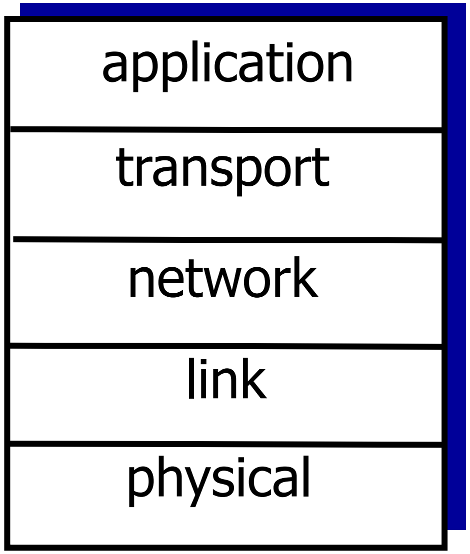
Transport layer and UDP

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley

Overview

- Principles underlying transport layer
  - Multiplexing/demultiplexing
  - Detecting errors
  - Reliable delivery
  - Flow control
  - Congestion control
- Major transport layer protocols:
  - User Datagram Protocol (UDP)
    - Simple unreliable message delivery
  - Transmission Control Protocol (TCP)
    - Reliable bidirectional stream of bytes



Chapter 3: Transport Layer

Goals:

- Understand principles behind transport layer services:
  - Multiplexing, demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Chapter 3 outline

3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

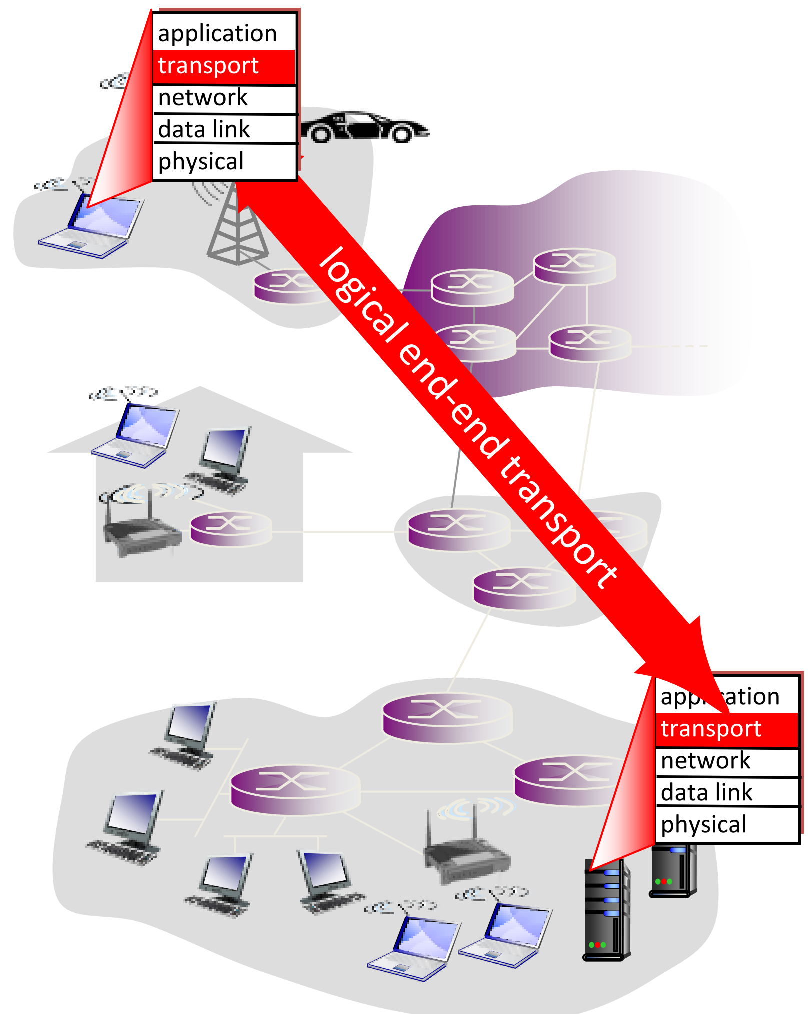
3.5 Connection-oriented transport: TCP
- Segment structure
- Reliable data transfer
- Flow control
- Connection management

