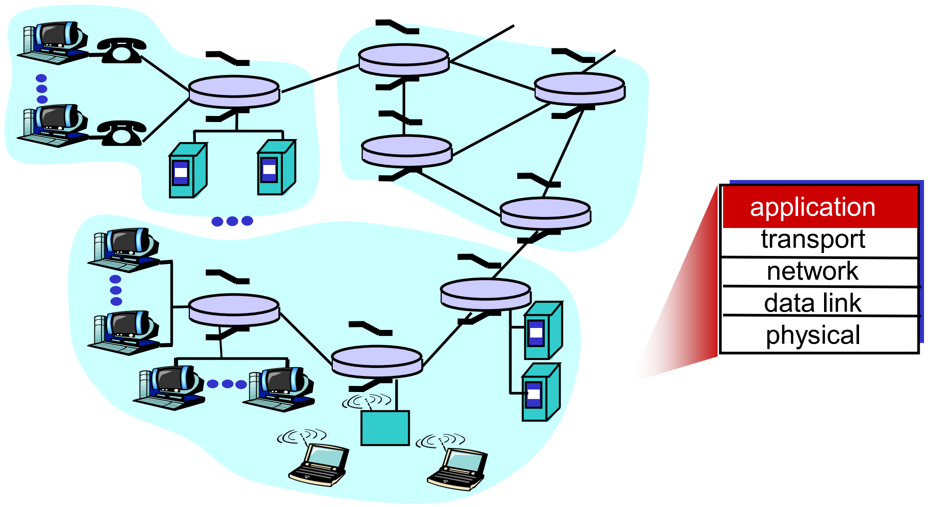
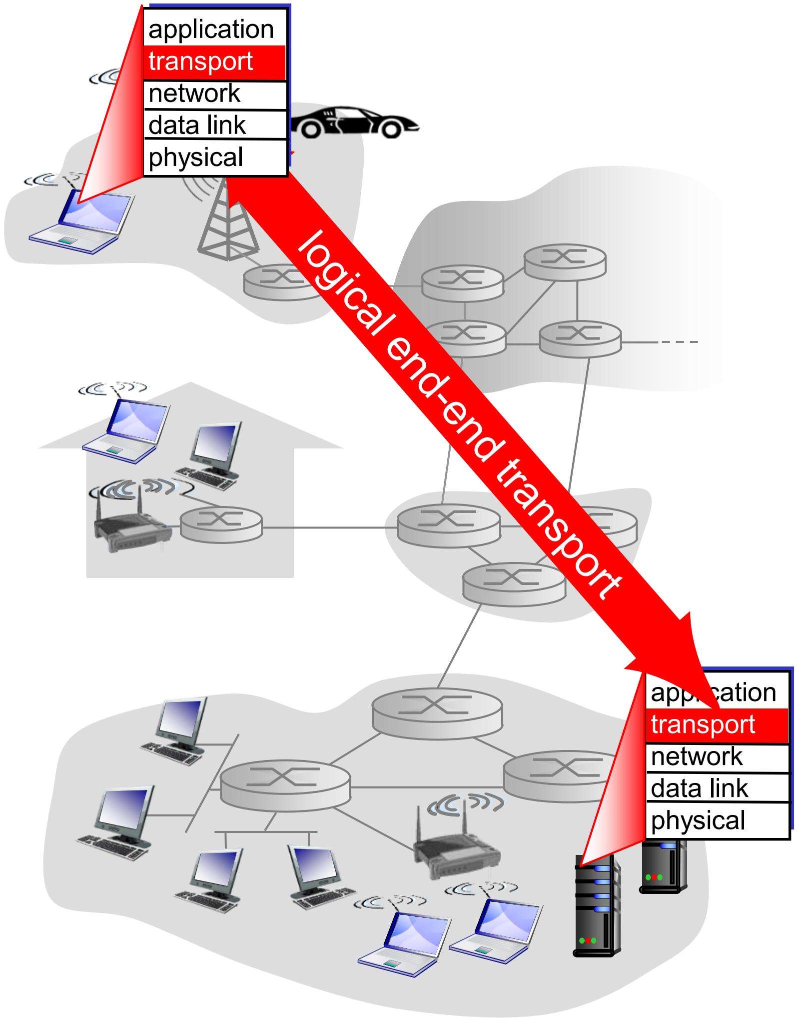
CS 4390
Computer Networks



