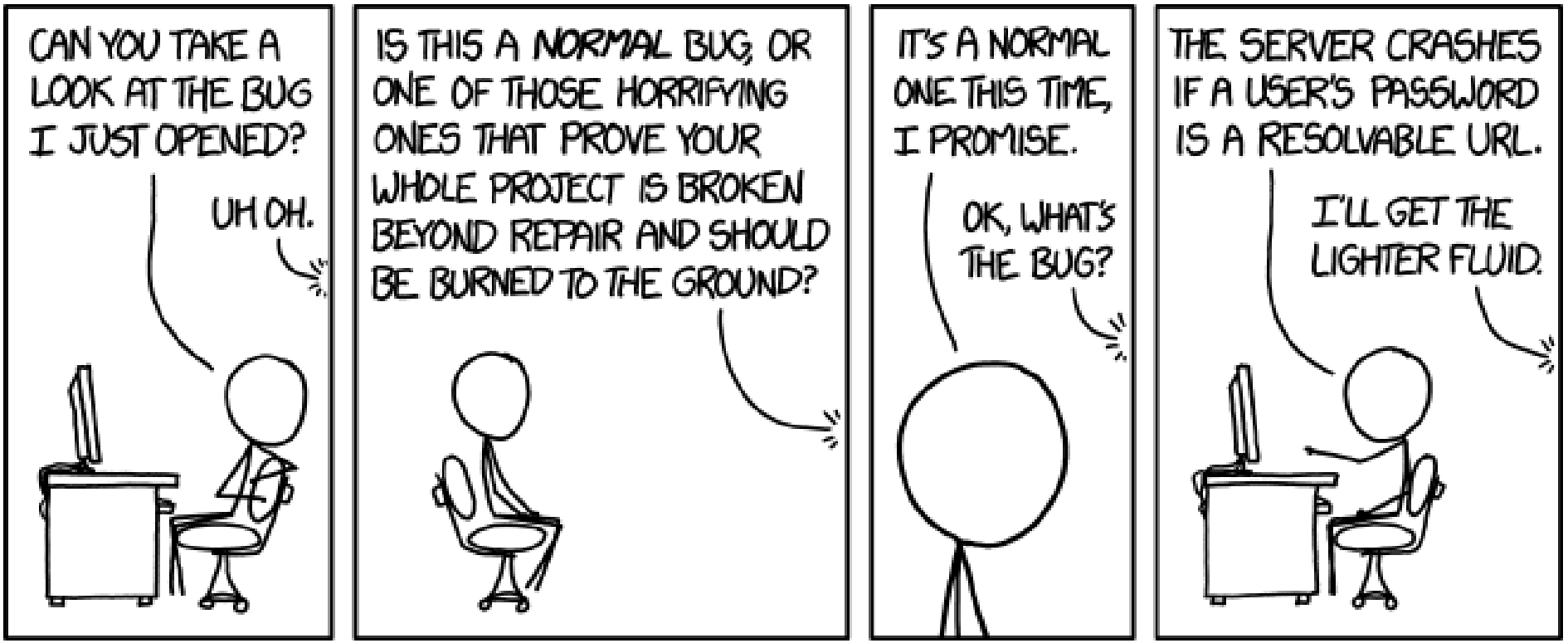


https://xkcd.com/1700

CS 60:
Computer Networks

Transport layer: 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 include:

- **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

Lab 2 looks at HTTP in detail

Web services define expected parameters and returned data in documentation (Flickr from CS10)

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 control
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



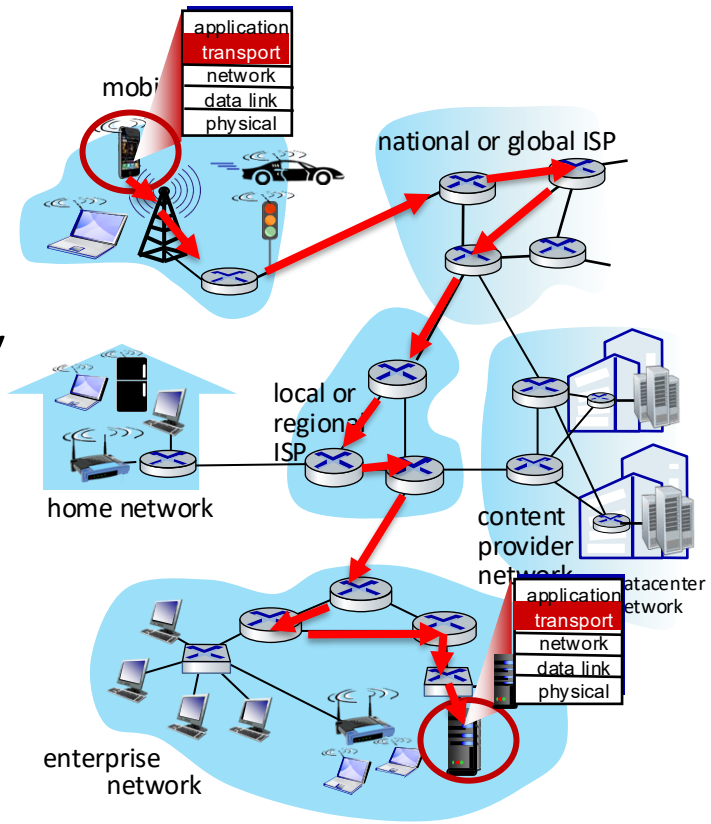
1. Transport (Layer 4) services
2. Multiplexing and Demultiplexing: How does data get to the right application?
3. UDP header bytes
4. UDP code

We will focus on TCP next class

5. Exercises

Transportation services provide logical rather than physical connections

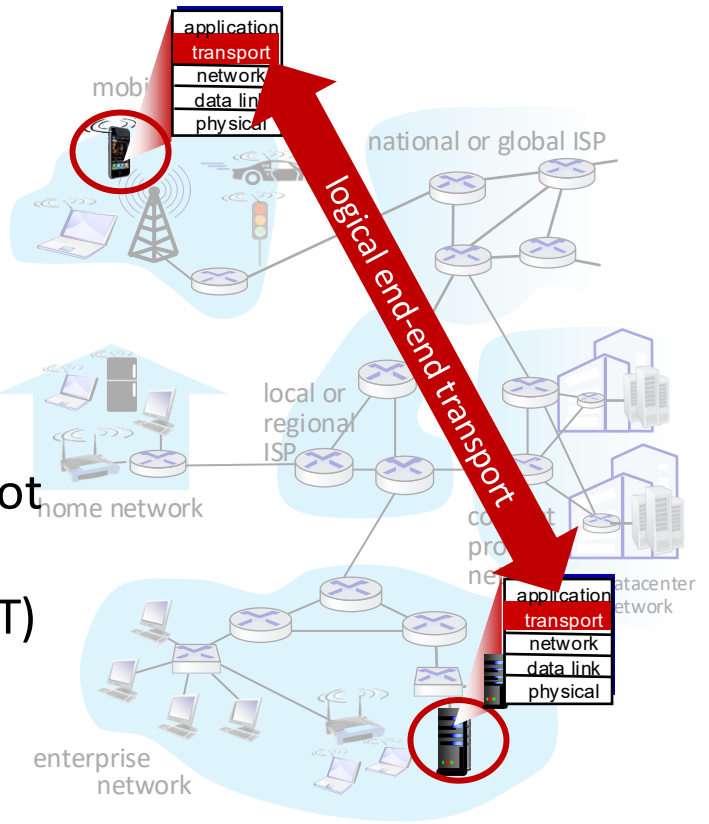
- Provide *logical communication* between application processes running on different hosts
  - From application's perspective, as if hosts were directly connected
  - Hosts may be far apart, connected by many routers and types of links
  - Applications do not need to worry about *how* data traversed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

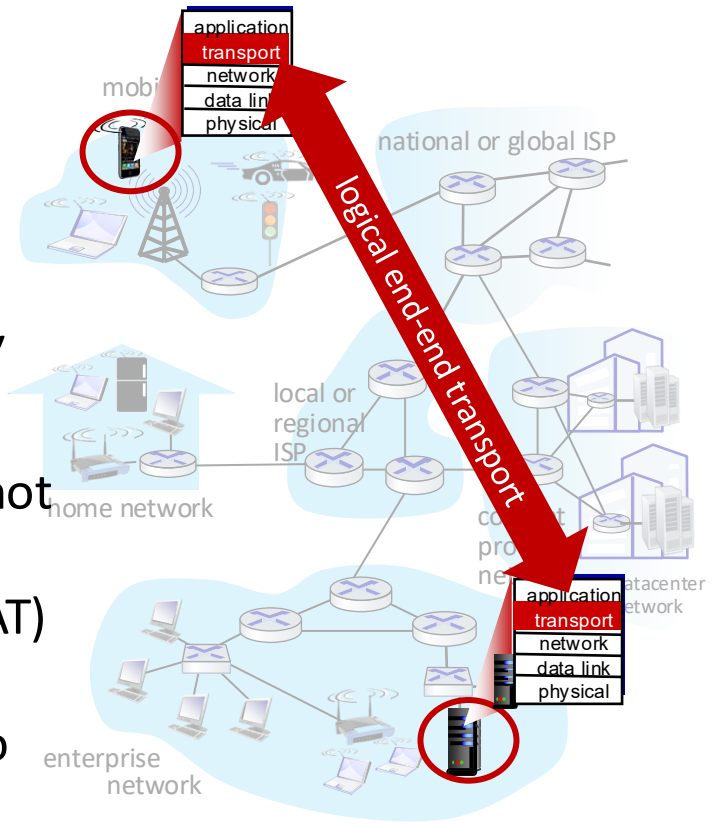


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

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 (except NAT)
  - 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



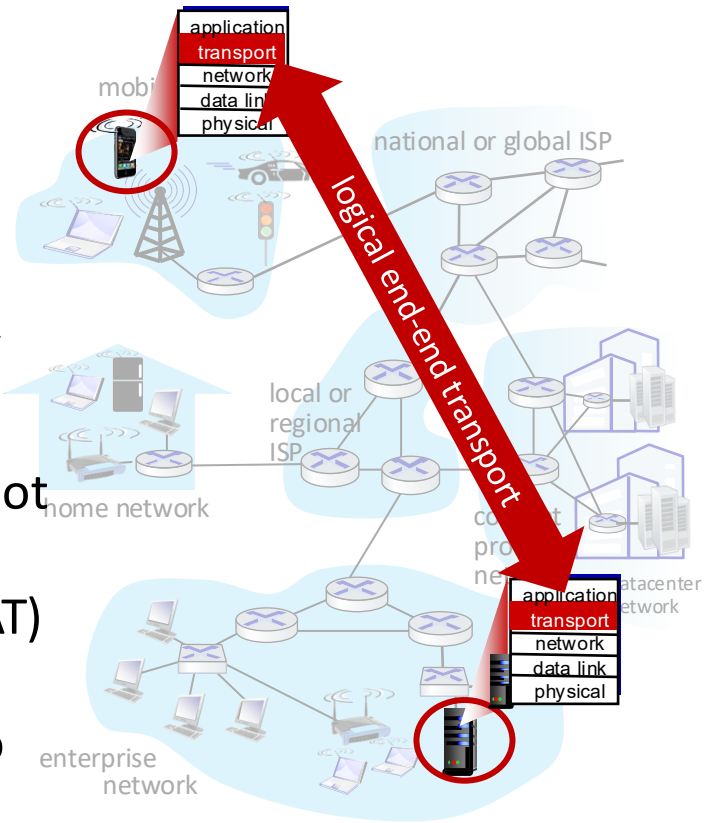
Routers determine path through network
Routers do not look at Transport Layer headers

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 (except NAT)
  - 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

Review: network layers

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	
3) Network (IP)	
2) Link (MAC)	
1) Physical	

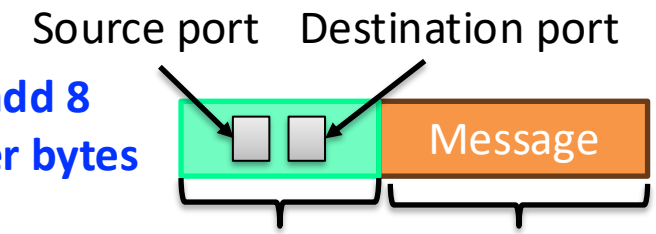


At Layer 4, payload is Layer 7 message

Review: network layers

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	TCP or UDP src and dst port
3) Network (IP)	
2) Link (MAC)	
1) Physical	

Today we add 8 UDP header bytes



Layer 4 uses ports to identify applications

- **A port is just an integer 0 – 65,535**

Server side:

- **Ports identify an application**
- **Different apps listen on different ports (e.g., web on port 80, telnet on port 23)**

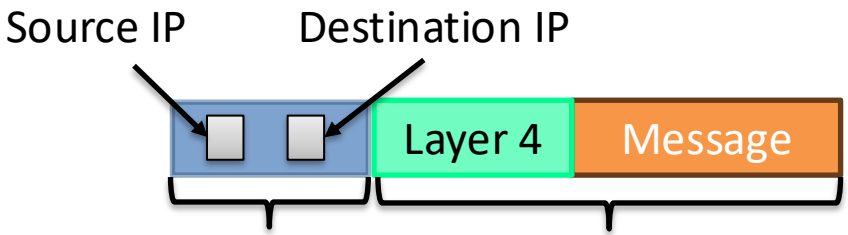
Client side:

- **Ports identify an instance of an application (might have multiple copies of an app running)**
- **Port numbers are randomly chosen by OS**

Key point: we are just adding some bytes into front of the message bytes (not anything more mysterious!)