3.6 Principles of congestion control

3.7 TCP congestion control

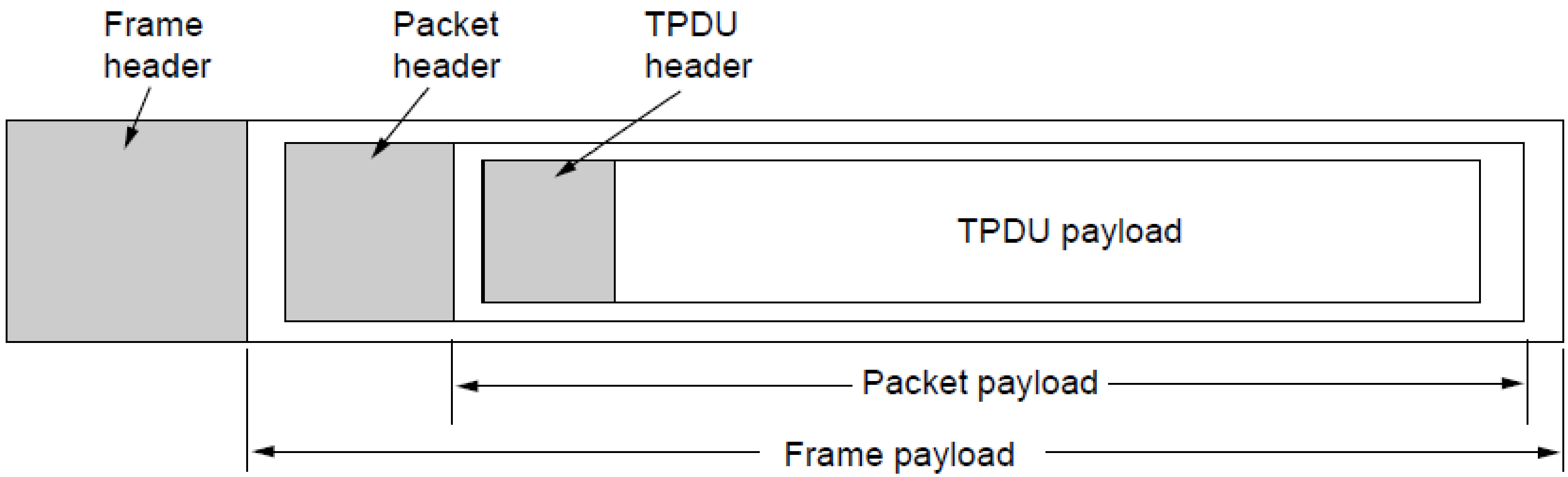
Transport services and protocols

- Provide *logical communication* between app processes running on different hosts
- Transport protocols run in end systems
  - Send side: breaks app messages into *segments*, passes to network layer
  - Recv side: reassembles segments into messages, passes to app layer
- More than one transport protocol available to apps
  - Internet: TCP and UDP

