Lecture 10: Transport Layer Overview and UDP

COMP 332, Spring 2023
Victoria Manfredi

WESLEYAN UNIVERSITY

Acknowledgements: materials adapted from Computer Networking: A Top Down Approach 7th edition: ©1996-2016, J.F Kurose and K.W. Ross, All Rights Reserved as well as from slides by Abraham Matta at Boston University, and some material from Computer Networks by Tannenbaum and Wetherall.

Today

1. Announcements
   – homework 4 due tomorrow
   – homework 5 posted, due Friday before break

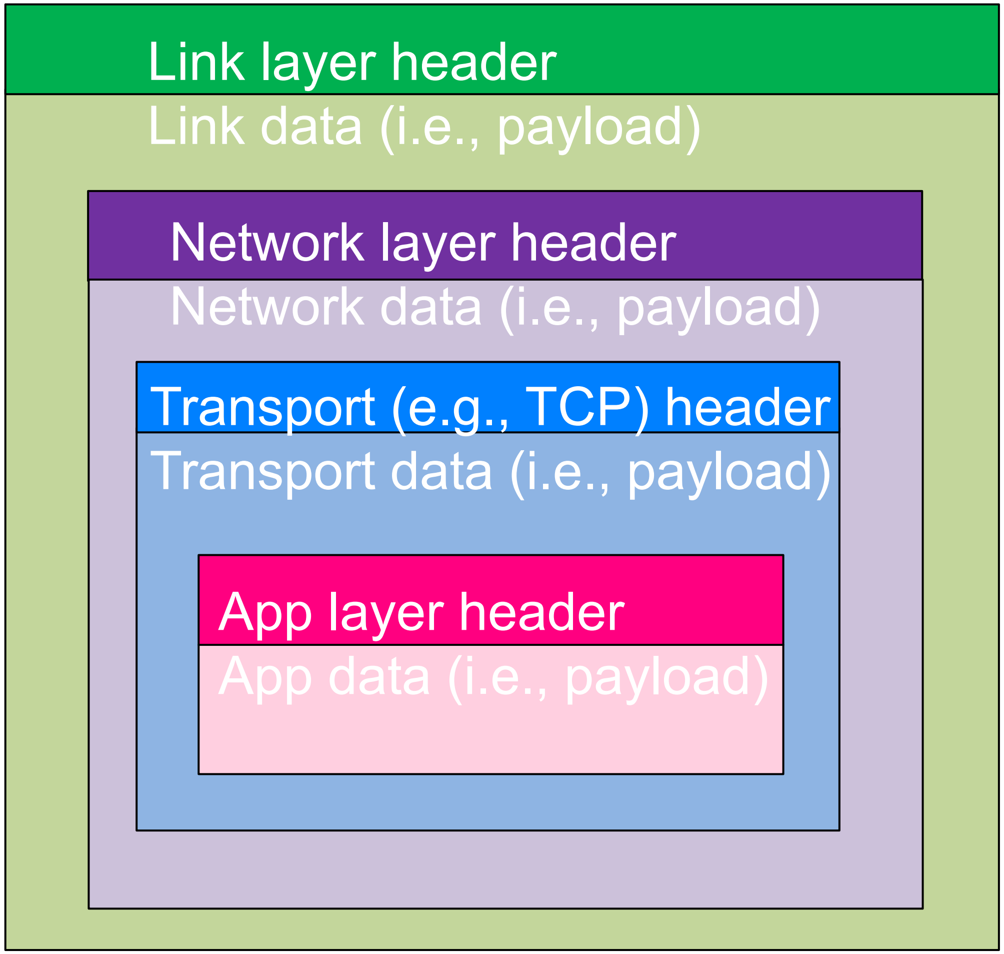
2. Headers and payloads
   – recap

3. Transport layer
   – overview
   – multiplexing and demultiplexing
   – User Datagram Protocol (UDP)

Headers and Payloads

RECAP

Headers and payloads



Each layer only looks at the header associated with that layer

Transport Layer

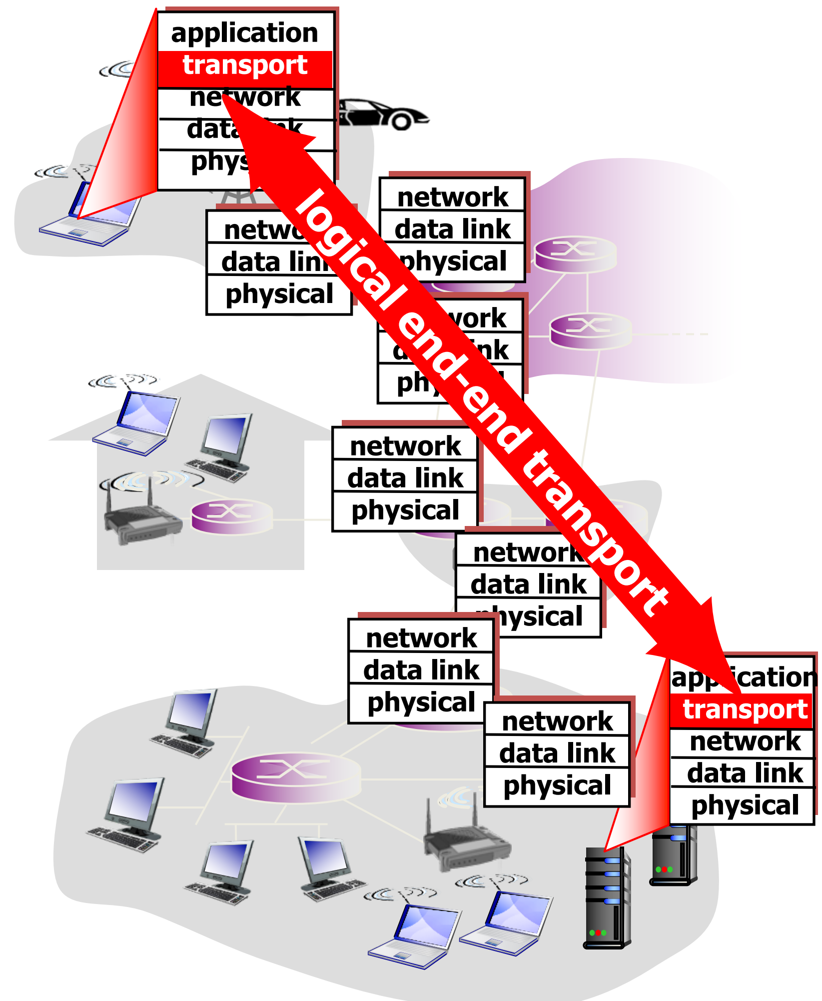
OVERVIEW

Why do we need a transport layer?

- Application
- Transport
- Network
- Link
- Physical

- Logical communication between processes on end hosts
- Relies on, enhances, network layer services

- Logical communication between end hosts
- IP header does not contain port #s



What problems must transport layer address?

Why do we need a transport layer?

Transport layer services

Problem 1: no port #s in network-layer (IP) header
- how do pkts get from host to process on host?

→ (De)Multiplexing

Problem 2: network layer protocol (IP) is best effort
- packets can be corrupted, dropped, duplicated, reordered, delayed
- pain for app developer to deal with

→ Reliable data transfer

Problem 3: IP gives no guidance about rate at which to send packets