Review: network layers

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	TCP or UDP src and dst port
3) Network (IP)	src and dst IP address **Routing!**
2) Link (MAC)	**Packets routed across LANs using IP addresses**
1) Physical	

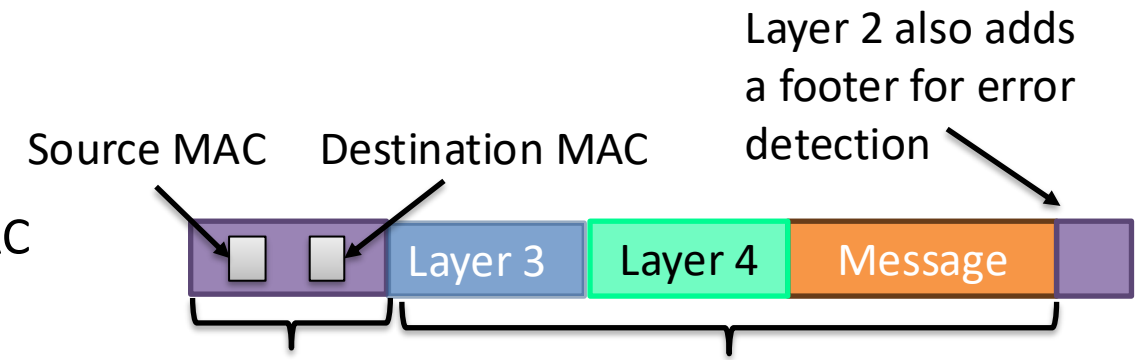


IP header for routing

At Layer 3, payload is Layer 4 datagram which is Layer 4 headers plus Layer 7 message

Review: network layers

Network model	Critical information
7) Application	Application message to be sent across network
4) Transport	TCP or UDP src and dst port
3) Network (IP)	src and dst IP address Routing!
2) Link (MAC)	src and dst MAC address **Switching!**
1) Physical	**Frames transmitted *within* a LAN based on MAC address**



Sender's data goes from Layer 7 to Layer 1

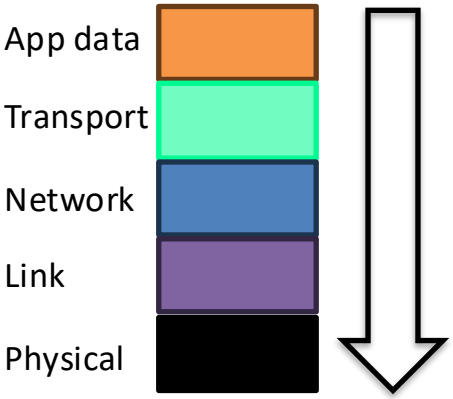
Client

Sending from Client to Server

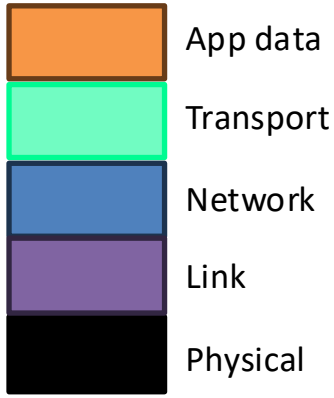
Server



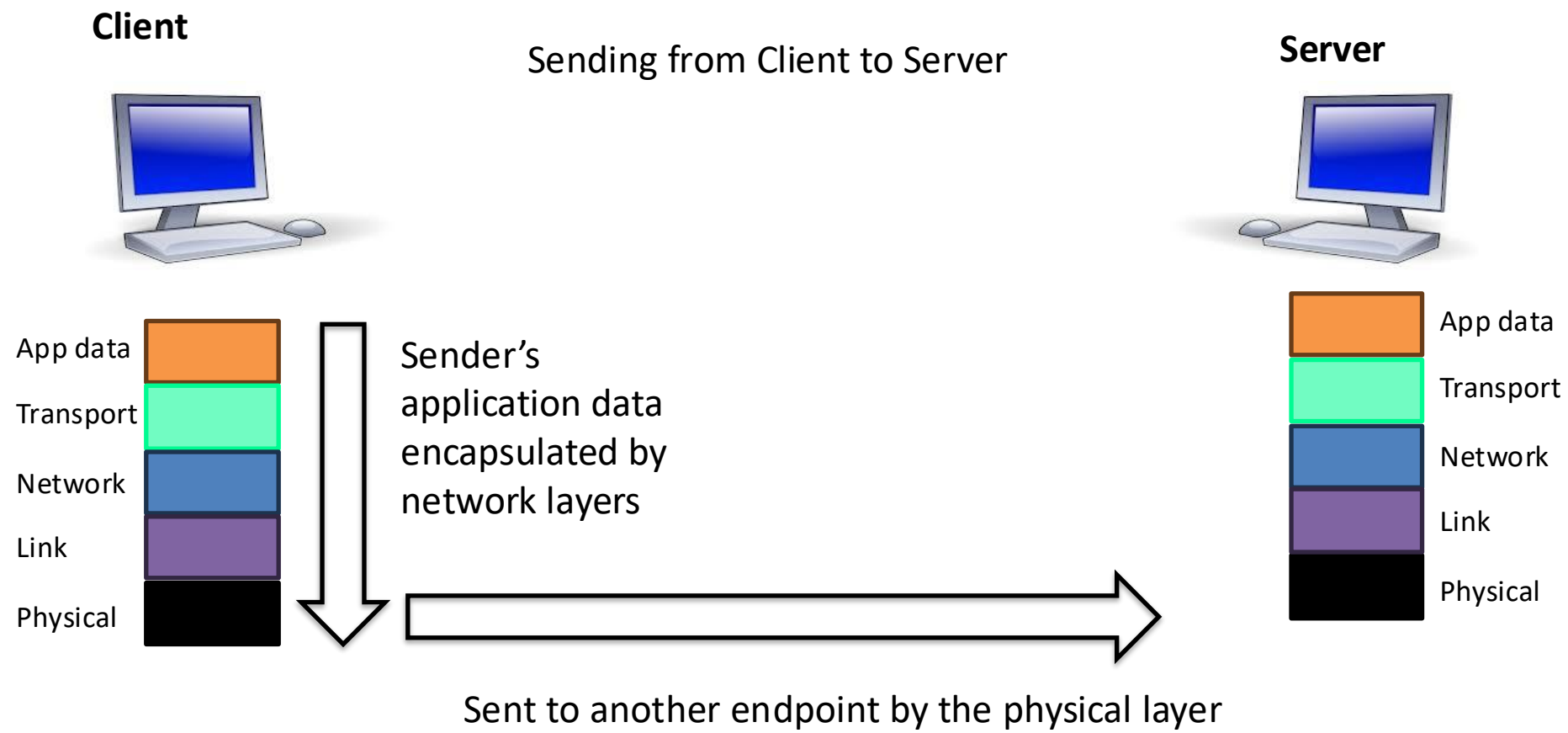




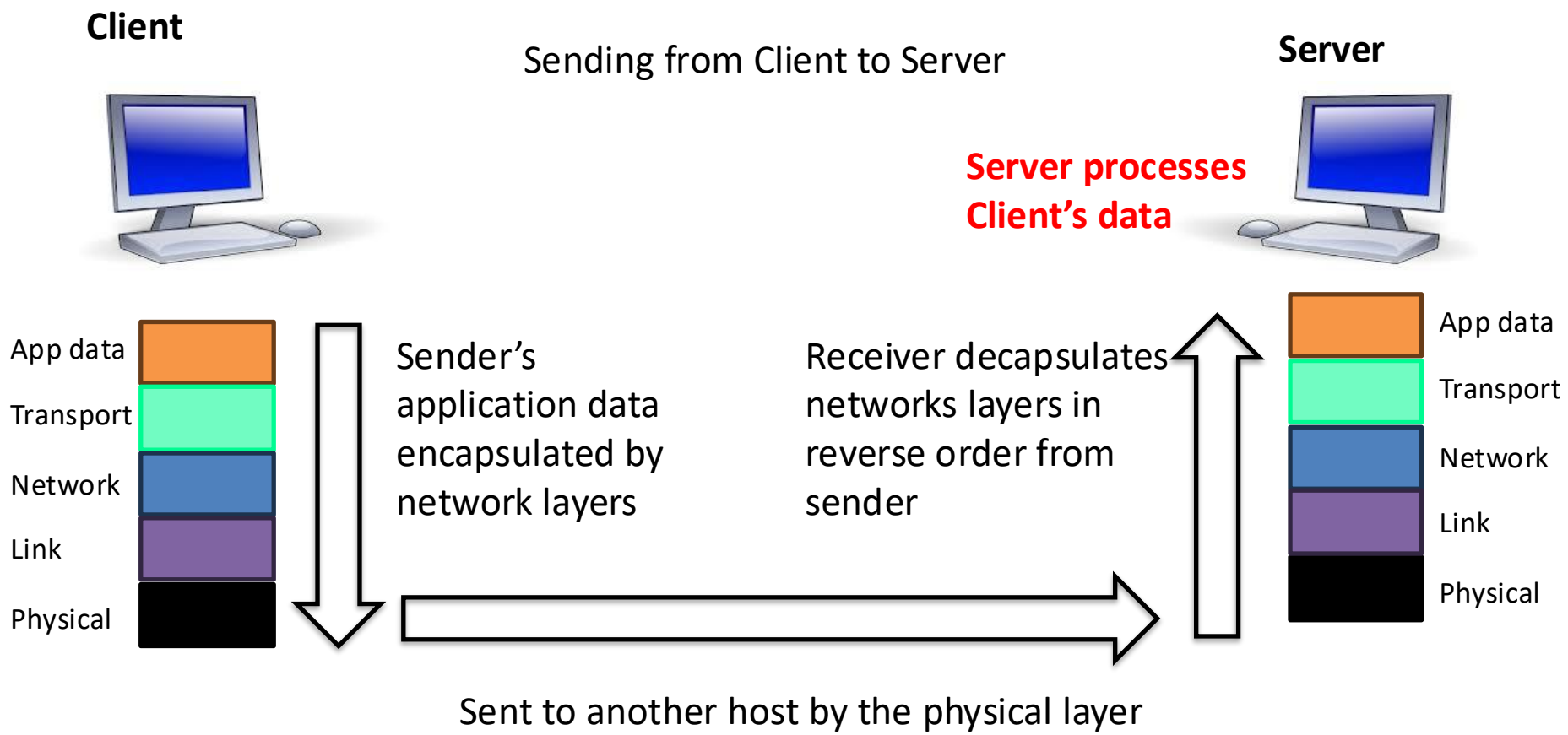
Sender's application data encapsulated by network layers



