



PES
UNIVERSITY
ONLINE

COMPUTER NETWORKS

Ashwini M Joshi

Department of Computer Science & Engineering

ashwinimjoshi@pes.edu

COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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

COMPUTER NETWORKS

In this segment

- Transport layer goals
- Transport layer services
- Transport services & protocols
- Transport vs Network layer
- Transport layer actions
- Internet transport layer protocols

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

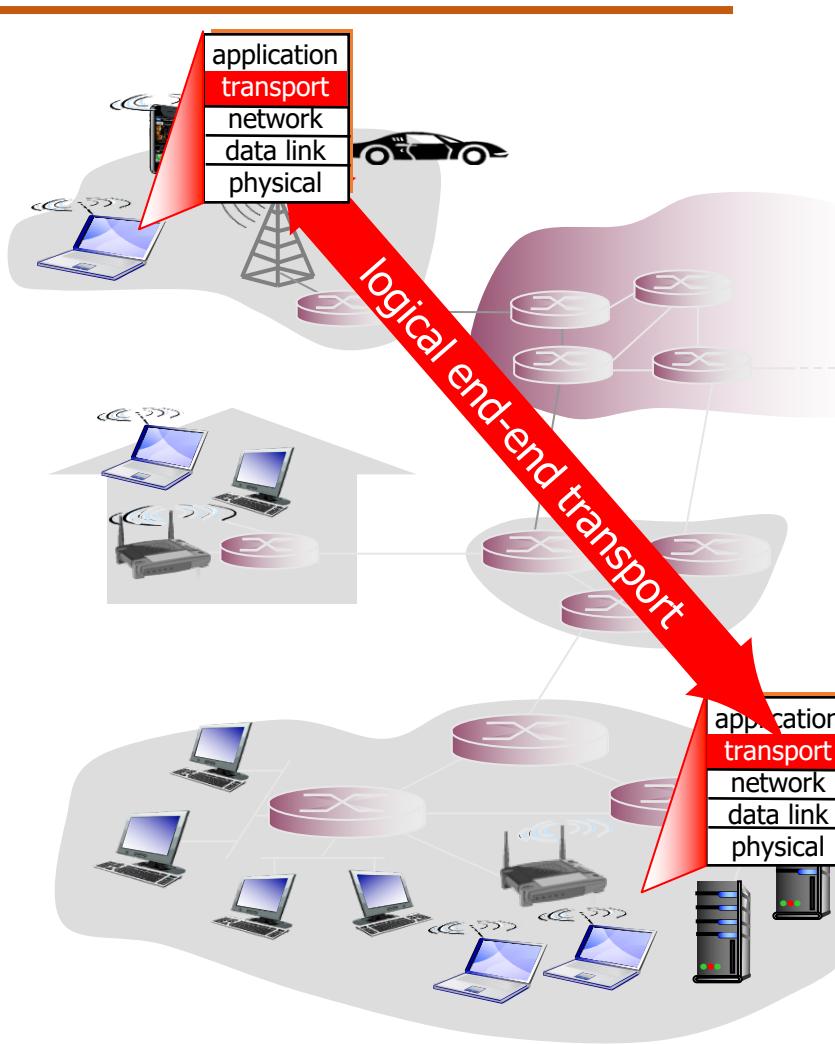
COMPUTER NETWORKS

Transport Layer Services

Ashwini M Joshi

Department of Computer Science & Engineering

- 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

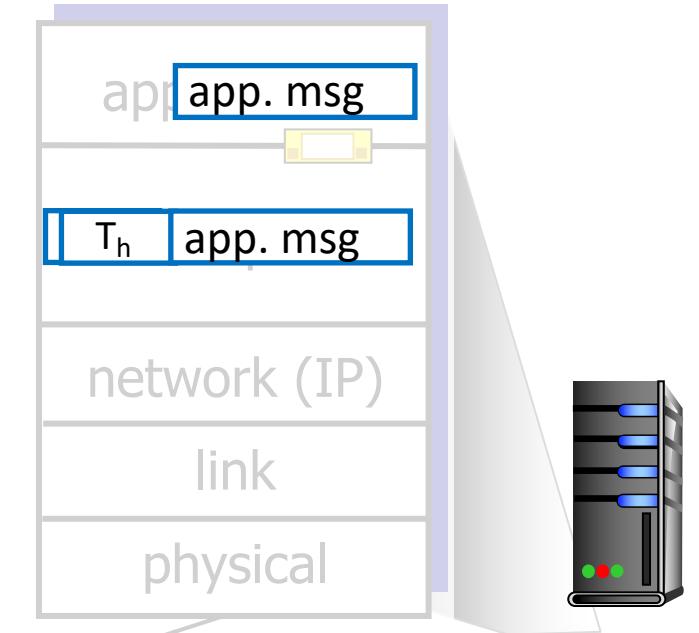
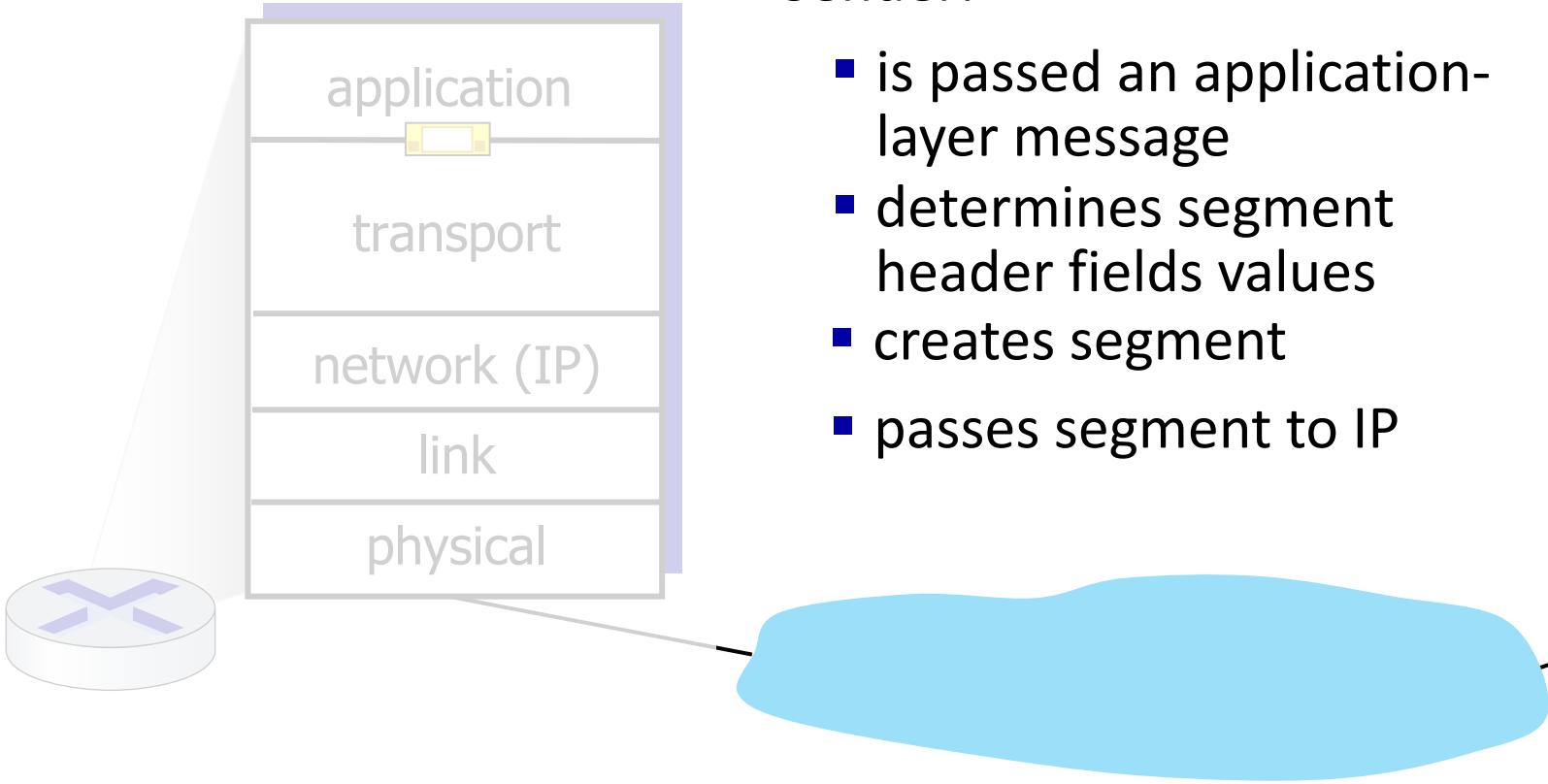
household analogy:

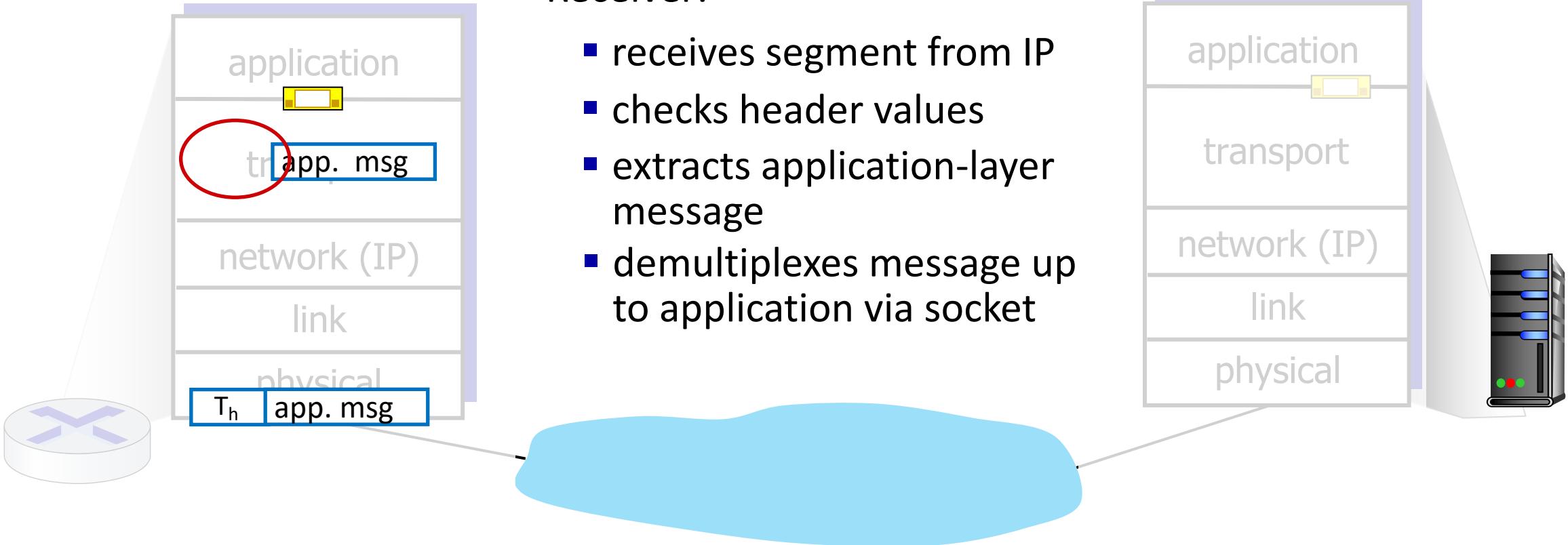
12 kids in Ann's house sending letters to 12 kids in Bill's house:

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

Sender:

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





Internet Transport-layer protocols

- **TCP:** Transmission Control Protocol

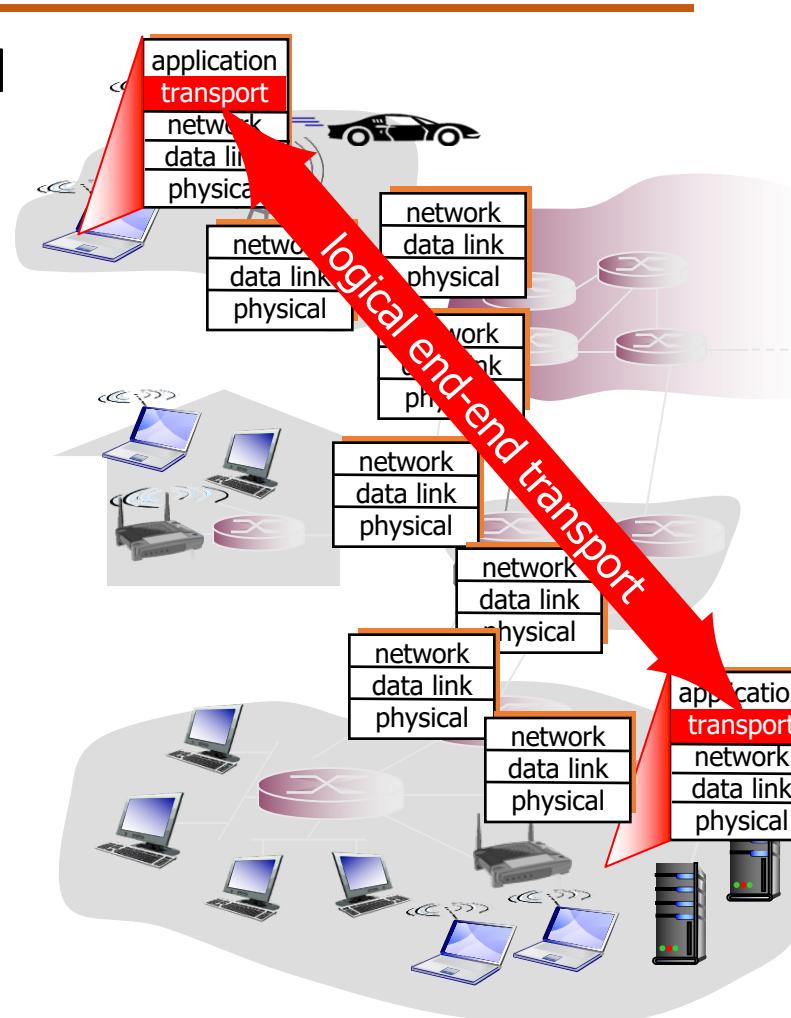
- reliable, connection oriented
- in-order delivery
- congestion control
- flow control
- connection setup

- **UDP:** User Datagram Protocol

- unreliable, connectionless
- unordered delivery
- no-frills extension of “best-effort” IP

- services not available:

- delay guarantees
- bandwidth guarantees



COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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

COMPUTER NETWORKS

Multiplexing & Demultiplexing

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

What is transport-layer multiplexing and demultiplexing

- How demultiplexing works
 - TCP / UDP segment format
- Connectionless demultiplexing - UDP
- Connectionless demux: Example
- Connection-oriented demux - TCP
- Connection-oriented demux: Example

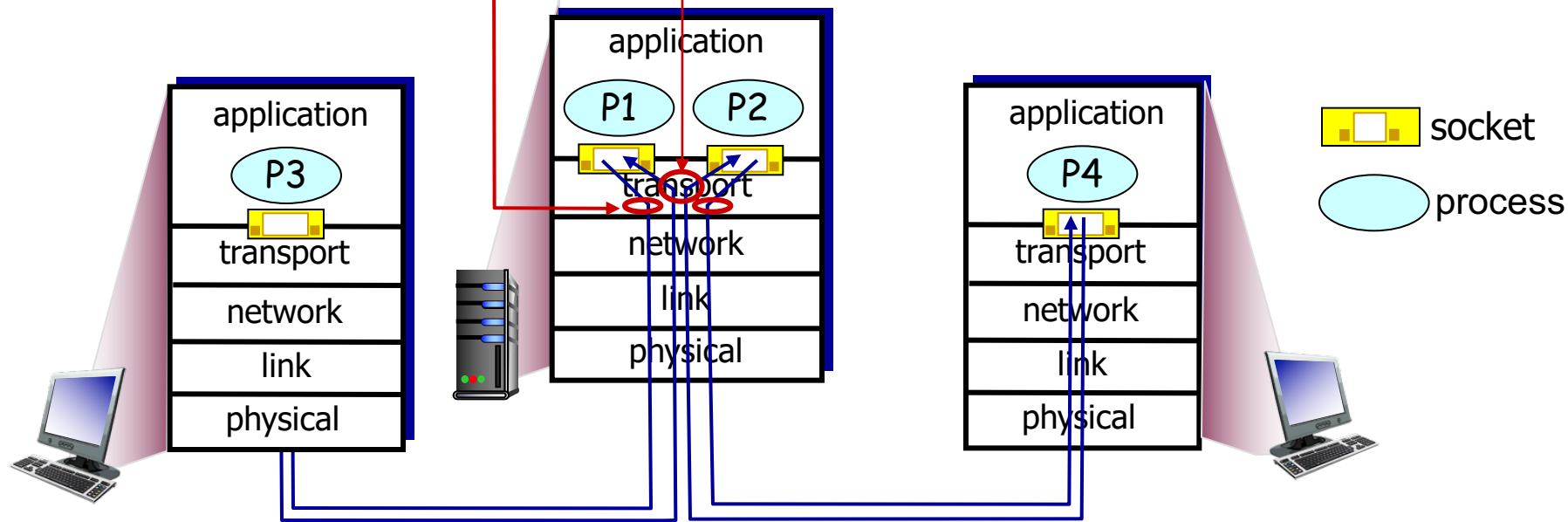
Multiplexing / demultiplexing

multiplexing at sender:

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

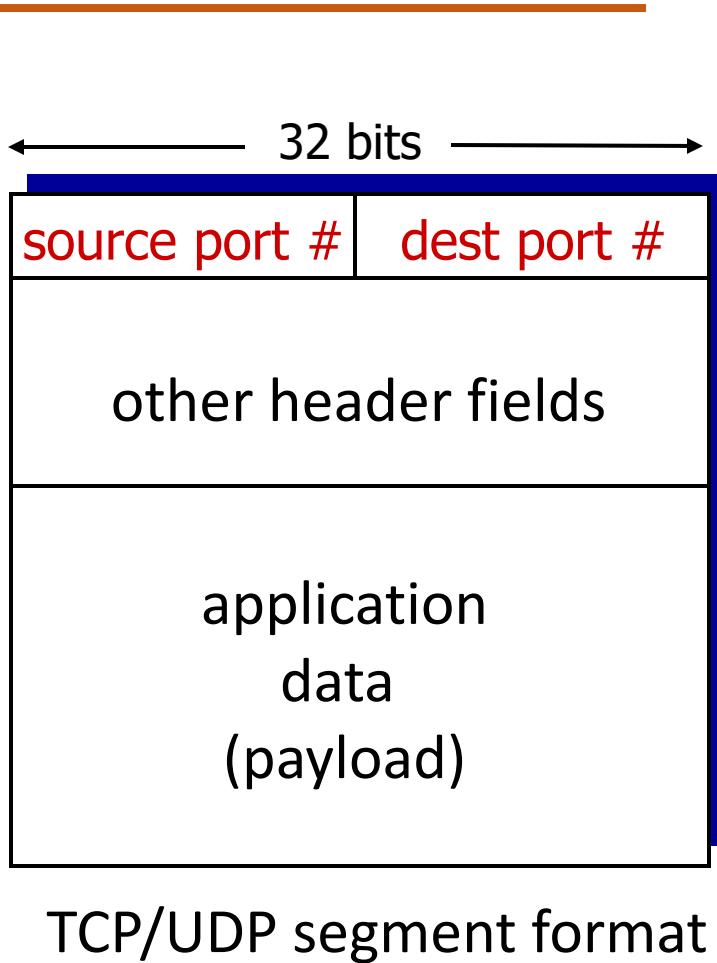
demultiplexing at receiver:

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



Connectionless demultiplexing

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

```
DatagramSocket mySocket1  
= new DatagramSocket(1234);
```

- *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 #



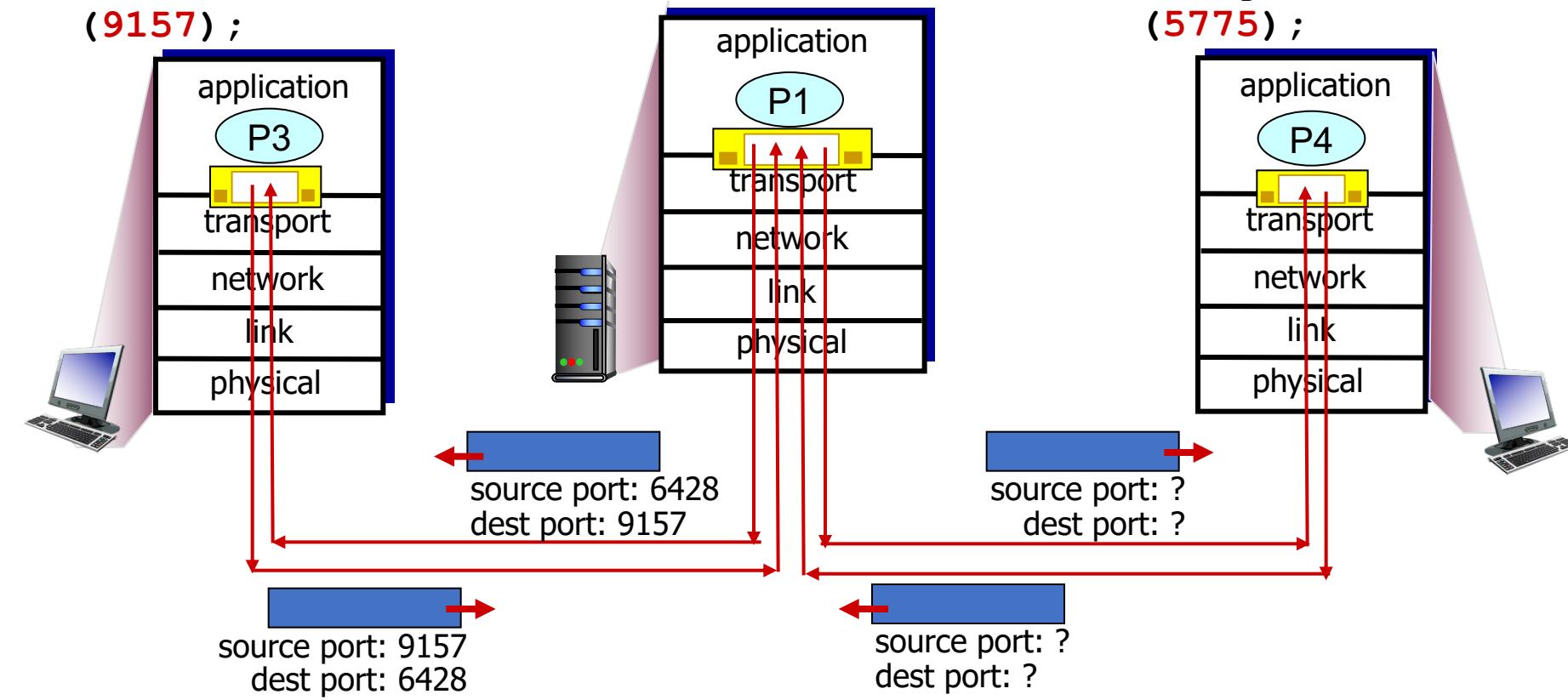
IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

Connectionless demux: example

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

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

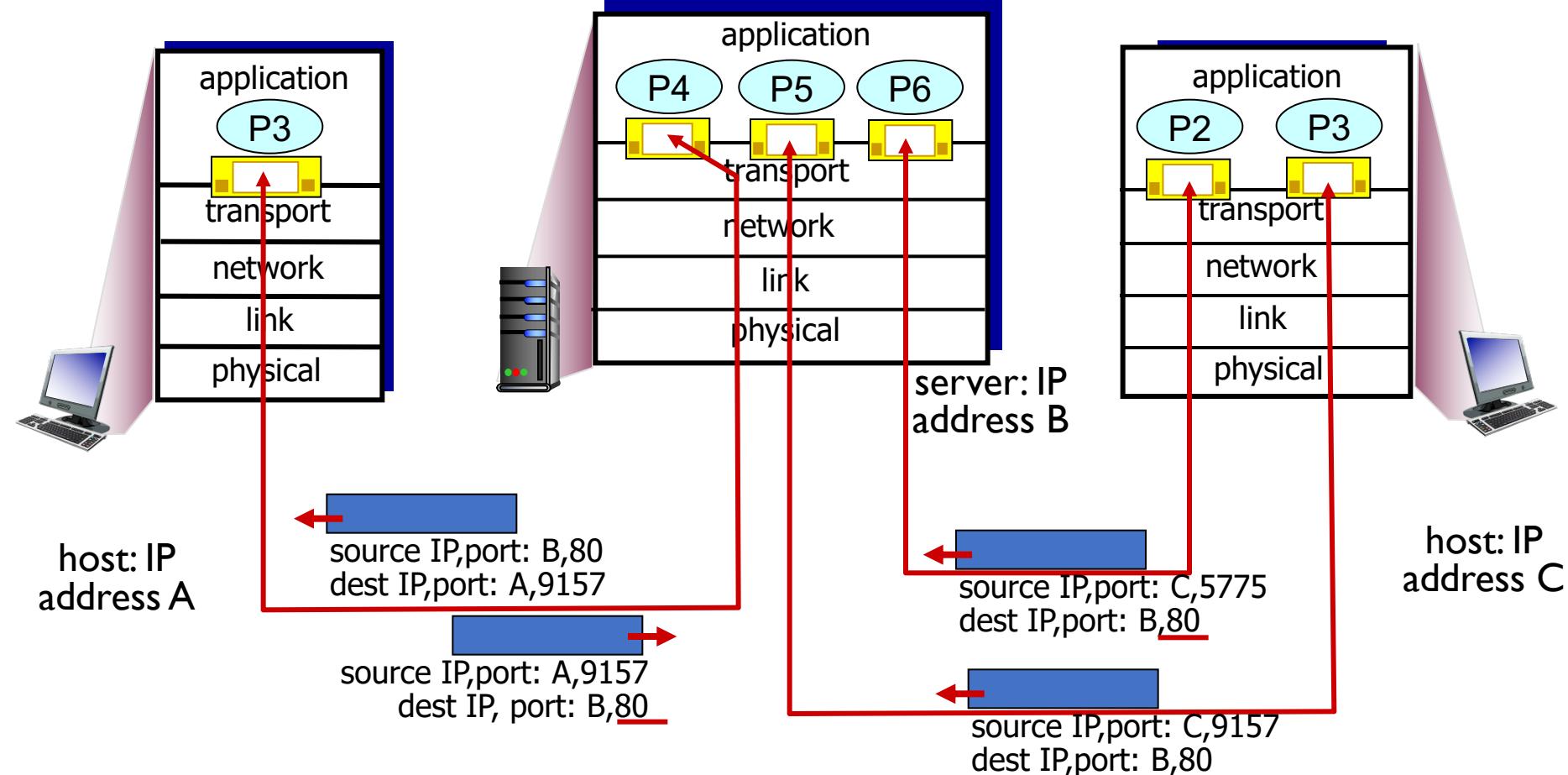
```
DatagramSocket
mySocket1 = new
DatagramSocket
(5775);
```



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 (4-tuple)** 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

COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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

COMPUTER NETWORKS

Connectionless Transport: UDP

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

- UDP: User Datagram Protocol [RFC 768]
- UDP: segment header
- UDP Checksum
 - Internet Checksum: example
 - Internet Checksum: weak protection!

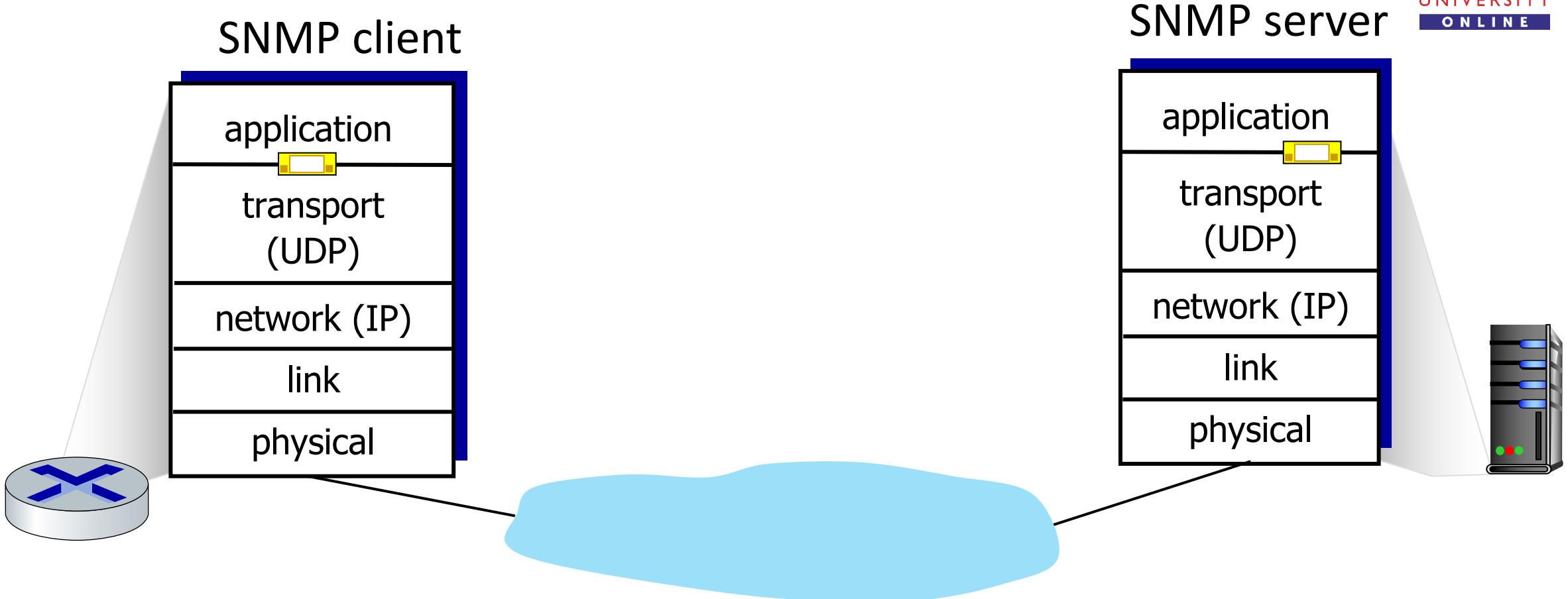
- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
 - lost
 - delivered out-of-order to app
- *connectionless*:
 - no handshaking between UDP sender, receiver
 - each UDP segment handled independently of others

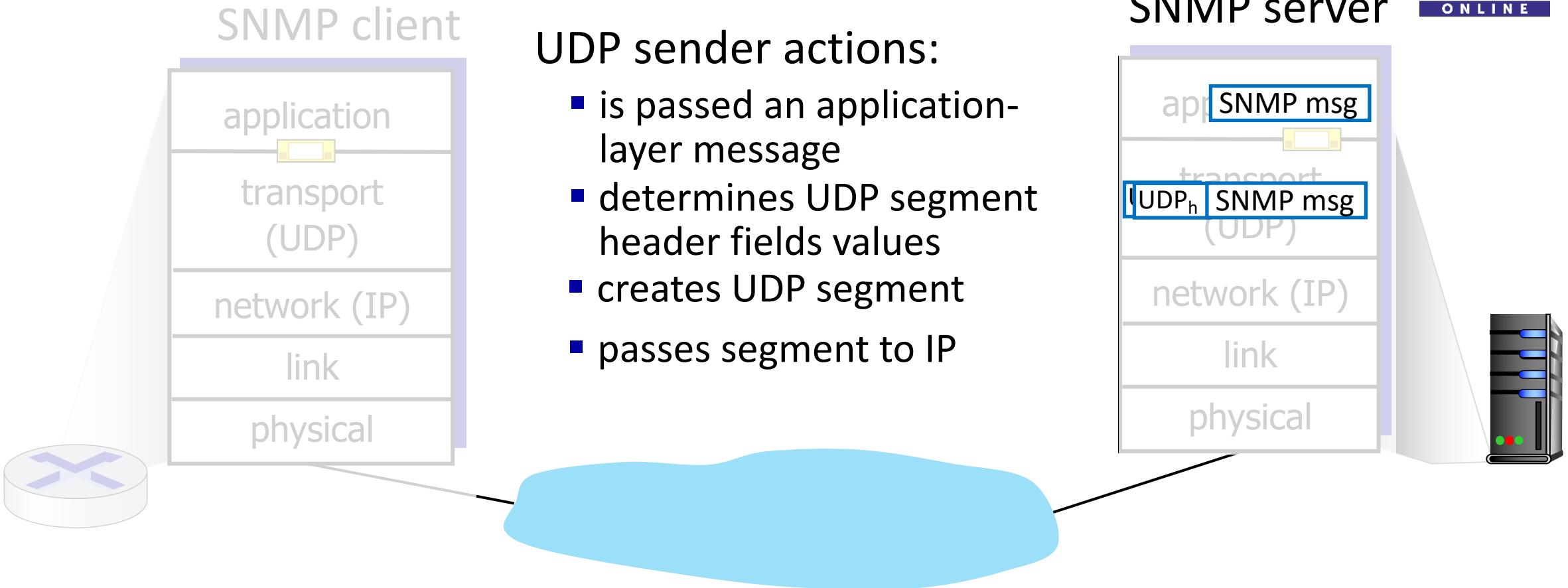
why is there a UDP?

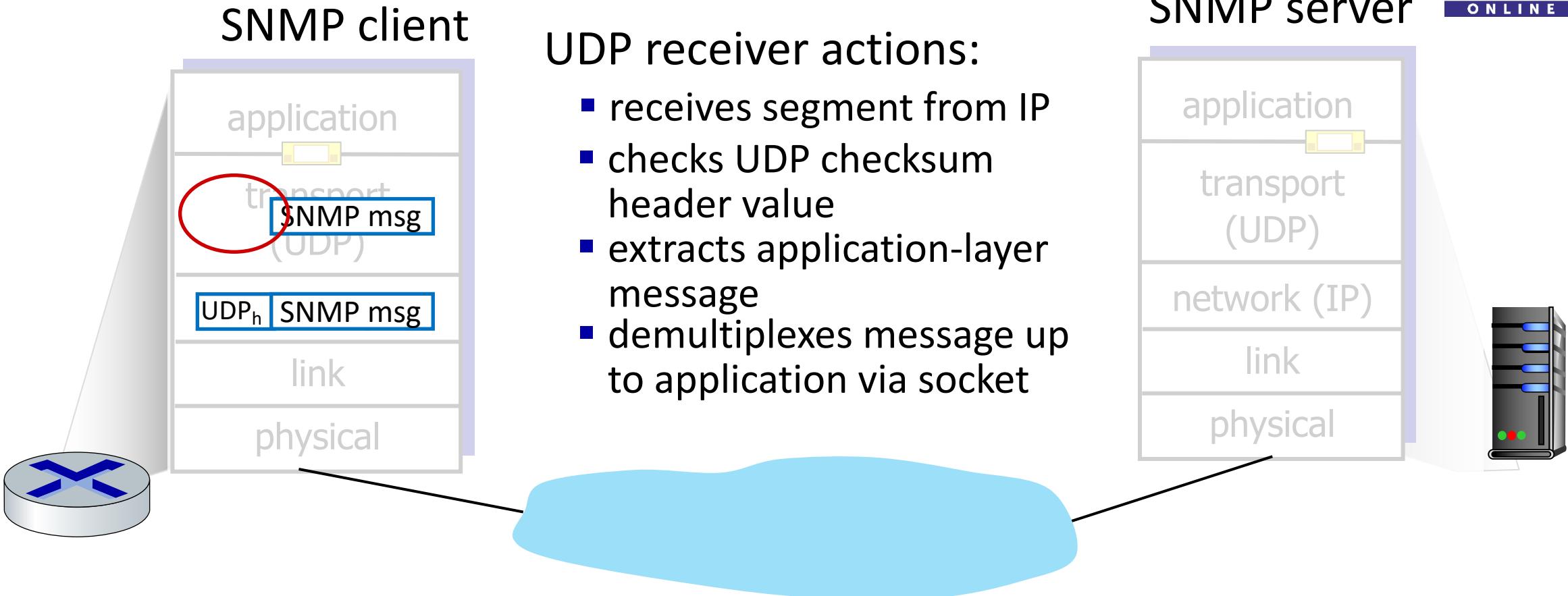
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
 - UDP can blast away as fast as desired
 - can function in the face of congestion

- UDP use:
 - streaming multimedia apps (loss tolerant, rate sensitive)
 - DNS
 - SNMP

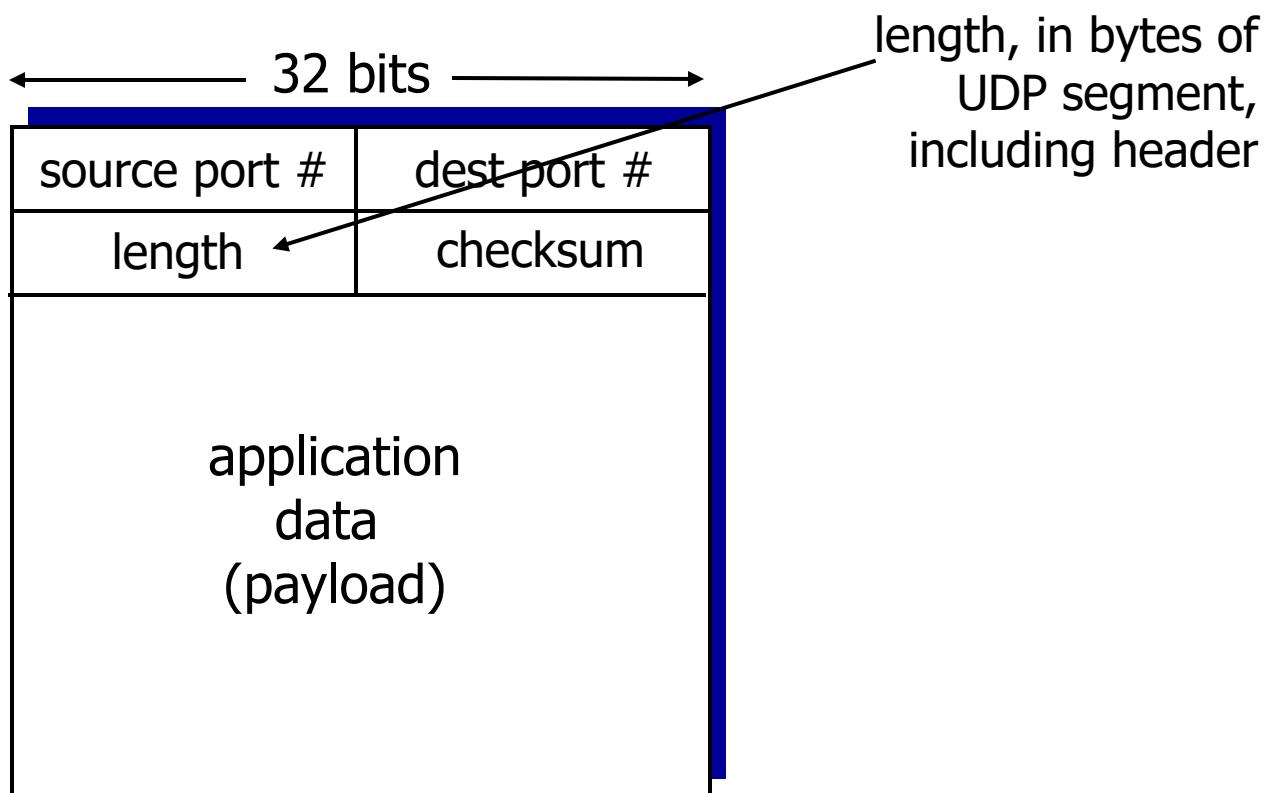
- If reliable transfer needed over UDP:
 - add needed reliability at application layer
 - application-specific error recovery!
 - add congestion control at application layer