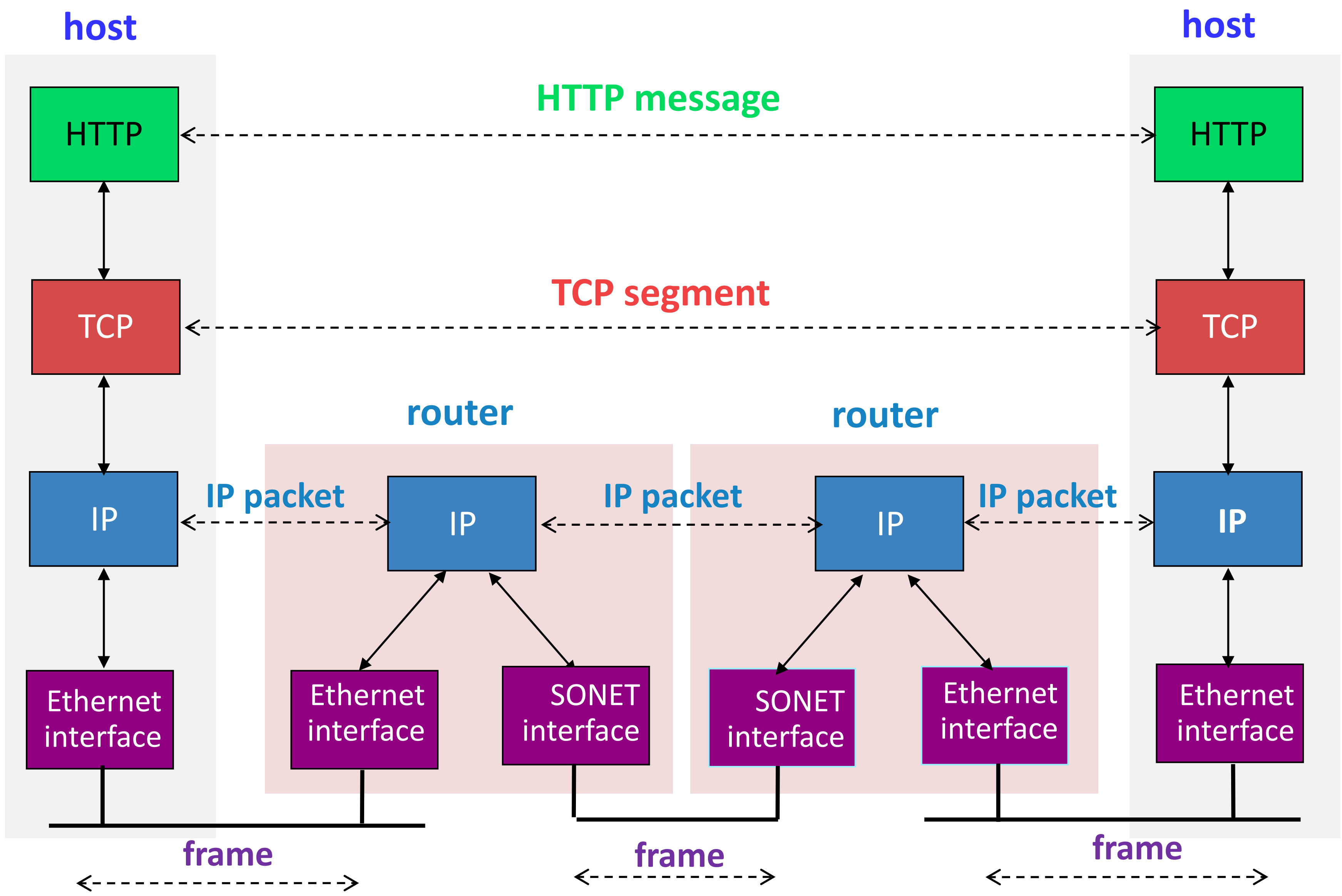


Segments

- Segment
  - Message sent from one transport entity to another transport entity
  - Term used by TCP, UDP, other Internet protocols
  - aka TPDU (Transport Protocol Data Unit)



Internet layering model

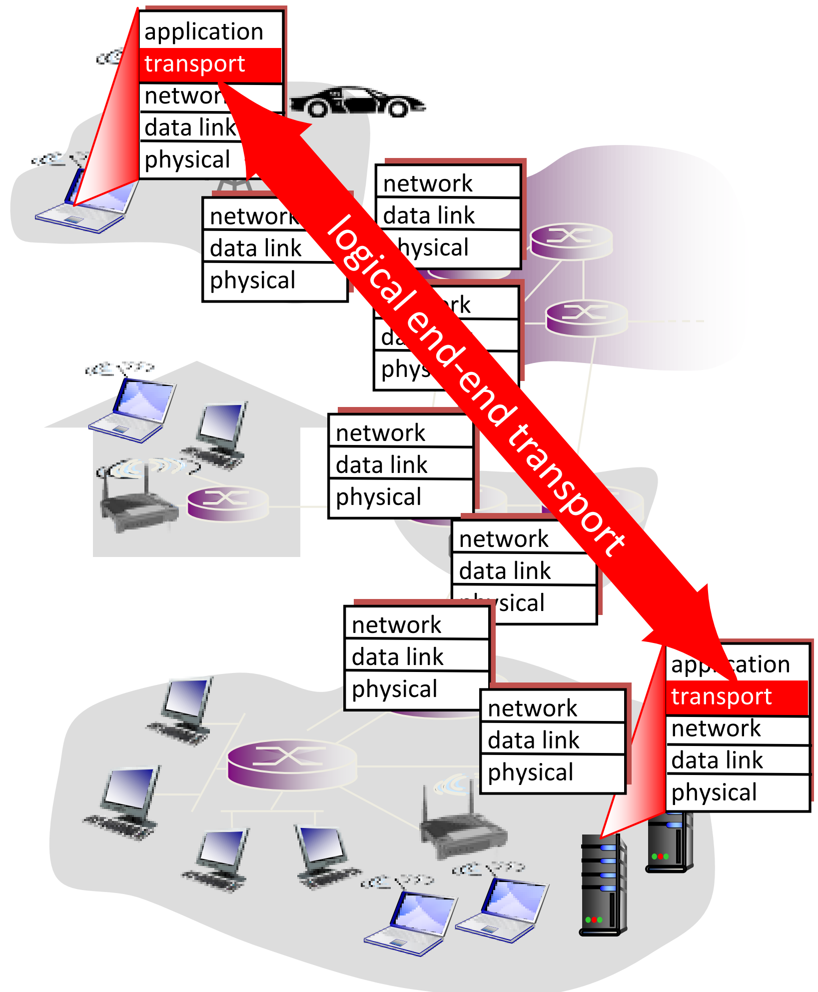


Transport layer challenges

- Running on best-effort network:
  - Messages may be dropped
  - Messages may be reordered
  - Duplicate messages may be delivered
  - Messages have some finite size
  - Messages may arrive after long delay
- Sender must not overrun receiver
- Network may be congested
- Hosts must support multiple applications

