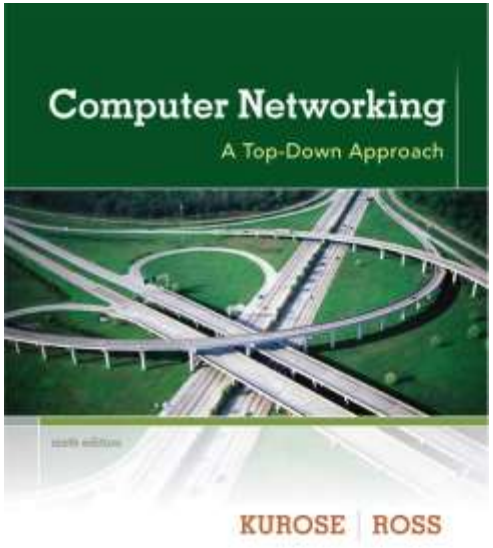
Chapter 4
Network Layer



A note on the use of these ppt slides:

We’ re making these slides freely available to all (faculty, students, readers). They’ re in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we’ d like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapter 4: outline

4.6 routing in the Internet

- RIP
- OSPF
- BGP

Chapter 4: outline

4.1 introduction

4.2 virtual circuit and datagram networks

4.3 what's inside a router

4.4 IP: Internet Protocol

- datagram format
- IPv4 addressing
- ICMP
- IPv6

4.5 routing algorithms

- link state
- distance vector
- hierarchical routing

4.6 routing in the Internet

- RIP
- OSPF
- BGP

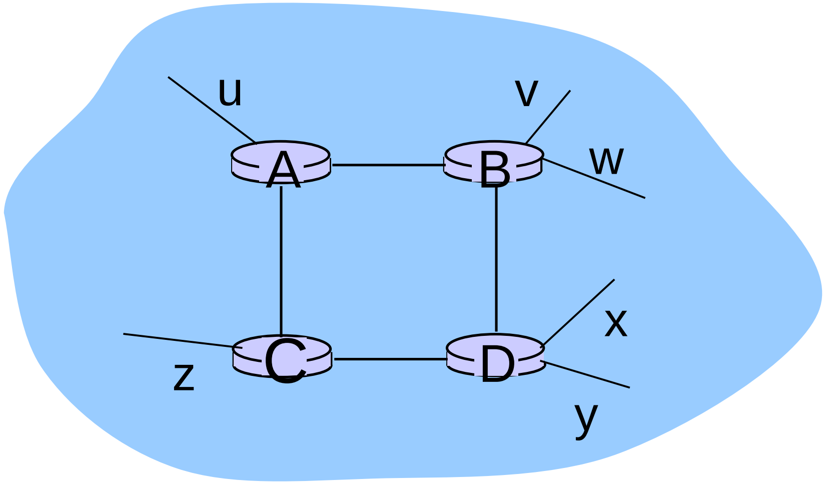
4.7 broadcast and multicast routing

Intra-AS Routing

- also known as *interior gateway protocols (IGP)*
- most common intra-AS routing protocols:
  - RIP: Routing Information Protocol
  - OSPF: Open Shortest Path First
  - IGRP: Interior Gateway Routing Protocol (Cisco proprietary)

RIP (Routing Information Protocol)

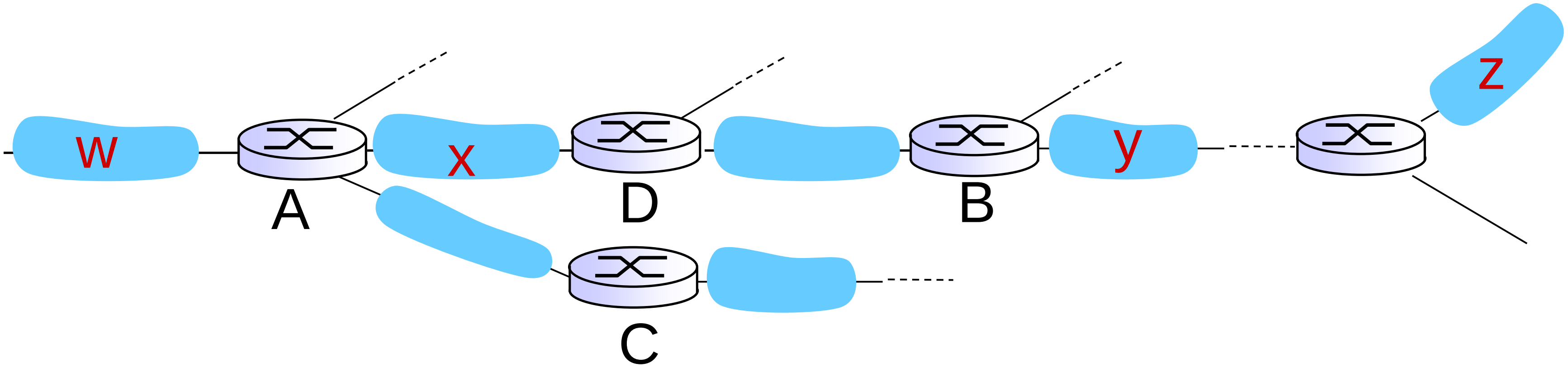
- included in BSD-UNIX distribution in 1982
- distance vector algorithm
  - distance metric: # hops (max = 15 hops), each link has cost 1
  - DVs exchanged with neighbors every 30 sec in response message (aka advertisement)
  - each advertisement: list of up to 25 destination *subnets* *(in IP addressing sense)*



from router A to destination *subnets:*

subnet	hops
u	1
v	2
w	2
x	3
y	3
z	2

RIP: example



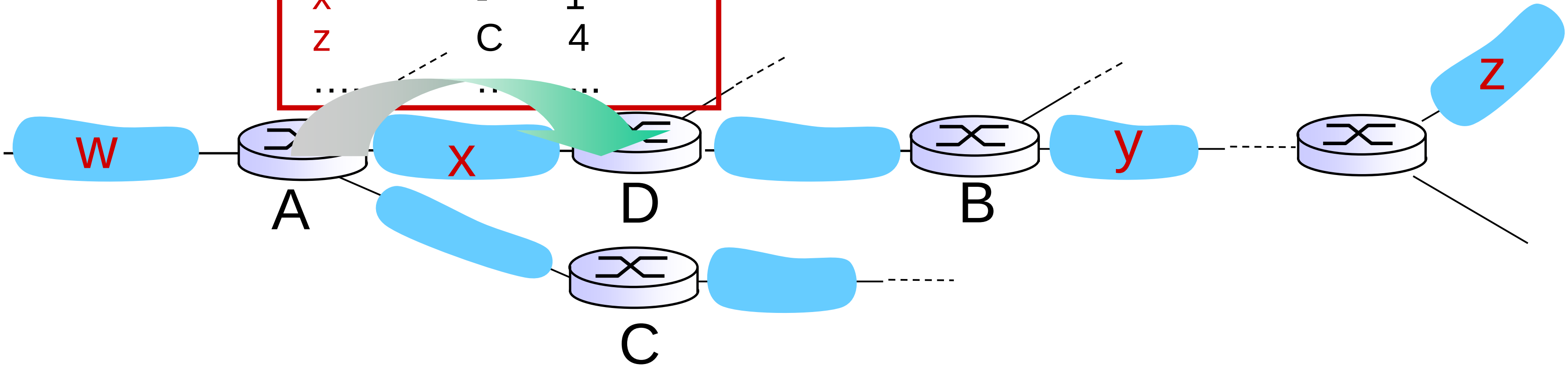
routing table in router D

destination subnet	next router	# hops to dest
w	A	2
y	B	2
z	B	7
x	--	1
….	….	….

RIP: example

A-to-D advertisement

dest	next	hops
w	-	1
x	-	1
z	C	4
….	….	….



routing table in router D

destination subnet	next router	# hops to dest
w	A	2
y	B	2
z	~~B~~ A	~~7~~ 5
x	--	1
….	….	….

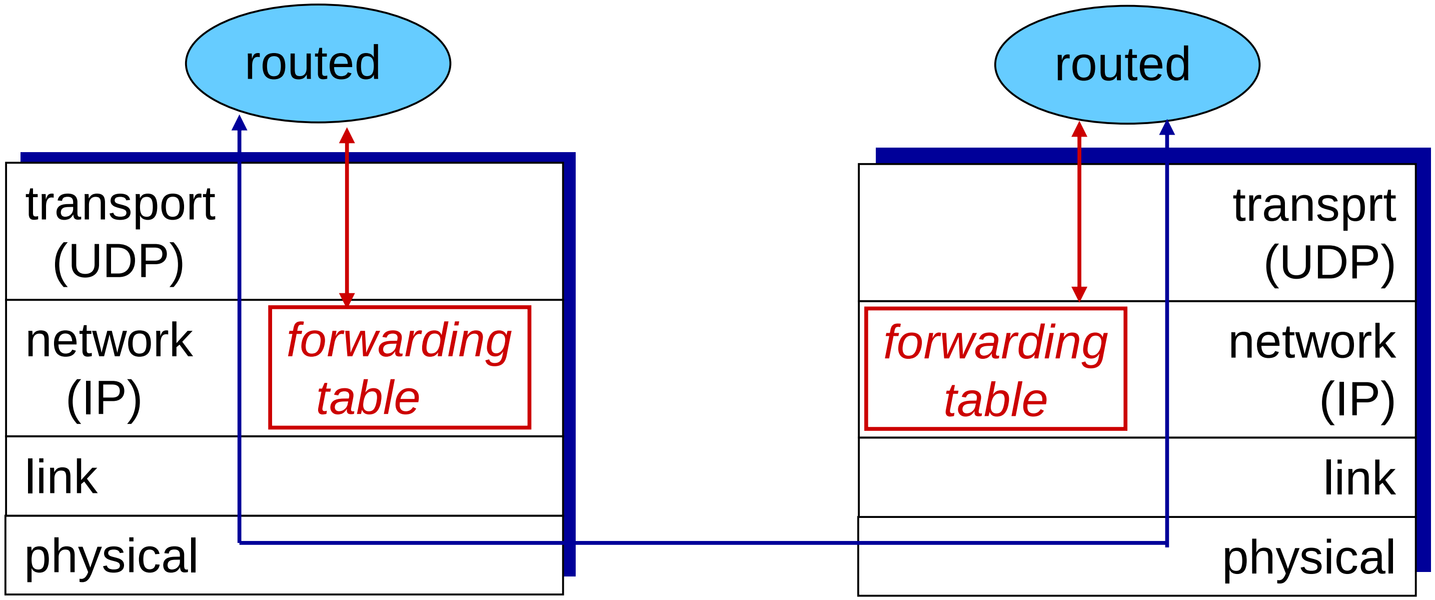
RIP: link failure, recovery

if no advertisement heard after 180 sec --> neighbor/link declared dead

- routes via neighbor invalidated
- new advertisements sent to neighbors
- neighbors in turn send out new advertisements (if tables changed)
- link failure info quickly (?) propagates to entire net
- *poison reverse* used to prevent ping-pong loops (infinite distance = 16 hops)

RIP table processing

- RIP routing tables managed by *application-level* process called route-d (daemon)
- advertisements sent in UDP packets, periodically repeated



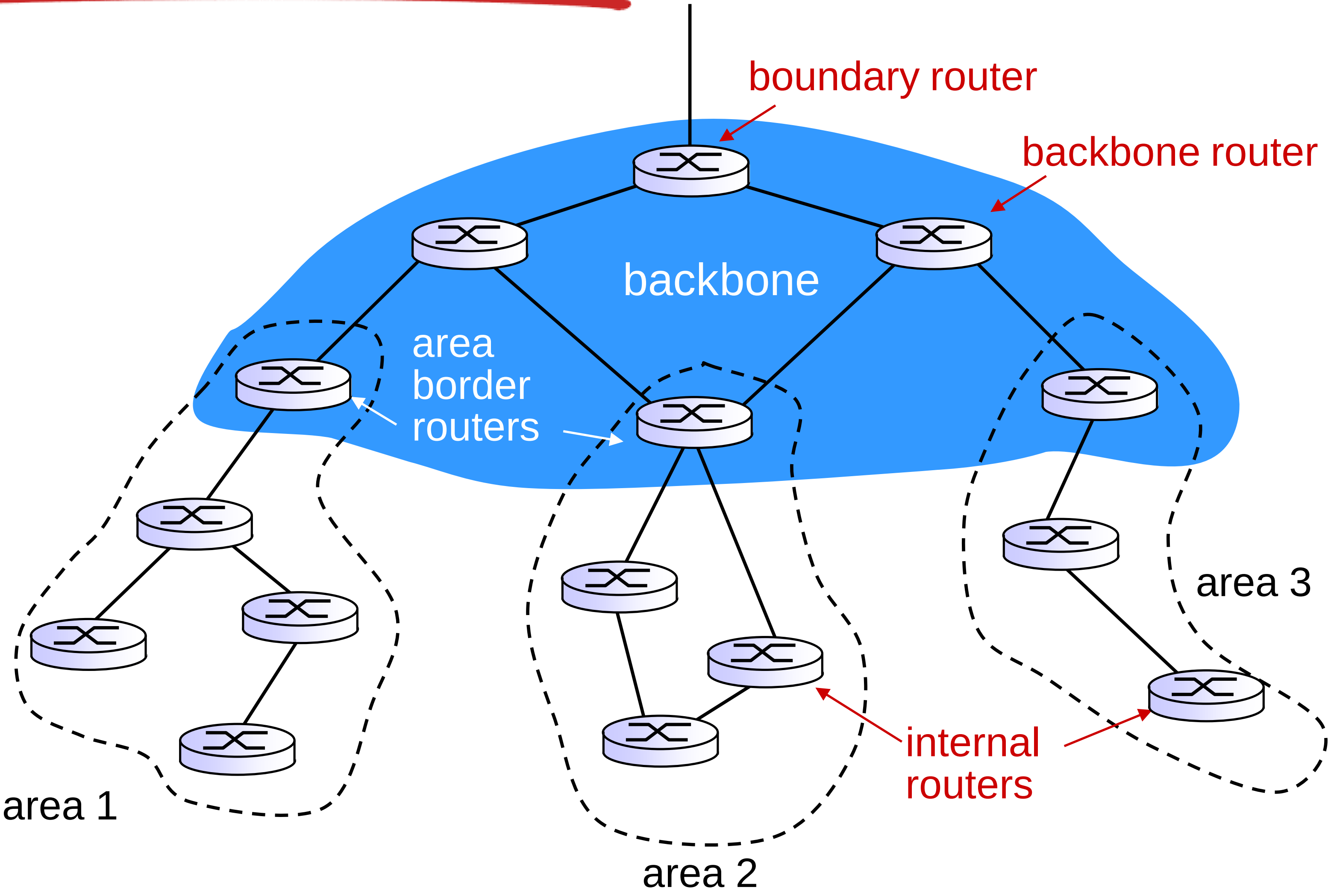
OSPF (Open Shortest Path First)

- "open": publicly available
- uses link state algorithm
  - LS packet dissemination
  - topology map at each node
  - route computation using Dijkstra's algorithm
- OSPF advertisement carries one entry per neighbor
- advertisements flooded to *entire* AS
  - carried in OSPF messages directly over IP (rather than TCP or UDP
- *IS-IS routing* protocol: nearly identical to OSPF

OSPF "advanced" features (not in RIP)

- *security:* all OSPF messages authenticated (to prevent malicious intrusion)
- multiple same-cost paths allowed (only one path in RIP)
- for each link, multiple cost metrics for different TOS (e.g., satellite link cost set "low" for best effort ToS; high for real time ToS)
- integrated uni- and multicast support:
  - Multicast OSPF (MOSPF) uses same topology data base as OSPF
- hierarchical OSPF in large domains.

Hierarchical OSPF



Hierarchical OSPF

- *two-level hierarchy:* local area, backbone.
  - link-state advertisements only in area
  - each nodes has detailed area topology; only know direction (shortest path) to nets in other areas.
- *area border routers:* "summarize" distances to nets in own area, advertise to other Area Border routers.
- *backbone routers:* run OSPF routing limited to backbone.
- *boundary routers:* connect to other AS' s.

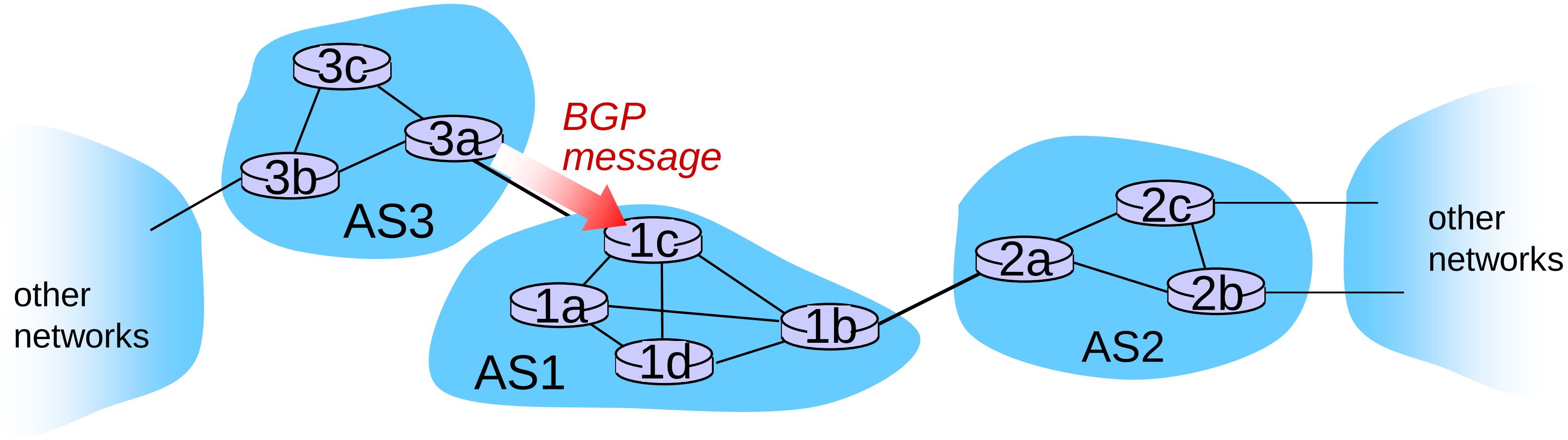
Internet inter-AS routing: BGP

- BGP (Border Gateway Protocol): *the* de facto inter-domain routing protocol
  - "glue that holds the Internet together"
- BGP provides each AS a means to:
  - eBGP: obtain subnet reachability information from neighboring ASs.
  - iBGP: propagate reachability information to all AS-internal routers.
  - determine "good" routes to other networks based on reachability information and policy.
- allows subnet to advertise its existence to rest of Internet: *"I am here"*

BGP basics

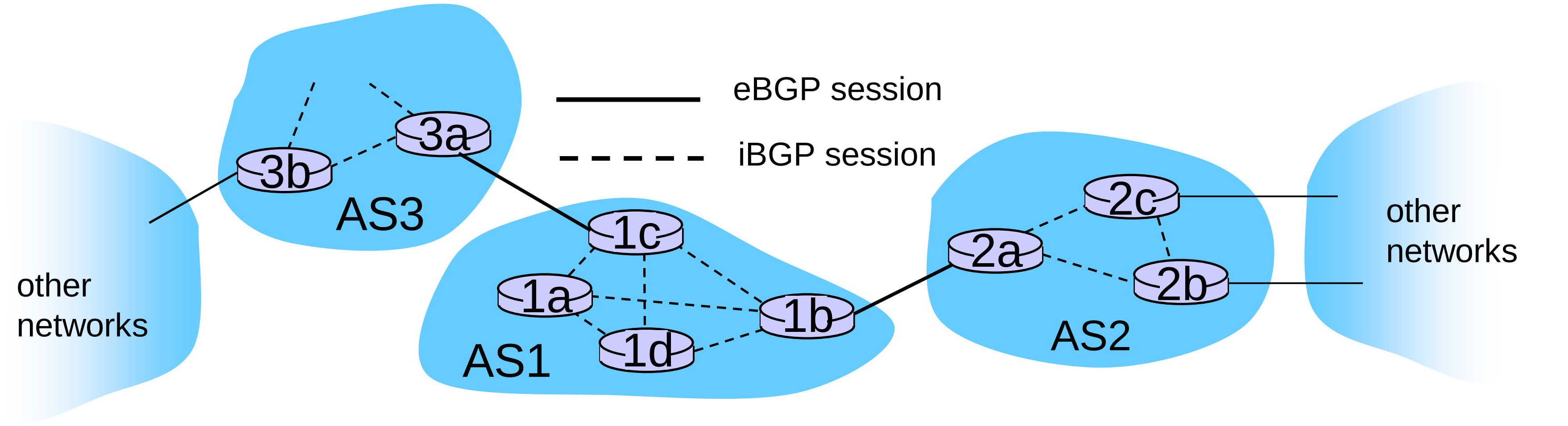
- BGP session: two BGP routers ("peers") exchange BGP messages:
  - advertising *paths* to different destination network prefixes ("path vector" protocol)
  - exchanged over semi-permanent TCP connections

- when AS3 advertises a prefix to AS1:
  - AS3 *promises* it will forward datagrams towards that prefix
  - AS3 can aggregate prefixes in its advertisement



BGP basics: distributing path information

- using eBGP session between 3a and 1c, AS3 sends prefix reachability info to AS1.
  - 1c can then use iBGP do distribute new prefix info to all routers in AS1
  - 1b can then re-advertise new reachability info to AS2 over 1b-to-2a eBGP session
- when router learns of new prefix, it creates entry for prefix in its forwarding table.



Path attributes and BGP routes

- advertised prefix includes BGP attributes
  - prefix + attributes = "route"
- two important attributes:
  - AS-PATH: contains ASs through which prefix advertisement has passed: e.g., AS 67, AS 17
  - NEXT-HOP: indicates specific internal-AS router to next-hop AS. (may be multiple links from current AS to next-hop-AS)
- gateway router receiving route advertisement uses import policy to accept/decline
  - e.g., never route through AS x
  - *policy-based* routing

BGP route selection

- router may learn about more than 1 route to destination AS, selects route based on:
  1. local preference value attribute: policy decision
  2. shortest AS-PATH
  3. closest NEXT-HOP router: hot potato routing
  4. additional criteria

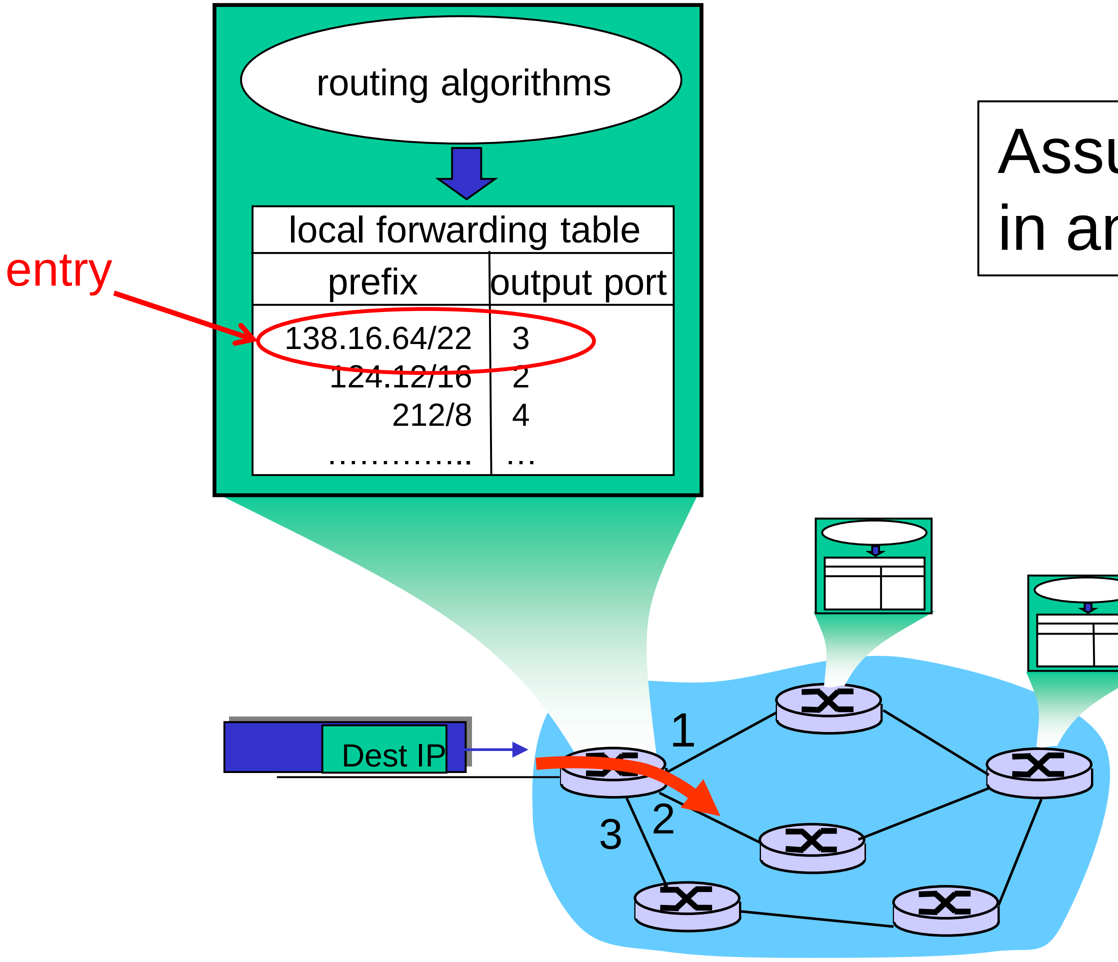
BGP messages

- BGP messages exchanged between peers over TCP connection
- BGP messages:
  - OPEN: opens TCP connection to peer and authenticates sender
  - UPDATE: advertises new path (or withdraws old)
  - KEEPALIVE: keeps connection alive in absence of UPDATES; also ACKs OPEN request
  - NOTIFICATION: reports errors in previous msg; also used to close connection

Putting it Altogether:
How Does an Entry Get Into a Router's Forwarding Table?

- Answer is complicated!
- Ties together hierarchical routing (Section 4.5.3) with BGP (4.6.3) and OSPF (4.6.2).
- Provides nice overview of BGP!

