DATA COMMUNICATOIN NETWORKING

Instructor: Ouldooz Baghban Karimi

Course Book: Computer Networking, A Top-Down Approach
By: Kurose, Ross

Course Overview

- **Basics of Computer Networks**
  - Internet & Protocol Stack
  - Application Layer
  - Transport Layer
  - Network Layer
  - Data Link Layer

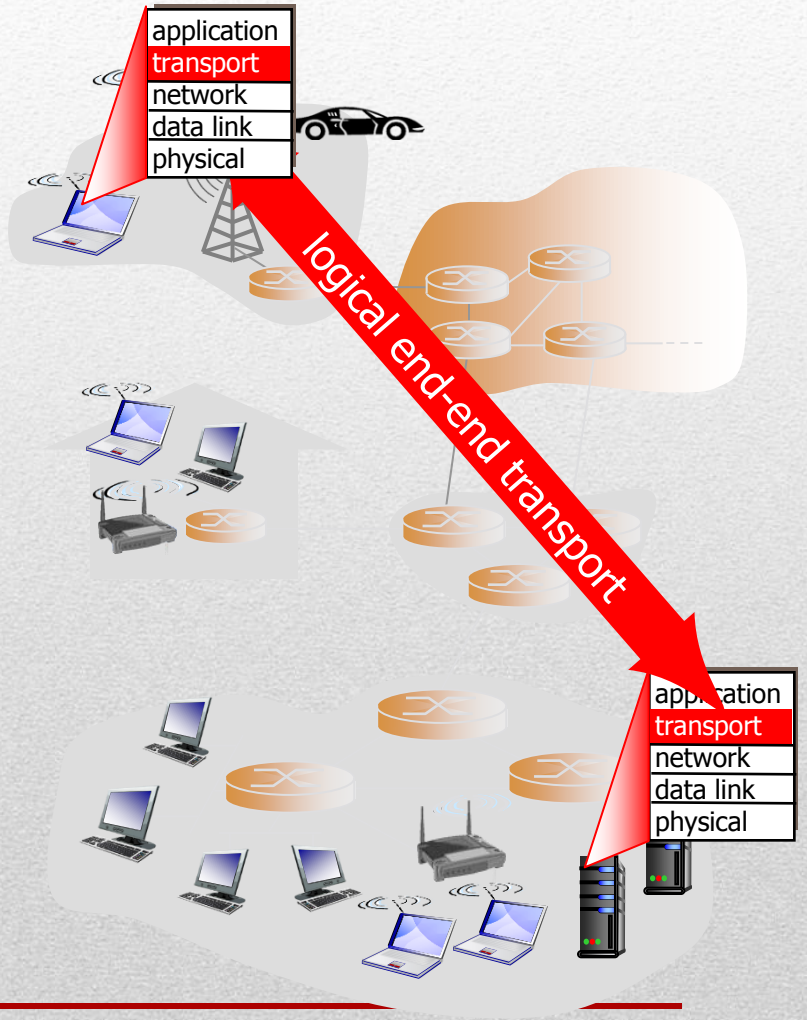
- **Advanced Topics**
  - Case Studies of Computer Networks
  - Internet Applications
  - Network Management
  - Network Security

Transport Layer

- Transport Layer Services
- Multiplexing & Demultiplexing
- UDP
- Reliable Data Transfer
- TCP
- Congestion Control
- TCP Congestion Control

Transport Services & Protocols

- Logical communication between application processes running on different hosts
- Transport protocols run in end systems
  - Sender side: breaks application messages into segments, passes to network layer
  - Receiver side: reassembles segments into messages, passes to application layer
- More than one transport protocol available to applications
  - Internet: TCP and UDP