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



Transport layer

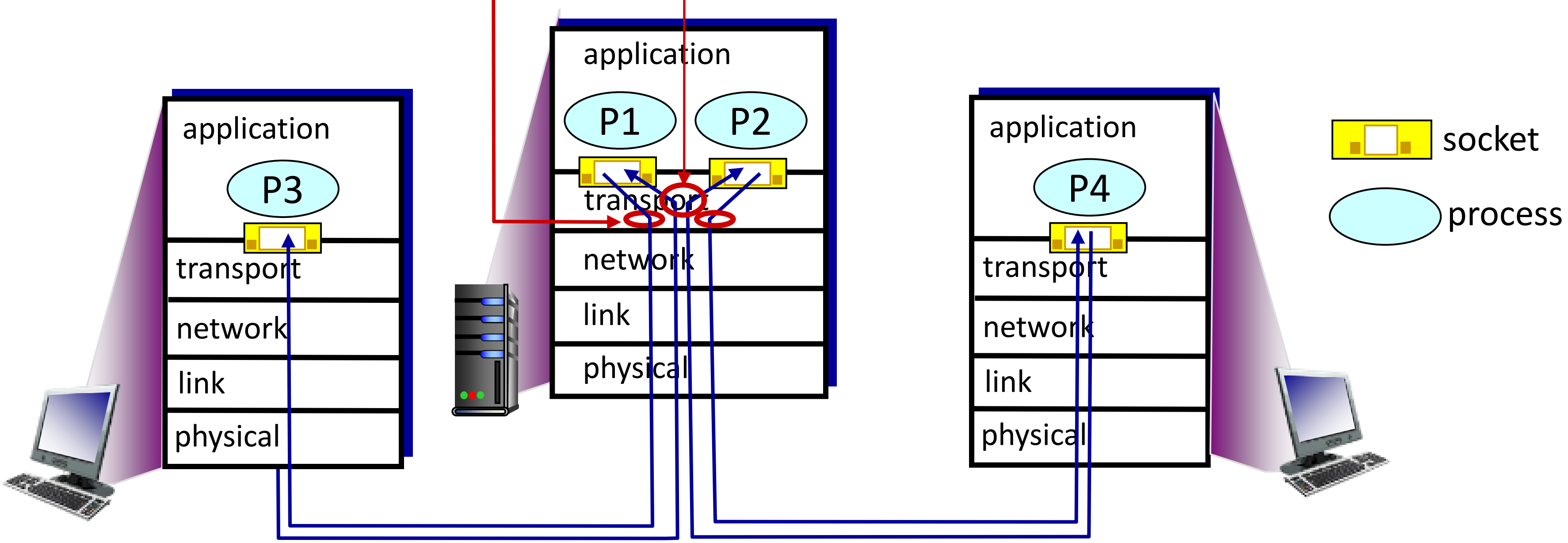
- Goal: End-to-end data transfer
  - Just getting to host machine isn't enough
  - Deliver data from process on sending host to correct process on receiving host
- Solution: OS demultiplexes to correct process
  - Port number, an abstract locater
  - OS demuxes combining with other info

UDP	<port, host>
TCP	<source port, source IP, dest port, dest IP>

Multiplexing/demultiplexing

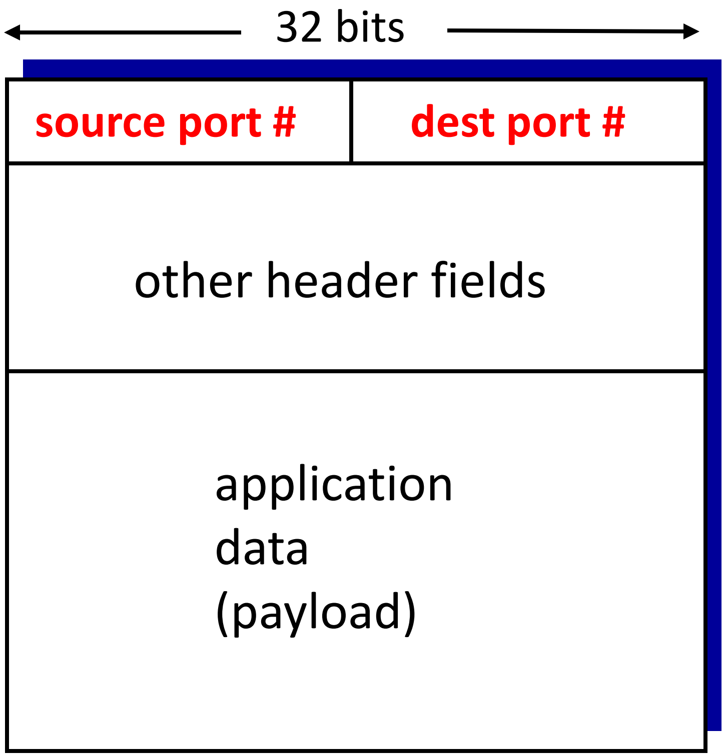
Multiplexing at sender:
Handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing at receiver:
Use header info to deliver received segments to correct socket



