Socket Programming: Part 1

Spring 22 TAs: Lakiotakis Manos, Plevridi Eleftheria,
manoslak@csd.uoc.gr, plevridi@csd.uoc.gr
Computer Science Department, University of Crete

Goal of this lab

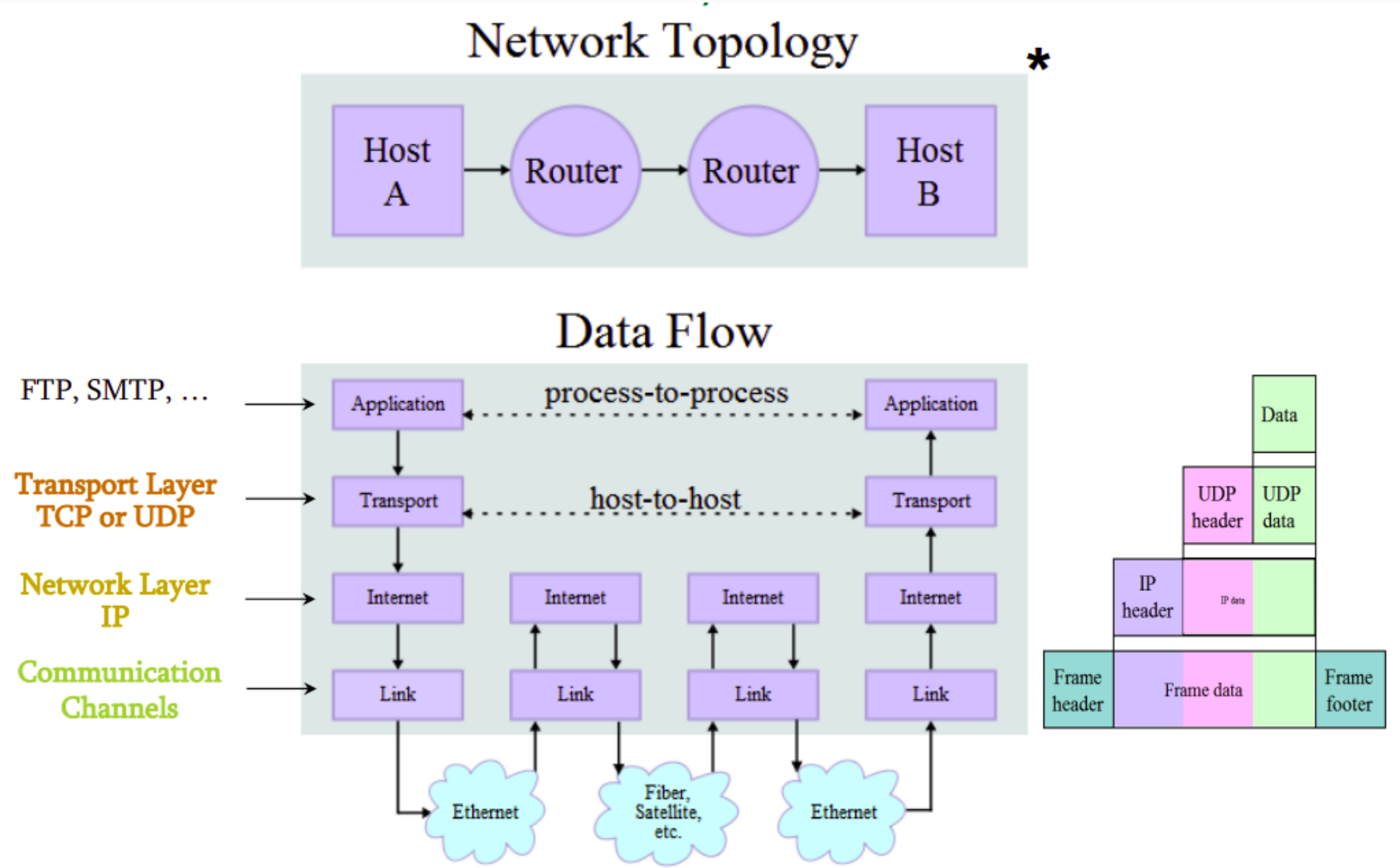
- Learn to create programs that communicate over a network
- Create TCP and UDP sockets using the POSIX Socket API
- Support of multiple connections within a program
- Change the default behavior of sockets

