CSCI-1680
Transport Layer 1

Chen Avin

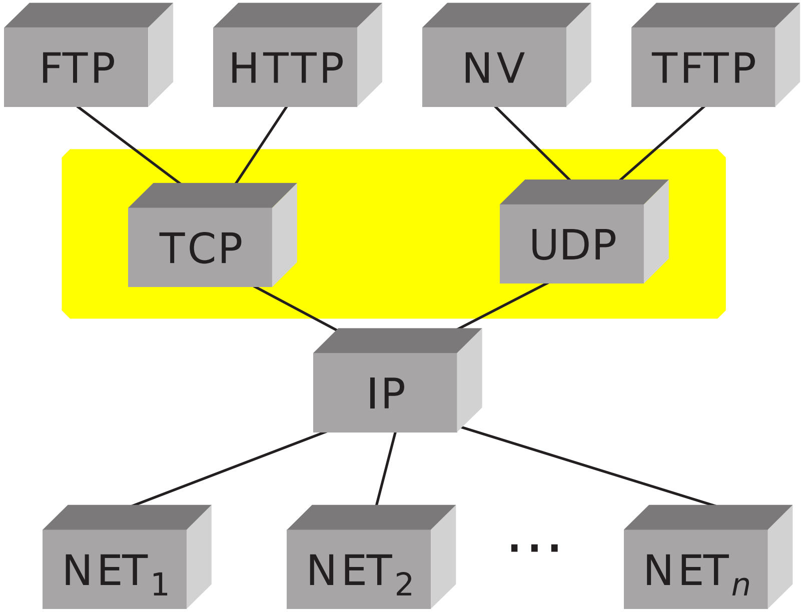
Based partly on lecture notes by David Mazières, Phil Levis, John Jannotti, Peterson & Davie, Rodrigo Fonseca

and “Computer Networking: A Top Down Approach” - 6th edition

Today

- **Transport Layer**
  - UDP
  - TCP Intro
    - Connection Establishment

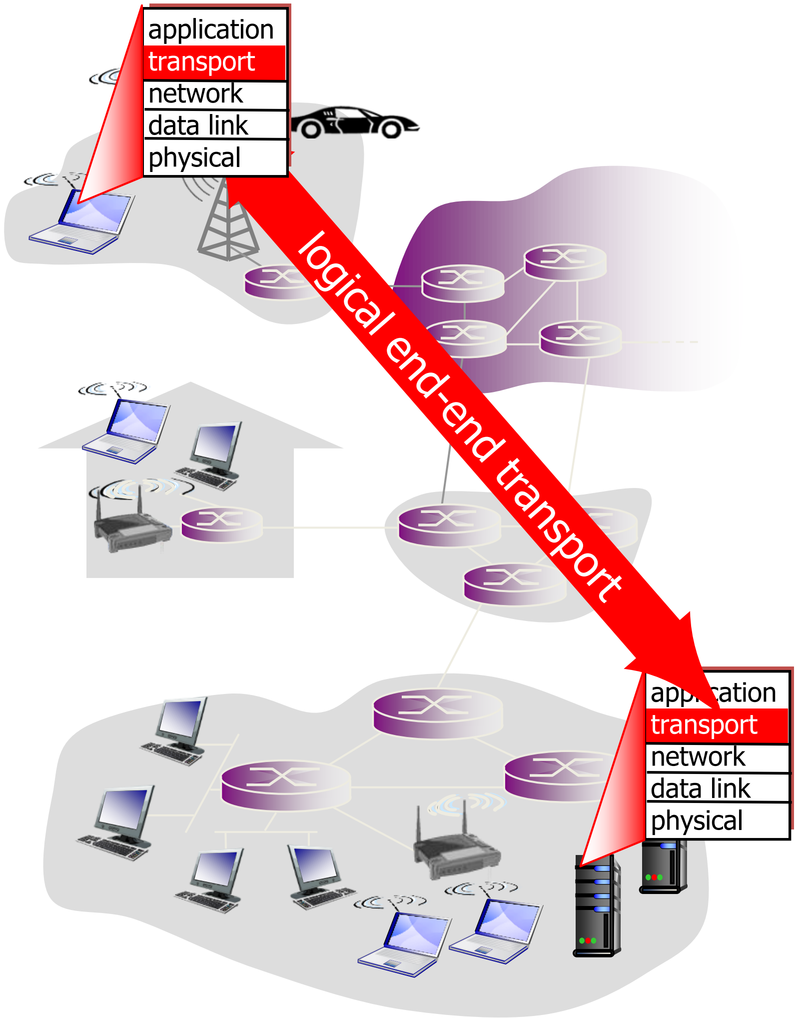
Transport Layer



- **Transport protocols sit on top of network layer**
- **Problem solved: communication among processes**
  - Application-level multiplexing ("ports")
  - Error detection, reliability, etc.

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
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