Transport Layer – Overview and UDP

Transport Services and Protocols

- provide *logical communication* between app processes running on different hosts
  - Apps require different *services*
- transport protocols run in end systems
  - *send side*: breaks app messages into *segments*, passes to network layer
  - *receive side*: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Challenges for Transport Layer

- Develop ***algorithms*** and ***techniques*** for transport protocols that address *transport requirements* imposed by application programs for
  - Datagram service – connection-less
  - Byte stream service – connection-oriented
- Take into account *limitations* of underlying layers (network and below)
  - Network layer services and protocol limitations
  - Underlying physical data transmission network limitations, e.g. higher bit-error rate in wireless channels

Transport Service Requirements

- Supports multiple application processes on a host
- Supports arbitrarily large messages
- Guarantees message delivery
- Guarantees no errors in delivered messages
- Delivers at most one copy of each message
- Delivers messages in the same order they were sent
- Ensures that messages not overwhelm the receiver
- Delivers messages with minimum delay
- …

Limitations of Underlying Layers

- Bit-errors in messages
  - Vary depending on physical channels, can be high in wireless
- Messages are lost
  - e.g. discarded by routers due to buffer overflow
- Messages are delivered out-of-order
  - Packet switching: packets may traverse different paths from sender to receiver
- Duplicated messages
  - Duplicated copies of the same messages may be received (due to retransmission)
- Limit messages size
  - Ethernet frames: 1.5 KB, IP: 64 KB
- Delayed delivery due to congestion
- ...

Transport Layer Topics

our goals:

- understand ***principles*** behind transport layer services:
  - multiplexing, de-multiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport v.s. Network Layer

- *network layer:* logical communication between ***hosts***
- *transport layer:* logical communication between ***processes***
  - relies on, enhances, network layer services

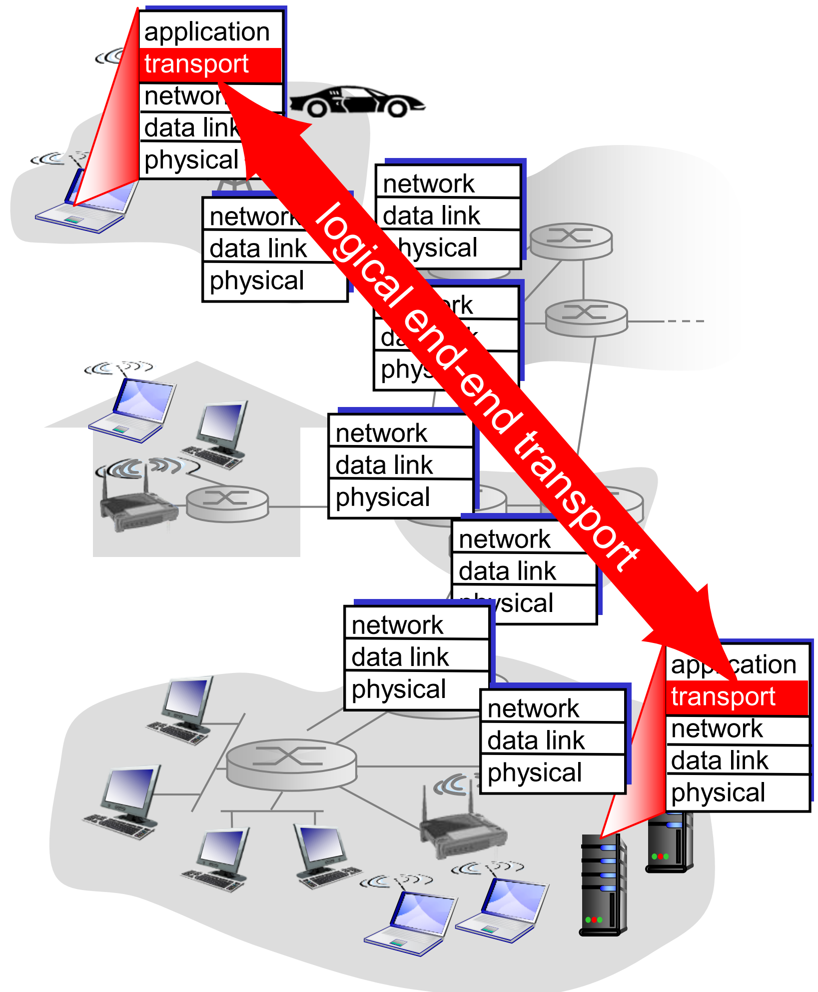
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