Introduction

Protocol Families - TCP/IP

- TCP/IP provides end-to-end connectivity specifying how data should be
  - formatted
  - addressed
  - transmitted
  - routed, and
  - received at the destination
- can be used in the internet and in stand-alone private networks
- it is organized into **layers**

TCP/IP


Network Topology
*
Host A
Router
Router
Host B
Data Flow
FTP, SMTP, …
Transport Layer
TCP or UDP
Network Layer
IP
Communication
Channels
Application
process-to-process
Application
Transport
host-to-host
Transport
Internet
Internet
Internet
Internet
Link
Link
Link
Link
Ethernet
Fiber,
Satellite,
etc.
Ethernet
Data
UDP
header
UDP
data
IP
header
IP data
Frame
header
Frame data
Frame
footer

Internet Protocol (IP)

- provides a datagram service (packets are handled and delivered independently)
- **best-effort** protocol (may loose, reorder or duplicate packets)
- each packet must contain an **IP address** of its destination

TCP vs UDP

- Both use **port numbers**
- 16-bit unsigned integer, thus ranging from 0 to 65535
- provide E2E transport

UDP: UserDatagram Protocol

- no acknowledgments , no retransmissions
- out of order, duplicates are possible
- connection-less

TCP: Transmission Control Protocol

- reliable byte-stream channel (in order, all arrive, no duplicates)
- flow control
- connection-oriented
- bidirectional

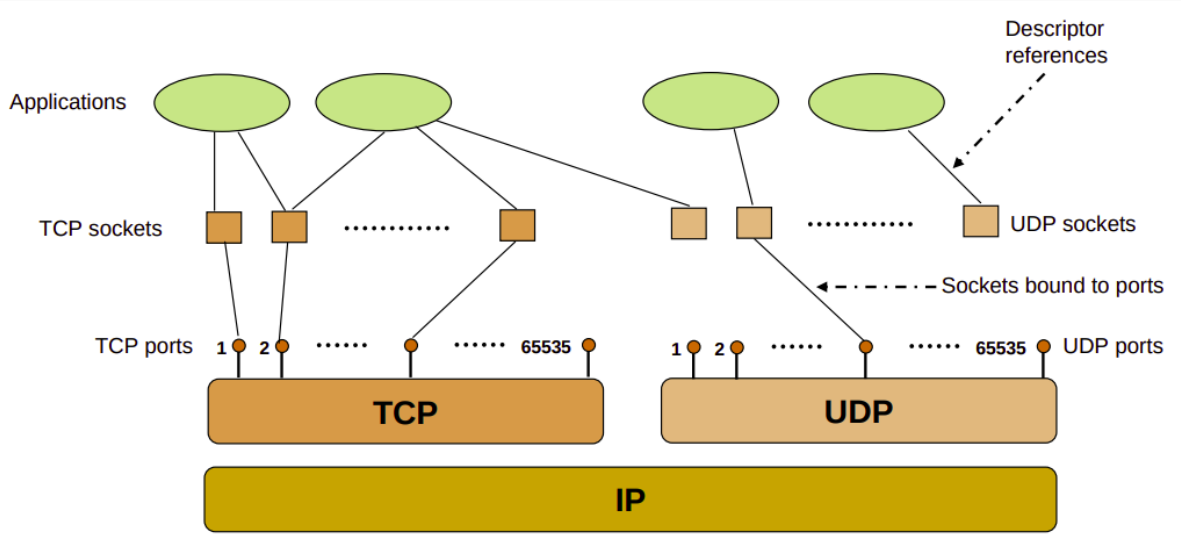
Sockets

Uniquely identified by:

- an internet address
- an end-to-end protocol (e.g. TCP or UDP)
- a port number

Two types of (TCP/IP) sockets

- **Stream Sockets** provide reliable byte-stream service
- **Datagram sockets** provide best-effort datagram service



The POSIX Socket API

The POSIX Socket API

What is POSIX?
Portable **O**perating **S**ystem **I**nterface, is a family of standards specified by the IEEE for maintaining compatibility between operating systems.