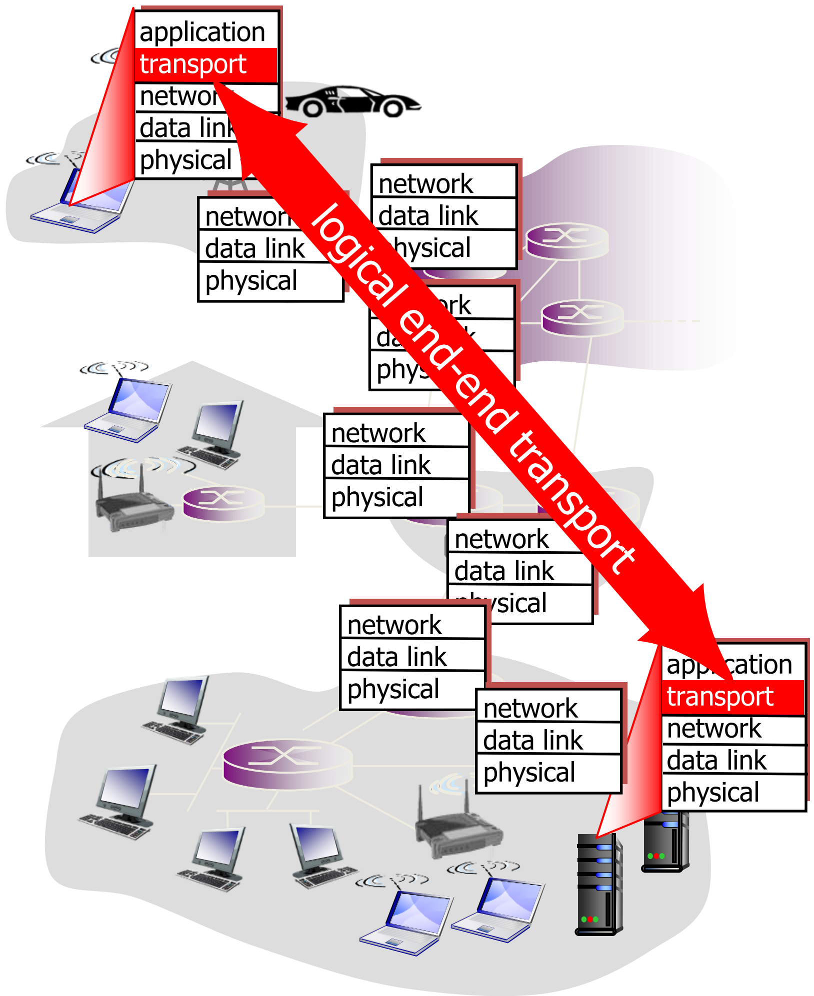


Transport vs. network layer

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

Internet transport-layer protocols

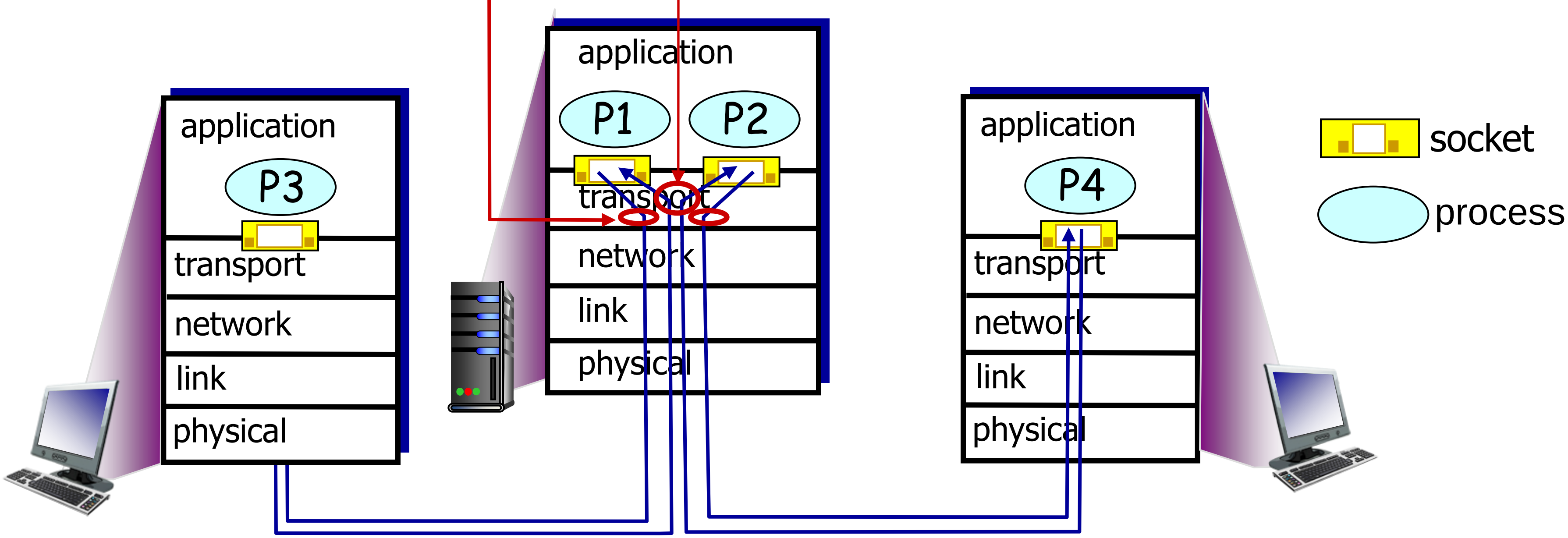
- **reliable, in-order delivery (TCP)**
  - congestion control
  - flow control
  - connection setup
- **unreliable, unordered delivery: UDP**
  - no-frills extension of "best-effort" IP
- **services not available:**
  - delay guarantees
  - bandwidth guarantees



Multiplexing/demultiplexing

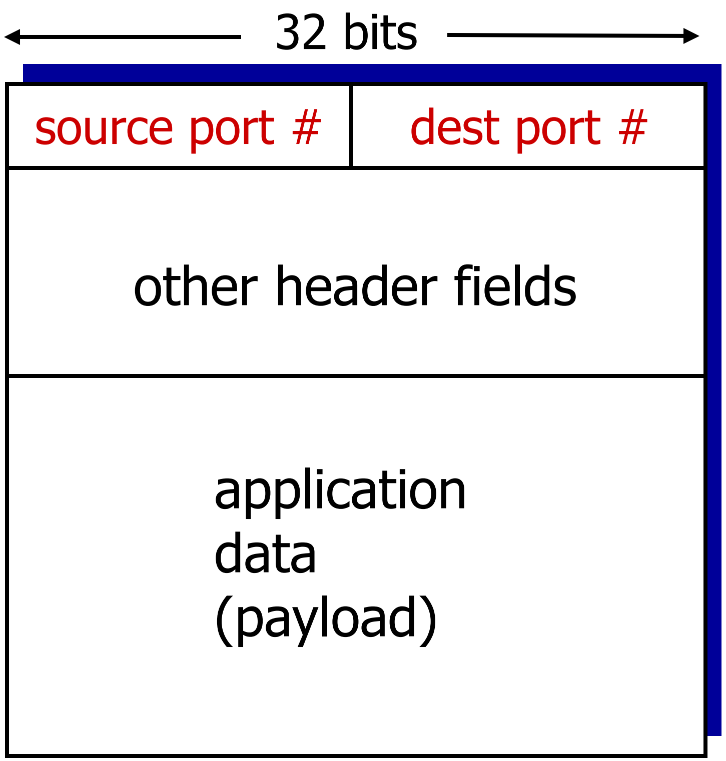
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

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



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 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 #