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

Internet Transport-layer Protocols

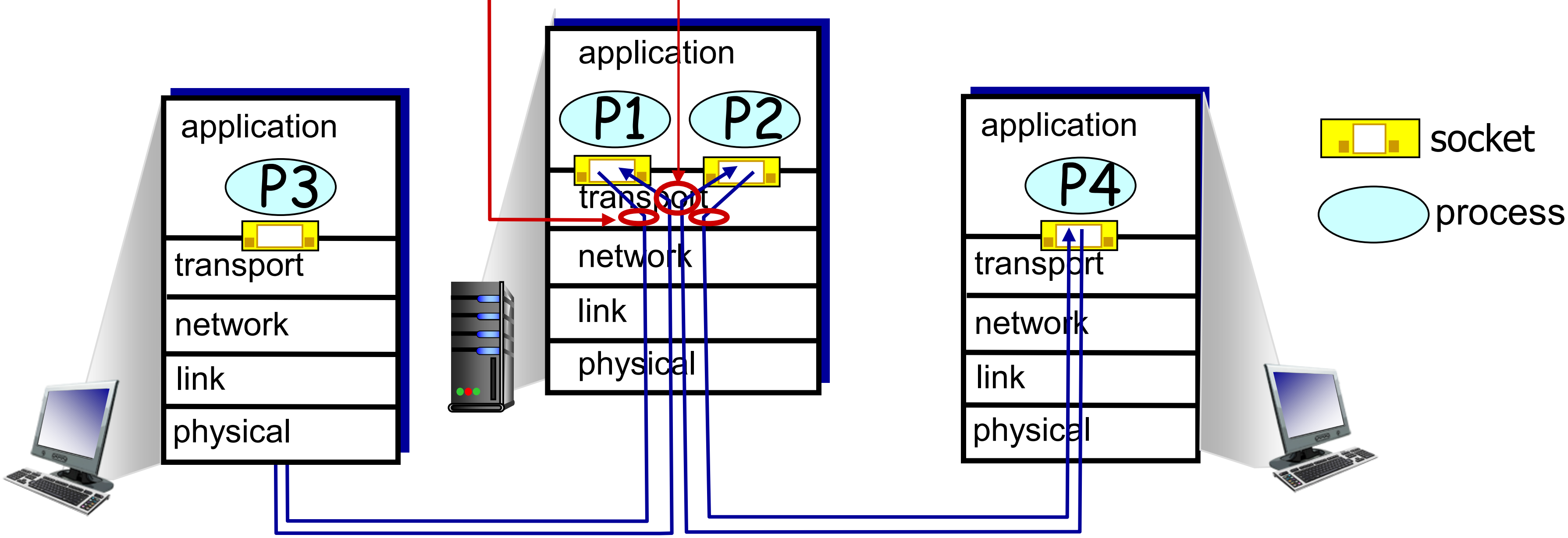
- reliable, in-order delivery: **TCP**
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: **UDP**
  - no-frills extension of "*best-effort*" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Multiplexing & Demultiplexing

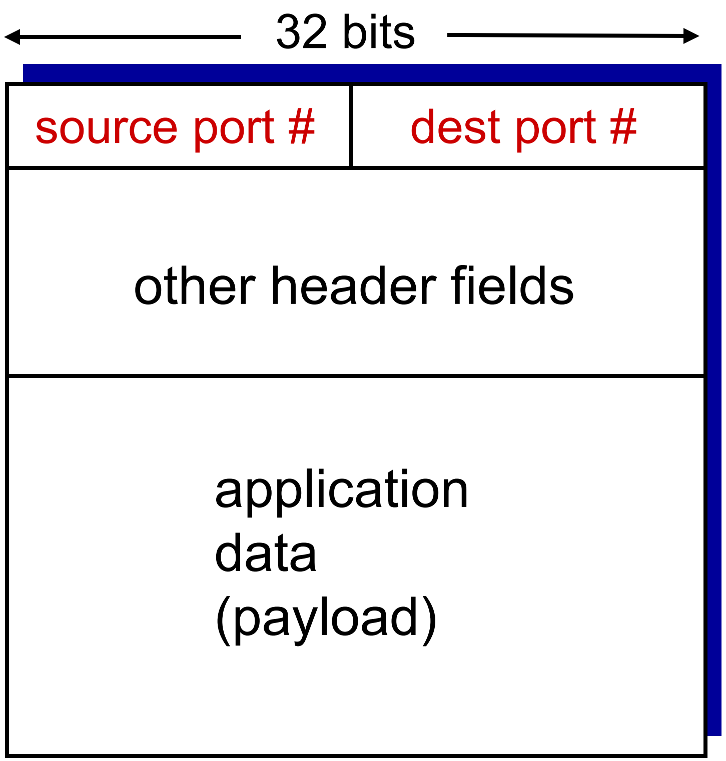
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