How does entry get in forwarding table?



local forwarding table

prefix	output port
138.16.64/22	3
124.12/16	2
212/8	4
…………..	…

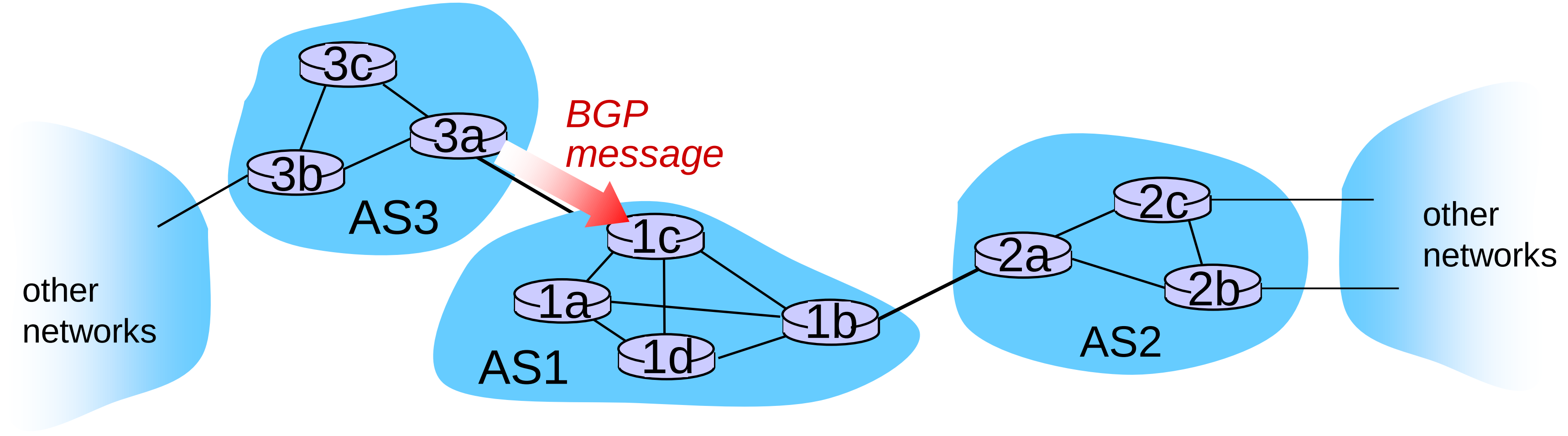
Assume prefix is in another AS.

How does entry get in forwarding table?

High-level overview

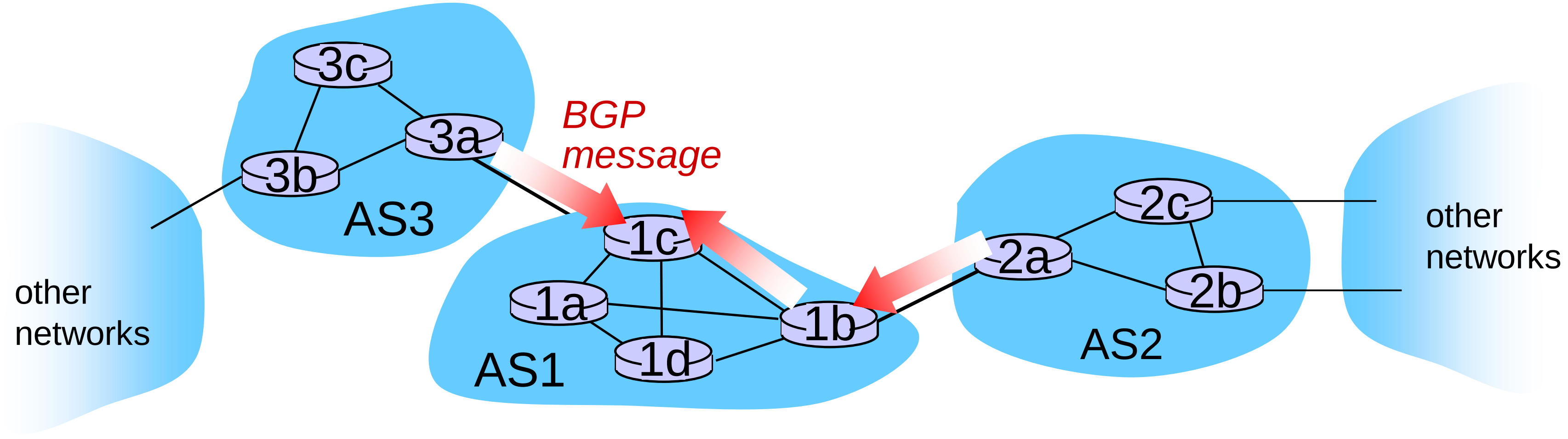
1. Router becomes aware of prefix
2. Router determines output port for prefix
3. Router enters prefix-port in forwarding table

Router becomes aware of prefix



- BGP message contains "routes"
- "route" is a prefix and attributes: AS-PATH, NEXT-HOP,…
- Example: route:
  - Prefix:138.16.64/22 ; AS-PATH: AS3 AS131 ; NEXT-HOP: 201.44.13.125

Router may receive multiple routes



- Router may receive multiple routes for same prefix
- Has to select one route

Select best BGP route to prefix

- Router selects route based on shortest AS-PATH

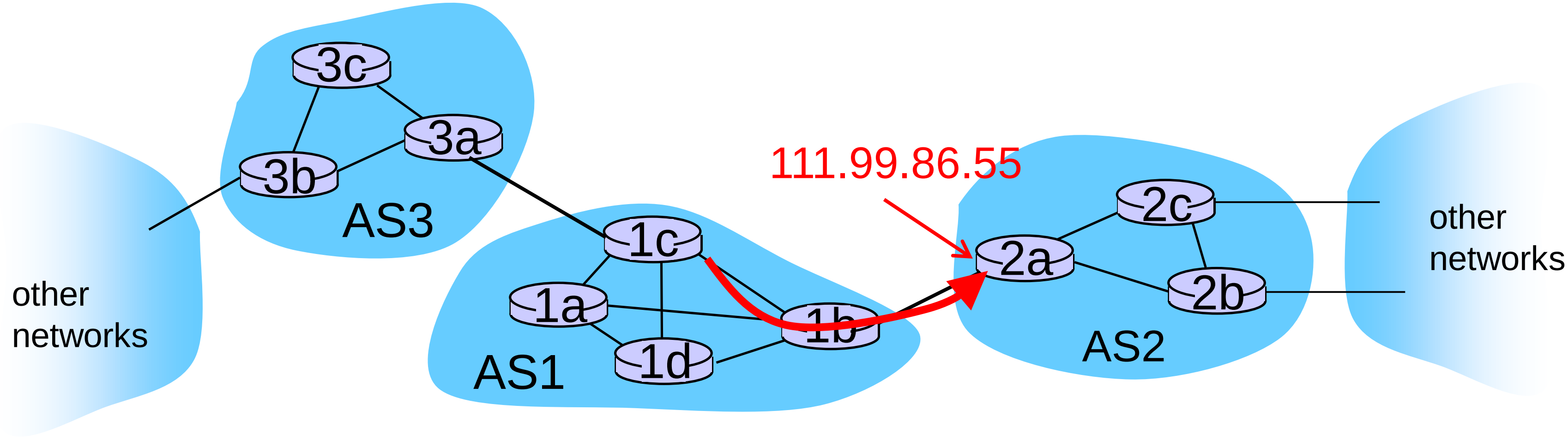
- Example:
  - AS2 AS17 to 138.16.64/22

    select
  - AS3 AS131 AS201 to 138.16.64/22

- What if there is a tie? We’ ll come back to that!

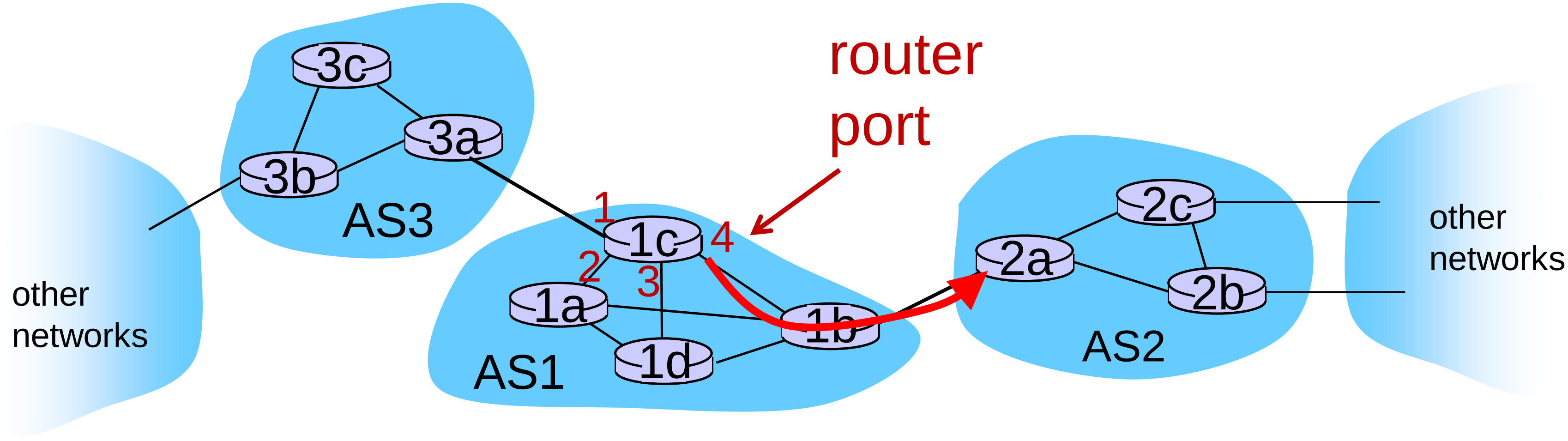
Find best intra-route to BGP route

- Use selected route’s NEXT-HOP attribute
  - Route’s NEXT-HOP attribute is the IP address of the router interface that begins the AS PATH.
- Example:
  - AS-PATH: AS2 AS17 ; NEXT-HOP: 111.99.86.55
- Router uses OSPF to find shortest path from 1c to 111.99.86.55



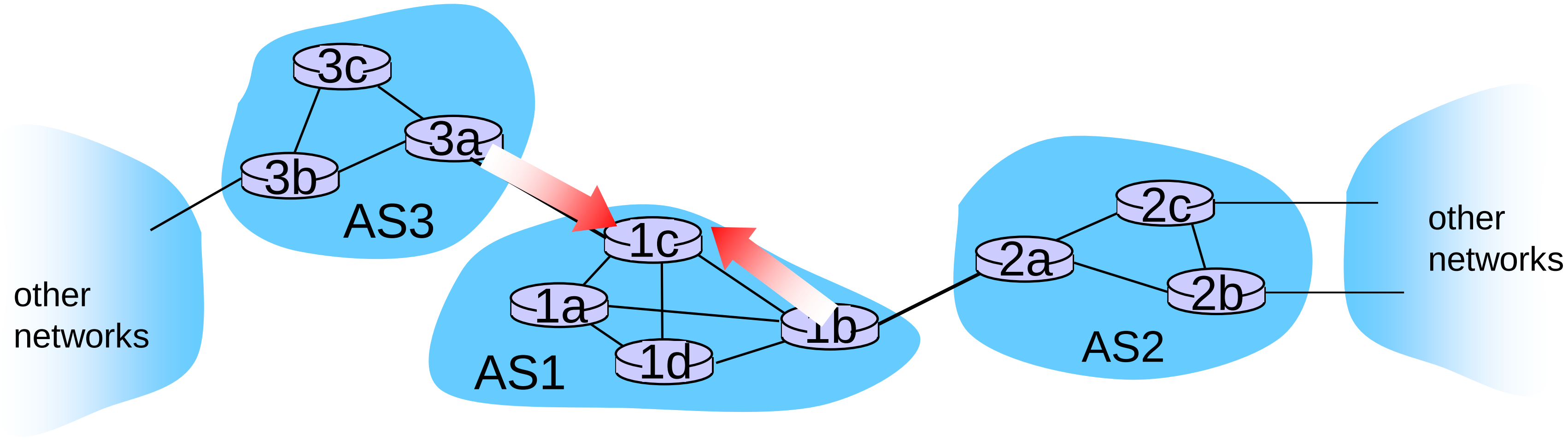
Router identifies port for route

- Identifies port along the OSPF shortest path
- Adds prefix-port entry to its forwarding table:
  - (138.16.64/22 , port 4)



Hot Potato Routing

- Suppose there two or more best inter-routes.
- Then choose route with closest NEXT-HOP
  - Use OSPF to determine which gateway is closest
  - Q: From 1c, chose AS3 AS131 or AS2 AS17?
  - A: route AS3 AS201 since it is closer

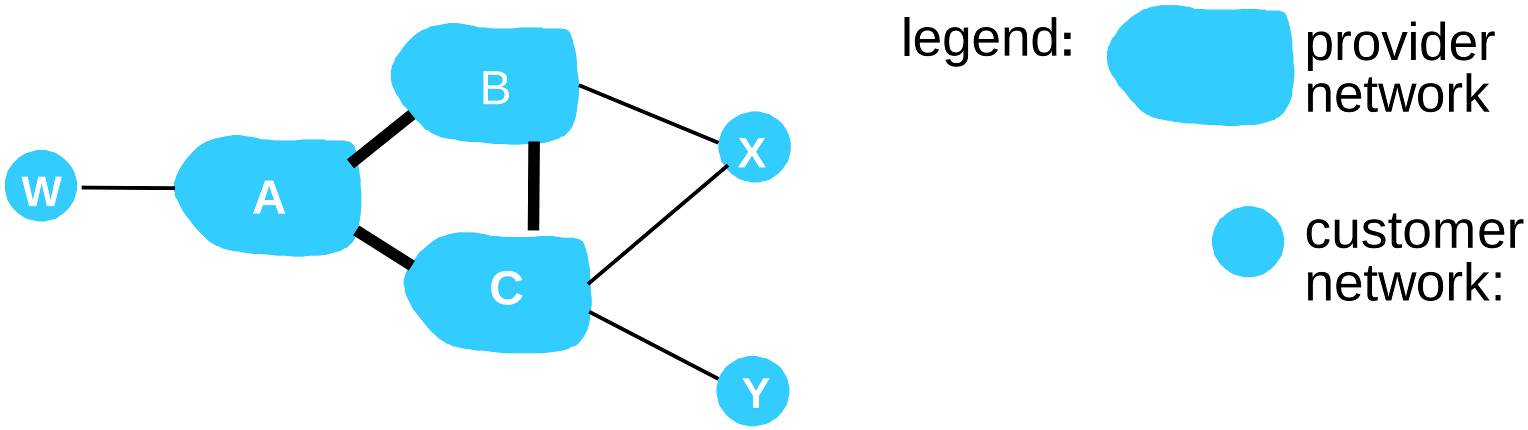


How does entry get in forwarding table?

Summary

1. Router becomes aware of prefix
   - via BGP route advertisements from other routers
2. Determine router output port for prefix
   - Use BGP route selection to find best inter-AS route
   - Use OSPF to find best intra-AS route leading to best inter-AS route
   - Router identifies router port for that best route
3. Enter prefix-port entry in forwarding table

BGP routing policy

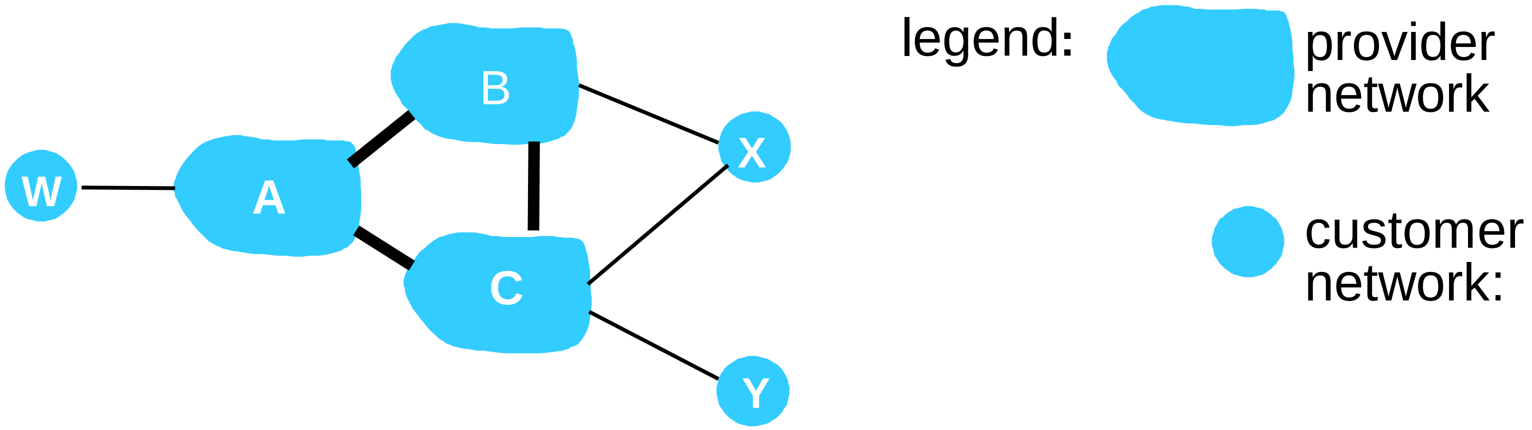


legend: provider network

customer network:

- A,B,C are *provider networks*
- X,W,Y are customer (of provider networks)
- X is *dual-homed:* attached to two networks
  - X does not want to route from B via X to C
  - .. so X will not advertise to B a route to C

BGP routing policy (2)



- A advertises path AW to B
- B advertises path BAW to X
- Should B advertise path BAW to C?
  - No way! B gets no “revenue” for routing CBAW since neither W nor C are B’ s customers
  - B wants to force C to route to w via A
  - B wants to route *only* to/from its customers!

Why different Intra-, Inter-AS routing ?

policy:

- inter-AS: admin wants control over how its traffic routed, who routes through its net.
- intra-AS: single admin, so no policy decisions needed

scale:

- hierarchical routing saves table size, reduced update traffic

performance:

- intra-AS: can focus on performance
- inter-AS: policy may dominate over performance

RAMAIAH
Institute of Technology

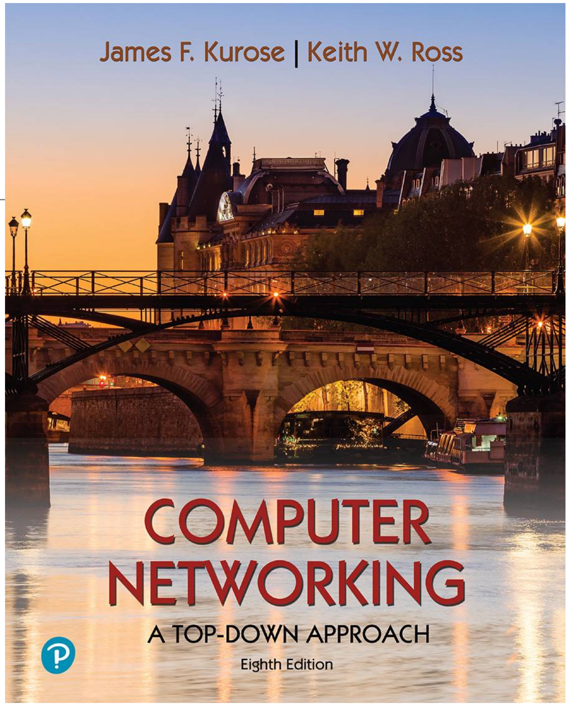
M.S. Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Subject Name: Data Communication & Networking
Subject Code: CS44
Credits: 4:0:0

Addressing

	MAC Address	IP Address	Port Numbers
Layer	Data Link	Network Layer	Transport Layer
Bits	EUI-48 bits EUI-64 bits	IPv4- 32bits IPv6- 128 bits	16 bits
Representation	Hexadecimal Ex:- E8-D8-D1-E9-49-5B	Dotted Decimal Notation 103.109.109.98	Decimal 52751
Uniqueness	Universally Unique	Universally Unique	Unique within host
Address Change from network to network	No	Yes	N/A
Allotment of Address	NIC Manufacturer (IEEE)	Internet Service Provider (IANA –Internet Assigned Numbers Authority)	Operating System
		Private IP and Public IP	Standard Port Numbers

Chapter 3
Transport Layer



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: overview

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

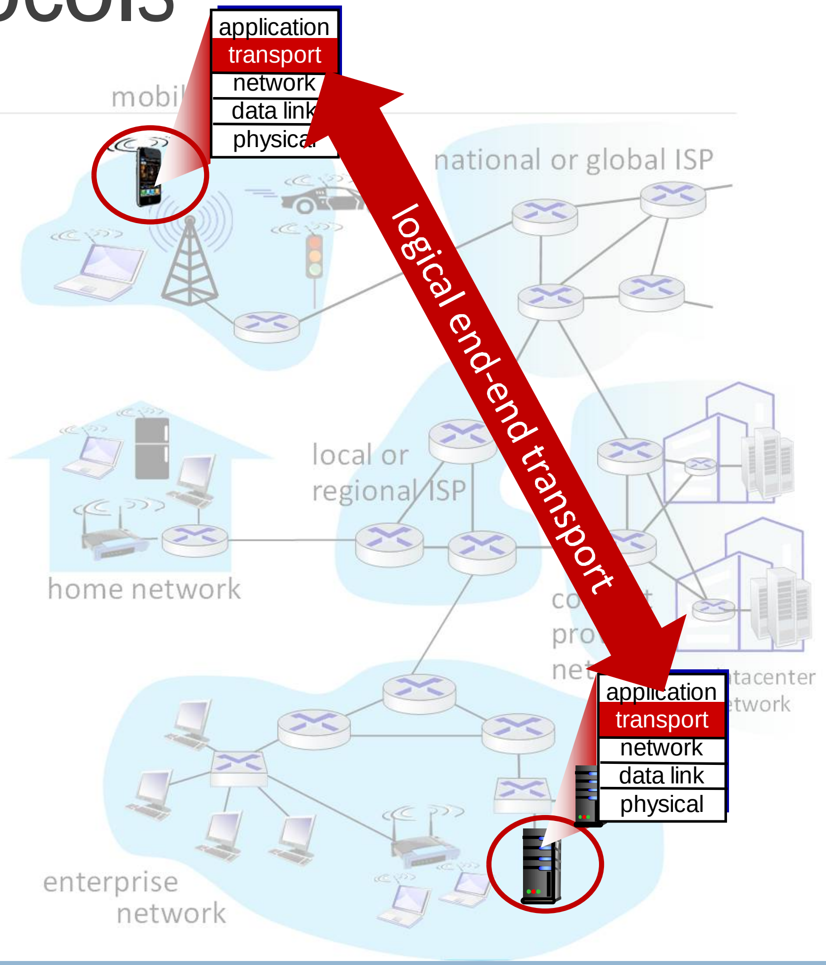
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Transport layer: roadmap

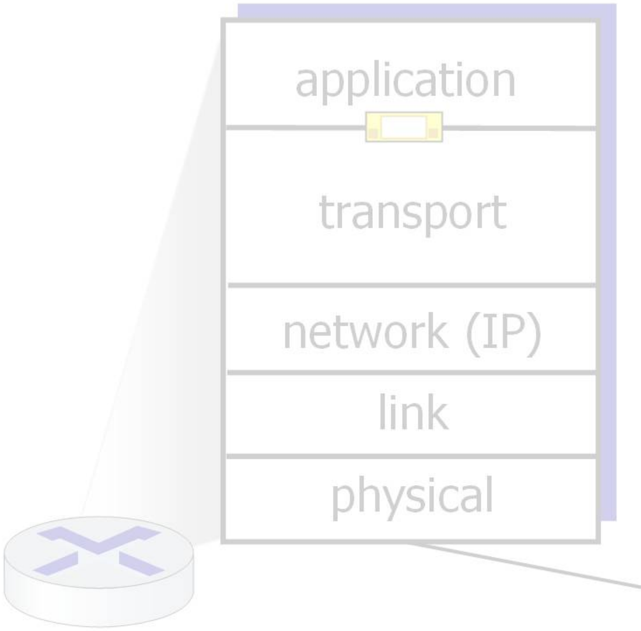
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP

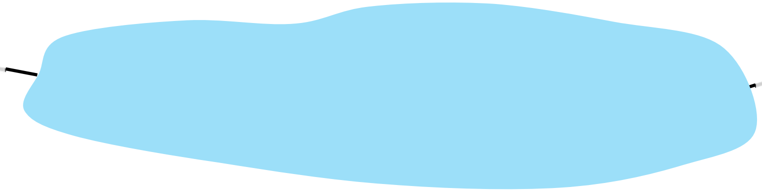


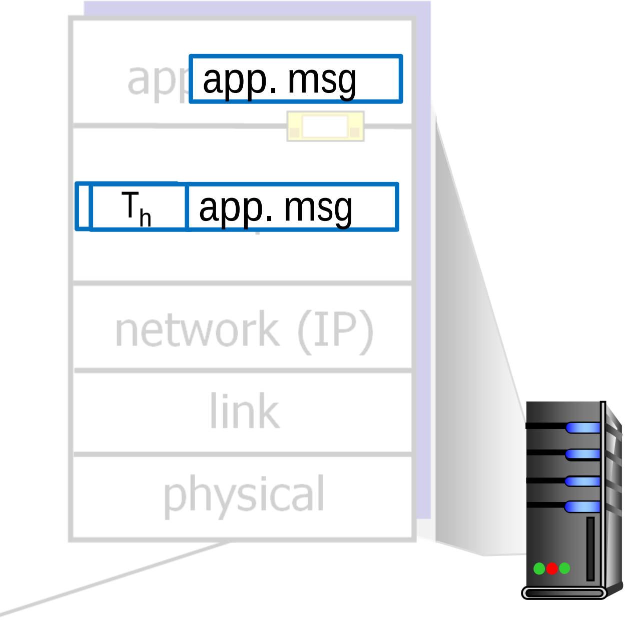
Transport Layer Actions



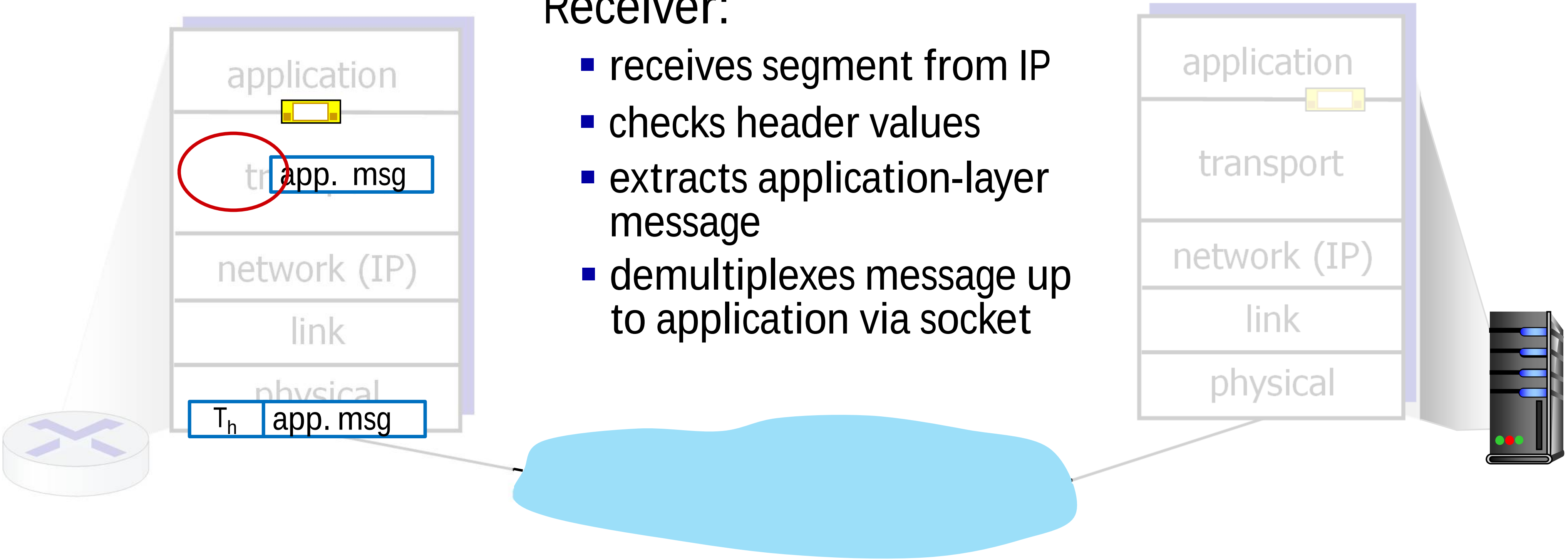
Sender:

- passes an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP





Transport Layer Actions

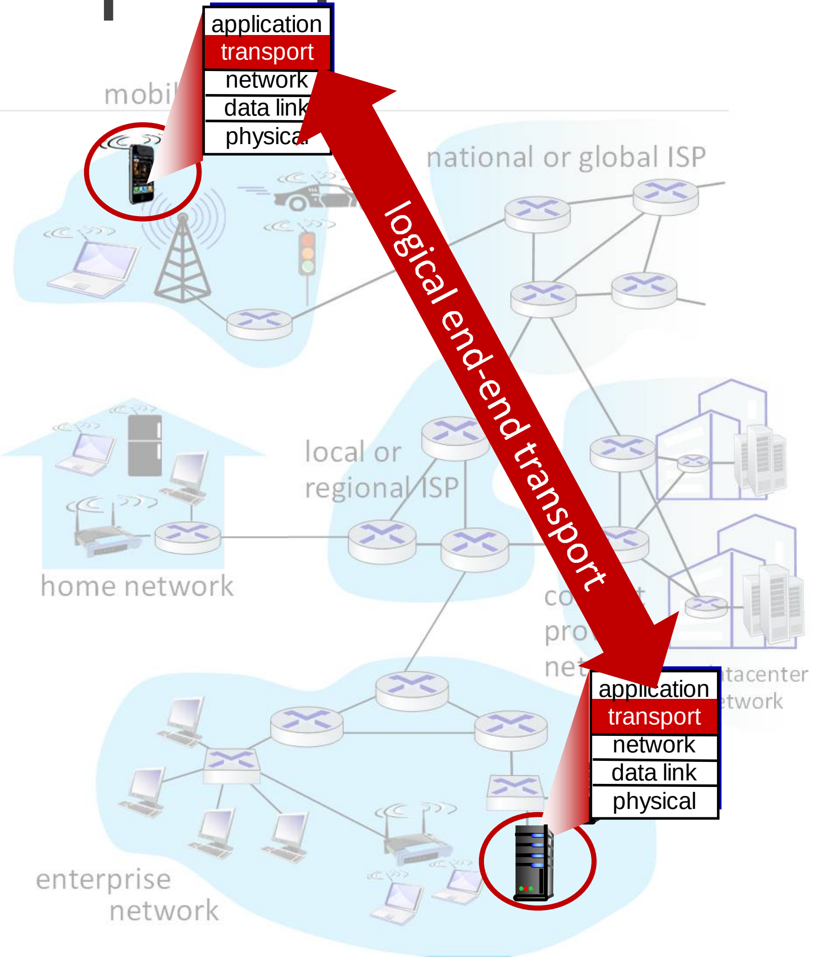


Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- demultiplexes message up to application via socket

Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
  - reliable, in-order delivery
  - congestion control
  - flow control
  - connection setup
- **UDP:** User Datagram Protocol
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Chapter 3: roadmap

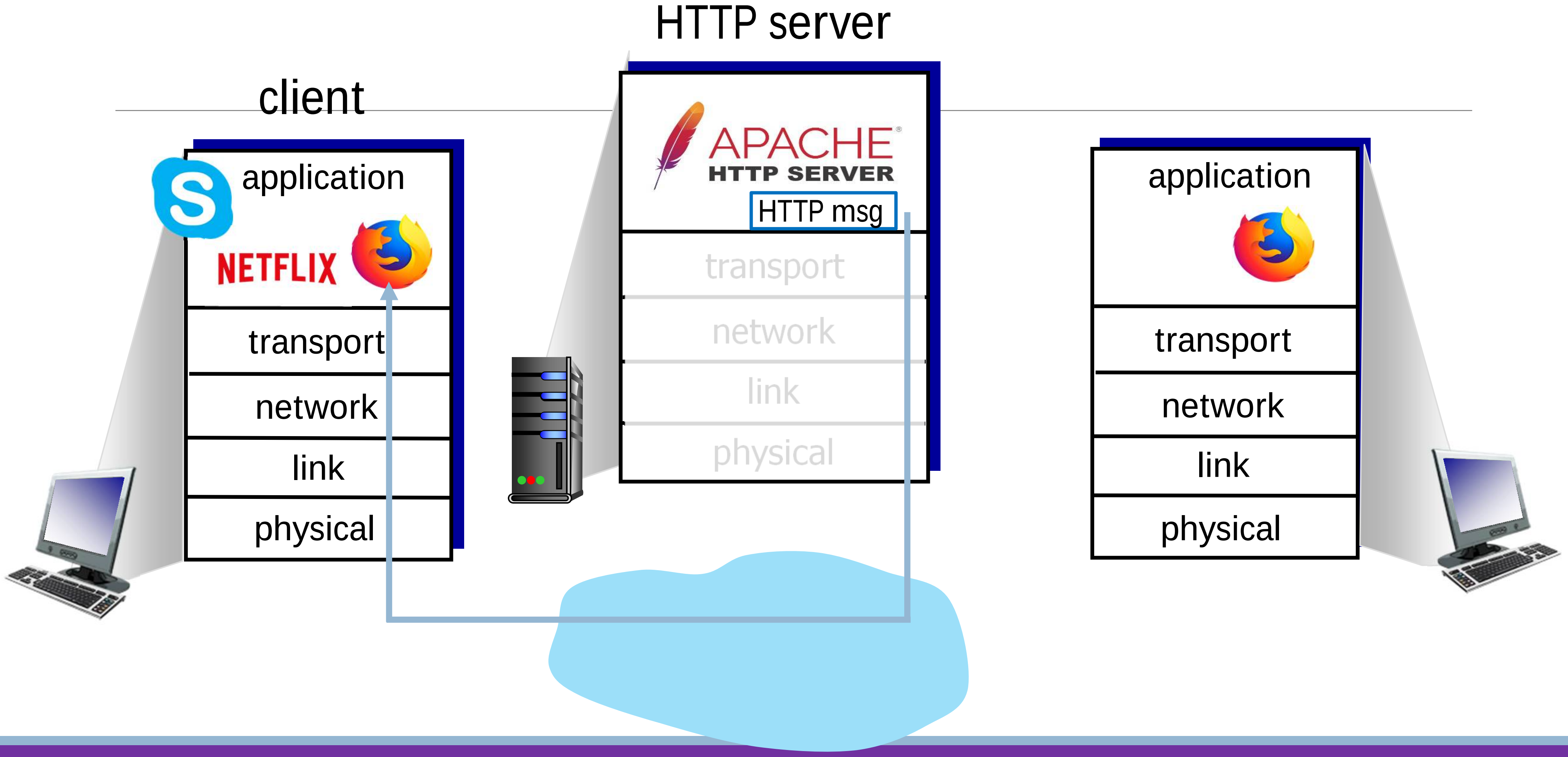
Transport-layer services

Multiplexing and demultiplexing

Connectionless transport: UDP

Connection-oriented transport: TCP

TCP congestion control



HTTP server

client

application

NETFLIX

transport

network

link

physical

APACHE HTTP SERVER

HTTP msg

transport

network

link

physical

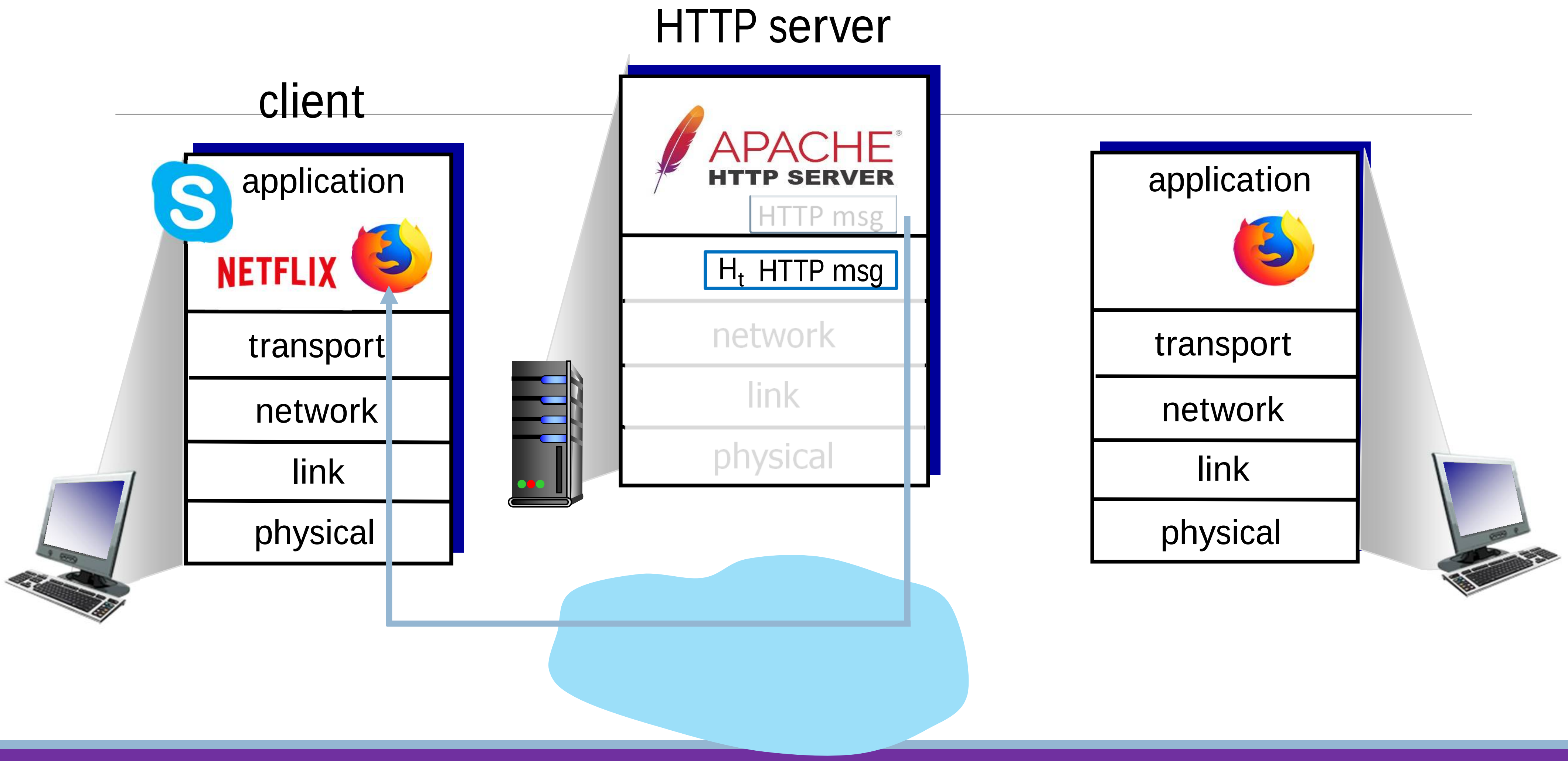
application

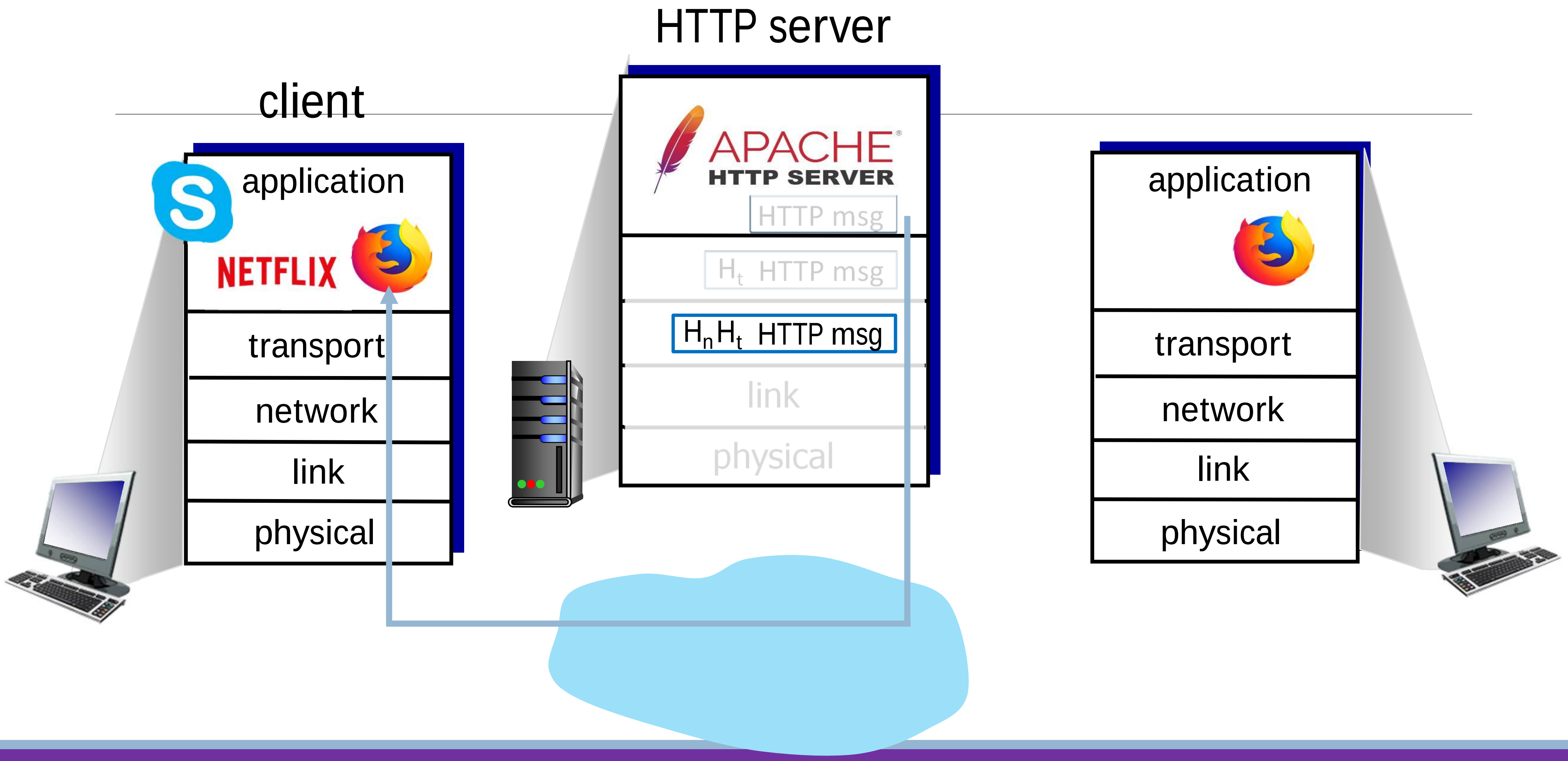
transport

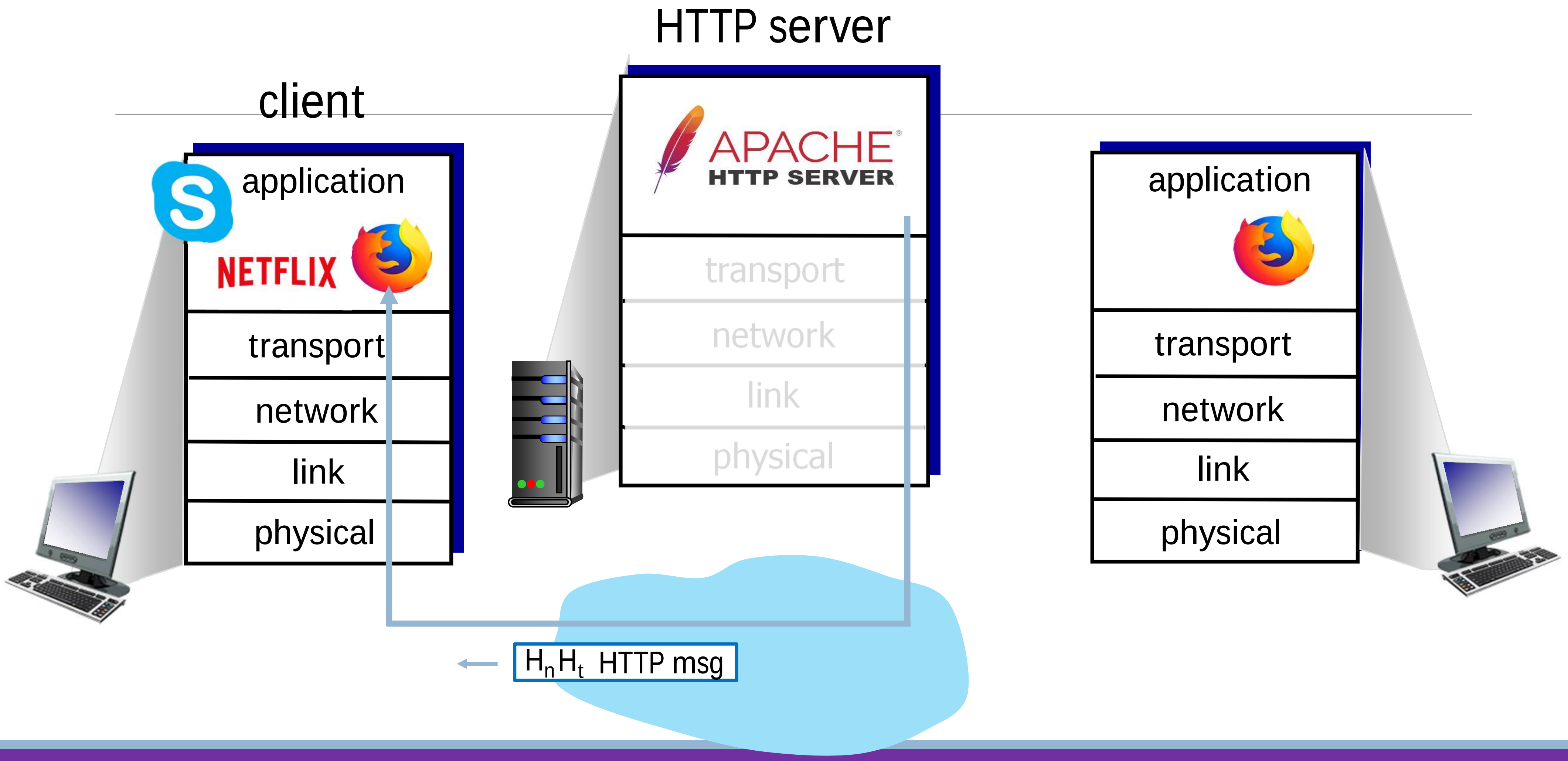
network

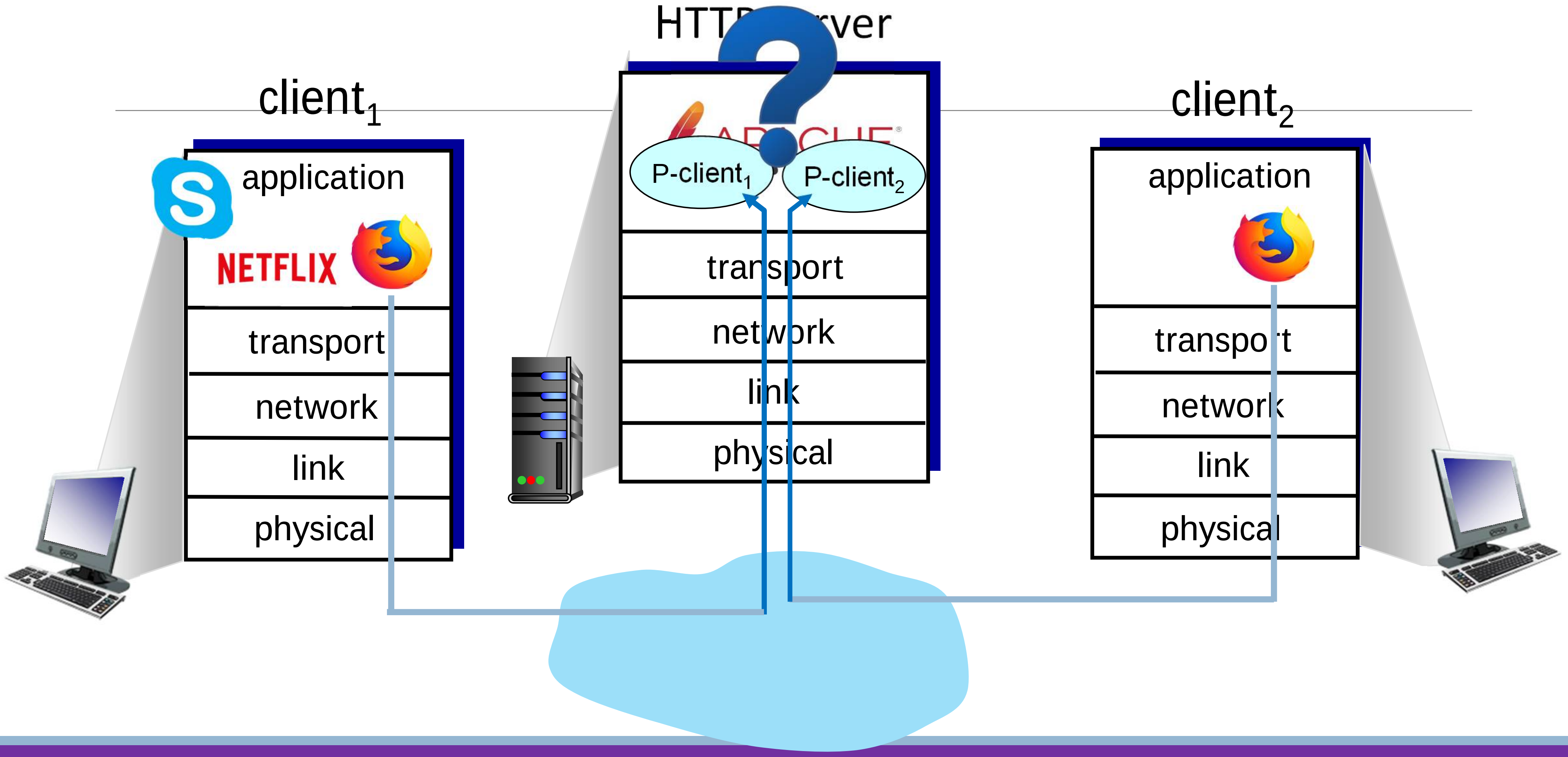
link

physical


HTTP server
client
application
transport
network
link
physical
NETFLIX
APACHE HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
H_nH_t HTTP msg
link
physical
application
transport
network
link
physical


HTTP server
client
application
NETFLIX
transport
network
link
physical
APACHE HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
$H_n H_t$ HTTP msg


HTTP server
client1
client2
application
transport
network
link
physical
NETFLIX
P-client1
P-client2
transport
network
link
physical
application
transport
network
link
physical

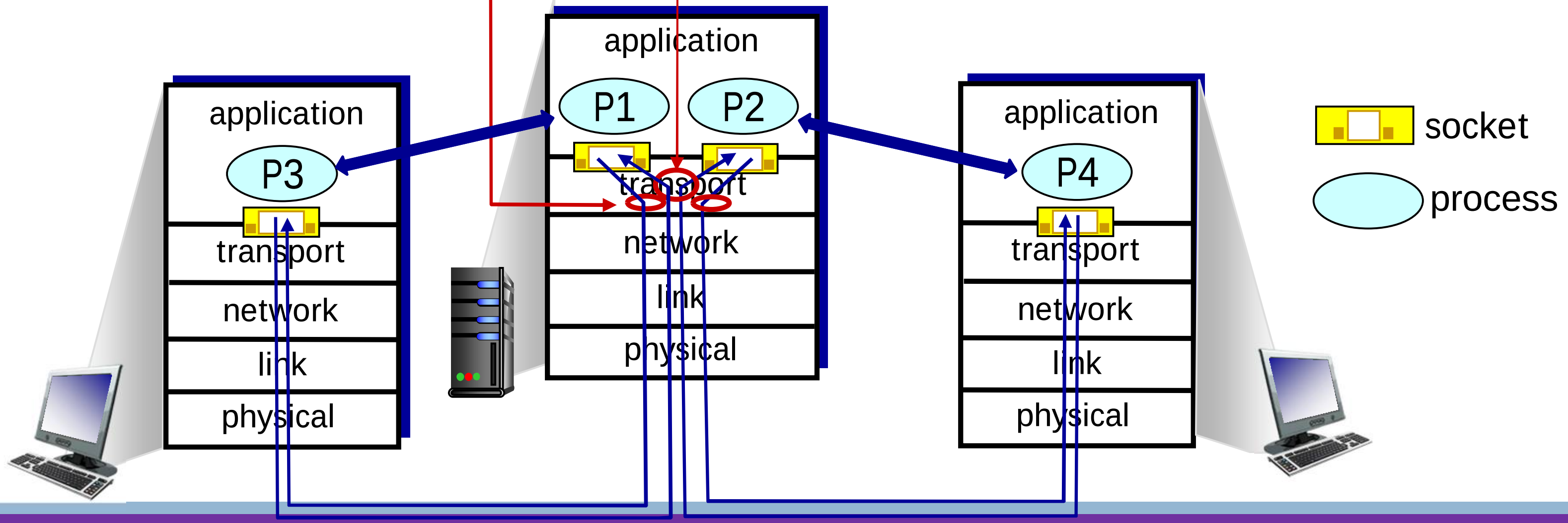
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

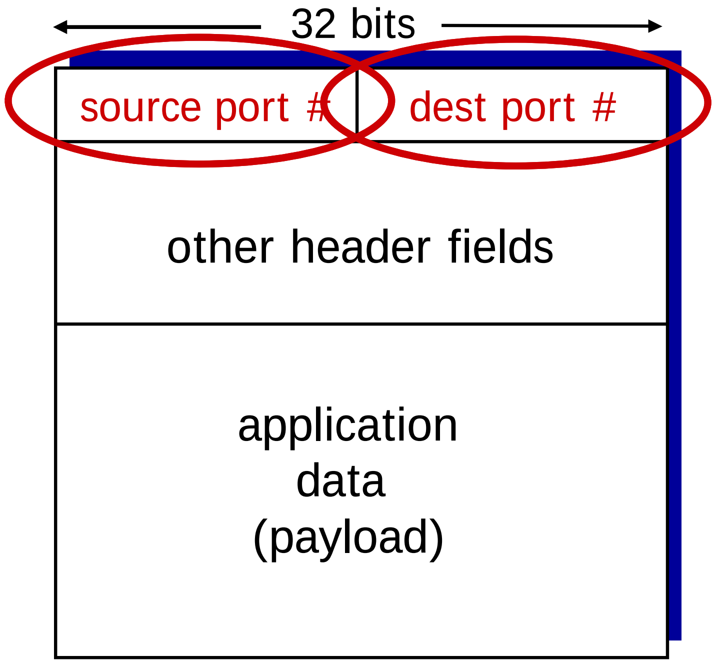
demultiplexing at receiver:

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:

- when creating socket, must specify *host-local* port #:

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

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

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #

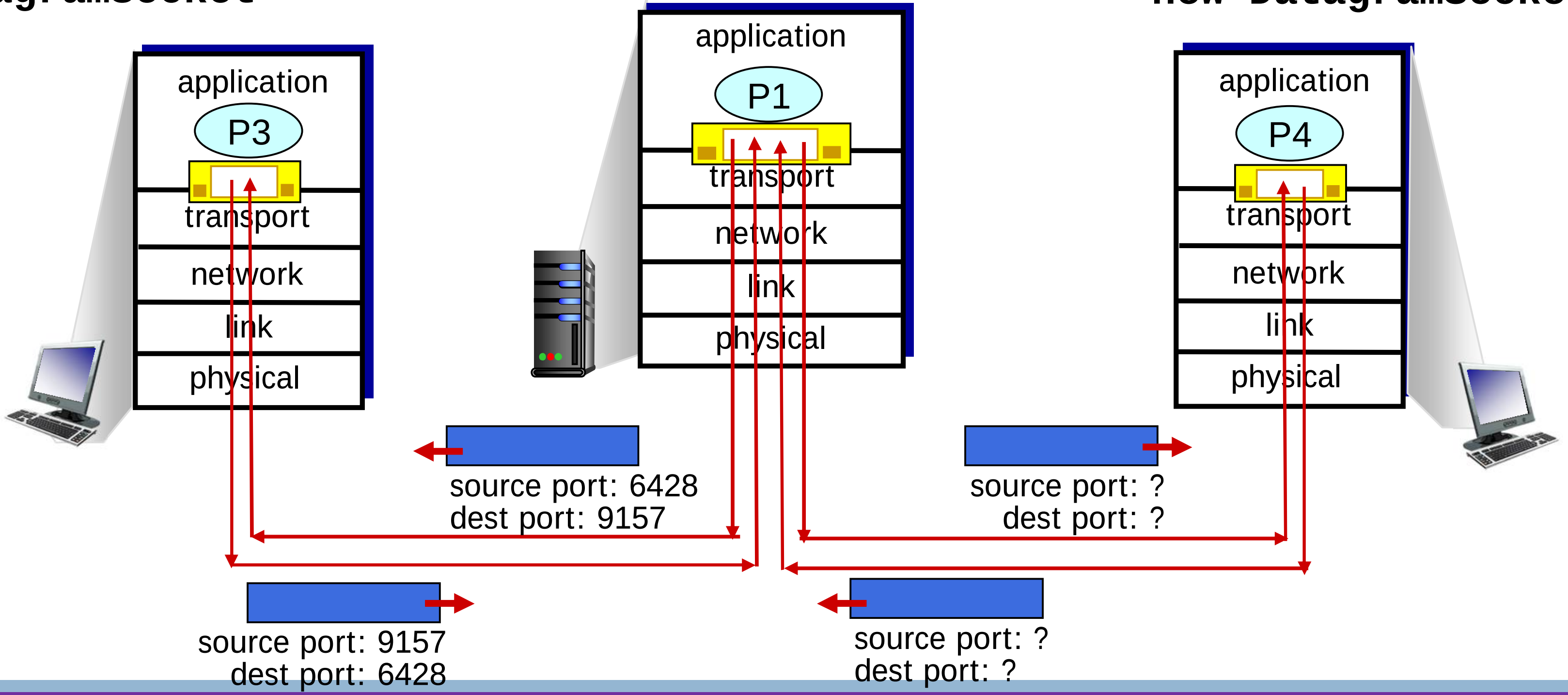
IP/UDP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

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

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

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



source port: 6428
dest port: 9157

source port: ?
dest port: ?

source port: 9157
dest port: 6428

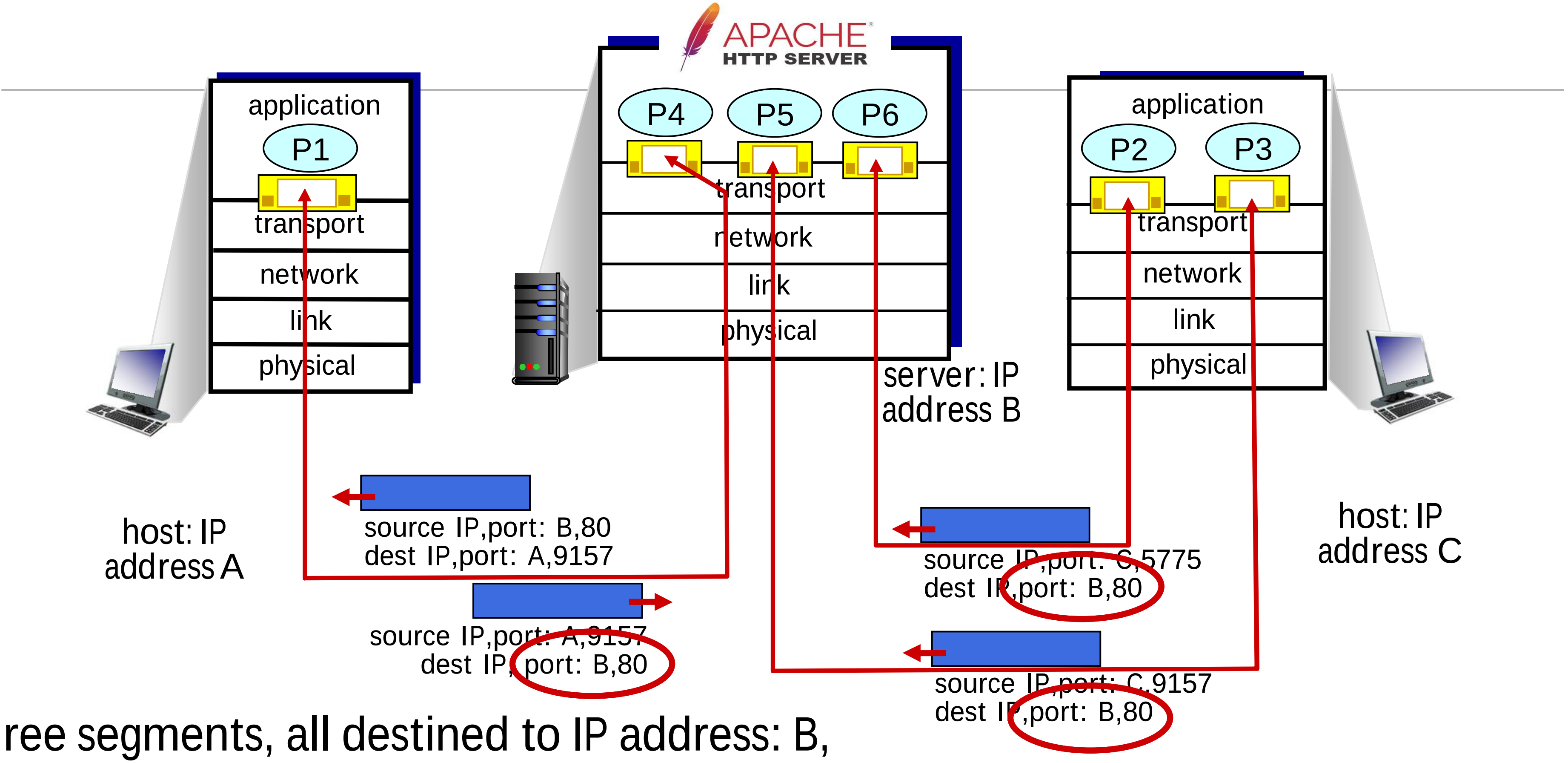
source port: ?
dest port: ?

Connection-oriented demultiplexing

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

- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

RAMAIAH
Institute of Technology

Chapter 3: roadmap

Transport-layer services

Multiplexing and demultiplexing

Connectionless transport: UDP

Connection-oriented transport: TCP

TCP congestion control

UDP: User Datagram Protocol

- "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

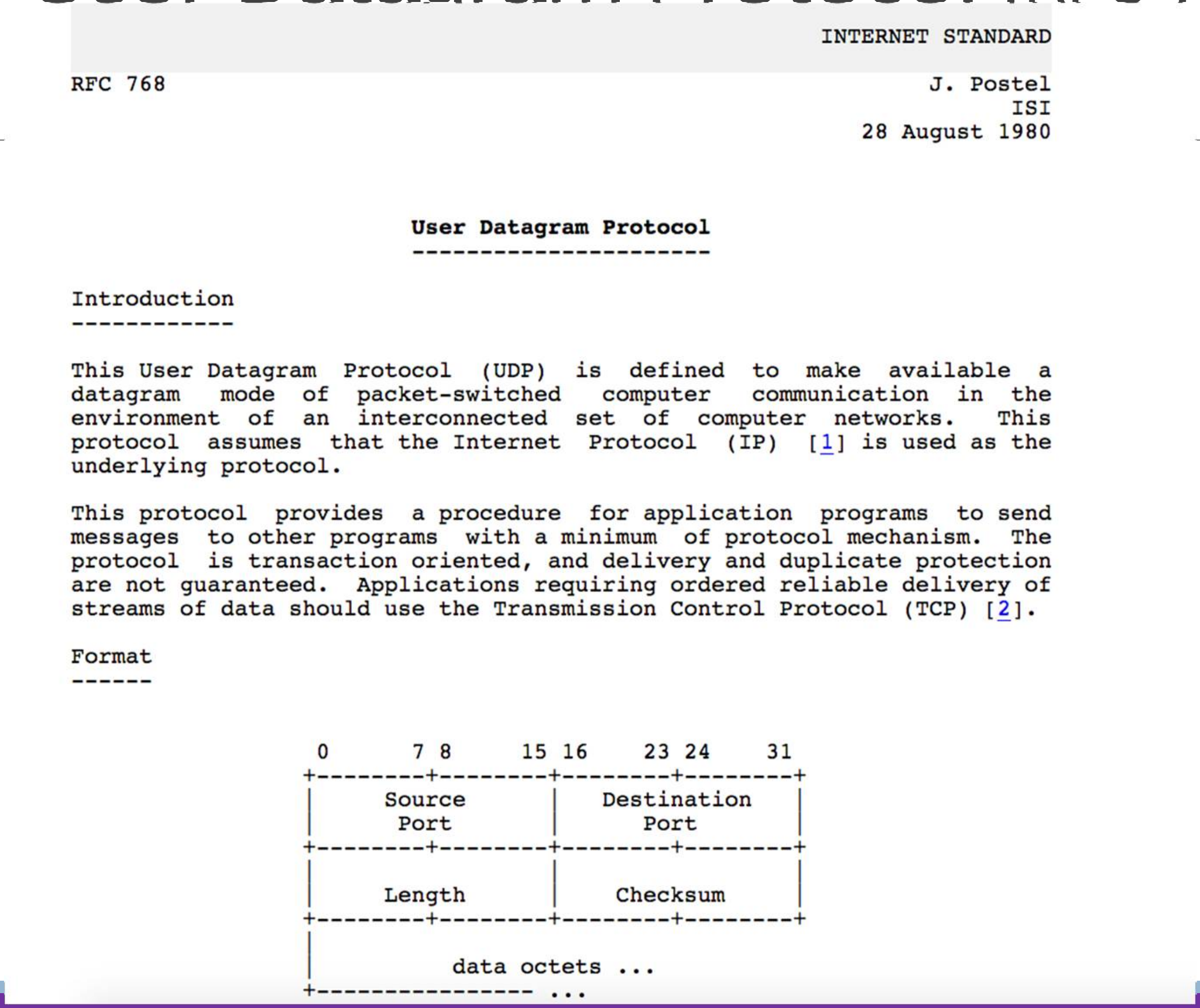
Why is there a UDP?

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

UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- if reliable transfer needed over UDP:
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol [RFC 768]



```
                                                      INTERNET STANDARD

RFC 768                                                        J. Postel
                                                                     ISI
                                                          28 August 1980
```

User Datagram Protocol

Introduction

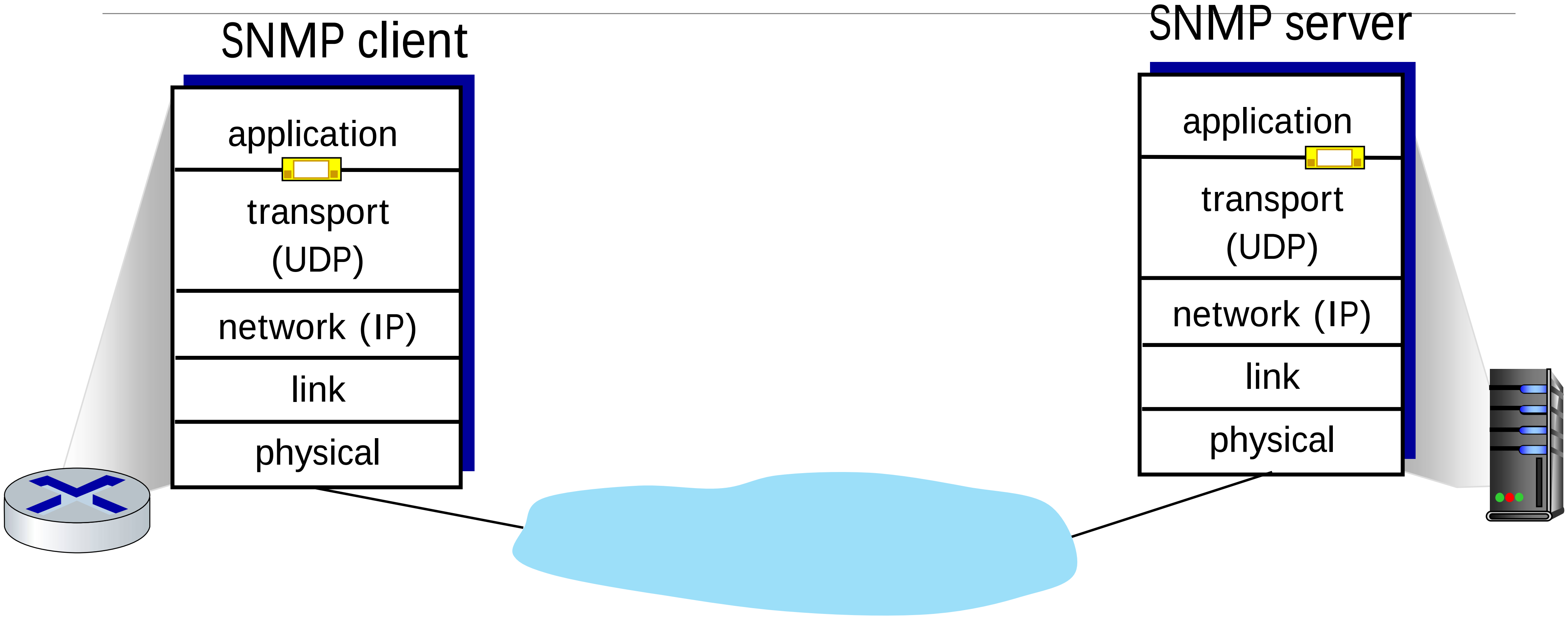
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

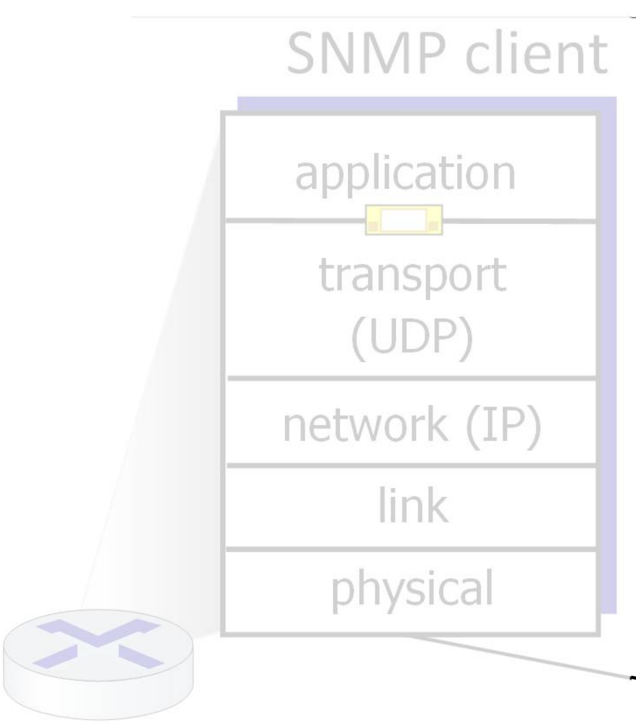
Format

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

UDP: Transport Layer Actions

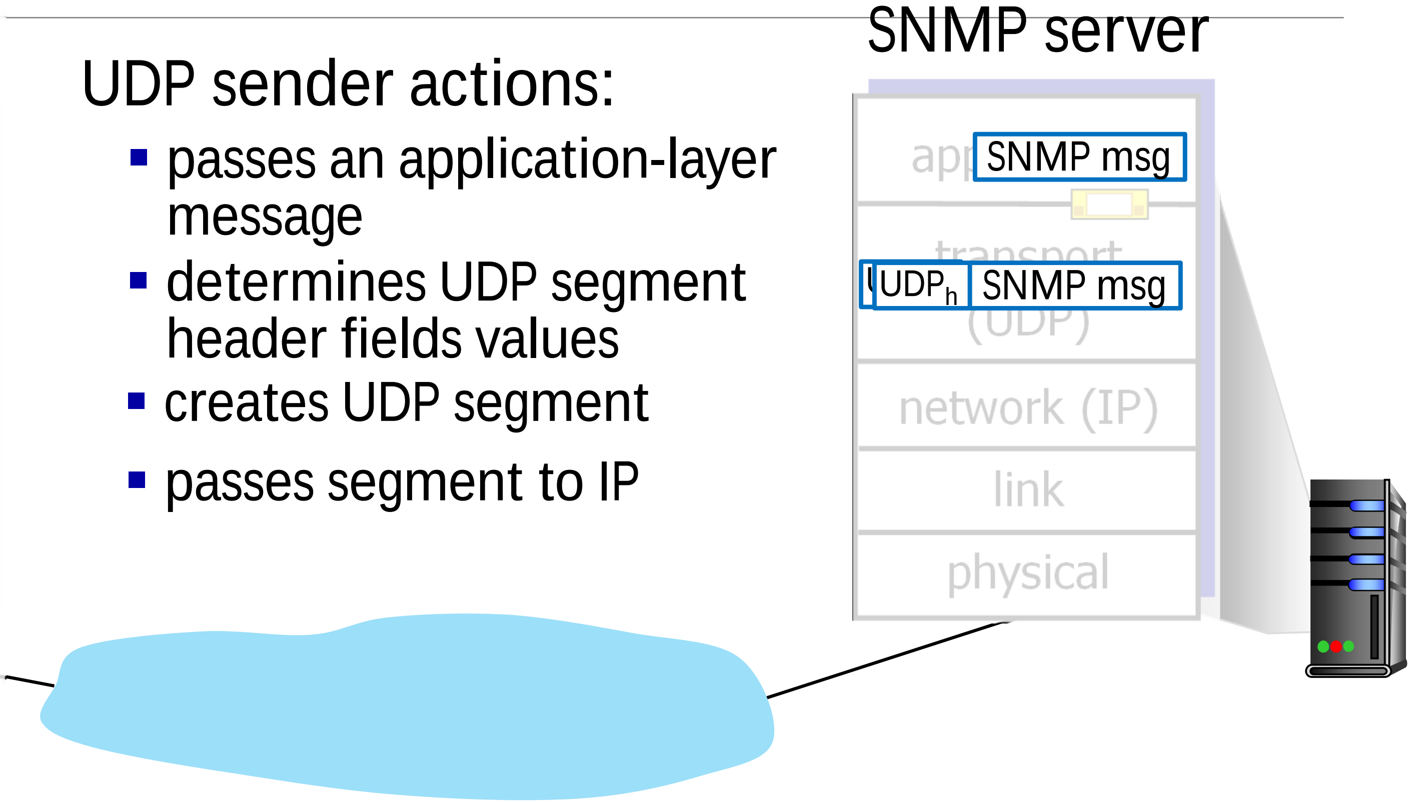


UDP: Transport Layer Actions

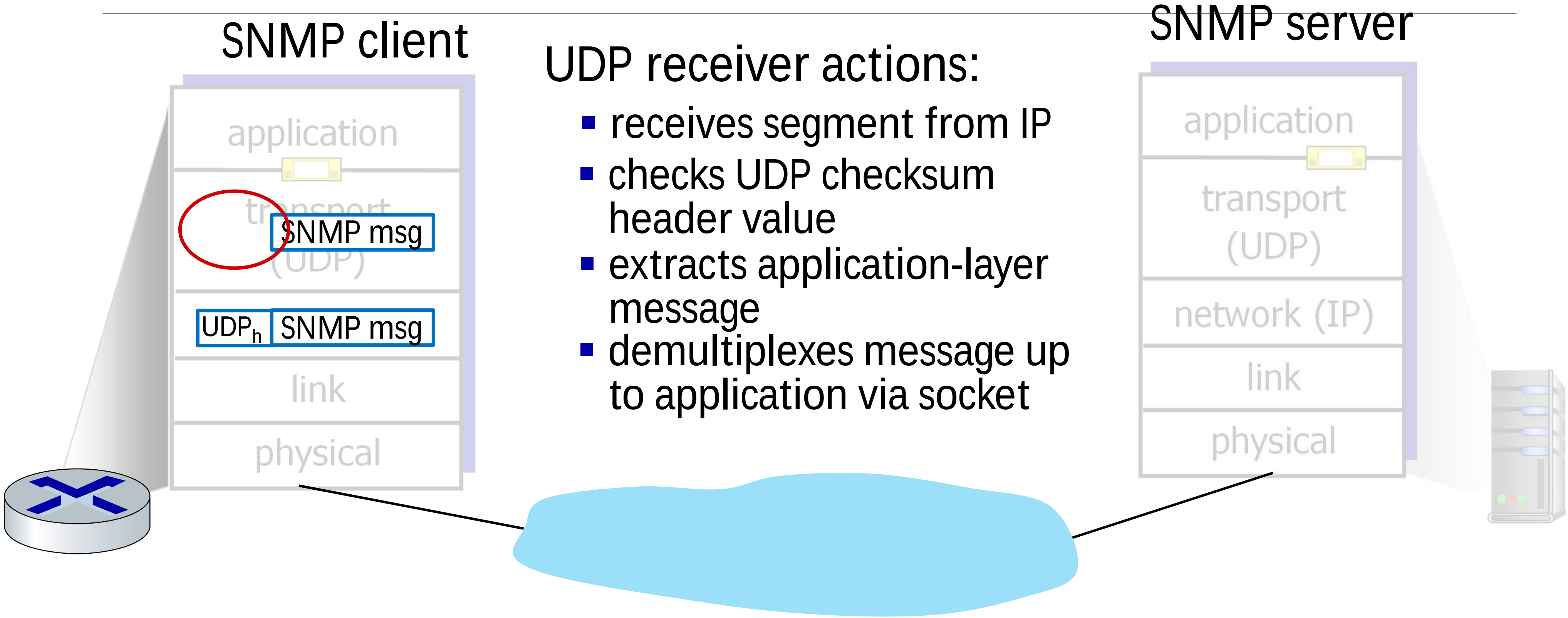


UDP sender actions:

- passes an application-layer message
- determines UDP segment header fields values
- creates UDP segment
- passes segment to IP



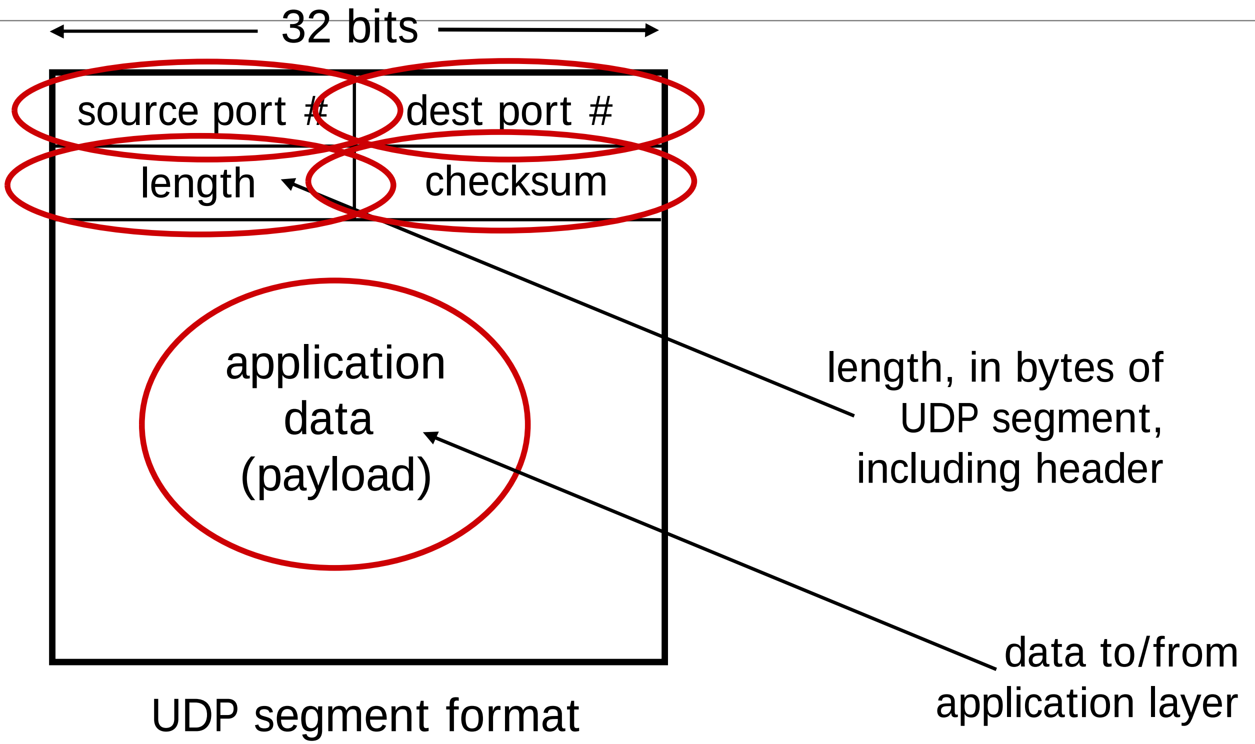
UDP: Transport Layer Actions



UDP receiver actions:

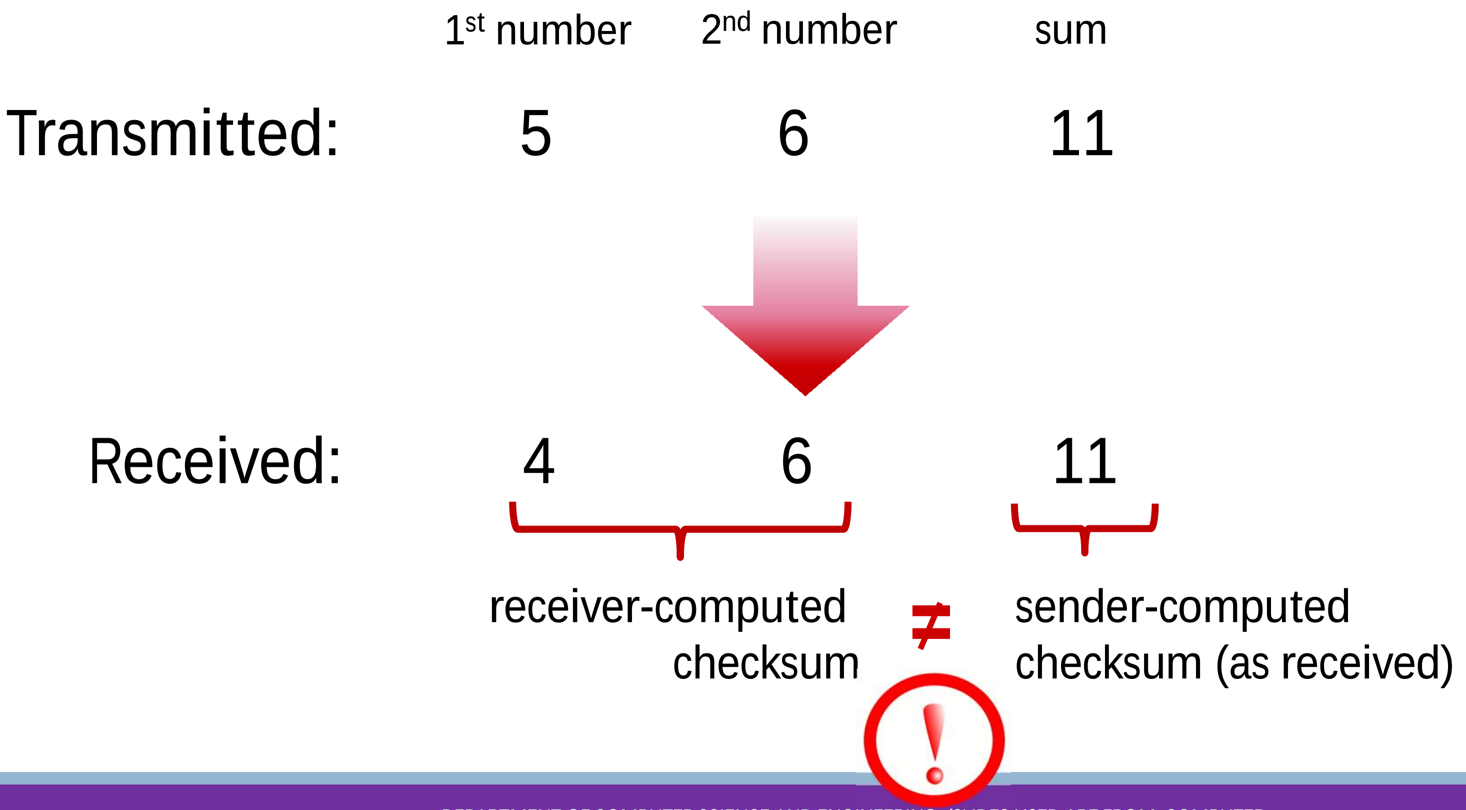
- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP segment header



UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment



	1st number	2nd number	sum
Transmitted:	5	6	11
Received:	4	6	11

receiver-computed checksum ≠ sender-computed checksum (as received)

UDP checksum

Goal: detect errors (*i.e.*, flipped bits) in transmitted segment

sender:

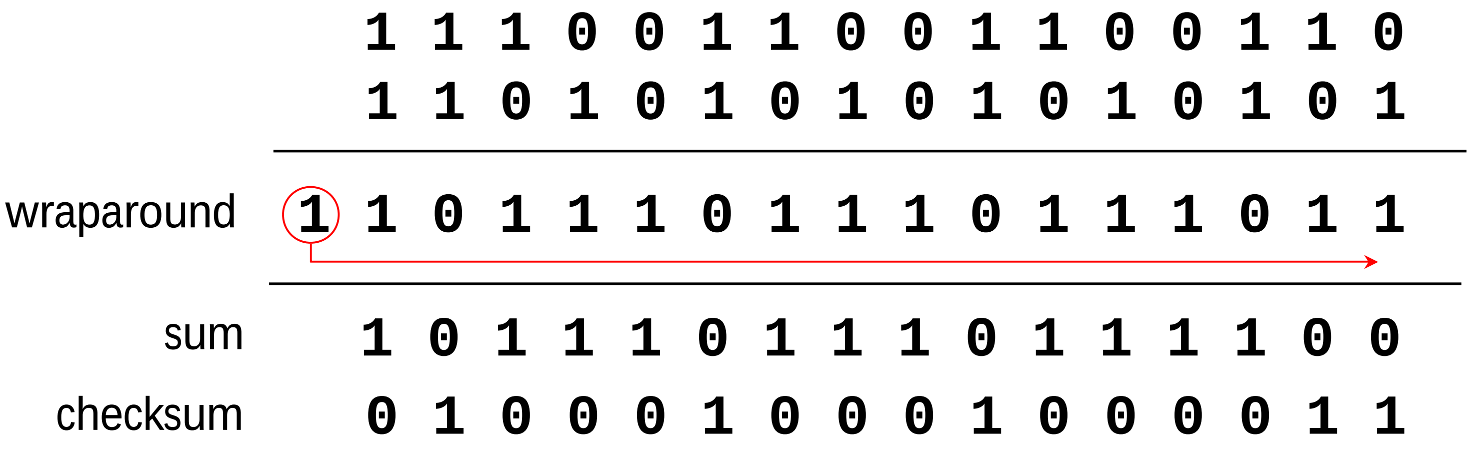
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: an example

example: add two 16-bit integers

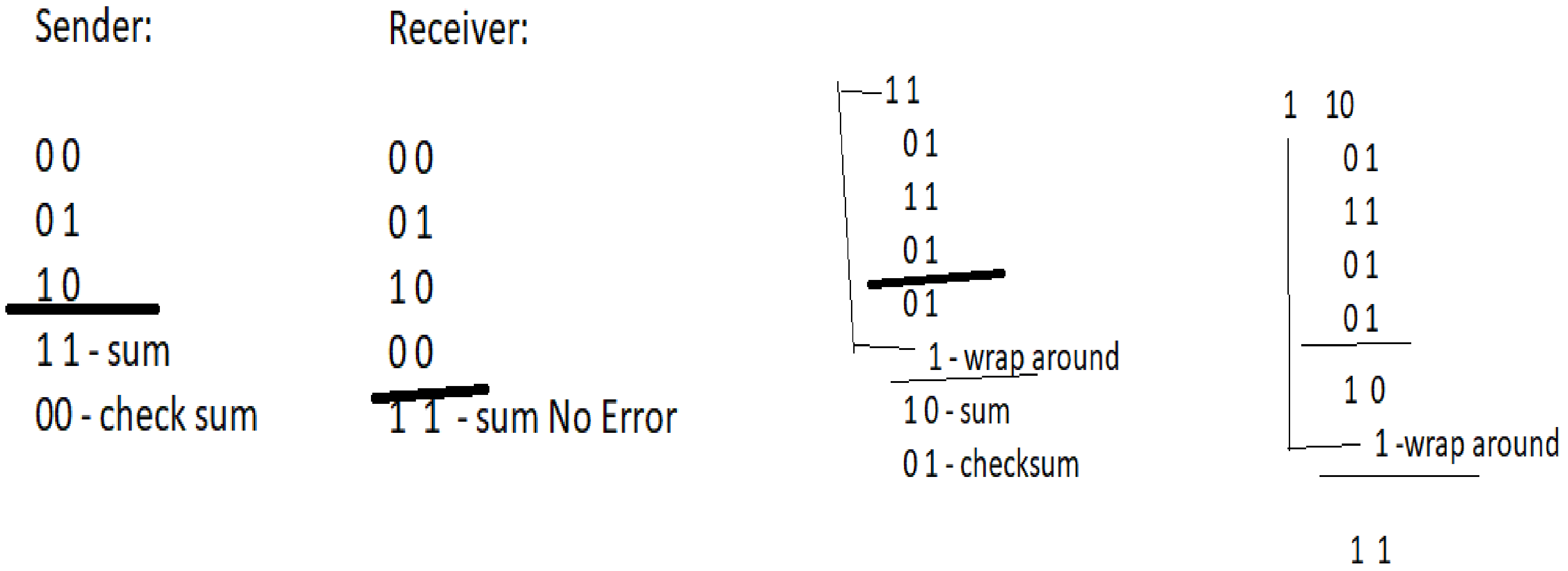


```
              1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
              1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
          ------------------------------------
wraparound  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
          ------------------------------------
sum           1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum      0 1 0 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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Checksum simple example:



Sender:

00
01
10
11 - sum
00 - check sum

Receiver:

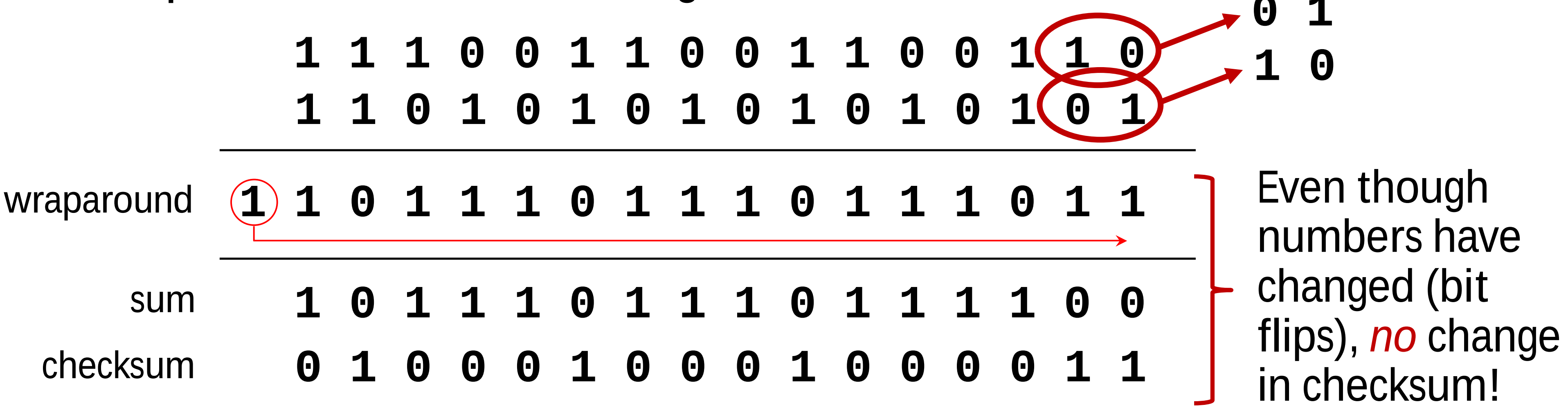
00
01
10
00
1 1 - sum No Error

11
01
11
01
01
1 - wrap around
10 - sum
01 - checksum

1 10
01
11
01
01
1 0
1 -wrap around
1 1

Internet checksum: weak protection!

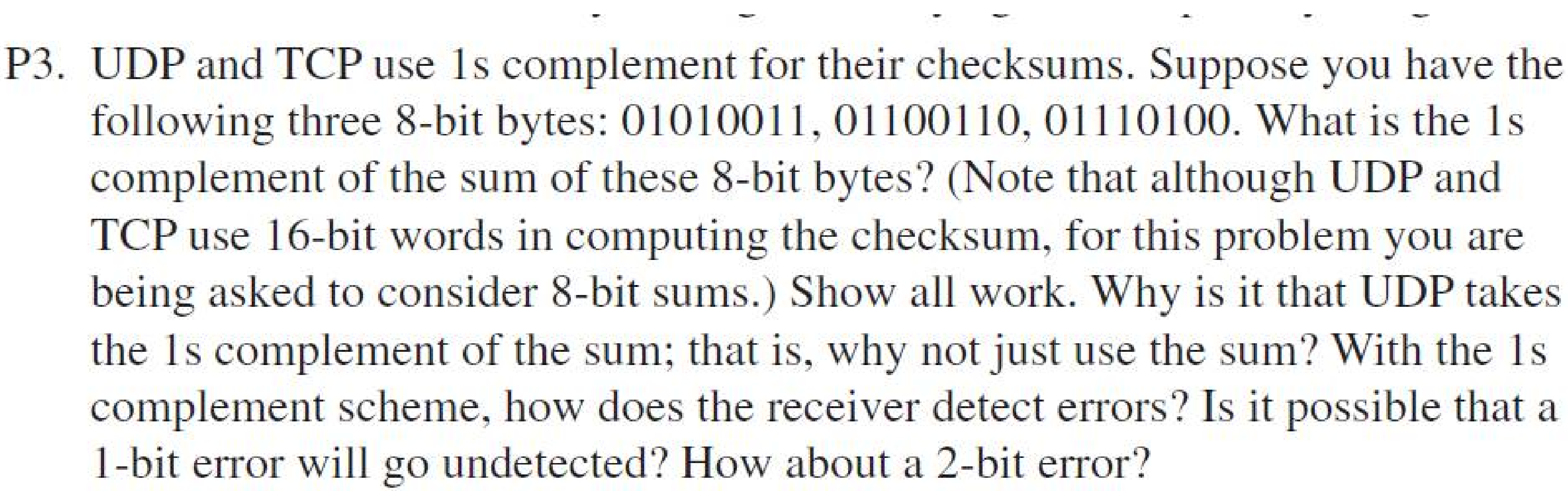
example: add two 16-bit integers



```
              1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0      0 1
              1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1      1 0

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
```

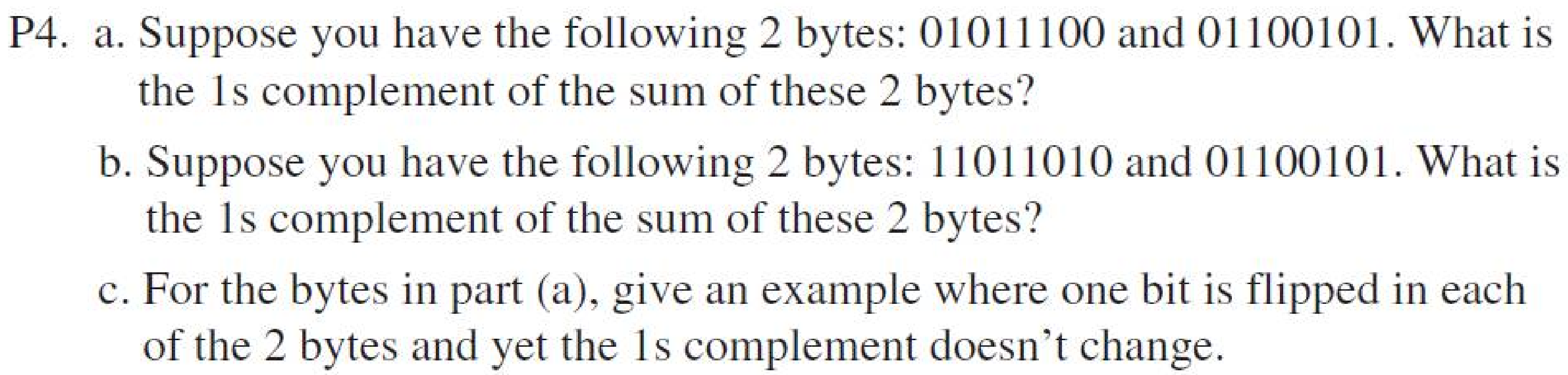
Even though numbers have changed (bit flips), *no* change in checksum!

Problems on Checksum



P3. UDP and TCP use 1s complement for their checksums. Suppose you have the following three 8-bit bytes: 01010011, 01100110, 01110100. What is the 1s complement of the sum of these 8-bit bytes? (Note that although UDP and TCP use 16-bit words in computing the checksum, for this problem you are being asked to consider 8-bit sums.) Show all work. Why is it that UDP takes the 1s complement of the sum; that is, why not just use the sum? With the 1s complement scheme, how does the receiver detect errors? Is it possible that a 1-bit error will go undetected? How about a 2-bit error?

Problems on Checksum (contd.)



P4. a. Suppose you have the following 2 bytes: 01011100 and 01100101. What is the 1s complement of the sum of these 2 bytes?

b. Suppose you have the following 2 bytes: 11011010 and 01100101. What is the 1s complement of the sum of these 2 bytes?

c. For the bytes in part (a), give an example where one bit is flipped in each of the 2 bytes and yet the 1s complement doesn't change.

Summary: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

Chapter 3: roadmap

Transport-layer services

Multiplexing and demultiplexing

Connectionless transport: UDP

Connection-oriented transport: TCP

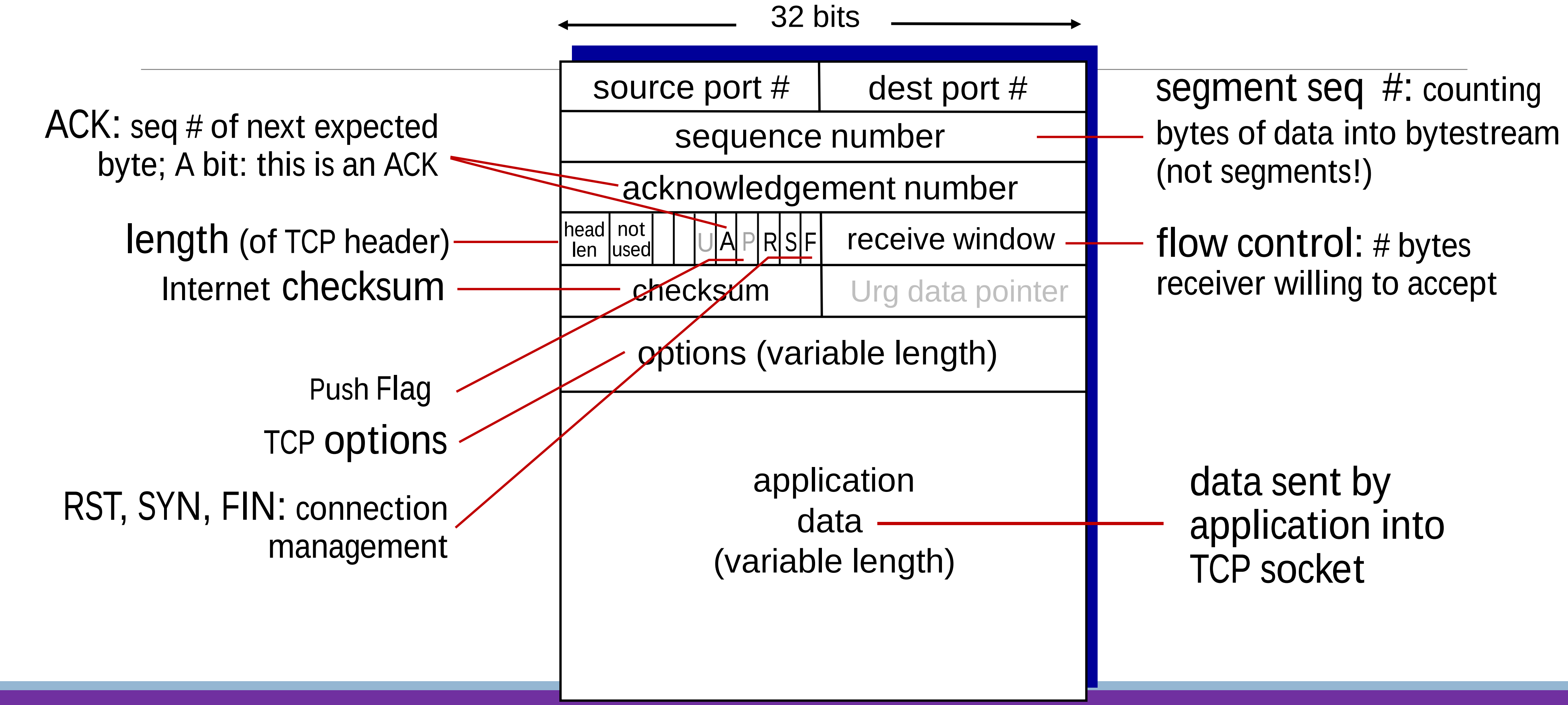
- segment structure
- Reliable Data Transfer
- flow control
- connection management

TCP congestion control

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control set window size
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

Push Flag

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

TCP sequence numbers, ACKs

Sequence numbers:

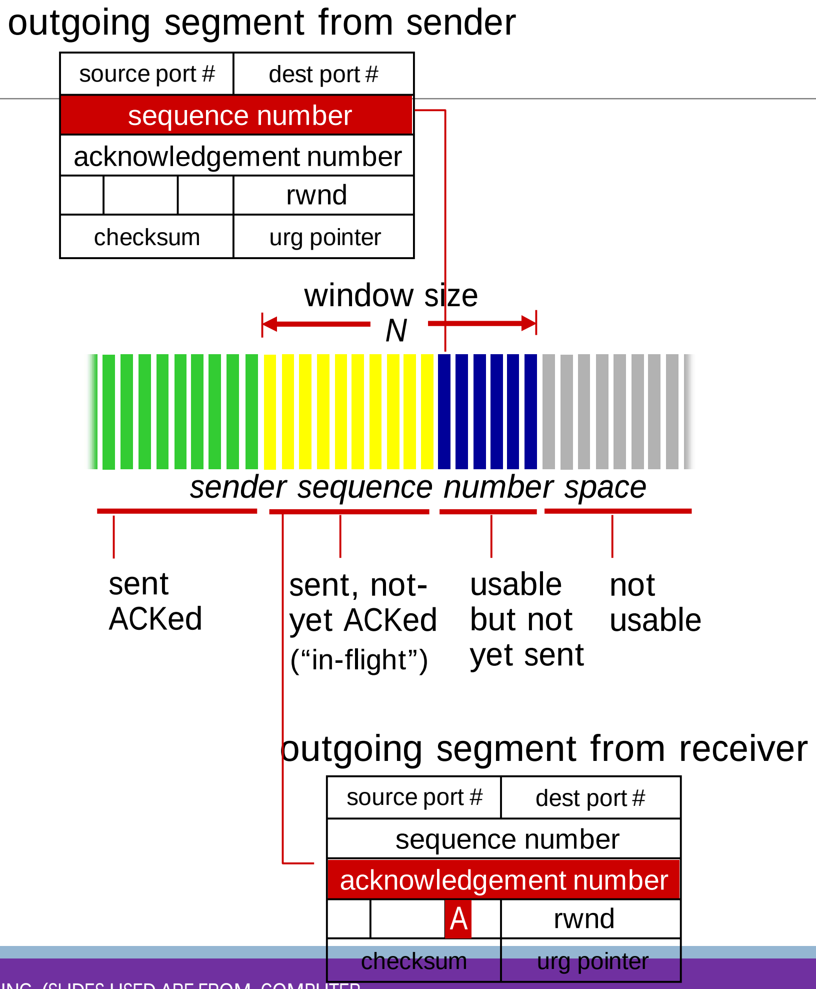
- byte stream "number" of first byte in segment's data

Acknowledgements:

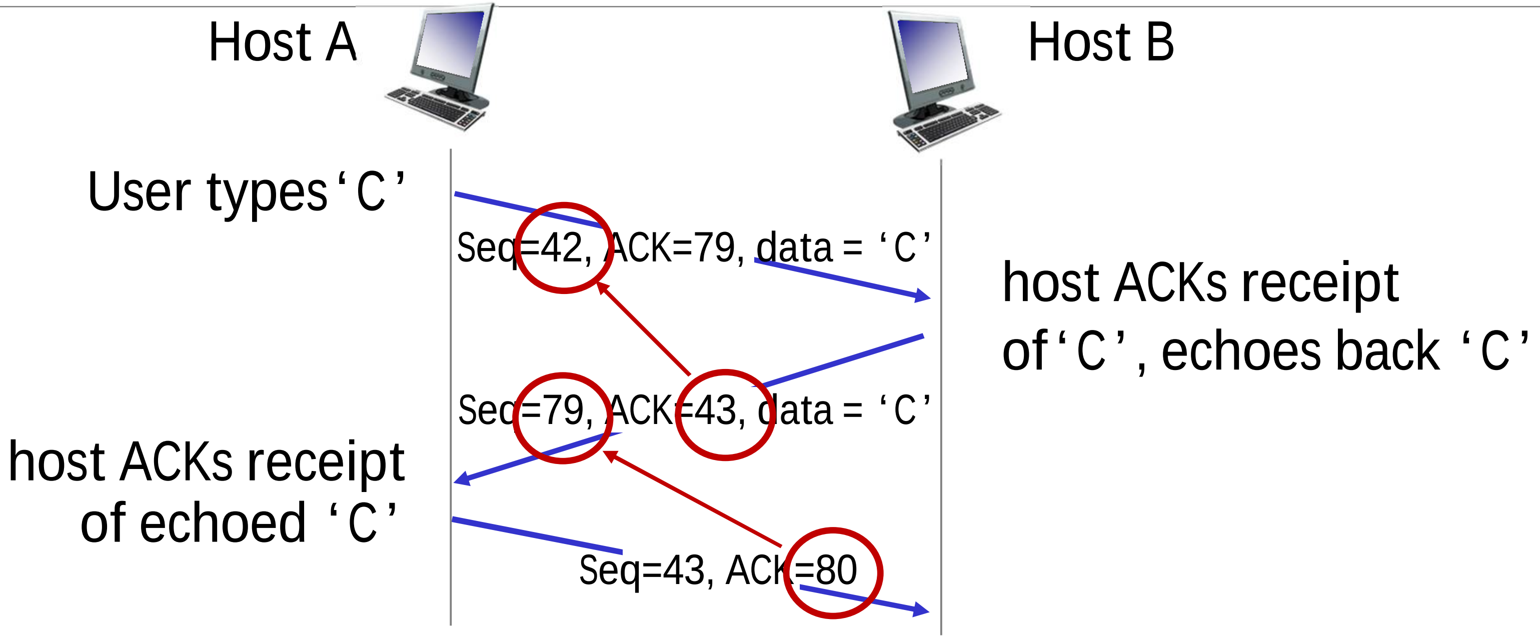
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP sequence numbers, ACKs


Host A
Host B
User types ‘C’
Seq=42, ACK=79, data = ‘C’
host ACKs receipt of ‘C’, echoes back ‘C’
Seq=79, ACK=43, data = ‘C’
host ACKs receipt of echoed ‘C’
Seq=43, ACK=80

simple telnet scenario

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT, but RTT varies!
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

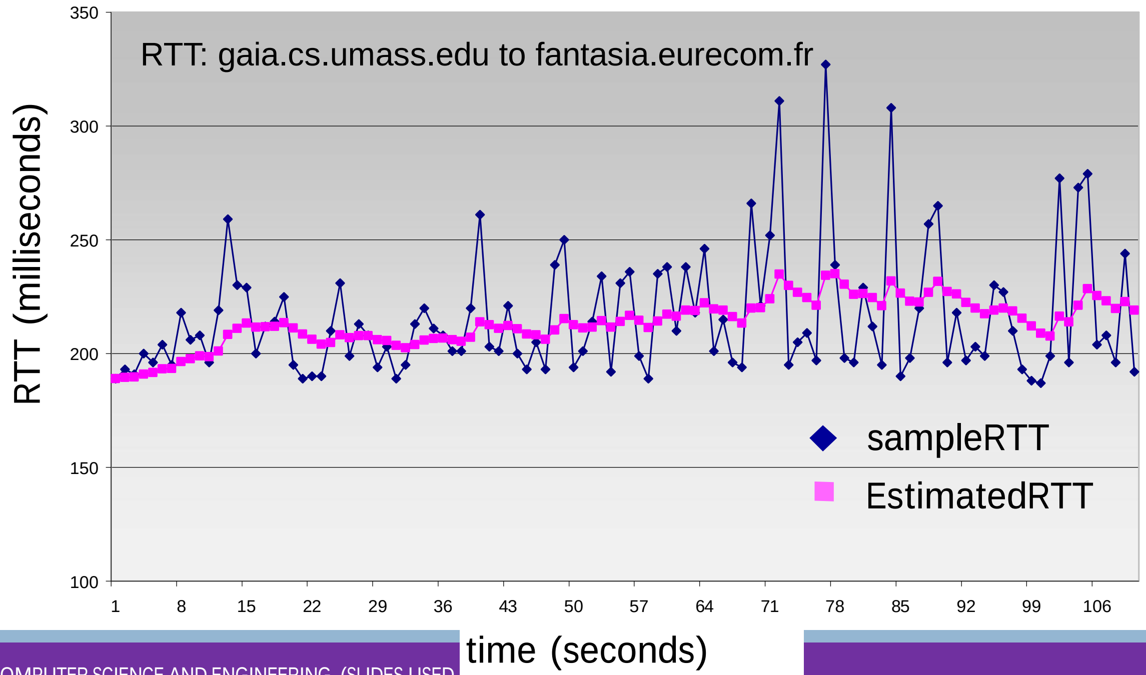
Q: how to estimate RTT?

- `SampleRTT:` measured time from segment transmission until ACK receipt
  - ignore retransmissions
- `SampleRTT` will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current `SampleRTT`

TCP round trip time, timeout

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT:** want a larger safety margin

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

- **DevRTT**: EWMA of **SampleRTT** deviation from **EstimatedRTT**:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
```