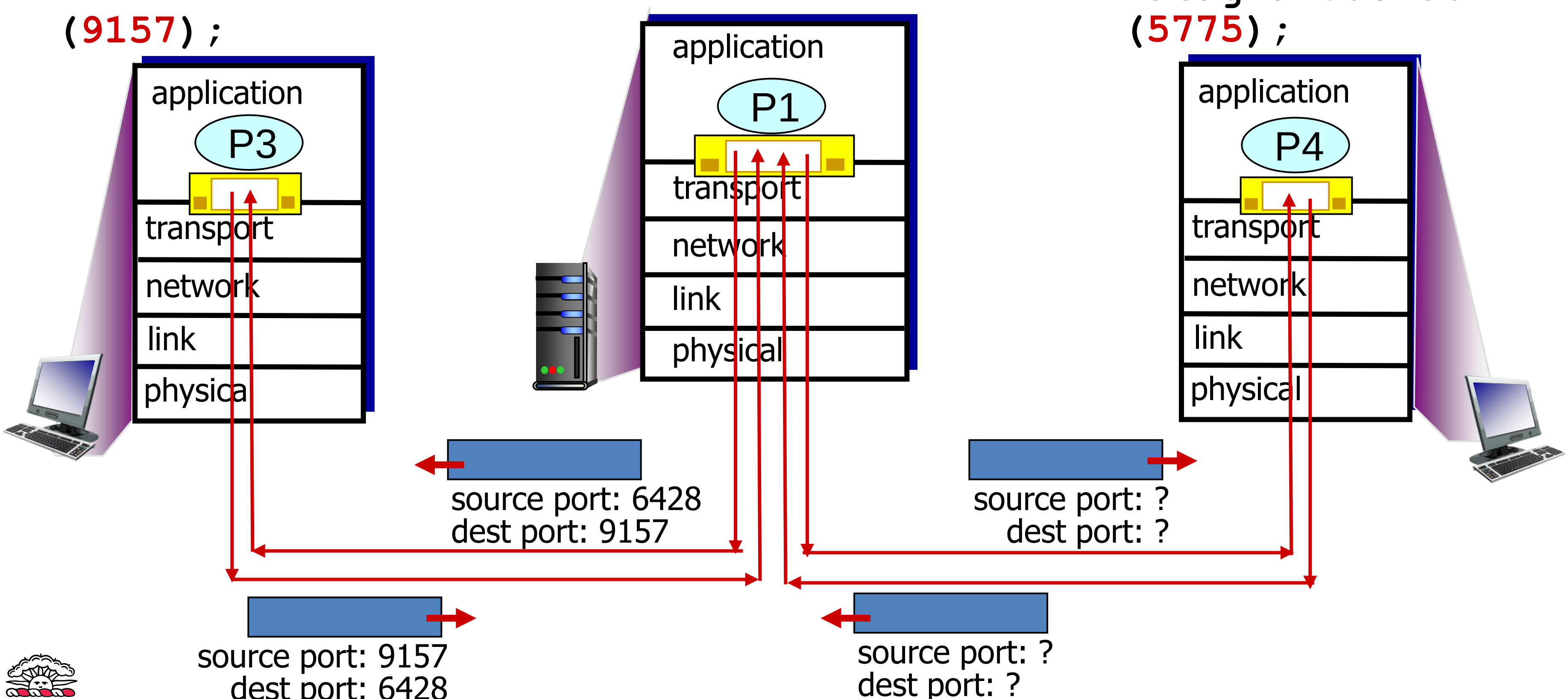
IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

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

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

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



Connection-oriented demux

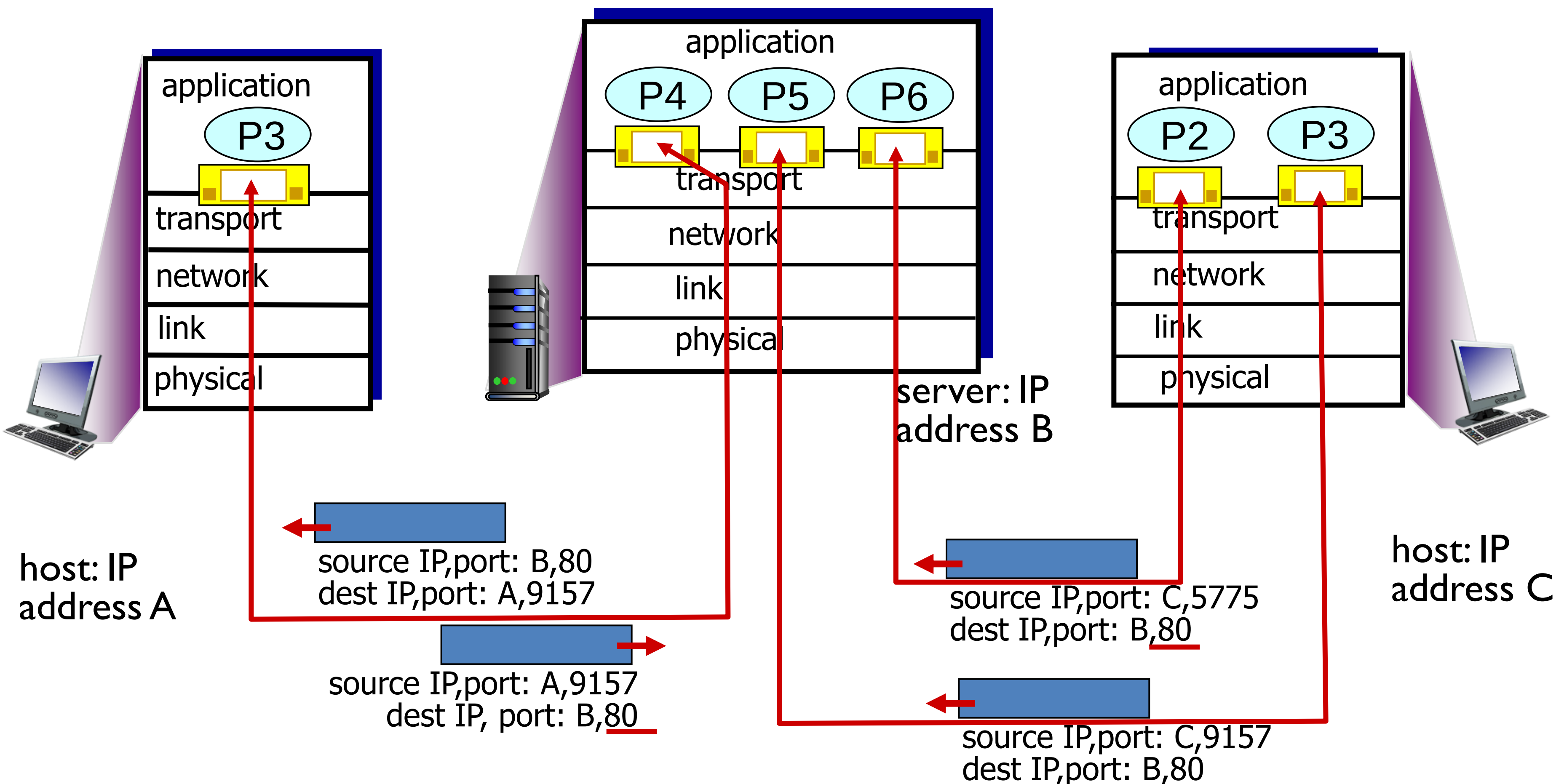
- **TCP socket identified by 4-tuple:**
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- **demux: receiver uses all four values to direct segment to appropriate socket**