De-multiplexing at receiver:
use header info to deliver received segments to correct socket



How De-multiplexing Works?

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, *destination port number*
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless De-multiplexing

- *recall:* created socket has host-local port #:

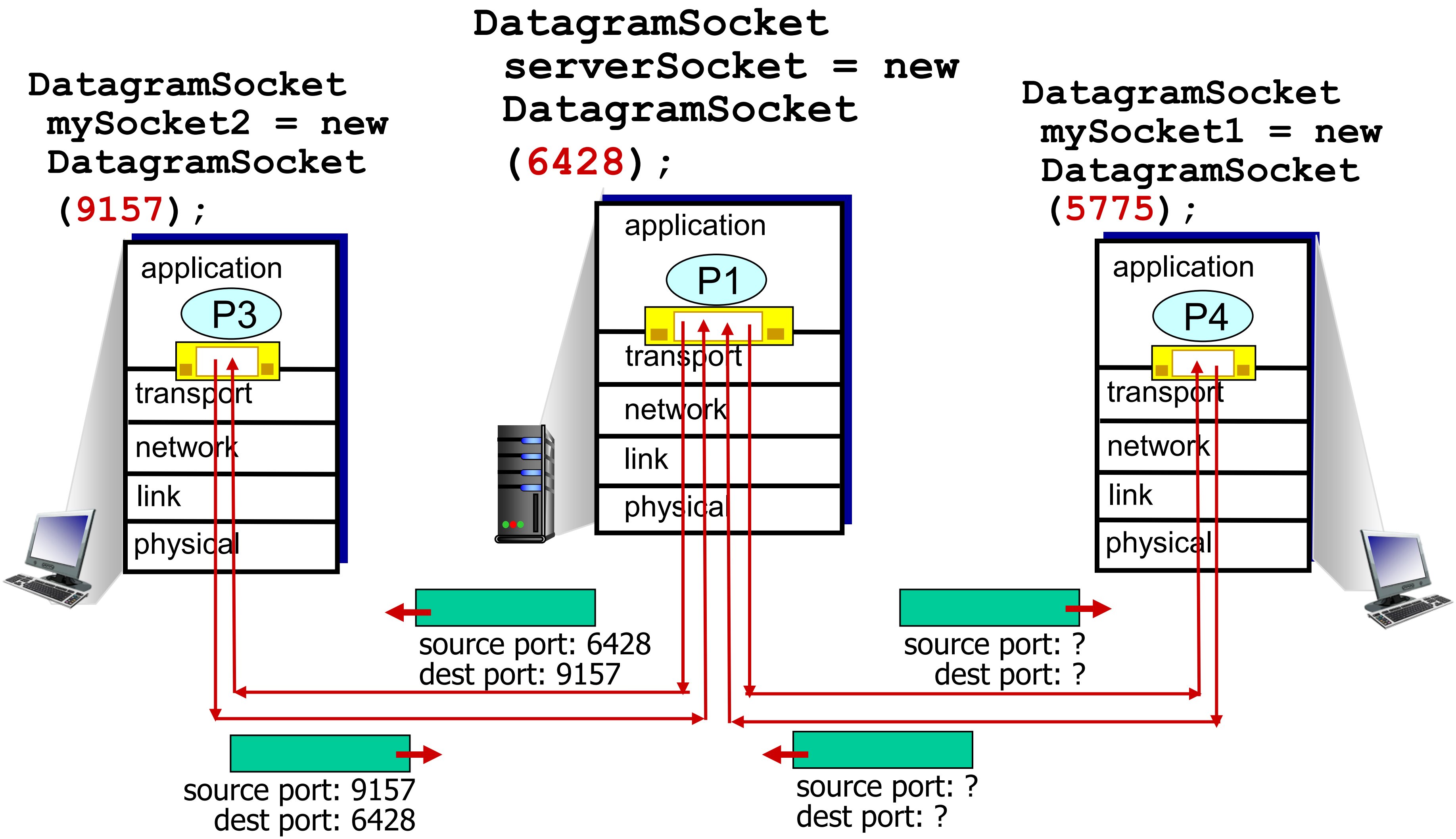
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

