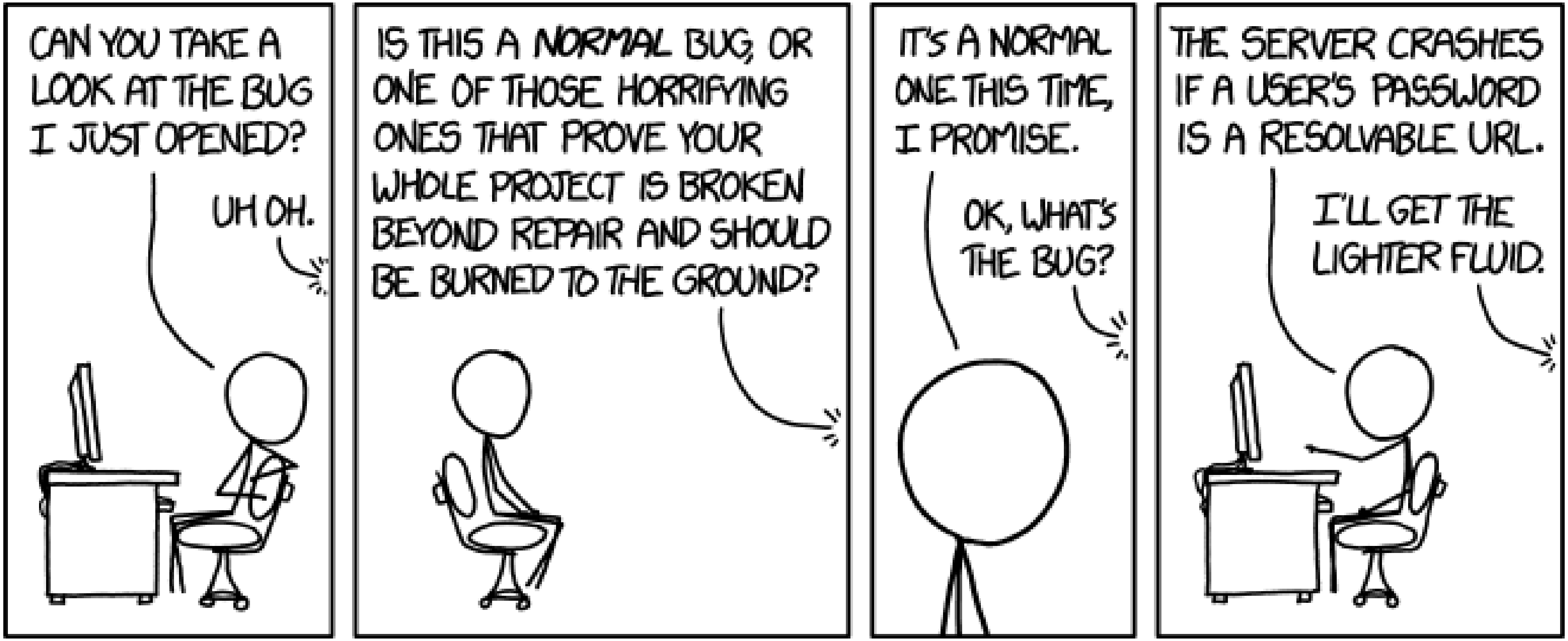

CAN YOU TAKE A LOOK AT THE BUG I JUST OPENED?
UH OH.
IS THIS A *NORMAL* BUG, OR ONE OF THOSE HORRIFYING ONES THAT PROVE YOUR WHOLE PROJECT IS BROKEN BEYOND REPAIR AND SHOULD BE BURNED TO THE GROUND?
IT'S A NORMAL ONE THIS TIME, I PROMISE.
OK, WHAT'S THE BUG?
THE SERVER CRASHES IF A USER'S PASSWORD IS A RESOLVABLE URL.
I'LL GET THE LIGHTER FLUID.

https://xkcd.com/1700

CS 60:
Computer Networks

Transport layer: TCP and UDP

Review from last class

The Application Layer (Layer 7) sends messages over the Internet between processes running on different hosts

The protocol (format and expected ordering) of the message varies by application

You can often find the message protocol in a Request for Comment (RFC)

Some Application Layer applications with defined protocols

- **RFC 1035: Domain Name System (DNS)** – looks up IP address given a host name like google.com
- **RFC 5321: Simple Mail Transfer Protocol (SMTP)** – Send email messages *to* (and between) mail servers; use IMAP (or POP or HTTP) to get message *from* mail servers
- **RFC 9112: HyperText Transfer Protocol (HTTP)** – web servers and browser applications format messages to request and return web pages

Review: Transport layer moves segments (or datagrams) across a network

Conceptual network layers

7) Application

Interacts with application programs to send ***messages***
Applications assigned a port, multiple instances can run (many browser pages)
Examples: HTTP, SSH, FTP, SMTP, DNS

4) Transport

Moves ***segments*** ***(or*** ***datagrams)***
May provide error control, flow control, application addressing (ports)
Examples: TCP (connection-oriented), UDP (connectionless)
TCP provides sequencing, dropped packet resend, traffic congestion routing

3) Network (IP)

Moves ***packets*** between local area networks (routing)
Each computer on the Internet identified by an IP address (IP v4 or v6)
Also called Layer 3 or IP layer (ICMP Ping is here)

2) Link (MAC)

Moves ***frames*** within a local area network (switching)
Each computer identified by a MAC address on its Network Interface Card (NIC)
Also called Layer 2, MAC layer, Data Link layer, or Ethernet layer

1) Physical

How data is physically transmitted

- Transmitter converts logical 1 and 0 ***bits*** to electrical/light pulses or phase/amplitude of radio frequency (RF) and sends down wire or over air
- Receiver converts electrical/light or RF back to logical 1 and 0 bits

https://www.ibm.com/think/topics/osi-model

Agenda

➡ 1. Transport (Layer 4) services

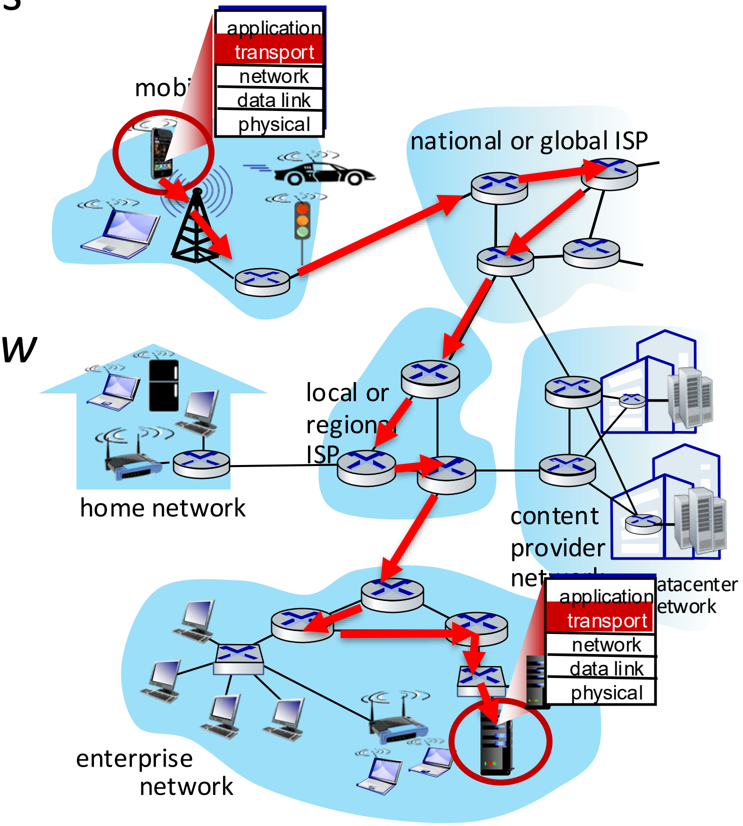
2. Multiplexing and Demultiplexing: How does data get to the right application?

3. Connectionless Transport: UDP

4. Exercises

Transportation services provide logical rather than physical connections

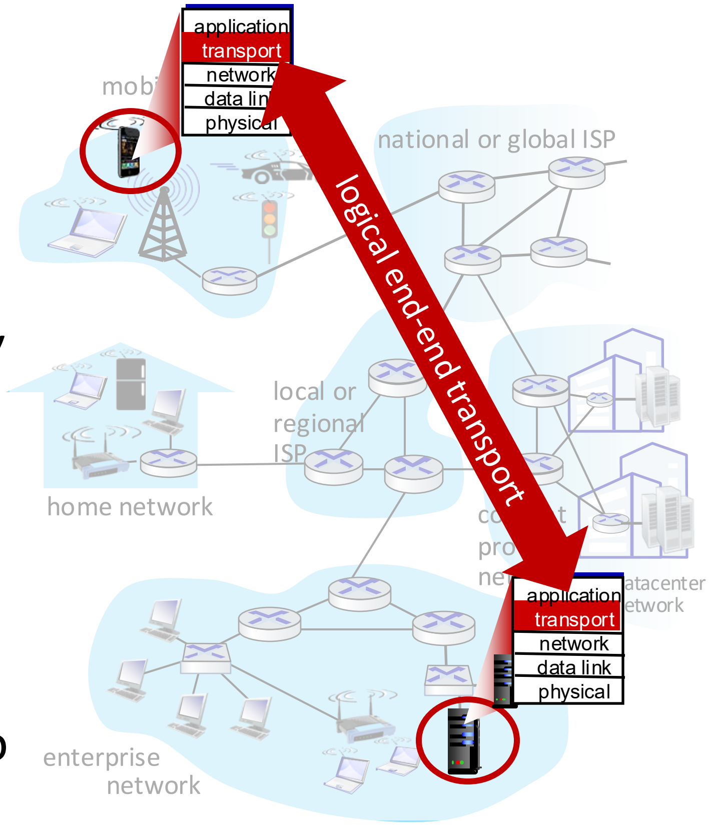
- Provide *logical communication* between application processes running on different hosts
  - From application’s perspective, as if hosts were directly connected
  - Hosts may be far apart, connected by many routers and types of links
  - Applications do not need to worry about *how* data traversed the network/links



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transportation services provide logical rather than physical connections

- Provide *logical communication* between application processes running on different hosts
  - From application's perspective, as if hosts were directly connected
  - Hosts may be far apart, connected by many routers and types of links
  - Applications do not need to worry about *how* data traversed the network/links
- Transport protocols actions are in end systems, not routers:
  - Routers do not know about Layer 4
  - Sender: breaks application messages into *segments* (TCP) or *datagrams* (UDP), passes to Network Layer
  - Receiver: reassembles segments into messages, passes to Application Layer via socket



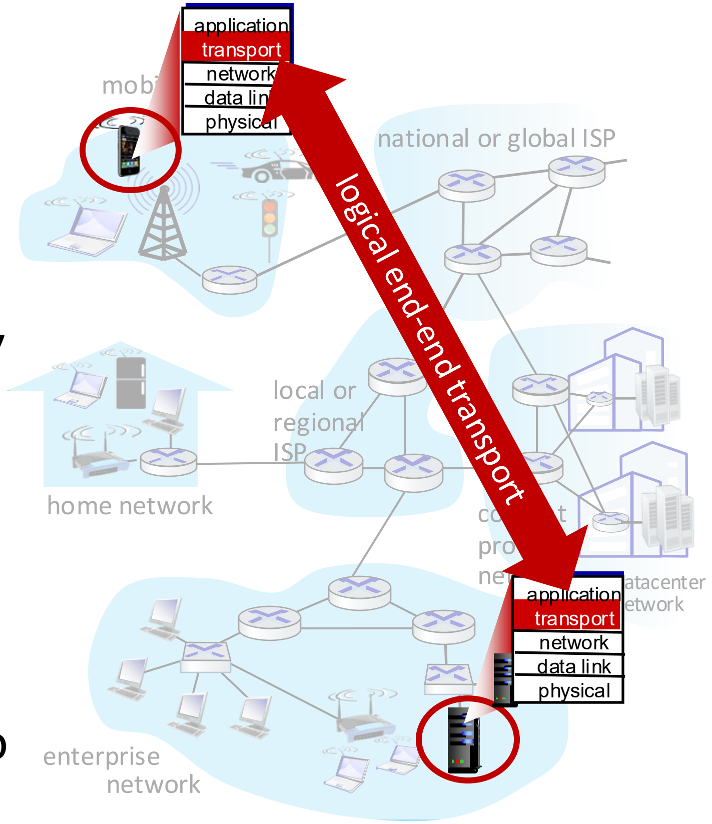
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transportation services provide logical rather than physical connections

- Provide *logical communication* between application processes running on different hosts
  - From application’s perspective, as if hosts were directly connected
  - Hosts may be far apart, connected by many routers and types of links
  - Applications do not need to worry about *how* data traversed the network/links
- Transport protocols actions are in end systems, not routers:
  - Routers do not know about Layer 4
  - Sender: breaks application messages into *segments* (TCP) or *datagrams* (UDP), passes to Network Layer
  - Receiver: reassembles segments into messages, passes to Application Layer via socket
- TCP and UDP are the transport layers used by the Internet
  - Others exist, but the Internet primarily uses these two

Transport Layer connects processes

Network Layer connects hosts



Router determine path through network

Routers do not look at Transport Layer headers

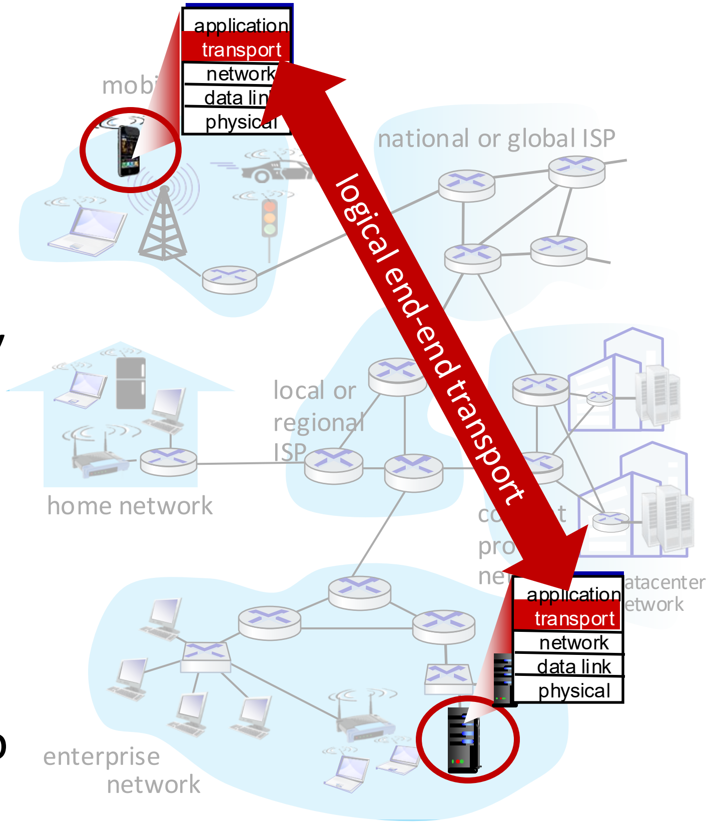
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transportation services provide logical rather than physical connections

- Provide *logical communication* between application processes running on different hosts
  - From application's perspective, as if hosts were directly connected
  - Hosts may be far apart, connected by many routers and types of links
  - Applications do not need to worry about *how* data traversed the network/links
- Transport protocols actions are in end systems, not routers:
  - Routers do not know about Layer 4
  - Sender: breaks application messages into *segments* (TCP) or *datagrams* (UDP), passes to Network Layer
  - Receiver: reassembles segments into messages, passes to Application Layer via socket
- TCP and UDP are the transport layers used by the Internet
  - Others exist, but the Internet primarily uses these two

Transport Layer connects processes

Network Layer connects hosts



Packets may get lost or delayed on Internet

Delivery is *not* guaranteed by the Network Layer

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

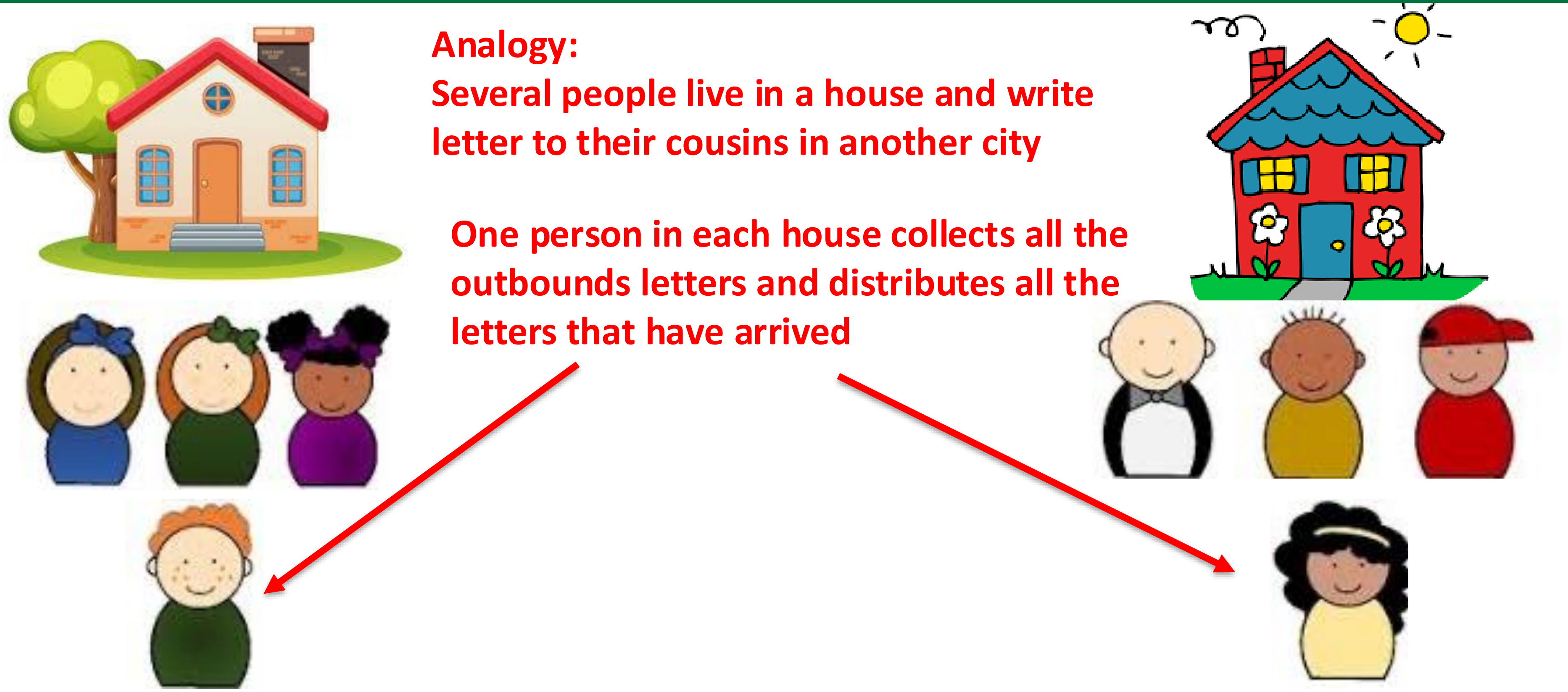
Transport layer connects applications, network layer connects hosts