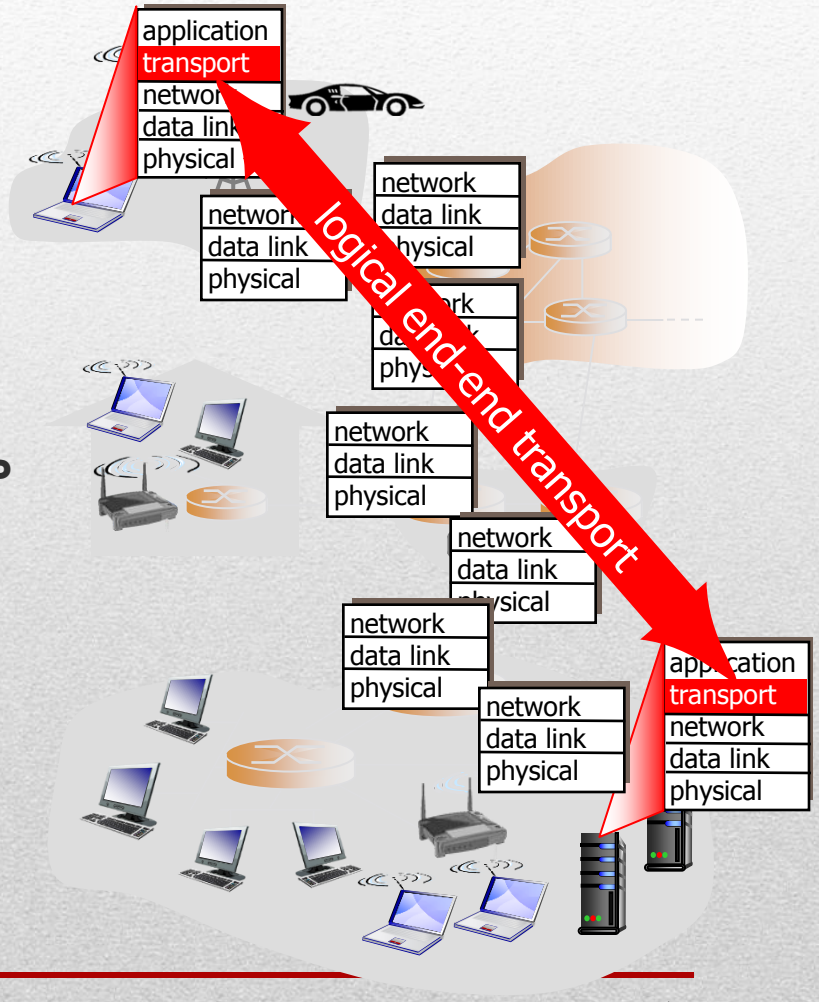


Transport vs. Network Layer

- **Network layer**
  - Logical communication between hosts
- **Transport layer**
  - Logical communication between processes
    - Relies on, enhances, network layer services

Internet Transport Layer Protocols

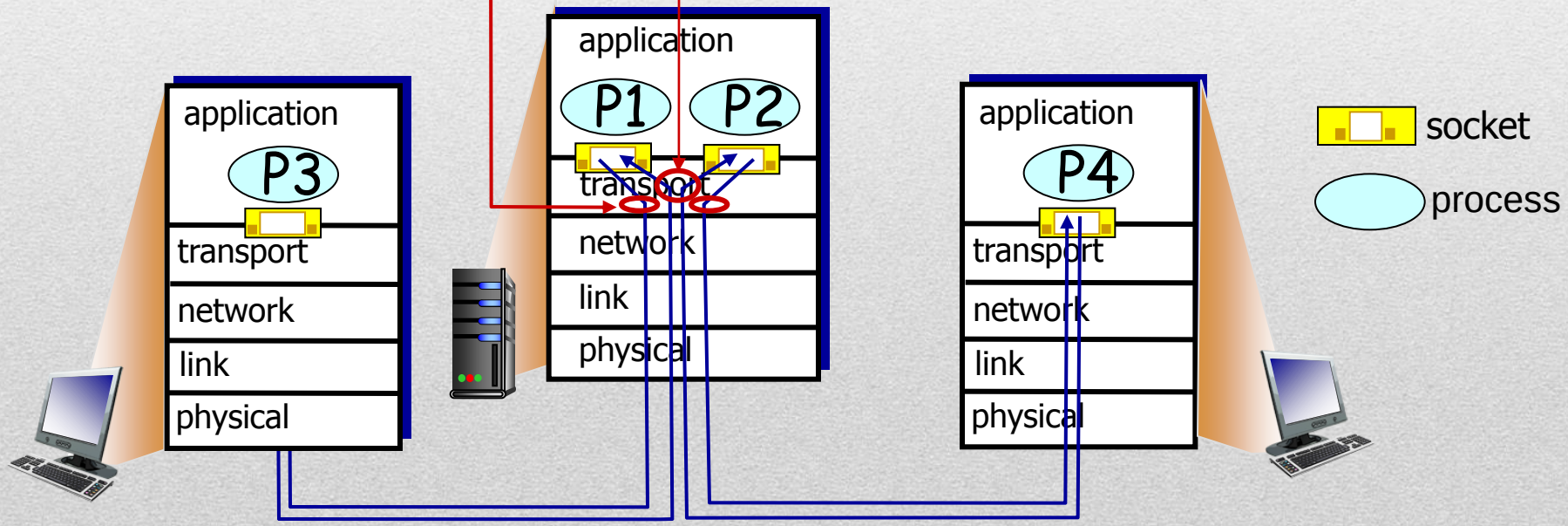
- **Reliable, in-order delivery: TCP**
  - Congestion control
  - Flow control
  - Connection setup
- **Unreliable, unordered delivery: UDP**
  - No-frills extension of "best-effort" IP
- **Services not available**
  - Delay guarantees
  - Bandwidth guarantees