- There are several Sockets implementations (e.g Berkeley, BSD)
- POSIX Socket API, provides a cross-platform and reliable way for network and inter-process communication

Creating a Socket

Prototype

- **socket()** creates a socket of a certain domain, type and protocol specified by the parameters
- Possible domains:
  - **AF_INET** for IPv4 internet protocols
  - **AF_INET6** for IPv6 internet protocols

Creating a Socket

Prototype

- Possible types:
  - **SOCK_STREAM** provides reliable two way connection-oriented byte streams (TCP)
  - **SOCK_DGRAM** provides connection-less, unreliable messages of fixed size (UDP)
- protocol depends on the domain and type parameters. In most cases 0 can be passed

Creating a Socket

SOCK_STREAM

Sockets of this type are full-dublex data streams that do not rely on a known data length. Before sending or receiving the socket must be in a connected state. To send and receive data, **send()** and **recv()** system calls may be used. By default, socket of this type are blocking, meaning that a call of **recv()** may block until data arrive from the other side. At the end, **close()** should be used to properly indicate the end of the communication session.

SOCK_DGRAM

This kind of sockets allowing to send messages of a specific size without the guarantee that they will be received from the other side. To send and receive messages **sendto()** and **recvfrom()** calls may be used.

TCP Sockets

TCP: Creating the socket

- Lets try to create our first TCP socket!

- Always check for errors! Using **perror()** printing a useful and meaningful message is very easy!
- Opening a TCP socket is exactly the same for both server and client side

Bind a Socket

Prototype

- **bind()** assigns an open socket to a specific network interface and port
- **bind()** is very common in TCP servers because they should waiting for client connections at specific ports

TCP: Bind the socket

- Always reset the struct **sockaddr_in** before use
- Addresses and ports must be assigned in **Network Byte Order**
- **INADDR_ANY** tells the OS to bind the socket at all the available network interfaces

Listening for incoming connections

Prototype

- After binding to a specific port a TCP server can listen at this port for incoming connections
- backlog parameter specifies the maximum possible outstanding connections
- Clients can connect using the **connect()** call

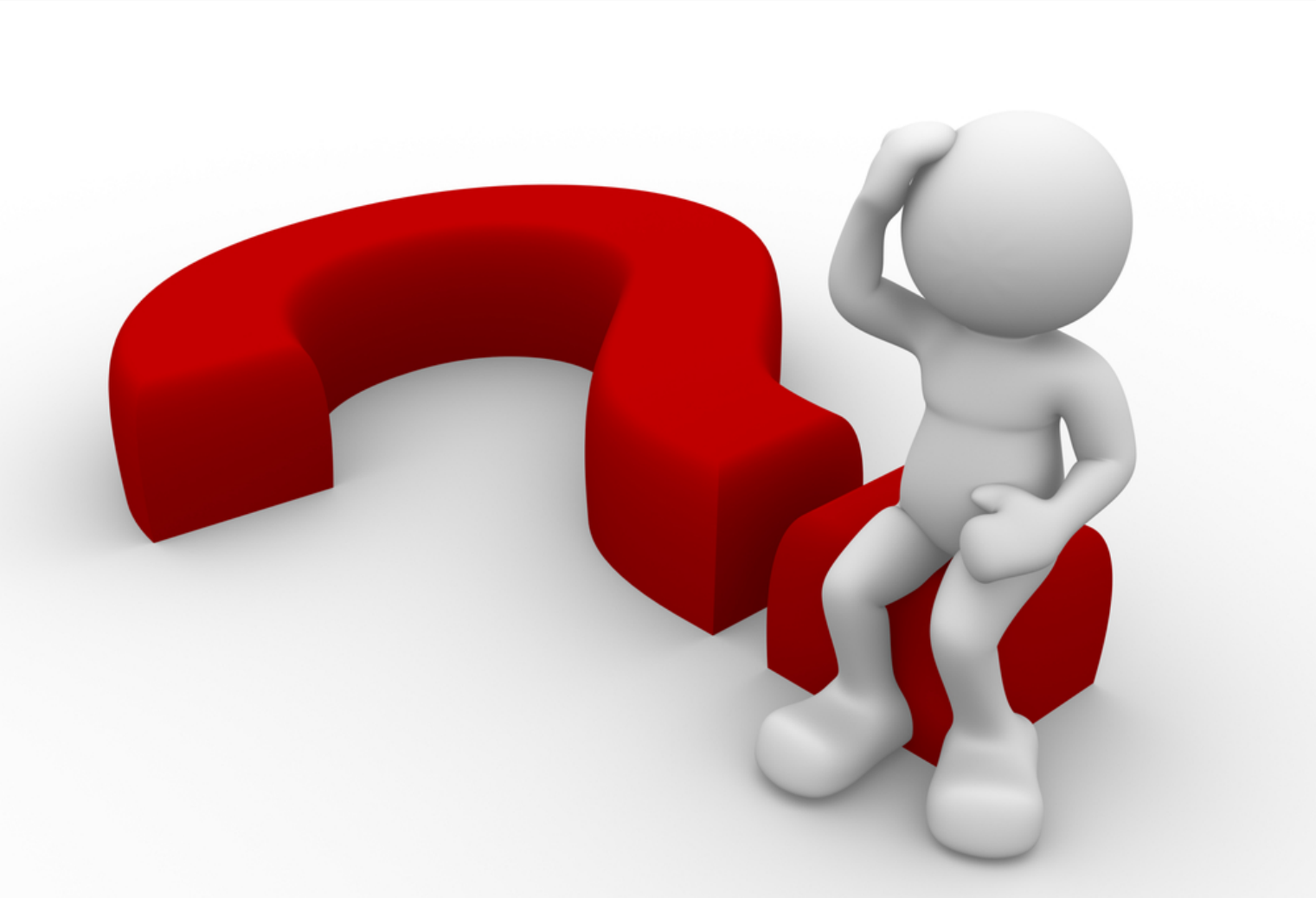
Hint!
For debugging you can use the **netstat** utility!