Analogy:

Several people live in a house and write letter to their cousins in another city

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts



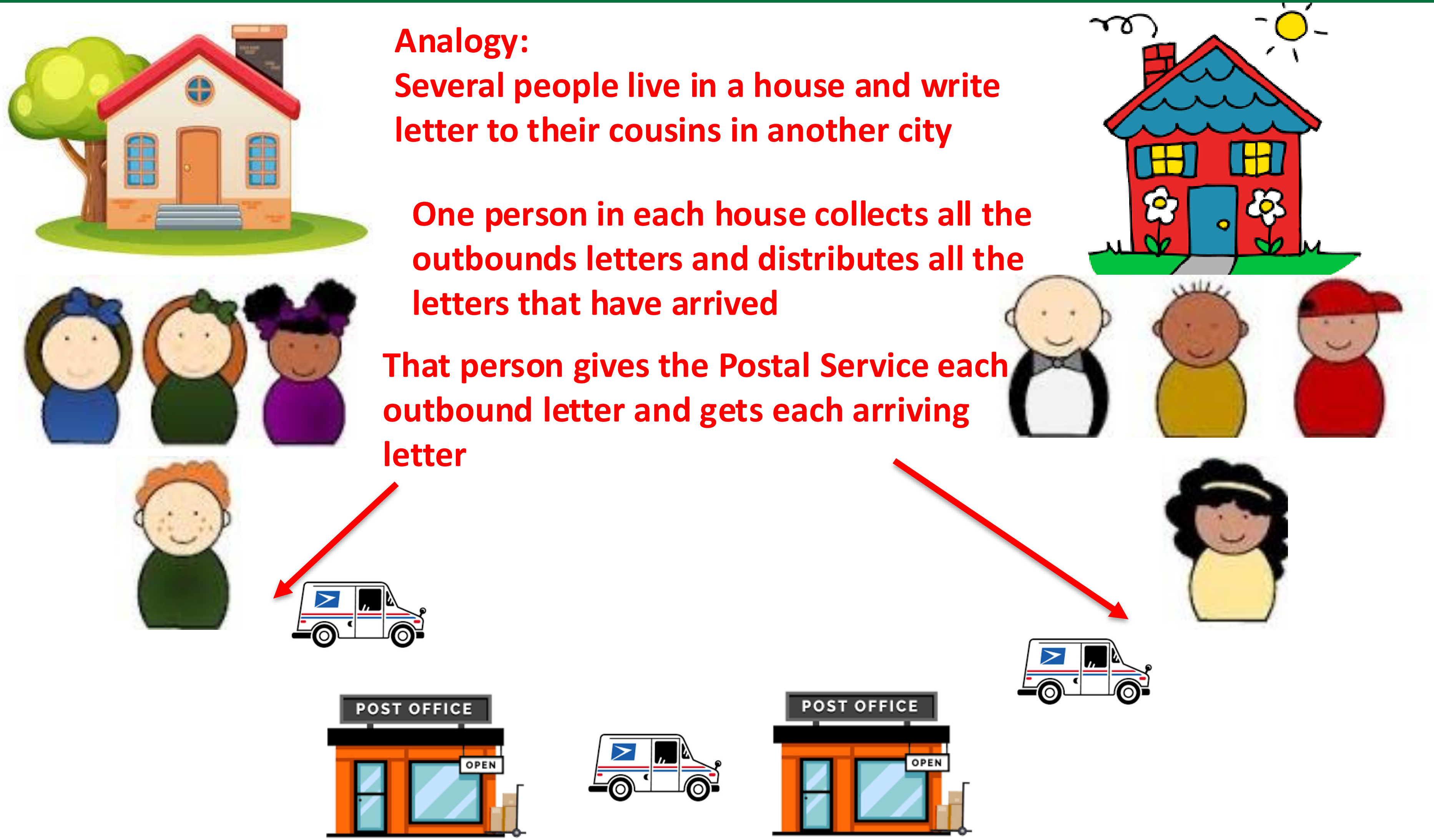
Analogy:

Several people live in a house and write letter to their cousins in another city

One person in each house collects all the outbounds letters and distributes all the letters that have arrived

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts



Analogy:

Several people live in a house and write letter to their cousins in another city

One person in each house collects all the outbounds letters and distributes all the letters that have arrived

That person gives the Postal Service each outbound letter and gets each arriving letter

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts

Analogy:

Several people live in a house and write letter to their cousins in another city

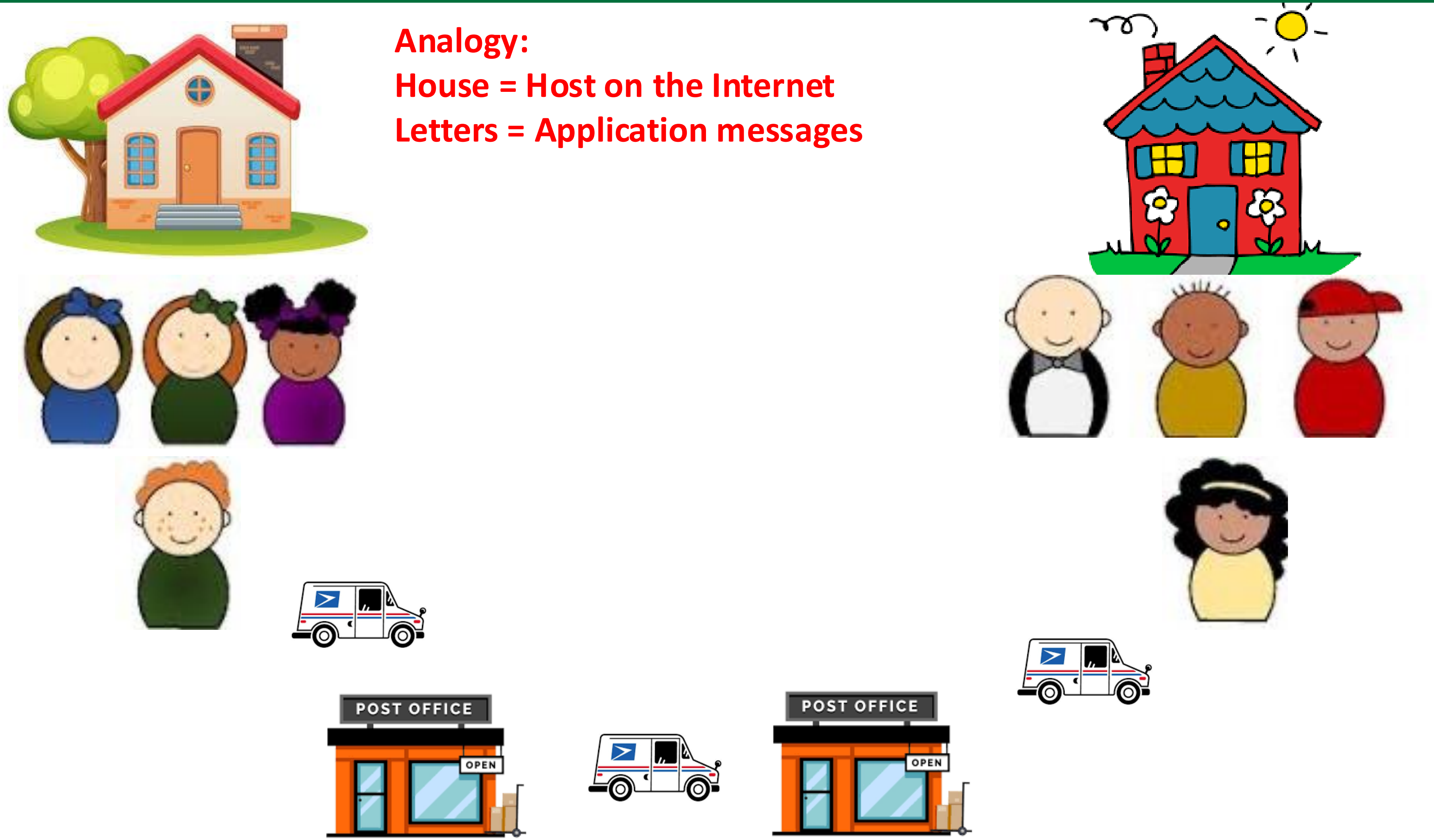
One person in each house collects all the outbounds letters and distributes all the letters that have arrived

That person gives the Postal Service each outbound letter and gets each arriving letter

The Postal Service handles sending letters between cities

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts



Analogy:
House = Host on the Internet
Letters = Application messages

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts

Analogy:

House = Host on the Internet

Letters = Application messages

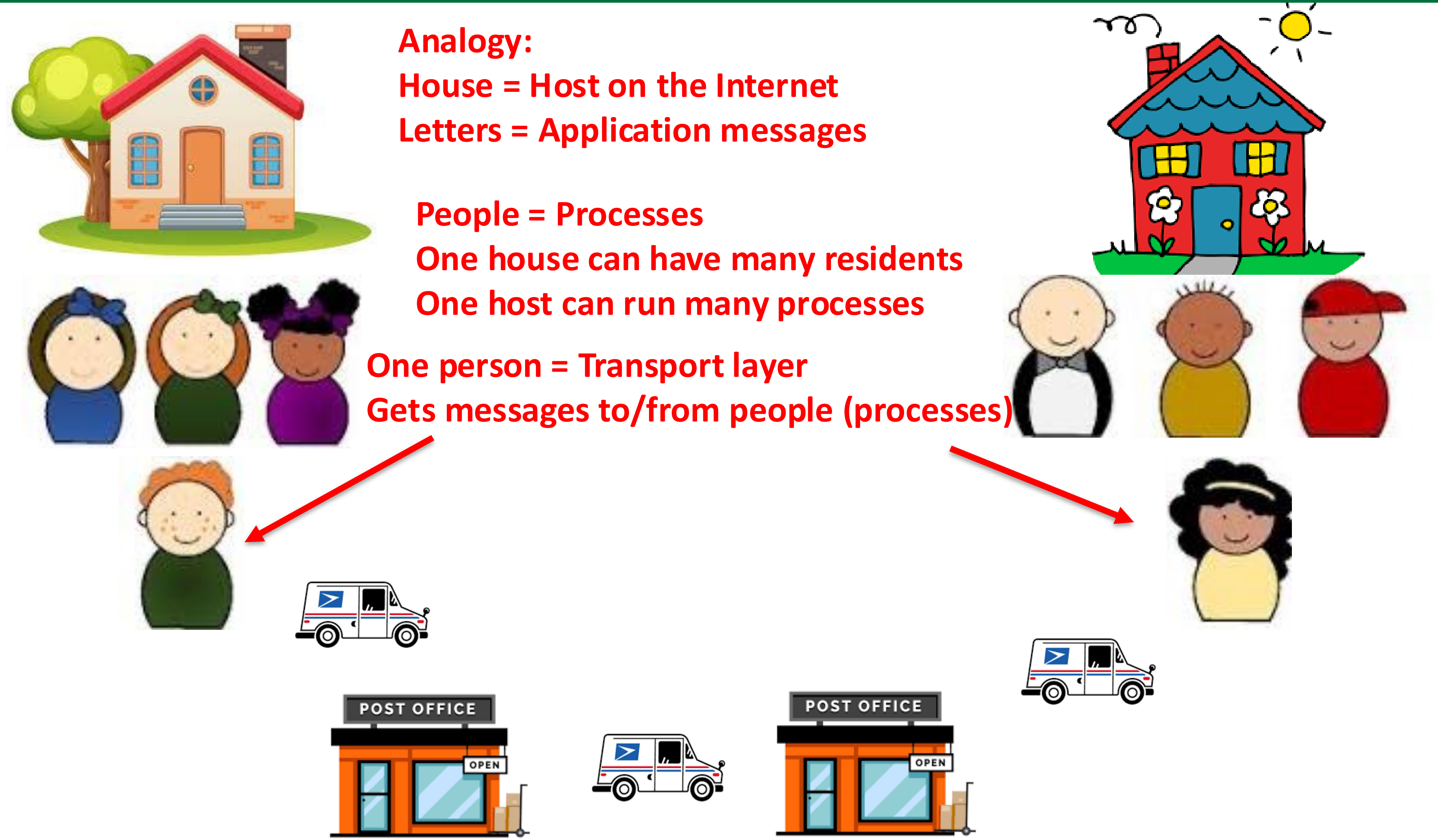
People = Processes

One house can have many residents

One host can run many processes

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts



Analogy:
House = Host on the Internet
Letters = Application messages

People = Processes
One house can have many residents
One host can run many processes

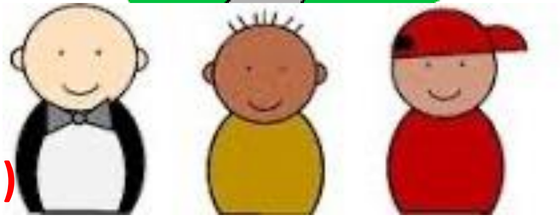
One person = Transport layer
Gets messages to/from people (processes)

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts

Analogy:
House = Host on the Internet
Letters = Application messages

People = Processes
One house can have many residents
One host can run many processes



One person = Transport Layer
Gets messages to/from people (processes)

Postal service = Network Layer
Handles routing between Post Offices (routers) and houses (hosts)

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts

Analogy:

House = Host on the Internet

Letters = Application messages

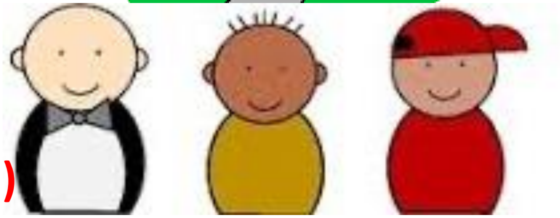
People = Processes

One house can have many residents

One host can run many processes

One person = Transport Layer

Gets messages to/from people (processes)



Postal service = Network Layer

Handles routing between Post Offices (routers) and houses (hosts)

Trucks = Physical layer

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer connects applications, network layer connects hosts

Analogy:

House = Host on the Internet

Letters = Application messages

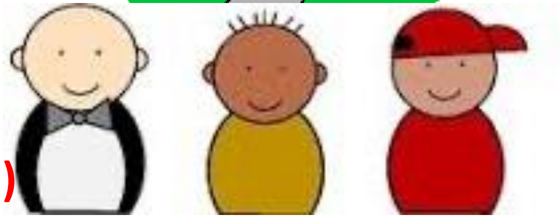
People = Processes

One house can have many residents

One host can run many processes

One person = Transport Layer

Gets messages to/from people (processes)



Postal service = Network Layer

Handles routing between Post Offices and (routers) and houses (hosts)

Trucks = Physical layer

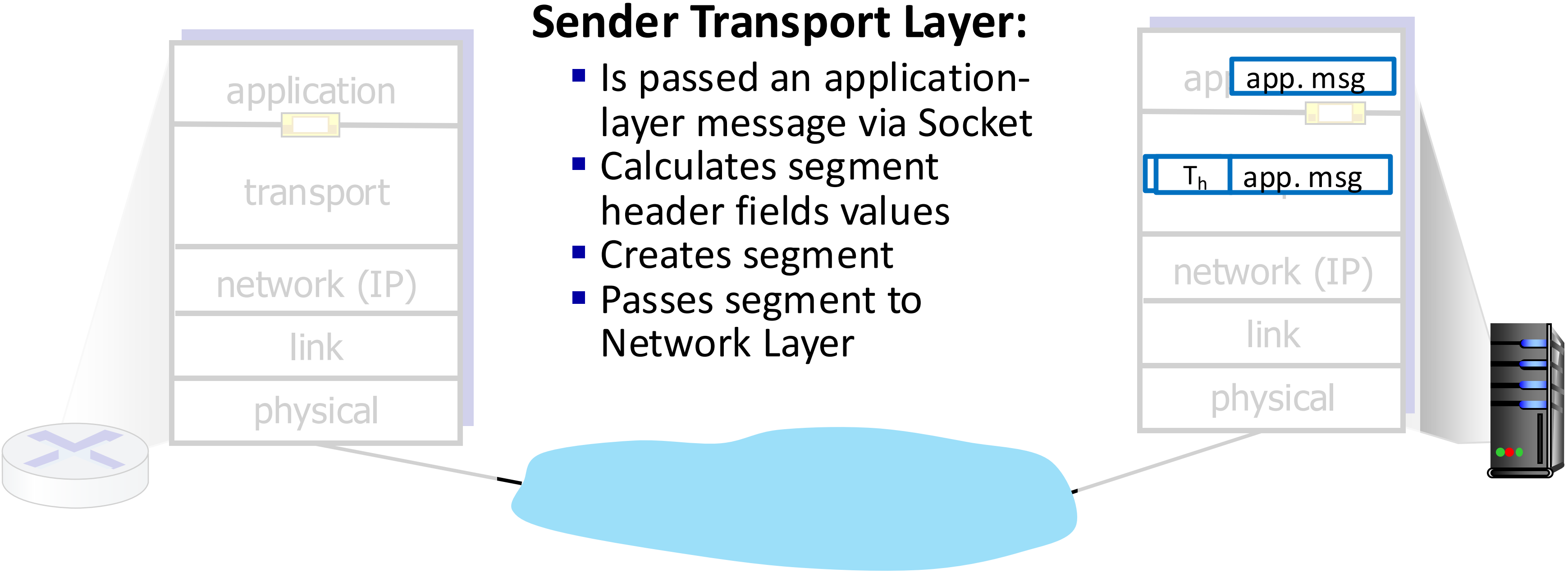
Transport Layer connects processes

Network Layer connects hosts

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer actions: Sender adds transport headers to app message