Multiplexing & Demultiplexing

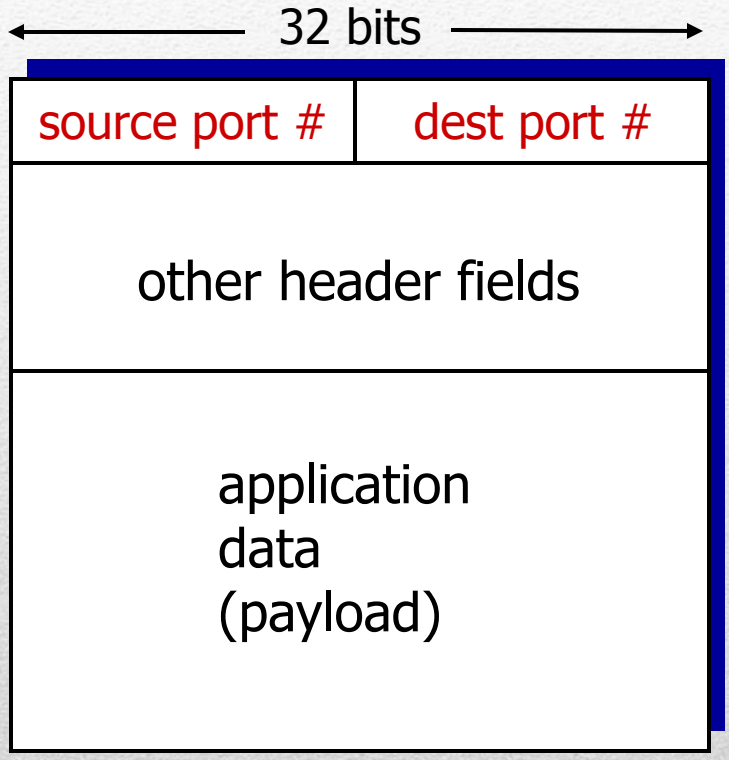
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



Demultiplexing

- 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

Demultiplexing

Socket has host-local port #:

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

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 #

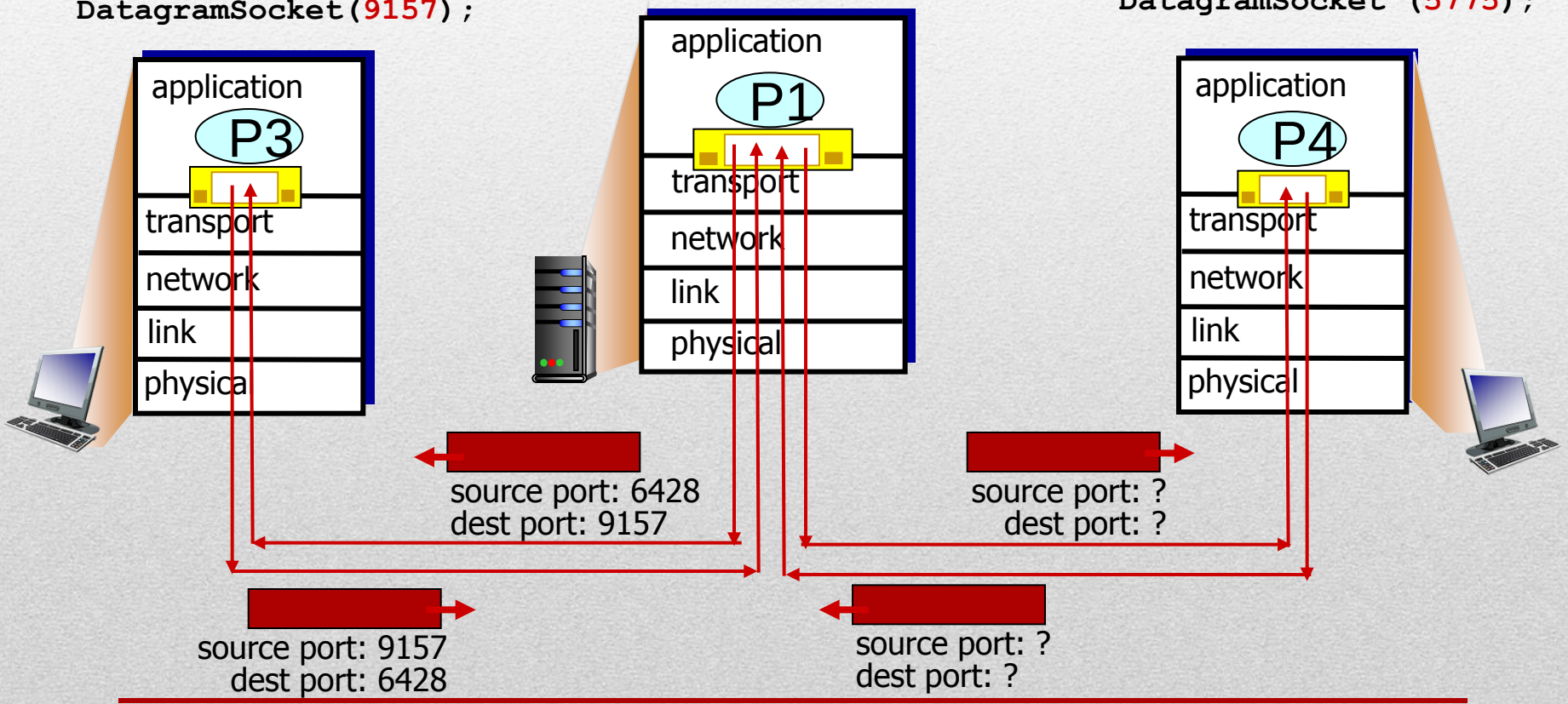
IP datagrams with *same destination port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at destination