Connectionless De-mux – Example

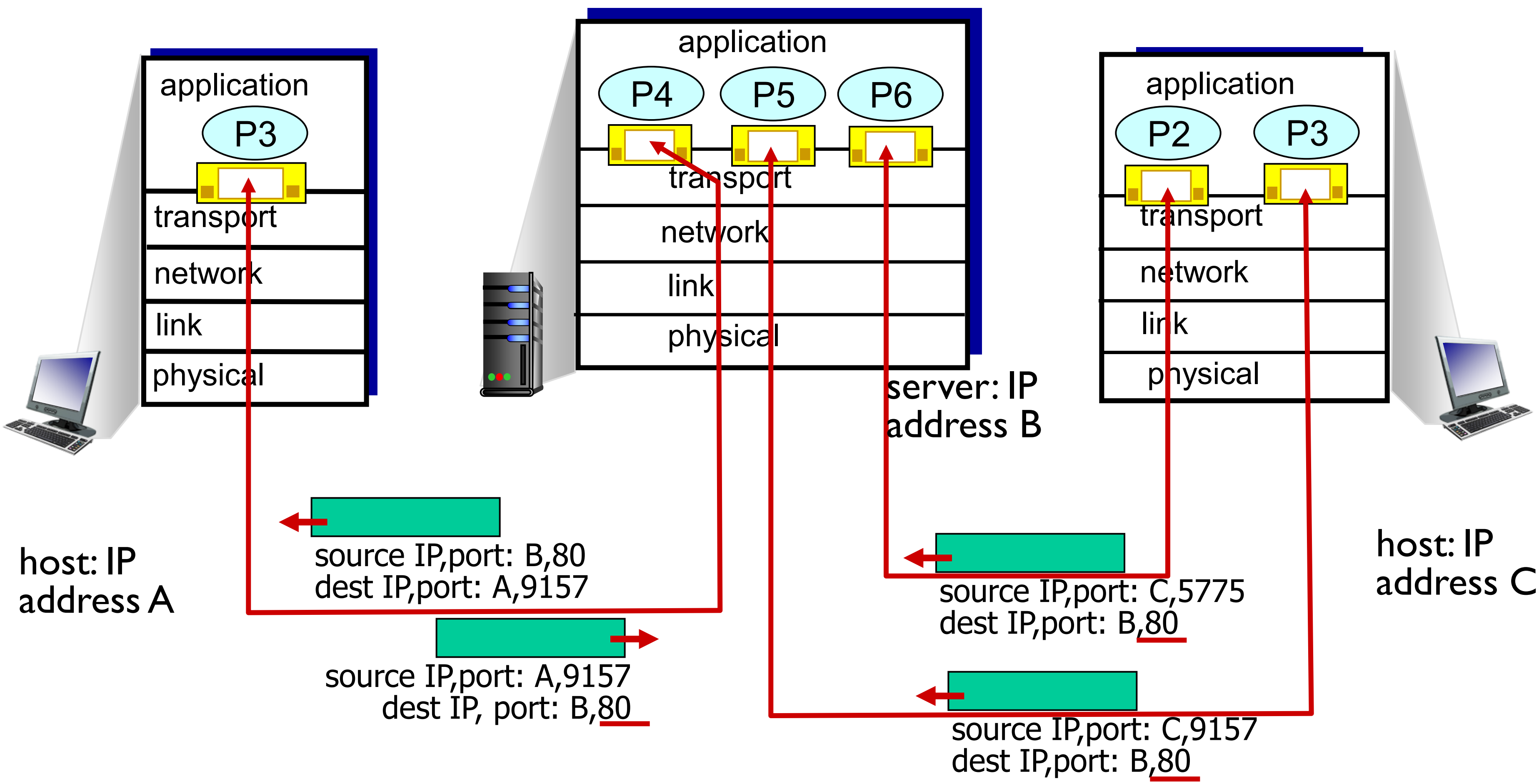


Connection-oriented De-mux

- TCP socket identified by a 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- De-mux: receiver uses all four values to direct segment to appropriate socket