- sends whatever it receives immediately
- traffic can easily overwhelm network, host

→ Congestion, flow control

Problem 4: IP packets must be reassembled back into original messages
- pain for app developer to deal with

→ Data stream

Why do we need a transport layer?

Transport layer services

Problem 1: no port #s in network-layer (IP) header

- how do pkts get from host to process on host?

(De)Multiplexing
Only service transport layer MUST provide!
UDP, TCP

Problem 2: network layer protocol (IP) is best effort

- packets can be corrupted, dropped, duplicated, reordered, delayed
- pain for app developer to deal with

Reliable data transfer
TCP

Problem 3: IP gives no guidance about rate at which to send packets

- sends whatever it receives immediately
- traffic can easily overwhelm network, host

Congestion, flow control
TCP

Problem 4: IP packets must be reassembled back into original messages

- pain for app developer to deal with

Data stream
TCP

Transport layer protocols on Internet

TCP: reliable, in-order delivery

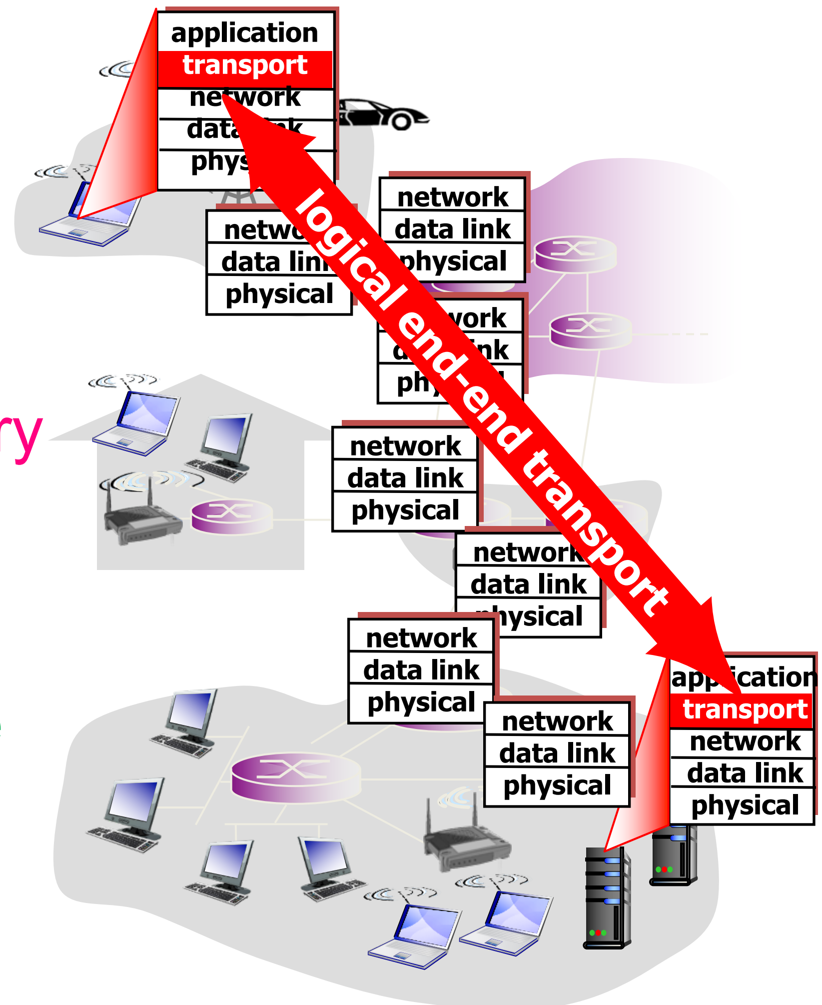
- connection-oriented
- congestion control
- flow control
- connection setup

UDP: unreliable, unordered delivery

- connectionless
- no-frills extension of best-effort IP

Q: What services are not available

- delay guarantees
- bandwidth guarantees



Transport Layer

MULTIPLEXING AND DEMULTIPLEXING

Transport layer

Transport protocols

- run in end systems
- provide logical communication
  - between app processes running on different hosts

Send side

- breaks app messages into segments (TCP) or datagrams (UDP)
- passes to network layer