Example

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

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

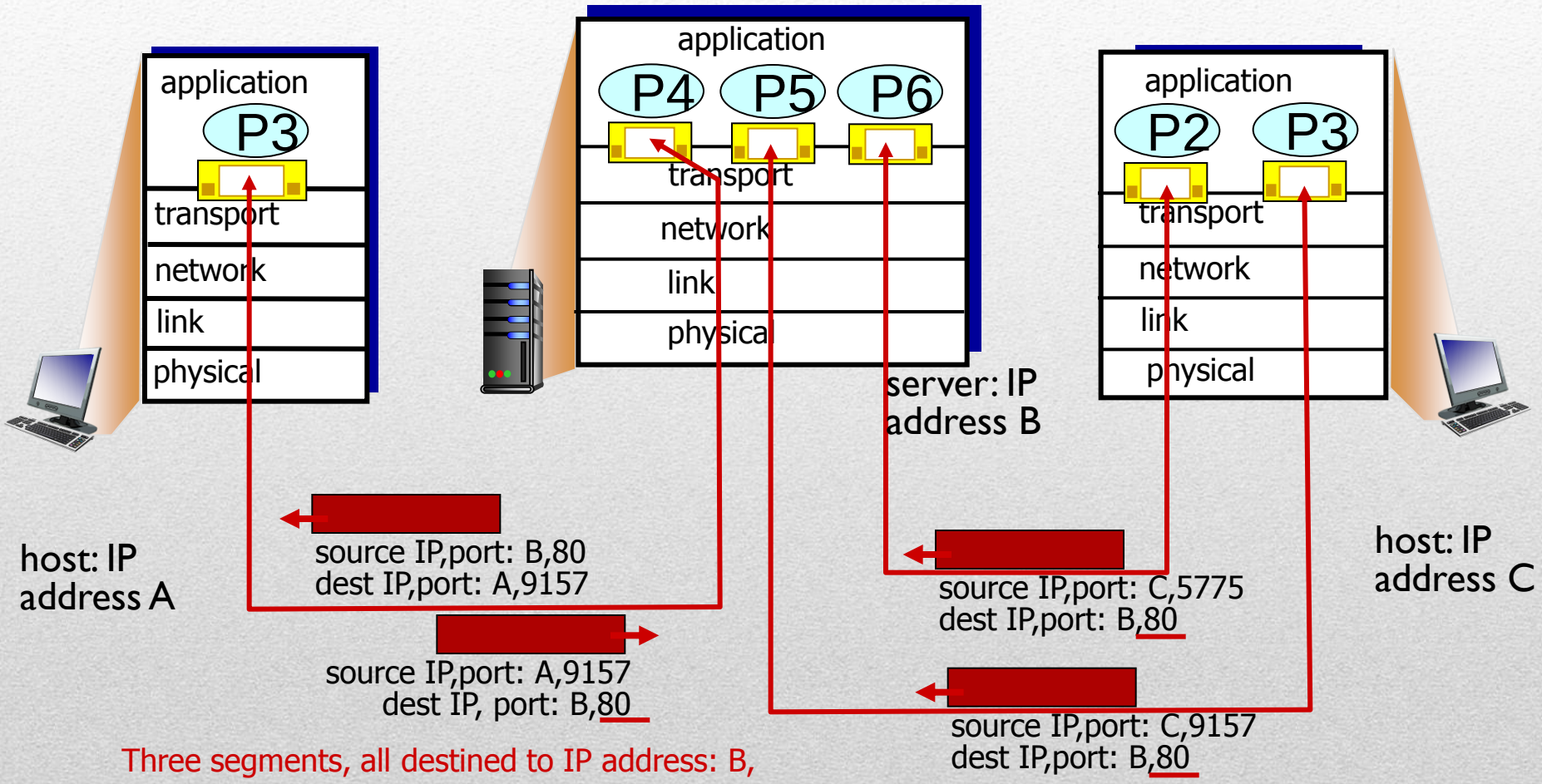
```
DatagramSocket
 mySocket1 = new
 DatagramSocket (5775);
```