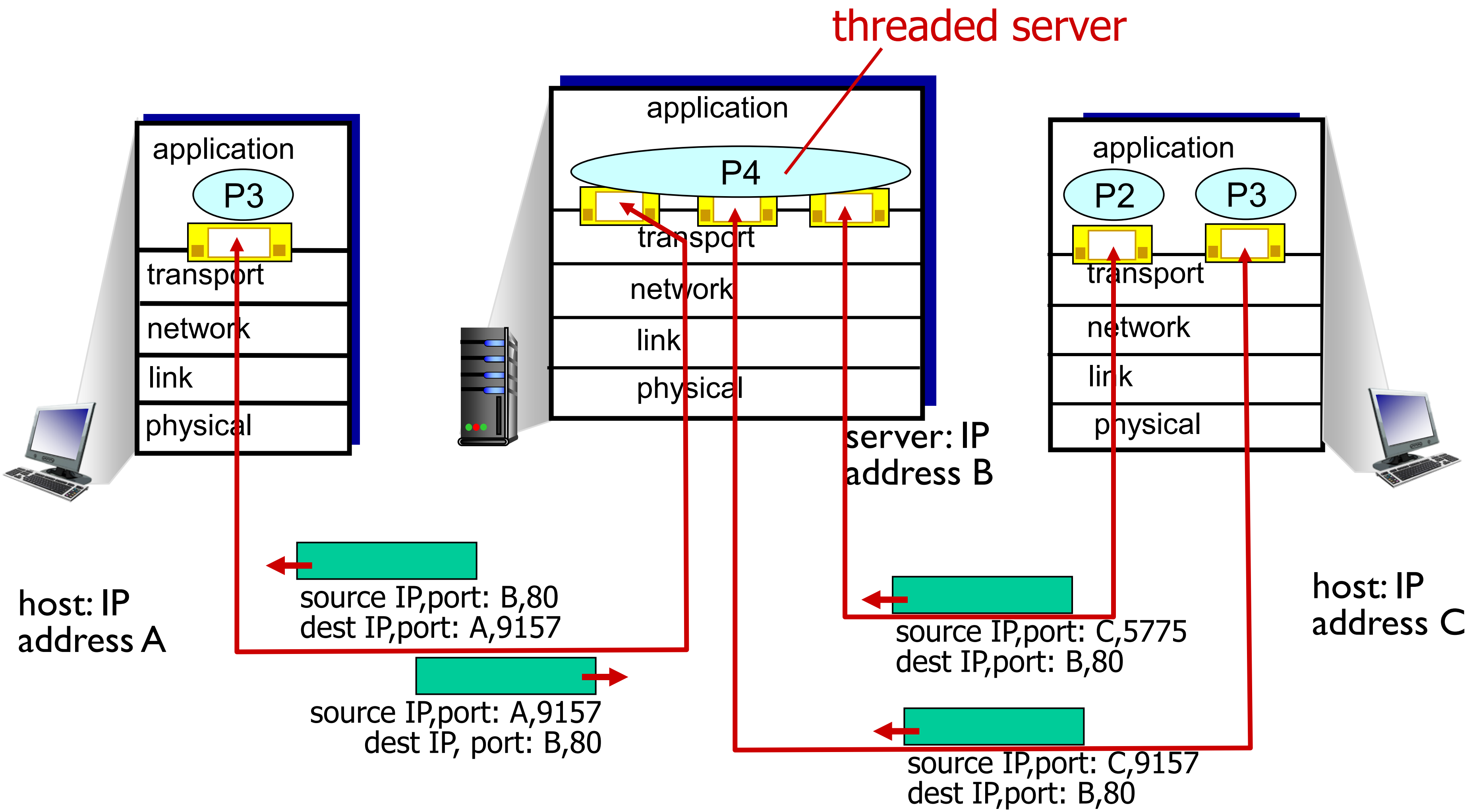
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented De-mux – Example