Receive side

- reassembles segments or datagrams into messages
- passes to app layer

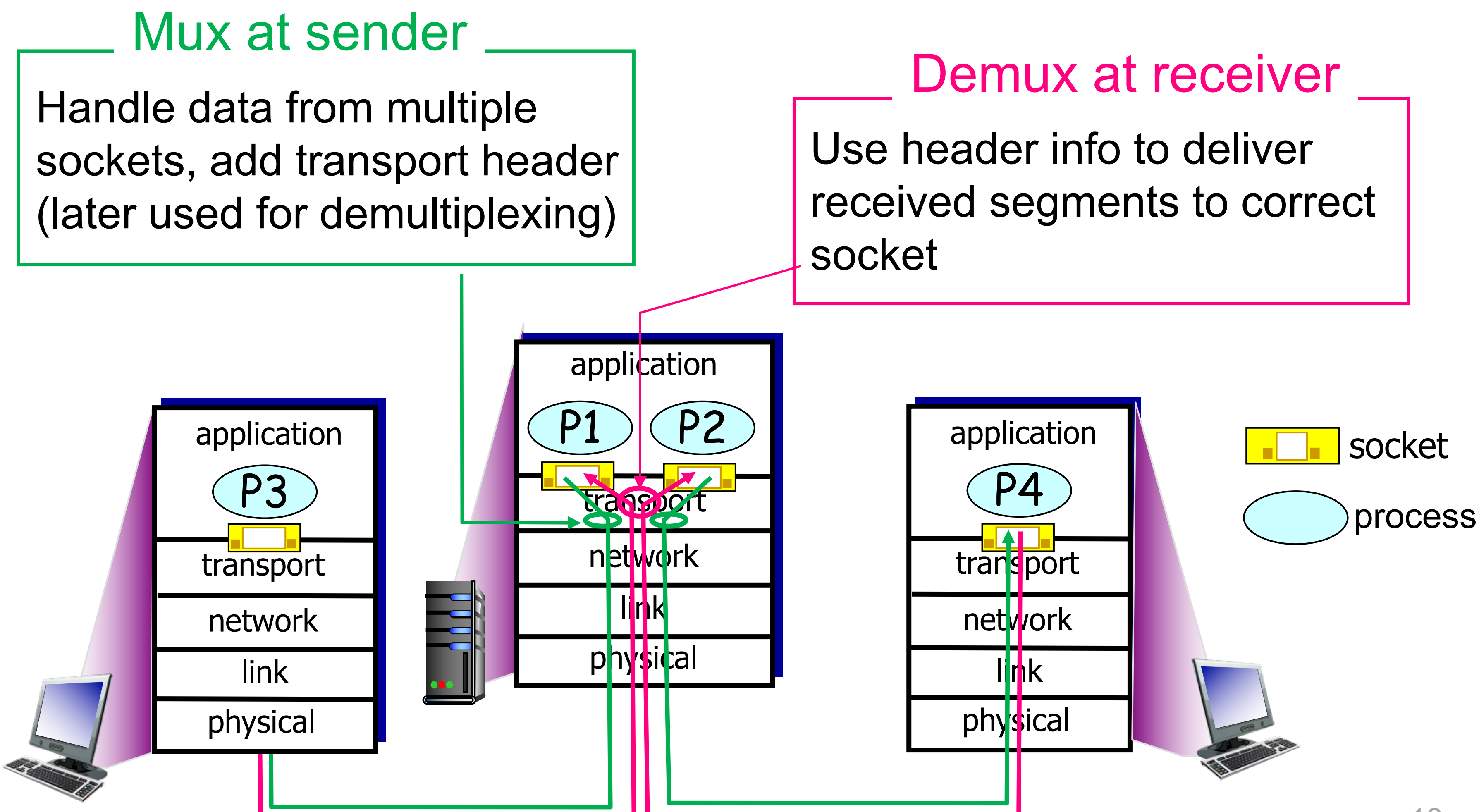
Household analogy

12 kids in Alice's house send letters to 12 kids in Bob's house

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Multiplexing and demultiplexing

Determines which packets go to which app

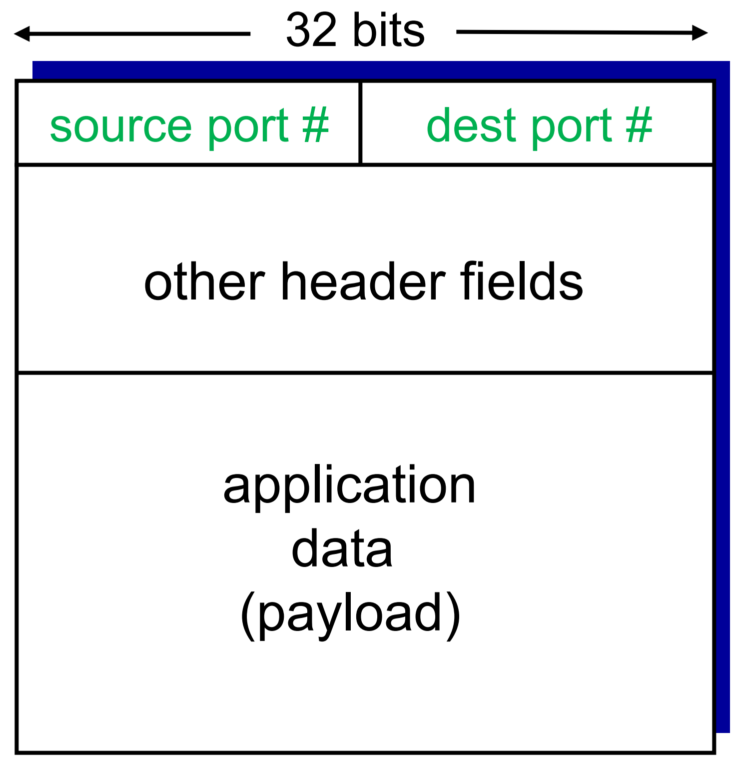


How demultiplexing works

Host receives IP packets

- packet header contains
  - source IP address
  - destination IP address
- packet payload is
  - one transport-layer segment or datagram
- transport-layer header contains
  - source port number
  - destination port number