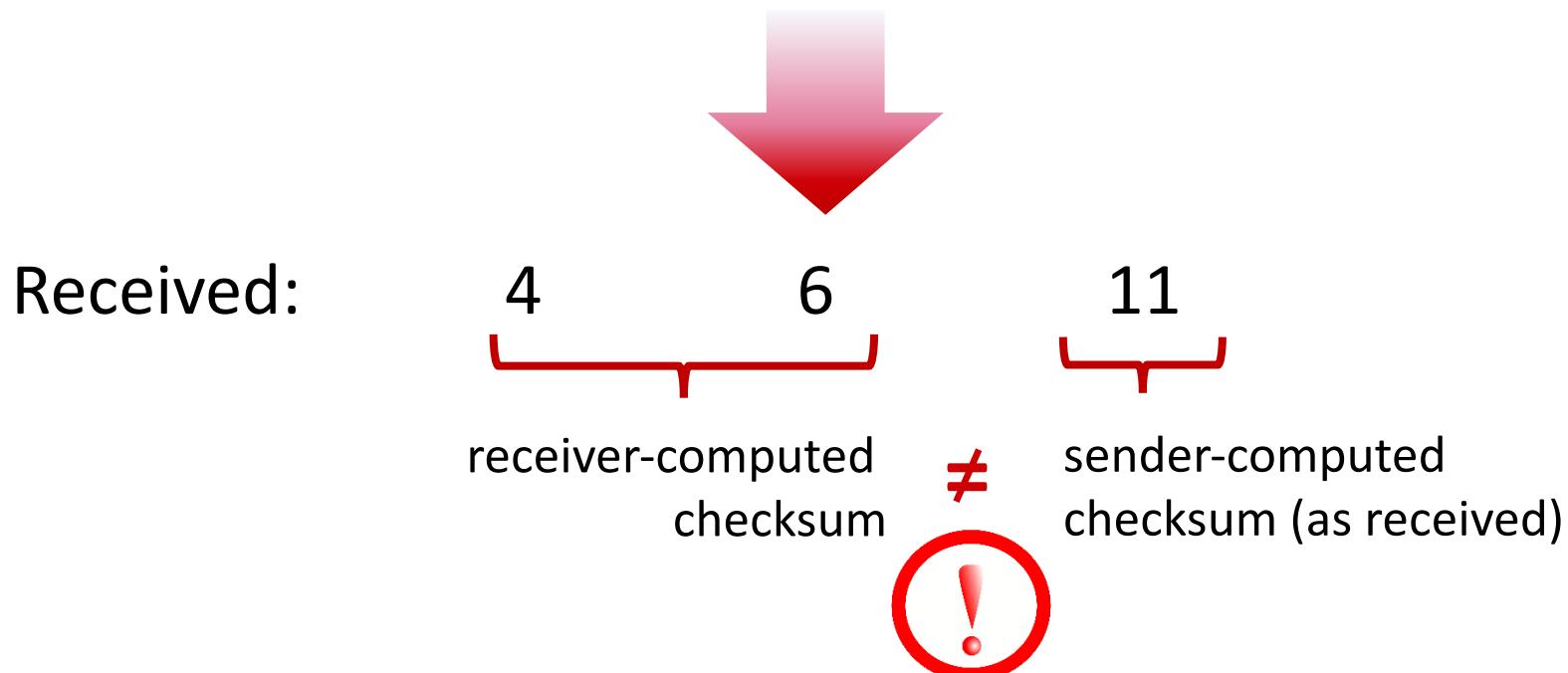
UDP: segment header



UDP segment format

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

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



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
...

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

example: add two 16-bit integers

	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	0
<hr/>																	
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	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1	0

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

COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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

COMPUTER NETWORKS

Principles of reliable date transfer

Ashwini M Joshi

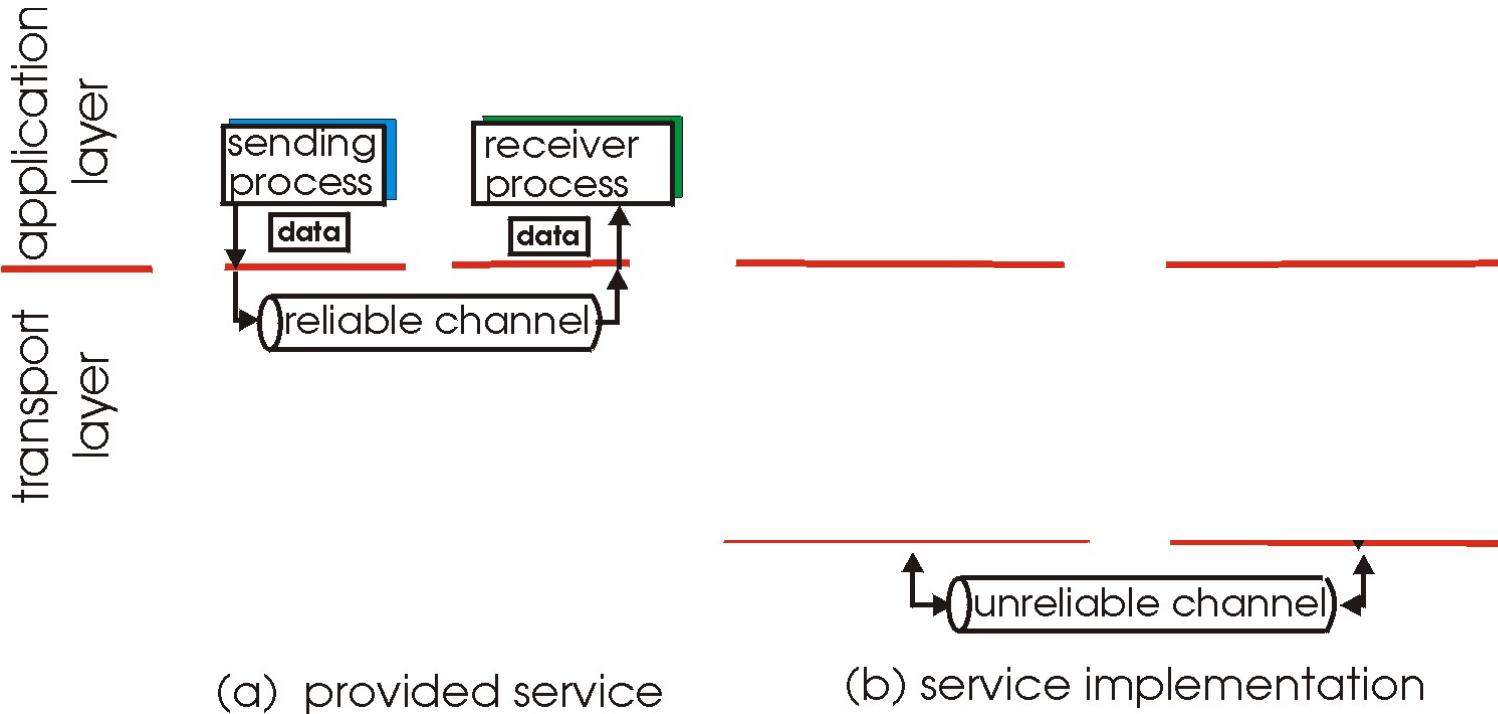
Department of Computer Science & Engineering

In this segment

- Principles of reliable data transfer
- Reliable data transfer: getting started
- rdt1.0: reliable transfer over a reliable channel

Principles of reliable data transfer

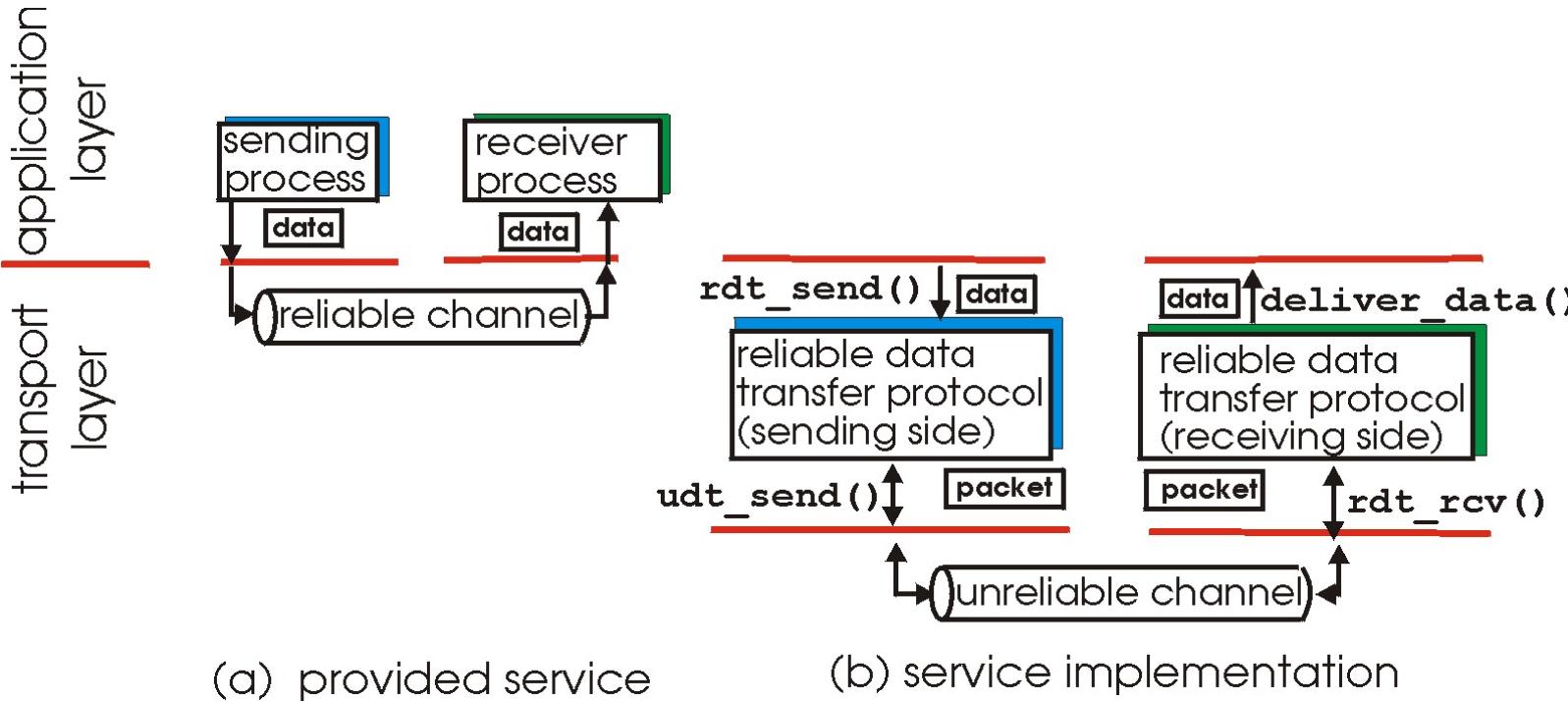
- important in application, transport, link layers
 - top-10 list of important networking topics!



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of reliable data transfer

- important in application, transport, link layers
 - top-10 list of important networking topics!



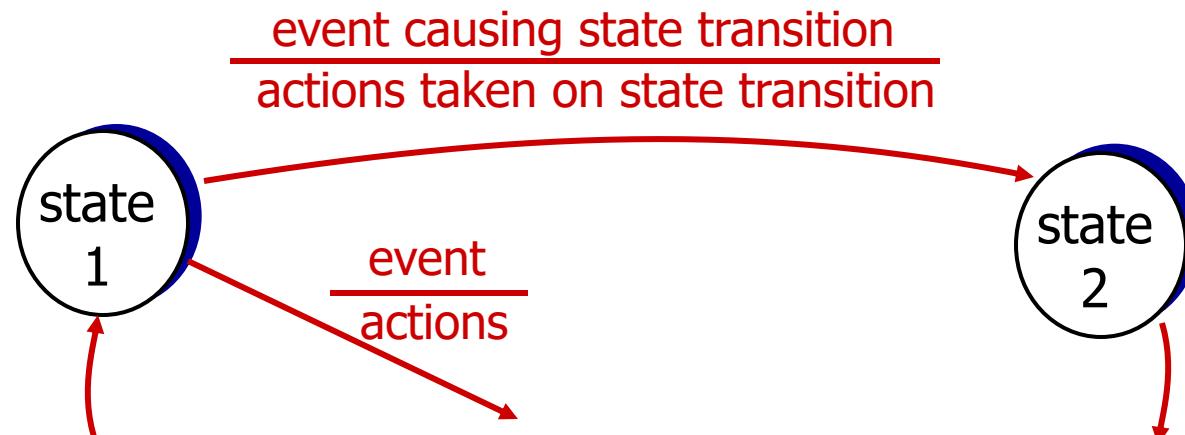
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

we'll:

- 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: when in this “state” next state uniquely determined by next event

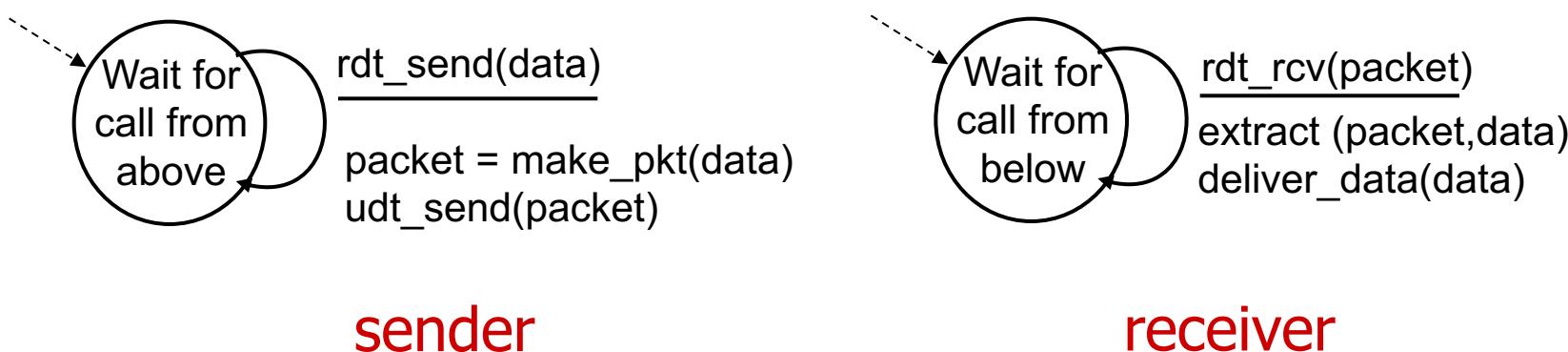


- underlying channel perfectly reliable

- no bit errors
 - no loss of packets

- separate FSMs for sender, receiver:

- sender sends data into underlying channel
 - receiver reads data from underlying channel



COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

COMPUTER NETWORKS

Principles of reliable date transfer

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

- rdt2.0: channel with bit errors
- rdt2.0: FSM specification
- rdt2.0: operation with no errors
- rdt2.0: error scenario
- rdt2.0 has a fatal flaw!
- rdt2.1: sender, handles garbled ACK/NAKs
- rdt2.1: discussion
- rdt2.2: a NAK-free protocol
- rdt2.2: sender, receiver fragments

- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the question:* how to recover from errors:

How do humans recover from “errors”
during conversation?

- underlying channel may flip bits in packet
 - checksum to detect bit errors
- *the question:* how to recover from errors:
 - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
 - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
 - sender **retransmits** pkt on receipt of NAK
- new mechanisms in **rdt2 . 0** (beyond **rdt1 . 0**):
 - error detection
 - feedback: control msgs (ACK,NAK) from receiver to sender

stop and wait

sender sends one packet, then waits for receiver response

rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait

sender sends one packet,
then waits for receiver
response

sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
 - state must “remember” whether “expected” pkt should have seq # of 0 or 1

receiver:

- must check if received packet is duplicate
 - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
 - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:
retransmit current pkt

COMPUTER NETWORKS

Principles of reliable date transfer

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

- rdt3.0: channels with errors and loss
- rdt3.0 sender
- rdt3.0 in action
- Performance of rdt3.0
- rdt3.0: stop-and-wait operation

new assumption: underlying channel can also lose packets (data, ACKs)

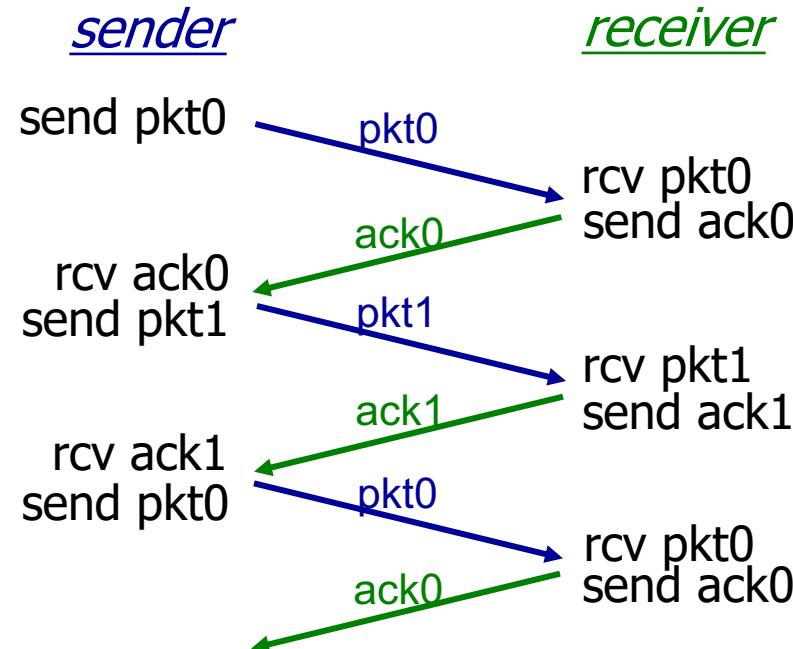
- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough



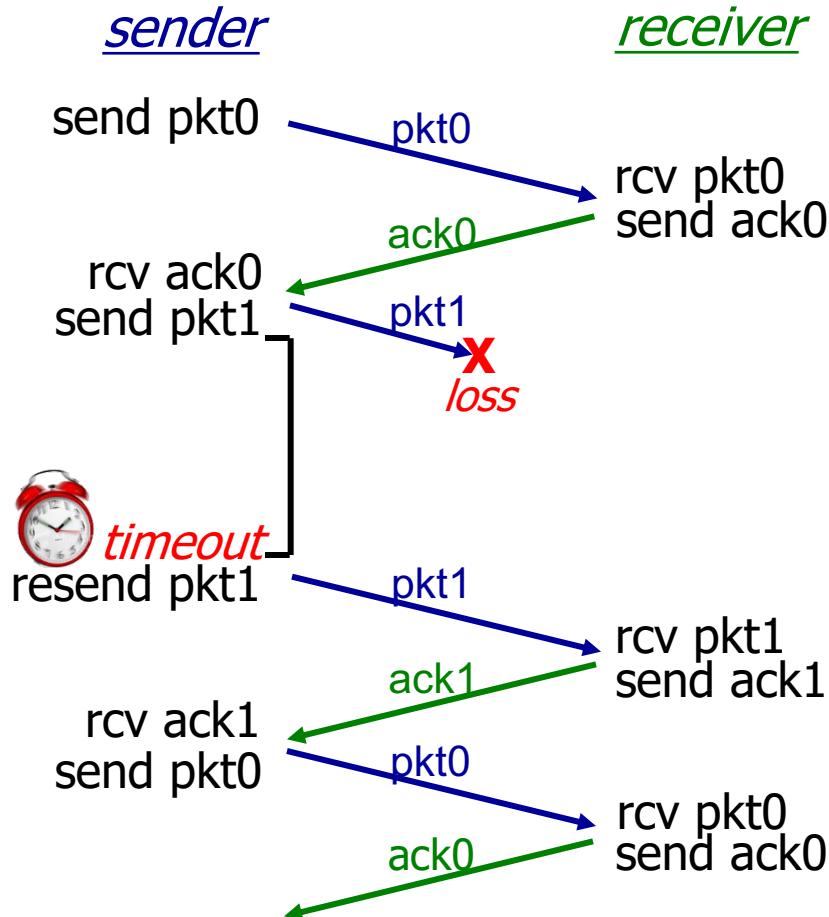
timeout

approach: sender waits “reasonable” amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but seq. #'s already handles this
 - receiver must specify seq # of pkt being ACKed
- requires countdown timer



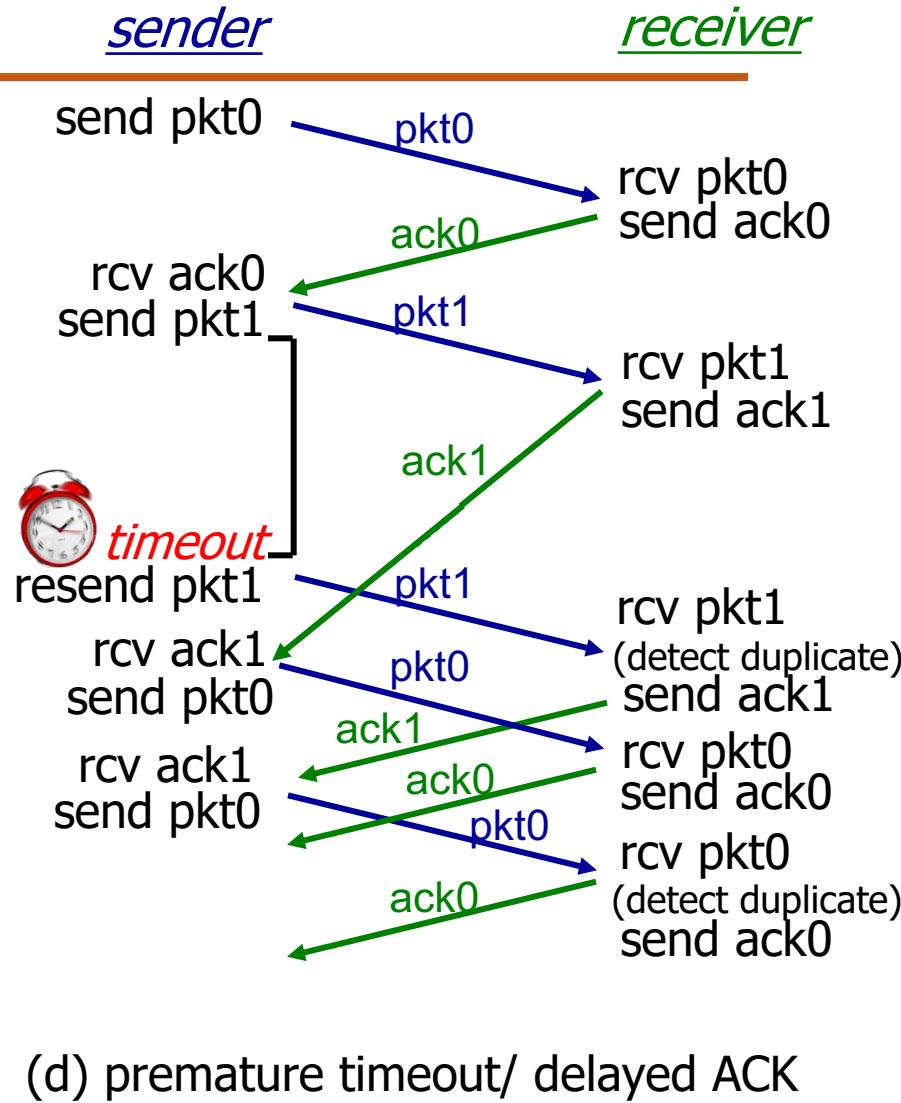
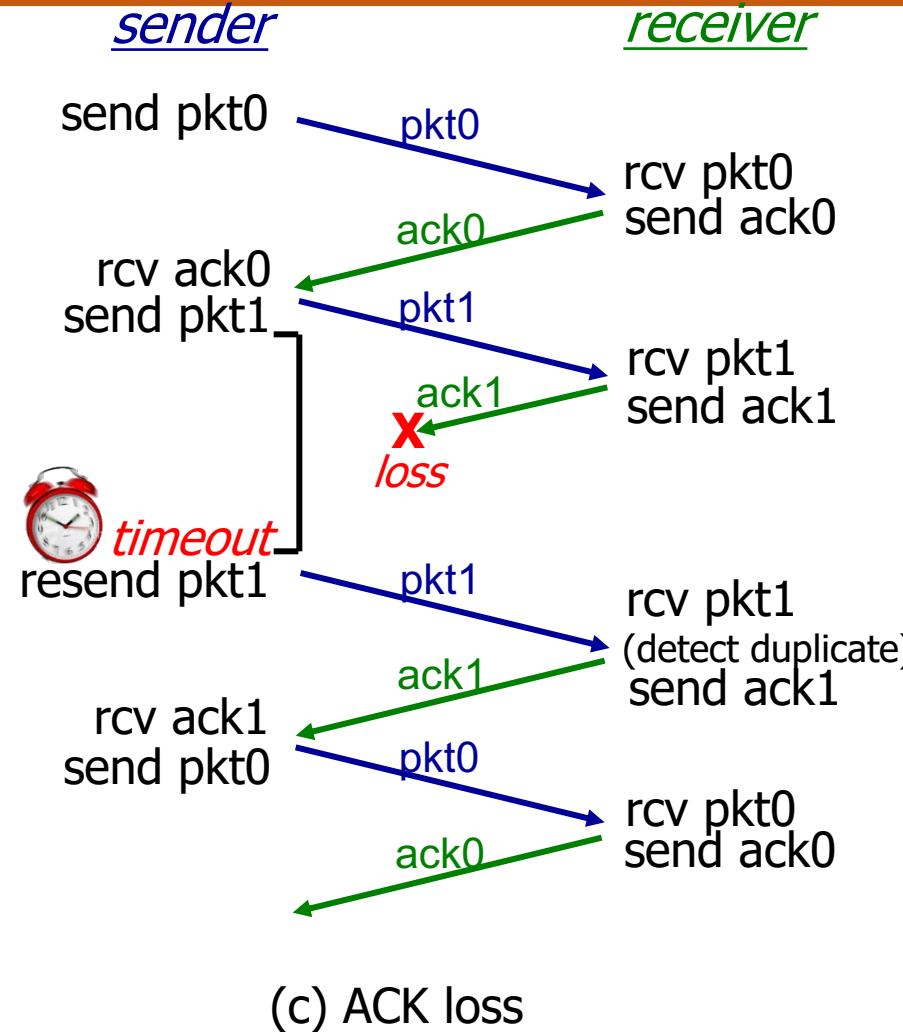
(a) no loss



(b) packet loss

COMPUTER NETWORKS

rdt3.0 in action



- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

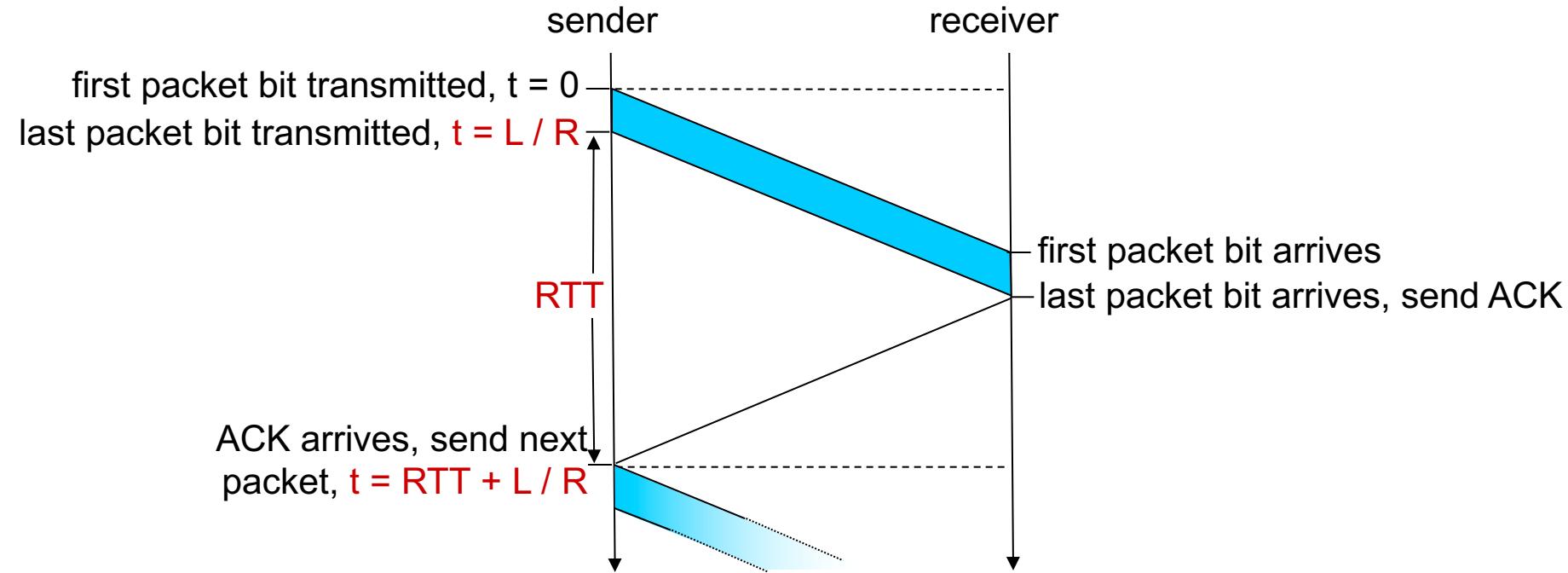
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- U_{sender} : *utilization* – fraction of time sender busy sending

$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation



$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

COMPUTER NETWORKS

Pipelined protocols

Ashwini M Joshi

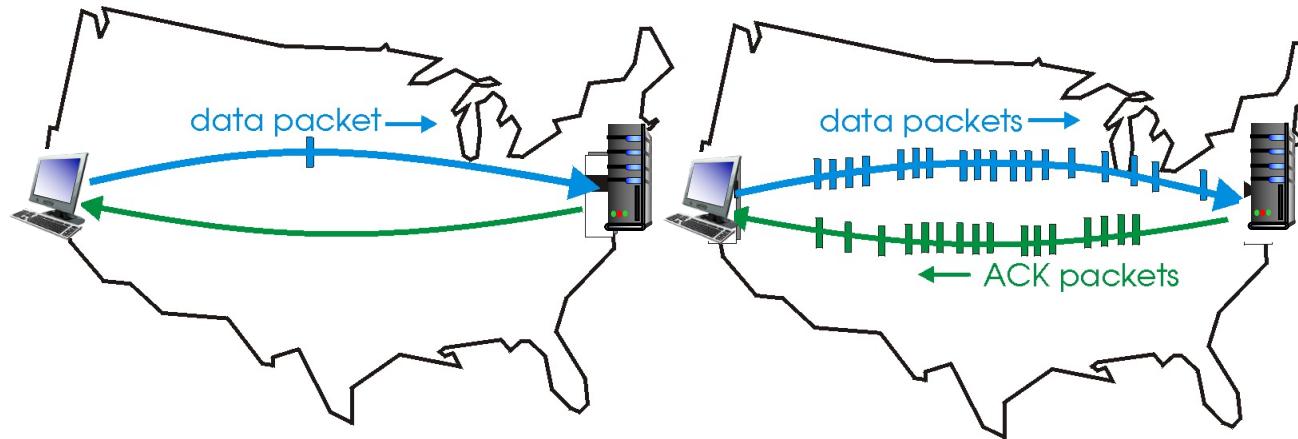
Department of Computer Science & Engineering

In this segment

- Pipelined protocols
- Pipelining: increased utilization
- Pipelined protocols: overview
- Go-Back-N: sender
- GBN: sender extended FSM
- GBN: receiver extended FSM
- Selective repeat
- Selective repeat: sender, receiver windows
- Selective repeat in action
- Selective repeat: dilemma

pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

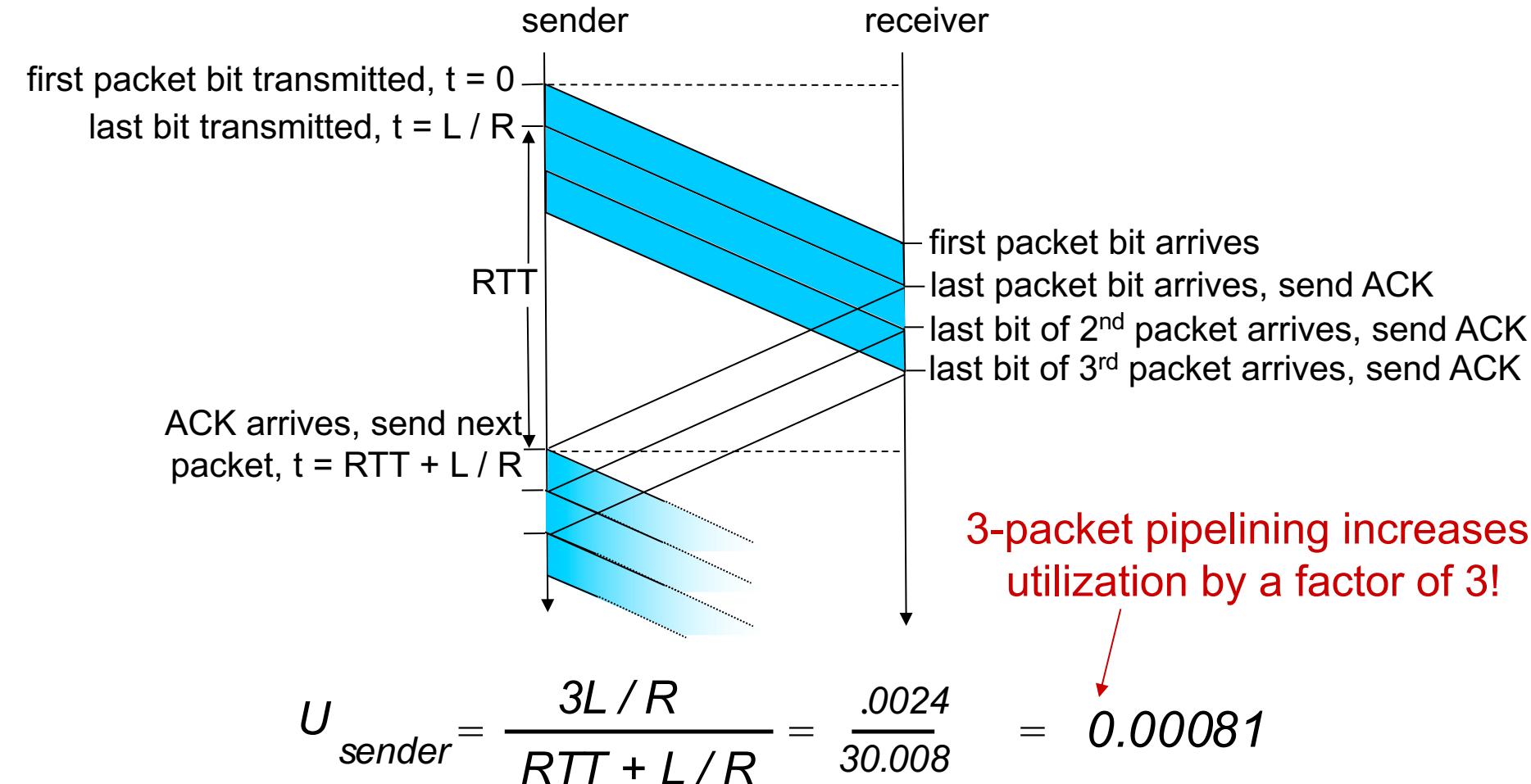


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- two generic forms of pipelined protocols: *go-Back-N*, *selective repeat*

Pipelining: increased utilization



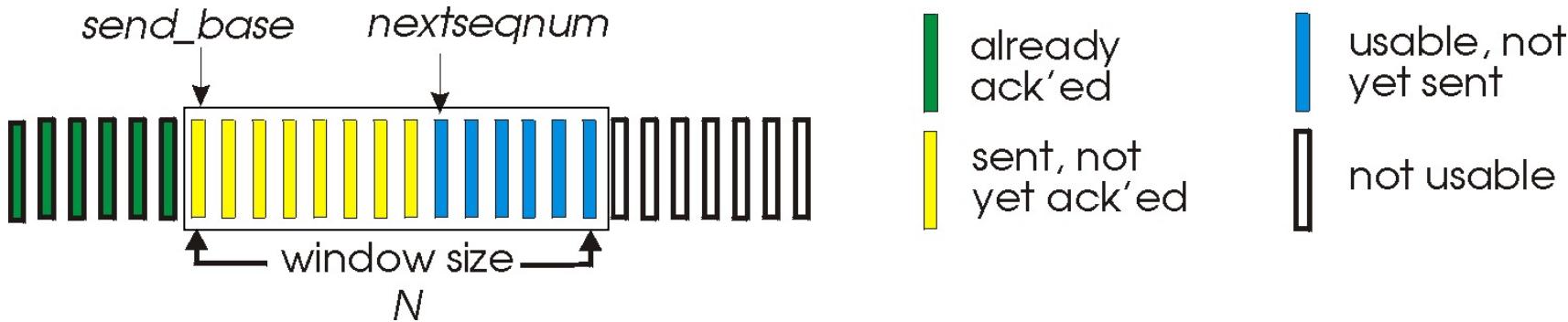
Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
 - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
 - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unacked packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
 - when timer expires, retransmit only that unacked packet