or

Trivia

Think!

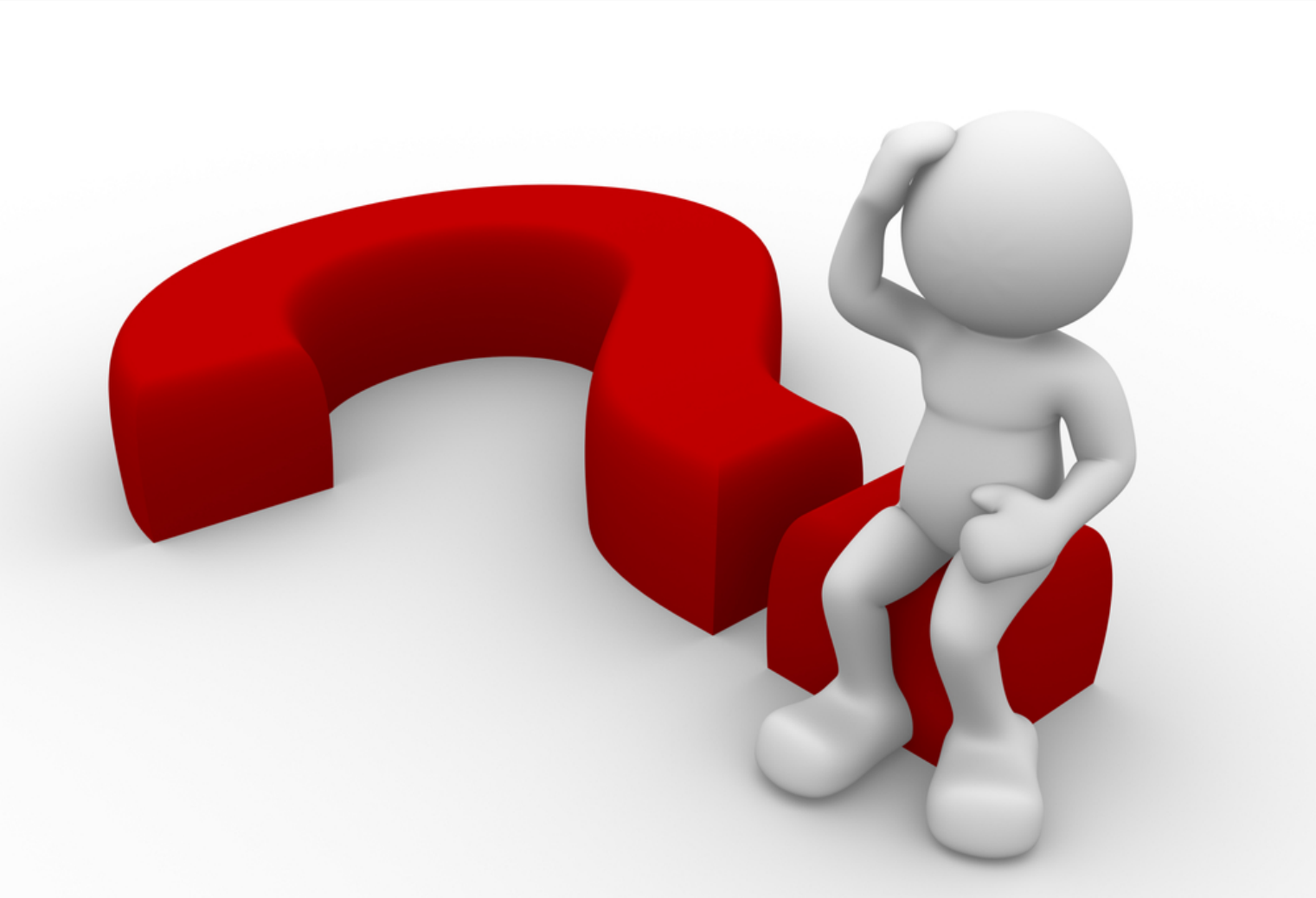
Which of the calls of the previous slides cause data to be transmitted or received over the network?