Data sent across the network on Layer 1



Receiver strips off headers from Layer 2 up to Layer 7



Strip Layer 2 header, validate MAC

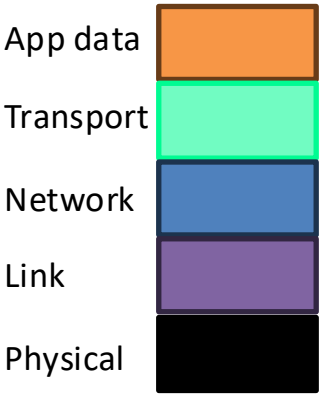
Strip Layer 3 header, validate IP, read protocol field (UDP, TCP, ICMP)

Strip Layer 4 header, use dport to send to destination app via socket

Reply from Server reverses the process back to the Client

Client

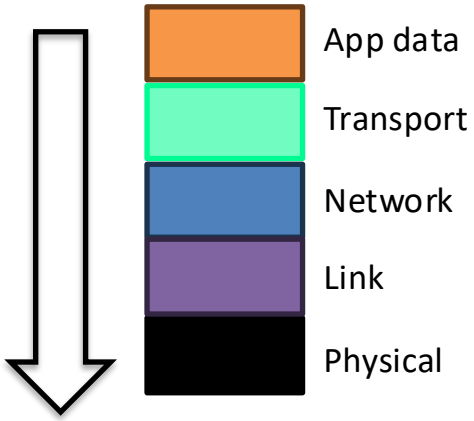




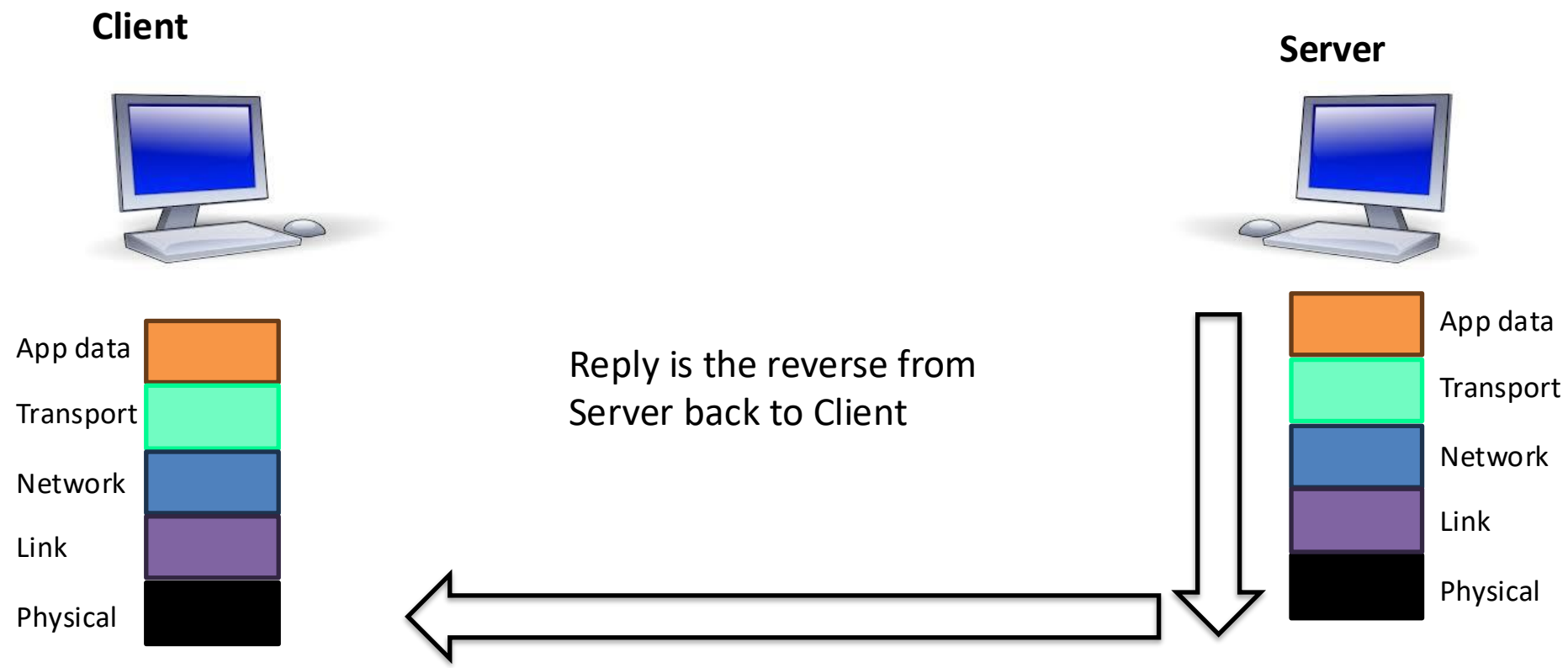
Reply is the reverse from Server back to Client

Server

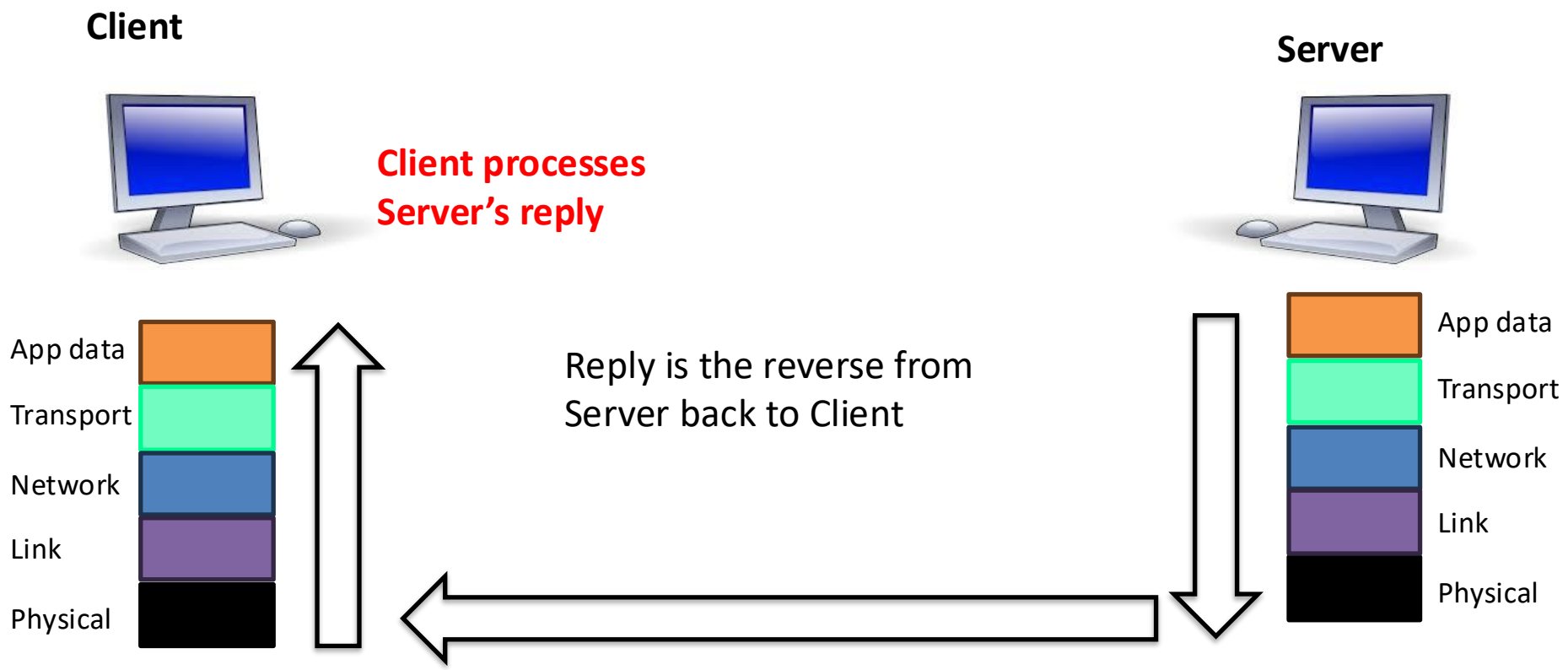




Reply from Server reverses the process back to the Client



Reply from Server reverses the process back to the Client



Agenda

1. Transport (Layer 4) services
2. Multiplexing and Demultiplexing: How does data get to the right application?
3. UDP header bytes
4. UDP code
5. 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