Host uses IP addresses & port numbers to direct segment or datagram to appropriate socket



Format of TCP segment or UDP datagram

Connection-oriented demultiplexing (TCP)

TCP socket identified by 4-tuple

1. source IP address
2. source port number
3. dest IP address
4. dest port number

Demux

- receiver uses all four values to direct segment to appropriate socket

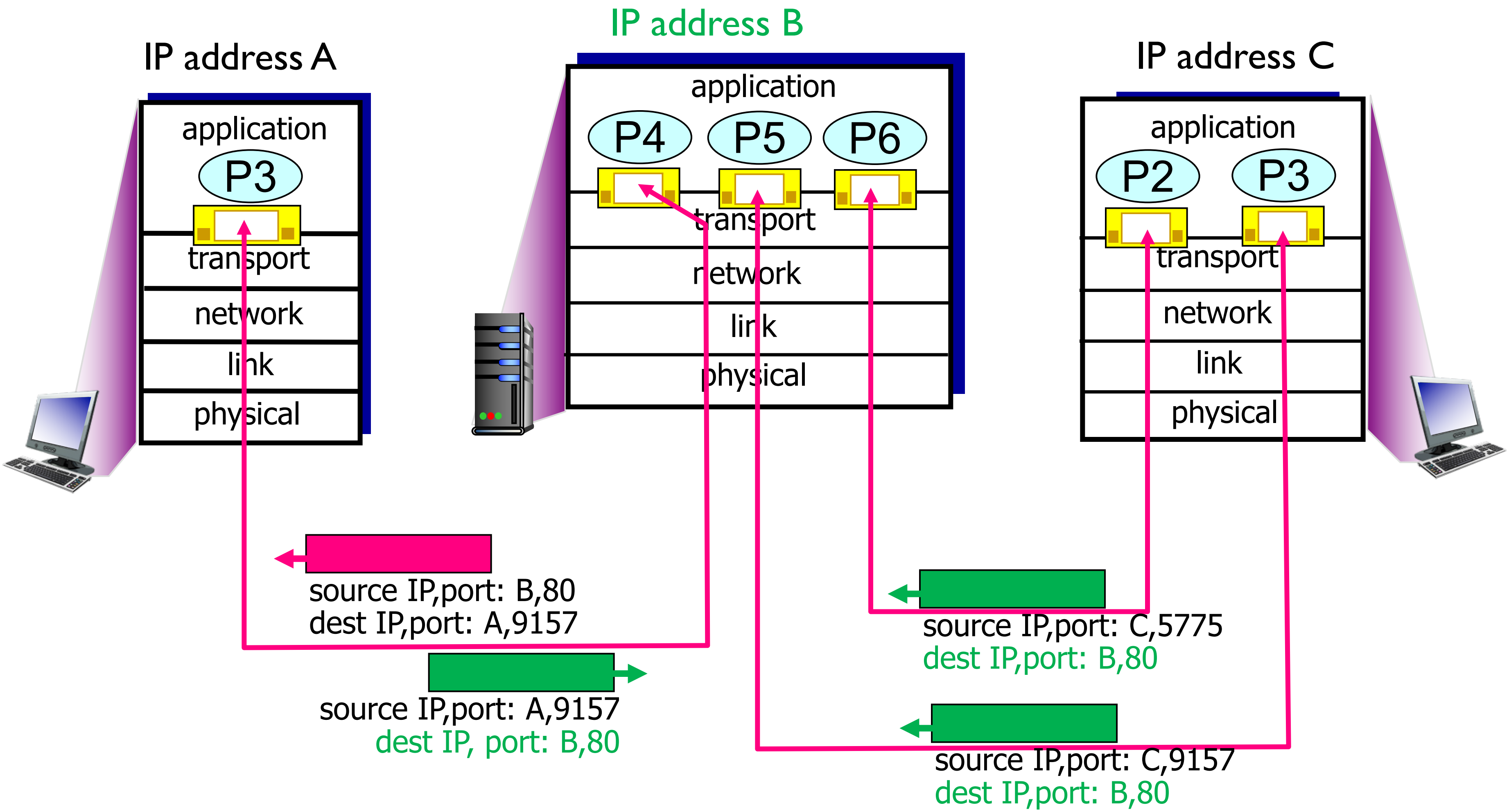
Server host

- may support many simultaneous TCP sockets
- each socket identified by its own 4-tuple