How demultiplexing 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 demultiplexing

- *Recall:* created socket can specify host-local port #:

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

- *Recall:* when creating datagram to sent into UDP socket, must specify:
  - Destination IP
  - Destination port #

- When host receives UDP segment:
  - Checks destination port # in segment
  - Directs UDP segment to socket with that port #

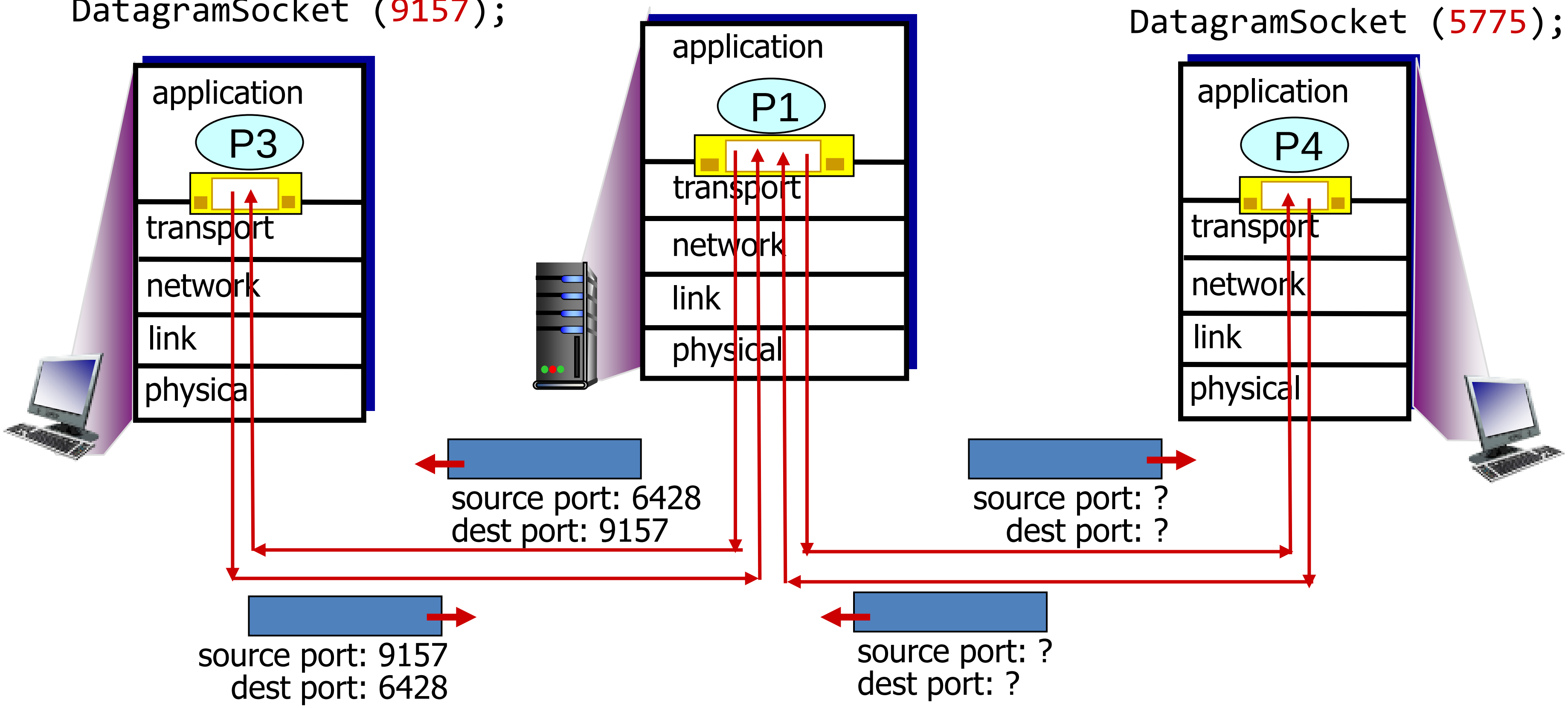
IP datagrams with *same destination port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

Connectionless demux: example

```
DatagramSocket
mySocket2 = new
DatagramSocket (9157);
```

```
DatagramSocket
serverSocket = new
DatagramSocket (6428);
```

