pre-established connection between sender and receiver
- Packets are not numbered, and can arrive **out of sequence**
- **No acknowledgement** of having received the packets
- **Unreliable**
- Uses UDP

Connection-Oriented

- A connection is first established between the sender and the receiver
- Has transport layer-level **flow and error control**
- **Reliable**
- Uses **TCP** or **SCTP**

User Datagram Protocol (UDP)

Characteristics of UDP

- UDP is connectionless and unreliable
- Very simple using a minimum of overhead
- Faster and more efficient for many lightweight or time-sensitive purposes
- Suitable for processes sending small messages and does not care much about reliability
- Used for **multicast** and **broadcast**
- Common network applications that use UDP:
  - Domain Name System (DNS)
  - Trivial File Transfer Protocol (TFTP)

User Datagram Protocol (UDP)

User Datagram

- UDP packets, called user datagrams, have a fixed-size 8 bytes header, containing **4 fields**:
  - **Source port number**
    The port number used by the process running on the source host (16-bit)
  - **Destination port number**
    The port number used by the process running on the destination host (16-bit)
  - **Length**
    16-bit field that defines the total length of the user datagram, header plus data (actually duplicated with the length field in IP)
    UDP length = IP length – IP header's length
  - **Checksum**
    A checksum for the user datagram

Transmission Control Protocol (TCP)

TCP: a stream-oriented protocol

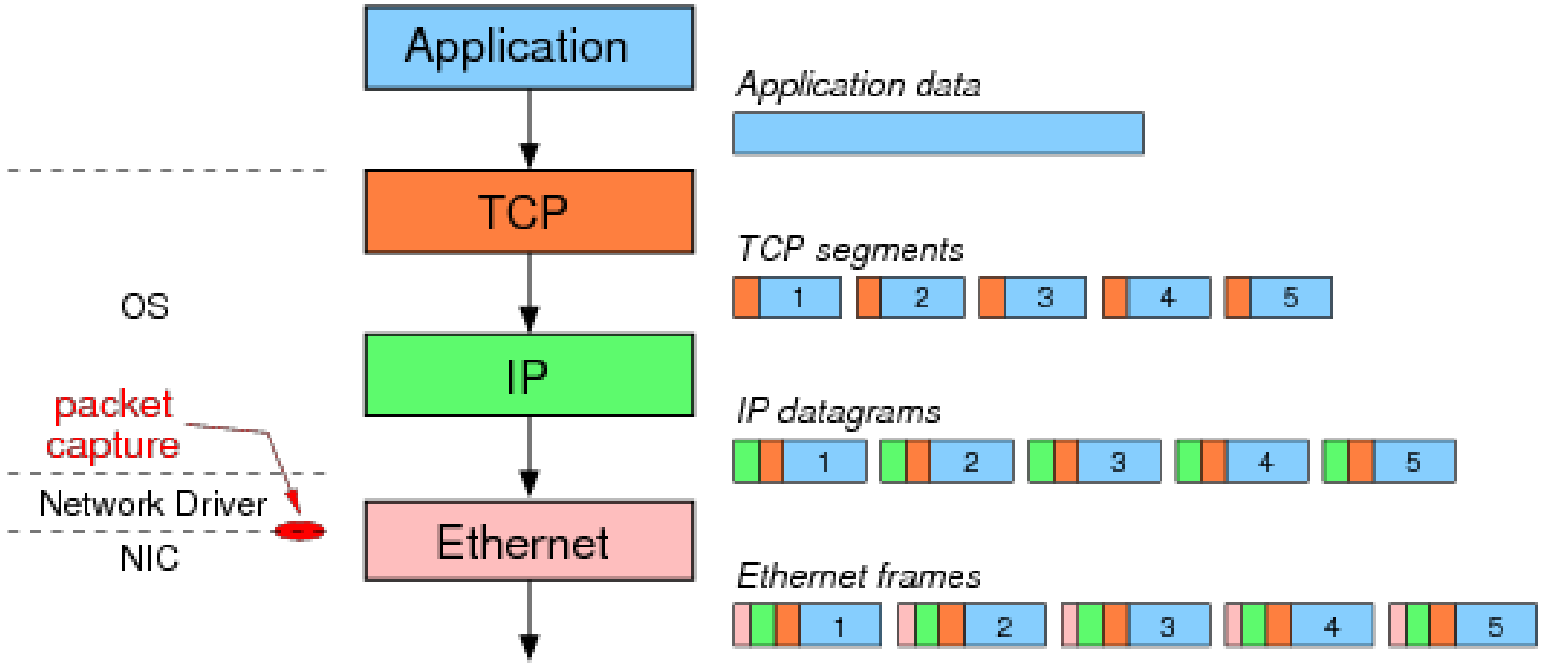
- Instead of independent datagrams, TCP delivers data as **a stream of bytes**
- A large chunk of data is divided into **segments**, these segments are related to one another
- TCP creates an environment in which the two processes seem to be connected by **an imaginary tunnel**

Transmission Control Protocol (TCP)

- The sending and the receiving processes may not write or read data at the same speed
- TCP needs **buffers** for storage, flow control, and error control
- One way to implement the buffer is to use a circular array of 1-byte locations
- TCP buffer size is configurable
  (e.g., buffer size = 2 * bandwidth * delay)
  (Can be up to megabytes)
- UDP does not have buffers and its queue length is relatively smaller

Transmission Control Protocol (TCP)

- TCP delivers data as **segments**
- TCP adds a header to each segment (for control purpose) and delivers the segment to the underlying IP layer for transmission
- The segments are encapsulated in IP datagrams and transmitted (The entire operation is transparent to the processes)



Why Learning All These?