Connection Oriented Demultiplexing

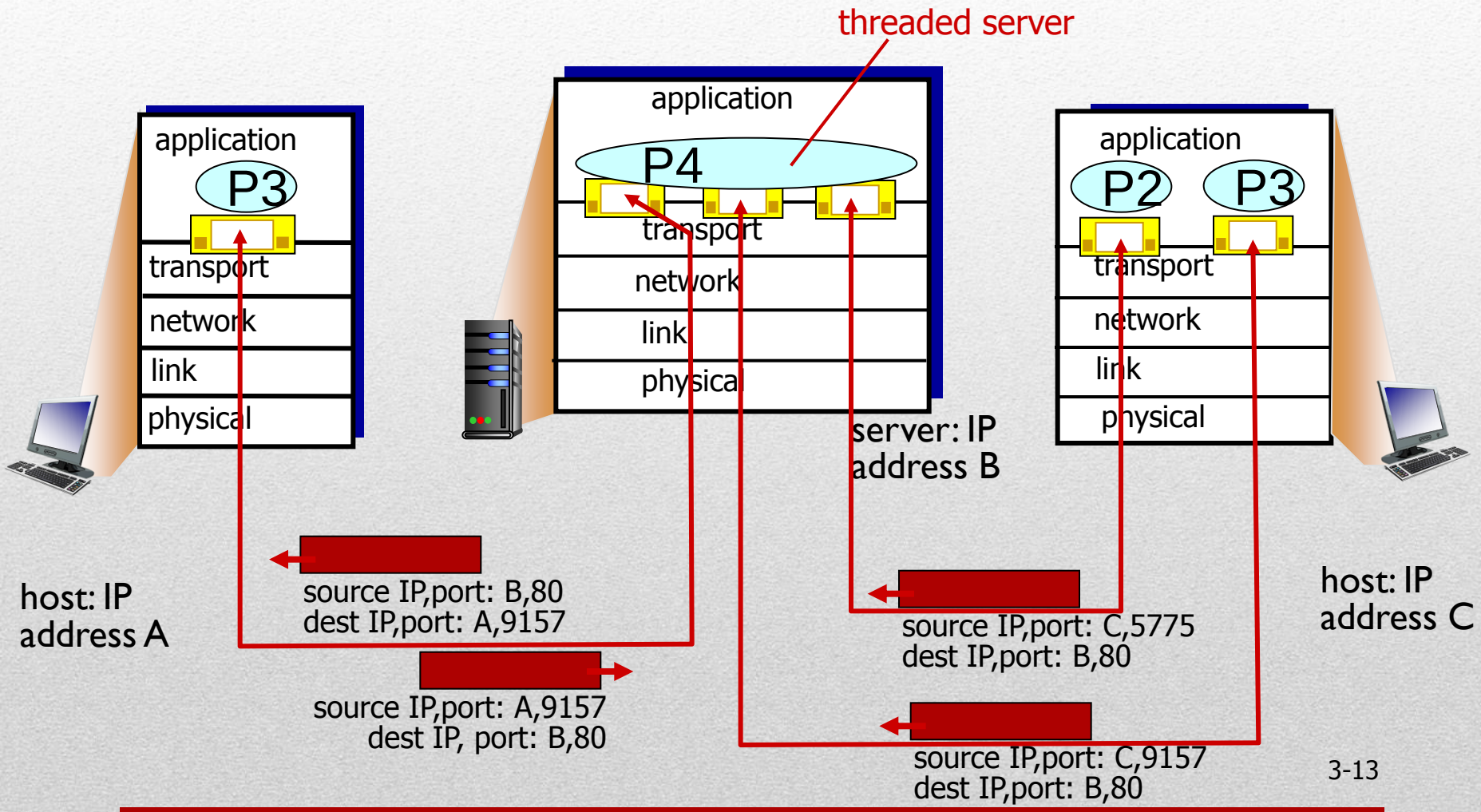
- **TCP socket identified by 4-tuple**
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
- **Demux**
  - Receiver uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets
  - Each socket identified 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