- **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

Sockets Client Vs. Server

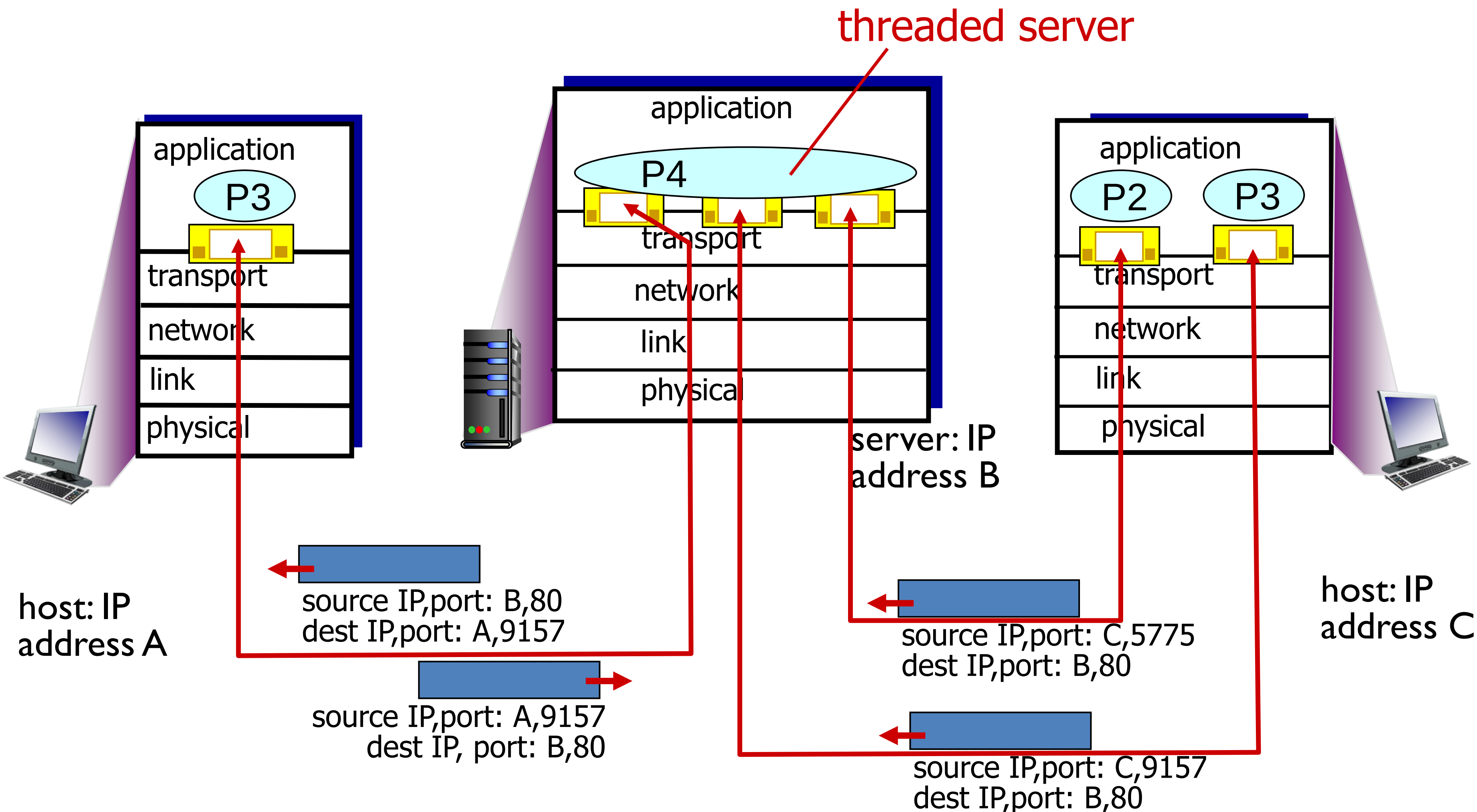
- **A *server* waits for requests at a well-known port that has been reserved for the service it offers.**
- **A *client* allocates an arbitrary, unused, non reserved port for its communication.**
- **Server Side:**
  - Open well-known port (Welcome Socket)
  - Wait for next client request
  - Create a new socket for the client
  - Create thread/process to handle 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



Some well known ports

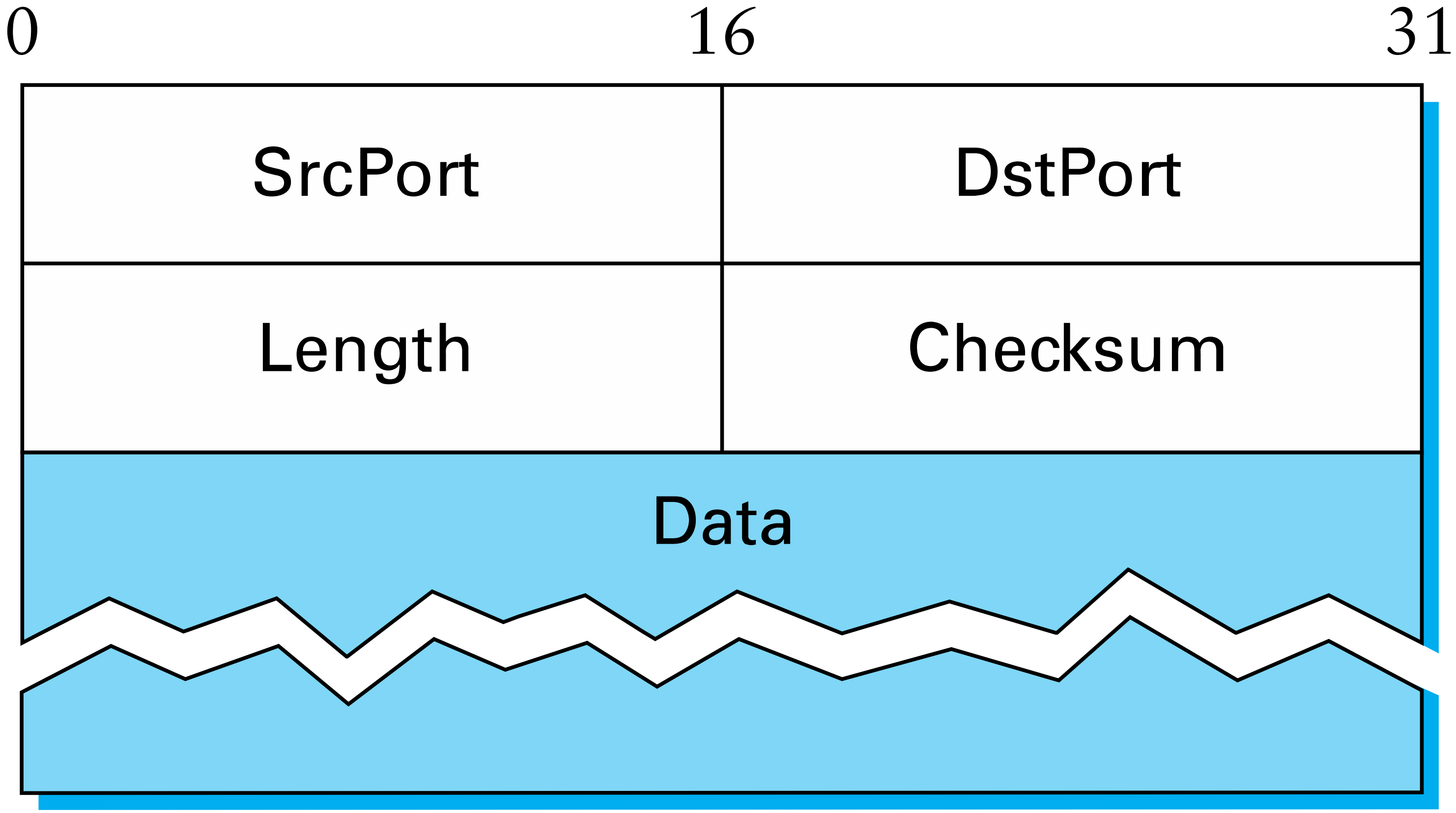
13/TCP,UDP	DAYTIME – (RFC 867)
20/TCP	FTP – data
21/TCP	FTP—control (command)
22/TCP,UDP	Secure Shell (SSH)—used for secure logins, file transfers (scp, sftp) and port forwarding
23/TCP	Telnet protocol—unencrypted text communications
25/TCP,UDP	Simple Mail Transfer Protocol (SMTP)—used for e-mail routing between mail servers
53/TCP,UDP	Domain Name System (DNS)
80/TCP,UDP	Hypertext Transfer Protocol (HTTP)
143/TCP,UDP	Internet Message Access Protocol (IMAP)—used for retrieving, organizing, and synchronizing e-mail messages
179/TCP	BGP (Border Gateway Protocol)
520/UDP	Routing—RIP
546/TCP,UDP	DHCPv6 client
547/TCP,UDP	DHCPv6 server