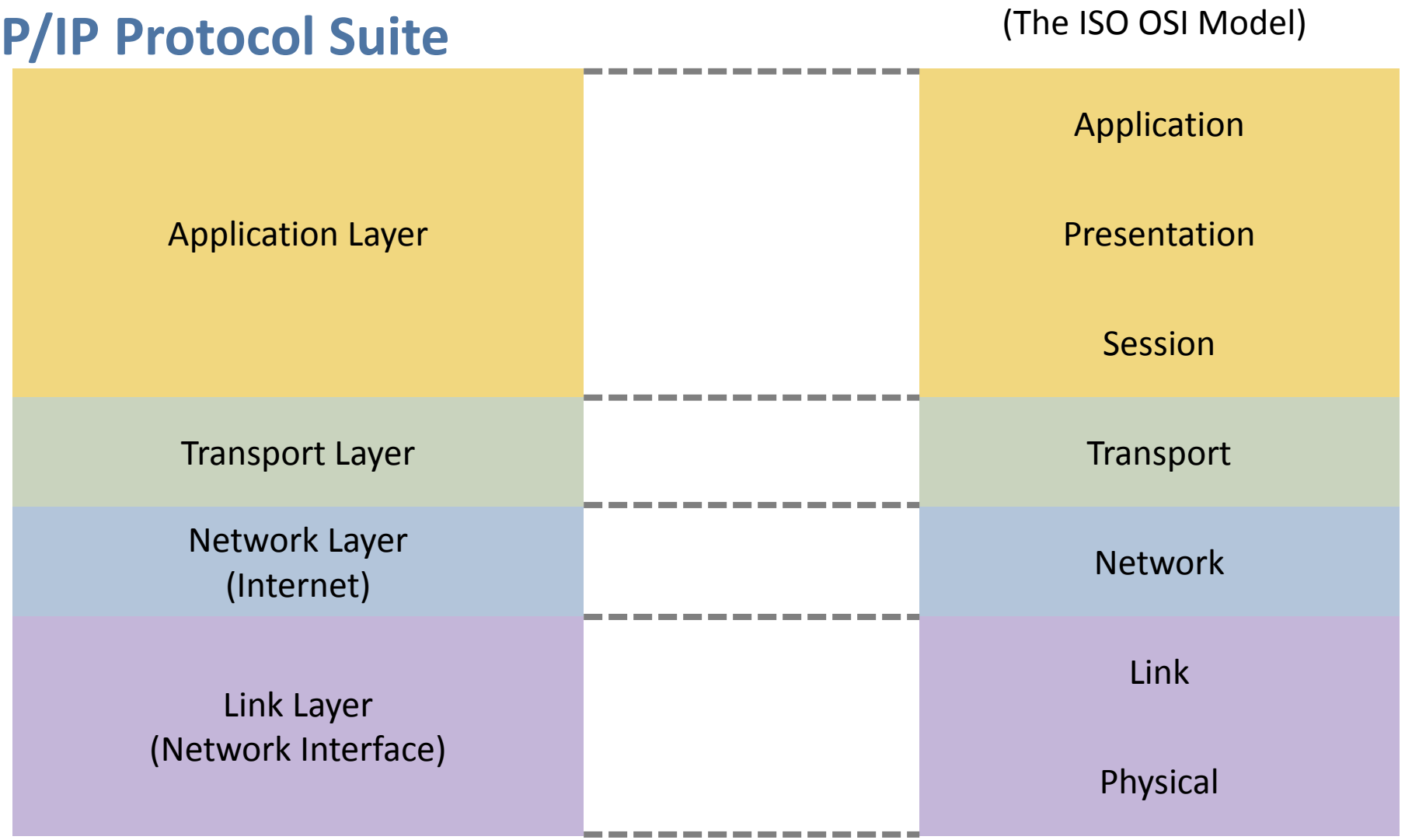
- In developing an app that uses the network, you need to determine how data are communicated between devices and with the server(s)
- Depending on the nature and requirement of your application, you may need to choose from one of the following:

Break
(Attendance)

Socket Programming

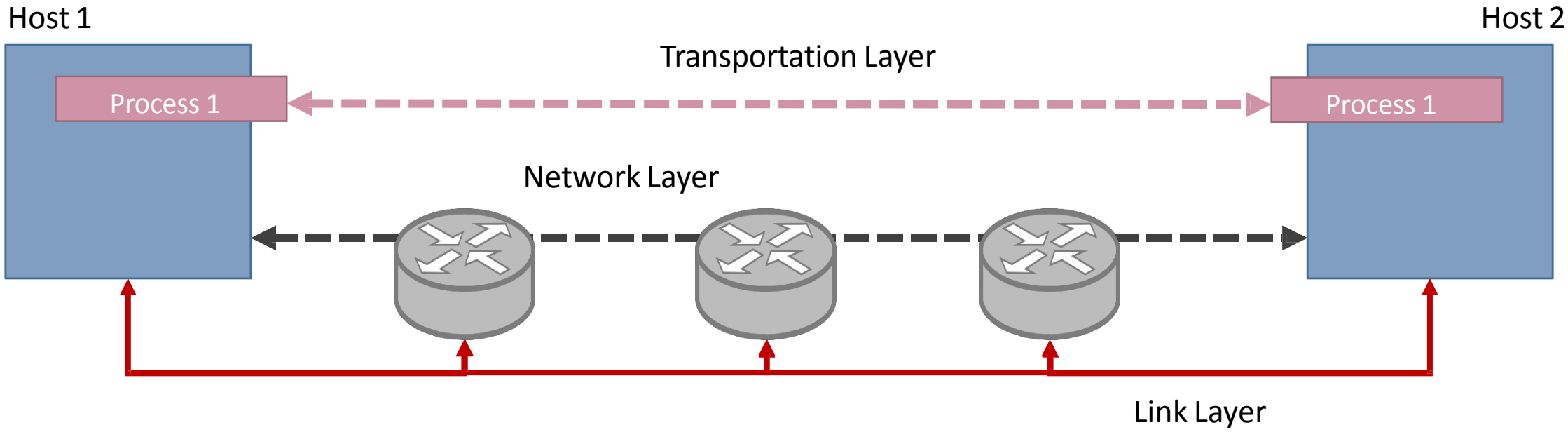
Layered Architecture

The TCP/IP Protocol Suite



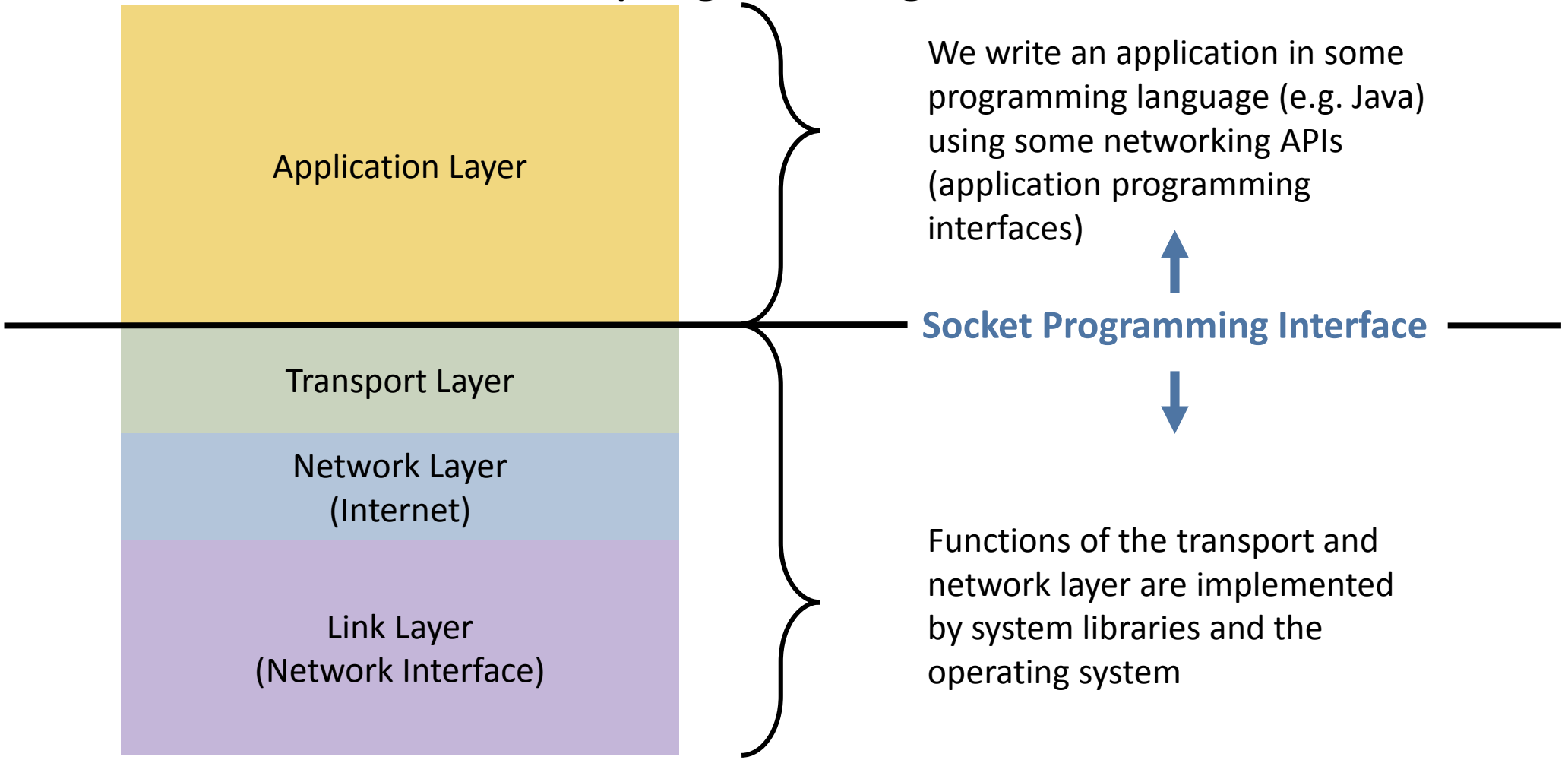
TCP/IP

- **Transport Layer**: responsible for process-to-process delivery
- **Network Layer**: host-to-host delivery
- **Link Layer**: node-to-node delivery (hop-by-hop)
- A process is an application program running on a host