Trivia

Think!

Which of the calls of the previous slides cause data to be transmitted or received over the network? **NONE!**



TCP: Accepting connections

Prototype

- **accept()** is by default a blocking call
- It blocks until a connection arrives to the listening socket
- On success a new socket descriptor is returned, allowing the listening socket to handle the next available incoming connection
- The returned socket is used for sending and receiving data
- If **address** is not NULL, several information about the remote client are returned
- **address_len** before the call should contain the size of the **address** struct. After the call should contain the size of the returned structure

TCP: Connecting

Prototype

- Connects a socket with a remote host
- Like **bind()**, zero the contains of **address** before use and assign remote address and port in Network Byte Order
- If **bind()** was not used, the OS assigns the socket to all the available interfaces and to a random available port

TCP: Sending Data

Prototype

- **send()** is used to send data using a connection oriented protocol like TCP
- Returns the actual number of bytes sent
- Always check the return value for possible errors or to handle situations where the requested buffer did not sent completely

Question!
Does this call block?

TCP: Sending Data

Prototype

- **send()** is used to send data using a connection oriented protocol like TCP
- Returns the actual number of bytes sent
- Always check the return value for possible errors or to handle situations where the requested buffer did not sent completely

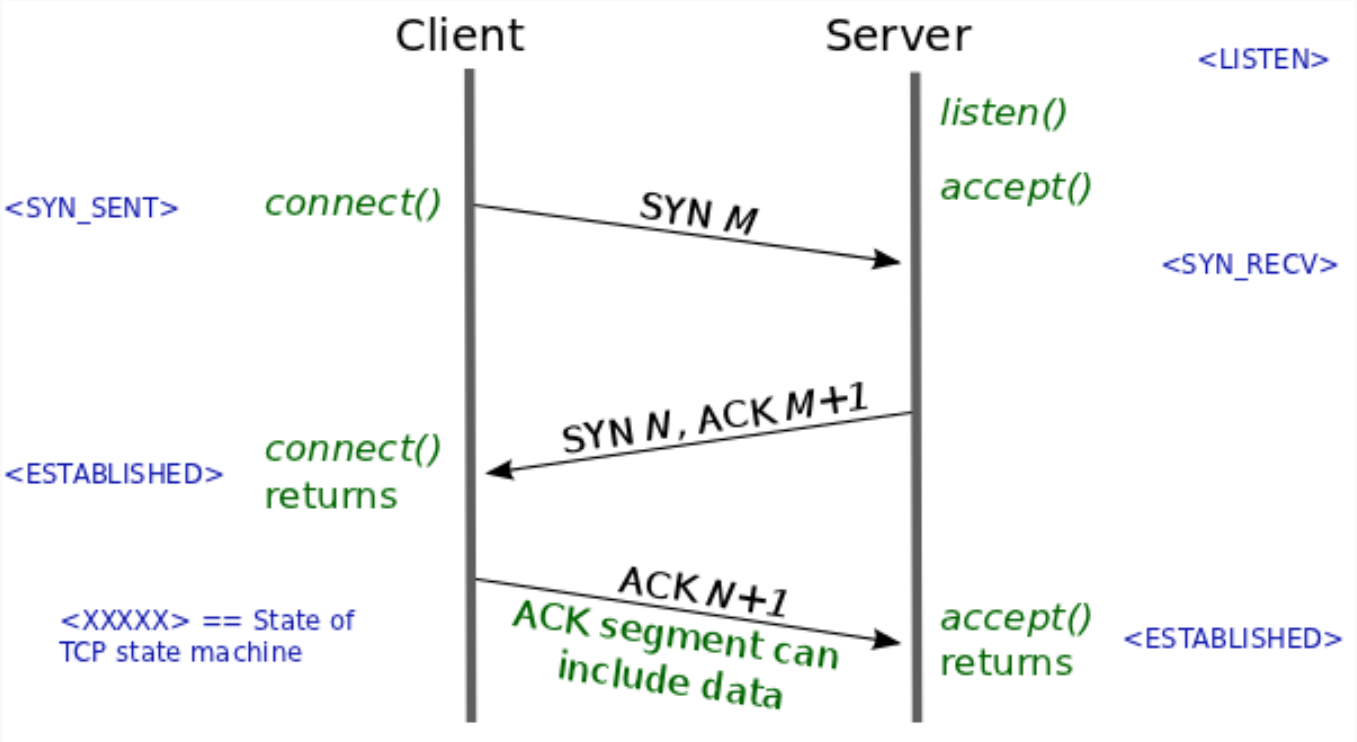
Question!
Does this call block? **YES!**