Web servers

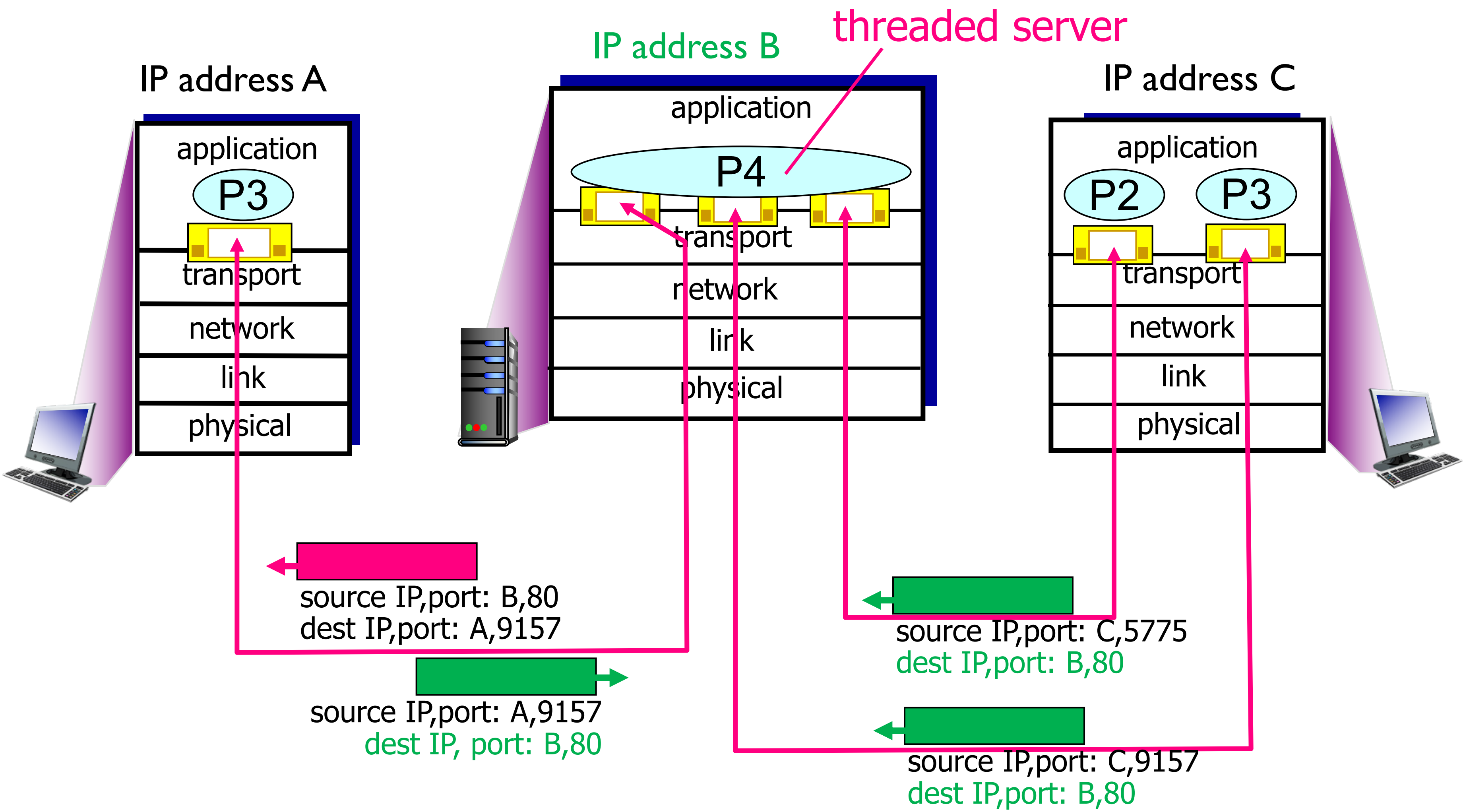
- have different sockets for each connecting client
- non-persistent HTTP will have different socket for each request

Connection-oriented demultiplexing (TCP)



3 segments, all destined to IP address B, dest port 80:
are demultiplexed to *different* sockets

Connection-oriented demultiplexing (TCP)



3 segments, all destined to IP address B, dest port 80:

are demultiplexed to *different* sockets

Connectionless demultiplexing (UDP)

UDP socket

- random host-local port # allocated

```
sock = socket(AF_INET,SOCK_DGRAM)
port# allocated: 9157
```

- when sending data into UDP socket, must specify
  1. destination IP address
  2. destination port #

Host receives UDP datagram

- checks destination port # in UDP header on datagram
- directs UDP datagram to socket with that port #

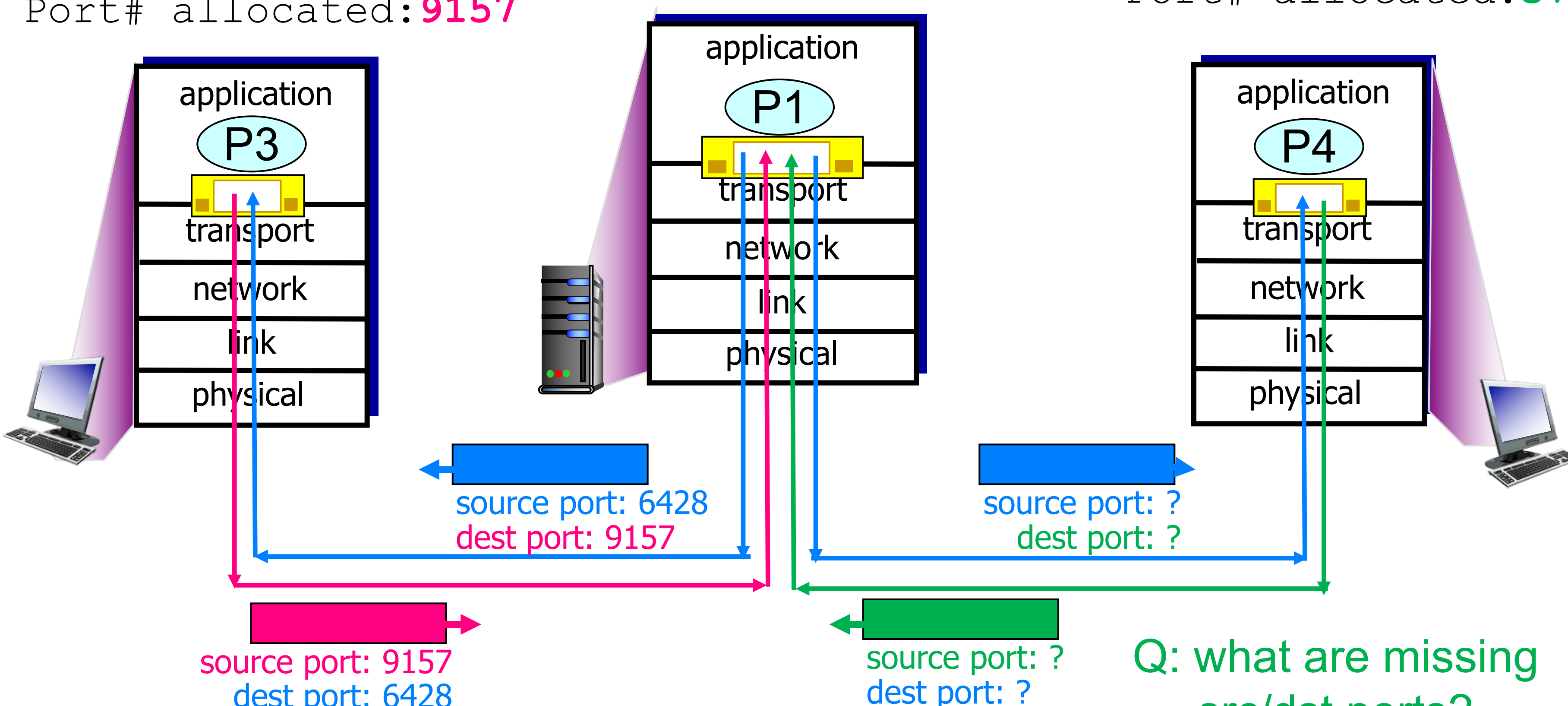
IP pkts with same dst IP, port # but different src IP addr and/or src port #s: will still be directed to same socket at dst!