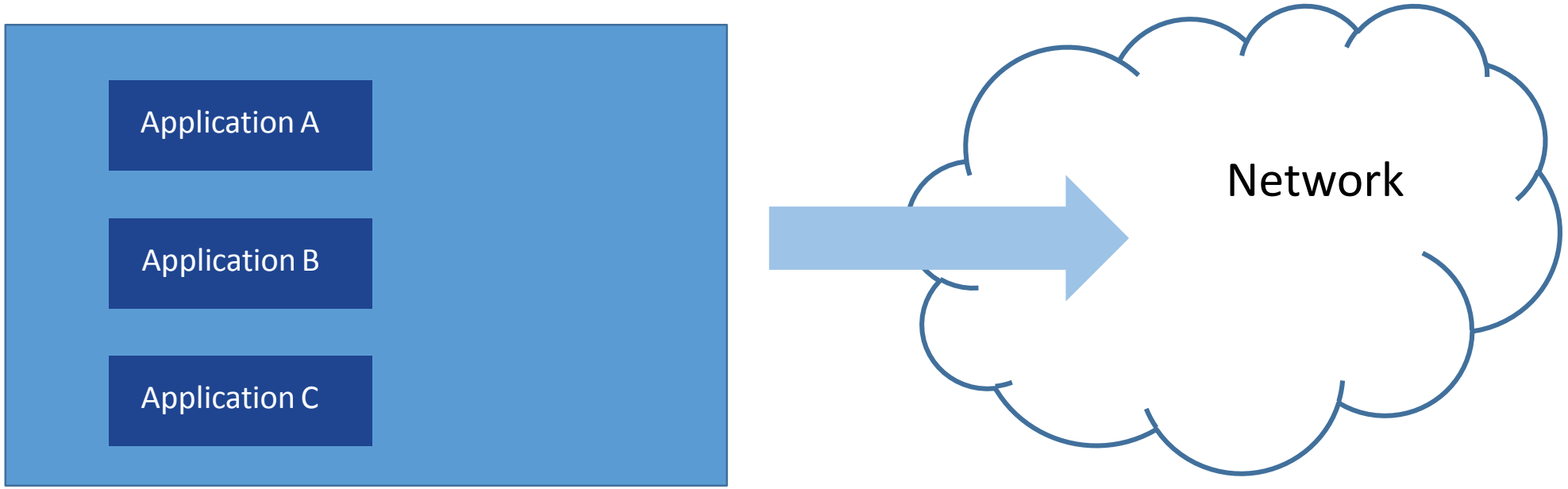
Network Programming in TCP/IP

- What do we do in network programming?



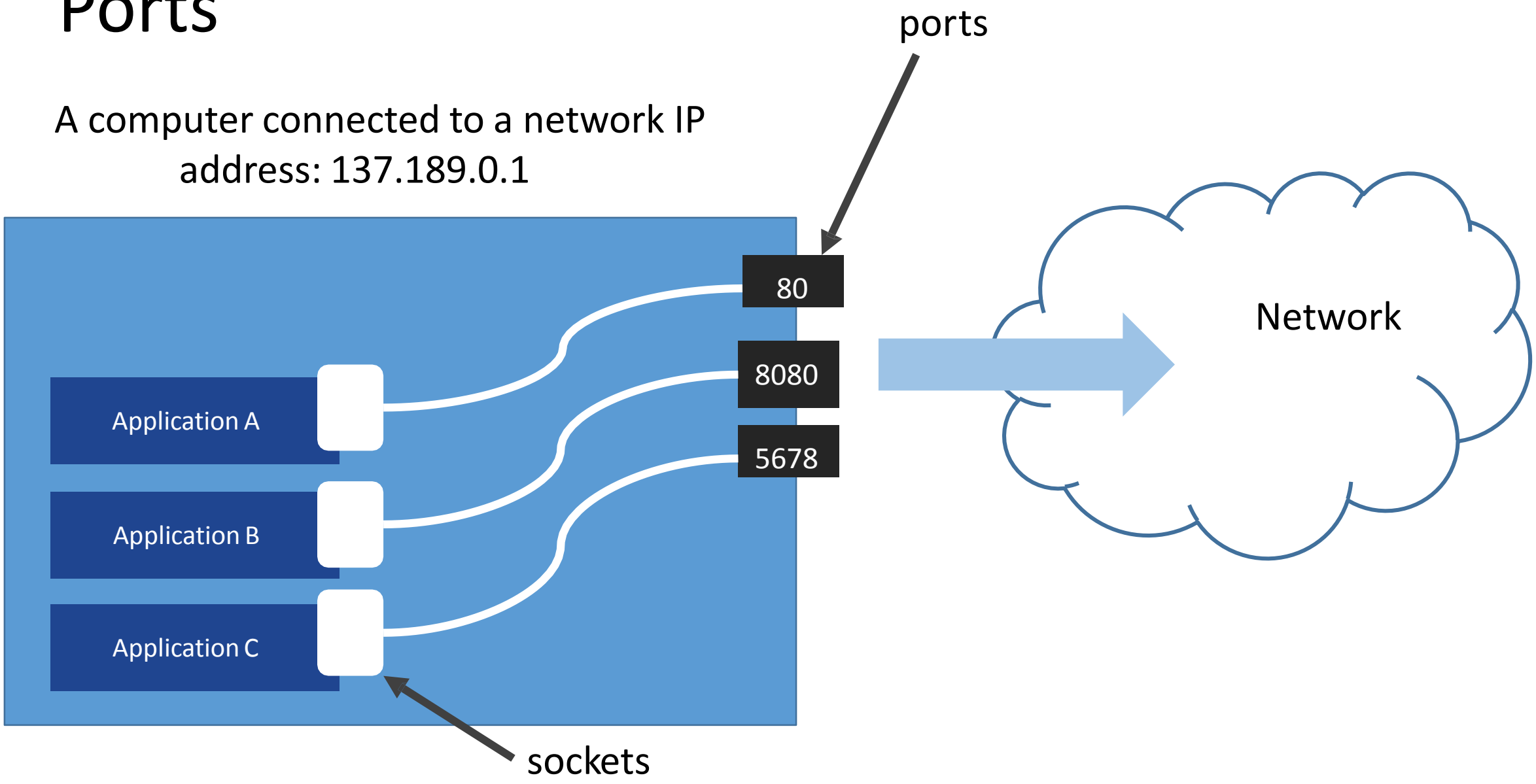
Ports

- Ports are “endpoints of communications” in a computer’s OS
- Ports allow different applications running on the same computer to share a single physical link to the network
- Each application must bind to a unique port (identified by a number) in order to communicate with the network