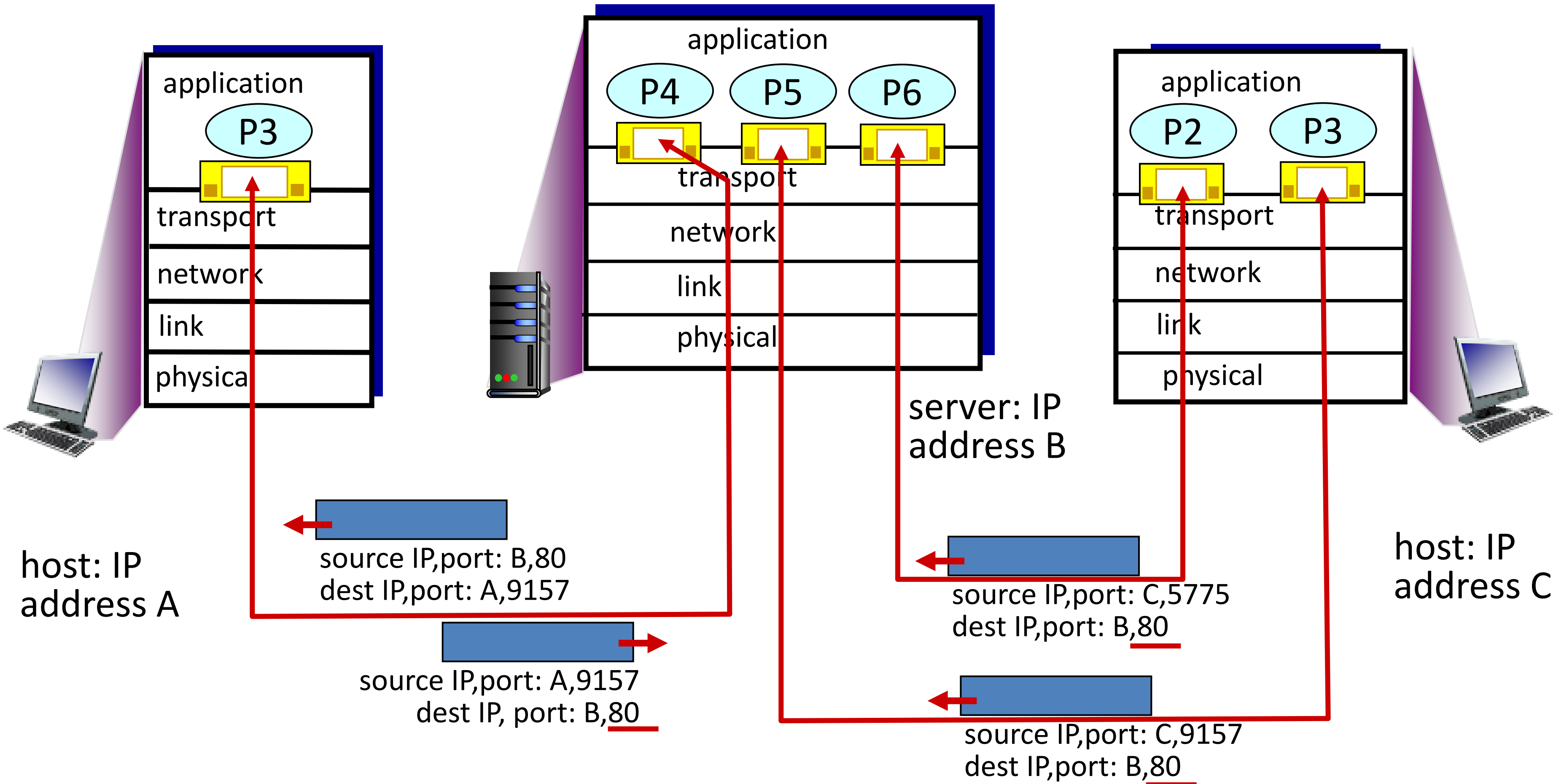
```
DatagramSocket
mySocket1 = new
DatagramSocket (5775);
```