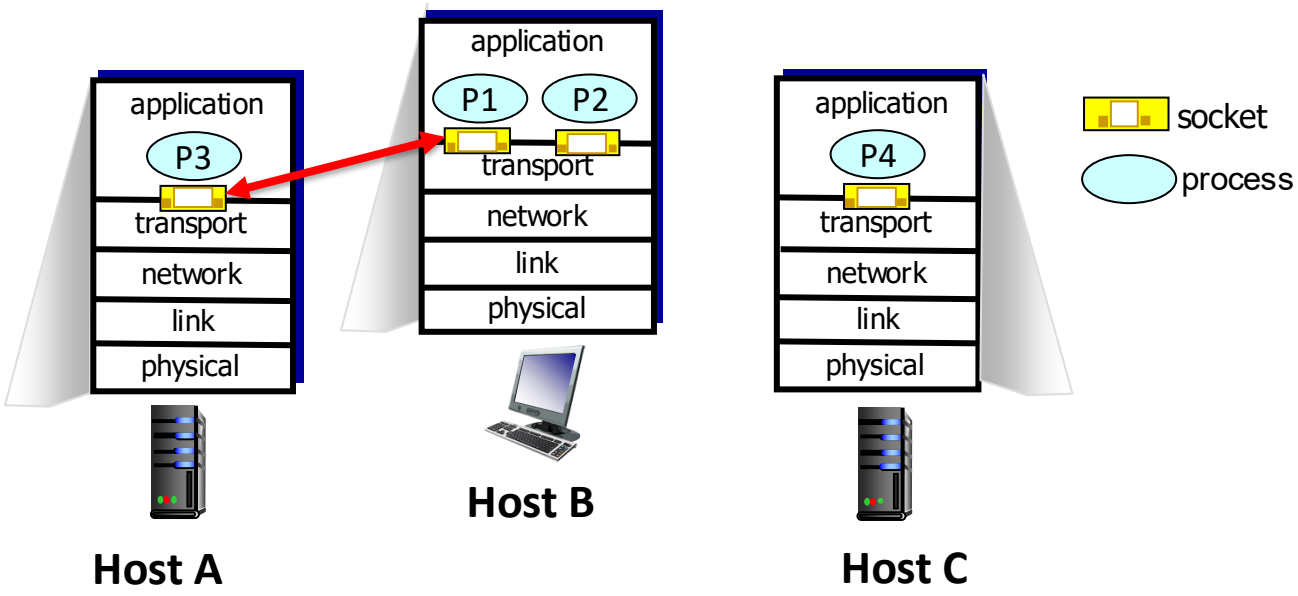
1024-49,151 user or registered ports

49,152-65,535 ephemeral ports

Find known ports at /etc/services

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

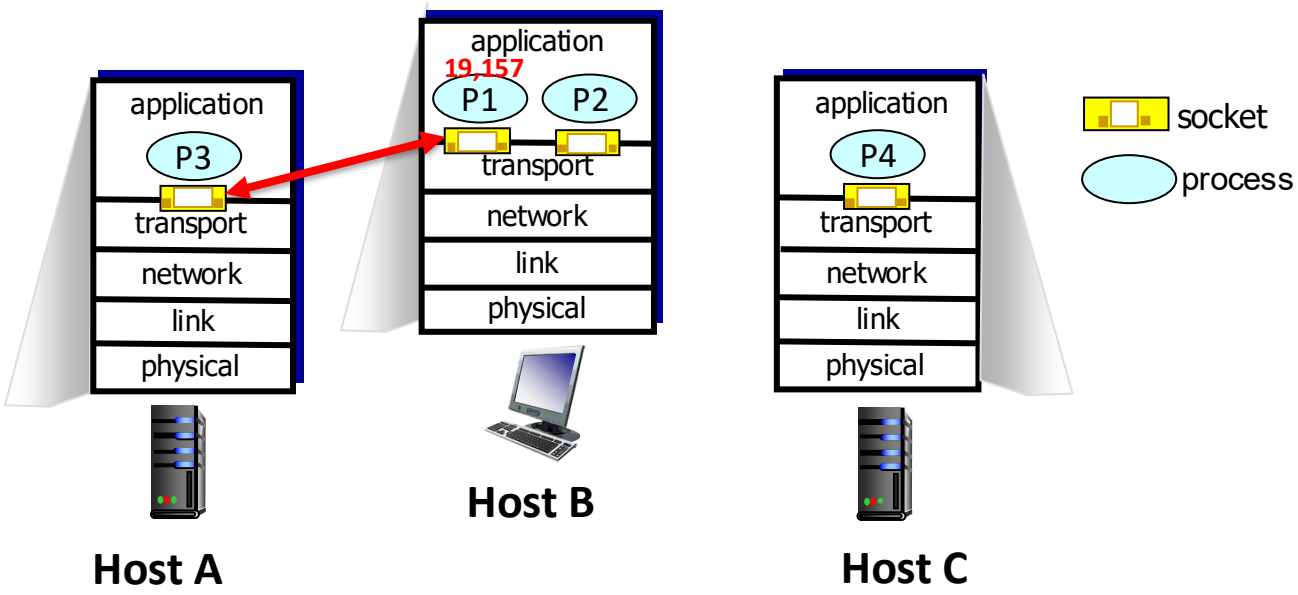
P1 on Host B initiates a connection to P3 on Host A through a Socket



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

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

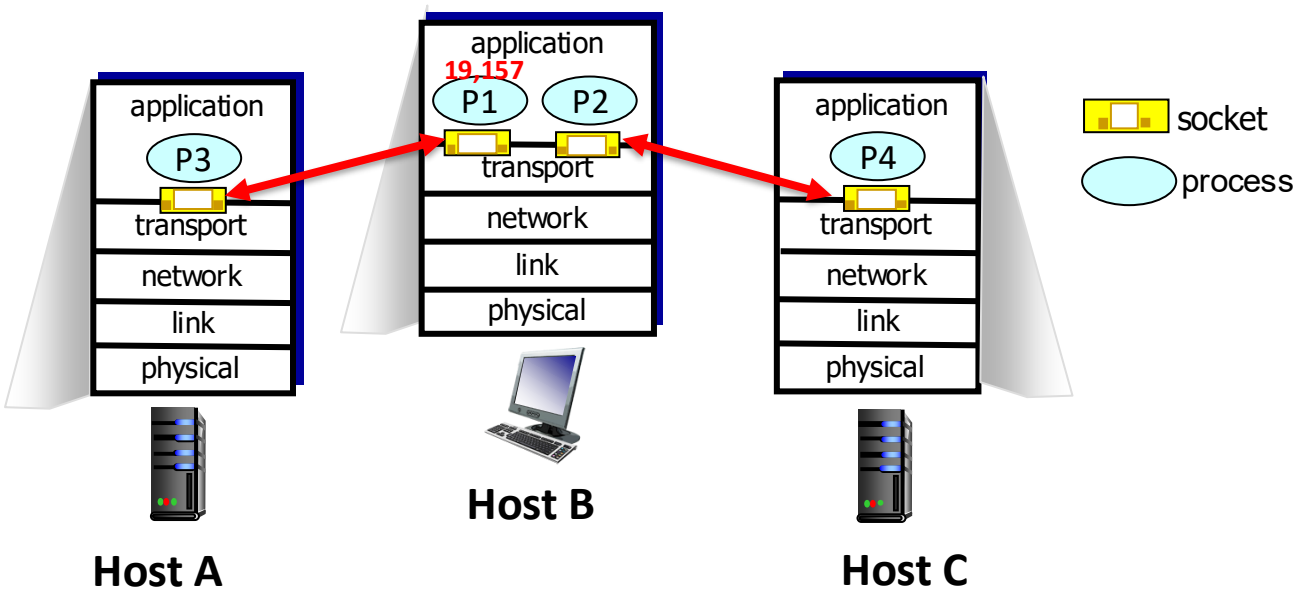
Host B creates a Socket with an unused Port number to talk to P3 (19,157)



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

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

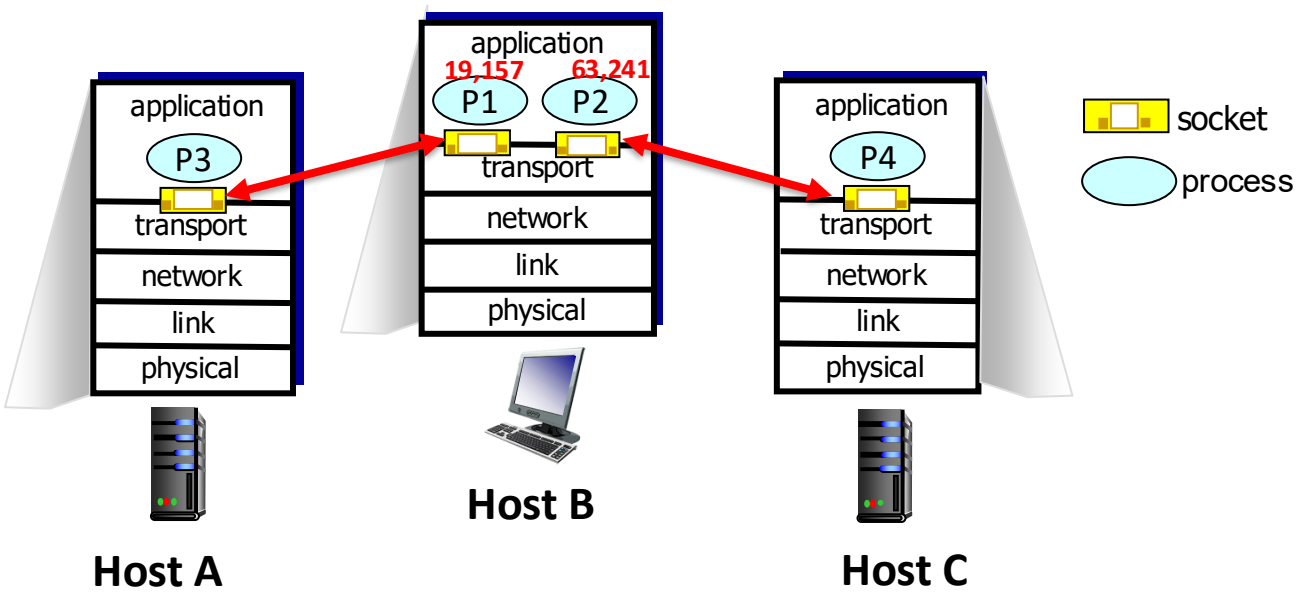
P2 on Host B initiates a connection to P4 on Host C through another Socket



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

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

Likewise, Host B creates a different Socket with a different Port number for P4 (63,241)



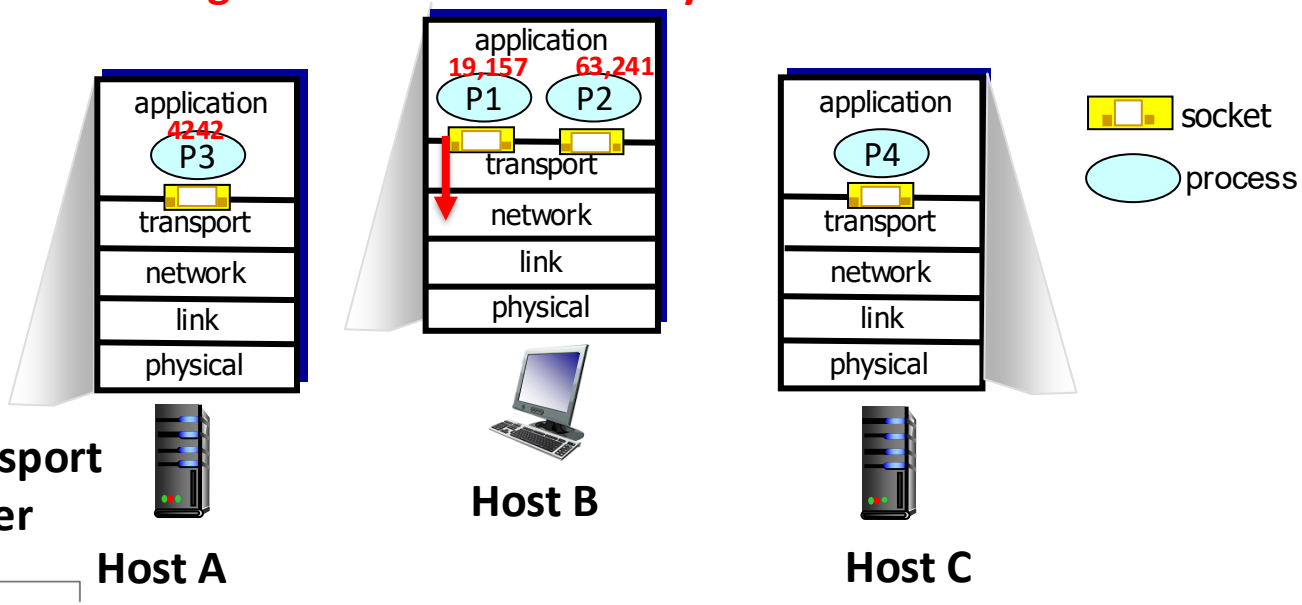
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

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

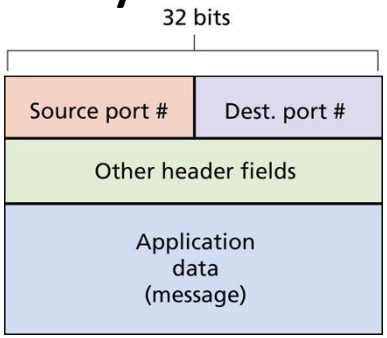
When Host B *sends* a datagram it adds a Transport Layer header

- **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 datagram to the Network Layer