Ports

A computer connected to a network IP address: 137.189.0.1



Ports

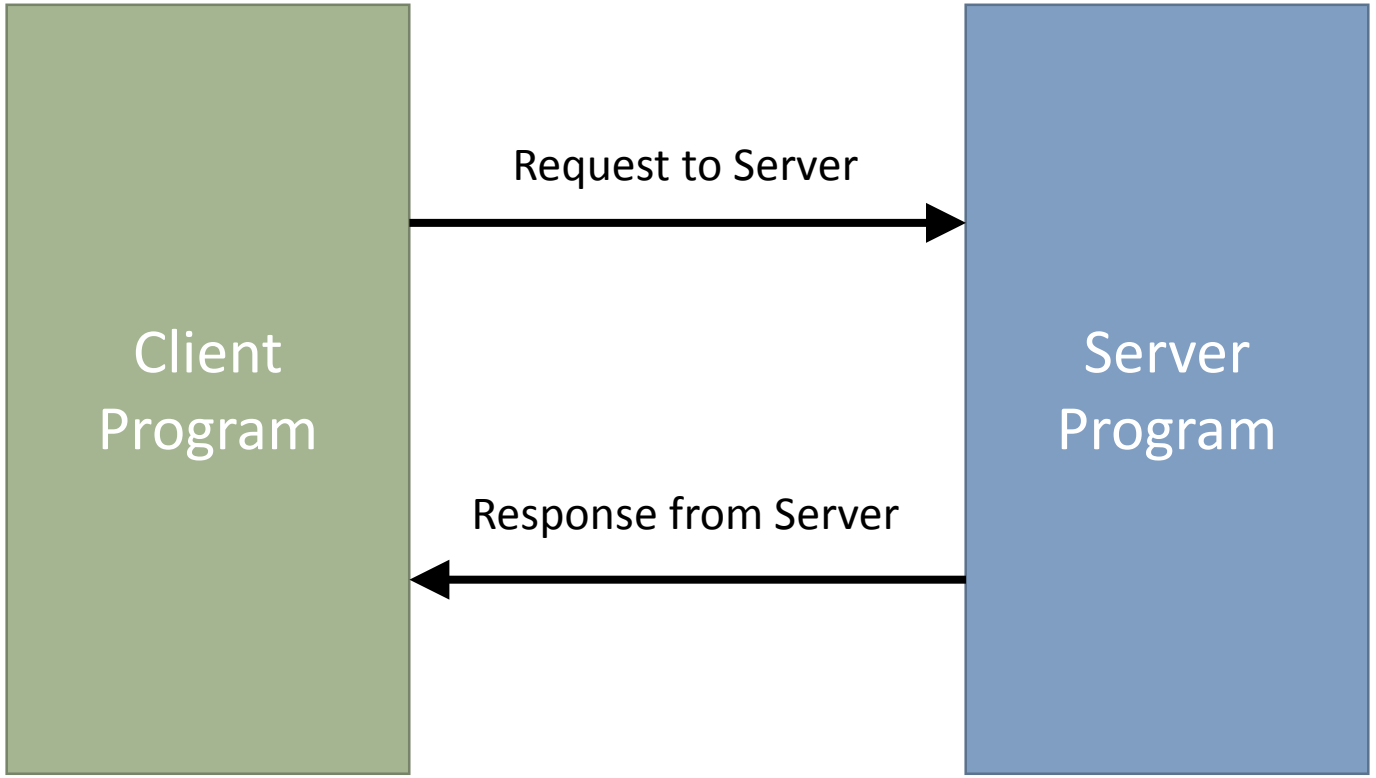
- Port number is a 16-bit unsigned integer (i.e. 0 to 65535)
- Port numbers are regulated and are divided into 3 different ranges (Regulated by the Internet Assigned Numbers Authority (IANA))

Well-known Ports (0–1023)	Registered Ports (1024–49151)	Dynamic/Private Ports (49151–65535)
Registered for well-known applications such as: 21: FTP 80: HTTP 443: HTTPS 465: SMTPS	**Registered for other applications**	**Can be used by private applications**

Reference: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

The Client-Server Model

- Many network applications follow the client-server model
- In such a model, servers are continuously running to wait for the request from clients



1. Server program is executed. It waits for requests from clients
2. The client program is executed, it then sends a request to the server
3. The server receives a request, processes it, retrieves data if necessary, and sends response back to the client
4. The client program receives the response, and presents the result to the user

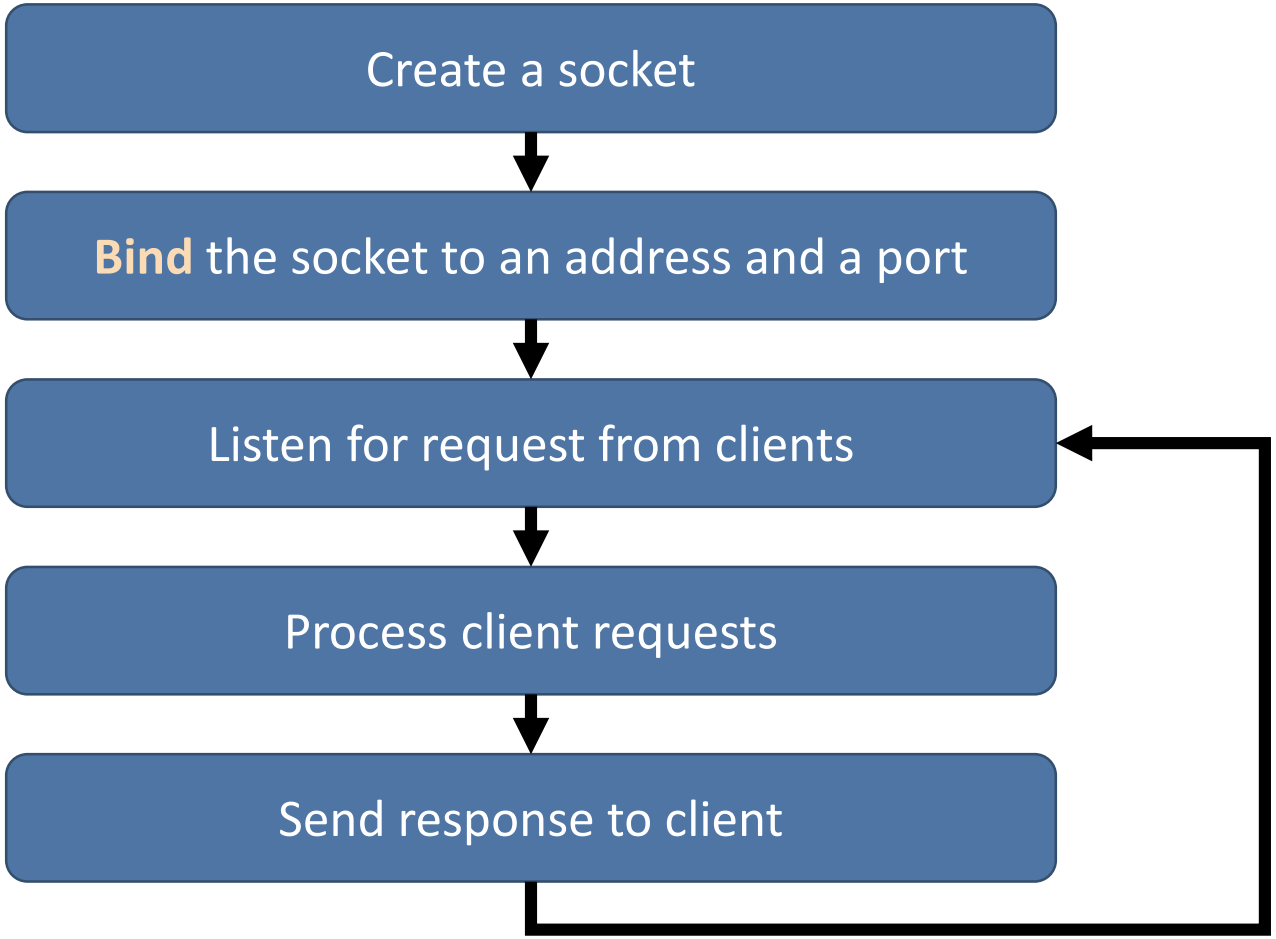
The Client-Server Model

Note:

- "**Client**" and "**server**" here refer to the role of the program at some instance
- One application can be running both a client and a server at the same time
- A mobile app can be a server, if it is serving data to another mobile app

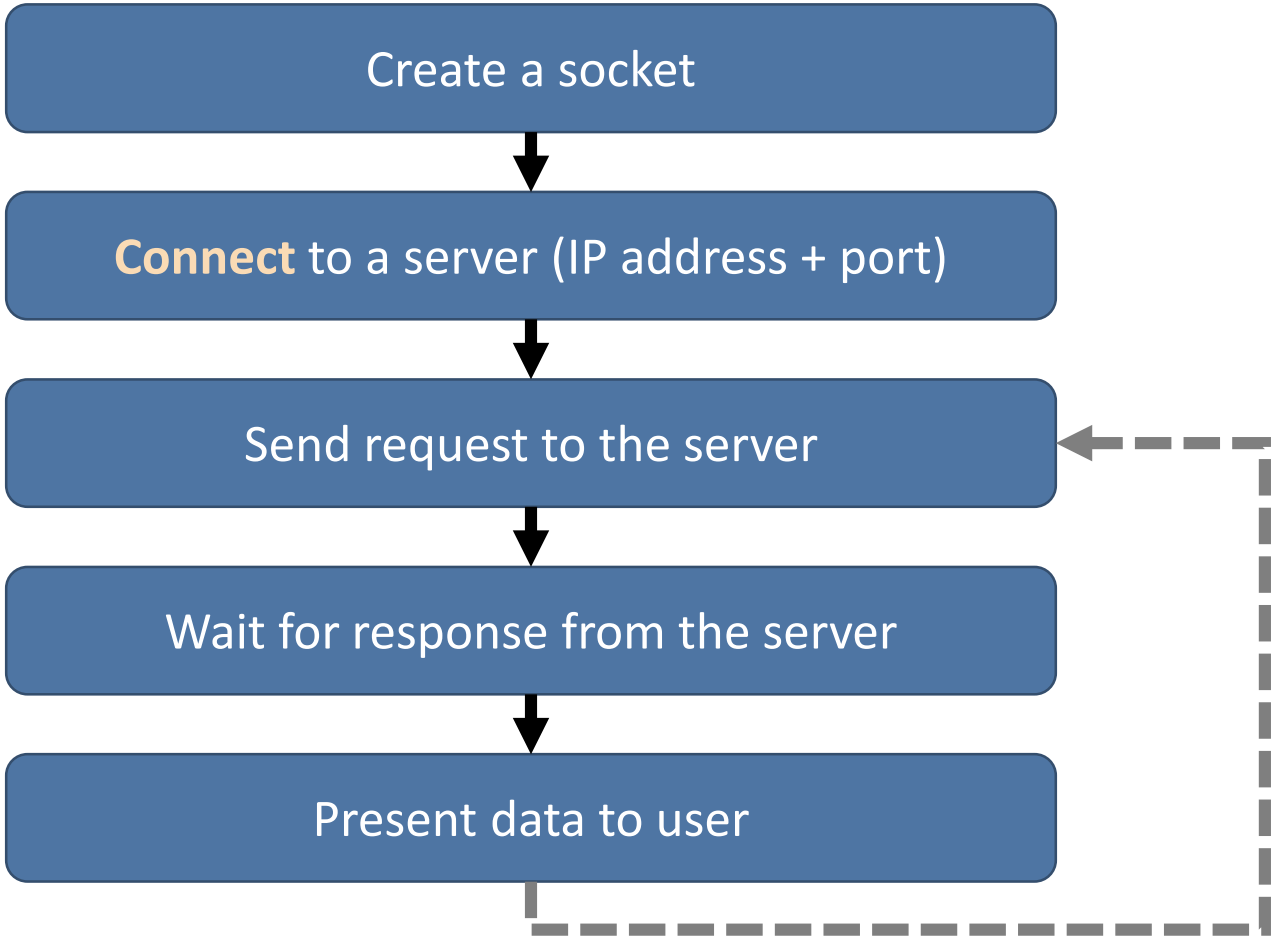
Client-Server Model

- How do we write a **server** program?



Client-Server Model

- How do we write a **client** program?



Network Programming in Java

- You can choose from TCP or UDP when developing your networking application
- Both the server and the client MUST use the same protocol in order to communicate with each other
- Classes that are important in network programming in Java
  - **TCP**: ServerSocket, Socket
  - **UDP**: DatagramSocket, DatagramPacket

Reference: https://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Using TCP in Java

- Let's look at a simple **server** program

Create a new server socket with port number 60001

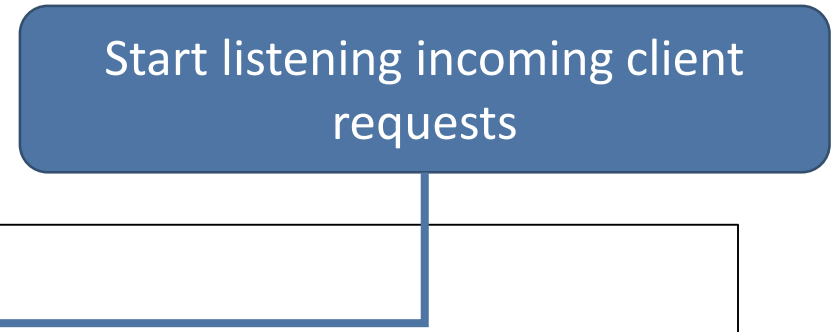
```
ServerSocket server_socket = new ServerSocket(60001);
Socket server = server_socket.accept();

DataInputStream ins = new DataInputStream(server.getInputStream());
String incoming = ins.readUTF();
System.out.println("Received connection from "
        + server.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(server.getOutputStream());
outs.writeUTF("Thanks for connecting to "
        + server.getLocalSocketAddress());
server.close();
```

Using TCP in Java

- Let's look at a simple **server** program



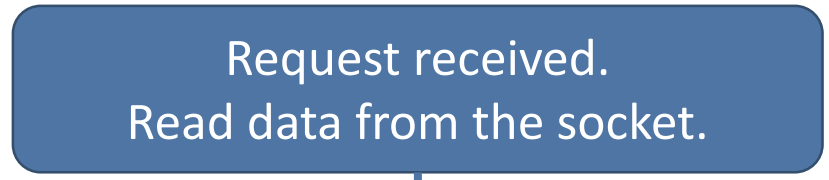

```
ServerSocket server_socket = new ServerSocket(60001);
Socket server = server_socket.accept();

DataInputStream ins = new DataInputStream(server.getInputStream());
String incoming = ins.readUTF();
System.out.println("Received connection from "
        + server.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(server.getOutputStream());
outs.writeUTF("Thanks for connecting to "
        + server.getLocalSocketAddress());
server.close();
```