Applications on two different hosts talk to each other over the Internet



Sender Transport Layer:

- Is passed an application-layer message via Socket
- Calculates segment header fields values
- Creates segment
- Passes segment to Network Layer

Primary header fields are source and destination port (plus error detection checksum)
Ports identify specific applications on hosts
Recall a host may run many copies of an application (e.g., multiple web browsers or tabs)

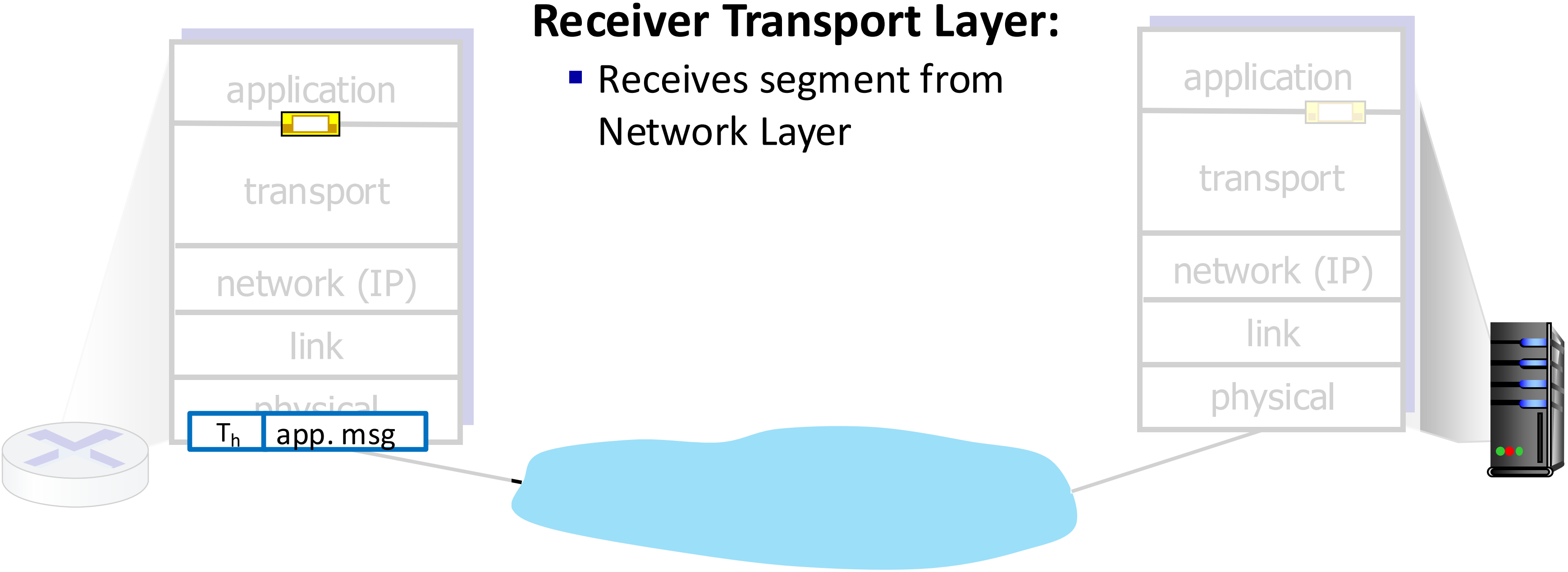
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer actions: Receiver removes headers and passes message to app

Applications on two different hosts talk to each other over the Internet

Receiver Transport Layer:

- Receives segment from Network Layer



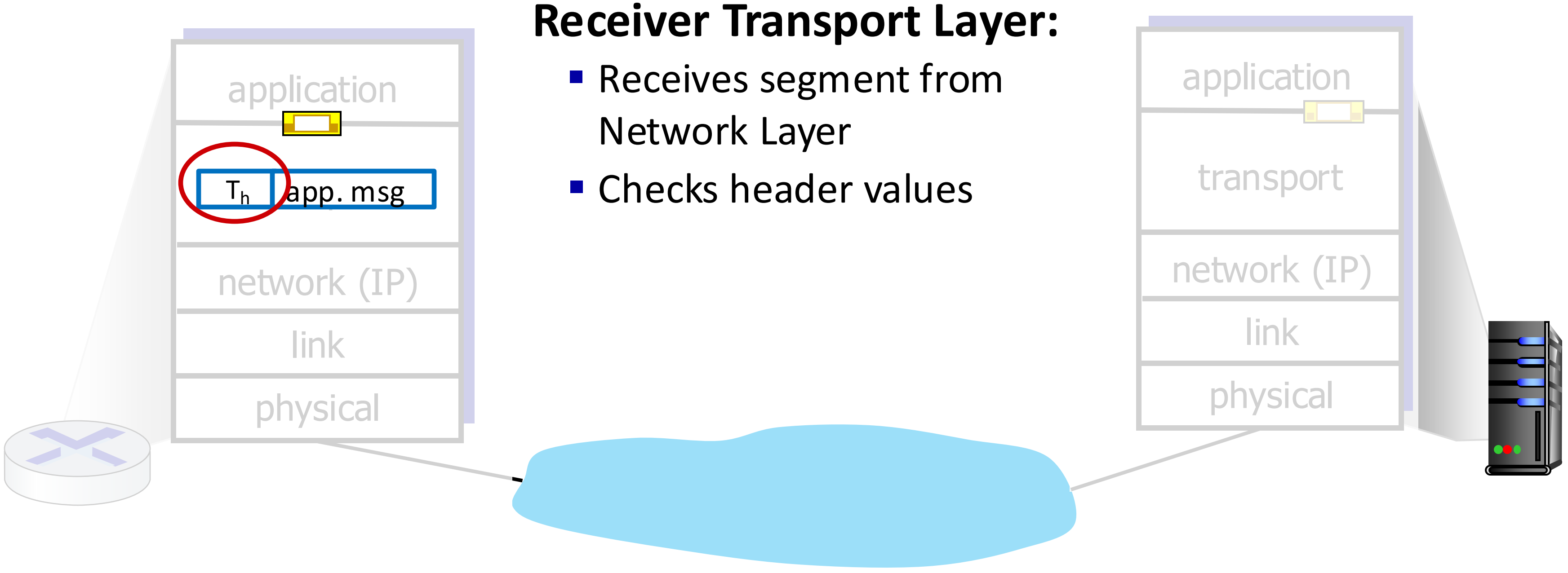
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer actions: Receiver removes headers and passes message to app

Applications on two different hosts talk to each other over the Internet

Receiver Transport Layer:

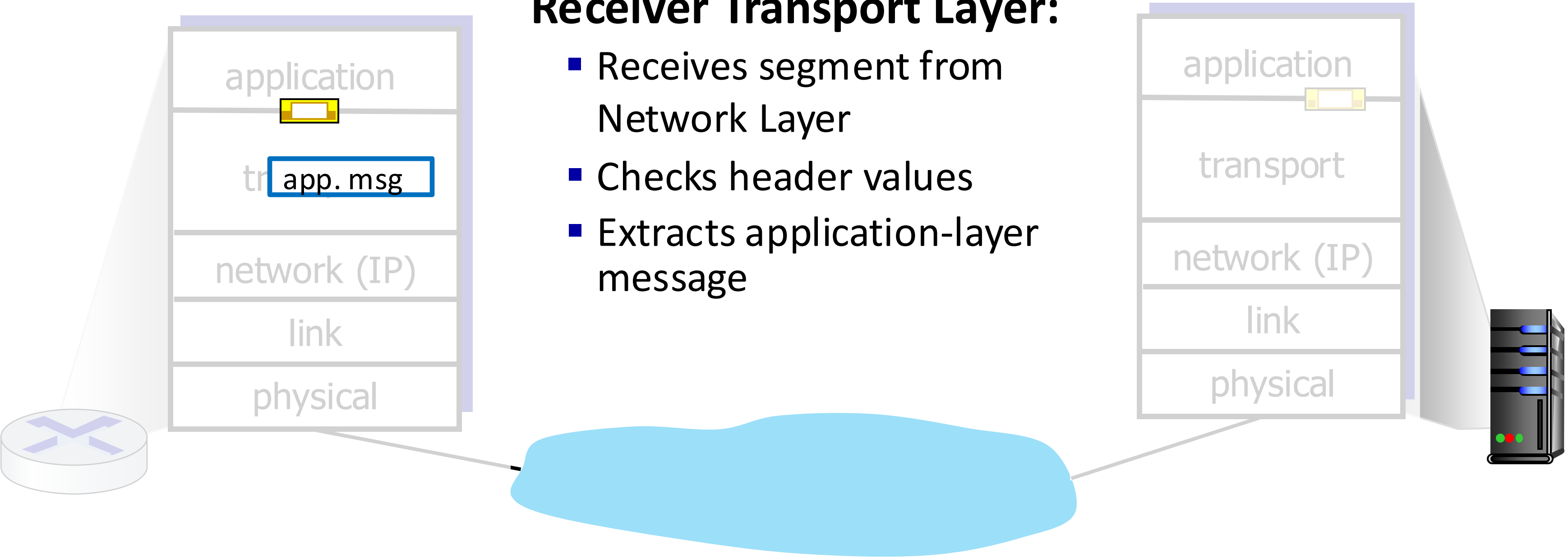
- Receives segment from Network Layer
- Checks header values



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer actions: Receiver removes headers and passes message to app

Applications on two different hosts talk to each other over the Internet



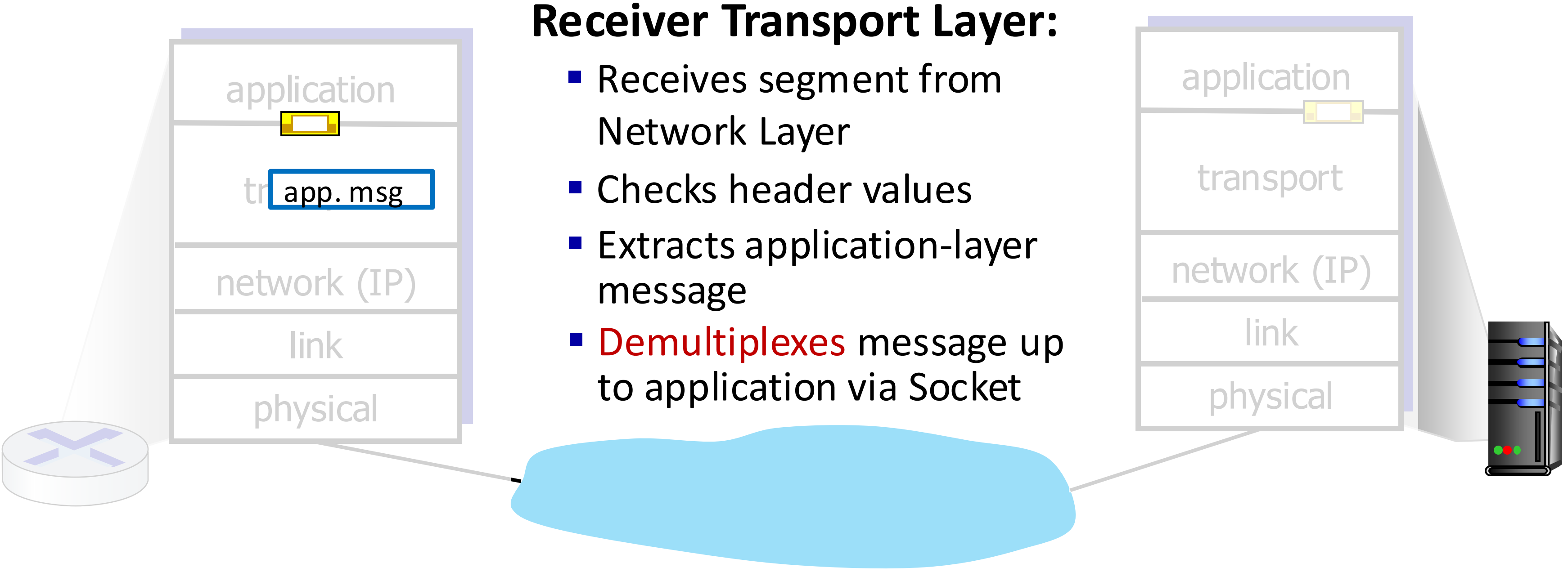
Receiver Transport Layer:

- Receives segment from Network Layer
- Checks header values
- Extracts application-layer message

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Transport layer actions: Receiver removes headers and passes message to app

Applications on two different hosts talk to each other over the Internet



Receiver Transport Layer:

- Receives segment from Network Layer
- Checks header values
- Extracts application-layer message
- Demultiplexes message up to application via Socket

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Agenda

1. Transport (Layer 4) services
2. Multiplexing and Demultiplexing: How does data get to the right application?
3. Connectionless Transport: UDP
4. Exercises

Review: Layer 4 ports identify an application running on a host

Ports

Your computer is running multiple applications at the same time (email, web browsing, others)

Without a means to differentiate, all received data would go to the same place (e.g., web browser gets email traffic and vice versa)

Ports identify applications by a number that ranges from 0 to 65,535 = 2^{16}-1

Servers run applications that listen for connections on well-known ports

Clients connect to servers on these ports

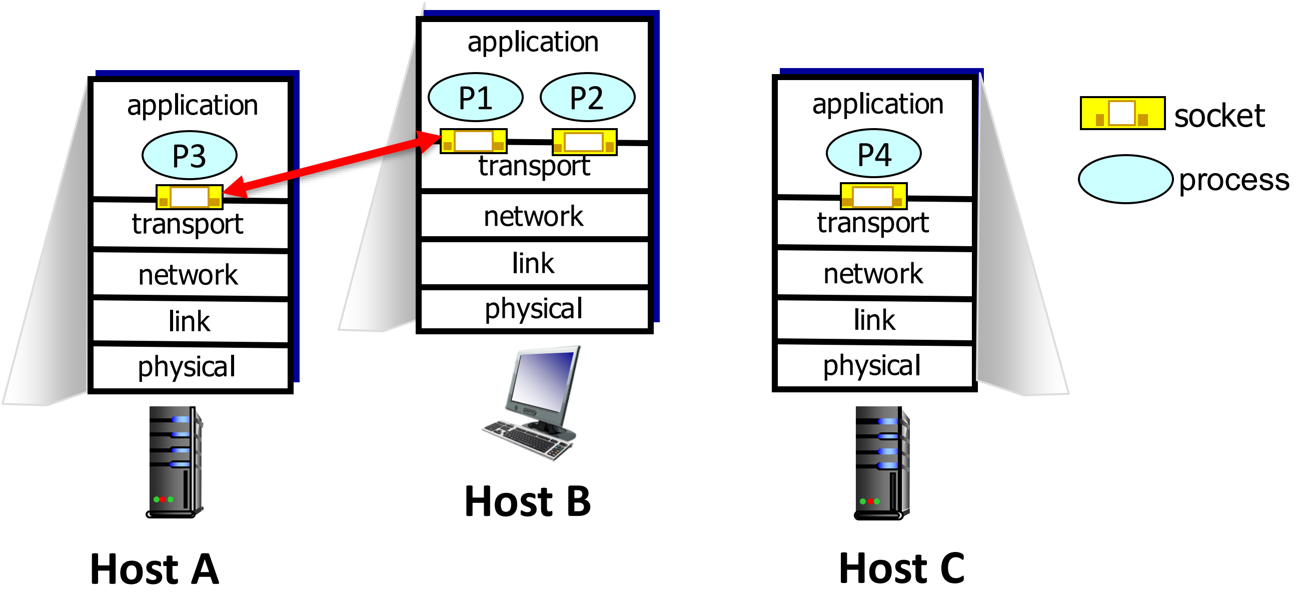
Well known ports	
Port	**Service**
20 and 21	FTP
22	SSH
53	DNS
80	HTTP
443	Encrypted HTTP
587 (old 25)	Email (SMTP)

Ports
0 - 1023 are well-known ports
Commonly reserved for system apps
1024-49,151 user or registered ports
49,152-65,535 ephemeral ports

Find known ports at /etc/services

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

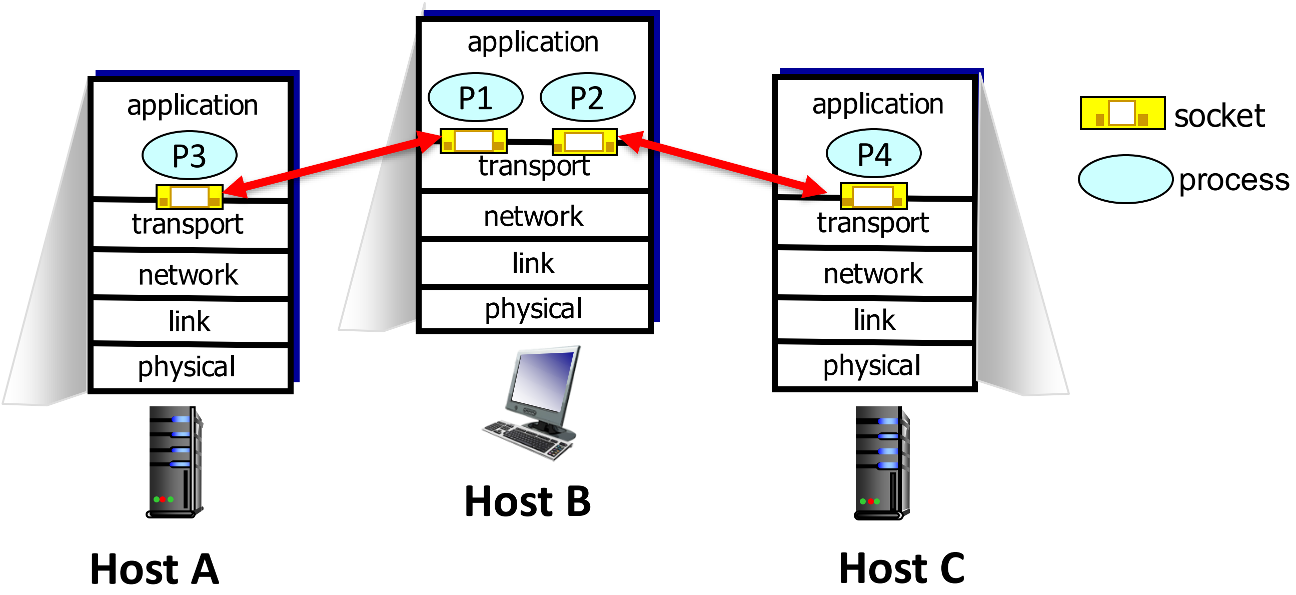
P1 on Host B talks to P3 on Host A through a Socket