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Chapter 3: roadmap

Transport-layer services

Multiplexing and demultiplexing

Connectionless transport: UDP

Connection-oriented transport: TCP

- segment structure
- Reliable Data Transfer
- flow control
- connection management

TCP congestion control

TCP Sender (simplified)

event: data received from application

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

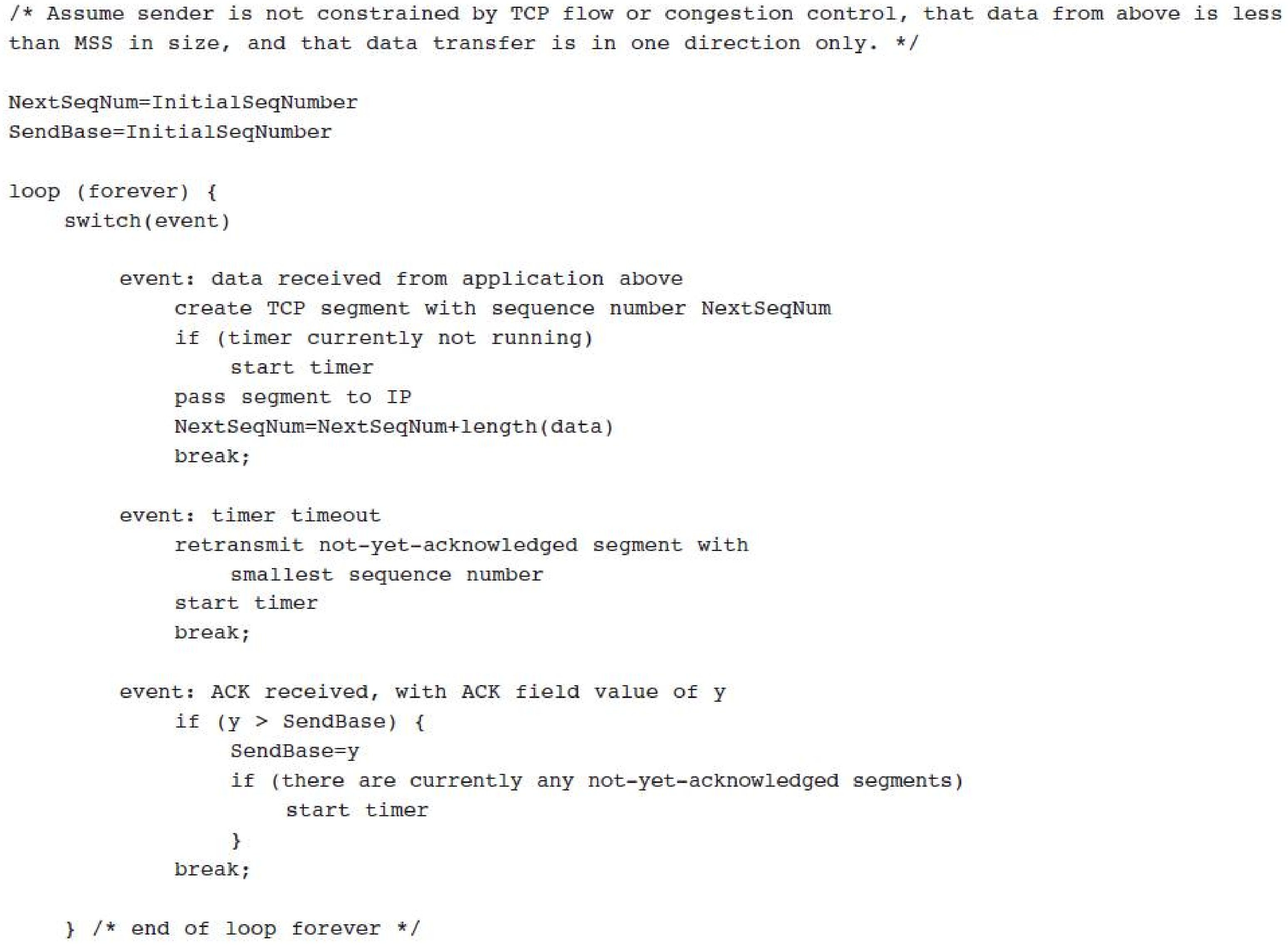
event: timeout

- retransmit segment that caused timeout
- restart timer

event: ACK received

- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - start timer if there are still unACKed segments

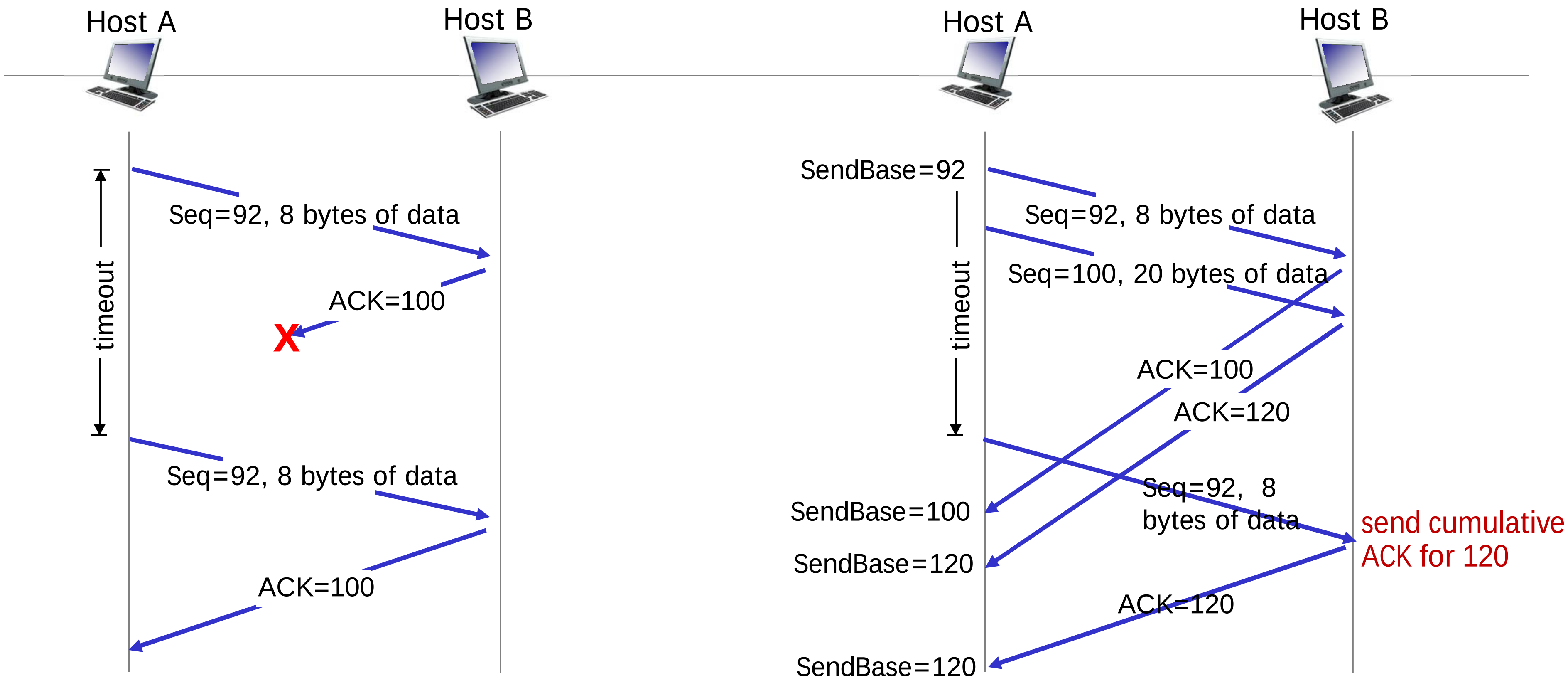
RAMAIAH
Institute of Technology



```
/* Assume sender is not constrained by TCP flow or congestion control, that data from above is less
than MSS in size, and that data transfer is in one direction only. */

NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

        event: data received from application above
            create TCP segment with sequence number NextSeqNum
            if (timer currently not running)
                start timer
            pass segment to IP
            NextSeqNum=NextSeqNum+length(data)
            break;

        event: timer timeout
            retransmit not-yet-acknowledged segment with
                smallest sequence number
            start timer
            break;

        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not-yet-acknowledged segments)
                    start timer
                }
            break;

    } /* end of loop forever */
```

Fig: TCP Simplified Sender

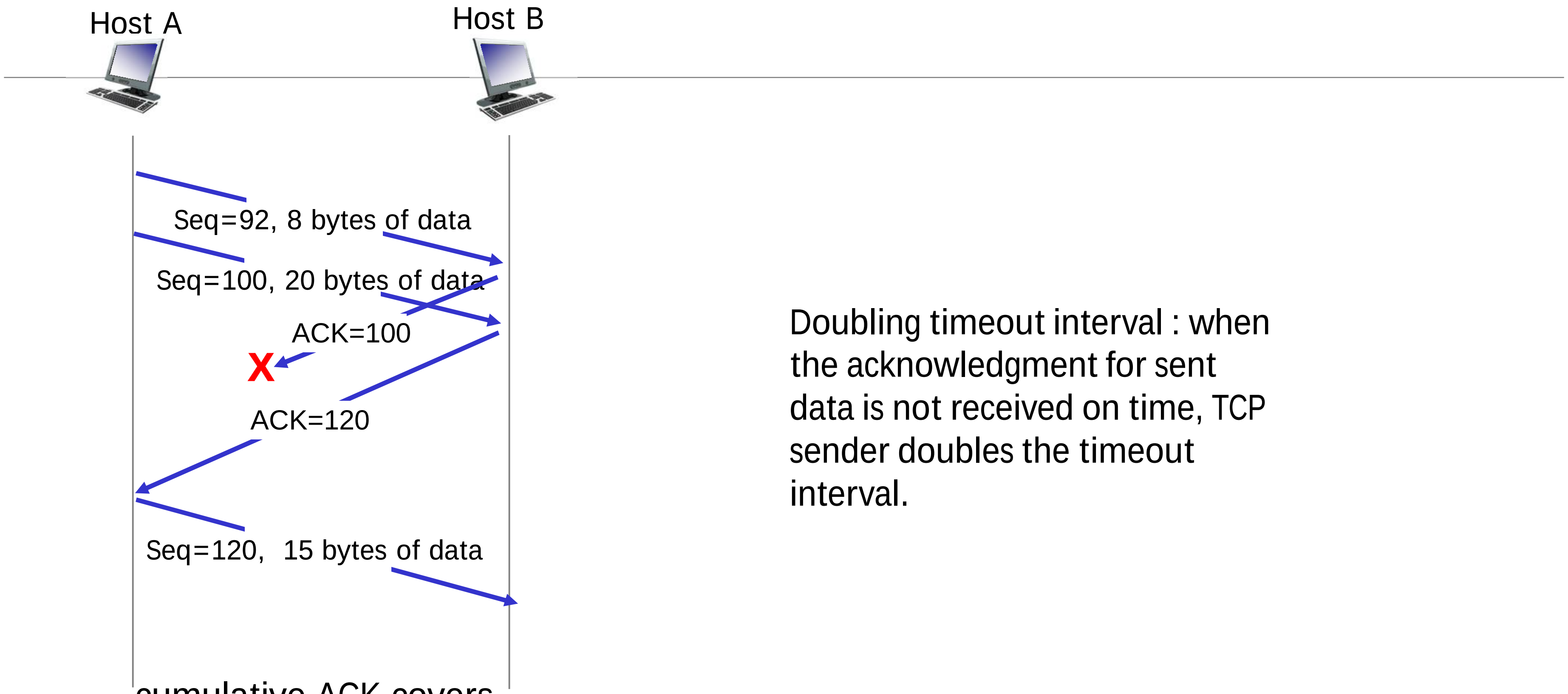
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios



cumulative ACK covers for earlier lost ACK

Doubling timeout interval : when the acknowledgment for sent data is not received on time, TCP sender doubles the timeout interval.

TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

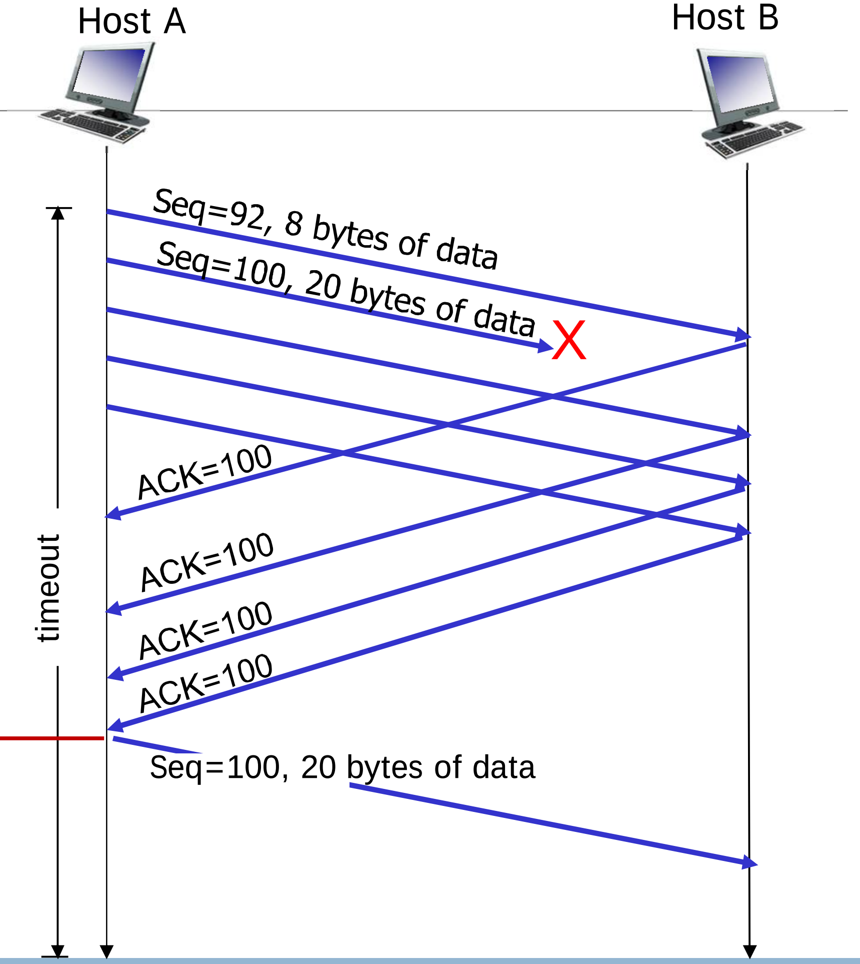
TCP fast retransmit

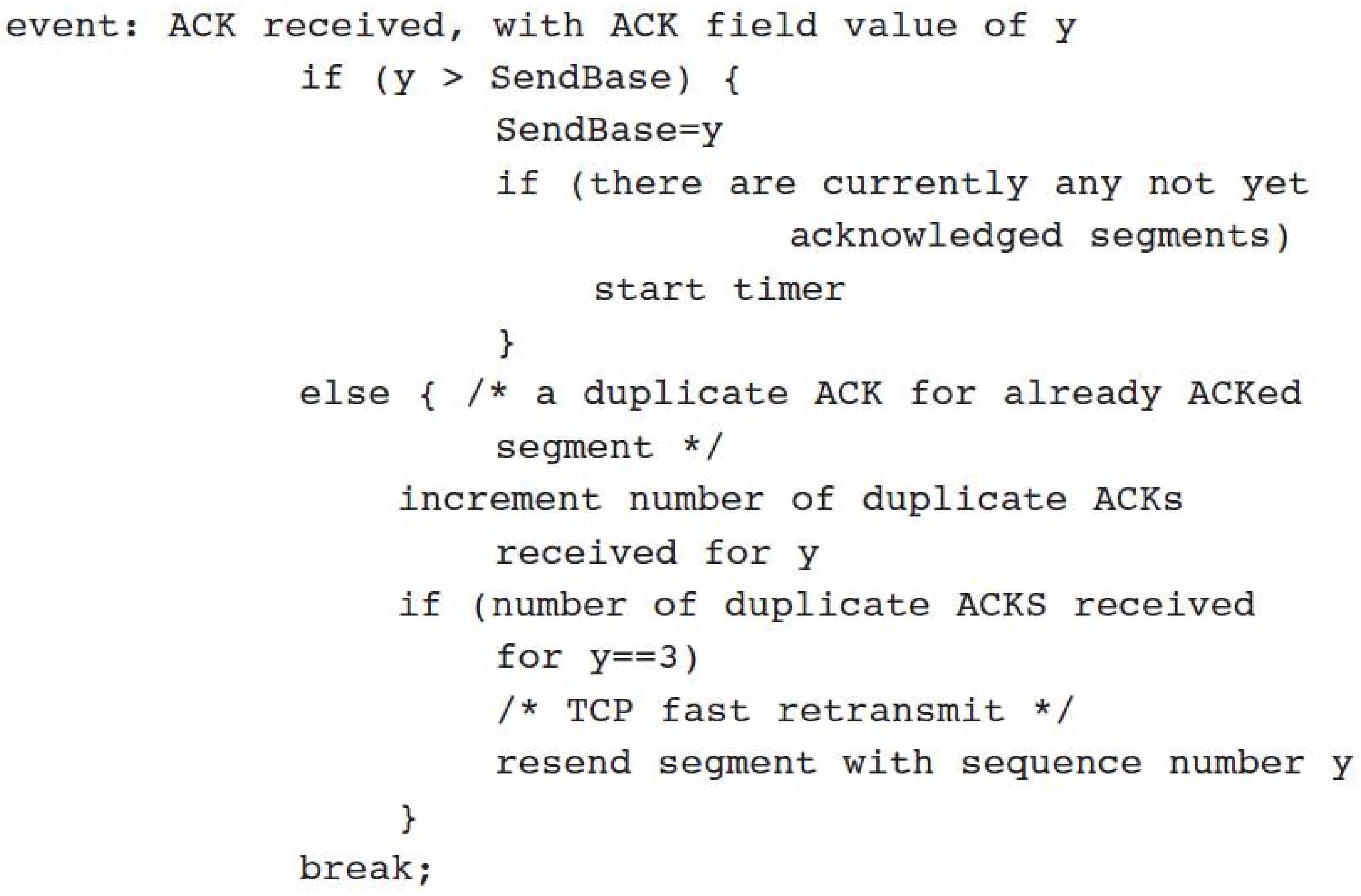
TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!





```
event: ACK received, with ACK field value of y
             if (y > SendBase) {
                     SendBase=y
                     if (there are currently any not yet
                                  acknowledged segments)
                         start timer
                     }
             else { /* a duplicate ACK for already ACKed
                     segment */
                 increment number of duplicate ACKs
                     received for y
                 if (number of duplicate ACKS received
                     for y==3)
                     /* TCP fast retransmit */
                     resend segment with sequence number y
                 }
             break;
```

Fig: TCP with Fast Retransmit

RAMAIAH
Institute of Technology

Chapter 3: roadmap

Transport-layer services

Multiplexing and demultiplexing

Connectionless transport: UDP

Principles of reliable data transfer

Connection-oriented transport: TCP

- segment structure
- Reliable Data Transfer
- flow control
- connection management

Principles of congestion control

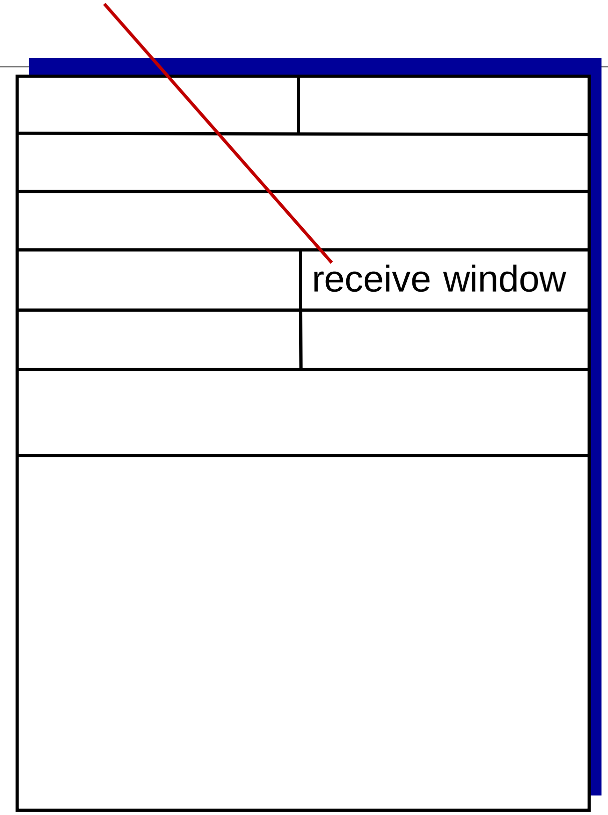
TCP congestion control

RAMAIAH
Institute of Technology

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

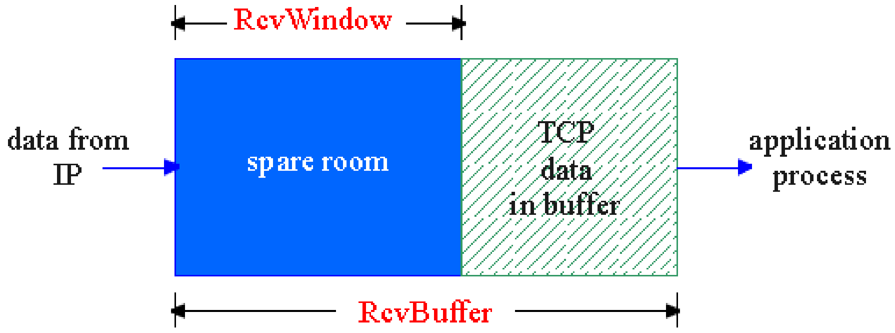
flow control: # bytes receiver willing to accept



TCP segment format

TCP Flow Control

receive side of TCP connection has a receive buffer:



- app process may be slow at reading from buffer

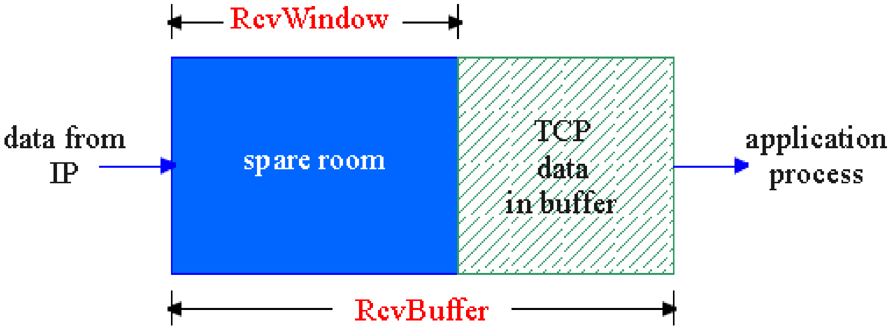
flow control

sender won’t overflow receiver’s buffer by transmitting too much, too fast

speed-matching service: matching the send rate to the receiving app’s drain rate

RAMAIAH
Institute of Technology

TCP Flow control: how it works



At Sender side:

`LastByteSent – LastByteAcked ≤ rwnd`

At Receiver side:

Because TCP is not permitted to overflow the allocated buffer, we must have

`LastByteRcvd – LastByteRead ≤ RcvBuffer`

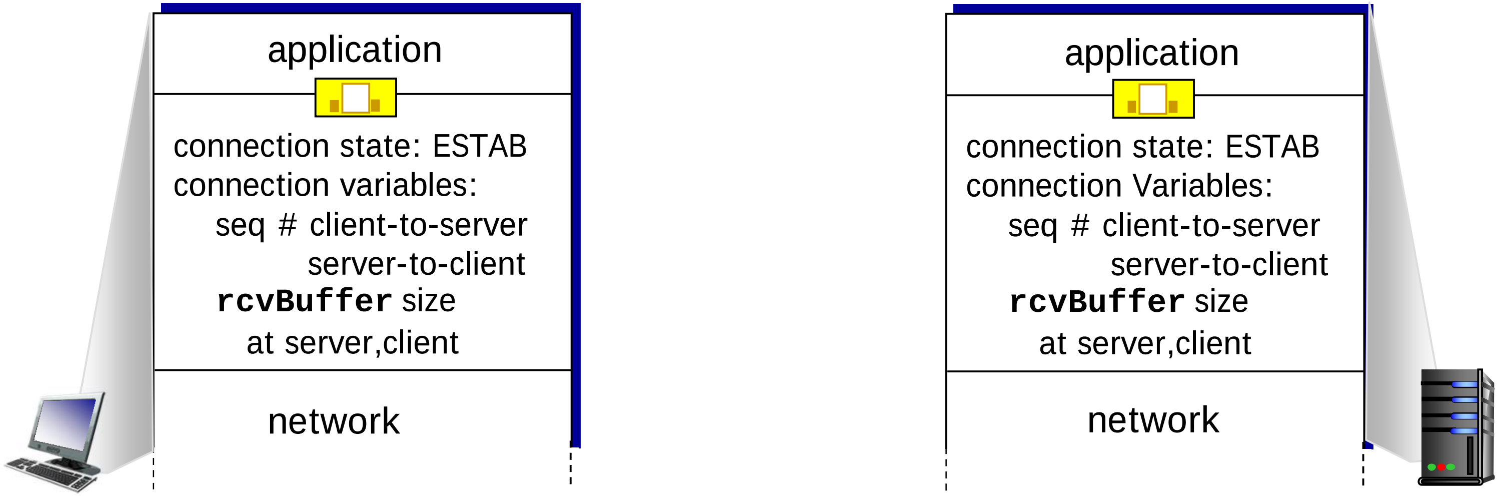
The receive window, denoted `rwnd` is set to the amount of spare room in the buffer:

`rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]`

TCP connection management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)



```
Socket clientSocket =
  newSocket("hostname","port number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```

TCP 3-way handshake

Server state

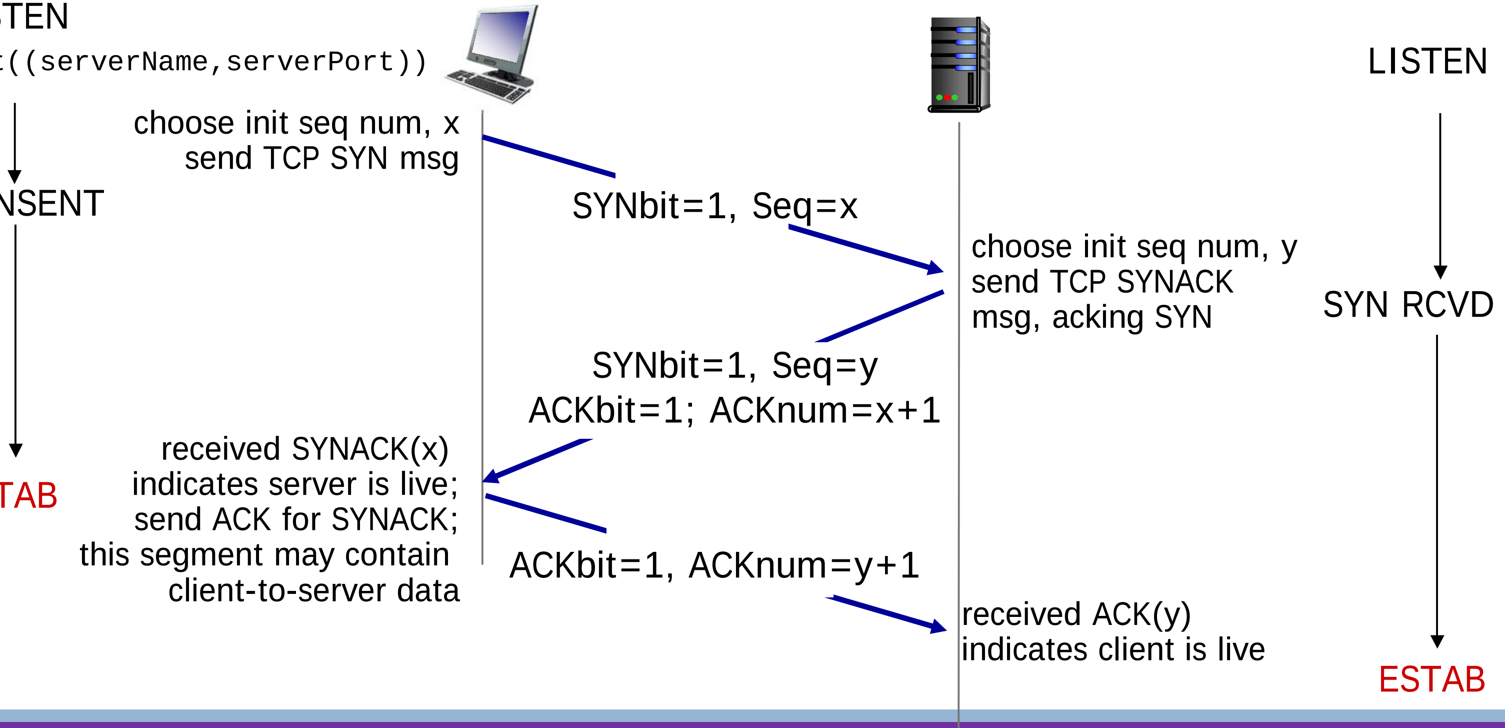
```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```



LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ESTAB

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

ESTAB

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

initialize TCP variables:

- seq. #s
- buffers, flow control info (e.g. `RcvWindow`)

client: connection initiator

```
 Socket clientSocket = new
Socket("hostname","port
number");
```

server: contacted by client

```
 Socket connectionSocket =
welcomeSocket.accept();
```

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (contd.)

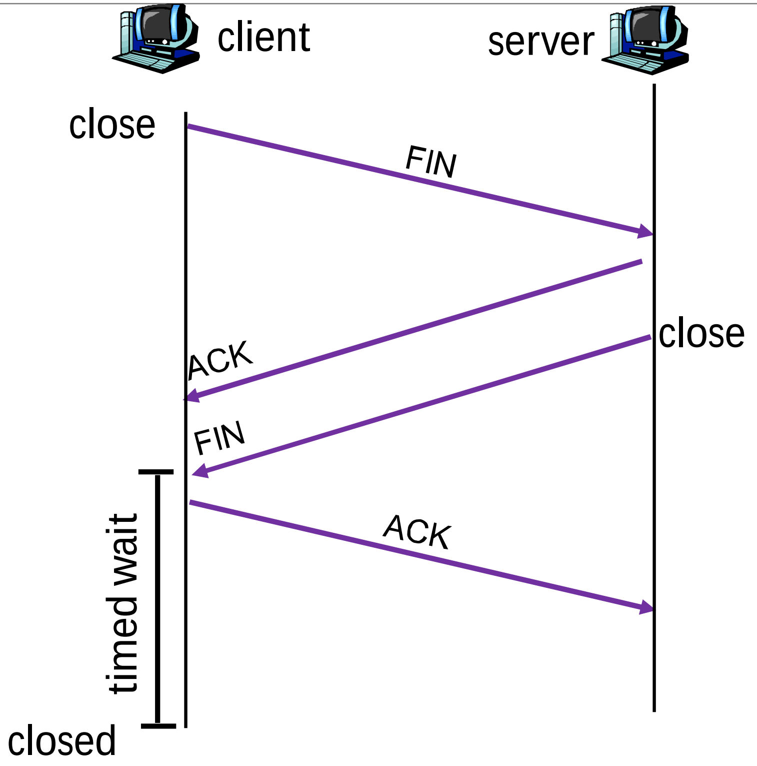
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



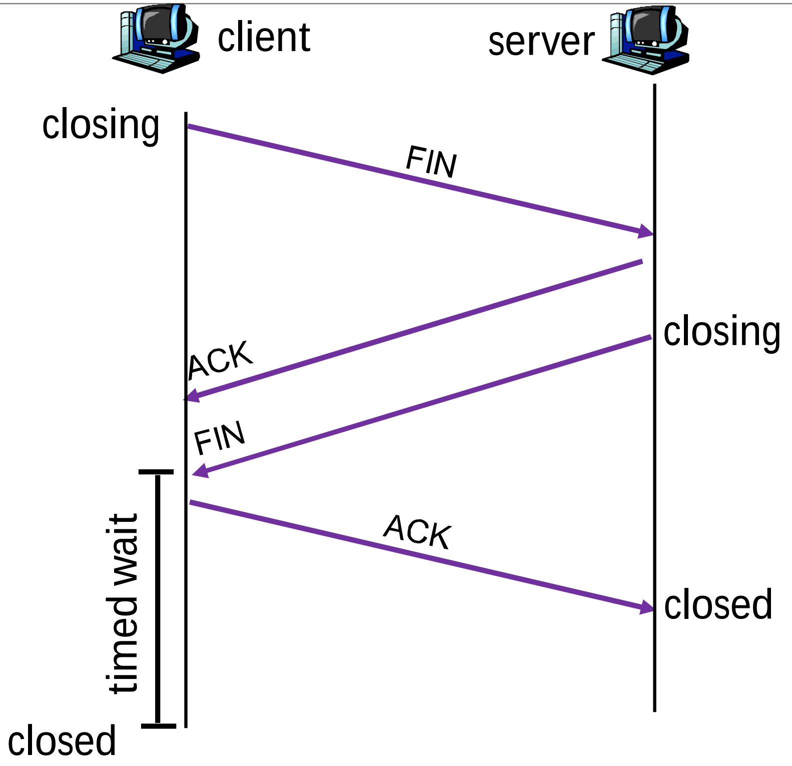
TCP Connection Management (contd.)

Step 3: client receives FIN, replies with ACK.

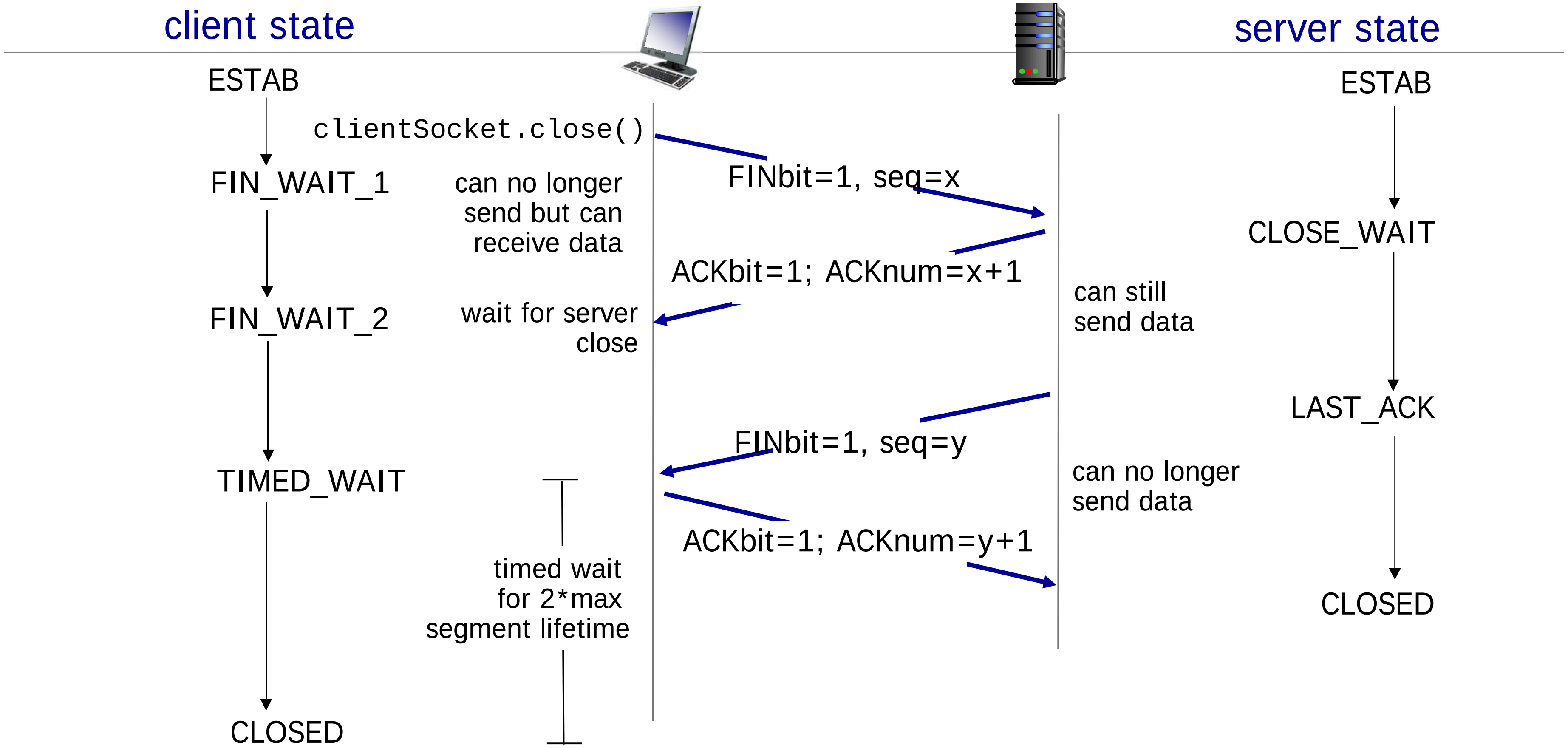
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



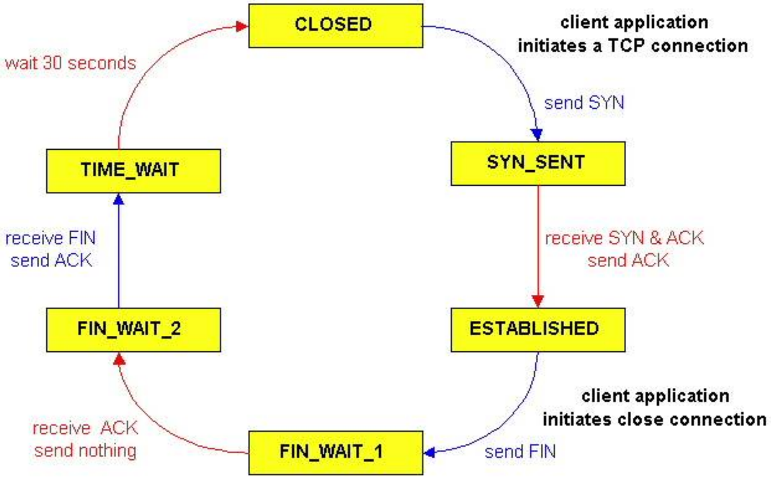
Closing a TCP connection



Closing a TCP connection

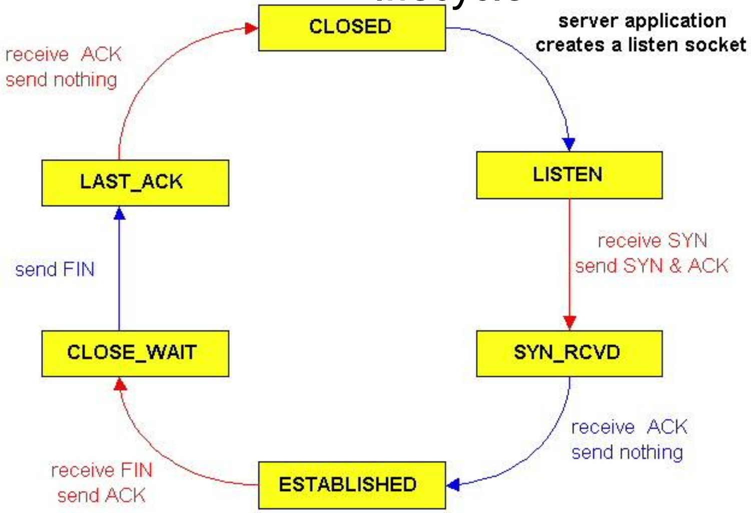
- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP Connection Management (contd.)



TCP client lifecycle

TCP server lifecycle



Chapter 3: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- Principles of congestion control
- TCP congestion control

Principles of congestion control

Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
- manifestations:
  - long delays (queueing in router buffers)
  - packet loss (buffer overflow at routers)
- different from flow control!

congestion control: too many senders, sending too fast

flow control: one sender too fast for one receiver

TCP congestion control: AIMD

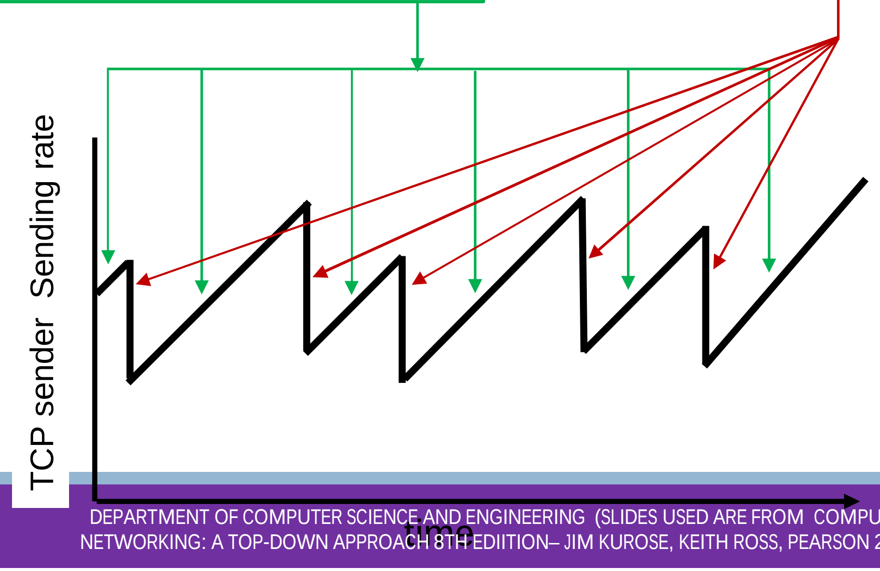
- *approach:* senders can increase sending rate until packet loss (congestion) occurs, then decrease sending rate on loss event

Additive Increase

increase sending rate by 1 maximum segment size every RTT until loss detected

Multiplicative Decrease

cut sending rate in half at each loss event



AIMD sawtooth behavior: *probing* for bandwidth

TCP AIMD: more

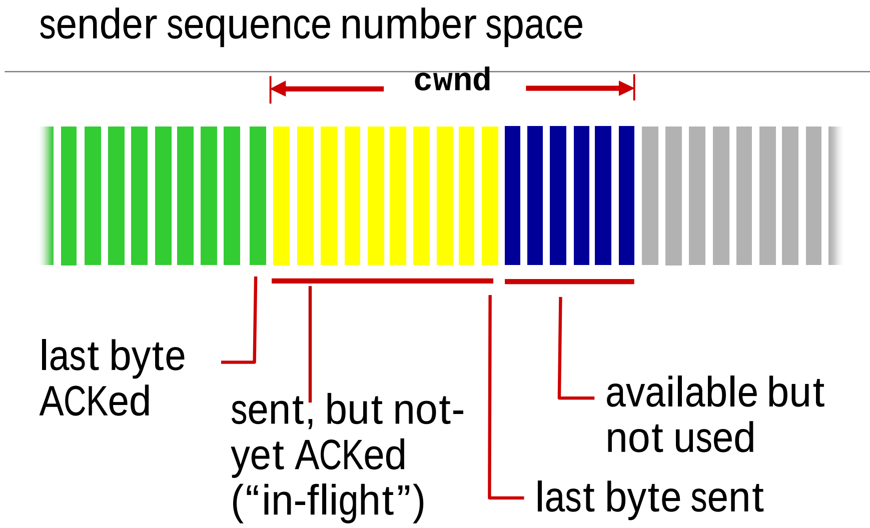
Multiplicative decrease detail: sending rate is

- Cut in half on loss detected by triple duplicate ACK (TCP Reno)
- Cut to 1 MSS (maximum segment size) when loss detected by timeout (TCP Tahoe)

Why AIMD?

- AIMD – a distributed, asynchronous algorithm – has been shown to:
  - optimize congested flow rates network wide!
  - have desirable stability properties

TCP congestion control: details


sender sequence number space
cwnd
last byte ACKed
sent, but not-yet ACKed ("in-flight")
available but not used
last byte sent

TCP sending behavior:

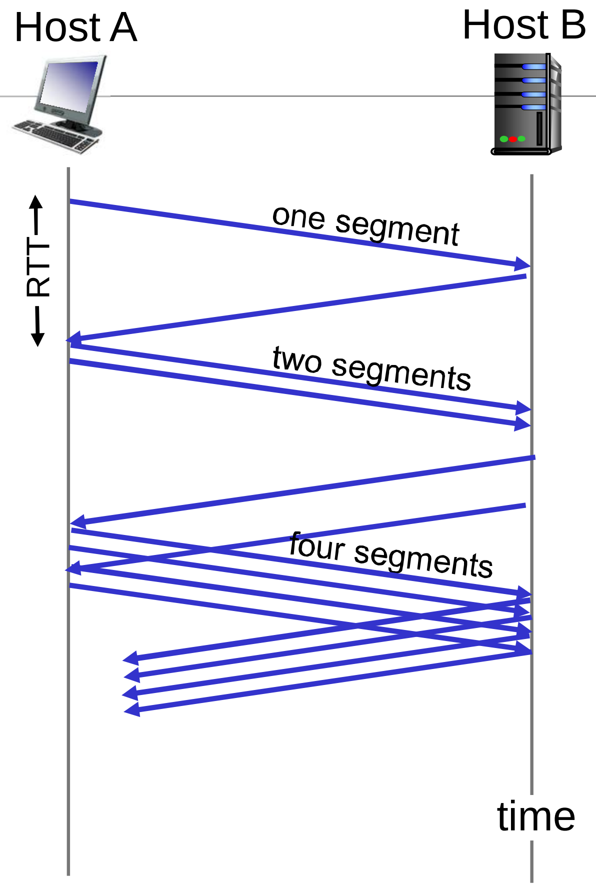
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- cwnd is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

TCP slow start

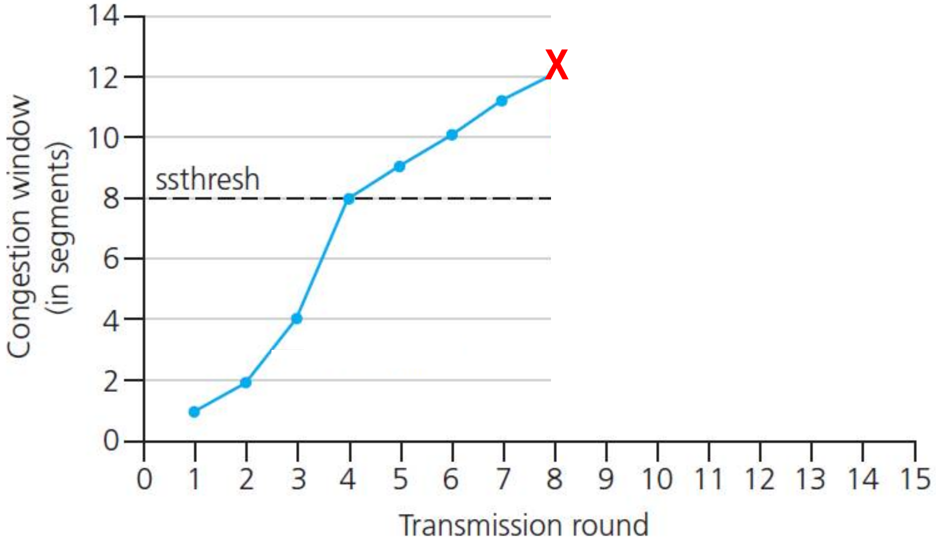
- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

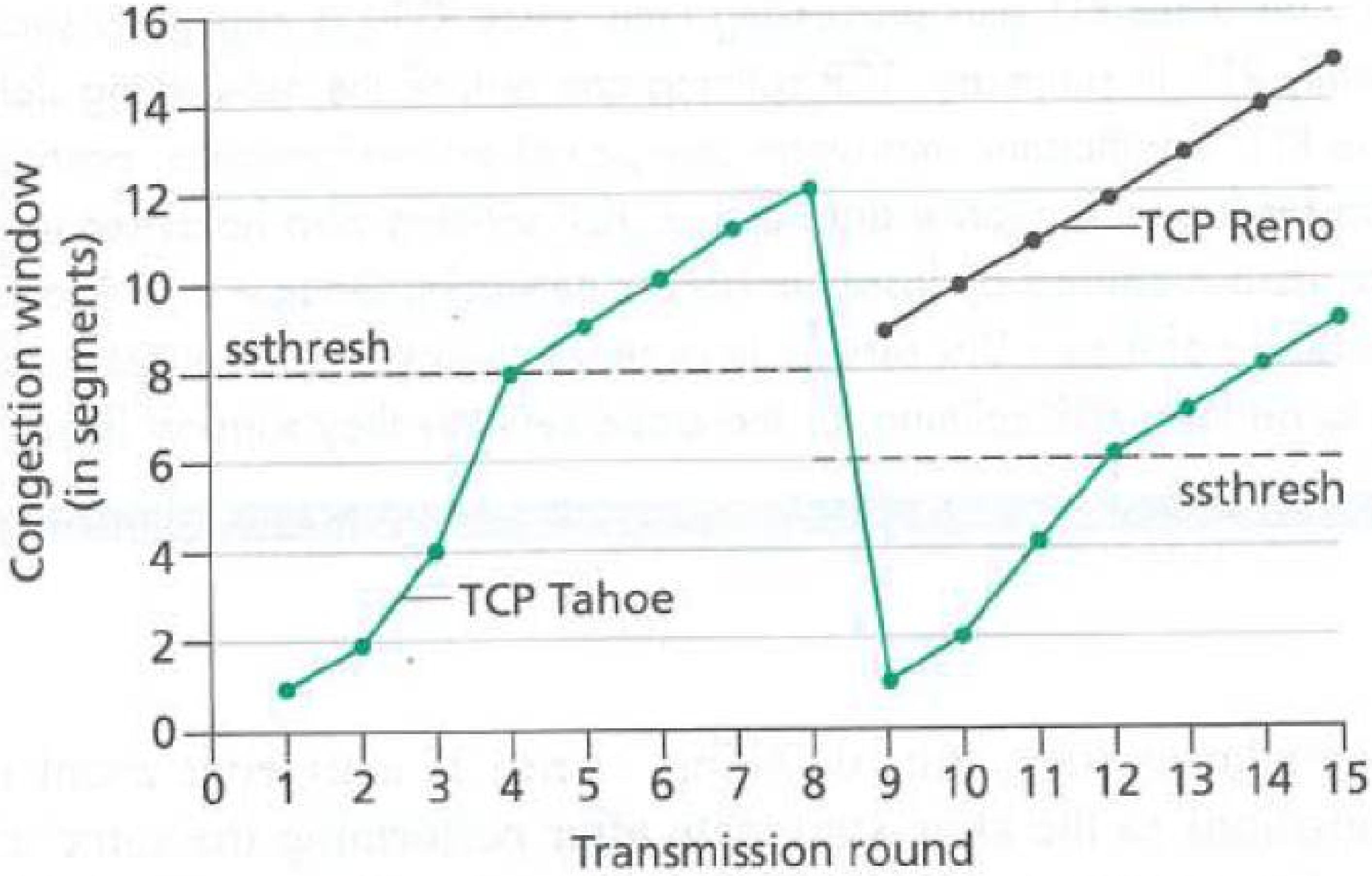
Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.



Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event


Congestion window (in segments)
16
14
12
10
8
6
4
2
0
ssthresh
TCP Reno
TCP Tahoe
ssthresh
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Transmission round

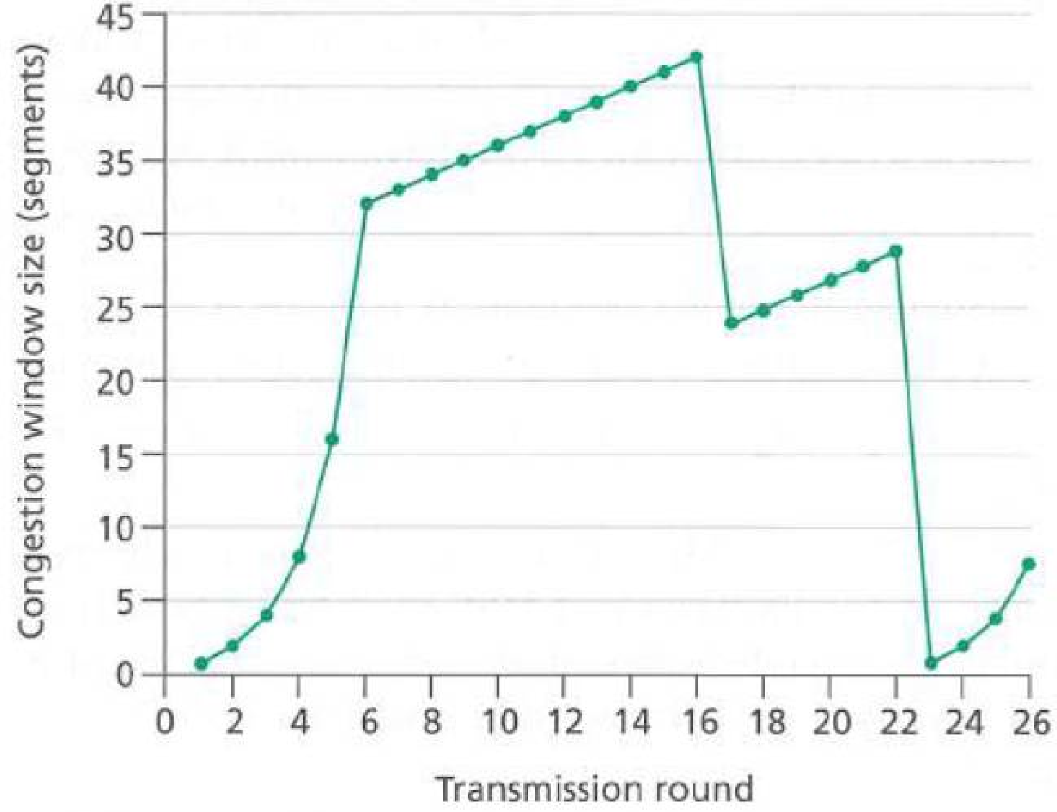
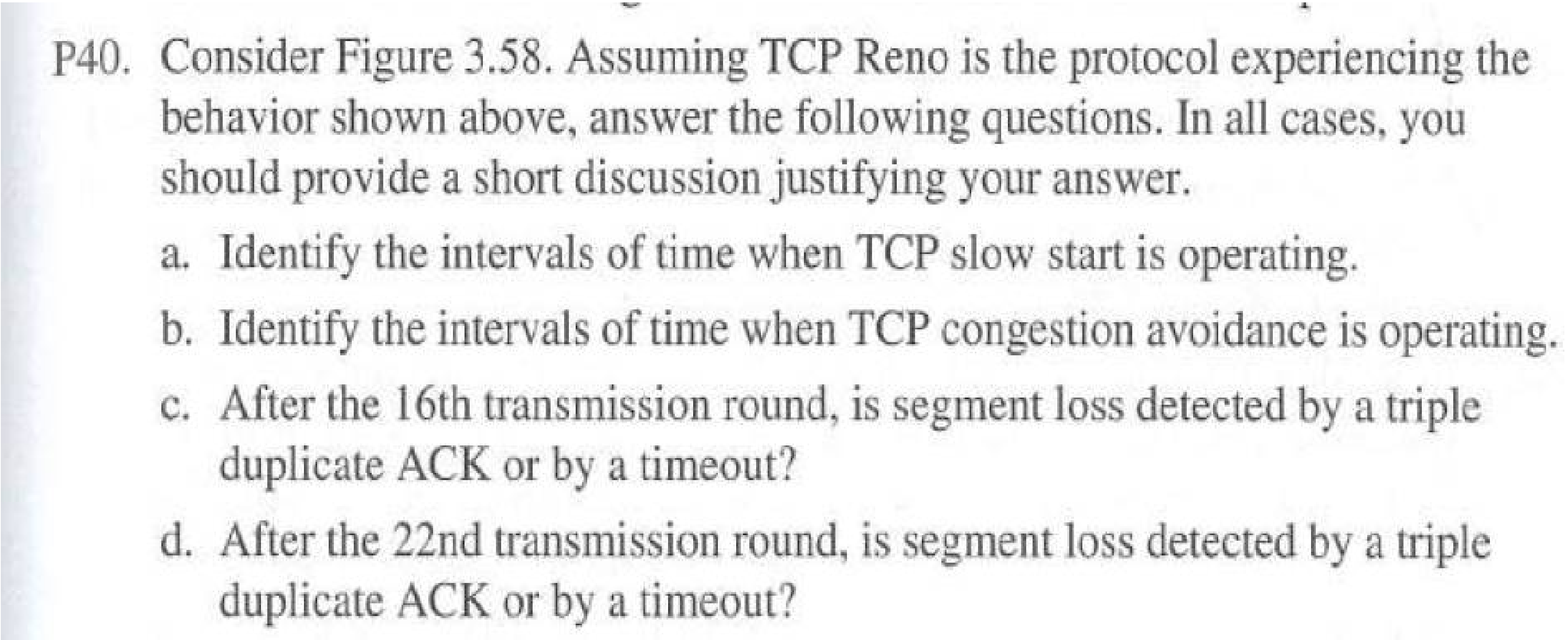


Figure 3.58 ♦ TCP window size as a function of time



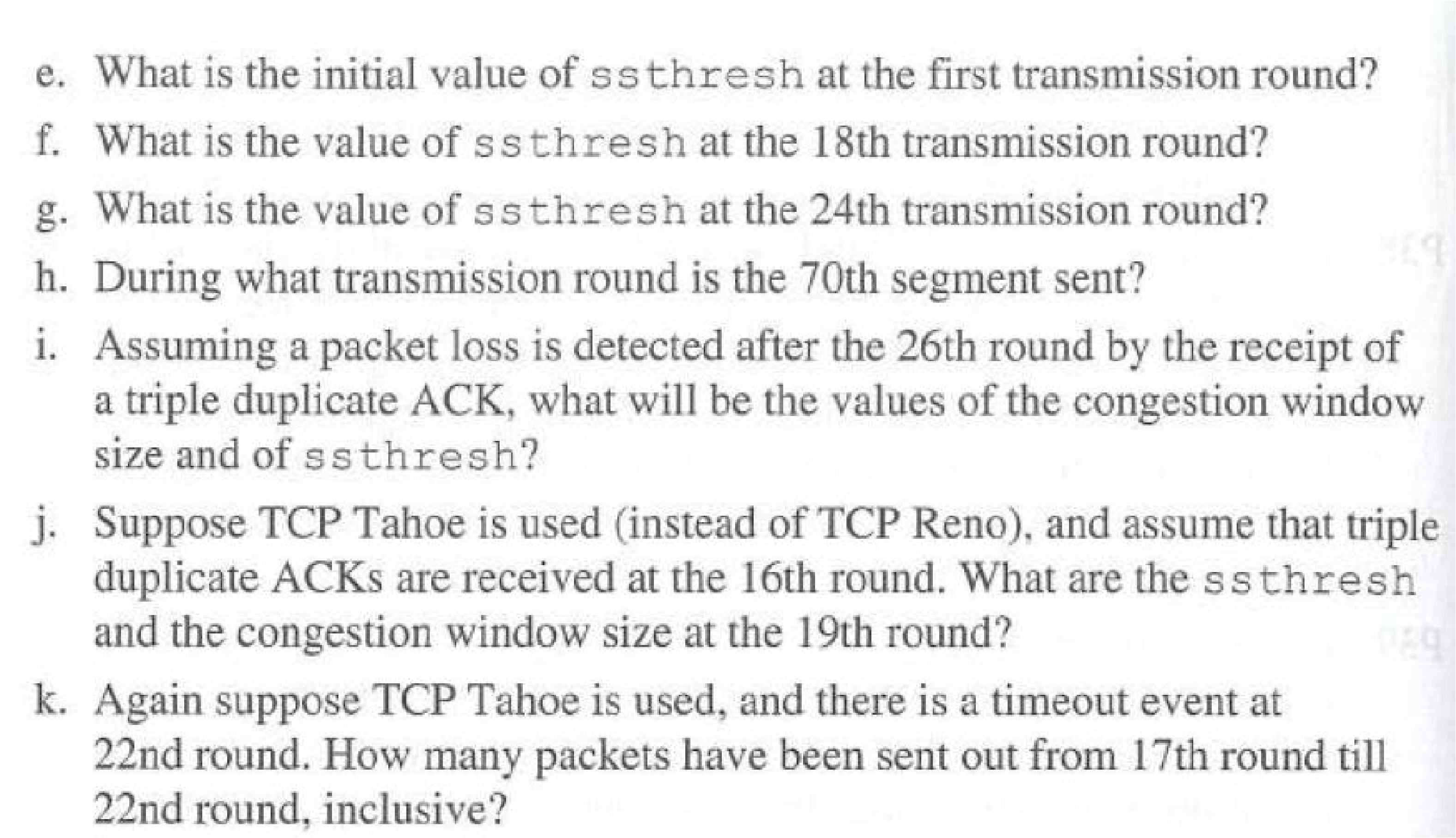
P40. Consider Figure 3.58. Assuming TCP Reno is the protocol experiencing the behavior shown above, answer the following questions. In all cases, you should provide a short discussion justifying your answer.

a. Identify the intervals of time when TCP slow start is operating.

b. Identify the intervals of time when TCP congestion avoidance is operating.

c. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?

d. After the 22nd transmission round, is segment loss detected by a triple duplicate ACK or by a timeout?



e. What is the initial value of ssthresh at the first transmission round?

f. What is the value of ssthresh at the 18th transmission round?

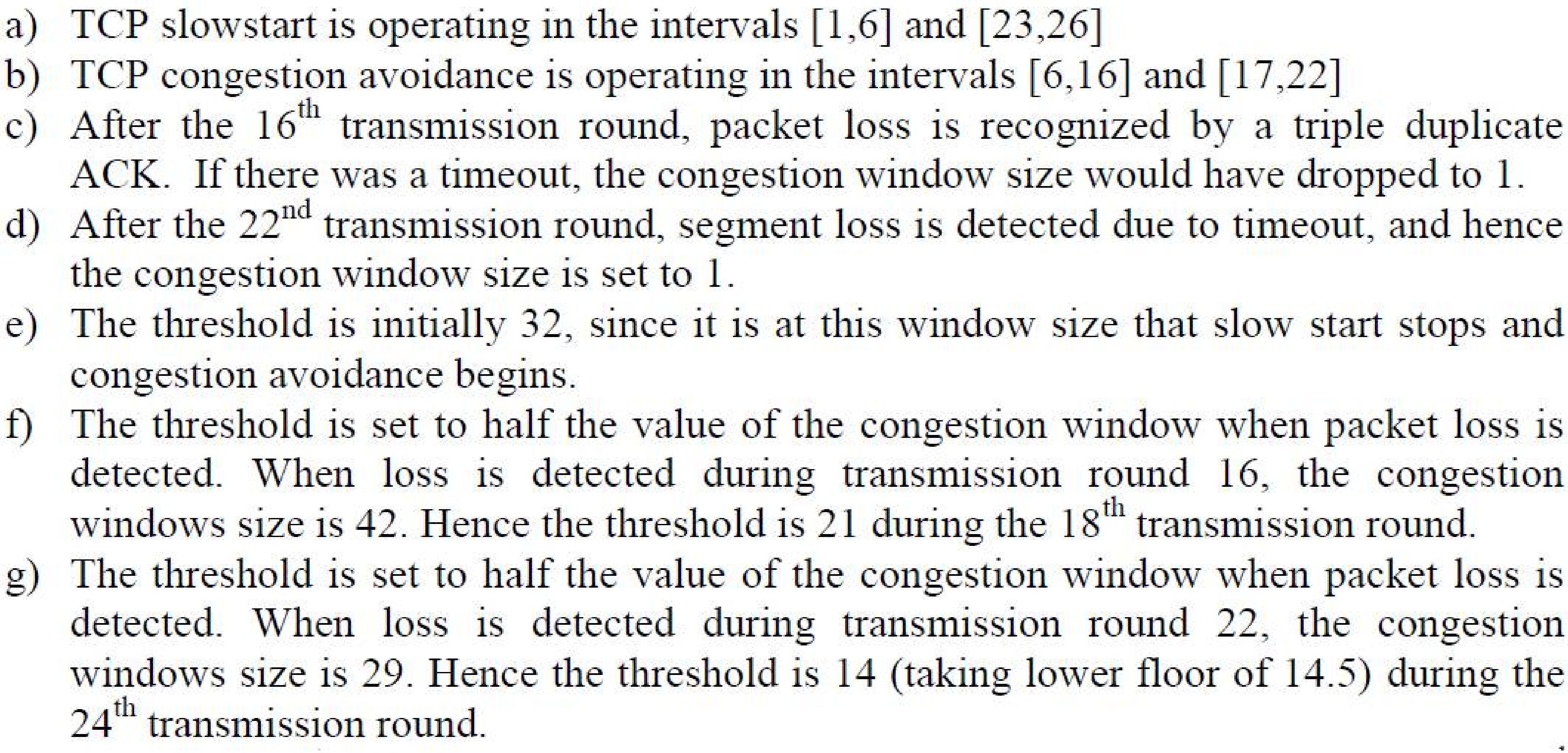
g. What is the value of ssthresh at the 24th transmission round?

h. During what transmission round is the 70th segment sent?

i. Assuming a packet loss is detected after the 26th round by the receipt of a triple duplicate ACK, what will be the values of the congestion window size and of ssthresh?

j. Suppose TCP Tahoe is used (instead of TCP Reno), and assume that triple duplicate ACKs are received at the 16th round. What are the ssthresh and the congestion window size at the 19th round?

k. Again suppose TCP Tahoe is used, and there is a timeout event at 22nd round. How many packets have been sent out from 17th round till 22nd round, inclusive?



a) TCP slowstart is operating in the intervals [1,6] and [23,26]

b) TCP congestion avoidance is operating in the intervals [6,16] and [17,22]

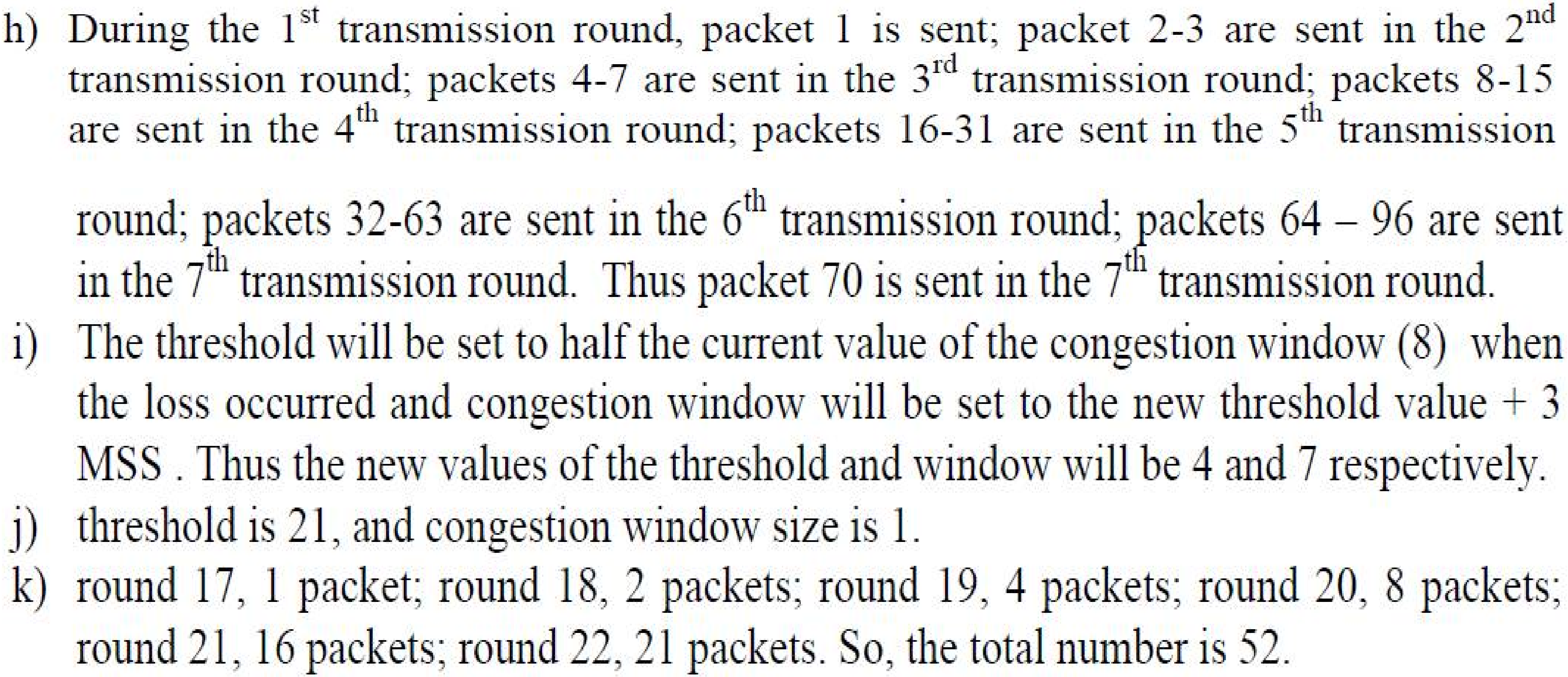
c) After the 16th transmission round, packet loss is recognized by a triple duplicate ACK. If there was a timeout, the congestion window size would have dropped to 1.

d) After the 22nd transmission round, segment loss is detected due to timeout, and hence the congestion window size is set to 1.

e) The threshold is initially 32, since it is at this window size that slow start stops and congestion avoidance begins.

f) The threshold is set to half the value of the congestion window when packet loss is detected. When loss is detected during transmission round 16, the congestion windows size is 42. Hence the threshold is 21 during the 18th transmission round.

g) The threshold is set to half the value of the congestion window when packet loss is detected. When loss is detected during transmission round 22, the congestion windows size is 29. Hence the threshold is 14 (taking lower floor of 14.5) during the 24th transmission round.



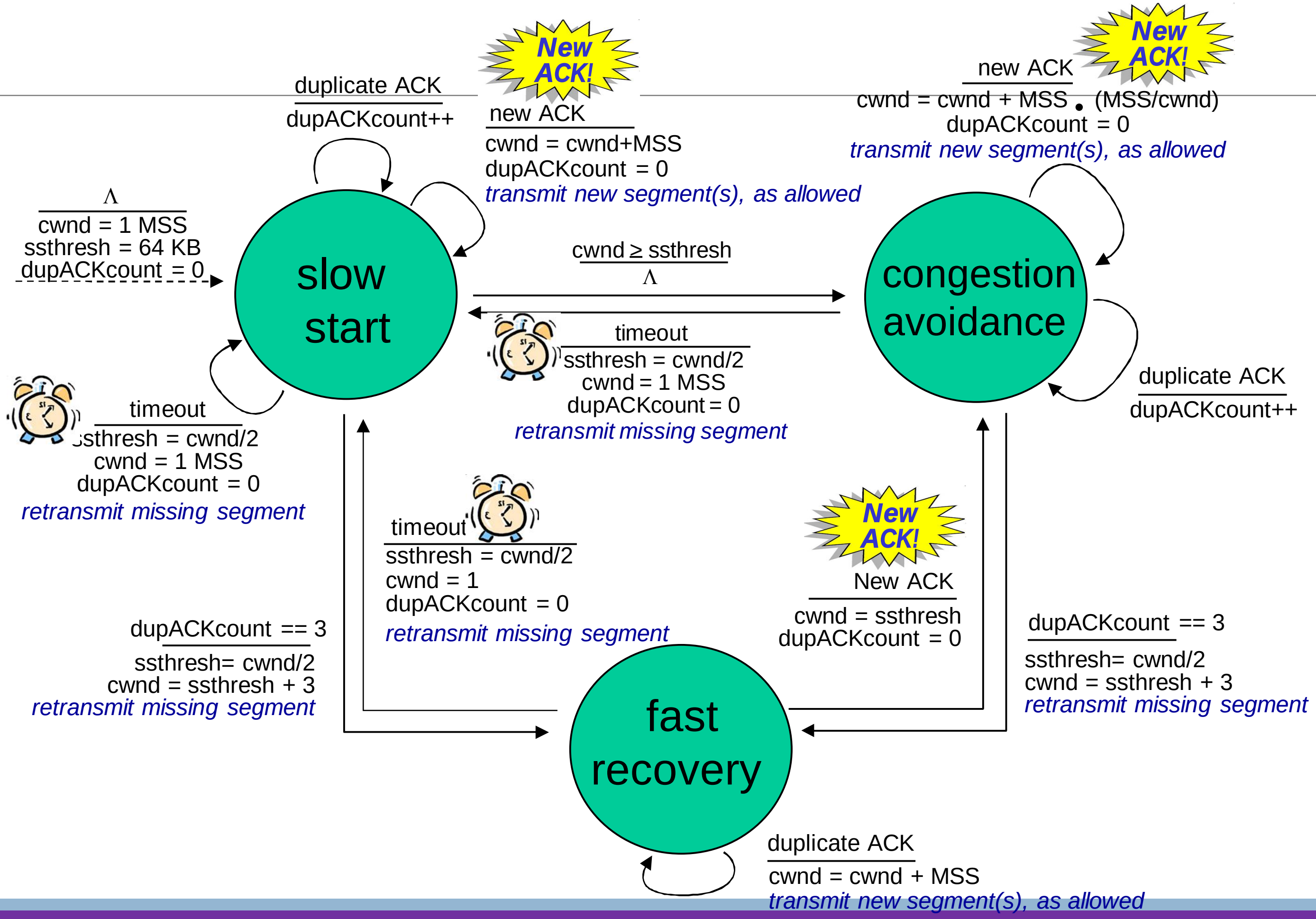
h) During the 1st transmission round, packet 1 is sent; packet 2-3 are sent in the 2nd transmission round; packets 4-7 are sent in the 3rd transmission round; packets 8-15 are sent in the 4th transmission round; packets 16-31 are sent in the 5th transmission round; packets 32-63 are sent in the 6th transmission round; packets 64 – 96 are sent in the 7th transmission round. Thus packet 70 is sent in the 7th transmission round.

i) The threshold will be set to half the current value of the congestion window (8) when the loss occurred and congestion window will be set to the new threshold value + 3 MSS . Thus the new values of the threshold and window will be 4 and 7 respectively.

j) threshold is 21, and congestion window size is 1.

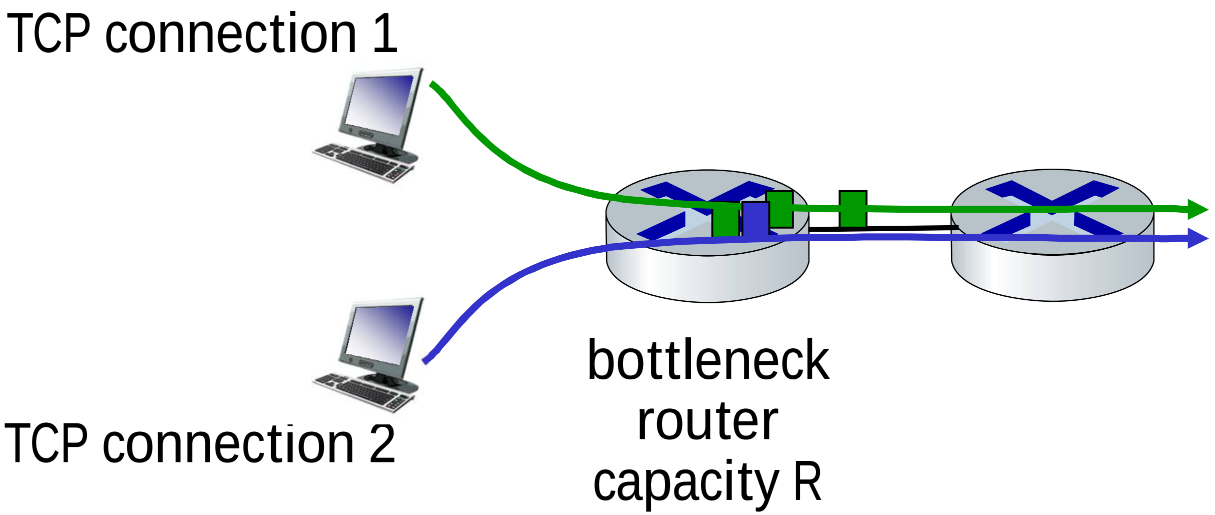
k) round 17, 1 packet; round 18, 2 packets; round 19, 4 packets; round 20, 8 packets; round 21, 16 packets; round 22, 21 packets. So, the total number is 52.

Summary: TCP congestion control



TCP fairness

Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*

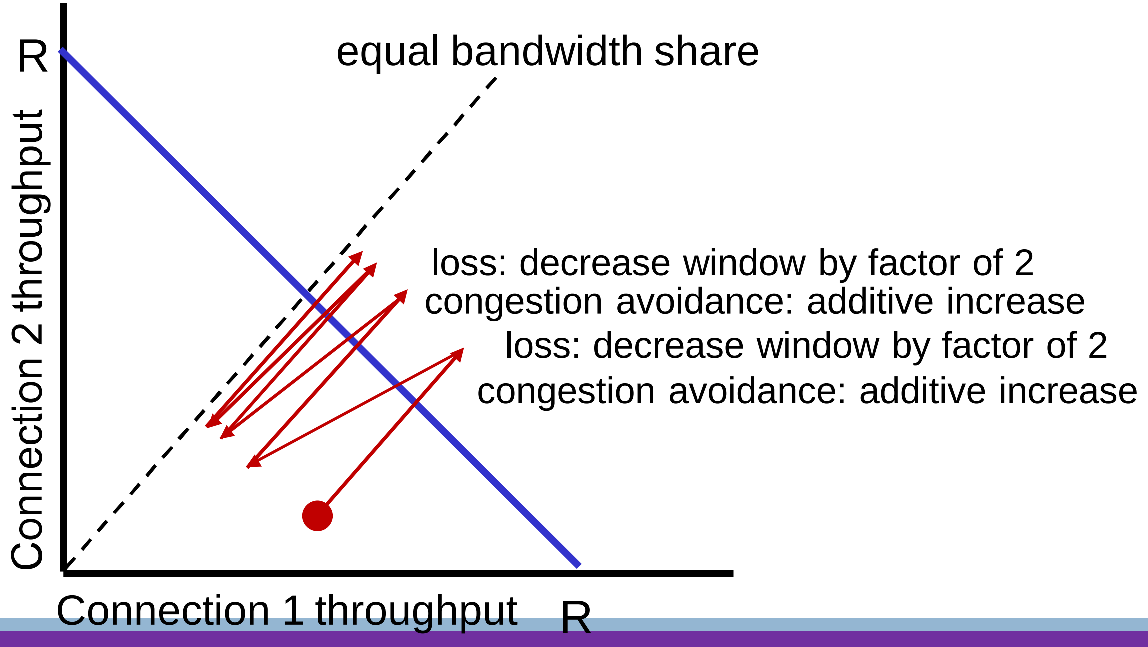


RAMAIAH
Institute of Technology

Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally


R
equal bandwidth share
loss: decrease window by factor of 2
congestion avoidance: additive increase
loss: decrease window by factor of 2
congestion avoidance: additive increase
Connection 2 throughput
Connection 1 throughput
R

Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be “fair”?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no “Internet police” policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

Thank you