Using TCP in Java

- Let's look at a simple **server** program




```
ServerSocket server_socket = new ServerSocket(60001);
Socket server = server_socket.accept();

DataInputStream ins = new DataInputStream(server.getInputStream());
String incoming = ins.readUTF();
System.out.println("Received connection from "
        + server.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(server.getOutputStream());
outs.writeUTF("Thanks for connecting to "
        + server.getLocalSocketAddress());
server.close();
```

Using TCP in Java

- Let's look at a simple **server** program

Send response to the client, and then close the server connection

```
ServerSocket server_socket = new ServerSocket(60001);
Socket server = server_socket.accept();

DataInputStream ins = new DataInputStream(server.getInputStream());
String incoming = ins.readUTF();
System.out.println("Received connection from "
        + server.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(server.getOutputStream());
outs.writeUTF("Thanks for connecting to "
        + server.getLocalSocketAddress());
server.close();
```

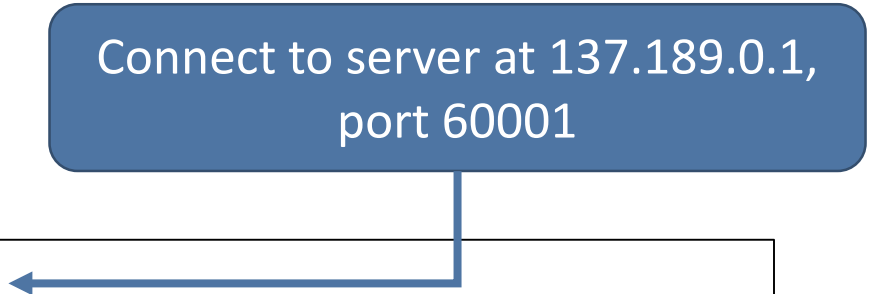
Using TCP in Java

Some notes about this simple **server** program

- It uses the **SocketServer** class, which is a class for creating a server that uses TCP
- **Socket.accept()** is a "**blocking**" function
- Data is received and sent through **data streams** (instead of packets)
- Client and server address can be extracted by using **getRemoteSocketAddress** and **getLocalSocketAddress**
- It can only serve **one client**
  (once the request is processed, the server is closed)

Using TCP in Java

- Let's look at a simple **client** program



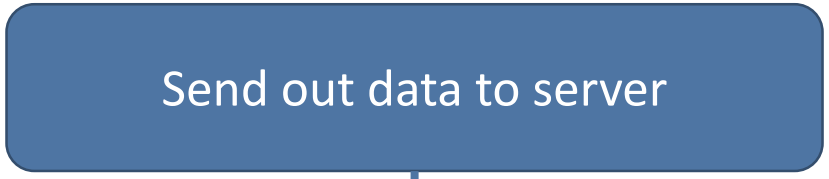

```
Socket client = new Socket("137.189.0.1", 60001);
System.out.println("Just connected to "
       + client.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(client.getOutputStream());
outs.writeUTF("Hello from "
       + client.getLocalSocketAddress());

DataInputStream ins = new DataInputStream(client.getInputStream());
System.out.println("Server says " + ins.readUTF());
client.close();
```

Using TCP in Java

- Let's look at a simple **client** program




```
Socket client = new Socket("137.189.0.1", 60001);
System.out.println("Just connected to "
       + client.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(client.getOutputStream());
outs.writeUTF("Hello from "
       + client.getLocalSocketAddress());

DataInputStream ins = new DataInputStream(client.getInputStream());
System.out.println("Server says " + ins.readUTF());
client.close();
```

Using TCP in Java

- Let's look at a simple **client** program

```
Socket client = new Socket("137.189.0.1", 60001);
System.out.println("Just connected to "
       + client.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(client.getOutputStream());
outs.writeUTF("Hello from "
       + client.getLocalSocketAddress());

DataInputStream ins = new DataInputStream(client.getInputStream());
System.out.println("Server says " + ins.readUTF());
client.close();
```

Receive data from server, present data to user, and close the connection

Using TCP in Java

Some notes about this simple **client** program

- It uses the **Socket** class, which is a class for creating a socket that uses TCP
- **Socket.getInputStream()** is a "**blocking**" function
- Data is received and sent through **data streams** (instead of packets)

Using UDP in Java

You can also write your network application using **UDP**

- In UDP, data is sent in the form of **packets**
- You will have to pack your data in a **DatagramPacket** before sending it to the server
- There is a possibility that the packet is “dropped” when being transmitted, and the client does not receive any response

Reference: https://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html

Using UDP in Java

- A simple “echo” **server** implemented in UDP

```
DatagramSocket socket = new DatagramSocket(60001);
DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);

socket.receive(packet);
System.out.println("Receive connection from "
        + packet.getAddress().getHostAddress()
        + ":" + packet.getPort());

socket.send(packet);
```