Connectionless demultiplexing (UDP)

```
sock2 =
socket(AF_INET,
SOCK_DGRAM)
Port# allocated:9157
```

```
server_sock =
socket(AF_INET,
SOCK_DGRAM)
server_sock.bind((
localhost,6428))
```

```
sock1 =
socket(AF_INET,
SOCK_DGRAM)
Port# allocated:5775
```



Q: what are missing src/dst ports?

Looking forward

Start with UDP

– since protocol is much simpler to understand

Then look at TCP

– start with toy protocol to build up pieces we need for full protocol

Transport Layer

USER DATAGRAM PROTOCOL

UDP: User Datagram Protocol [RFC 768]

No frills Internet transport protocol

- **best effort service**
  - UDP segments may be lost, delivered out-of-order to app
- to **add reliable transfer** over UDP
  - add reliability at application layer
  - application-specific error recovery!
- **uses** of UDP
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS, SNMP

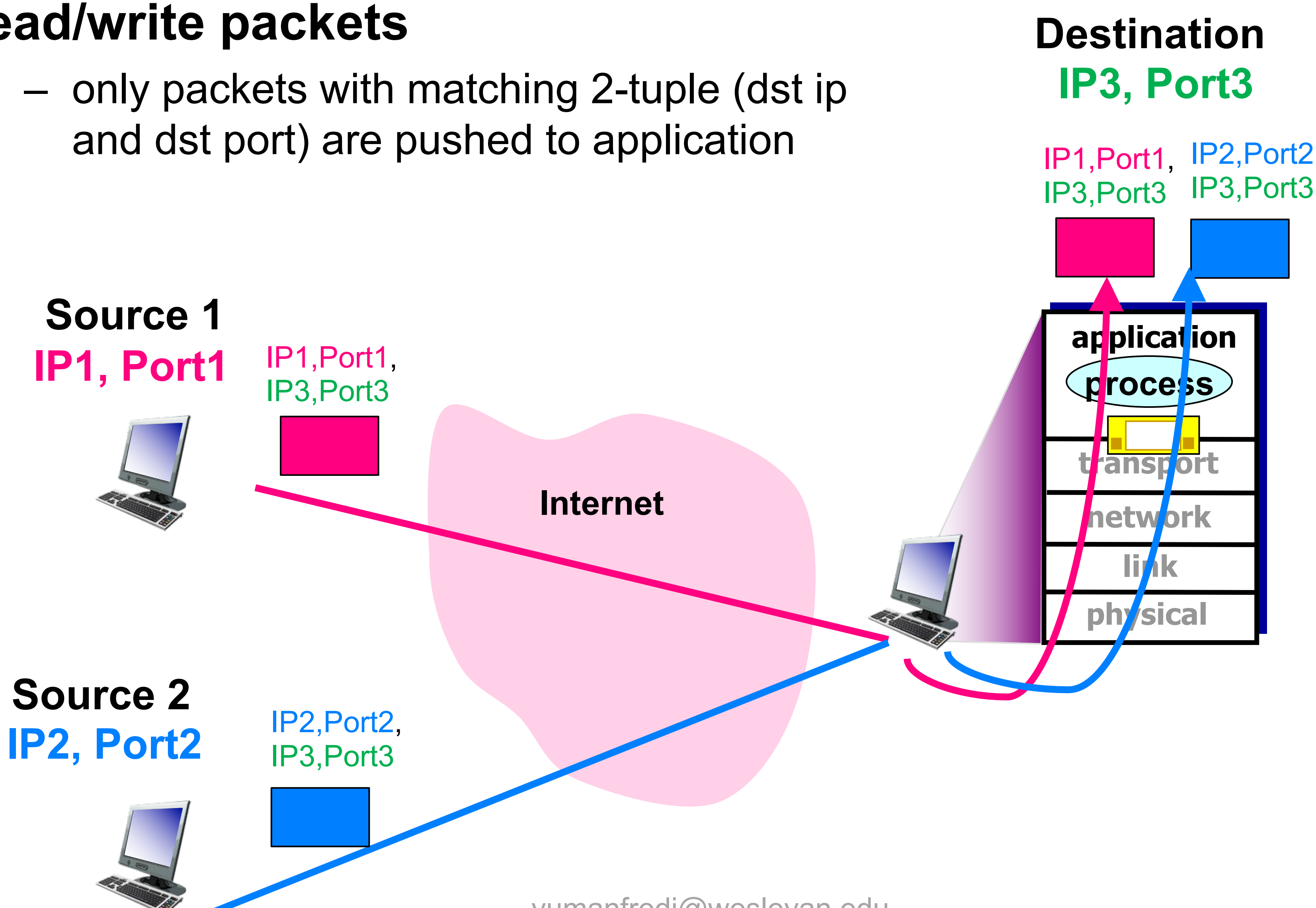
Connectionless

- **no handshaking** between UDP sender, receiver
- each UDP segment handled **independently** of others