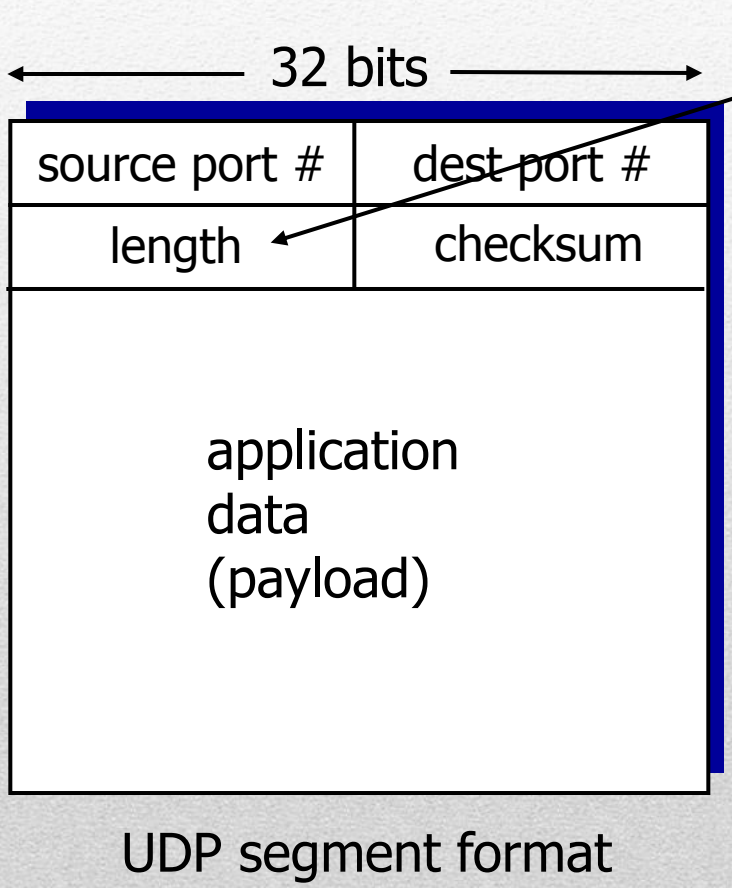
Connection Oriented Demux: Example



Connectionless Transport: UDP

- **"Best effort" service**
  - Lost segments
  - Delivered out-of-order to application
- **Connectionless**
  - No handshaking between UDP sender & receiver
  - Each UDP segment handled independently of others
- **UDP Usage**
  - Multimedia Streaming applications
  - DNS
  - SNMP
- **Reliable transfer over UDP**
  - Add reliability at application layer
  - Application-specific error recovery!

UDP: Segment Header



length, in bytes of UDP segment, including header

UDP segment format

Why There is 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 (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 …