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!

UDP Header



UDP Checksum

- **Uses the same algorithm as the IP checksum**
  - Set Checksum field to 0
  - Sum all 16-bit words, adding any carry bits to the LSB
  - Flip bits to get checksum (except 0xffff->0xffff)
  - To check: sum whole packet, including sum, should get 0xffff
- **How many errors?**
  - Catches any 1-bit error
  - Not all 2-bit errors
- **Optional in IPv4: not checked if value is 0**

Pseudo Header

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|          source address           |
+--------+--------+--------+--------+
|        destination address        |
+--------+--------+--------+--------+
|  zero  |protocol|   UDP length    |
+--------+--------+--------+--------+
```

- **UDP Checksum is computer over *pseudo-header* prepended to the UDP header**
  - For IPv4: IP Source, IP Dest, Protocol (=17), plus UDP length
- **What does this give us?**
- **What is a problem with this?**
  - Is UDP a layer on top of IP?

Next Problem: Reliability

- **Review: reliability on the link layer**

Problem	Mechanism
Dropped Packets	Acknowledgments + Timeout
Duplicate Packets	Sequence Numbers
Packets out of order	Receiver Window
Keeping the pipe full	Sliding Window (Pipelining)

- **Single link: things were easy… ☺**

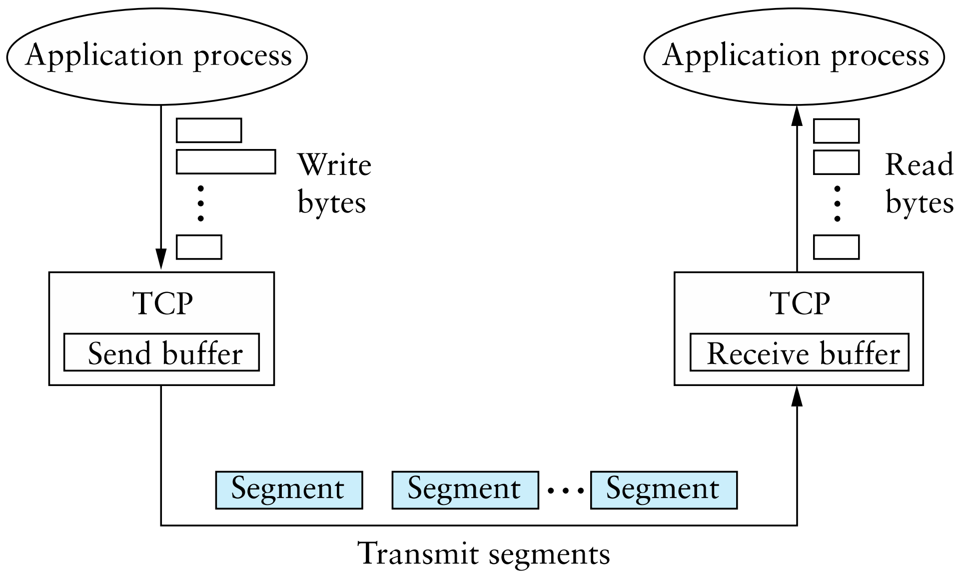