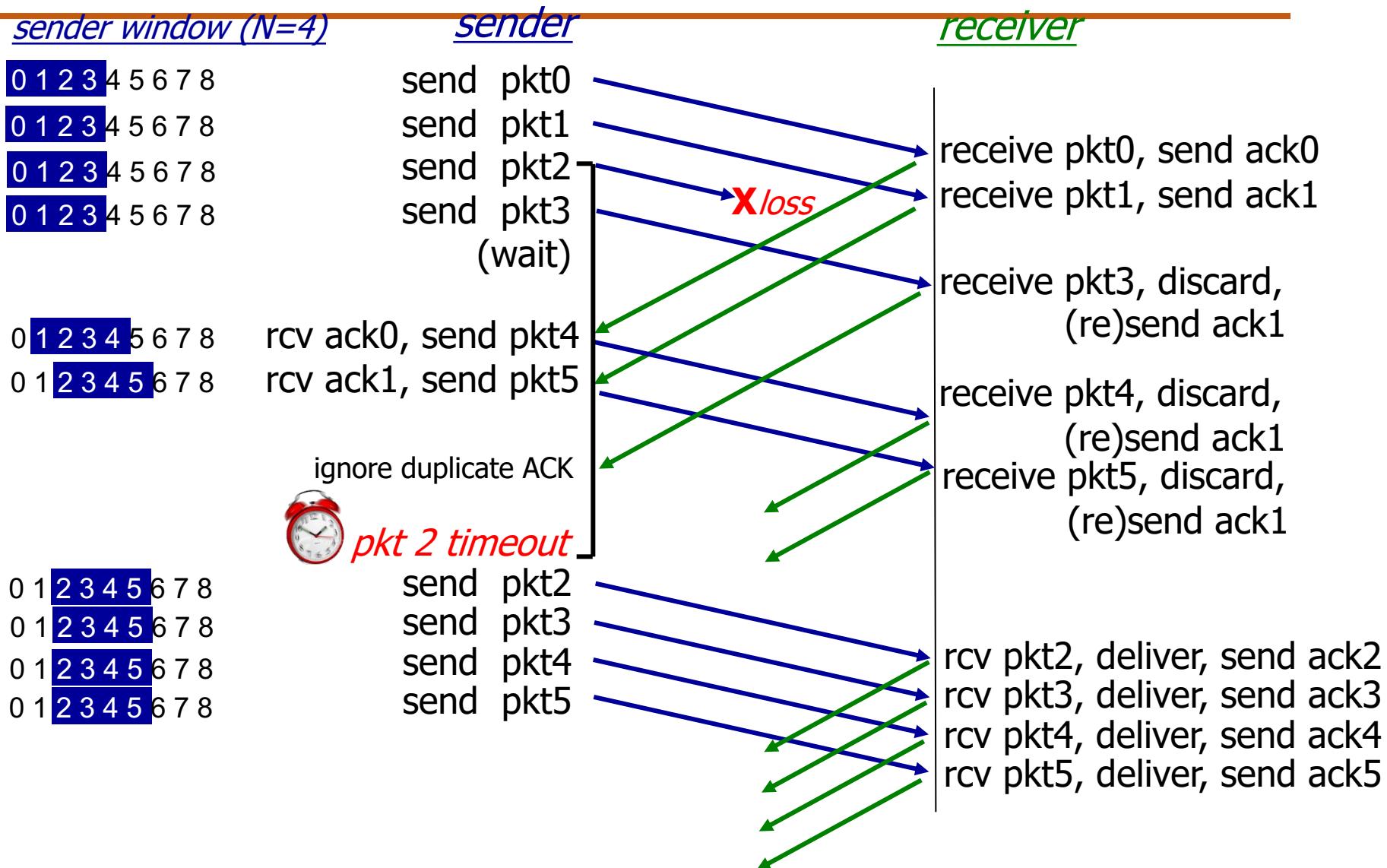
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - “**cumulative ACK**”
 - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

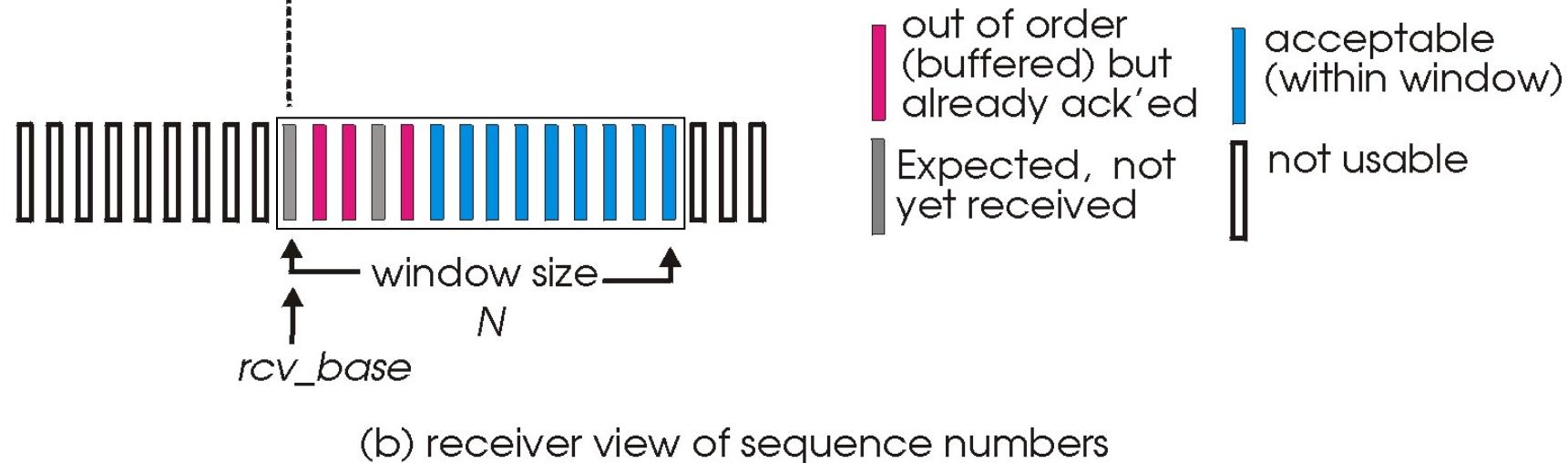
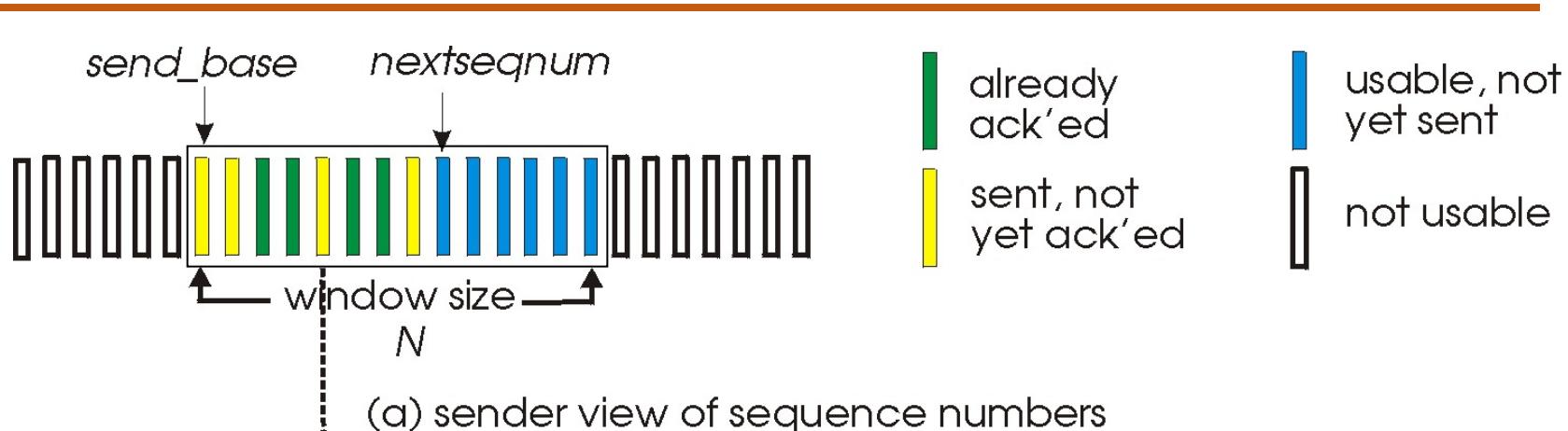
COMPUTER NETWORKS

GBN in action



- receiver *individually* acknowledges all correctly received pkts
 - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
 - sender timer for each unACKed pkt
- sender window
 - N consecutive seq #'s
 - limits seq #'s of sent, unACKed pkts

Selective repeat: sender, receiver windows





sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in

[sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in $[rcvbase, rcvbase+N-1]$

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

pkt n in $[rcvbase-N, rcvbase-1]$

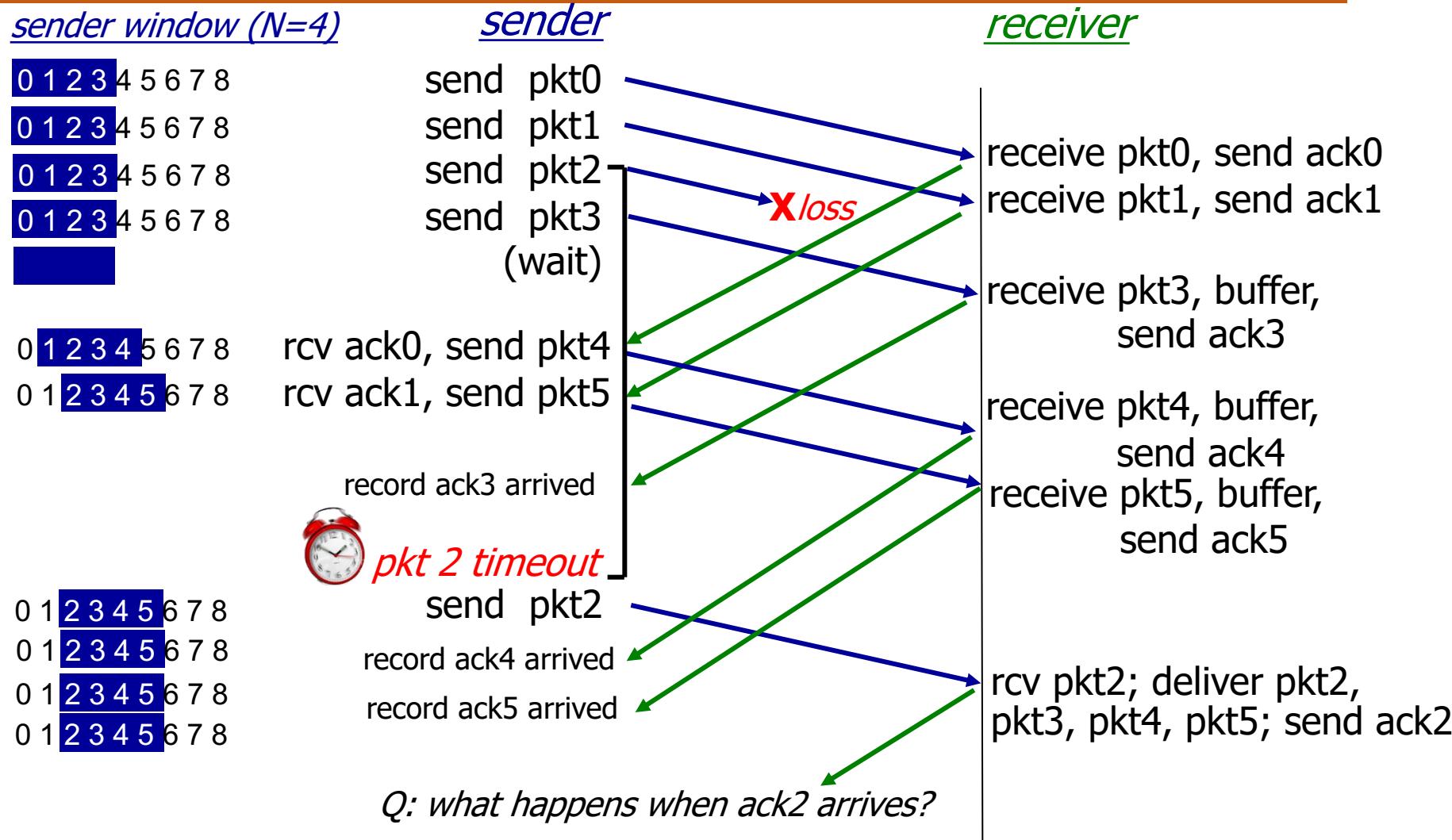
- ACK(n)

otherwise:

- ignore

COMPUTER NETWORKS

Selective repeat in action

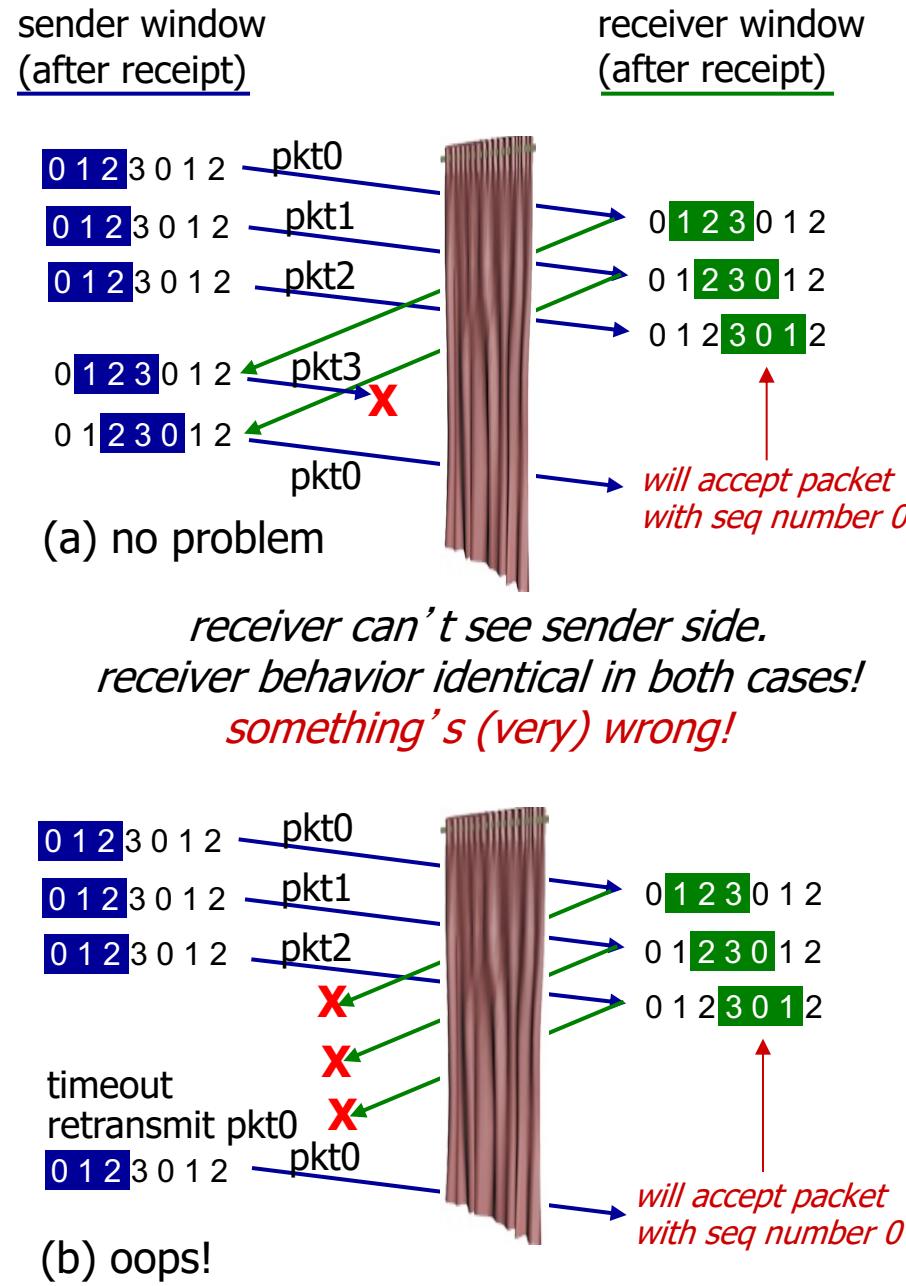


Selective repeat: dilemma

example:

- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



Exercise Problem

In GB3 if every 5th packet that is transmitted is lost and if we have to send 10 packets. Then how many transitions are required?

In GB3 if every 5th packet that is transmitted is lost and if we have to send 10 packets. Then how many transitions are required?

1 2 3 4 5 6 7 | 8 9 10 ^ \$ (packet no. 5 lost)

1 2 3 4 5 6 7 5 6 7 8 9 | 10

* ^ \$

1 2 3 4 5 6 7 5 6 7 8 9 7 8 9 10

* ^ \$

1 2 3 4 5 6 7 5 6 7 8 9 7 8 9 10 9 10

(count starts from * till ^)

(from ^ to \$ retransmission is done)

From the Last packet is window size to lost pocket we resend the entire window.

Total no. of transmissions = 18

COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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

COMPUTER NETWORKS

Connection-oriented transport: TCP

Ashwini M Joshi

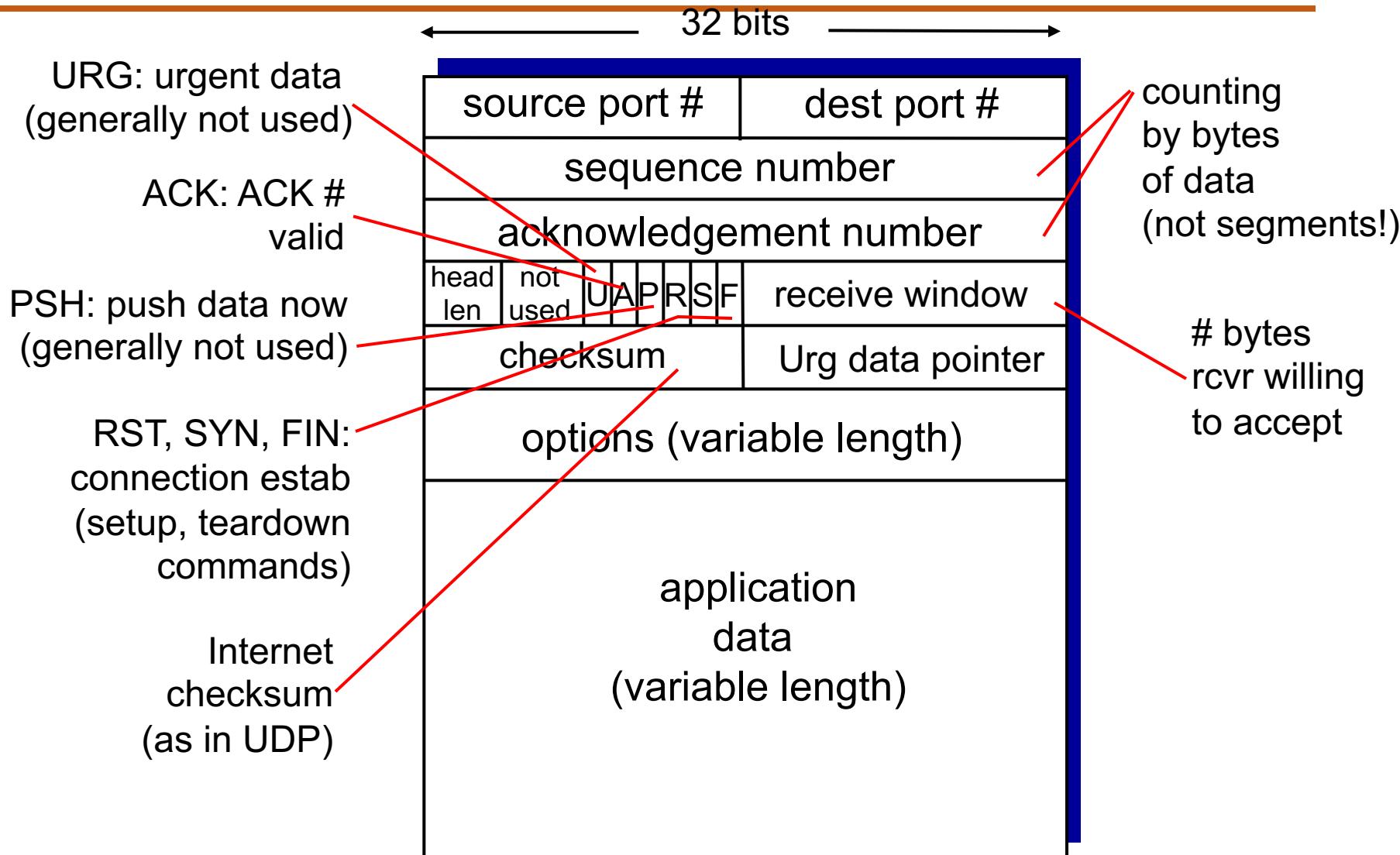
Department of Computer Science & Engineering