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

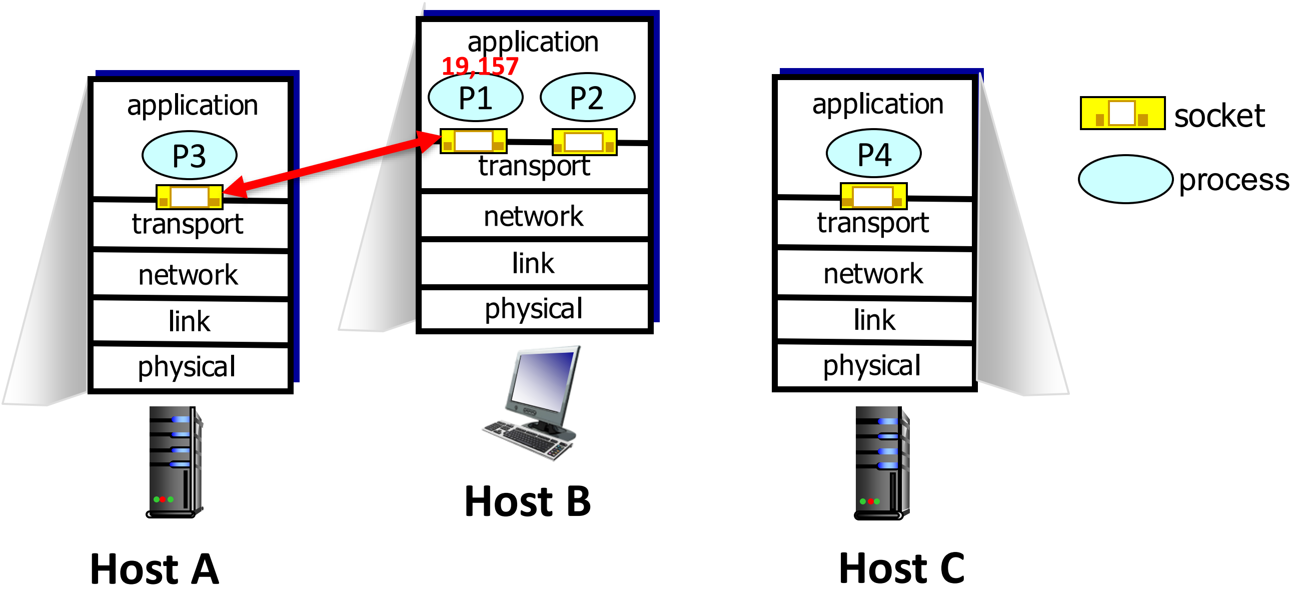
P1 on Host B talks to P3 on Host A through a Socket
P2 on Host B talks to P4 on Host C through another Socket



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

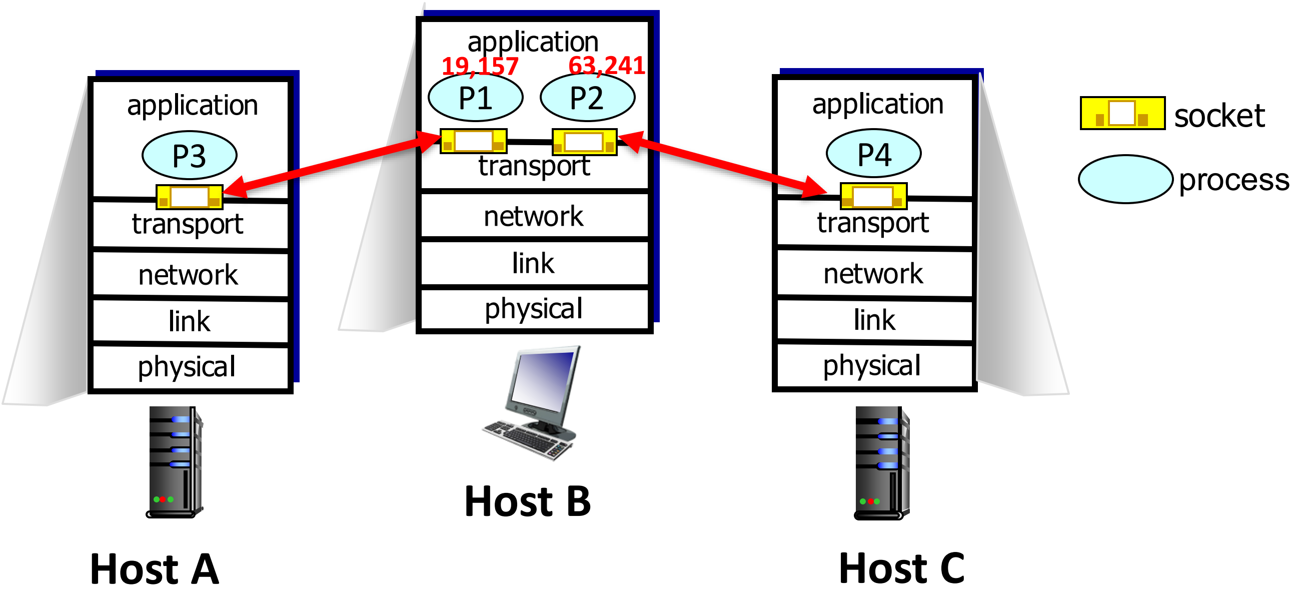
Host B created a Socket with an unused Port number to talk to P3 (19,157)



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

Host B created a Socket with an unused Port number to talk to P3 (19,157)
Likewise, Host B created a different Socket with a different Port number for P4 (63,241)



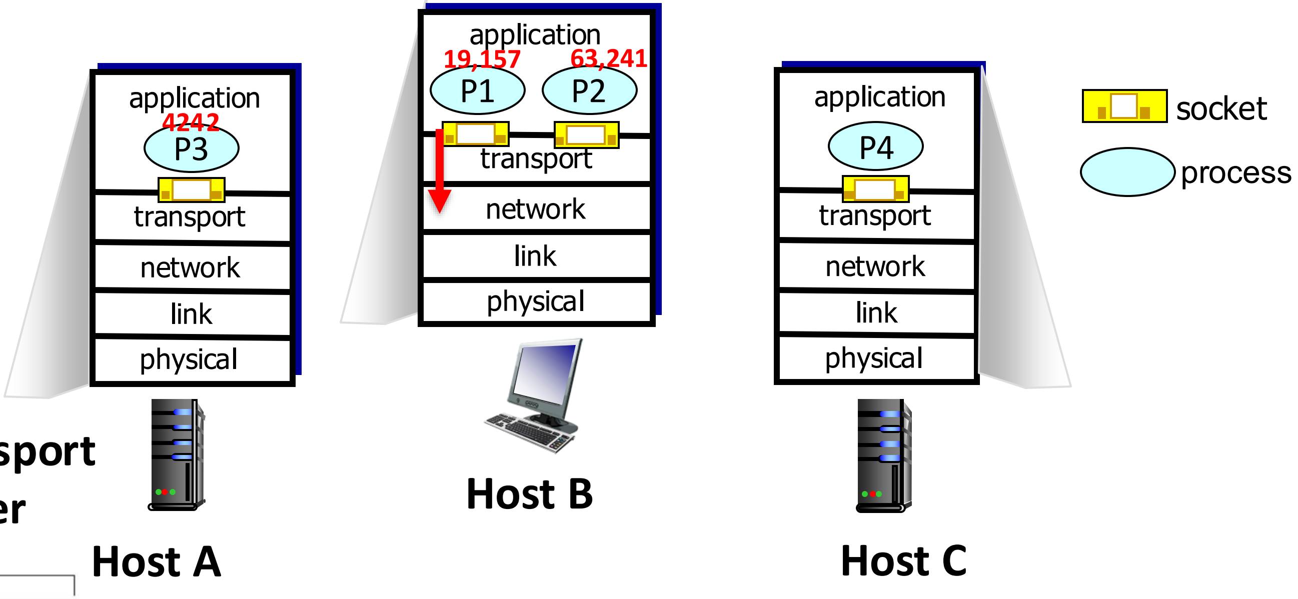
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

When Host B *sends* a Segment it adds a Transport Layer headers

- **Sets header source port to the Socket's port number (19,157)**
- **Sets header destination port (4242) to the other host's port number**
- **Also adds a checksum (covered soon)**

Then passes Segment to the Network Layer



Simplified Transport Layer Header

32 bits

Source port #	Dest. port #
Other header fields	
Application data (message)	

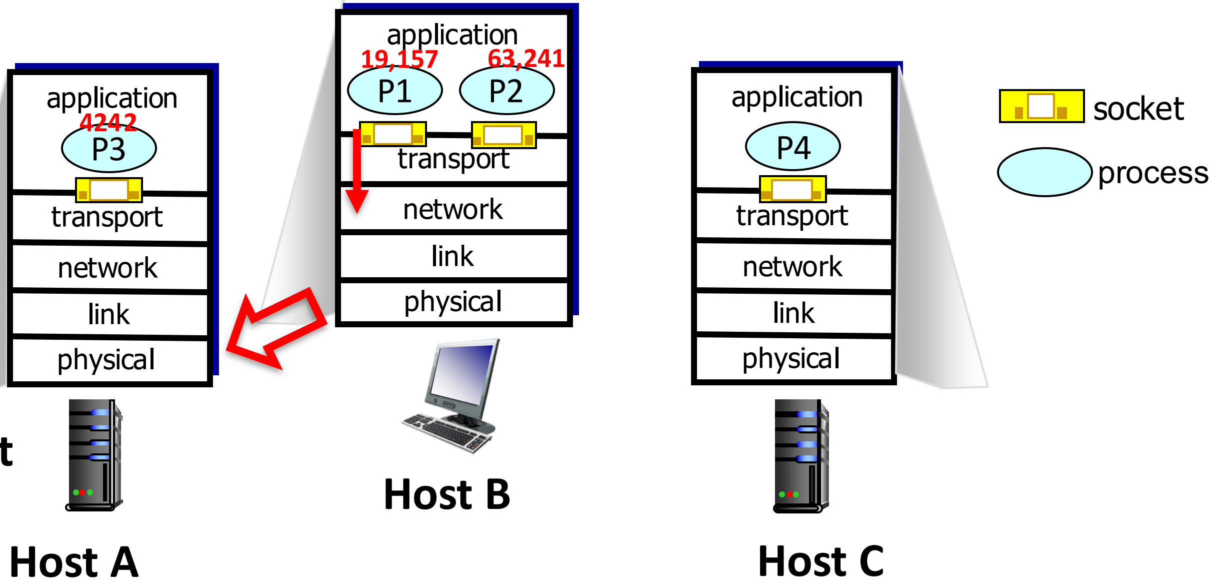
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

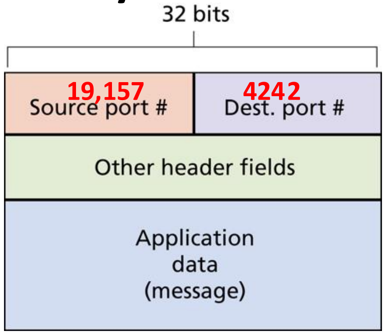
When Host B *sends* a Segment it adds a Transport Layer headers

- **Sets header source port to the Socket’s port number (19,157)**
- **Sets header destination port (4242) to the other host’s port number**
- **Also adds a checksum (covered soon)**

Then passes Segment to the Network Layer



Simplified Transport Layer Header



19,157 Source port #	4242 Dest. port #
Other header fields	
Application data (message)	

Network layer sends Segment to Link Layer

Link Layer sends frame to Physical Layer

Physical Layer sends data across the network

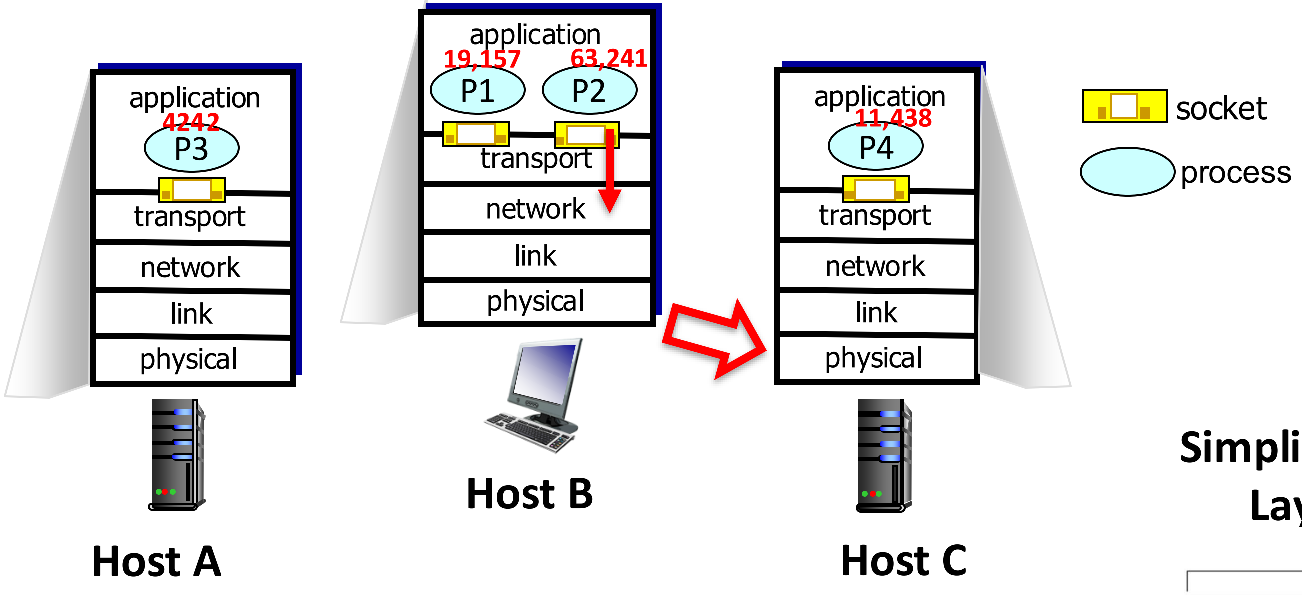
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

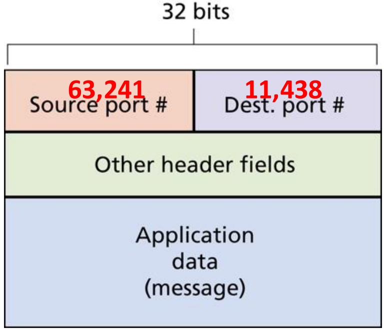
When Host B *sends* a Segment it adds a Transport Layer headers

- **Sets header source port to the Socket's port number (19,157)**
- **Sets header destination port to the other host's port number**
- **Also adds a checksum (covered soon)**

Then passes Segment to the Network Layer



Simplified Transport Layer Header



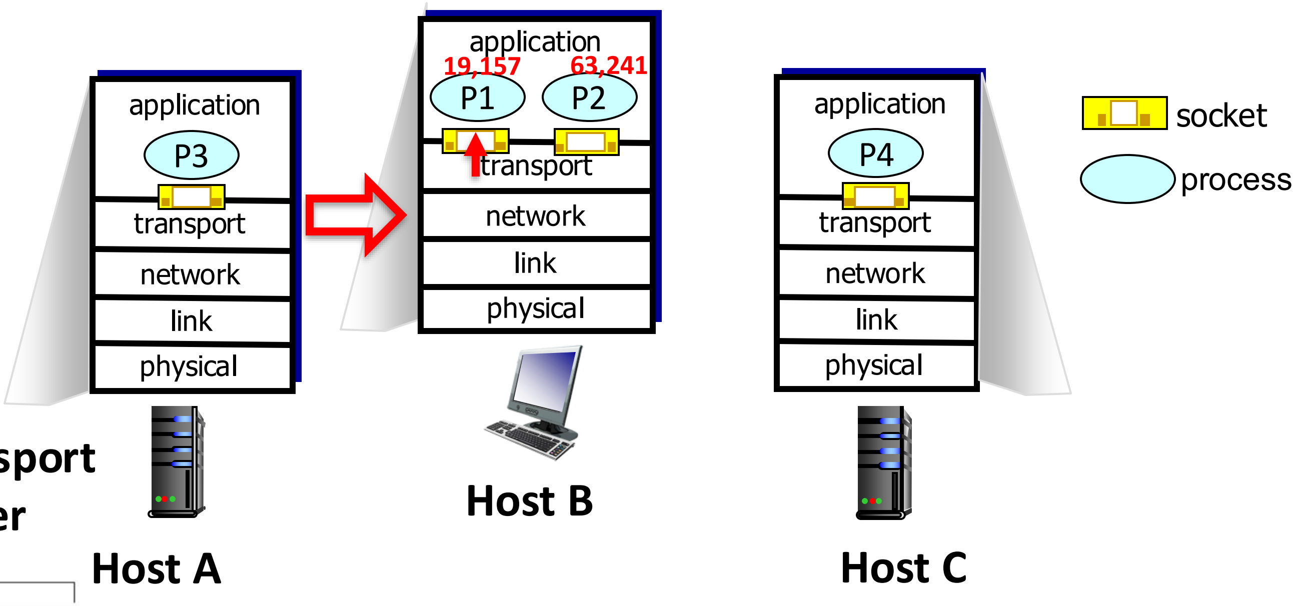
Choosing which Socket to use to send is called *multiplexing*