Transport Layer Reliability

- **Extra difficulties**
  - Multiple hosts
  - Multiple hops
  - Multiple potential paths
- **Need for connection establishment, tear down**
  - Analogy: dialing a number versus a direct line
- **Varying RTTs**
  - Both across connections and *during* a connection
  - Why do they vary? What do they influence?

Extra Difficulties (cont.)

- **Out of order packets**
  - Not only because of drops/retransmissions
  - Can get very old packets (up to 120s), must not get confused
- **Unknown resources at other end**
  - Must be able to discover receiver buffer: flow control
- **Unknown resources in the network**
  - Should not overload the network
  - But should use as much as safely possible
  - Congestion Control (next class)

TCP – Transmission Control Protocol



- **Service model: “reliable, connection oriented, full duplex point-to-point byte stream”**
  - Endpoints: <IP Address, Port>
- **Flow control**
  - If one end stops reading, writes at other eventually stop/fail
- **Congestion control**
  - Keeps sender from overloading the network

TCP

- **Specification**
  – RFC 793 (1981), RFC 1222 (1989, some corrections), RFC 5681 (2009, congestion control), …
- **Was born coupled with IP, later factored out**
  – We talked about this, don't always need everything!
- **End-to-end protocol**
  – Minimal assumptions on the network
  – All mechanisms run on the end points
- **Alternative idea:**
  – Provide reliability, flow control, etc, link-by-link
  – Does it work?

Why not provide (*) on the network layer?

- **Cost**
  - These functionalities are not free: don’t burden those who don’t need them
- **Conflicting**
  - Timeliness and in-order delivery, for example
- **Insufficient**
  - Example: reliability

* may be security, reliability, ordering guarantees, …

End-to-end argument

- **Functions placed at lower levels of a system may be redundant or of little value**
  - They may **need** to be performed at a higher layer anyway
- **But they may be justified for performance reasons**
  - Or just because they provide *most* of what is needed
  - Example: retransmissions
- **Lesson: weigh the costs and benefits at each layer**
  - Also: the *end* also varies from case to case

TCP segment structure

counting by bytes
of data (not segments!)

32 bits

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20	160	Options (if Data Offset > 5,padded at end with "0" bytes if necessary)																															
...	...	...																															

bytes
rcvr willing
to accept

ACK: ACK # valid

PSH: push data now
(generally not used)

URG: urgent data
(generally not used)

?

ECN

Congestion
Window
Reduced
(CWR)

e.g.,
MSS,
Window scaling,
timestemp