TCP: Receiving Data

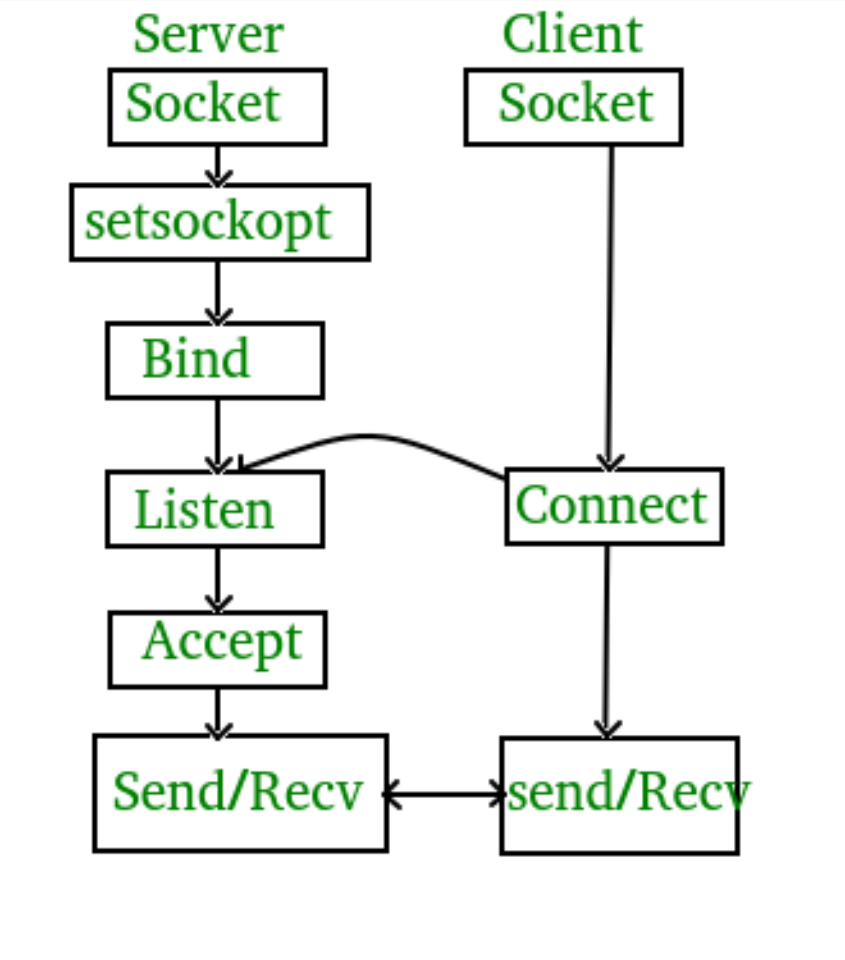
Prototype

- **recv()** is by default a blocking call that receives data from a connection-oriented opened socket
- **length** specifies the size of the buffer and the maximum allowed received data chunk
- Returns the number of bytes received from the network
- **recv()** may read less bytes than **length** parameter specified, so use only the return value for your logic
- If you do not want to block if no data are available, use non-blocking sockets (hard!) or **poll()**