Simplified Transport Layer Header



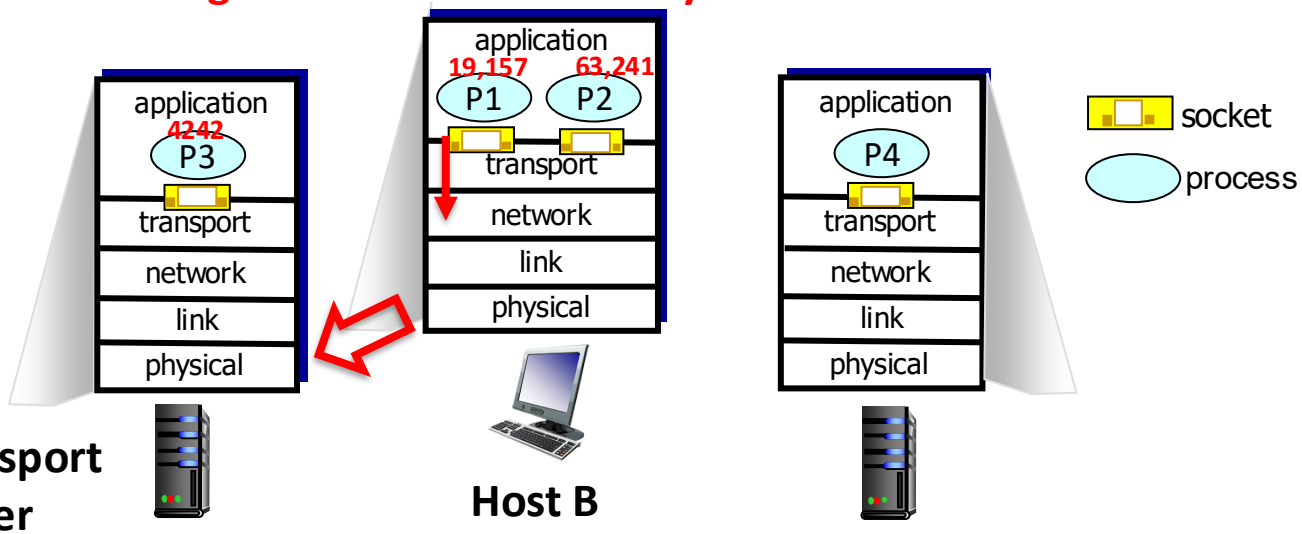
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

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

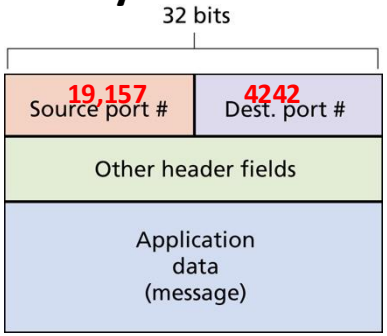
When Host B *sends* a datagram it adds a Transport Layer header

- **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 datagram to the Network Layer



Simplified Transport Layer Header



Network layer sends datagram to Link Layer

Link Layer sends frame to Physical Layer

Physical Layer sends data across the network

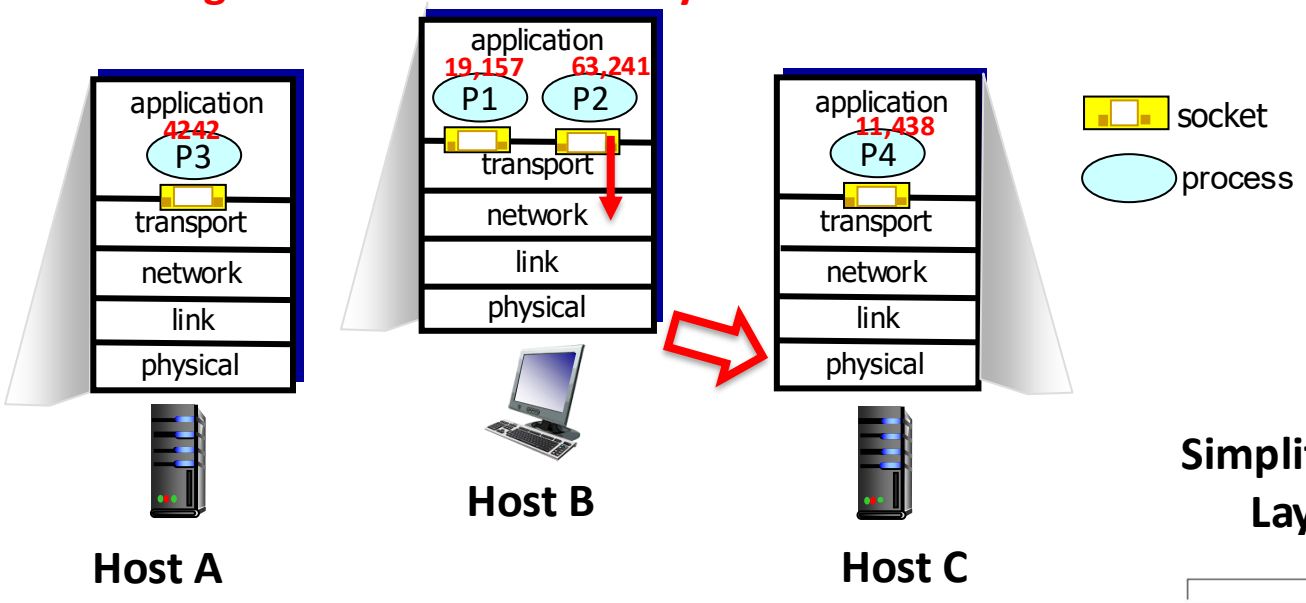
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

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

When Host B *sends* a datagram it adds a Transport Layer header

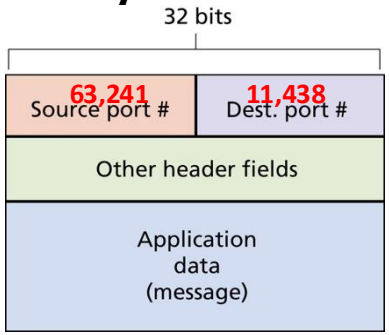
- **Sets header source port to the Socket's port number (63,241)**
- **Sets header destination port (11,438) to the other host's port number**
- **Also adds a checksum (covered soon)**

Then passes datagram to the Network Layer



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

Simplified Transport Layer Header

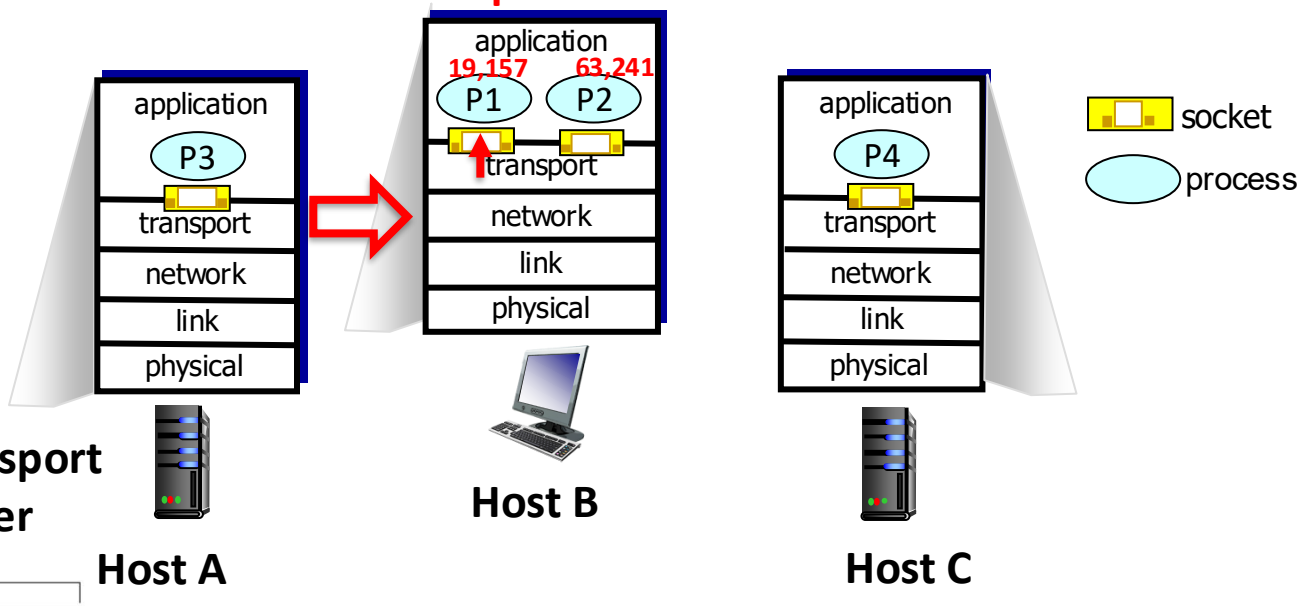


Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

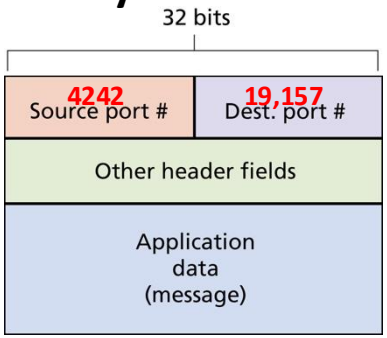
Suppose Host B is running processes P1 and P2, talking to processes on other hosts

When Host B *receives* a datagram

- **Packet traverses up the network stack (MAC and IP addresses checked at Layers 2 and 3)**
- **Layer 4 examines the Transport Layer header**
- **Finds the destination port in the Transport header (19,157)**
- **Sends data to the Socket with that port number**



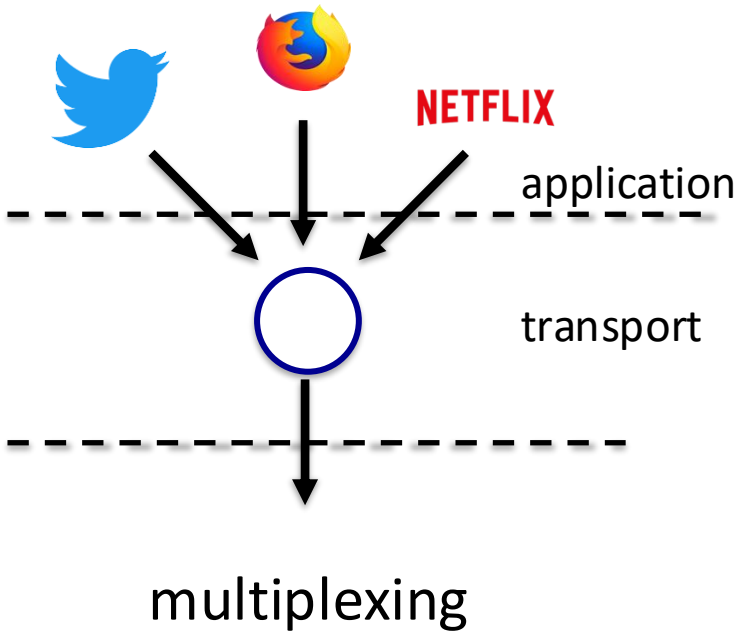
Simplified Transport Layer Header



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

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

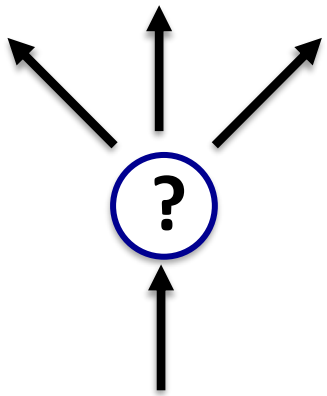
Multiplexing: sends datagrams out the proper Socket to the Transport Layer