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

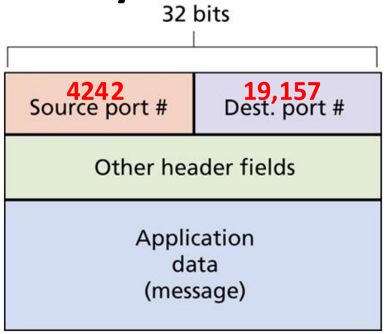
Suppose Host B is running two process, P1 and P2 talking to processes on other hosts

When Host B *receives* a Segment, it examines the Transport Layer headers

- **Finds the destination port in the Transport header (19,157)**
- **Sends data to the Socket with that port number**
- **(Technically it also checks the IP address, but we will cover that soon)**



Simplified Transport Layer Header



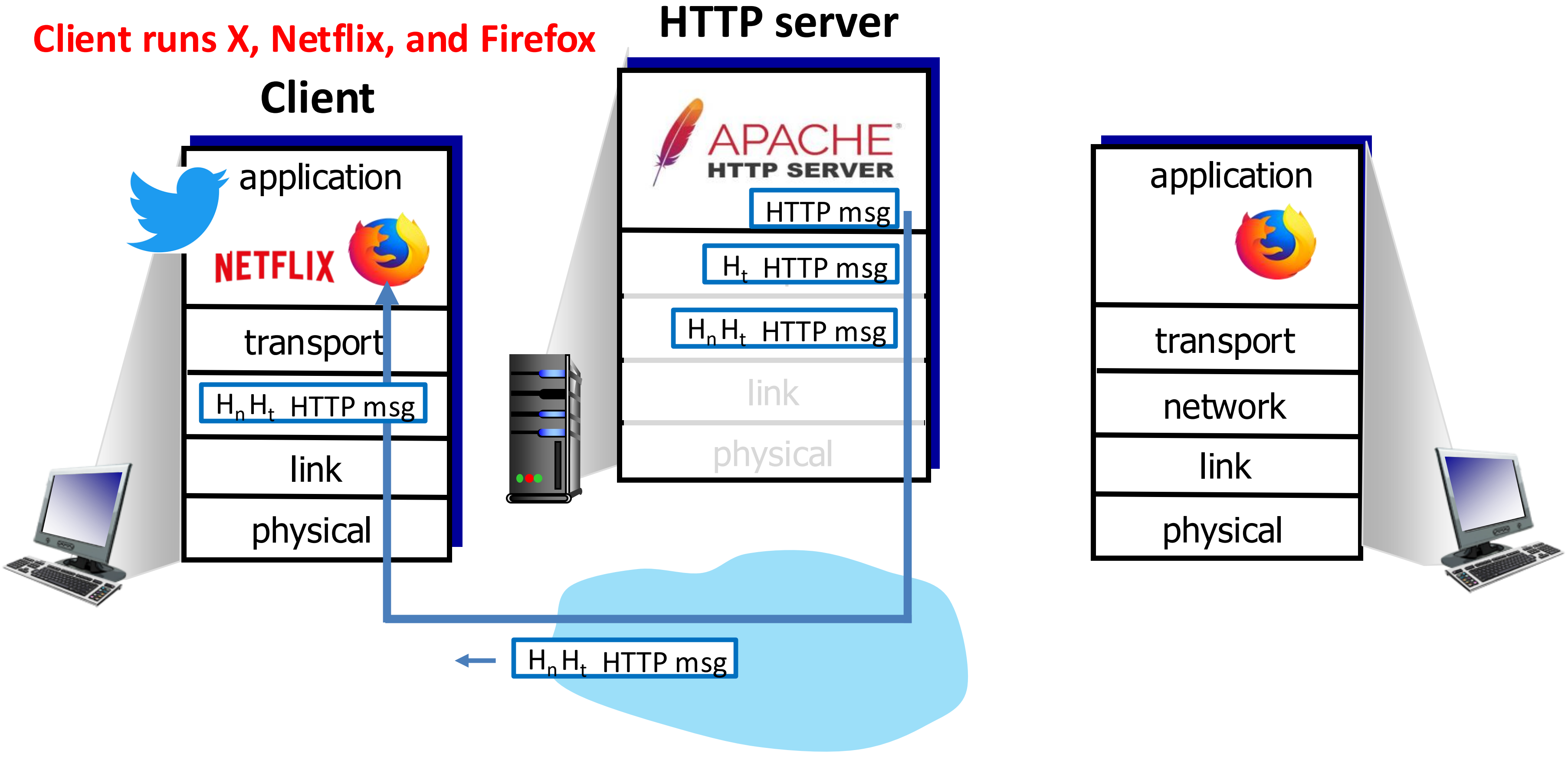
4242 Source port #	19,157 Dest. port #
Other header fields	
Application data (message)	

Choosing which Socket to use to receive is called *demultiplexing*

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Example: Client runs three applications

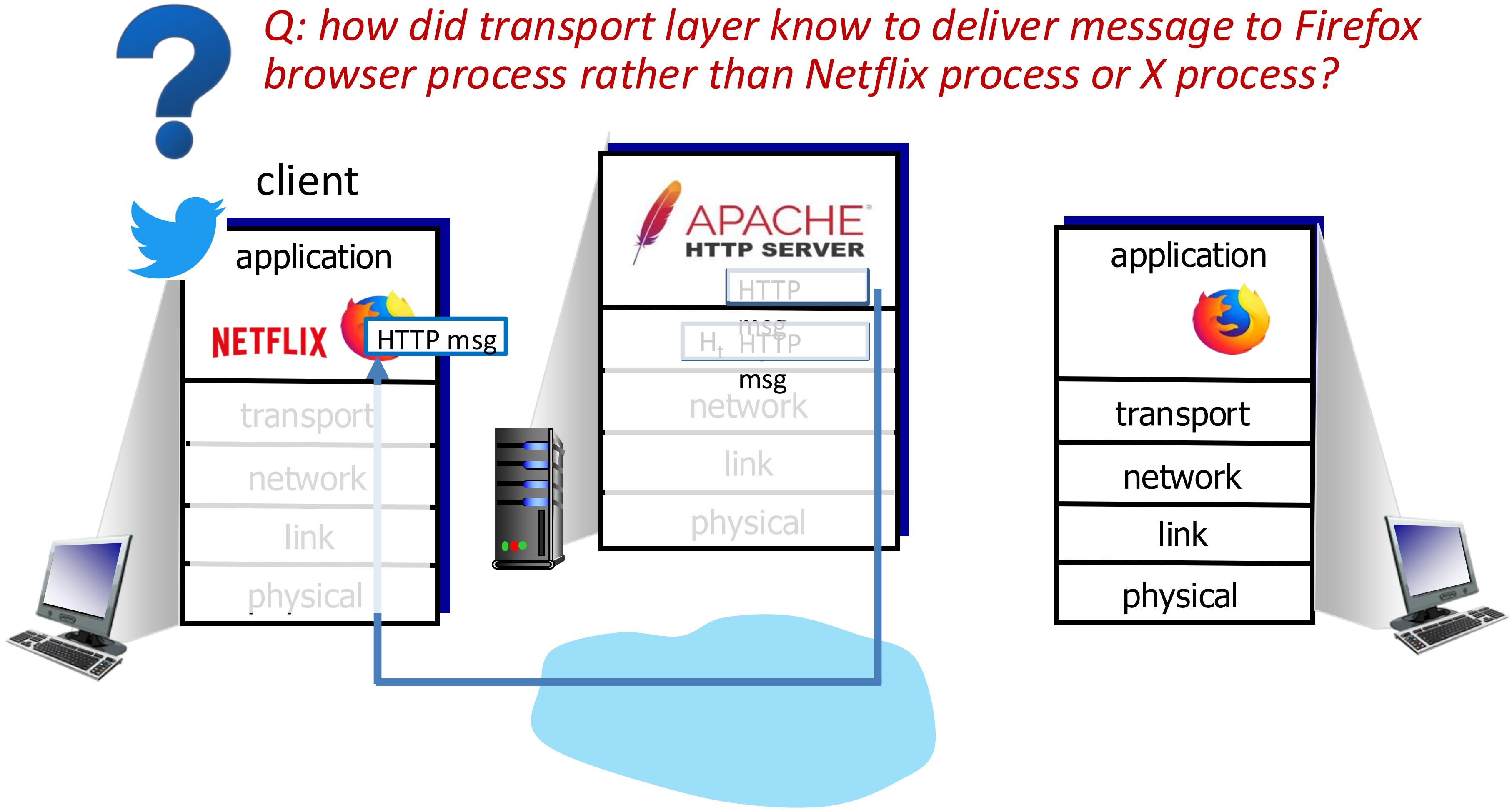
Client runs X, Netflix, and Firefox



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

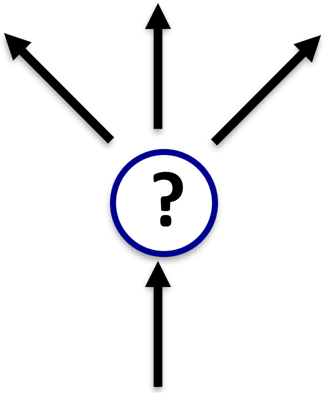
Example: Client runs three applications

Q: how did transport layer know to deliver message to Firefox browser process rather than Netflix process or X process?



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

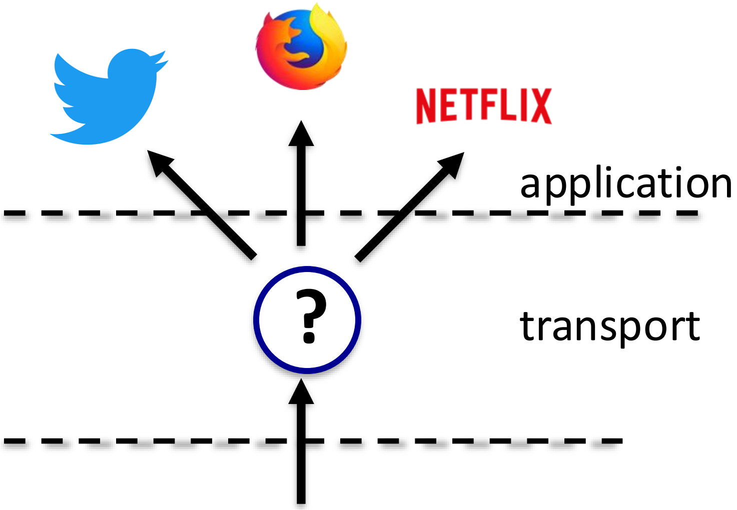
Demultiplexing: sends received Segment to the proper application



de-multiplexing

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Demultiplexing: sends received Segment to the proper application



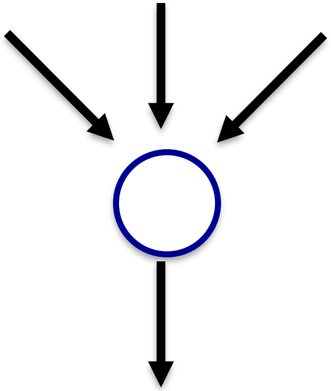
de-multiplexing

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Demultiplexing: sends received Segment to the proper application

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

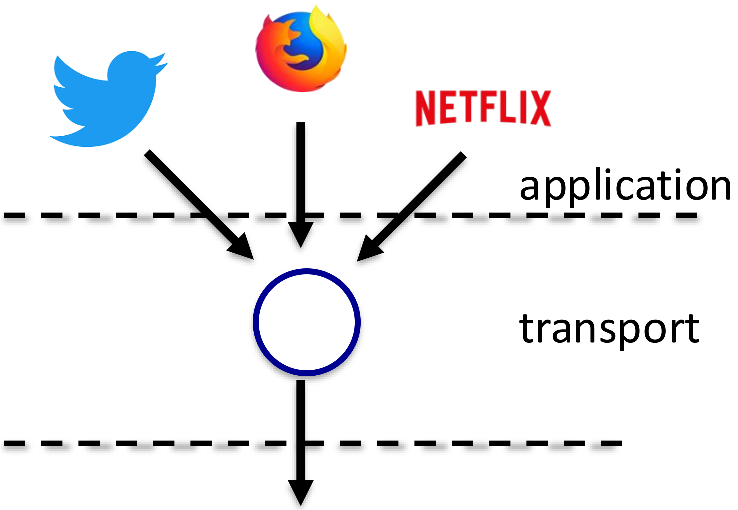
Multiplexing: sends Segments out the proper Socket to the Transport Layer



multiplexing

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Multiplexing: sends Segments out the proper Socket to the Transport Layer



multiplexing

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

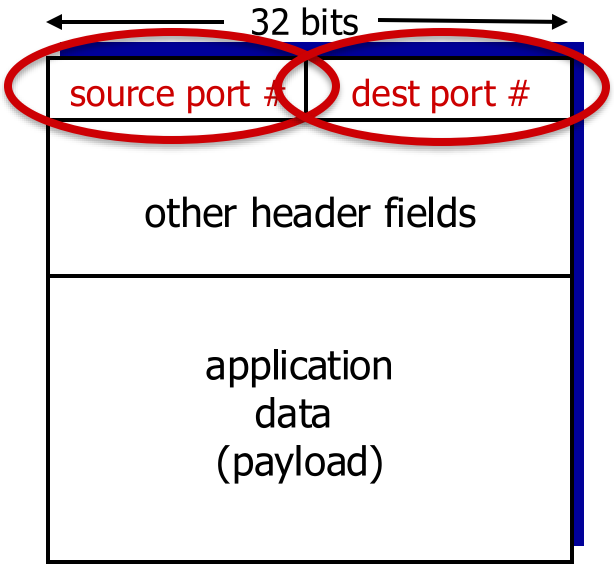
Multiplexing: sends Segments out the proper Socket to the Transport Layer

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

How demultiplexing works

- Host receives Layer 3 IP datagrams
  - Each Layer 3 datagram has source IP address, destination IP address
  - If destination IP is this host, send to Layer 4
- Each Layer 3 datagram carries one Layer 4 transport-layer Segment as its payload
  - Each Segment has source and destination port numbers

- Host uses *IP addresses & port numbers* to direct segment to appropriate Socket



TCP/UDP segment format

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP connectionless demultiplexing

Recall:

- When creating Socket to <u>*listen*</u> for connections, must specify *source* port number:

```
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_DGRAM)

sock.bind((ip_addr,9090))
```

Note: if not specified with bind, OS chooses random, unused source port

- When creating datagram to <u>*send*</u> into UDP Socket, must specify destination IP address and port number

```
sock = socket.socket(socket.AF_INET,
                     socket.SOCK_DGRAM)
sock.sendto(msg,(dest_addr,9090))
```

When receiving host receives *UDP* segment:

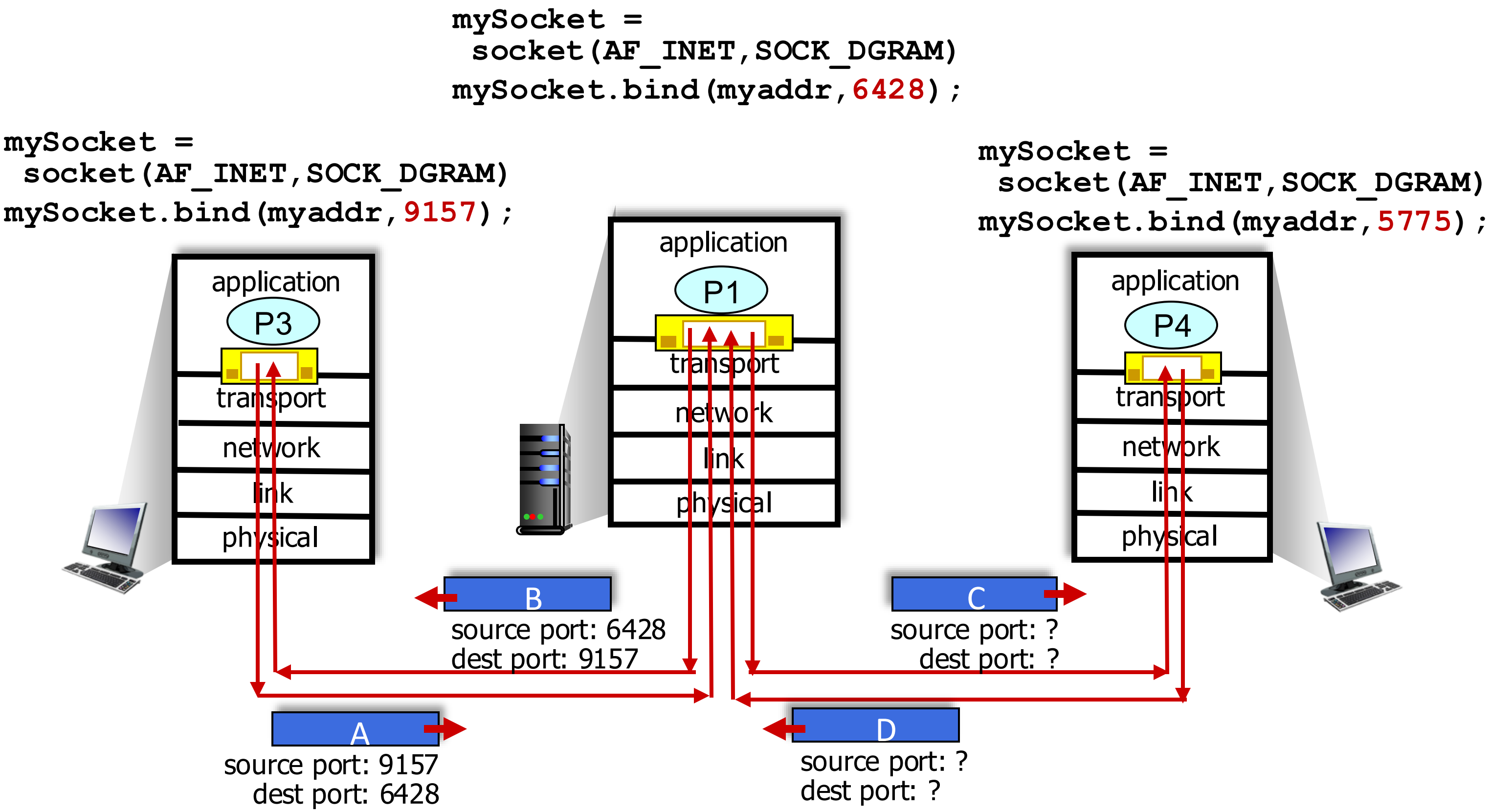
- Checks destination port number in Segment
- Directs UDP segment to Socket with that port number