TCP Overview 1/3



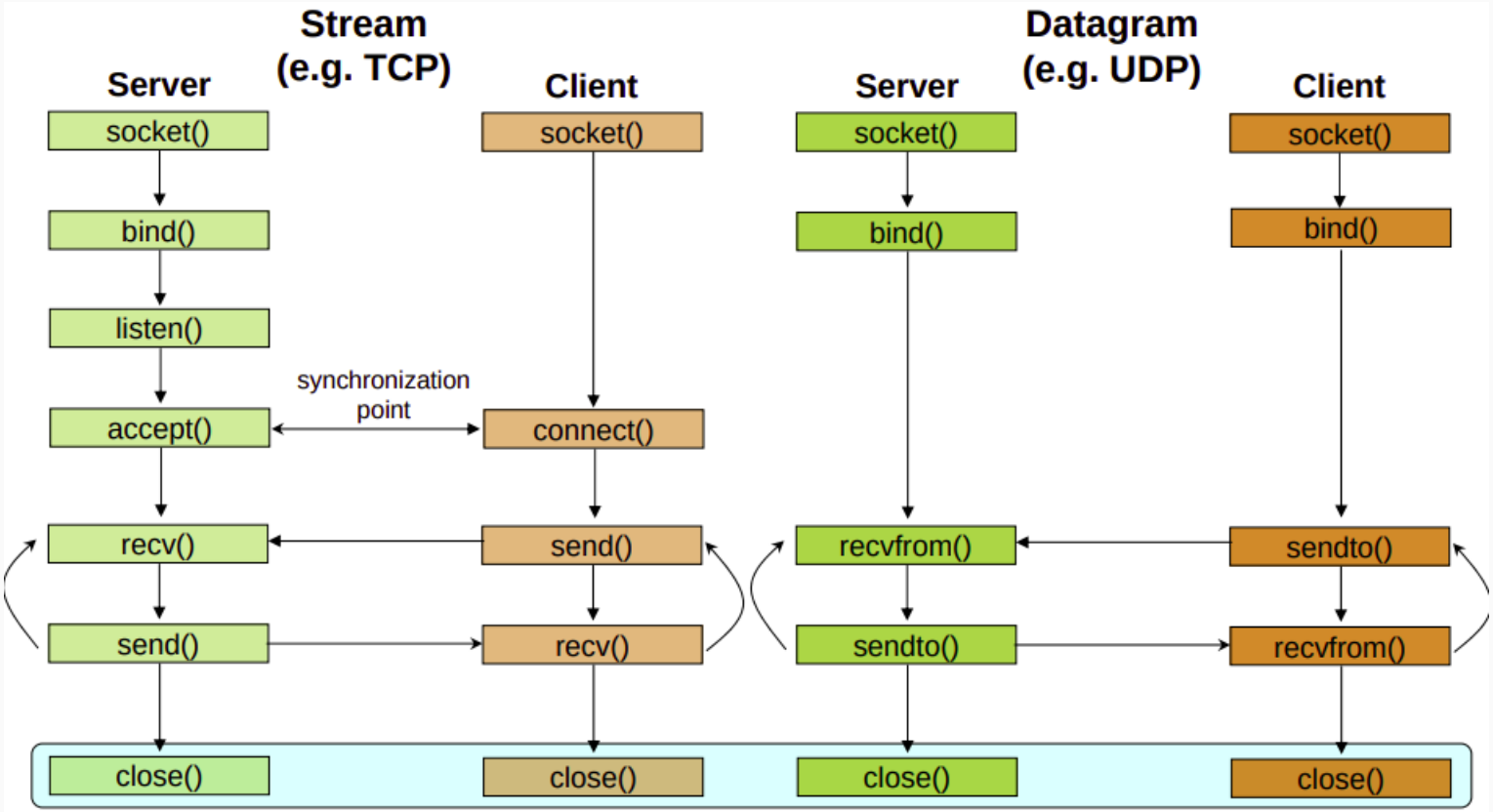
TCP Overview 2/3



TCP Overview 3/3

- **TCP Server:**
  - 1. using create(), Create TCP socket.
  - 2. using bind(), Bind the socket to server address.
  - 3. using listen(), put the server socket in a passive mode, where it waits for the client to approach the server to make a connection
  - 4. using accept(), At this point, connection is established between client and server, and they are ready to transfer data.
  - 5. Go back to Step 3.
- **TCP Client:**
  - 1. Create TCP socket.
  - 2. Connect newly created client socket to server.

Client - Server Communication



UDP Sockets

UDP: Creating the socket

- Creating a UDP socket is quite the same as with TCP

- Only **type** and **protocol** parameters are different
- **bind()** is also exactly the same for UDP too

UDP: Connection-less

UDP is connection-less!!!

No need to call **accept()** or **connect()**!!!

UDP: Receiving data

Prototype

- **length** specifies the length of the buffer in bytes
- **address** if not NULL, after the call should contain information about the remote host
- **address_len** is the size of the struct **address**
- Returns the number of bytes actually read. May be less that **length**

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**
- What if the message sent is greater that your buffer?

UDP: Problems at receiving

- Have in mind that **recvfrom()** is a blocking call
- How you can probe if data are available for receiving?
  - Use **poll()**
- What if the message sent is greater that your buffer?
  - Use **recvfrom()** in a loop with **poll()**

UDP: Sending data

Prototype

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

UDP: Sending data

Prototype

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

Trivia!
Does **sendto()** block?