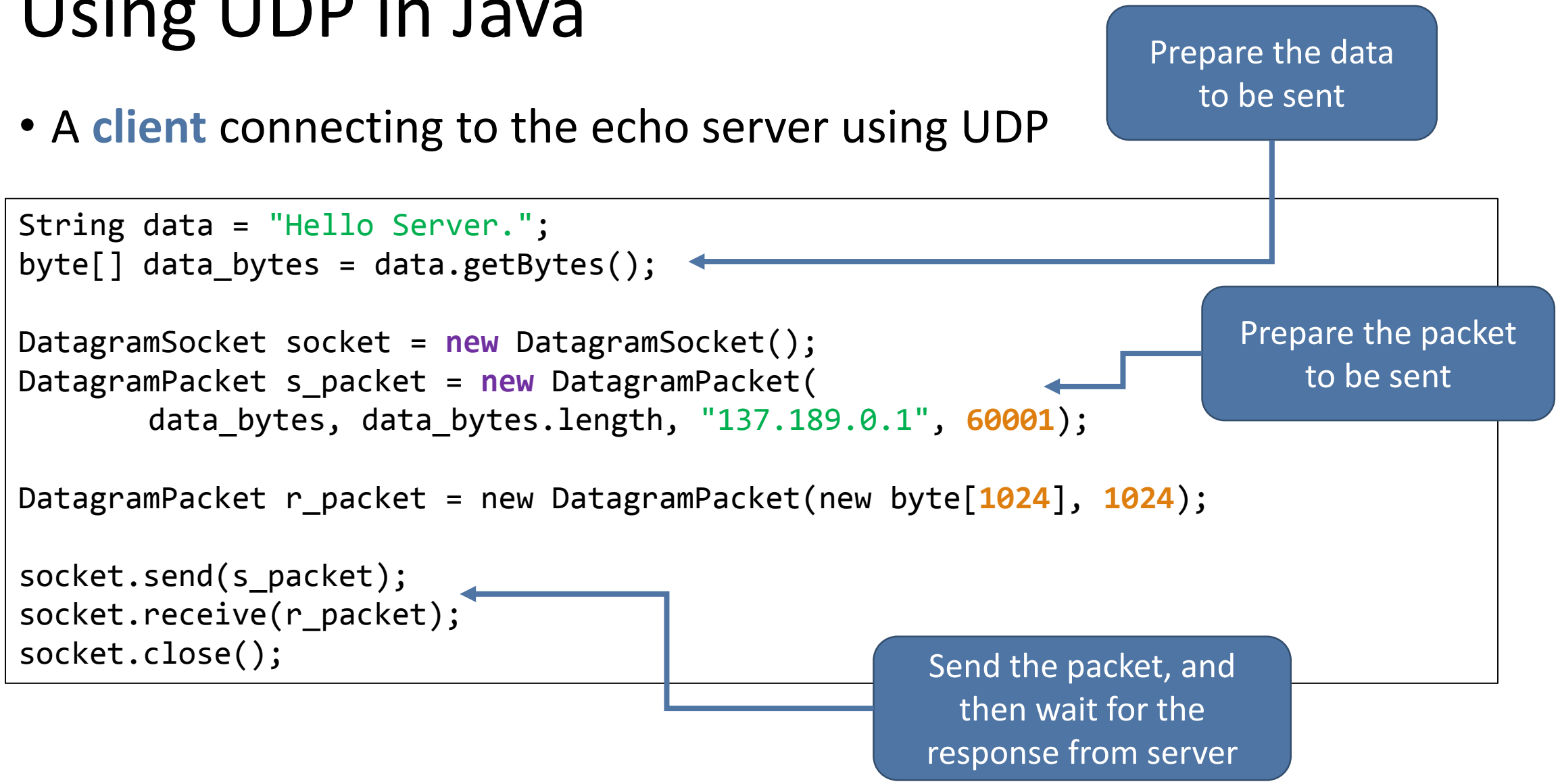
Create a socket that binds to a port

Create an empty packet for receiving data

Receive the packet from the socket

Using UDP in Java

- A **client** connecting to the echo server using UDP




```
String data = "Hello Server.";
byte[] data_bytes = data.getBytes();

DatagramSocket socket = new DatagramSocket();
DatagramPacket s_packet = new DatagramPacket(
        data_bytes, data_bytes.length, "137.189.0.1", 60001);

DatagramPacket r_packet = new DatagramPacket(new byte[1024], 1024);

socket.send(s_packet);
socket.receive(r_packet);
socket.close();
```

Using UDP in Java

Notes

- Packet size should not be too large (up to 65508 bytes)
- If you send multiple packets, they may arrive out of order
- On the client size, you may need to handle errors such as:
  - Timeout (the server does not respond for some time)
  - Received packet from another server
  - Out-of-order arrival of the packets

Multi-threading

Supporting Multiple Clients

- Consider our simple TCP server program, what is the problem?

```
ServerSocket server_socket = new ServerSocket(60001);
Socket server = server_socket.accept();

DataInputStream ins = new DataInputStream(server.getInputStream());
String incoming = ins.readUTF();
System.out.println("Received connection from "
        + server.getRemoteSocketAddress());

DataOutputStream outs = new DataOutputStream(server.getOutputStream());
outs.writeUTF("Thanks for connecting to "
        + server.getLocalSocketAddress());
server.close();
```

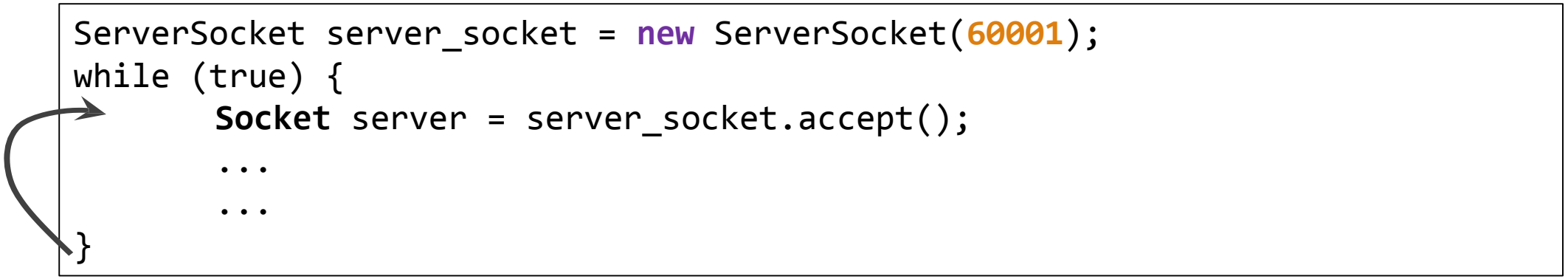
Supporting Multiple Clients

- Commands and operations are executed in a program in sequentially
- Most of the commands and operations you write in your program is "**blocking**" or "**synchronous**"
  - Meaning that one command must be finished before another command can be executed
- A problem if the operations to serve a client is time consuming, e.g.:
  - Retrieving large amount of data from database
  - Read and write files (I/O operations)
  - Heavy computation (e.g. ranking and sorting data)

Supporting Multiple Clients

Allow the server to run continuously to serve different clients

- Once the server have finished serving one client, it will go back to listen for requests again
- **Problem?**



```
ServerSocket server_socket = new ServerSocket(60001);
while (true) {
        Socket server = server_socket.accept();
        ...
        ...
}
```

Supporting Multiple Clients

- What if it takes a long time to serve a client?

```
ServerSocket server_socket = new ServerSocket(60001);
while (true) {
        Socket server = server_socket.accept();
        ...
        ...
        ...
        ...
        ...
}
```

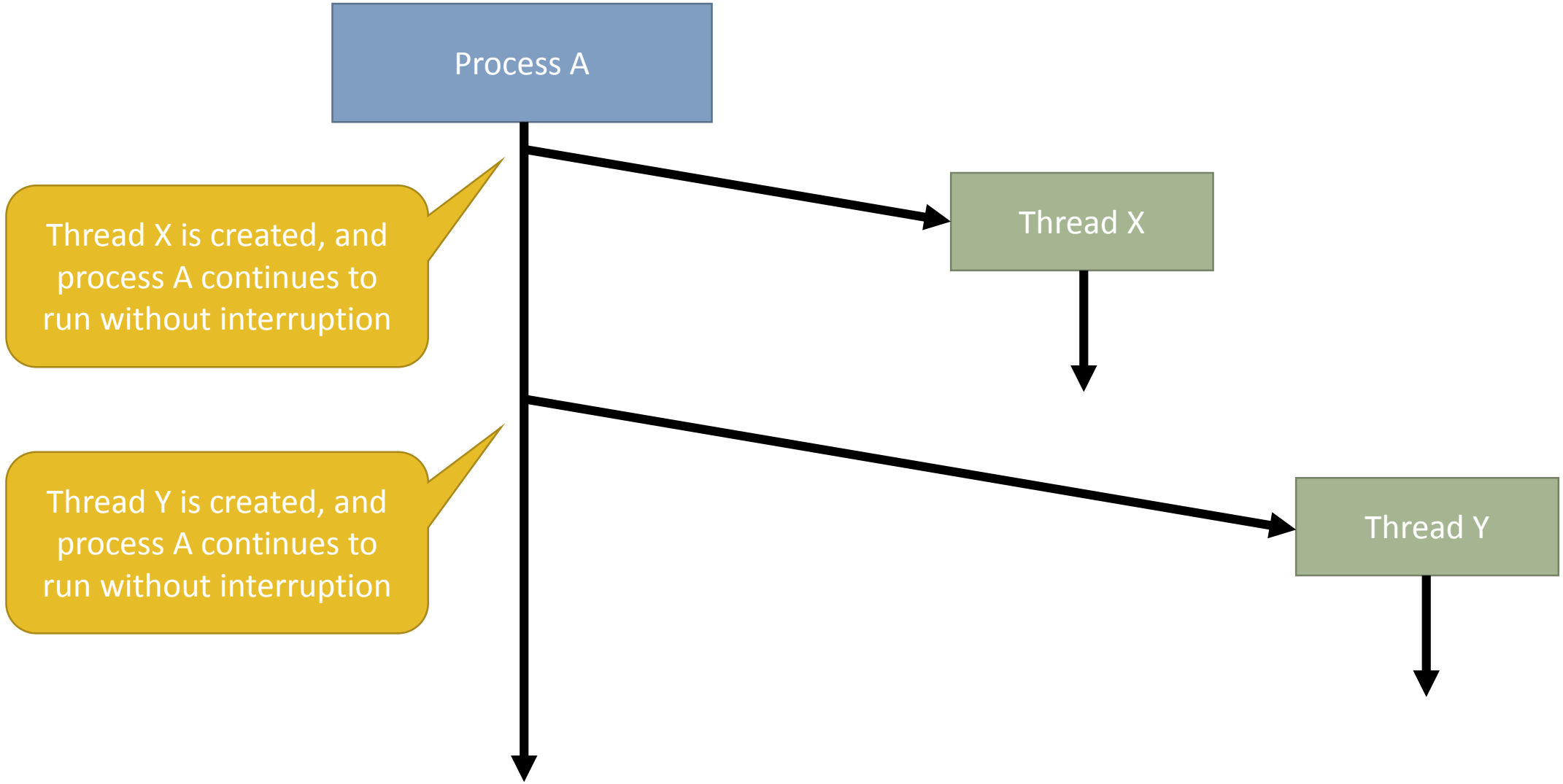
While running this part of the code, the server will NOT be able to accept any connection.