↓

UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP connectionless demultiplexing example



```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,6428);
```

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,9157);
```

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,5775);
```

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

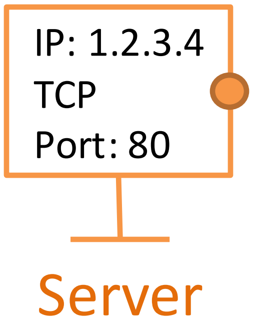
TCP connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

- Server may support many simultaneous TCP sockets:
  - Each socket identified by its own 4-tuple
  - Each socket associated with a different connecting client

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

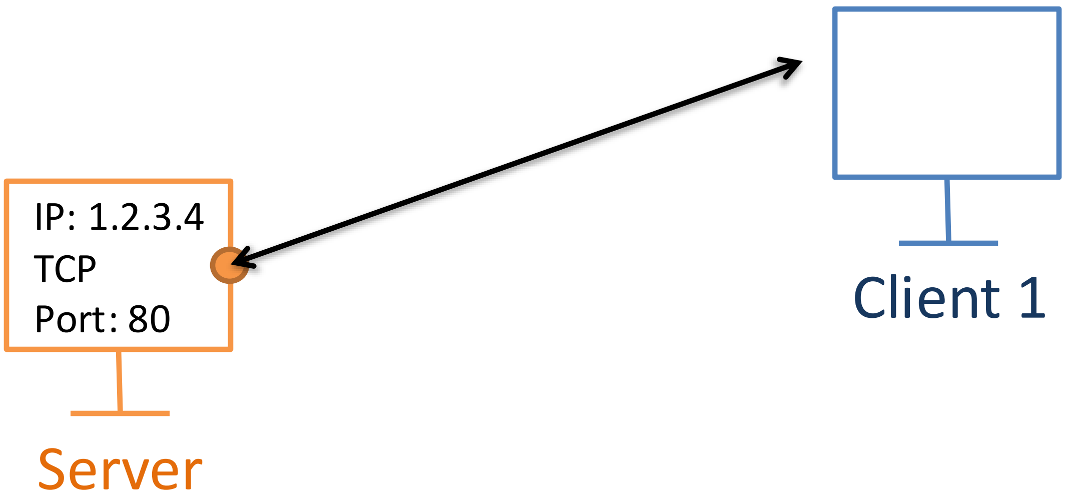
CS10 review: Servers often listen on a known port, create new sockets for clients



Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

CS10 review: Servers often listen on a known port, create new sockets for clients

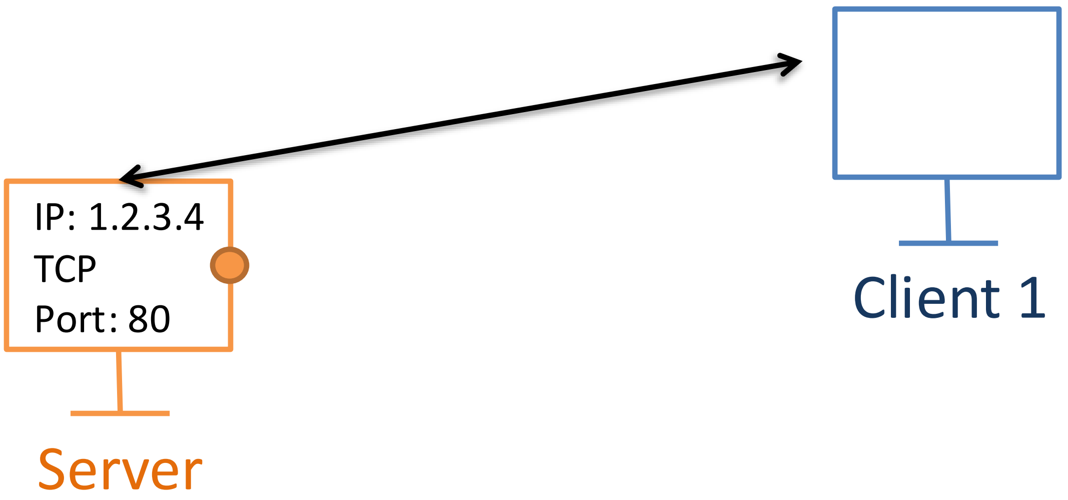


- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket

Server is listening on a socket
(socket = address
            + protocol
            + port)

Port 80 = HTTP

CS10 review: Servers often listen on a known port, create new sockets for clients

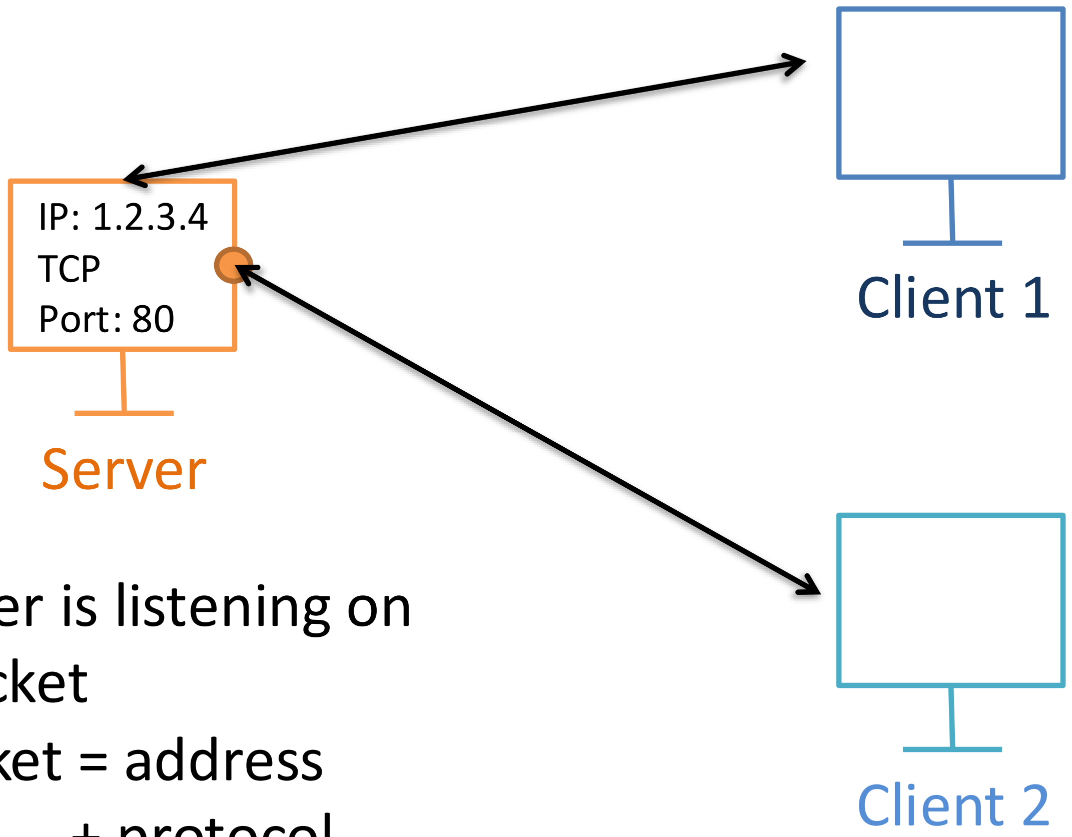


Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

- Client 1 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening
- Server talking to Client 1 and ready for others

CS10 review: Servers often listen on a known port, create new sockets for clients

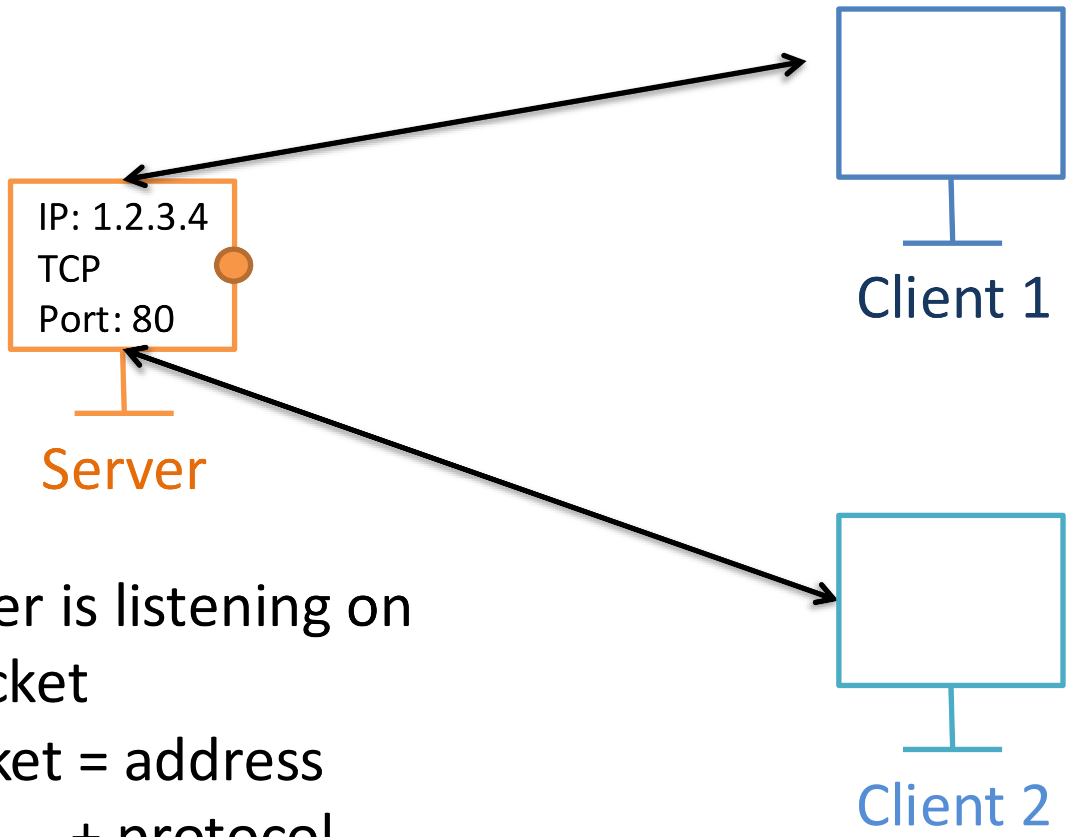


- Client 2 makes connection over socket

Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

CS10 review: Servers often listen on a known port, create new sockets for clients



Server is listening on a socket
(socket = address
+ protocol
+ port)

Port 80 = HTTP

- Client 2 makes connection over socket
- Server receives connection, moves communications to own socket
- Server returns to listening
- Server talking to client 1 and 2 ready for others

Single client TCP *server* example (non-multithreaded)

receive_tcp.py

```
#define host ip and port
HOST = ‘0.0.0.0’ #any localhost interface
PORT = 9090

#create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))

#listen for connections
server_socket.listen()
print(f"Server listening on {HOST}:{PORT}...")
try:
   client_socket, addr = server_socket.accept()
   print(f"Connected by {addr}")
   while True:
      data = client_socket.recv(1024)
      if not data:
         break
      print(f"Received: {data.decode()}")
      client_socket.sendall(b"Message received: " + data)
finally:
   client_socket.close()
   server_socket.close()
```

Server is running any localhost interface
Will listen on port 9090

SOCK_STREAM is a TCP socket
SOCK_DGRAM for UDP

Server **binds to IP address and port**
Client **typically has random port**

Listen for TCP connections to port 9090

accept() **blocks until client connects**
Returns a socket just for that client
What does UDP do for accept?

Receive data from client
Print to console (break if client hangs up)

Send reply back to client over *client_socket*

Sendall sends multiple packets if needed

Close both sockets when done

```
$ python3 receive_tcp.py
Message received: hello
Message received: another
message
```

```
$ nc -t localhost 9090
hello
Message received: hello
another message
Message received: another message
```

Single client TCP *server* example (non-multithreaded)

receive_tcp.py

```
#define host ip and port
HOST = ‘0.0.0.0’ #any localhost interface
PORT = 9090

#create TCP socket listening on ip address and port
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))

#listen for connections
server_socket.listen()
print(f"Server listening on {HOST}:{PORT}...")
try:
    client_socket, addr = server_socket.accept()
    print(f"Connected by {addr}")
    while True:
        data = client_socket.recv(1024)
        if not data:
            break
        print(f"Received: {data.decode()}")
        client_socket.sendall(b"Message received: " + data)
finally:
    client_socket.close()
    server_socket.close()
```

TCP server socket comprises

1. **Source IP (localhost here)**
2. **Source port (9090 here)**
3. **Destination IP (client IP)**
4. **Destination port (client port)**

All four used for demux’ing when using TCP

If this were a multithreaded example, we would need all four to identify a socket because each client (with own ip and port) would get its own socket from *socket.accept()*!

Single *client* TCP Server example

send_tcp.py

```
#define server's ip and port
HOST = '127.0.0.1'
PORT = 9090
```

Server is running on localhost and listening on port 9090

SOCK_STREAM is a TCP socket
SOCK_DGRAM for UDP

```
#create TCP socket to connect to server
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect((HOST, PORT))
```

Connect to server on ip and port using TCP

```
try:
   message = "Hello, Server!"
   client_socket.sendall(message.encode()) #encode converts to bytes
   data = client_socket.recv(1024) #get server's response
   print(f"Received from server: {data.decode()}")
finally:
   client_socket.close()
```

Send message over socket
***sendall()* sends all chunks (*send()* only one)**

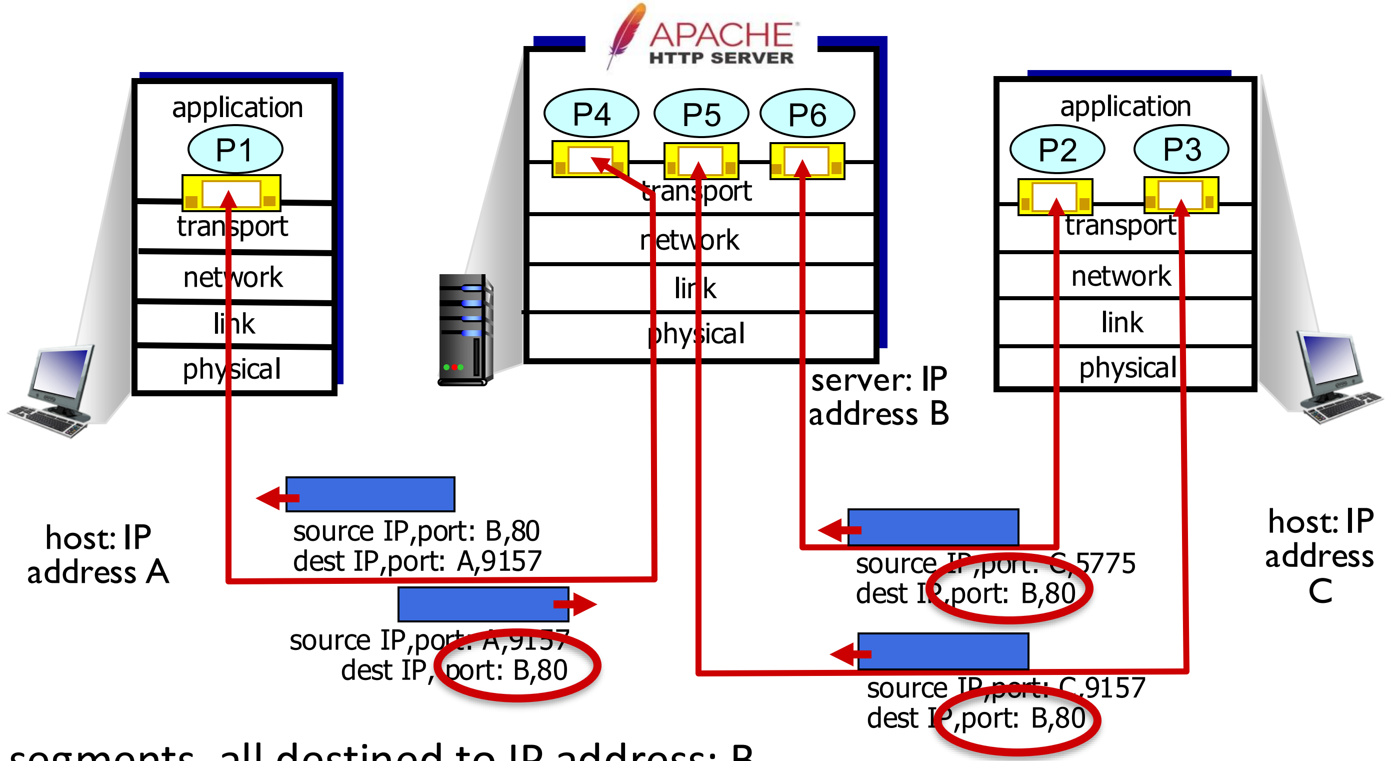
Get and print server's response

Close socket

```
$ python3 send_tcp.py
Received from server: Message received: Hello, Server!
```

```
$ python3 receive_tcp.py
Message received: Hello,
Server!
```

TCP connection-oriented demultiplexing example



Three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets on Host B based on 4-tuple

- src IP
- src port
- dst IP
- dst port

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers

Agenda

1. Transport (Layer 4) services
2. Multiplexing and Demultiplexing: How does data get to the right application?
3. Connectionless Transport: UDP