In this segment

- TCP: Overview RFCs: 793,1122,1323, 2018, 2581
- TCP segment structure
- TCP seq. numbers, ACKs
- TCP round trip time, timeout

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

TCP segment structure



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

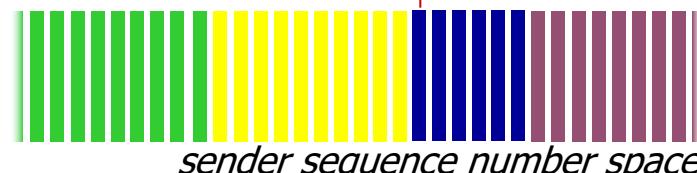
- A: TCP spec doesn’t say, - up to implementor

outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

window size

N



sender sequence number space

sent
ACKed

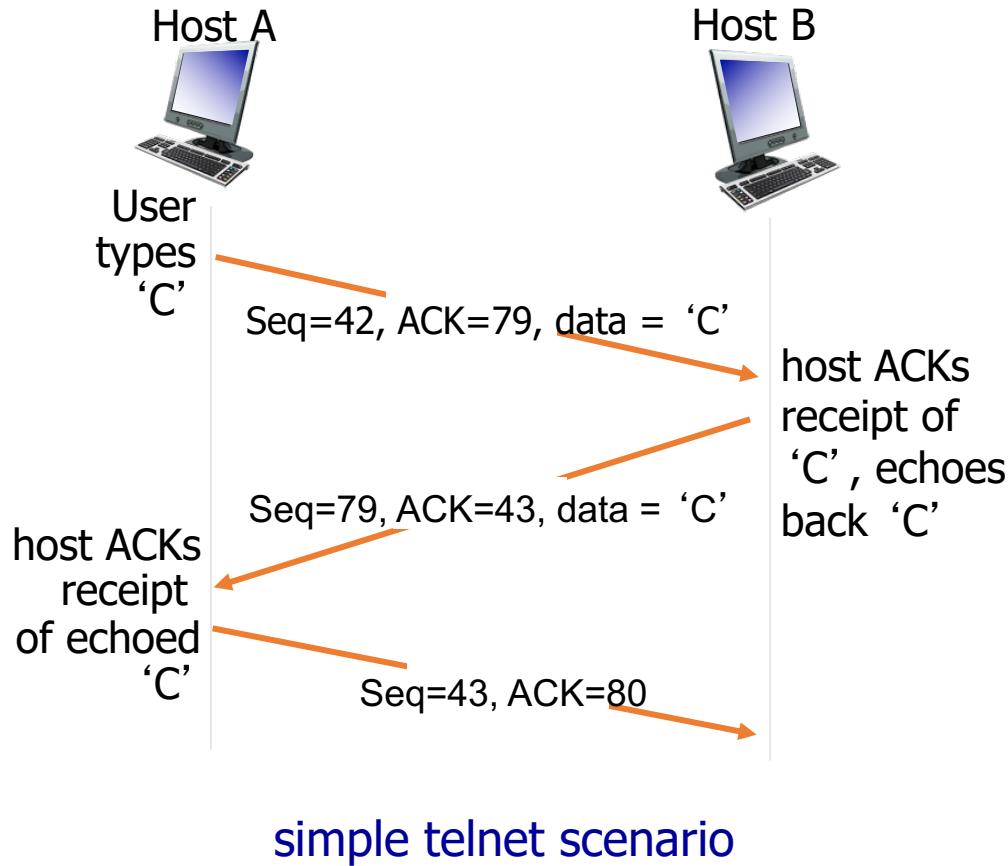
sent, not-
yet ACKed
("in-
flight")

usable
but not
yet sent

not
usable

incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
A	rwnd
checksum	urg pointer



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

COMPUTER NETWORKS

**Connection-oriented transport: TCP
Reliable data transfer**

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

- TCP reliable data transfer
- TCP sender events
- TCP sender (simplified)
- TCP: retransmission scenarios
- TCP ACK generation [RFC 1122, RFC 2581]
- TCP fast retransmit

- TCP creates rdt service on top of IP's unreliable service
 - pipelined segments
 - cumulative acks
 - single retransmission timer
 - retransmissions triggered by:
 - timeout events
 - duplicate acks
- let's initially consider simplified TCP sender:
- ignore duplicate acks
 - ignore flow control, congestion control

data rcvd from app:

- 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

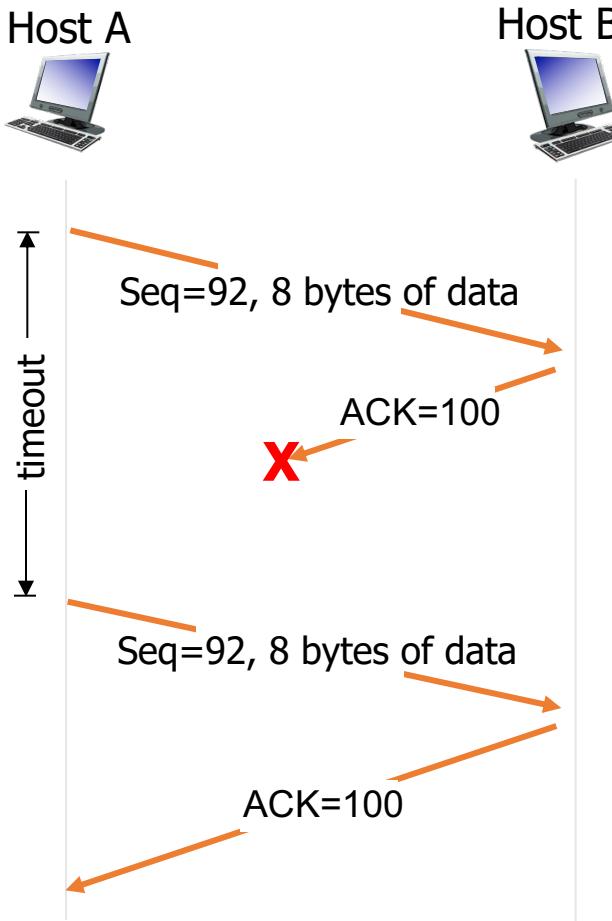
timeout:

- retransmit segment that caused timeout
- restart timer

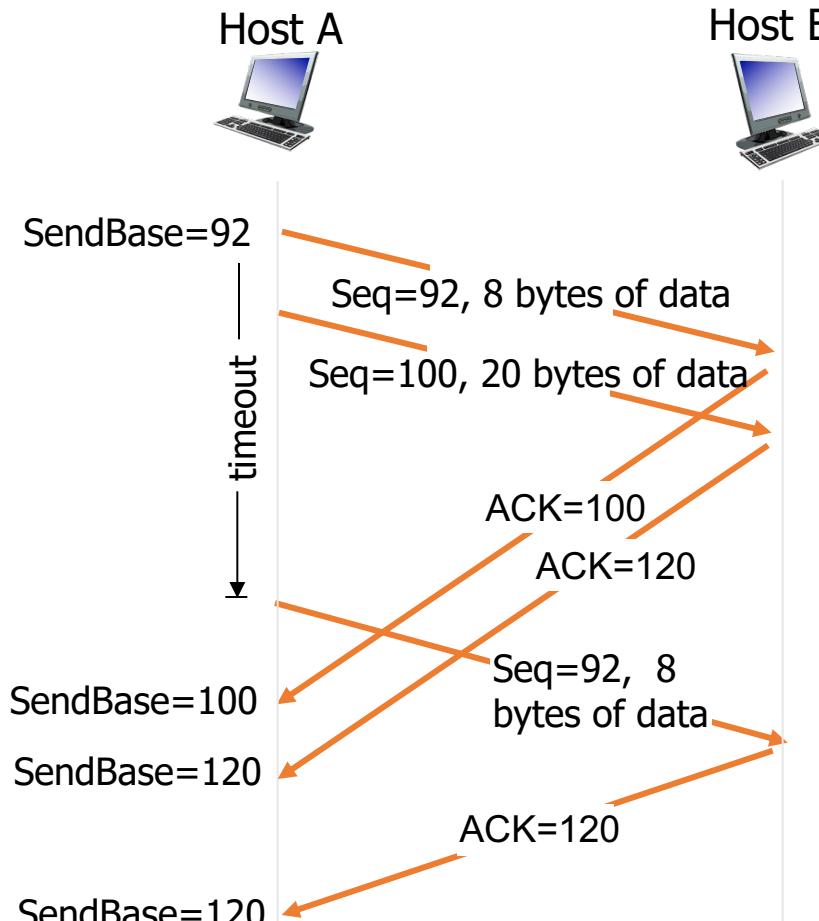
ack rcvd:

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

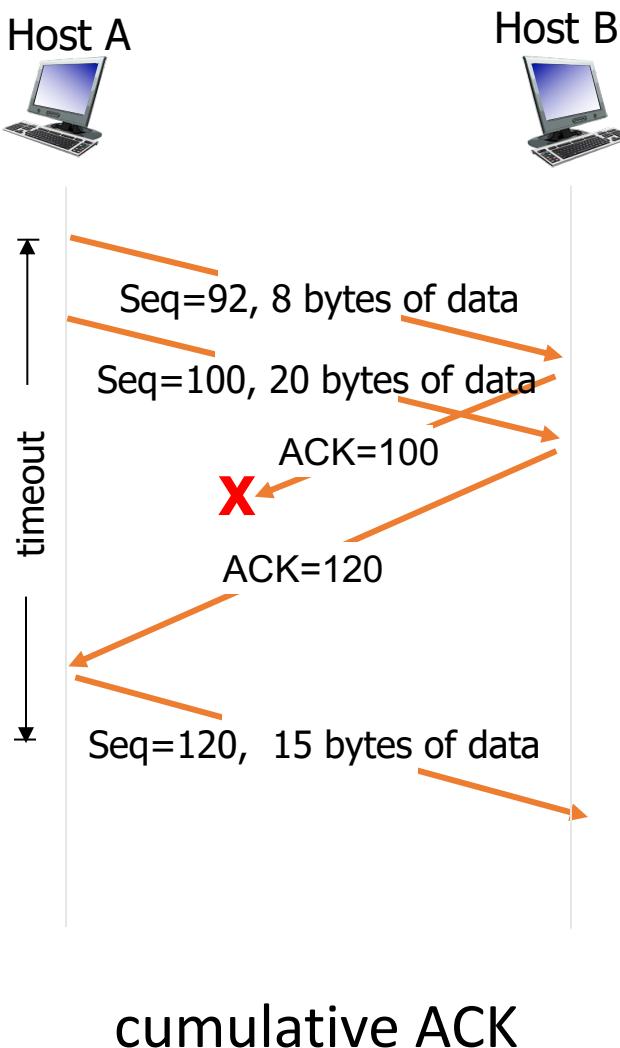
TCP: retransmission scenarios



lost ACK scenario



premature timeout



<i>event at receiver</i>	<i>TCP receiver action</i>
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send <i>duplicate ACK</i> , indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

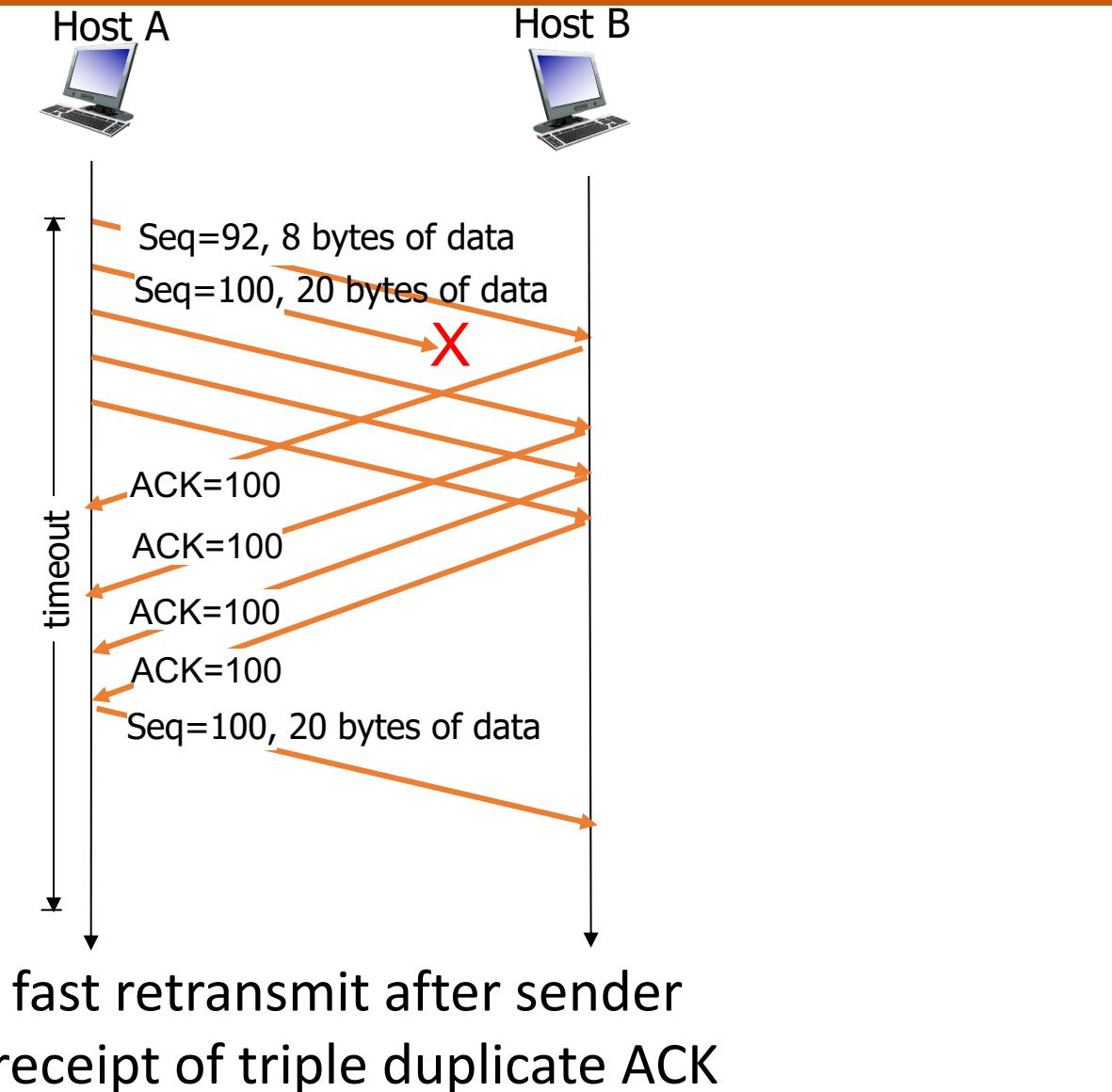
- time-out period often relatively long:
 - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
 - sender often sends many segments back-to-back
 - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data

(“triple duplicate ACKs”),
resend unacked segment with smallest seq #

- likely that unacked segment lost, so don’t wait for timeout



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

COMPUTER NETWORKS

Connection-oriented transport: TCP Flow control

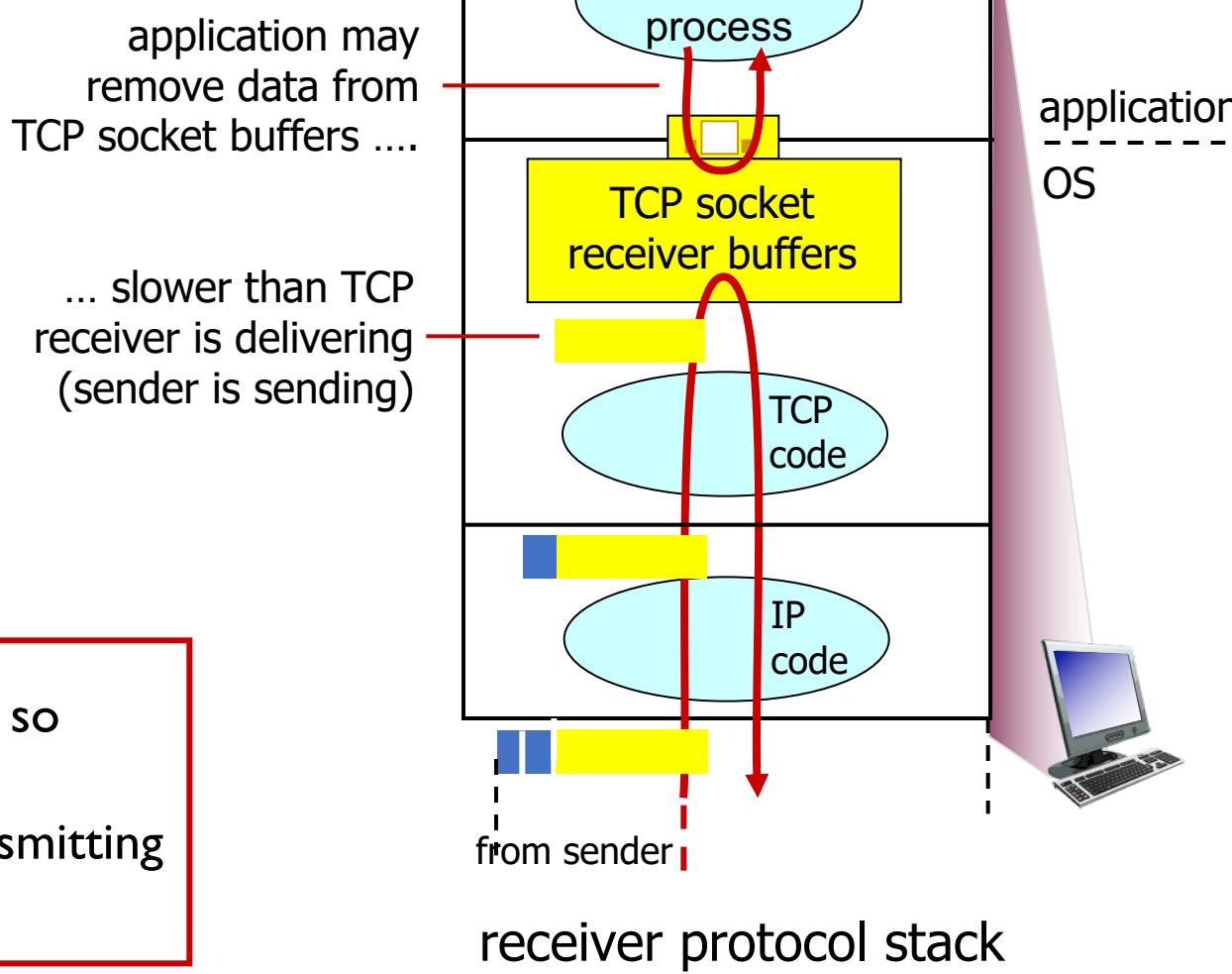
Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