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

Connection-oriented De-mux – Example



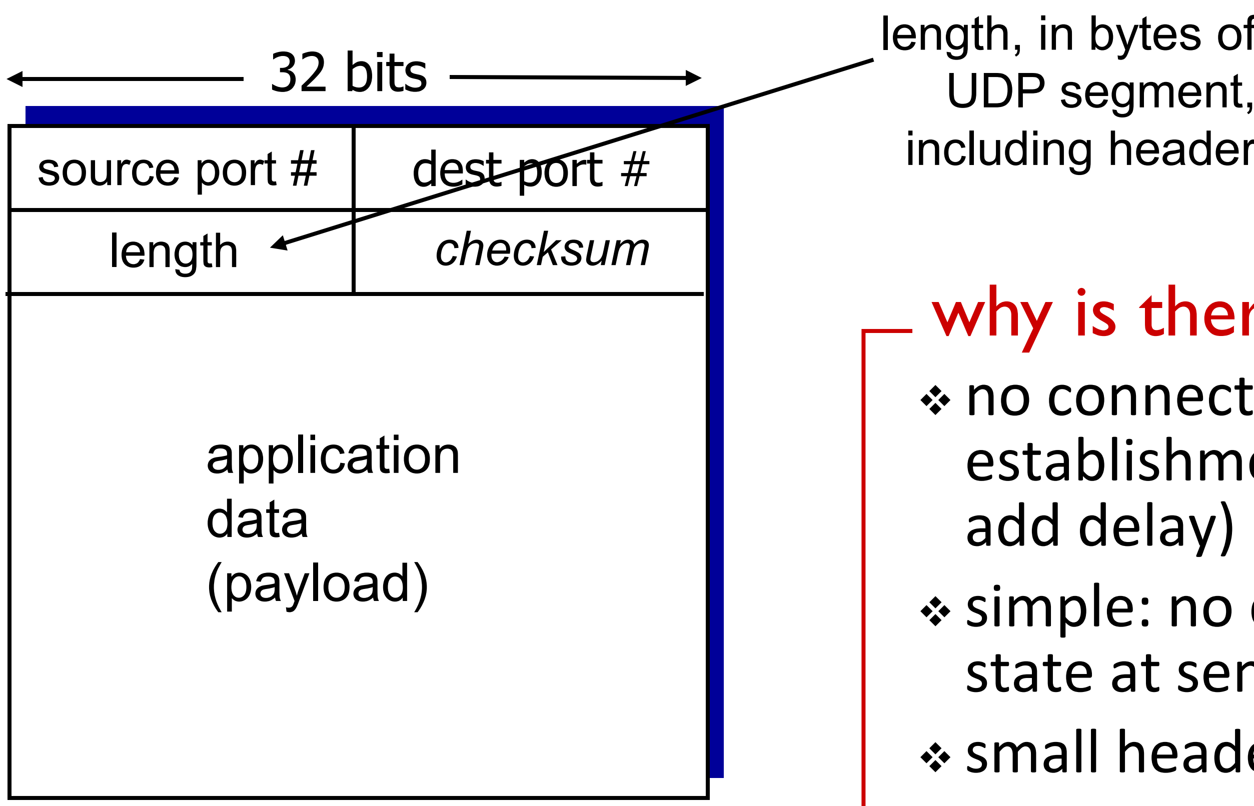
UDP: User Datagram Protocol [RFC 768]

- "*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
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

What are two main functions does UDP add to IP?

UDP: Segment Header



UDP segment 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 Checksum

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

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (1-complement sum) of segment contents
- sender puts checksum value into UDP checksum field
  - can be all 0s: no checksum

receiver:

(If the received checksum is not all 0s)

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. (*But maybe errors nonetheless?* More later)

  ….

What if the sender wants to add checksum and the calculated checksum is all 0s?

Internet Checksum Algorithm

1. Break the data (e.g. the UPD message, including header) as a series of 16-bit integers
2. Add them together
3. If there is a carry-over, add 1 to the result
4. Take 1-complement of the result (i.e. reverting all bits) to get the checksum for the data

Internet Checksum – Example

Message is 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
Carry-over	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
1-compliment 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