UDP Socket

Read/write packets

- only packets with matching 2-tuple (dst ip and dst port) are pushed to application



Client/server socket interaction: UDP

Server running on serverIP

Create socket, bind it to port= x:
serverSocket = socket(AF_INET,SOCK_DGRAM)

Read datagram from serverSocket

Write reply to serverSocket specifying clientIP, port = y

Client running on clientIP

Create socket, bind it to port = y:
clientSocket = socket(AF_INET,SOCK_DGRAM)

Create datagram with serverIP and port=x; send datagram via clientSocket

Read datagram from clientSocket

Close clientSocket

vumanfredi@wesleyan.edu

Application example: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print ("The server is ready to receive")
while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        clientAddress)
```

- create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
- bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
- loop forever → while True:
- Read from UDP socket into message, getting client's address (client IP and port) → message, clientAddress = serverSocket.recvfrom(2048)
- send upper case string back to this client → serverSocket.sendto(modifiedMessage.encode(), clientAddress)

vumanfredi@wesleyan.edu

Application example: UDP client

Python UDPClient

include Python's socket library → `from socket import *`

create UDP socket for server → `clientSocket = socket(AF_INET, SOCK_DGRAM)`

get user keyboard input → `message = raw_input('Input lowercase sentence:')`

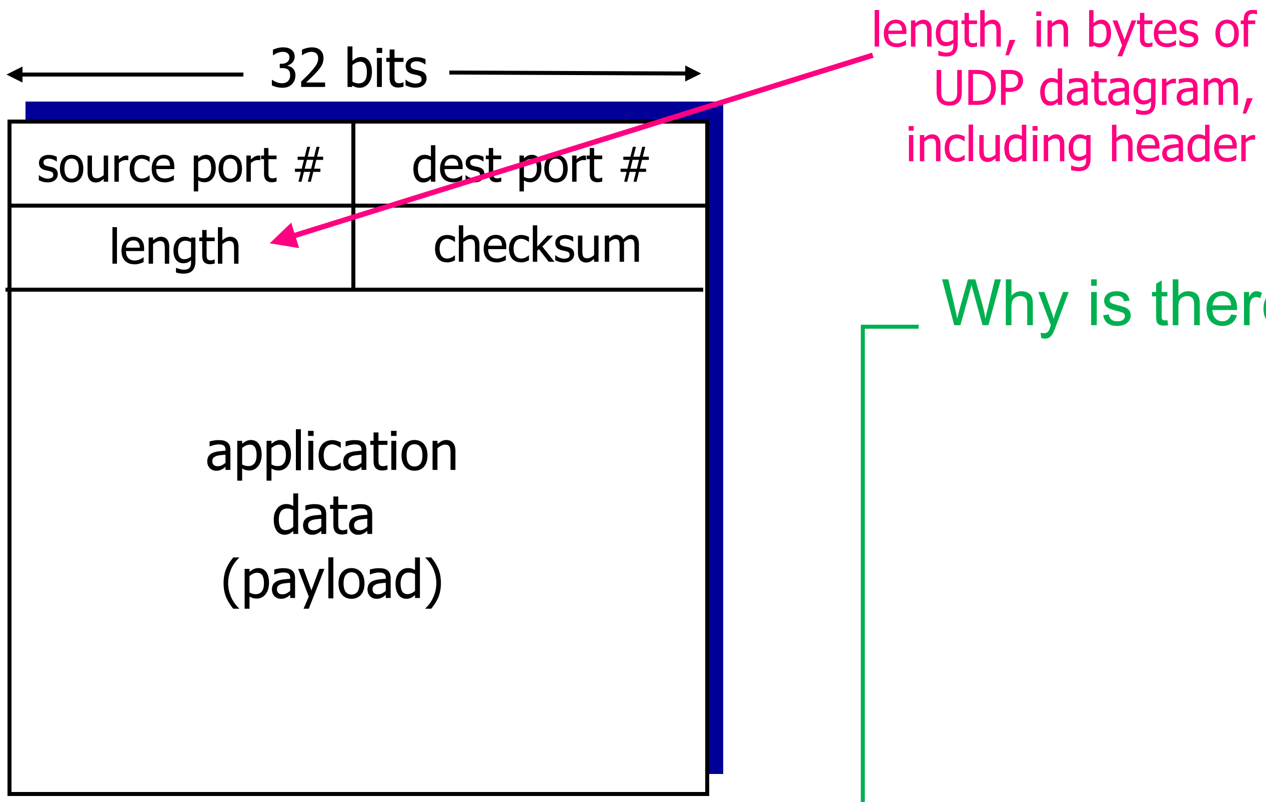
Attach server name, port to message; send into socket → `clientSocket.sendto(message.encode(), (serverName, serverPort))`

read reply characters from socket into string → `modifiedMessage, serverAddress = clientSocket.recvfrom(2048)`

print out received string and close socket → `print modifiedMessage.decode()`, `clientSocket.close()`

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(),
                              (serverName, serverPort))
modifiedMessage, serverAddress =
                              clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

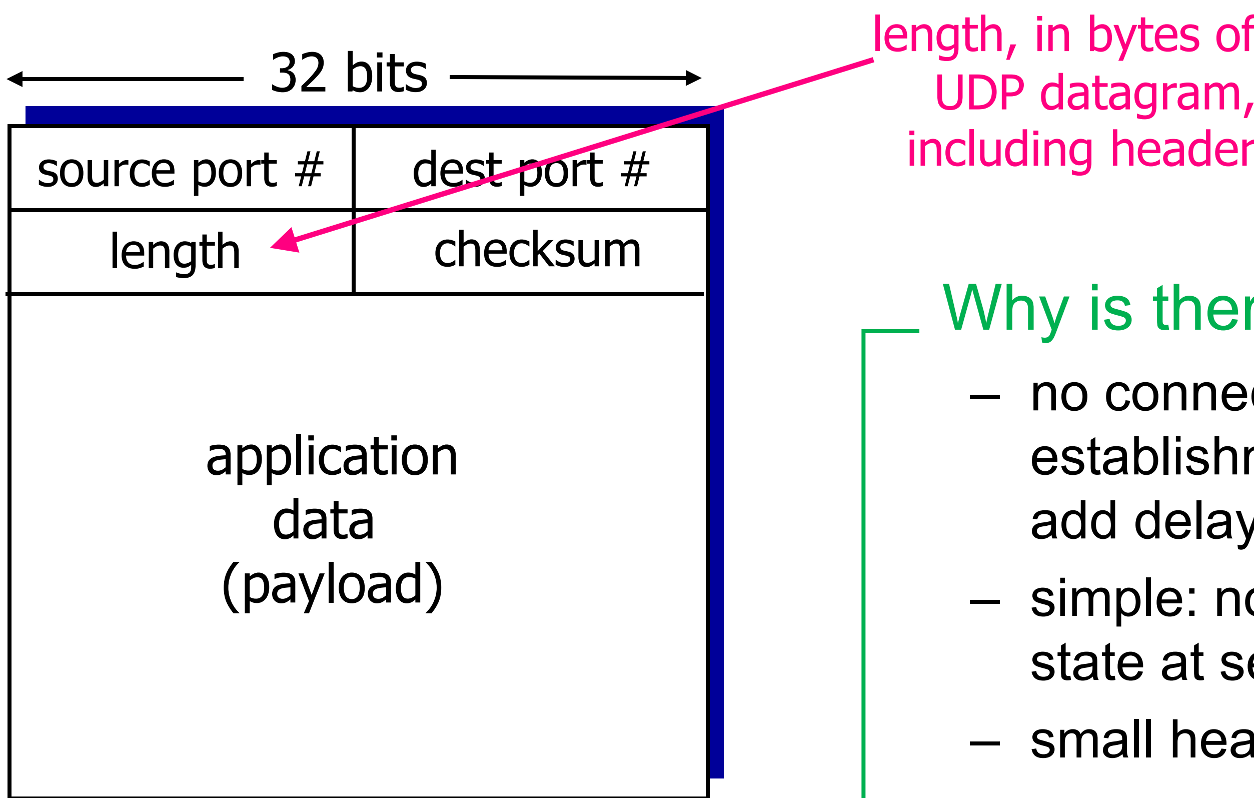
UDP datagram header



UDP datagram format

Why is there a UDP?

UDP datagram header



UDP datagram format

Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP error detection vs. recovery

Errors

- not just introduced during transmission over links
- can be introduced in memory, at router, at lower layer

UDP does not provide error recovery

- may drop datagram
- may pass datagram data to app with warning

UDP does provide error detection

- it's useful to know something damaged even if don't fix
- Q: How?
    - Checksum

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted datagram

Sender

1. Views datagram contents, including header fields and user data, as sequence of 16-bit integers
   - skip checksum field
2. Computes checksum
   - adds 16-bit integers together using 1s complement arithmetic and then takes 1s complement of result
3. Puts checksum value in UDP checksum field

Receiver

1. Computes its own checksum over datagram including checksum in UDP header
2. Result should equal all 0s if no errors
   - NO: error detected
   - YES: no error detected
   - Q: can there still be errors?

Internet checksum example

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
```