- TCP flow control

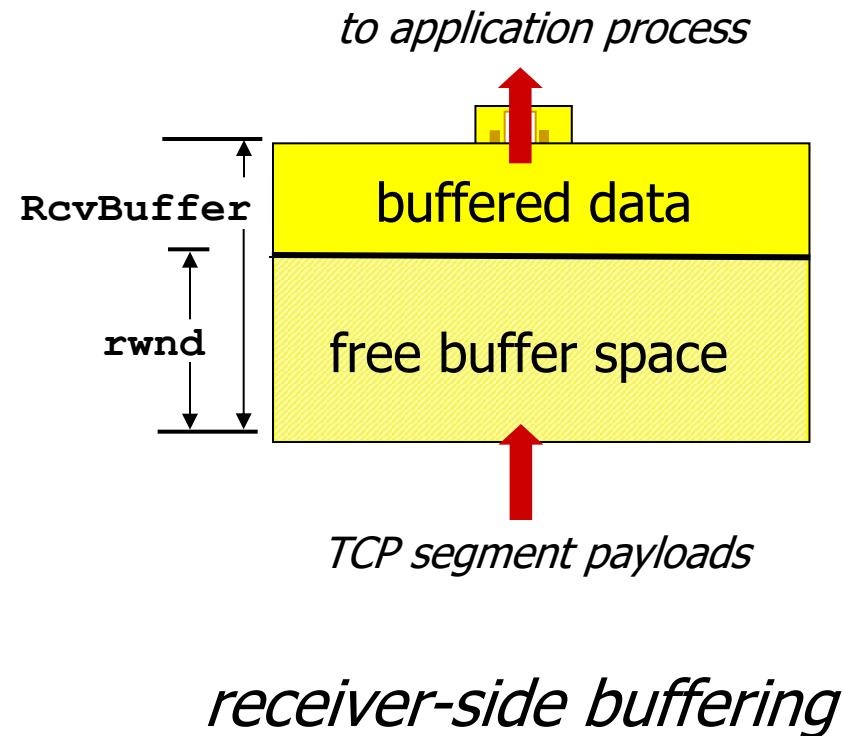
TCP flow control



flow control

receiver controls sender, so
sender won't overflow
receiver's buffer by transmitting
too much, too fast

- receiver “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
 - **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 receiver’s **rwnd** value
- guarantees receive buffer will not overflow



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

COMPUTER NETWORKS

Connection-oriented transport: TCP Connection Management

Ashwini M Joshi

Assistant Professor, Department of Computer Science & Engineering

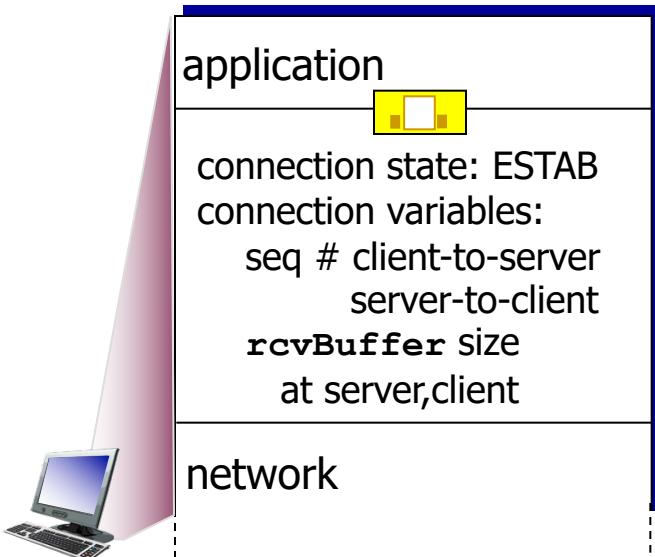
In this segment

- Connection Management
- Agreeing to establish a connection
- TCP 3-way handshake
- TCP 3-way handshake: FSM
- TCP: closing a connection

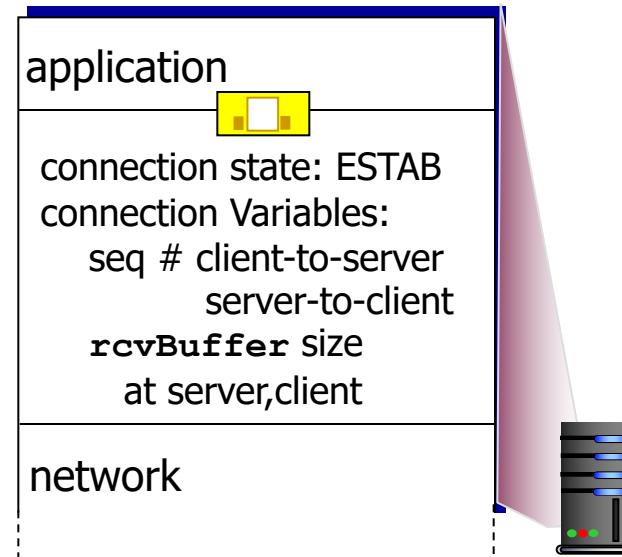
Connection Management

before exchanging data, sender/receiver “handshake”:

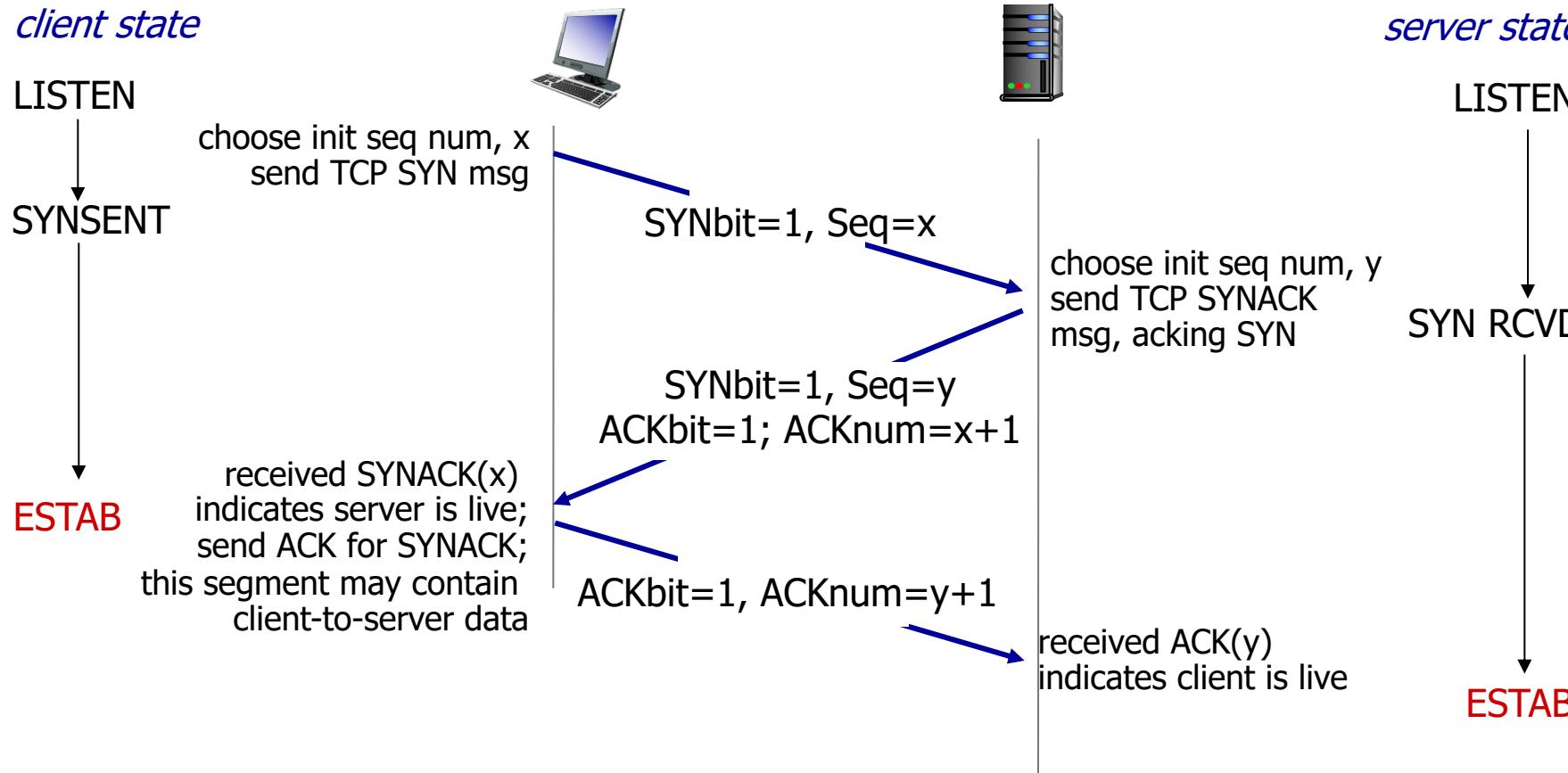
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



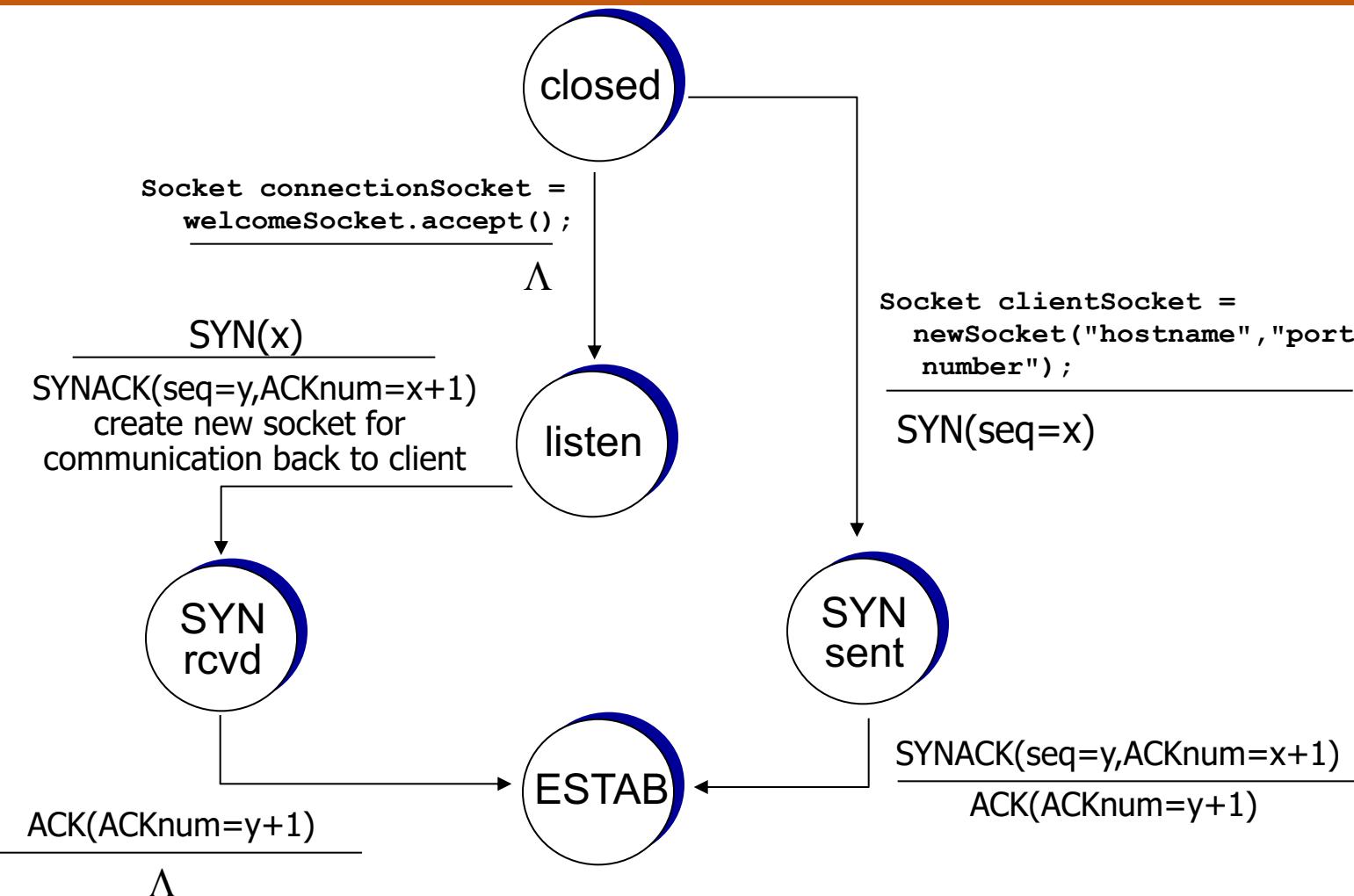
```
Socket clientSocket =  
    newSocket("hostname", "port  
    number");
```



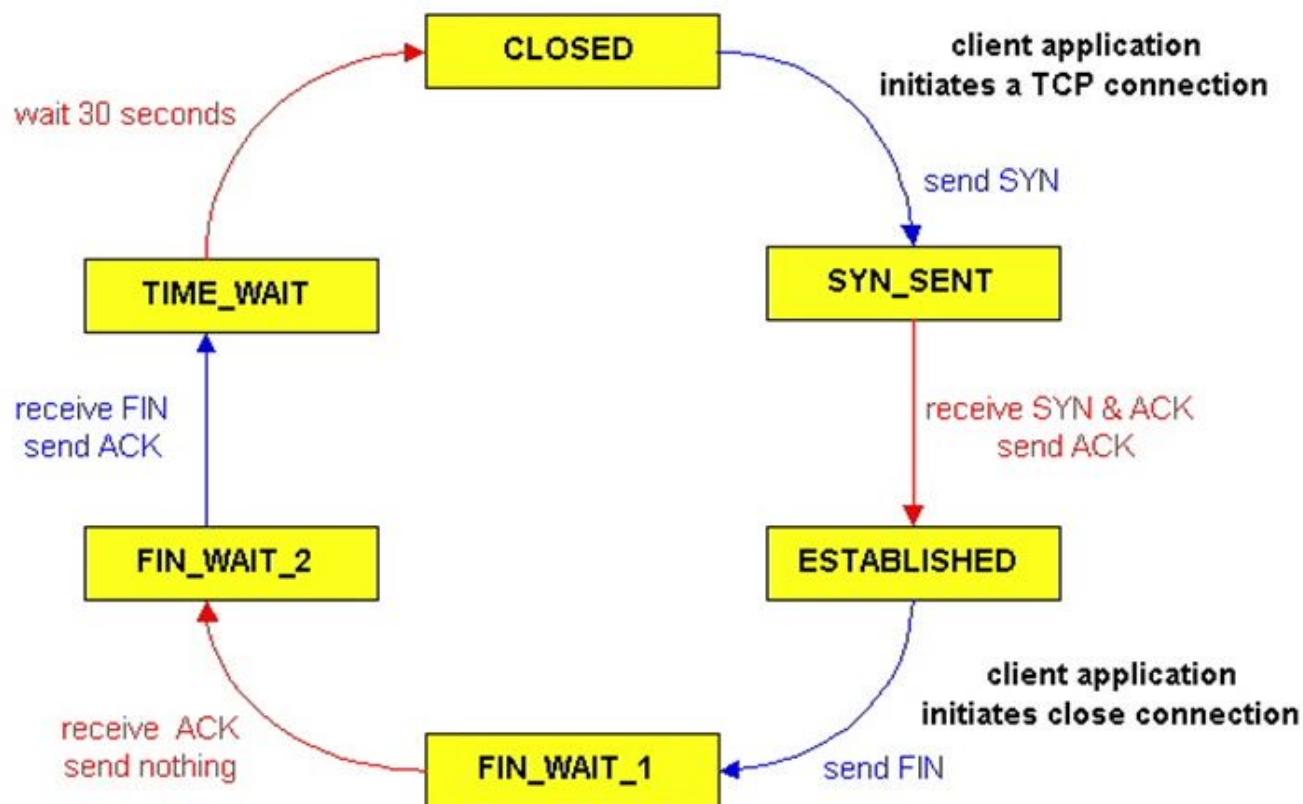
```
Socket connectionSocket =  
    welcomeSocket.accept();
```