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

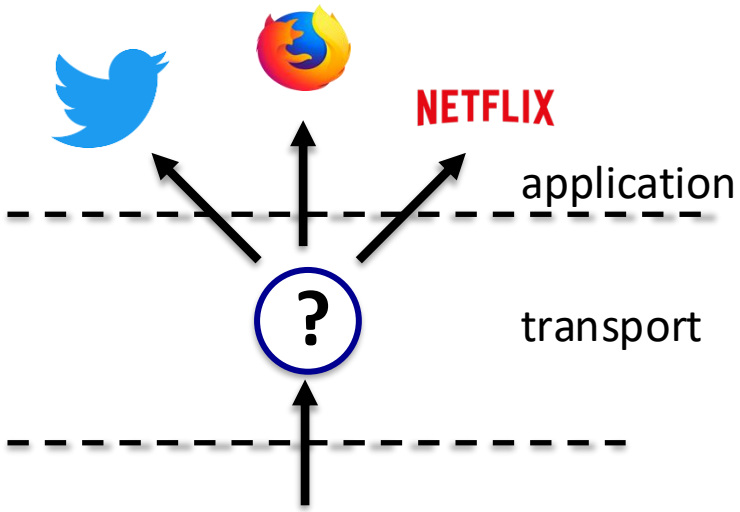


Demultiplexing: sends received datagrams to the proper application



de-multiplexing

Demultiplexing: sends received datagrams to the proper application



de-multiplexing

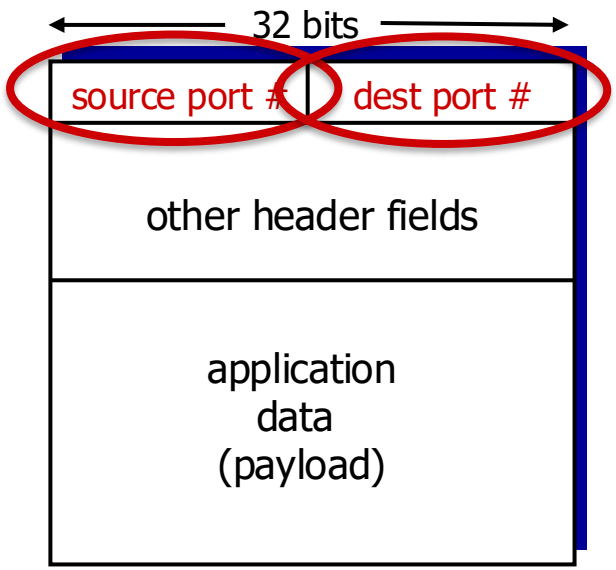
Adapted from Kurose and Ross: Computer Networking: A Top-Down Approach

Demultiplexing: sends received datagrams to the proper application



How demultiplexing works

- Host receives Layer 3 IP packet
  - Each Layer 3 packet has source IP address, destination IP address
  - If destination IP is this host, send to Layer 4
- Each Layer 3 packet carries one Layer 4 transport-layer datagram as its payload
  - Each datagram has source and destination port numbers
- Host uses *port numbers* to direct datagram to appropriate Socket



UDP datagram format

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

UDP connectionless demultiplexing

Recall:

- When creating Socket to *listen* for connections, 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 *send* into UDP Socket, 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* datagram:

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

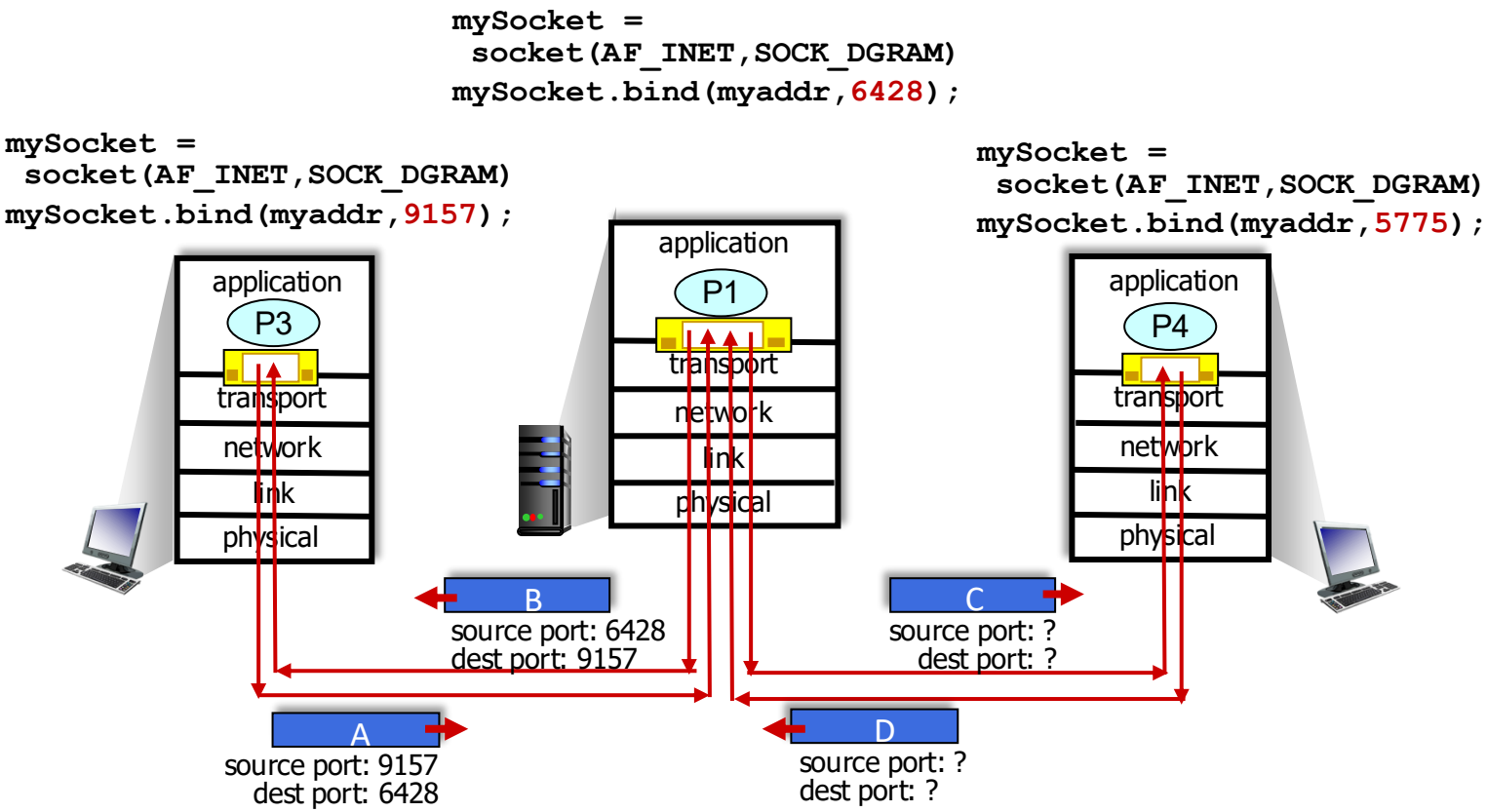


UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Next class we will see TCP works slightly differently

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

UDP connectionless demultiplexing example



Replies flip source and destination ports

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. UDP header bytes
4. UDP code
5. Exercises

UDP: User Datagram Protocol RFC is simple

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

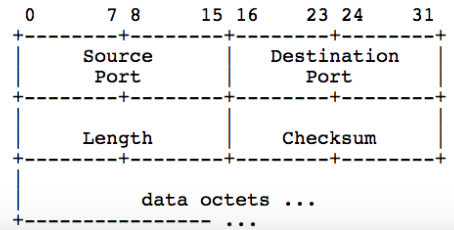
User Datagram Protocol

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



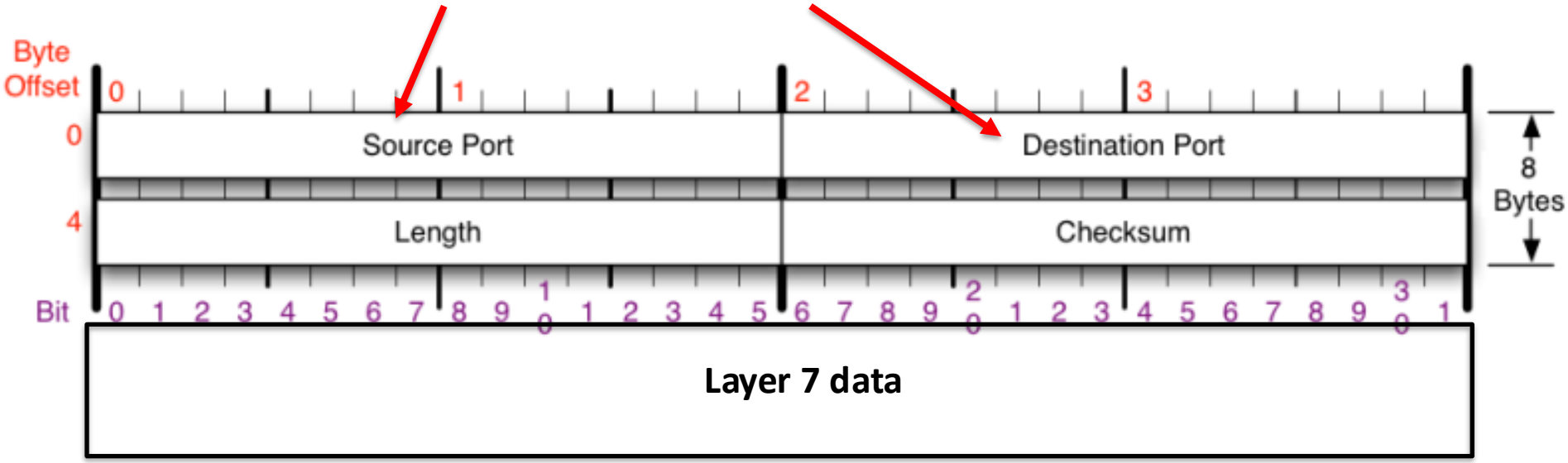
UDP is defined in Request For Comment (RFC) 768

It is only 3 pages long!

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

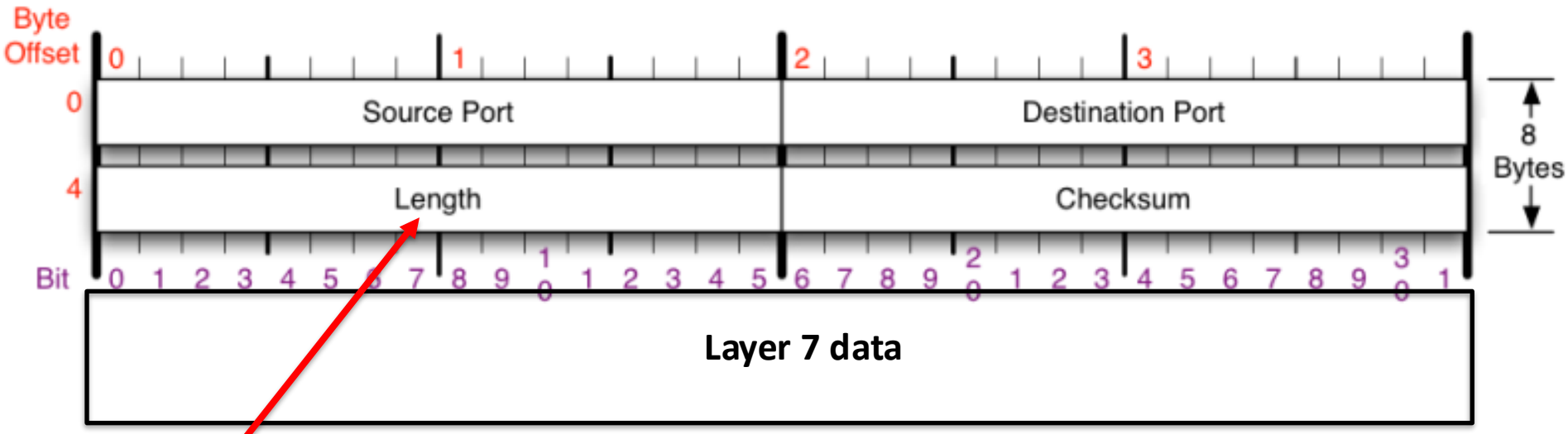
UDP header is only 64 bits (8 bytes) long; port numbers connect applications