We will focus on TCP next class

4. Exercises

Why UDP: User Datagram Protocol

- "No frills," "bare bones" Internet transport protocol
- "Best effort" service, UDP segments may be:
  - Lost
  - Delivered out-of-order to app
- *Connectionless:*
  - No handshaking between UDP sender, receiver (so it is fast)
  - Each UDP segment handled independently of others

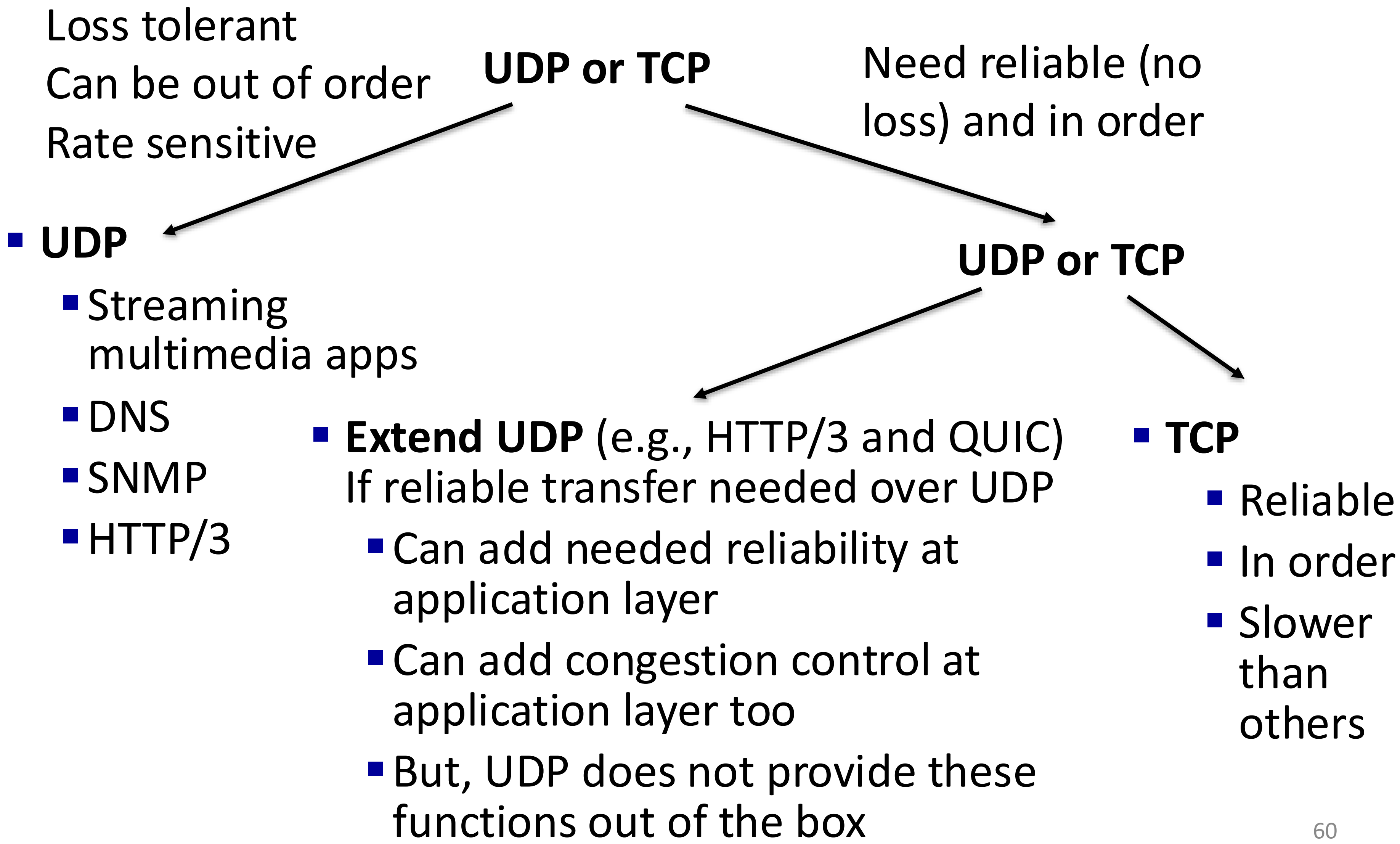
Other than providing Mux/Demux services to get data to the proper application, UDP is essentially Layer 3 (Network Layer)

Why is there a UDP?

- No connection establishment (which can add RTT delay)
- Simple: no connection state at sender, receiver
- Small header size (8 bytes vs 20 bytes for TCP)
- No congestion control
  - UDP can blast away as fast as desired!
  - Can function in the face of congestion

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Use UDP when it is not critical all Segments are delivered (or delivered in order) quickly



UDP or TCP

Loss tolerant

Can be out of order

Rate sensitive

- **UDP**
  - Streaming multimedia apps
  - DNS
  - SNMP
  - HTTP/3

Need reliable (no loss) and in order

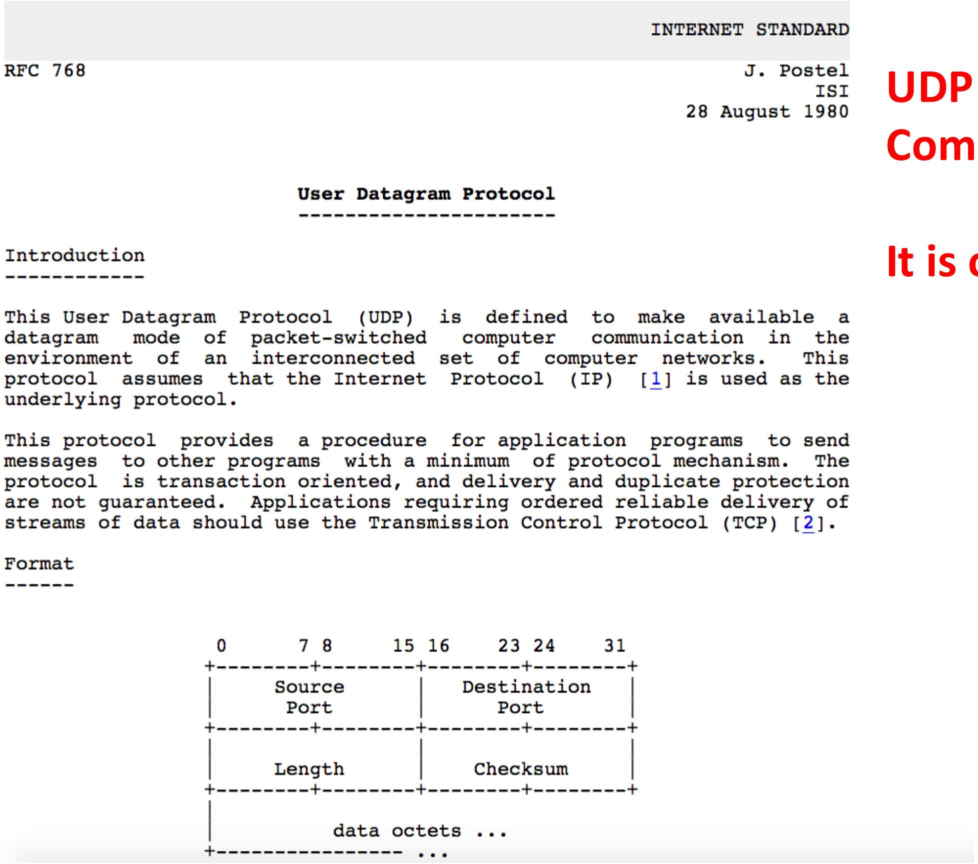
UDP or TCP

- **Extend UDP** (e.g., HTTP/3 and QUIC)
  If reliable transfer needed over UDP
  - Can add needed reliability at application layer
  - Can add congestion control at application layer too
  - But, UDP does not provide these functions out of the box

- **TCP**
  - Reliable
  - In order
  - Slower than others

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP: User Datagram Protocol RFC is simple



INTERNET STANDARD

```
RFC 768                                                        J. Postel
                                                                     ISI
                                                          28 August 1980
```