Summing these should give all 1s, flip bits should give 0

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

Q: Why 1s complement? Why check for 0s?

- for efficiency: computed very fast in hardware
- independent of machine endianness

Looking at UDP in Wireshark

```
▶ Frame 237: 143 bytes on wire (1144 bits), 143 bytes captured (1144 bits) on ir
▶ Ethernet II, Src: JuniperN_1e:18:01 (3c:8a:b0:1e:18:01), Dst: 78:4f:43:73:43:2
▶ Internet Protocol Version 4, Src: intdns.wesleyan.edu (129.133.52.12), Dst: vm
▼ User Datagram Protocol, Src Port: 53 (53), Dst Port: 57332 (57332)
    Source Port: 53
    Destination Port: 57332
    Length: 109
  ▼ Checksum: 0x0f73 [validation disabled]
      [Good Checksum: False]
      [Bad Checksum: False]
    [Stream index: 1]
▶ Domain Name System (response)

0000  78 4f 43 73 43 26 3c 8a  b0 1e 18 01 08 00 45 00   xOCsC&<. ......E.
0010  00 81 87 f4 00 00 3e 11  01 b3 81 85 34 0c 81 85   ......>. ....4...
0020  bb ae 00 35 df f4 00 6d  0f 73 e6 72 81 80 00 01   ...5...m .s.r....
0030  00 01 00 00 00 00 03 32  32 37 03 31 39 30 02 33   .......2 27.190.3
0040  33 02 31 33 07 69 6e 2d  61 64 64 72 04 61 72 70   3.13.in- addr.arp
0050  61 00 00 0c 00 01 c0 0c  00 0c 00 01 00 01 51 8d   a....... ......Q.
0060  00 2d 14 73 65 72 76 65  72 2d 31 33 2d 33 33 2d   .-.serve r-13-33-
0070  31 39 30 2d 32 32 37 05  62 6f 73 35 30 01 72 0a   190-227. bos50.r.
0080  63 6c 6f 75 64 66 72 6f  6e 74 03 6e 65 74 00      cloudfro nt.net.
```