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet checksum
(as in UDP)
data + header

TCP seq. numbers, ACKs

sequence numbers:

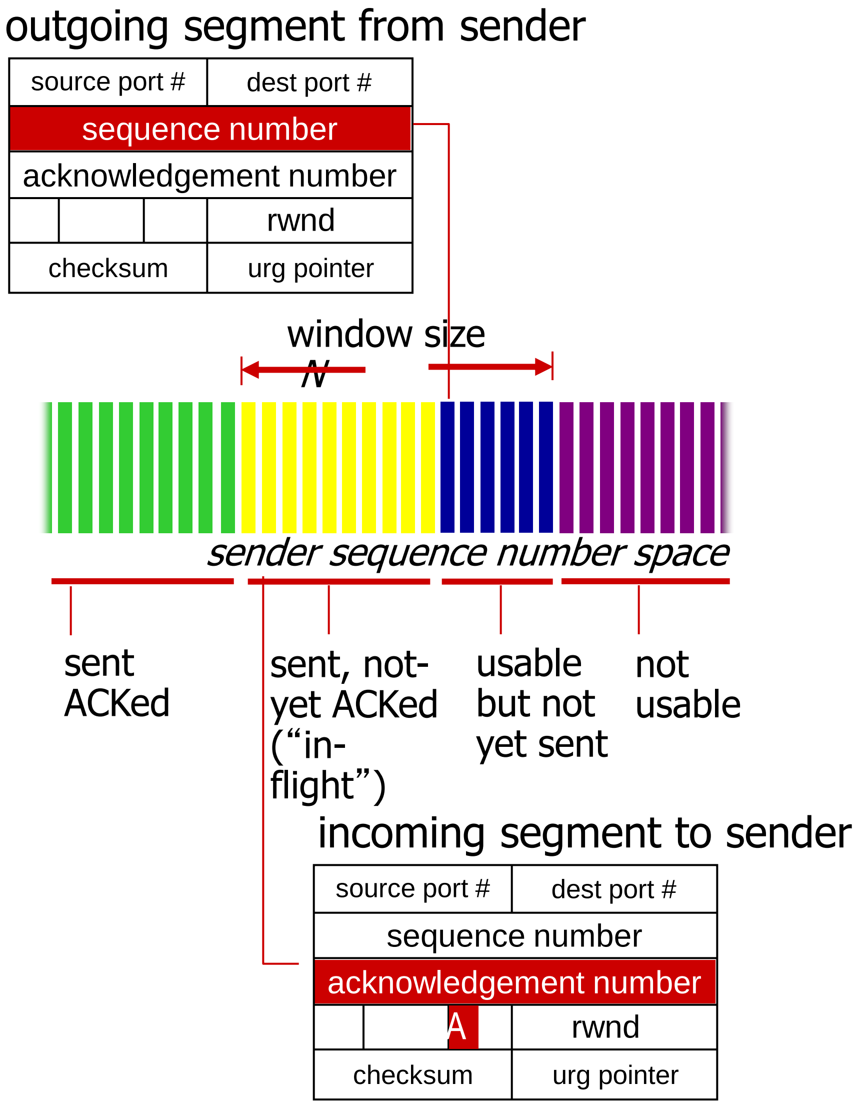
- byte stream "number" of first byte in segment's data

acknowledgements:

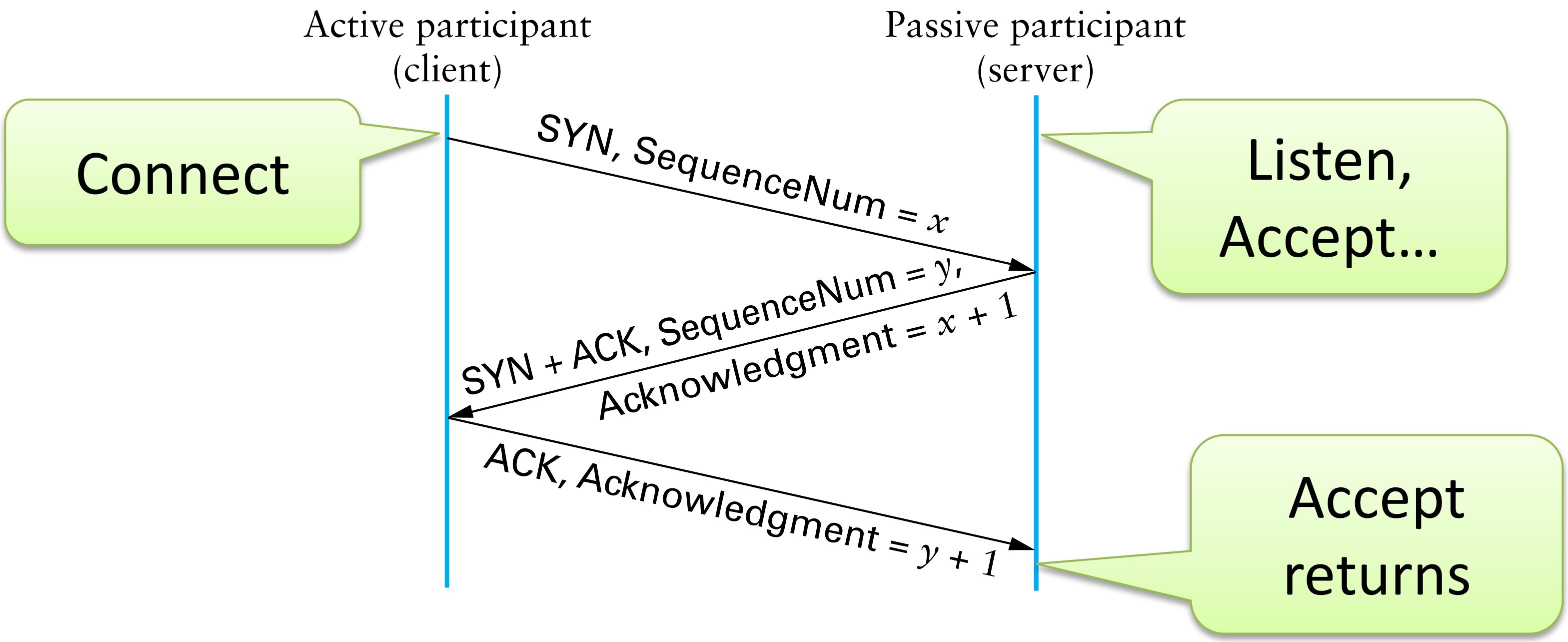
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor

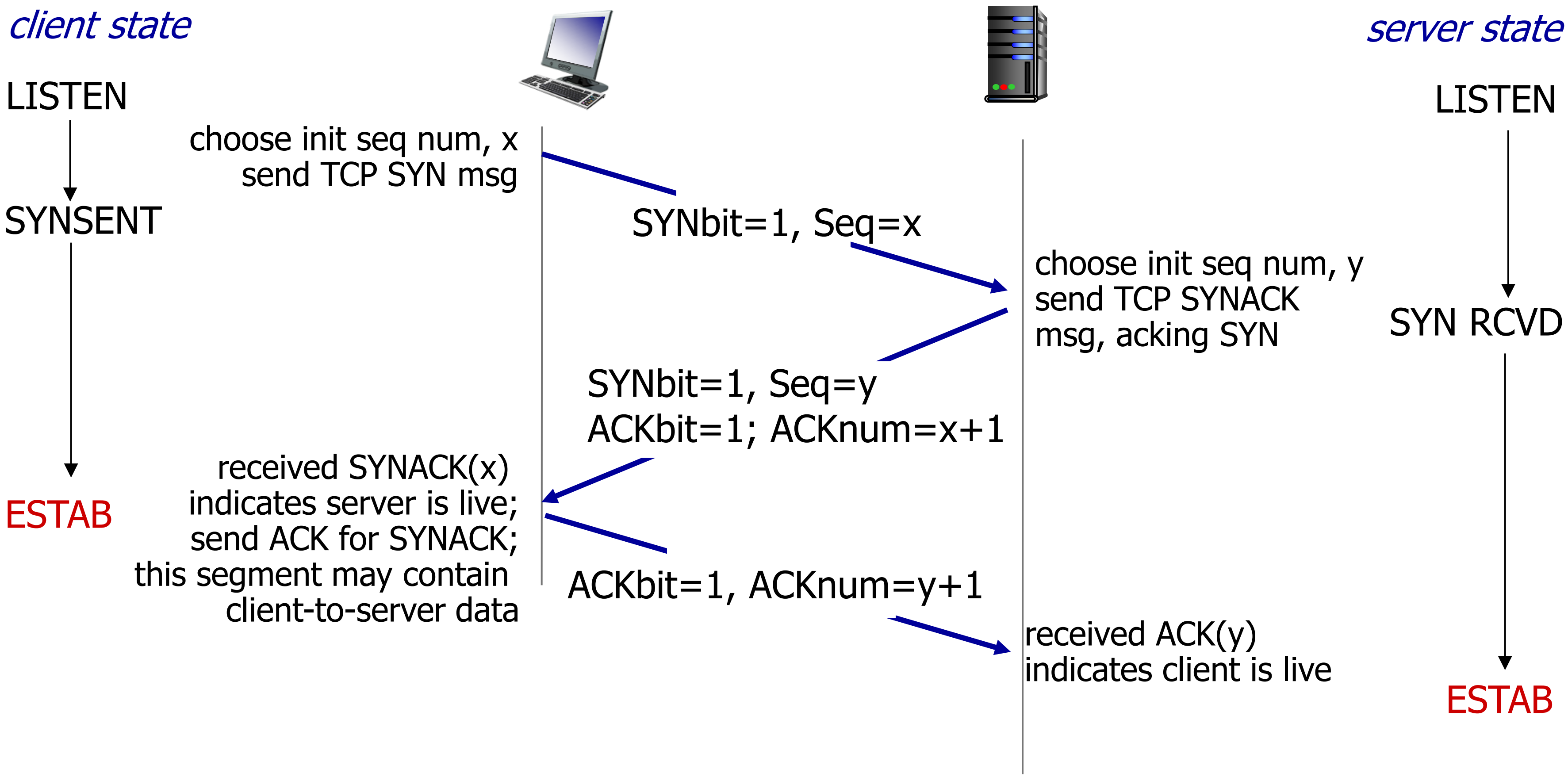


Establishing a Connection

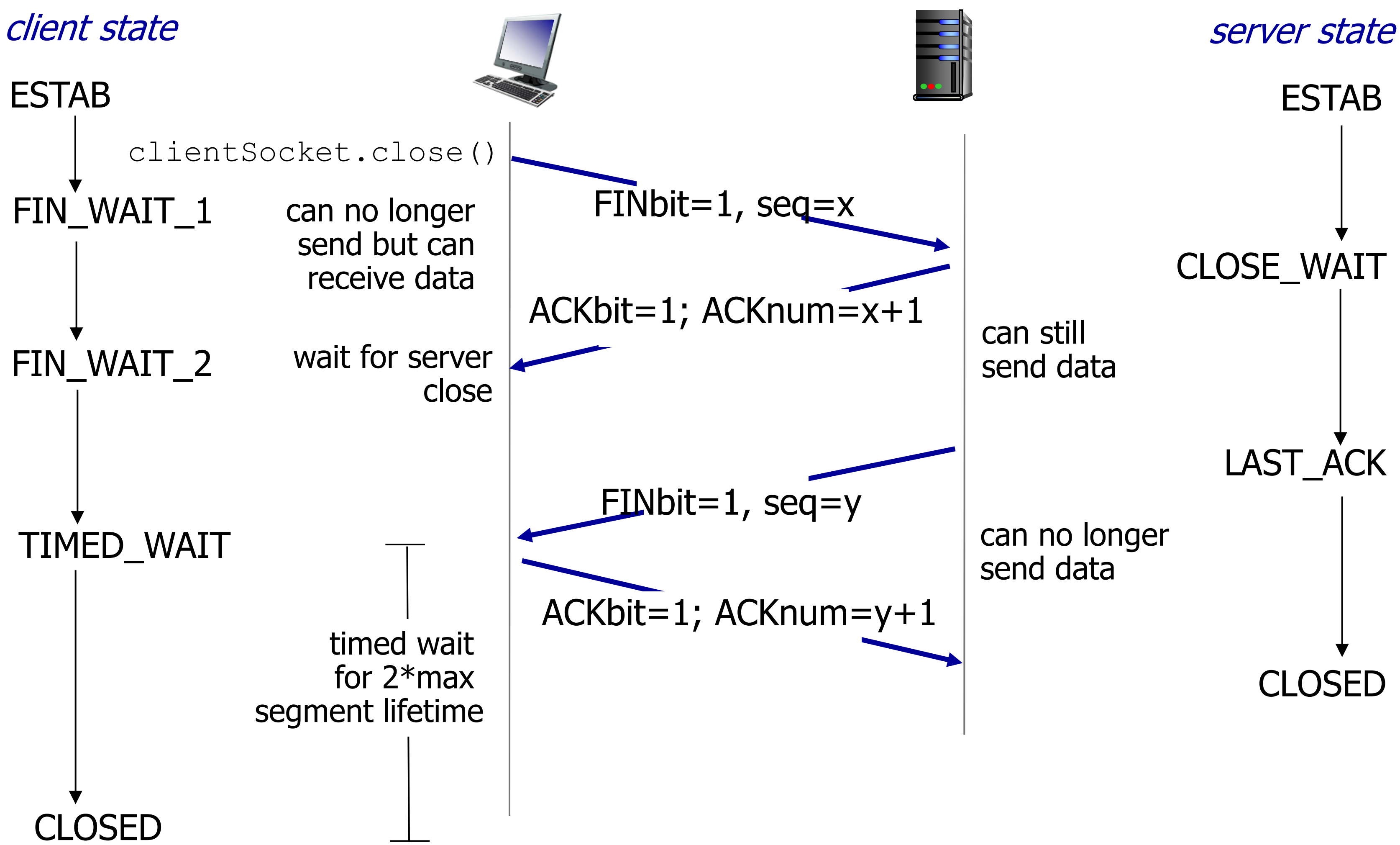


- **Three-way handshake**
  - Two sides agree on respective initial sequence nums
- **If no one is listening on port: server sends RST**
- **If server is overloaded: ignore SYN**
- **If no SYN-ACK: retry, timeout**

TCP 3-way handshake



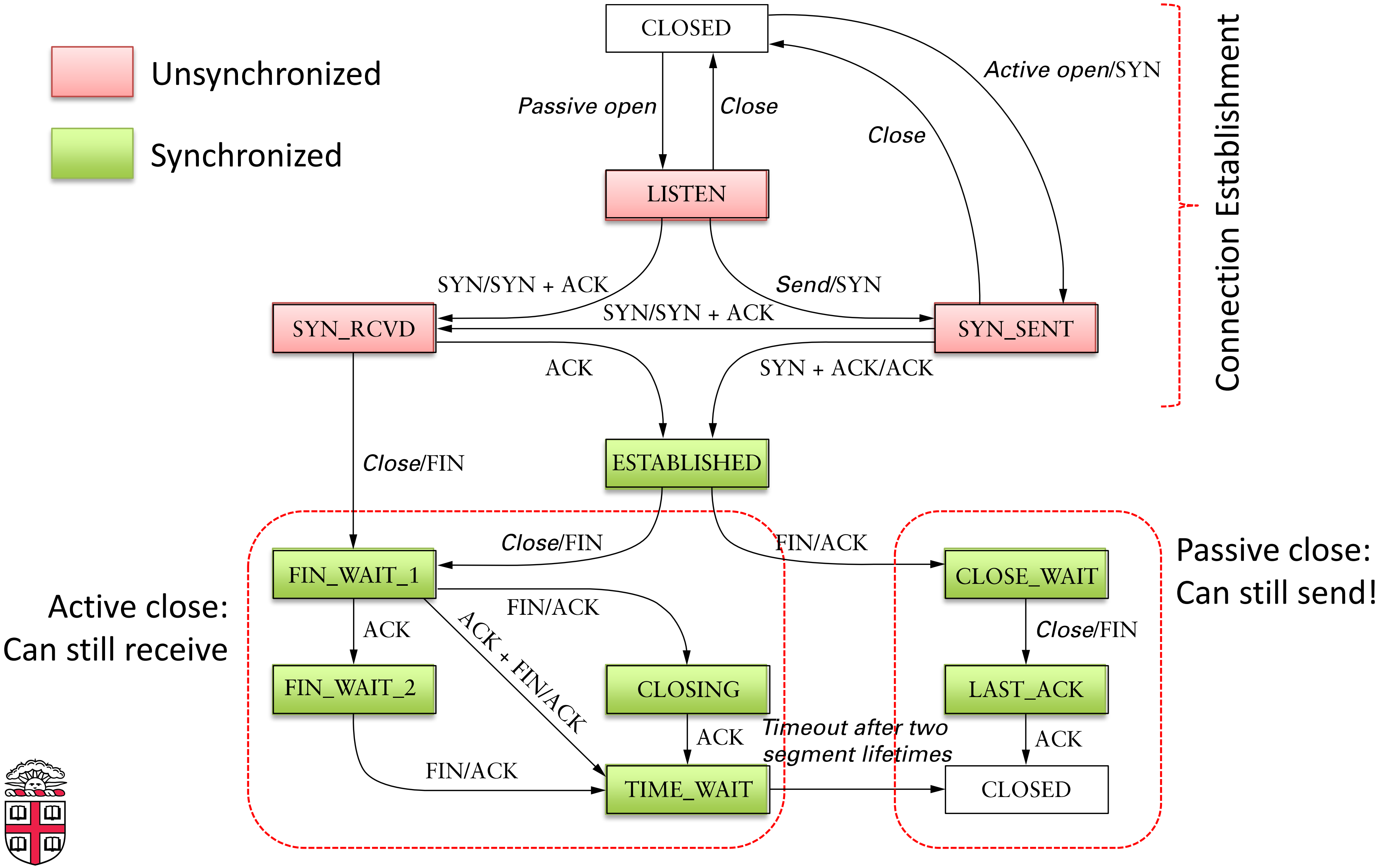
TCP: closing a connection



TIME_WAIT

- **Why do you have to wait for 2MSL in TIME_WAIT?**
  – What if last ack is severely delayed, AND
  – Same port pair is immediately reused for a new connection?
- **Solution: active closer goes into TIME_WAIT**
  – Waits for 2MSL (Maximum Segment Lifetime)
- **Can be problematic for active servers**
  – OS has too many sockets in TIME_WAIT, can accept less connections
    - Hack: send RST and delete socket, SO_LINGER = 0
  – OS won't let you re-start server because port in use
    - SO_REUSEADDR lets you rebind

Summary of TCP States



Next class

- Sending data over TCP