User Datagram Protocol

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format

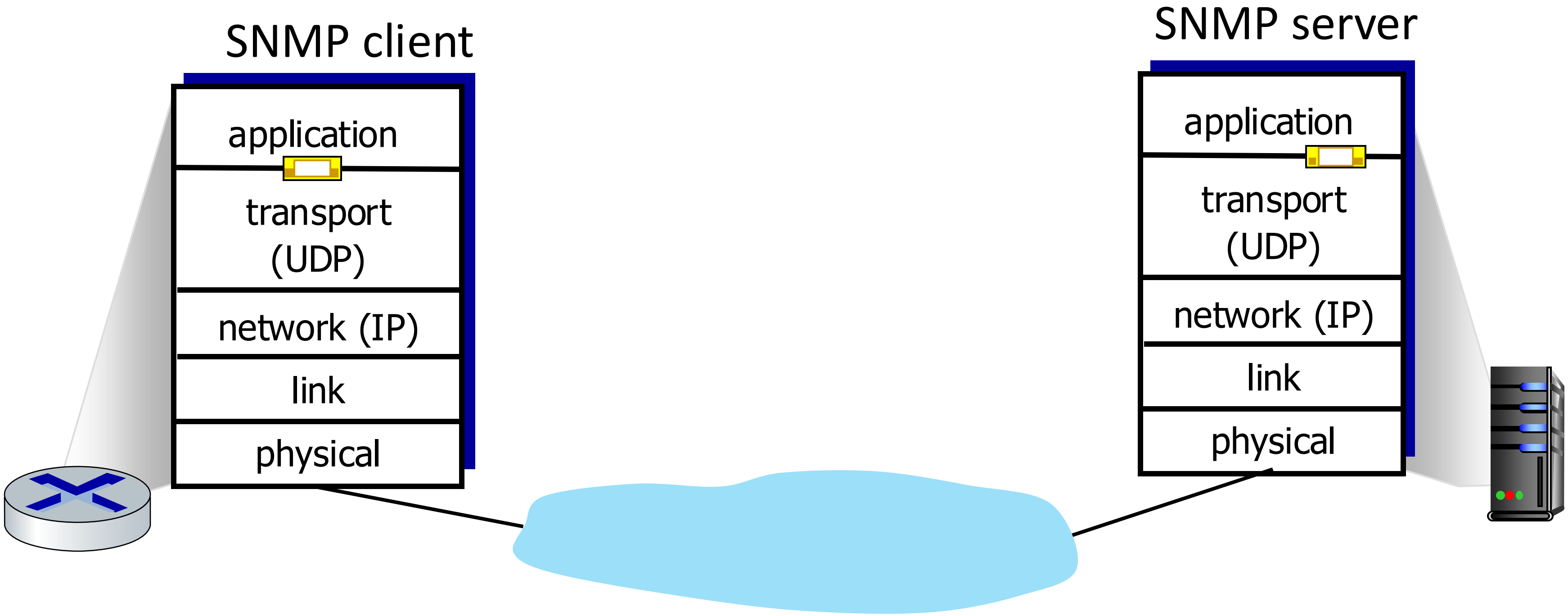
```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

UDP is defined in Request For Comment (RFC) 768

It is only 3 pages long!

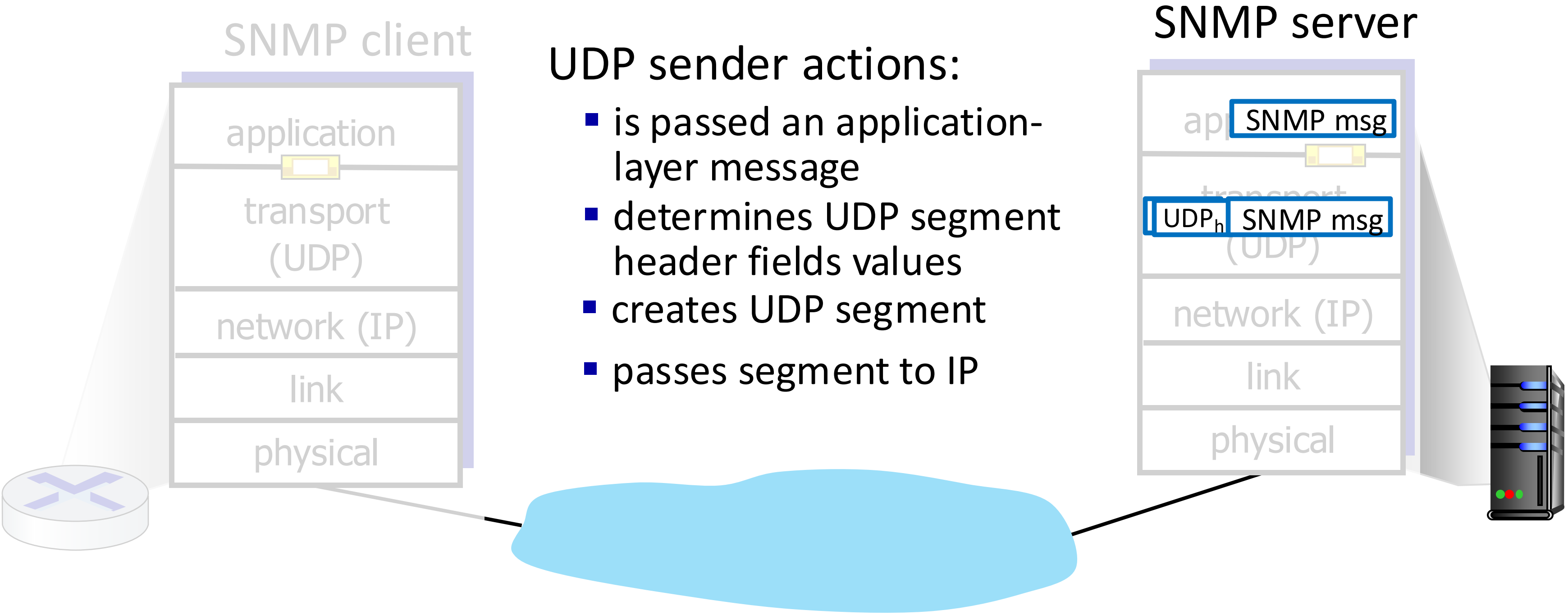
https://datatracker.ietf.org/doc/html/rfc768

UDP transport layer actions



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP transport layer actions

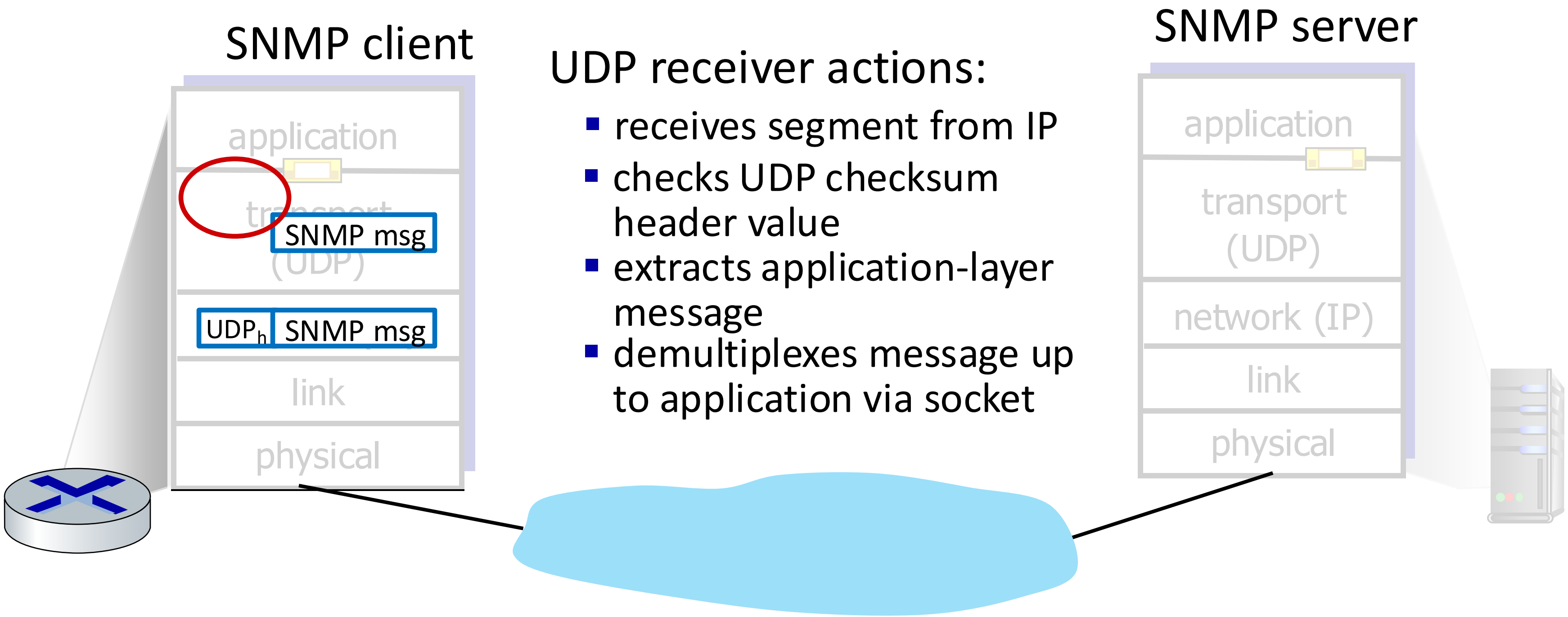


UDP sender actions:

- is passed an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP transport layer actions



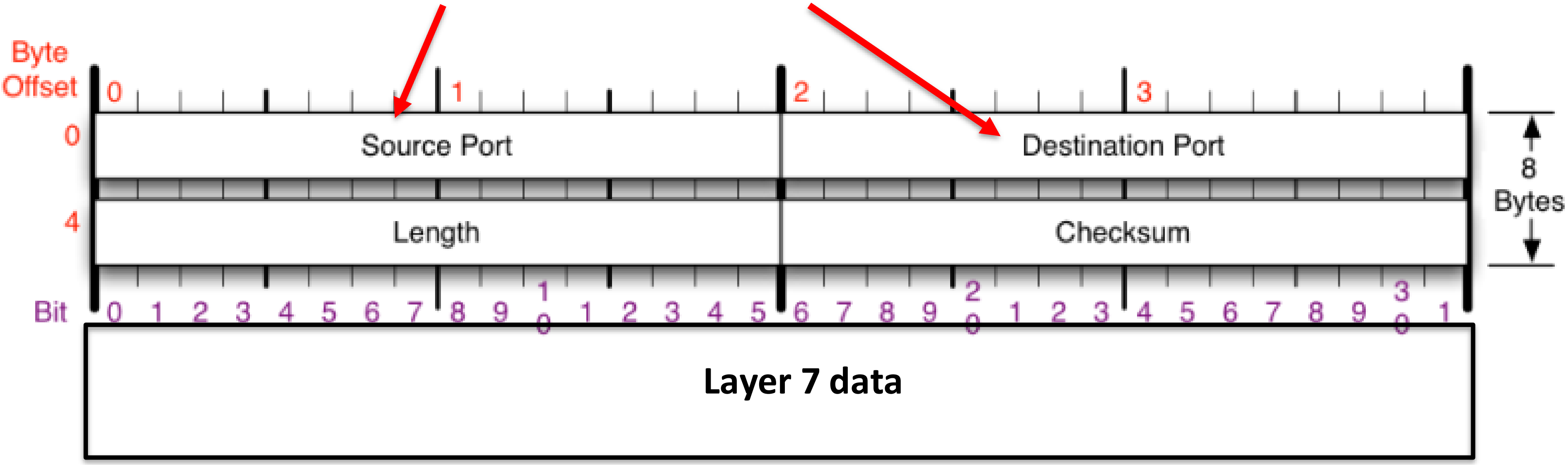
UDP receiver actions:

- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

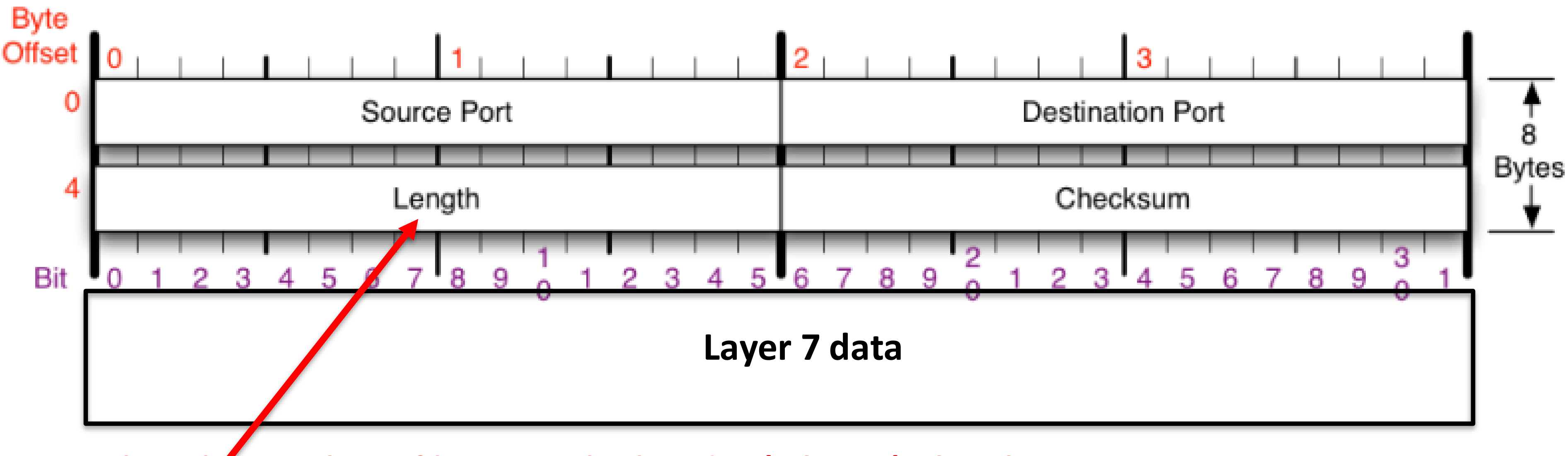
UDP Header is only 32 bits (8 bytes) long; port numbers connect applications

Port numbers range between 0 and 65,535 (=2^{16} bits =2 bytes)



https://nmap.org/book/tcpip-ref.html

Length is the number of bytes in the header plus the Layer 7 data



Length is the number of bytes in the header (8 bytes) plus the number of bytes in the Layer 7 data

So, if we send 20 bytes of Application Layer 7 data using UDP, the length field is set to:

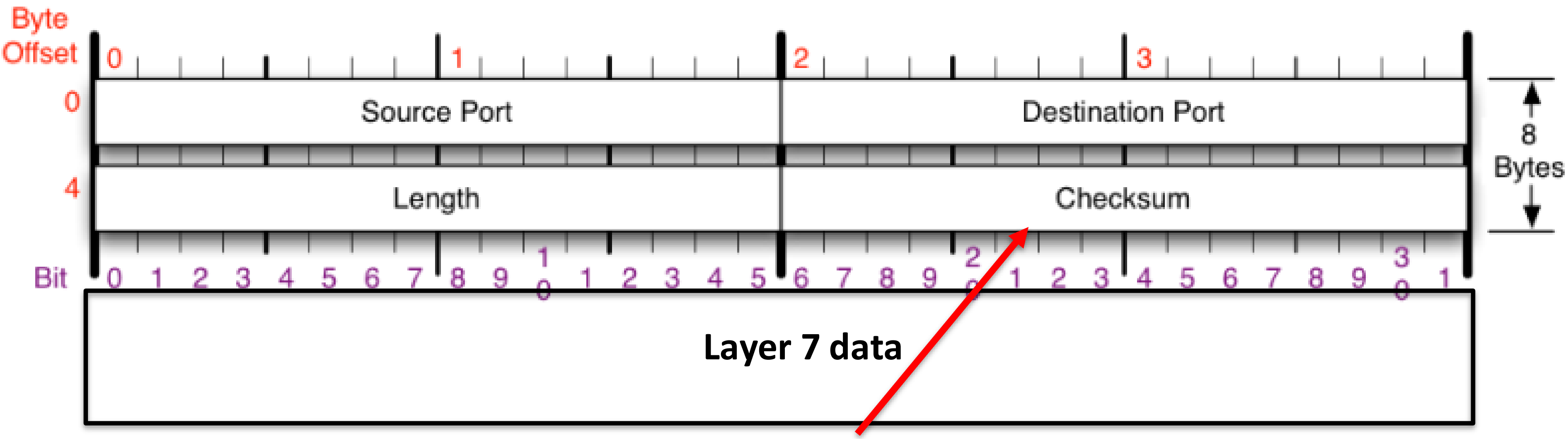
8 header bytes + 20 data bytes = 28

Theoretical max UDP Segment is 2^{16}-1 = 65,535 bytes

Subtract 8 bytes for header = 65,527 bytes of data

https://nmap.org/book/tcpip-ref.html

Checksum does error detection



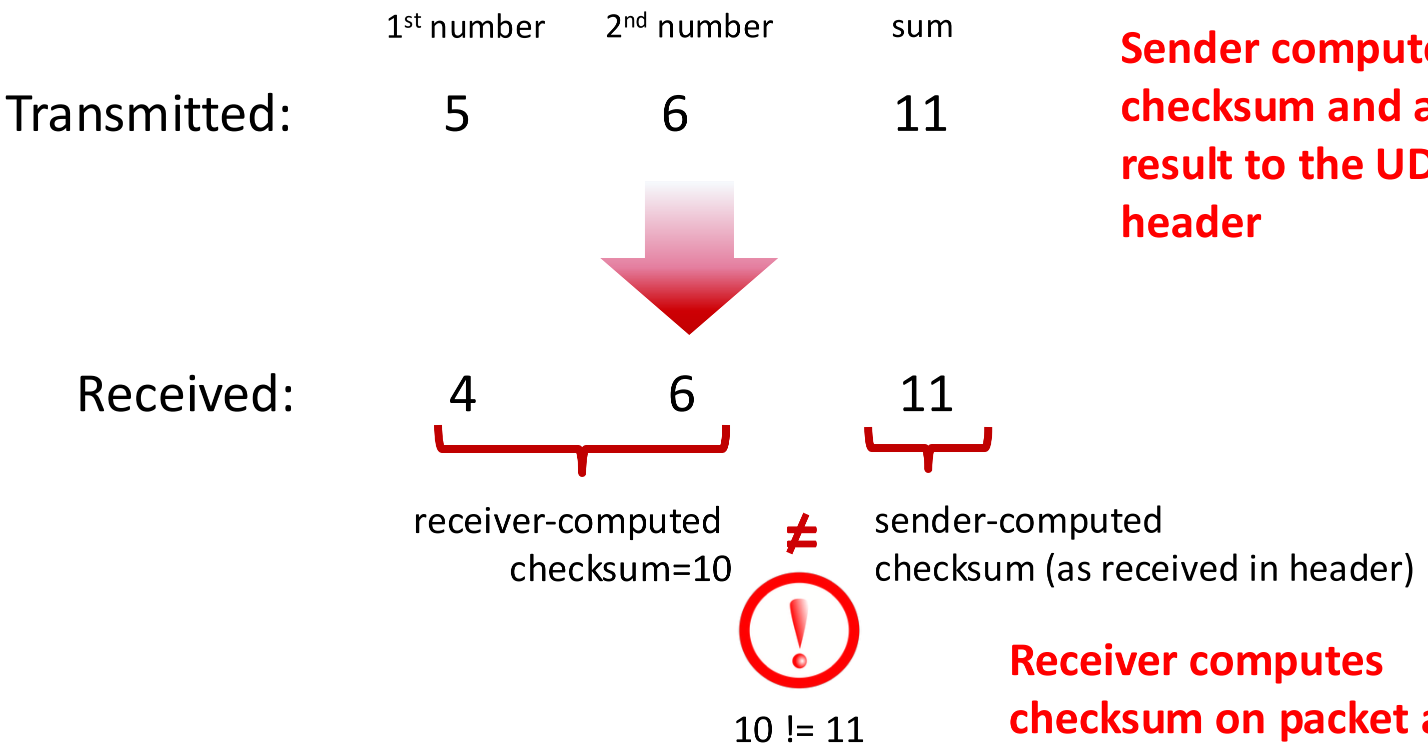
Checksum does error detection (but not error correction!)

Detects (some of the time) whether bits have been altered in transit (for example: by noisy links or while stored in routers) as the Segment moved from source to destination

https://nmap.org/book/tcpip-ref.html

UDP checksum helps detect bit errors

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment



	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum=10 ≠ sender-computed checksum (as received in header)

10 != 11

Sender computes checksum and adds result to the UDP packet header

Receiver computes checksum on packet and compares with checksum stored in header

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Internet checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment

Sender:

- Treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- Checksum: bit addition (one's complement sum) of segment content
- Checksum value put into UDP checksum field

Receiver:

- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ....

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Sender calculates the checksum of the UDP Segment and includes it in the header

Review

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 (carry 1)

1 + 1

Carry

Example: add two 16-bit integers

```
                                      1
     1  1  1  0  0  1  1  0  0  1  1  0  0  1  1  0
     1  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
  -------------------------------------------------
  1  1  0  1  1  1  0  1  1  1  0  1  1  1  0  1  1
```

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Sender calculates the checksum of the UDP Segment and includes it in the header

Review
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 (carry 1)

Example: add two 16-bit integers

```
                 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
                 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
              -----------------------------------
wraparound     1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
              -----------------------------------
       sum       1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
```

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Sender calculates the checksum of the UDP Segment and includes it in the header

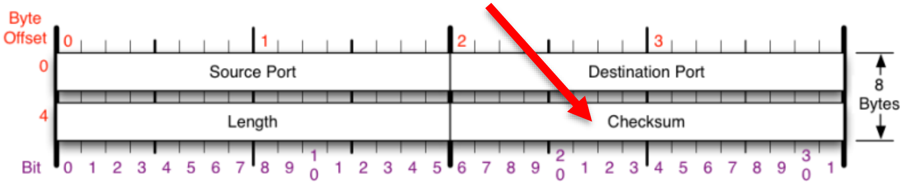
Example: add two 16-bit integers

```
                 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
                 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
               ---------------------------------
wraparound     1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
               ---------------------------------
sum              1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum         0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

Checksum is the 1's compliment of the sum (flip all bits)

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

Sender puts the checksum in the UDP header



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Receiver calculates the sum of data it received and adds sum to the checksum

Example: add two 16-bit integers

$$\begin{array}{r c cccccccccccccccc}
 & & 1&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0\\
 & & 1&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\ \hline
\text{wraparound} & 1 & 1&0&1&1&1&0&1&1&1&0&1&1&1&0&1&1\\ \hline
\text{sum} & & 1&0&1&1&1&0&1&1&1&0&1&1&1&1&0&0
\end{array}$$

Receiver calculates the sum the same as the sender from the data is received

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Result should be all 1’s, otherwise there is a bit error

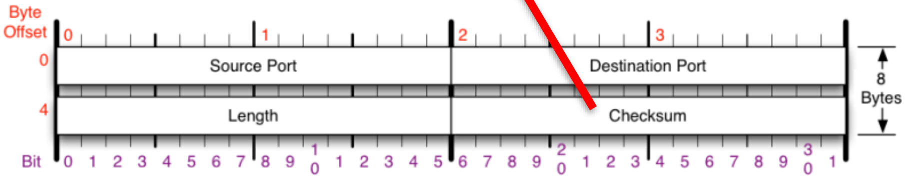
Example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Receiver calculates the sum the same as the sender from the data is received

Receiver adds the checksum from the sender in the UDP header

Result should be all 1’s if there are no bit errors



Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Result should be all 1’s, otherwise there is a bit error

One bit error in received data
(sender sent a 1 bit here)

Example: add two 16-bit integers

```
                1 1 1 0 0 0 1 0 0 1 1 0 0 1 1 1
                1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
              ---------------------------------
wraparound    1 1 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1
              ---------------------------------
sum             1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 0
```

Calculate sum

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Result should be all 1's, otherwise there is a bit error

One bit error in received data (sender sent a 1 bit here)

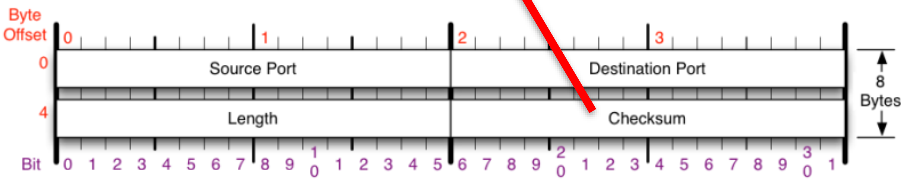
Example: add two 16-bit integers

		1	1	1	0	0	0	1	0	0	1	1	0	0	1	1	1
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	0	1	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	0	1	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1
		1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1

Calculate sum

Add checksum from the UDP header

Result is *not* all 1's
Indicates bit error

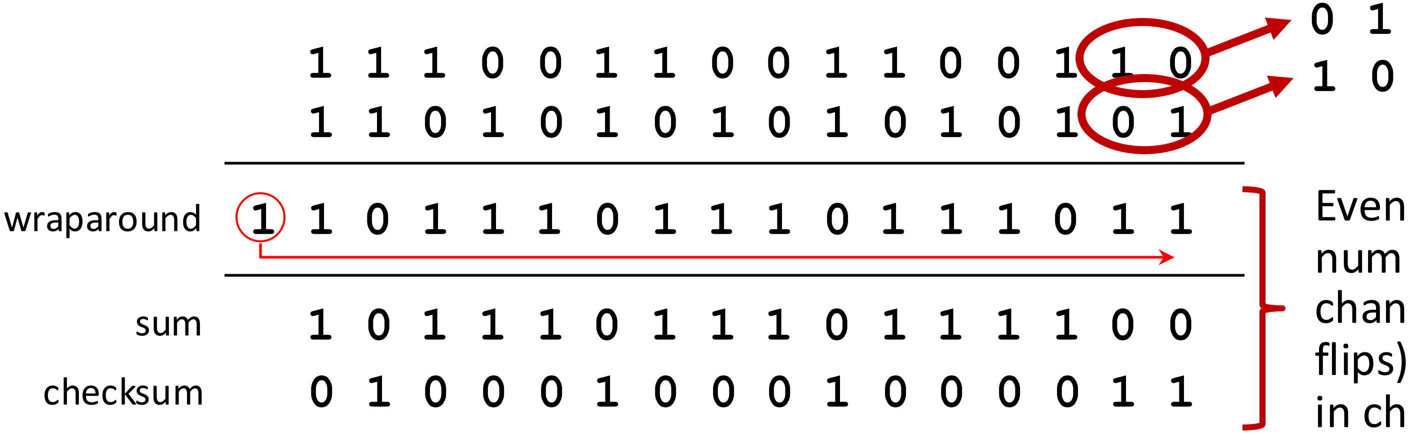


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Internet checksum has weak protection

Example: add two 16-bit integers

4 bit errors



	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
wraparound	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum	1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum	0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

Even though numbers have changed (bit flips), *no* change in checksum!

Checksum calculation may indicate no errors if there are an even number of bit errors

Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

UDP summary

- "No frills" protocol:
  - Segments may be lost, delivered out of order
  - Best effort service: "send and hope for the best"
- UDP has its plusses:
  - No setup/handshaking needed (no RTT incurred)
  - Can function when network service is compromised
  - Helps with reliability (checksum)
- Can build additional functionality on top of UDP in application layer (e.g., HTTP/3 or QUIC)

Agenda

1. Transport (Layer 4) services
2. Multiplexing and Demultiplexing: How does data get to the right application?
3. Connectionless Transport: UDP
4. Exercises

Exercise

Implement a chat program where:

- VPN to Dartmouth's network first (Dartmouth blocks traffic)!
- The client takes input from the keyboard and sends it to the server over TCP
- Server responds back to client with message converted to all uppercase
- Start with send_tcp.py/receive_tcp.py

Time permitting

- Implement the same in C
- Start with send_udp.c/receive_udp.c