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



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

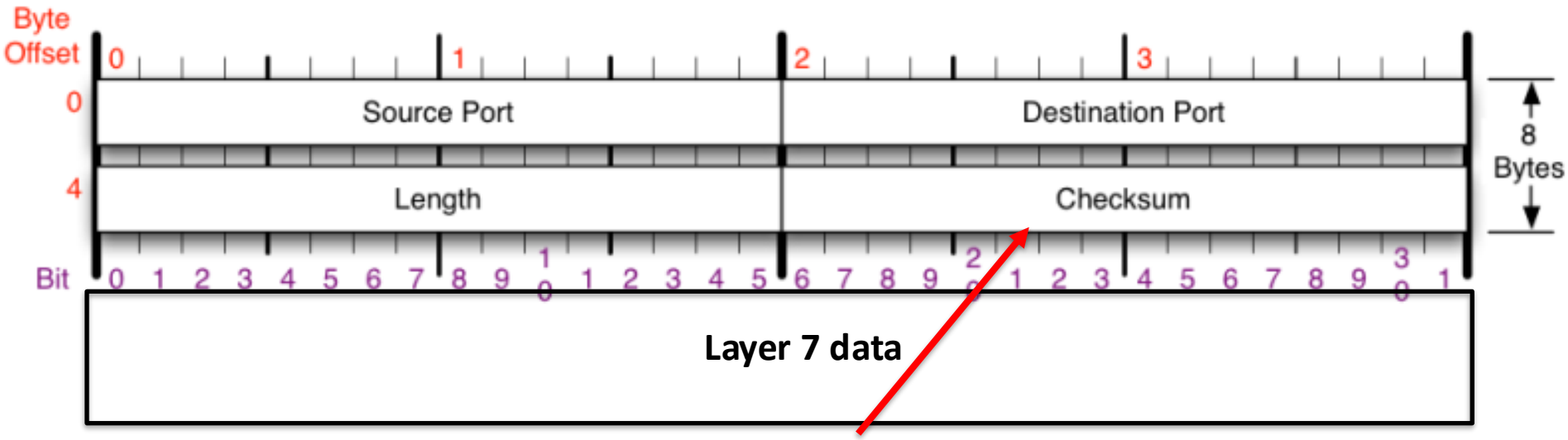
With length we know where the datagram ends

Theoretical max UDP datagram is 2^{16}-1 = 65,535 bytes
Subtract 8 bytes for header = 65,527 bytes of data

In practice, lower layers will constrain the max data size to a smaller value (at least in IPv4)

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

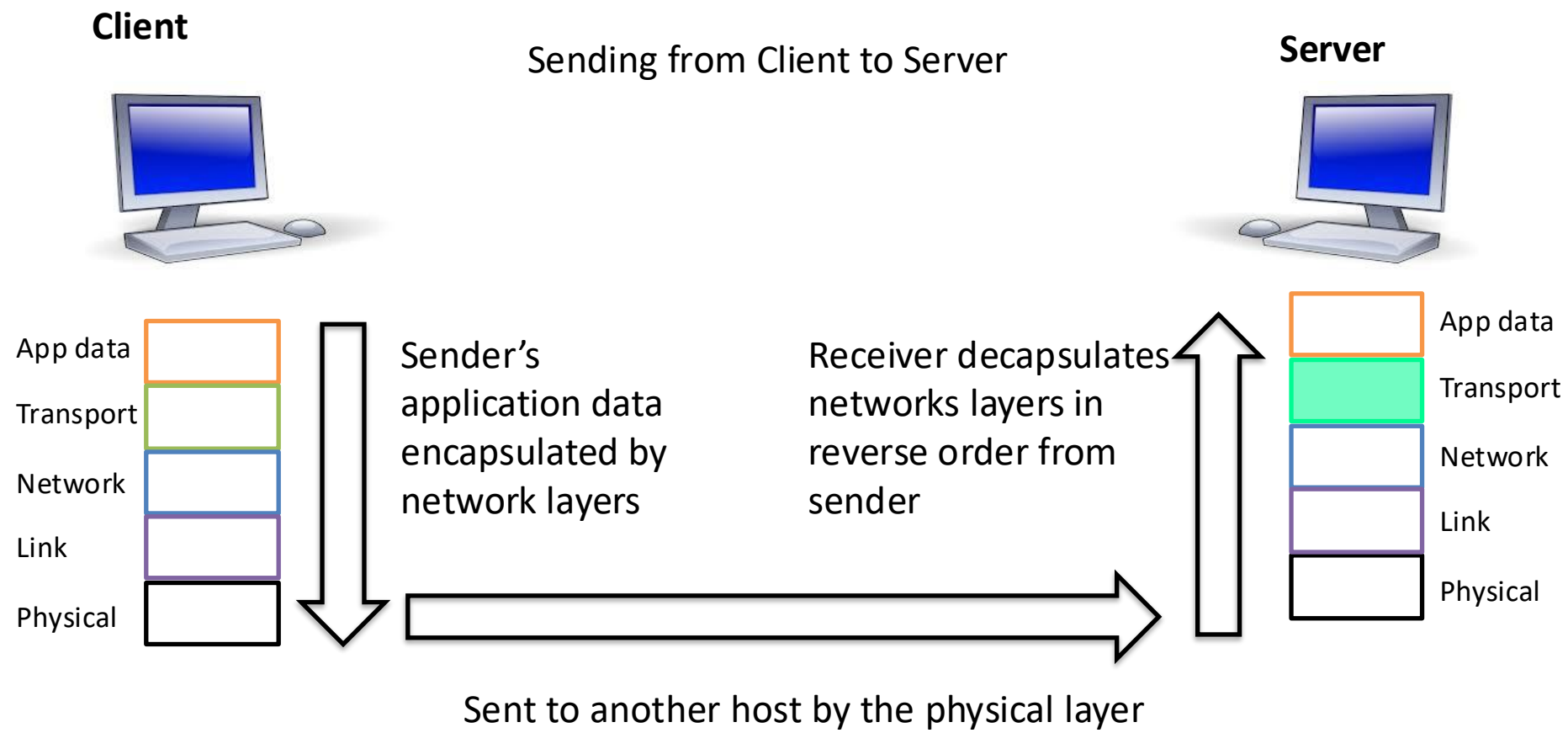
Checksum does error detection



Checksum does error detection (but not error correction!)
Calculation shown soon
Why do we need a checksum?
Detects (some of the time) whether bits have been altered in transit (for example: by noisy links or while stored in routers) as the datagram moved from source to destination

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

Layer 4 validates checksum, if pass, send data to socket using destination port

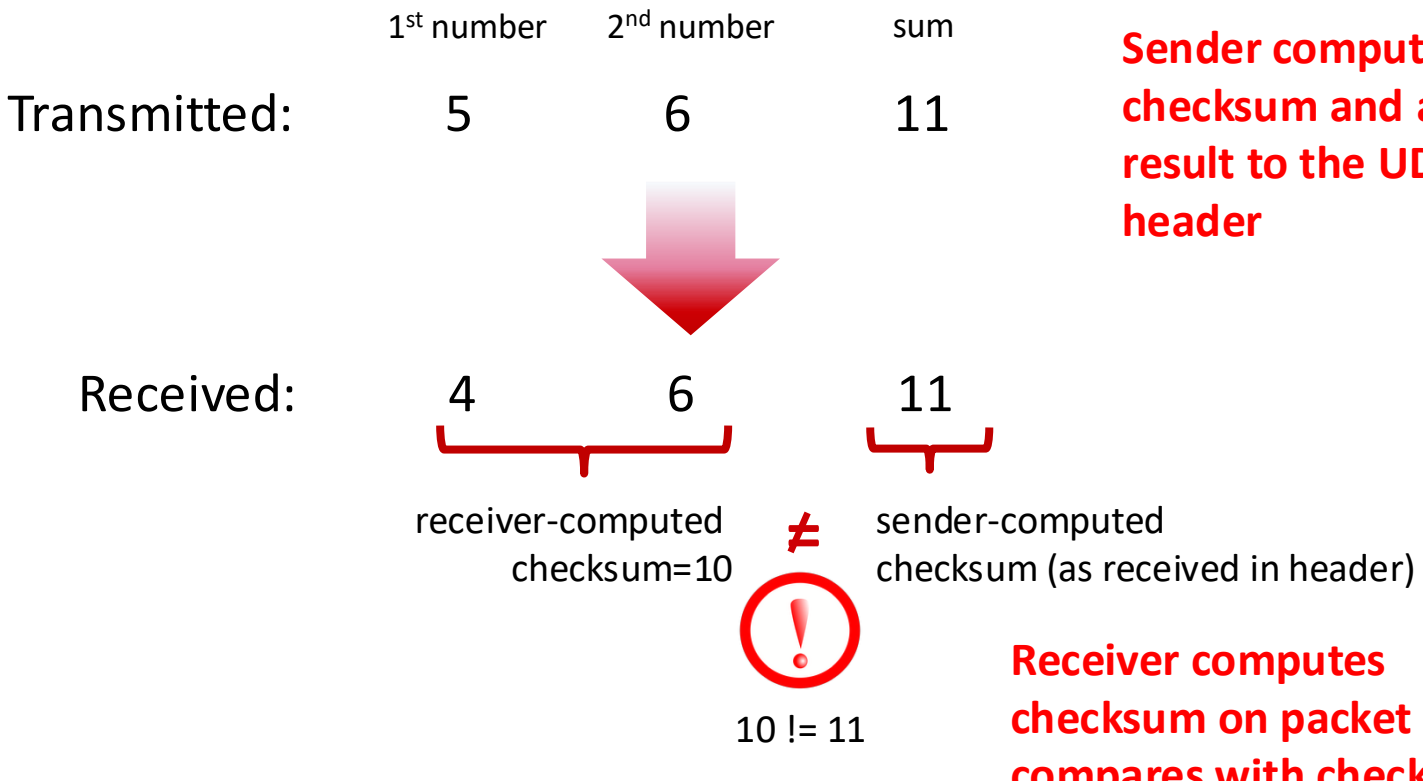


Receiver at Layer 4:

- **Calculates checksum**
- **Compares with checksum transmitted in UDP header field**
- **If match, pass to appropriate socket using destination port**

UDP checksum helps detect bit errors

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



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 datagram

Sender:

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

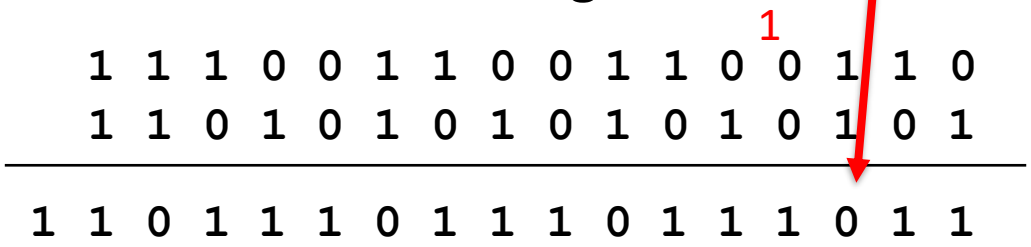
Receiver:

- Compute checksum of received datagram
- 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 datagram and includes it in the header

Example: add two 16-bit integers



Review

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 (carry 1)

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

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

Review

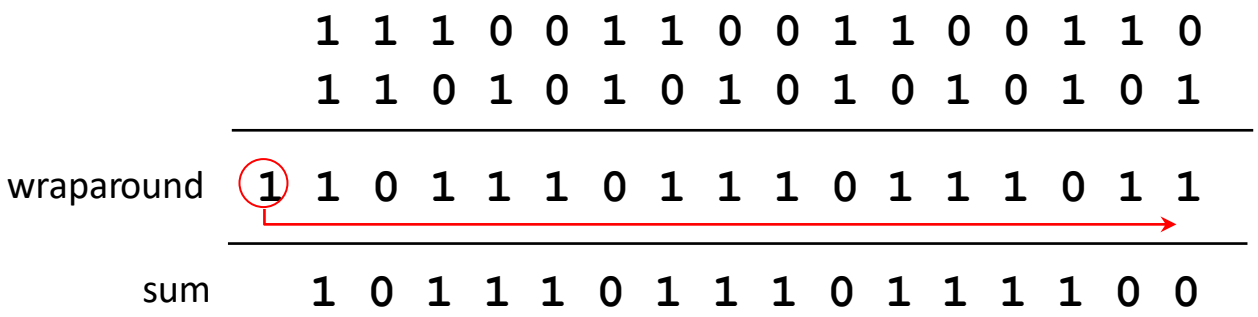
0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 (carry 1)