Connection-oriented demux

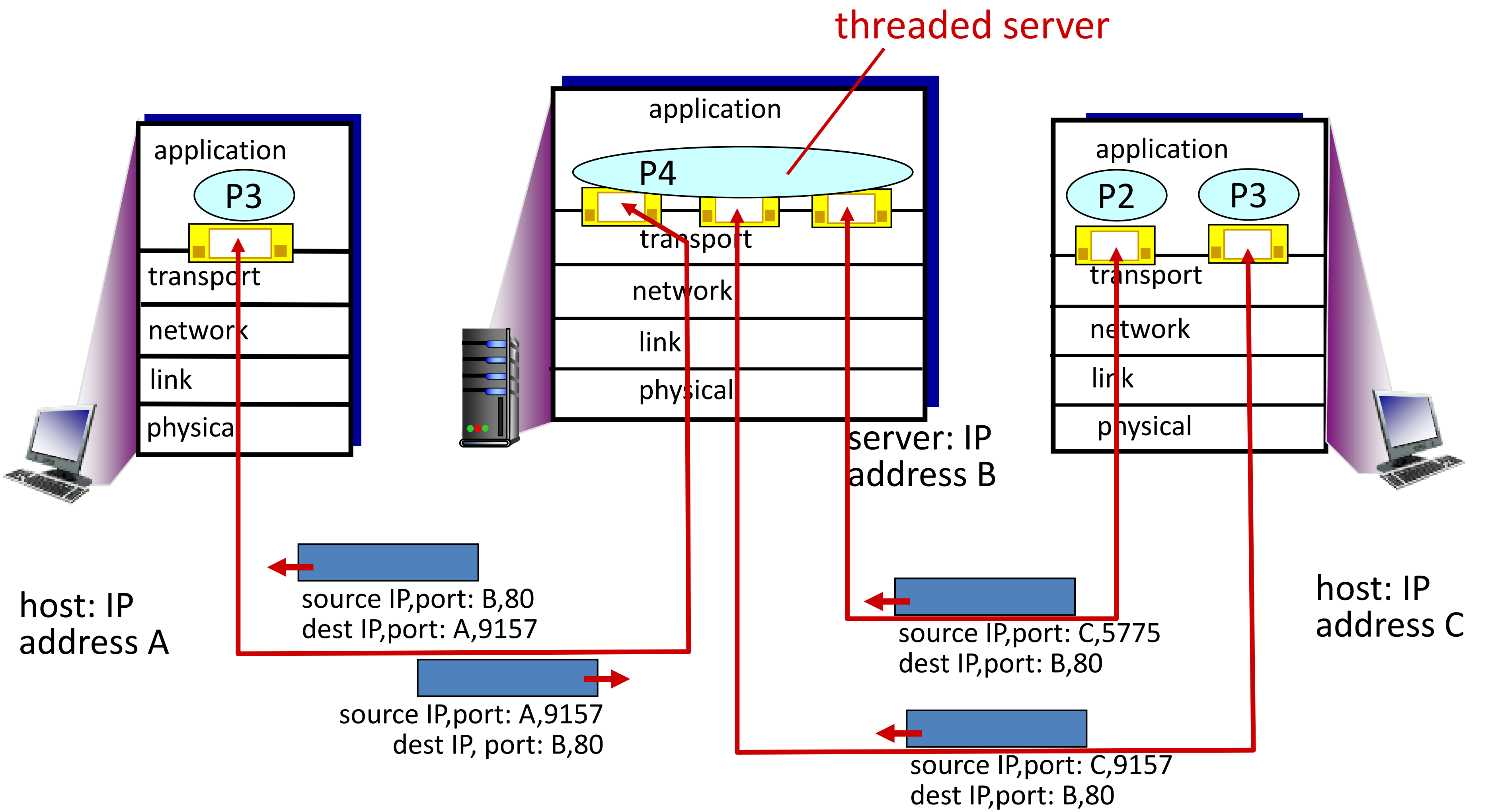
- TCP socket identified by 4-tuple:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Demux: receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - Each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



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

Connection-oriented demux: example



Why use UDP?

- Provides:
  - Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Precise control of when data is sent
    - As soon as app writes to socket, UDP packages and sends
  - No delay establishing a connection
  - No connection state, scales to more clients
  - Small packet overhead, header only 8 bytes long
- Does not provide:
  - Flow control
  - Congestion control
  - Retransmission on error

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 (one's complement sum) of segment contents
- Sender puts checksum value into UDP checksum field

Receiver:

- 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 ….

Internet checksum: example

example: add two 16-bit integers

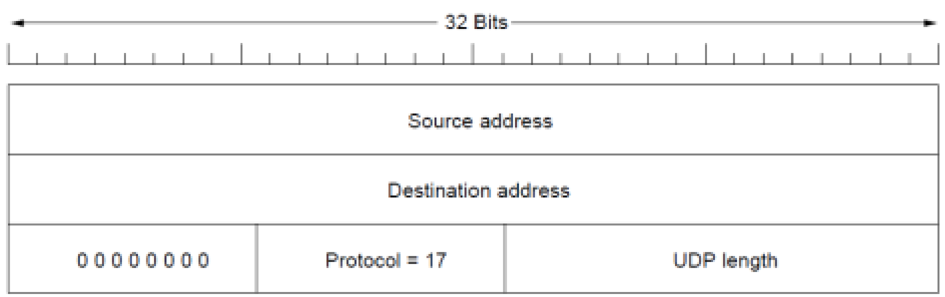
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