UDP: Sending data

Prototype

- **length** is the number of the bytes that are going to be sent from buffer **message**
- **dest_addr** contains the address and port of the remote host
- Returns the number of bytes sent. May be less that **length** so the programmer should take care of it

Trivia!
Does **sendto()** block? **NO!**

Endianness

Endianness

- Networks are heterogenous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **nthohs()**, **nthohs()**, **htonl()**, **ntohl()**

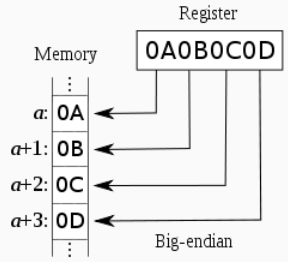
Endianness

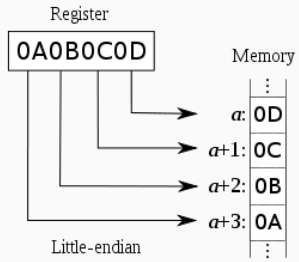
- Networks are heterogenous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **nthohs()**, **nthohs()**, **htonl()**, **ntohl()**

Trivia!
When sending large strings do we have to convert in Network Byte Order?

Endianness

- Networks are heterogenous with many different OS's, architectures, etc
- Endianess is a serious problem when sending data to other hosts
- When sending entities that are greater that a byte, **always** convert them in **Network Byte Order**
- By default Network Byte Order is Big-Endian
- Use **nthohs()**, **nthohs()**, **htonl()**, **ntohl()**





Trivia!
When sending large strings do we have to convert in Network Byte Order? **NO!**

Customize sockets

Prototype

- Default settings of a socket can be changed with **setsockopt()**
- The list of the available options can be found at the manpage of **socket(7)**

Accurate time measurements

- Most of the network experiments require accurate time measurements

- What can go wrong?
  - Low accuracy on time retrieval (e.g *gettimeofday()*)
  - Time adjustments during the experiment (NTP, PTP, e.t.c)

- **Solution:**
  - *clock_gettime()*
  - Use the *CLOCK_MONOTONIC_RAW* option

Useful man pages

- socket(7)
- ip(7)
- setsockopt(3p)
- tcp(7)
- udp(7)

Questions??