Checksum Example

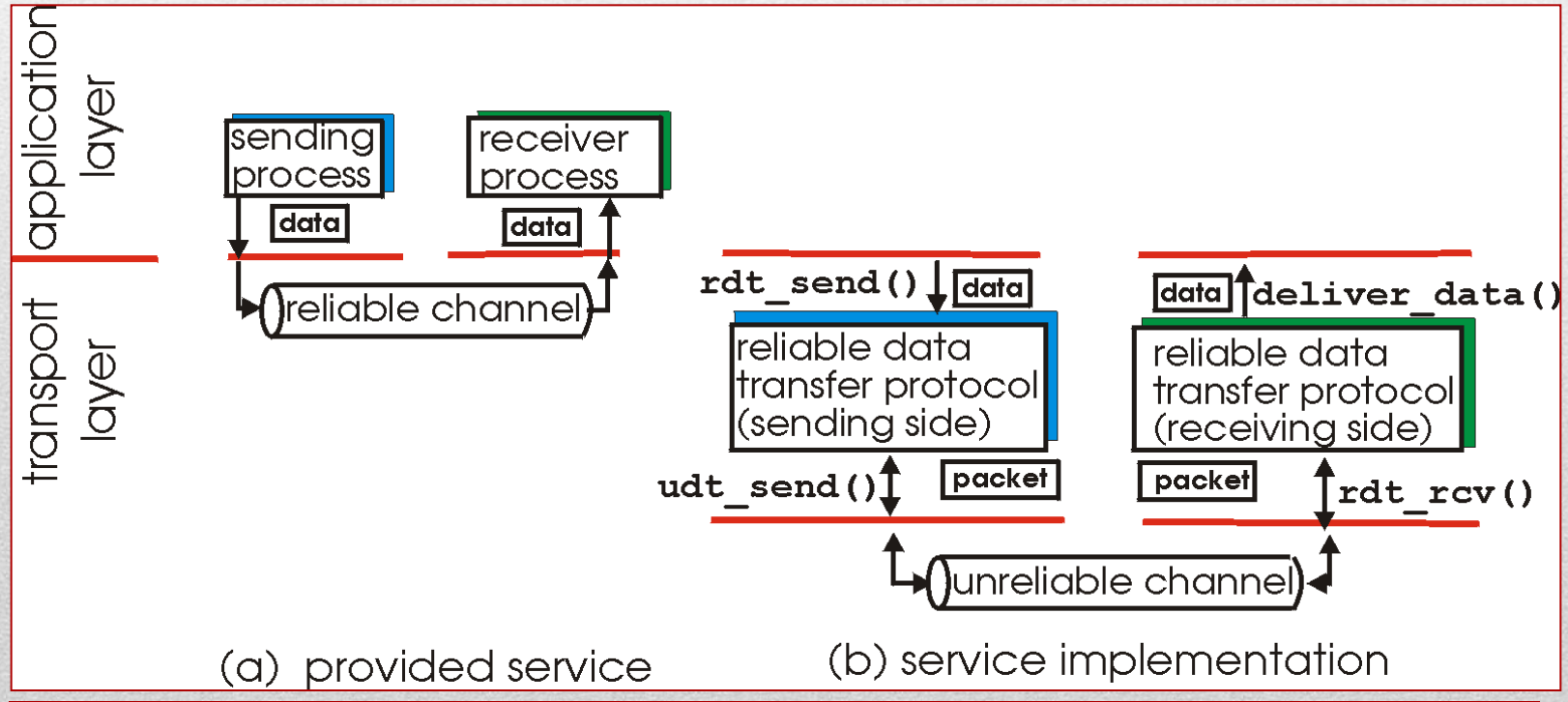
Add two 16-bit integers

	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	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	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	1	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

Principles of Reliable Data Transfer

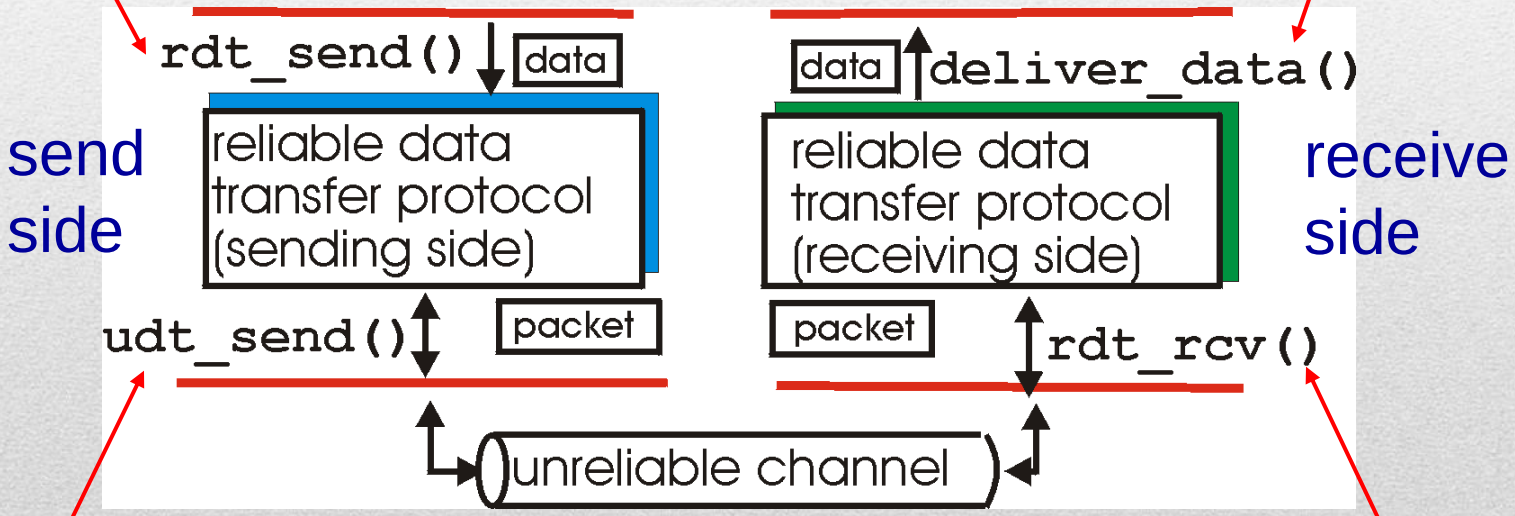
- **Importance**
  - important in application, transport, link layers
  - Top-10 list of important networking topics!
  - Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (RDT)