Threads

Multi-threading and Concurrent Programming

- A program or an app can be referred to as a **process**
- **Threads** are “light-weighted processes” that carry out different operations within a process at the “same” time (run in parallel)
- The CPU divides processing time among different processes and among different threads within a process
- Every process has at least one thread (**the main thread**)

Threads


Process A
Thread X is created, and process A continues to run without interruption
Thread X
Thread Y is created, and process A continues to run without interruption
Thread Y

Threads in Java

There are two different ways to write programs that use multi-threading in Java

1. Implement the **Runnable** interface
2. Subclass the **Thread** class and implement the **run()** method

The Runnable Interface

- The **Runnable** interface defines a single method **run()**. You put the operations that need to be performed in the thread inside this function.
- Example:

```
public class HelloRunnable implements Runnable {

        public void run() {
                System.out.println("Hello from a thread!");
        }

        public static void main(String args[]) {
                (new Thread(new HelloRunnable())).start();
        }
}
```

The operations to be performed in the new thread

Start the new thread in this way

The Thread Class

- The **Thread** class implements the Runnable interface, but you have to implement the **run()** function by yourself
- Example:

```
public class HelloThread extends Thread {

        public void run() {
                System.out.println("Hello from a thread!");
        }

        public static void main(String args[]) {
                (new HelloThread()).start();
        }
}
```

Extends the thread class

Define the run() function

Start the thread

Managing Threads

Having too many threads running in the same process may affect the performance of the whole application

- Threads may compete to use computing resources (e.g. memory, CPU time)
- If you are doing database operations, too many threads accessing the DB can be a problem

Managing Threads

- To limit the number of threads that can exist at the same time, we can create a **thread pool** using the **ExecutorService** class
- Example:

```
public static void main(String[] args) {
        ExecutorService pool = Executors.newFixedThreadPool(2);
        Thread t1 = new MyThread();
        Thread t2 = new MyThread();
        Thread t3 = new MyThread();
        pool.execute(t1);
        pool.execute(t2);
        pool.execute(t3);
        pool.shutdown();
}
```

Managing Threads

Using a thread pool:

- **newFixedThreadPool(n)** creates a thread pool of size **n**
- At any time, at most **n** threads can be active in the process
- If additional tasks are added to the pool, they will have to wait in the queue until a thread in the pool is available
- Other ways of creating a pool including
  - CachedThreadPool()
  - ScheduledThreadPool()

Supporting Multiple Clients

Improving our simple TCP server

- Once a connection is accepted from the client (when socket.accept returns), we create a new thread to handle the request
- The server can listen to new connections again immediately

```
ServerSocket server_socket = new ServerSocket(60001);
while (true) {
        Socket server = server_socket.accept();
        new MyThread(server).start();
}
```

Problem?

Supporting Multiple Clients

Improving our simple TCP server

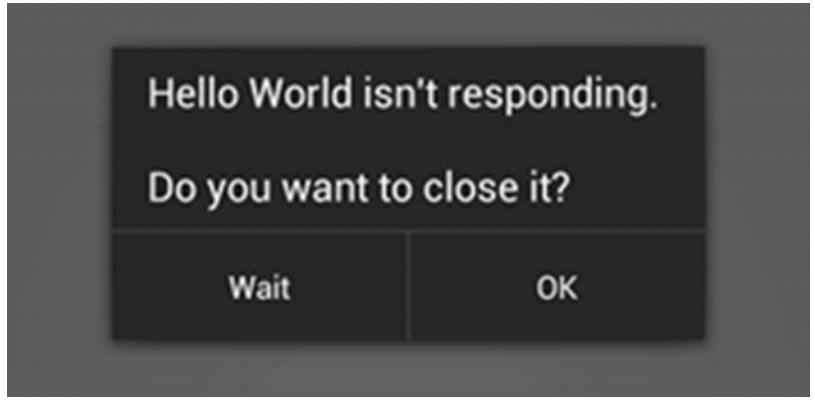
- We may want to restrict the number of clients we are serving at the same time
- Use ExecutorService to create a thread pool:

```
ServerSocket server_socket = new ServerSocket(60001);
ExecutorService pool = Executors.newFixedThreadPool(2);
while (true) {
        Socket server = server_socket.accept();
        MyThread t = new MyThread(server);
        pool.execute(t);
}
```

Multi-threading in Android

Processes and Threads in Android

- In Android, all components of an application runs in the same process and thread (the **main** thread)
- The thread of execution for the application takes care of drawing the layout, taking user input, so it is also called the **UI thread** too
- **NO long operations should be performing on the UI thread**
  - If the UI thread is blocked for a few seconds, an “application not responding” (ANR) dialog will appear
- Also, **other threads should not manipulate the UI**



Processes and Threads in Android

- The multi-threading methods introduced for Java can be used in Android as well
- However, **two rules** must be followed:
  1. **Do not block the UI thread at any time**
  2. **Do not access UI components from threads other than the main thread**

Processes and Threads in Android

- Consider the following piece of code:

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            image_view.setImageBitmap(b);
        }
    }).start();
}
```

What is the problem?

Processes and Threads in Android

To make it “thread-safe”, we can use one of the following functions

- **Activity.runOnUiThread**(Runnable)
- **View.post**(Runnable)
- **View.postDelayed**(Runnable, long)

These functions make sure that manipulation of UI components are done on the UI thread by posting the task to the queue of the UI thread

Processes and Threads in Android

- For example:

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
            final Bitmap b = loadImageFromNetwork("http://example.com/image.png");
            mImageView.post(new Runnable() {
                public void run() {
                    mImageView.setImageBitmap(b);
                }
            });
        }
    }).start();
}
```

This part will be executed on the UI thread

Processes and Threads in Android

- However, as your application becomes more complex, it might be difficult to maintain codes like this
- In the next lecture, we will discuss about how to use some mechanisms provided by Android to do multi-threading

Learning Resources

- **Java Network Programming**
  - Lesson: All About Sockets
    https://docs.oracle.com/javase/tutorial/networking/sockets/index.html
  - Pick a Java Network Programming book from the library
  -
- **Android Network Programming**
  - Process and Threads
    https://developer.android.com/guide/components/processes-and-threads.html
  - Perform Network Operations in Android
    https://developer.android.com/training/basics/network-ops/index.html

End of Lecture 3