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

UDP checksums

- UDP checksum
  - Add up 16-bit words in one's complement
  - Take one's complement of the sum
  - Done on UDP header, data, IP pseudoheader
    - Helps detect misdelivered packets
    - Violates layers, looking into network layer

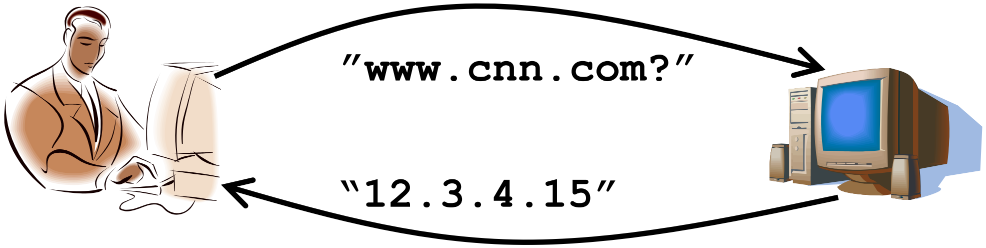
UDP header



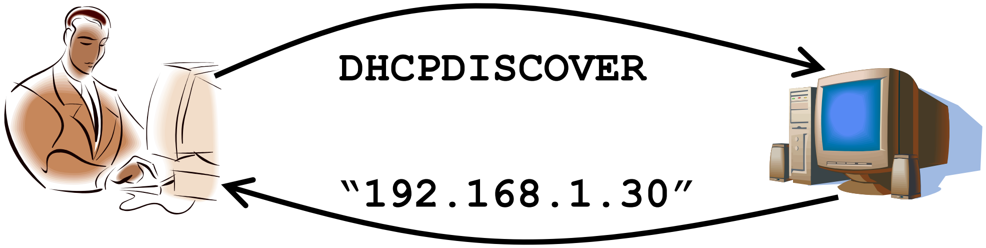
IP pseudoheader

Type of UDP apps, part 1/3

- Simple query protocols
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed
  - e.g. DNS, UDP port 53

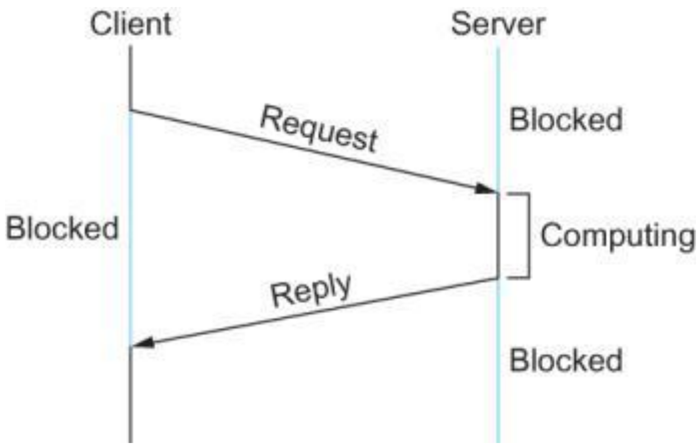


  - e.g. DHCP, UDP port 67/68



Type of UDP apps, part 2/3

- Request/reply style interaction
  - Client sends request to server
    - Blocks while waiting for reply
  - Server responds with reply
  - Must deal with:
    - Identify process that can handle request
    - Possible loss of request or reply
    - Correlate request with reply



Request/reply example

- Remote Procedure Call (RPC)
  - Request/reply paradigm over UDP
  - Allow programs to call procedures located on a remote host
  - Invisible to the application programmer
    - Client code blocks while request made and response waited for from remote host
  - Object-oriented languages:
    - Remote Method Invocation(RMI), e.g. Java RMI

Type of UDP apps, part 3/3

- Multimedia streaming
  - e.g. Voice over IP, video conferencing
  - Time is of the essence
    - By time packet is retransmitted, it's too late!
    - Interactive applications:
      - Human-to-human interaction
      - e.g. conference, first-person shooters
    - Streaming applications:
      - Computer-to-human interaction
      - e.g. Netflix, Spotify

Summary

- Transport layer
  - Providing end-to-end process communication
    - Port numbers allow multiple processes per host
  - Provide reliable transport on best-effort network
- User Datagram Protocol (UDP)
  - Lightweight protocol running on top of IP
  - Three typical classes of applications:
    - Simple queries (DNS, DHCP)
    - Request/reply semantics (RPC)
    - Real-time data (Skype)