Reliable Data Transfer

`rdt_send()`: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

`deliver_data()`: called by **rdt** to deliver data to upper



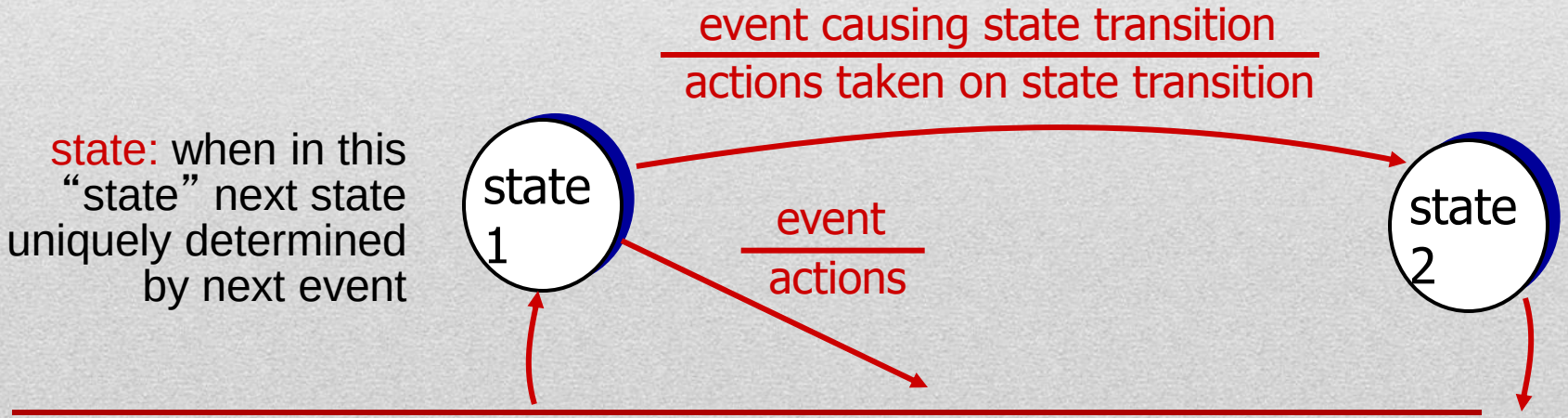
`udt_send()`: called by rdt, to transfer packet over unreliable channel to receiver

`rdt_rcv()`: called when packet arrives on rcv-side of channel

Reliable Data Transfer

Our Plan

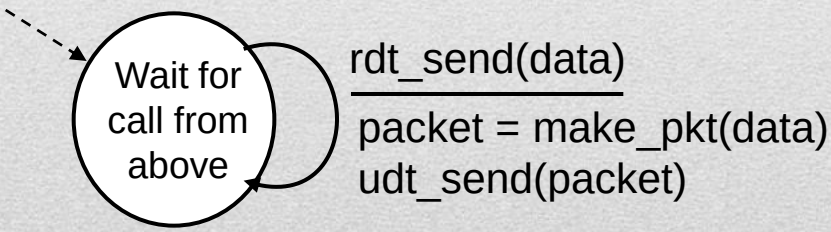
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
  - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender & receiver state, events and actions



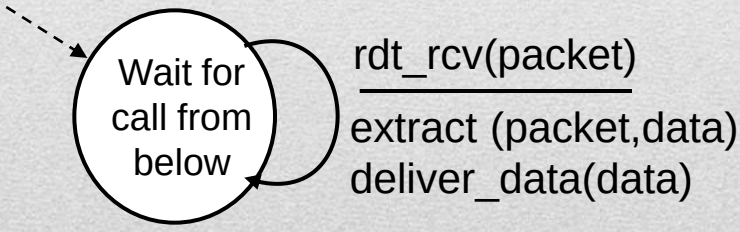
RDT 1.0

- Perfectly reliable underlying channel
  - No bit errors
  - No loss

- Separate FSMs for sender & receiver



sender



receiver