Example: add two 16-bit integers



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 datagram and includes it in the header

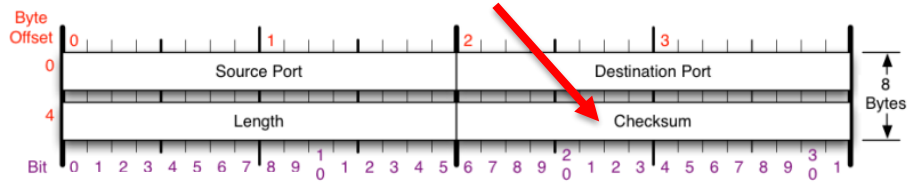
Example: add two 16-bit integers

	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 complement 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



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

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

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

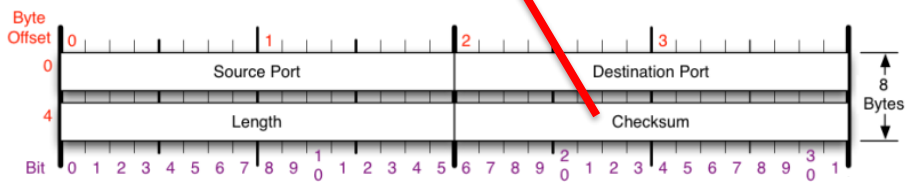
Example: add two 16-bit integers

	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 received

Receiver adds the checksum from the sender in the UDP header

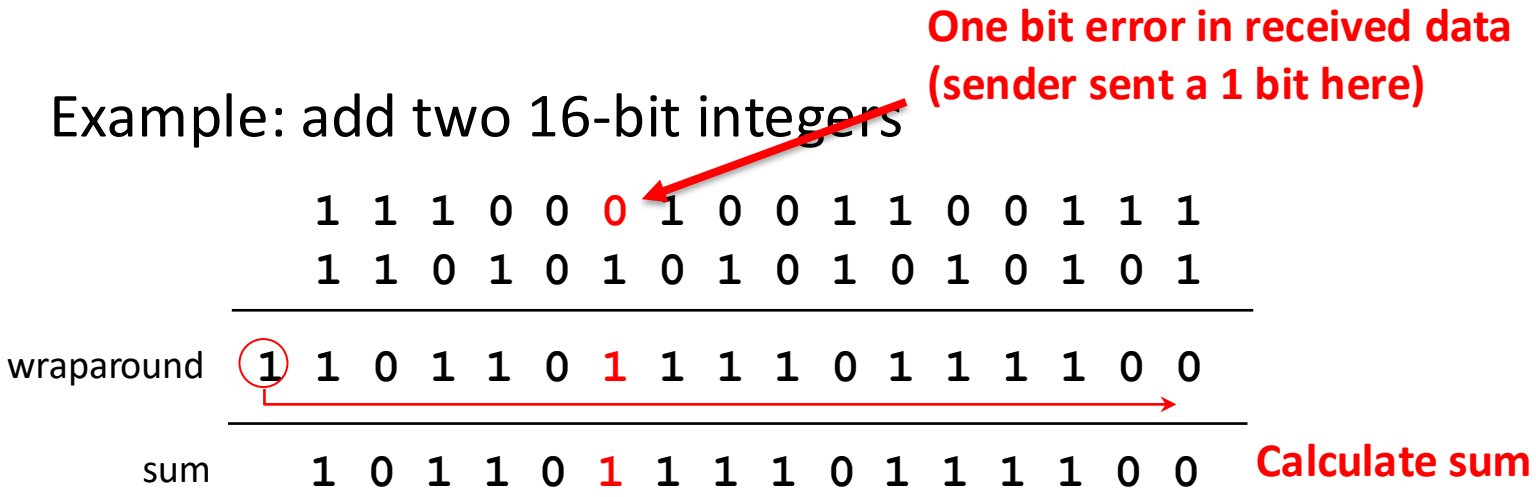
Result should be all 1's if there are no bit errors



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

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

Example: add two 16-bit integers



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

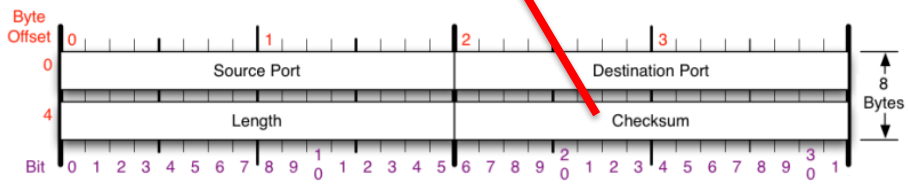
One bit error in received data (sender sent a 1 bit here)

	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

Receiver calculates sum

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

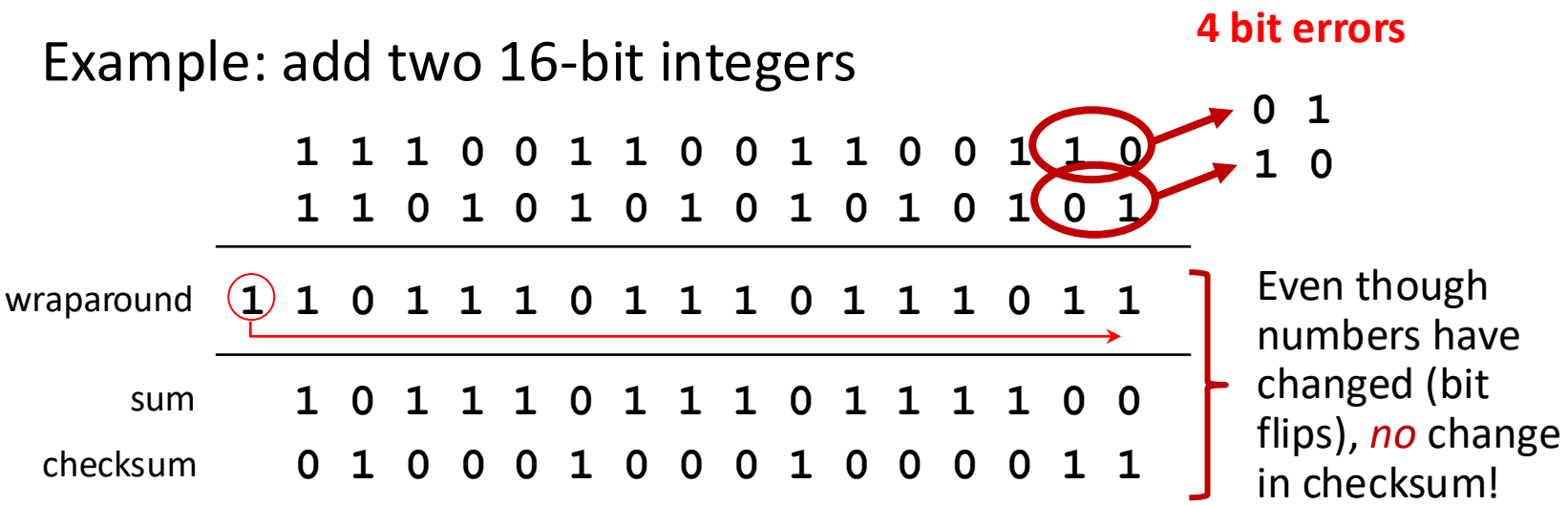
Adds checksum from the UDP header



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

Internet checksum has weak protection

Example: add two 16-bit integers



In some cases, checksum calculation may indicate no errors if there are bit errors

Convince yourself that statement is true outside of class

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

UDP summary

- "No frills" protocol:
  - Datagrams may be lost, delivered out of order
  - Best effort service: "send and hope for the best"
  - Message oriented: data is not segmented into multiple packets like TCP
  - If you want to send more than max bytes, you must split them up yourself
  - At the receiver, a *recvfrom* will not get data from multiple UDP datagrams (unlike TCP)

- UDP has its plusses:
  - No setup/handshaking needed (no RTT incurred)
  - Can function when network service is congested
  - Helps to detect errors (checksum)

- Can build additional functionality on top of UDP in application layer (e.g., HTTP/3 or QUIC)

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. UDP header bytes
4. UDP code
5. Exercises

Sending a UDP datagram using a socket is not difficult with Python

send_udp.py

```
import socket

dest_addr = "127.0.0.1"
port = 9090
msg = b'Hello world!'


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(msg,(dest_addr,port))
    sock.close()
```

Sending a UDP datagram using a socket is not difficult with Python

send_udp.py

Python hides a lot of complexity!

```
import socket

dest_addr = "127.0.0.1"
port = 9090
msg = b'Hello world!'


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(msg,(dest_addr,port))
```

Note: we *do not* set a source port on the client side
OS picks a random unused port for us
We *do* set a port on the server side
Client needs to know what port the server is listening on

Listen for UDP in a terminal on port 9090 using netcat (nc)

Run program

Terminal 1
```
$ python3 send_udp.py
```

Terminal 2
```
$ nc -luv 9090
Hello world!
```

netcat receives UDP message when sent

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f"{ip}:{port} {msg.decode('utf-8')}")
```

Computer might have multiple NICs
ip_addr of zeros means accept packets from any of them

Set up socket for UDP as before

Bind socket to listen on ip_addr and port
Note: we pick a port on the server (receiver) side

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))
    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f"{ip}:{port} {msg.decode('utf-8')}")
```

Loop forever

Block until packet arrives
recvfrom gets data
here print client ip, port, and msg

Use recvfrom for UDP
Use recv for TCP

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f "{ip}:{port} {msg.decode('utf-8')}")
```

Receiver gets msg, ip addr, and port

Sender types msg

Terminal 1 (receiver)

$ python3 receive_udp.py
127.0.0.1:64369 hello

Terminal 2 (sender)

$ nc –u 127.0.0.1 9090
hello

UDP receiver in Python using a Socket

receive_udp.py

We did not pick a source port on the sender (Terminal 2)
OS chooses source port randomly
When this is run, see *source* port other than 9090 (it was 64369 here)
On *receiver*, OS can sort out replies if multiple instances of application are running at the same time based on port

```
import socket

MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))


    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f"{ip}:{port} {msg.decode('utf-8')}")
```

Sender types msg

Receiver gets msg, ip addr, and port

Terminal 1 (receiver)
$ python3 receive_udp.py
127.0.0.1:64369 hello

Terminal 2 (sender)
$ nc –u 127.0.0.1 9090
hello

UDP receiver in Python using a Socket

receive_udp.py

```
import socket


MAX_SIZE = 1500 #max message size in bytes
ip_addr = "0.0.0.0" #0.0.0.0 means bind to all interfaces
port = 9090  #listen on this port


if __name__ == '__main__':
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((ip_addr,port))

    while (True):
        msg, (ip, port) = sock.recvfrom(MAX_SIZE)
        print(f "{ip}:{port} {msg.decode('utf-8')}")
```

Run with two senders on the same computer (localhost)
Each sender got different source port
Receiver can tell them apart

Terminal 1 (receiver)

```
$ python3 receive_udp.py
127.0.0.1:53072 hello
127.0.0.1:52815 sup?
```

Terminal 2 (sender)

```
$ nc –u 127.0.0.1 9090
hello
```

Terminal 3 (sender)

```
$ nc –u 127.0.0.1 9090
sup?
```

Agenda

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

Exercise

Write a UDP client and server

1. Client: Use the VM to write a simple client that
   - Reads a message from the keyboard
   - Sends a UDP datagram with the message to a server app on your neighbor's VM
   - Listens for a reply and prints it
2. Server:
   - Loop forever
     - Receives UDP datagrams containing message from client
     - Replies to the client with the message in all caps
3. Start the VPN to send to another student in class (or Dartmouth blocks)
4. Use ip a (ifconfig) to find ip addresses, use the *tun0* interface ip address

Look at a Wireshark capture of the exchange

- Double check the UDP header is only 8 bytes