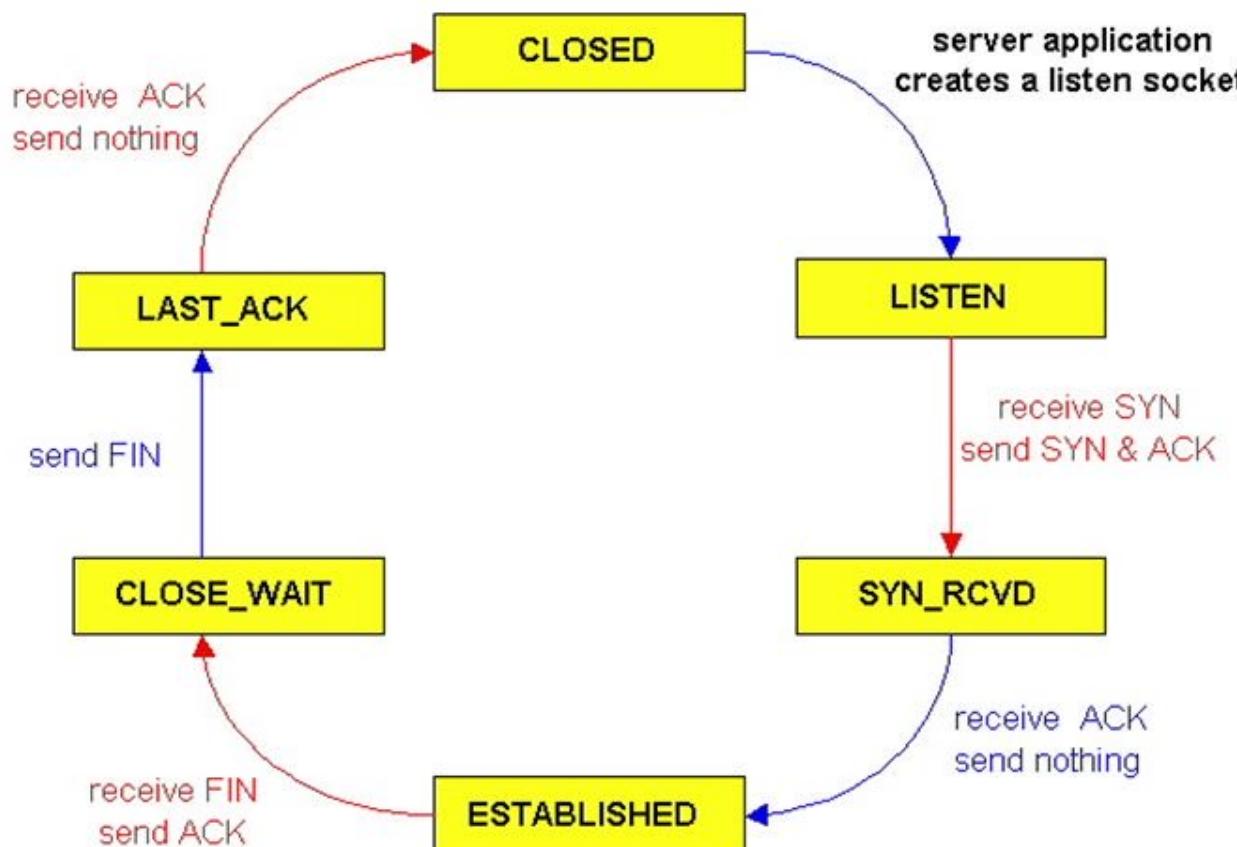
TCP 3-way handshake: FSM



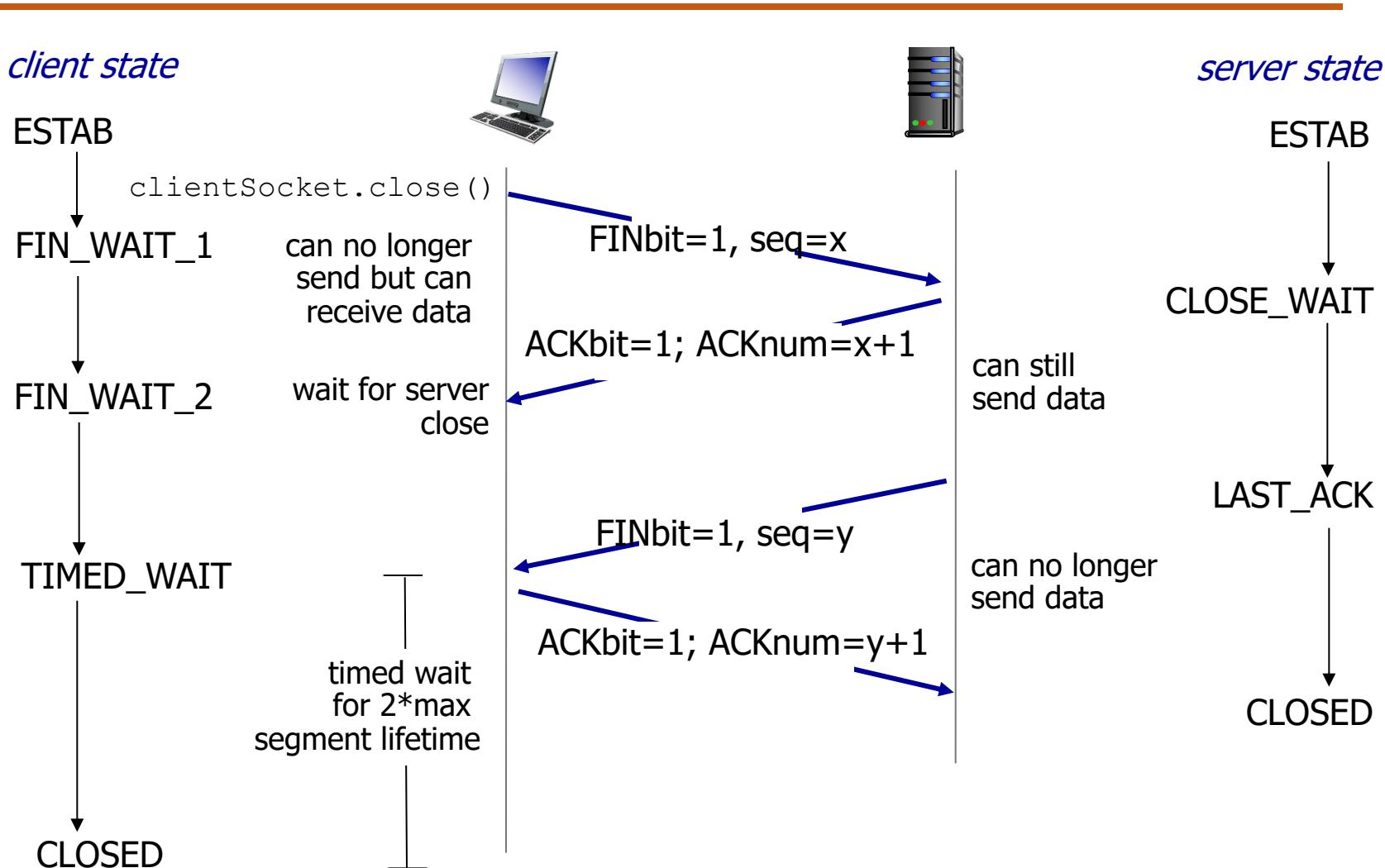
A Typical Sequence of States Visited By A Client TCP



A Typical Sequence of States Visited By Server-Side TCP



- 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



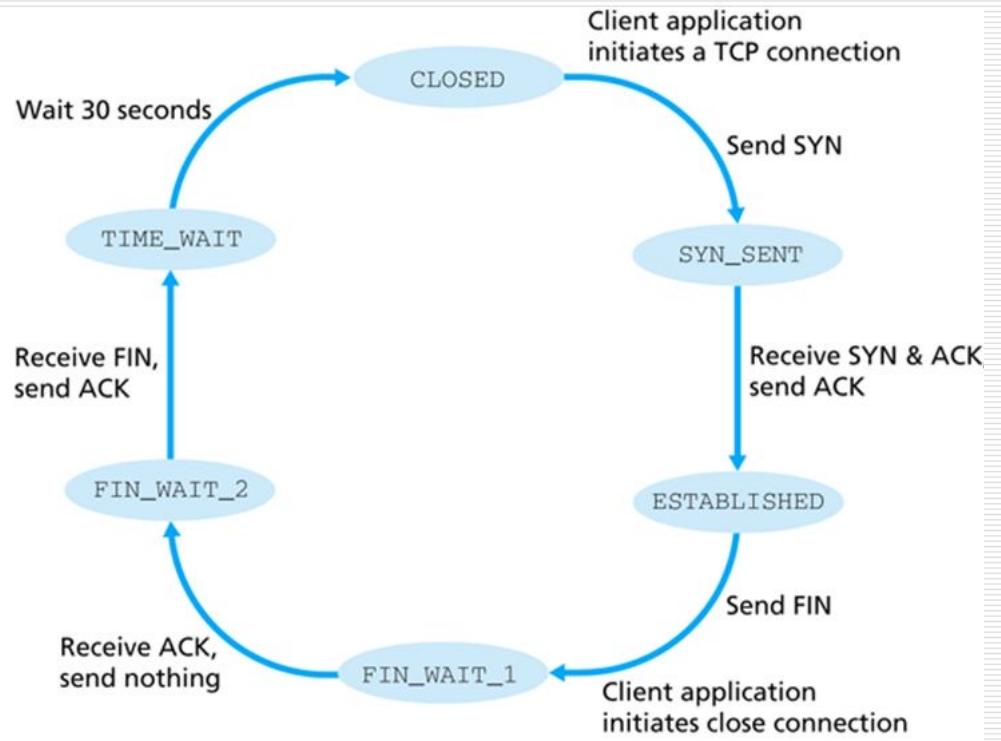


Figure 3.41 ♦ A typical sequence of TCP states visited by a client TCI

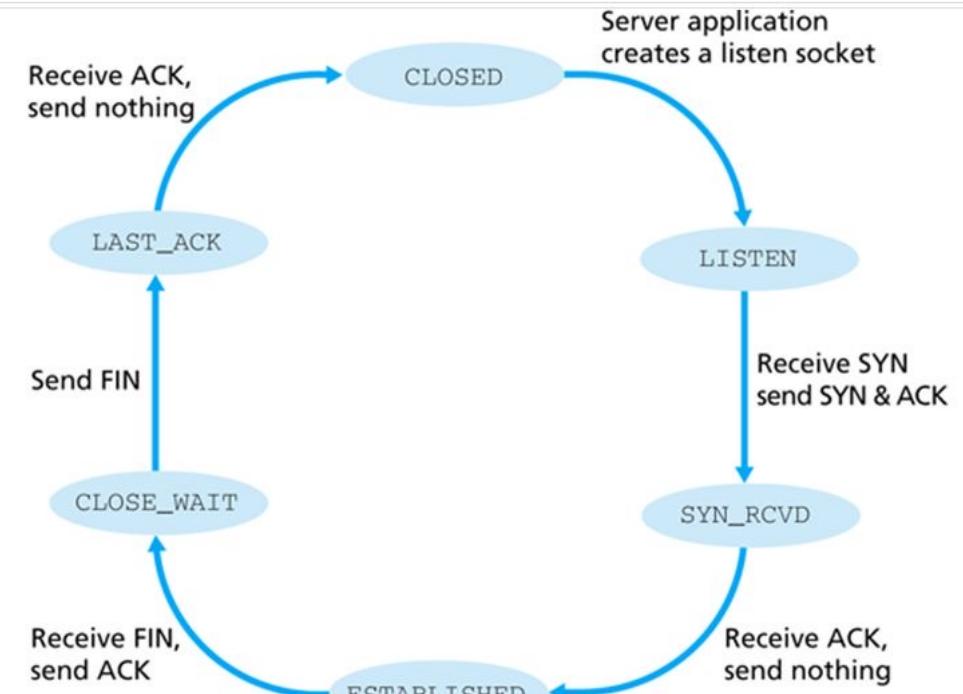


Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

SYN Flooding Attack

- Security problem
- A malicious attacker sends a large number of SYN segments to the server, pretending that each of them is coming from a different client by faking source IP addresses
- Server starts allocating necessary resources like buffers for each request
- Server also sends SYN+ACK segments to the fake clients, which are lost.
- Server eventually runs out of resources and may crash
- Server will deny any further requests from even valid clients
- This is called denial-of service attack

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

COMPUTER NETWORKS

Principles of Congestion Control

Ashwini M Joshi

Department of Computer Science & Engineering

In this segment

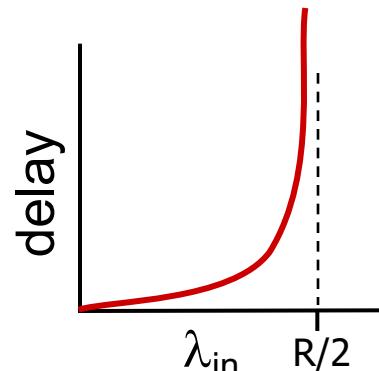
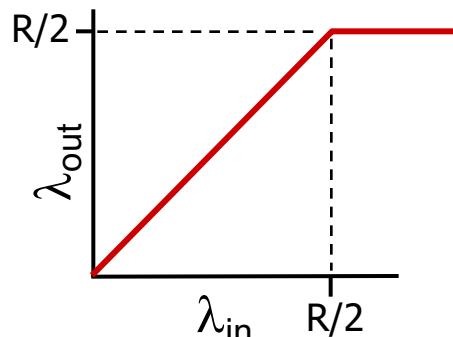
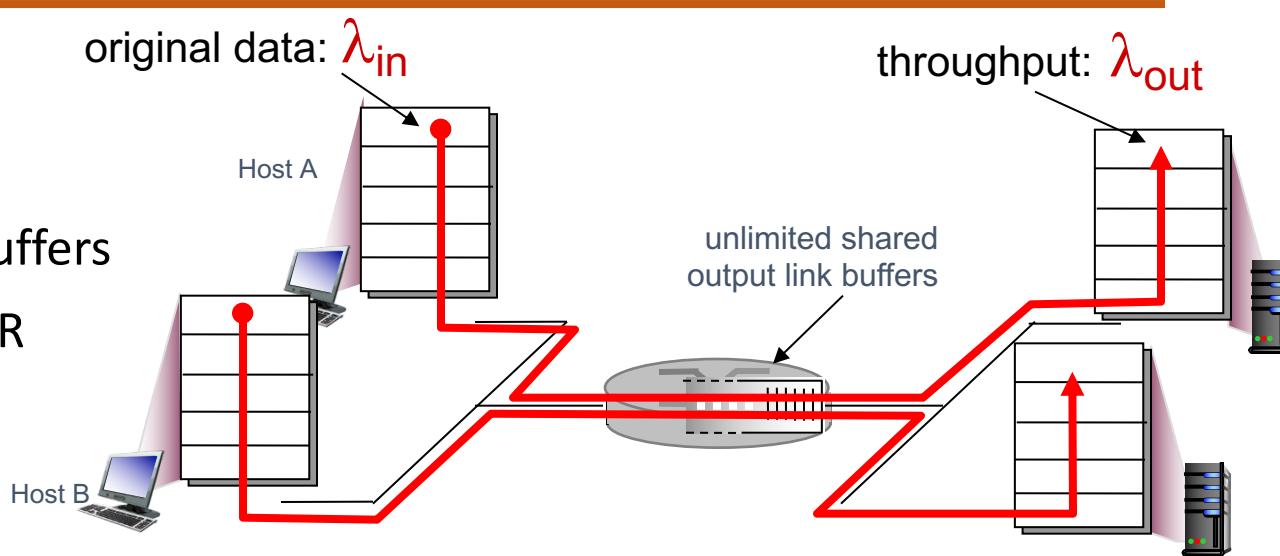
- Principles of congestion control
- Causes/costs of congestion: scenario 1
- Causes/costs of congestion: scenario 2
- Causes/costs of congestion: scenario 3

congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

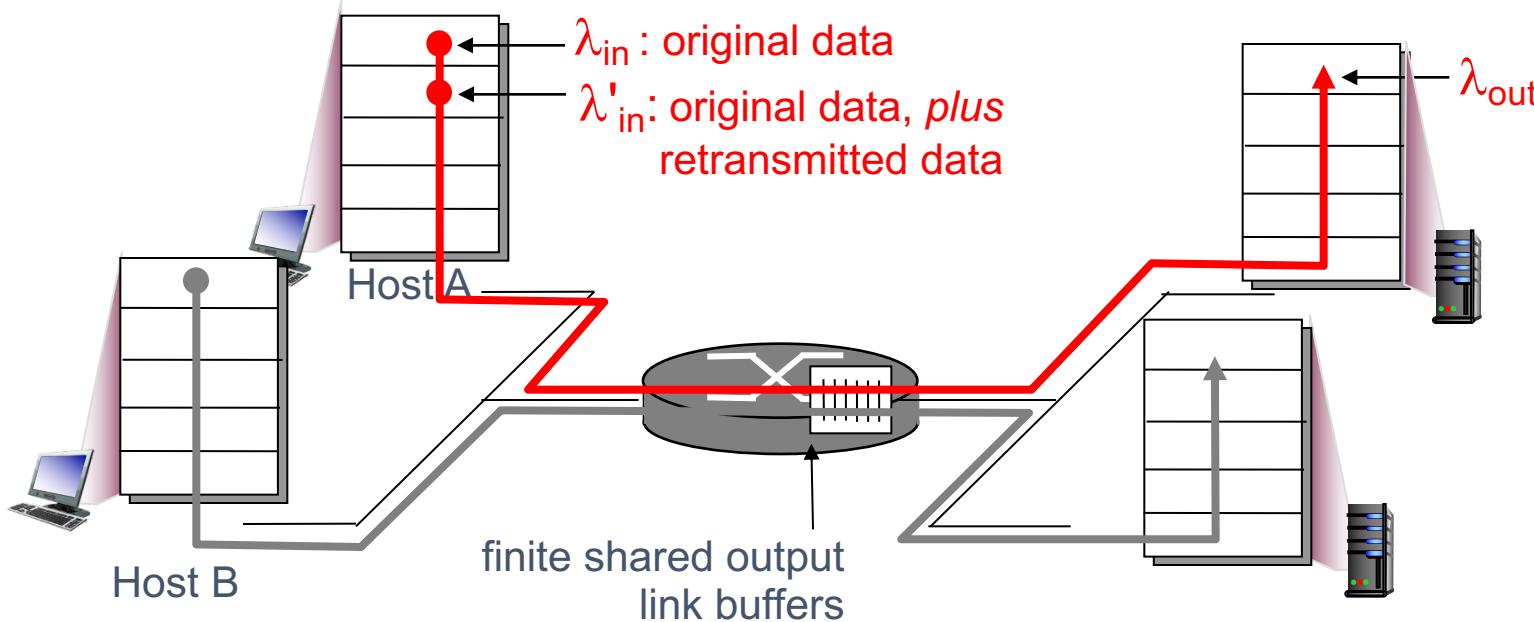
- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



- maximum per-connection throughput: $R/2$
- ❖ large delays as arrival rate, λ_{in} , approaches capacity

Causes/costs of congestion: scenario 2

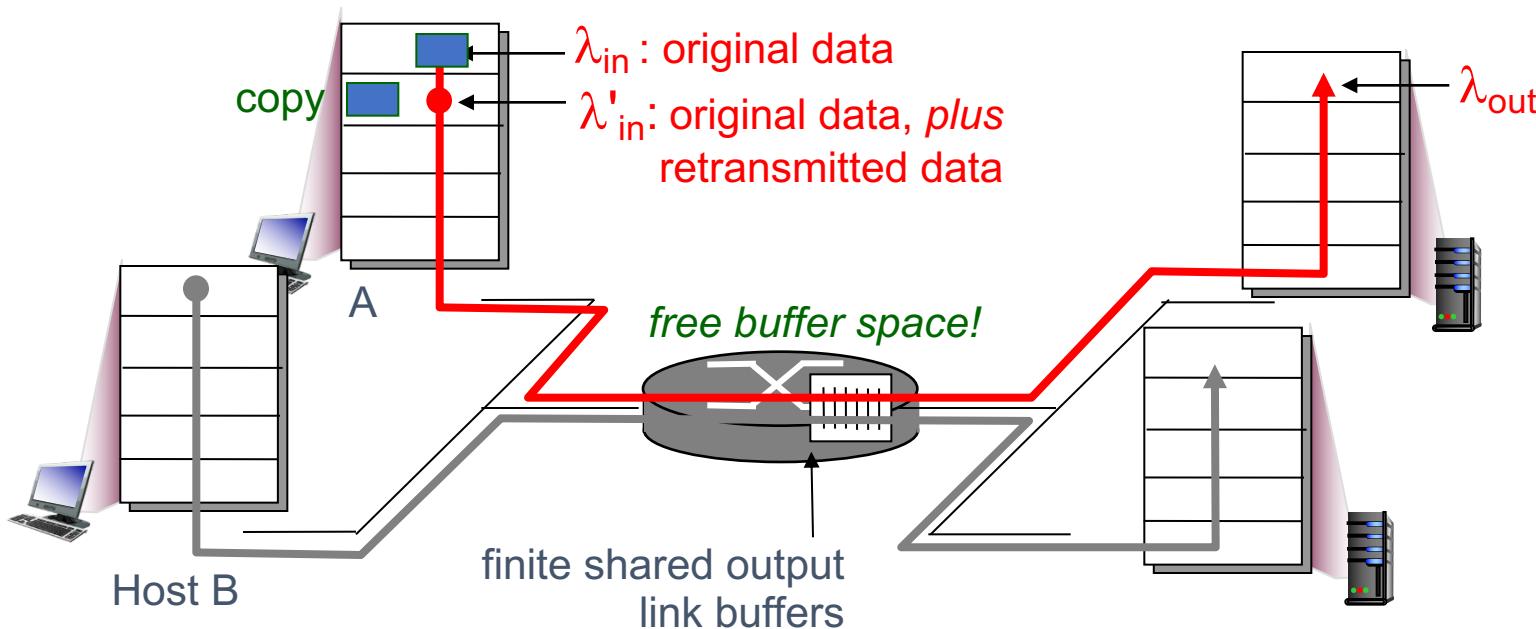
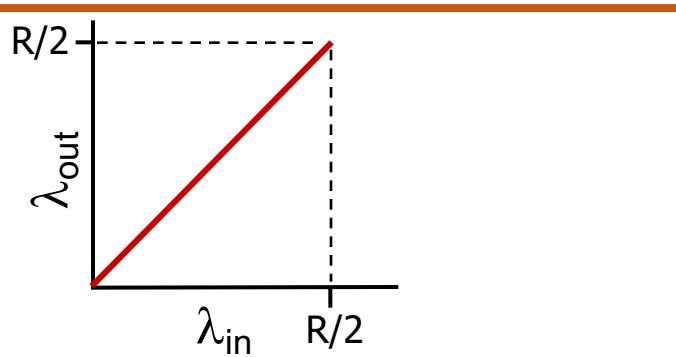
- one router, *finite* buffers
- sender retransmission of timed-out packet
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

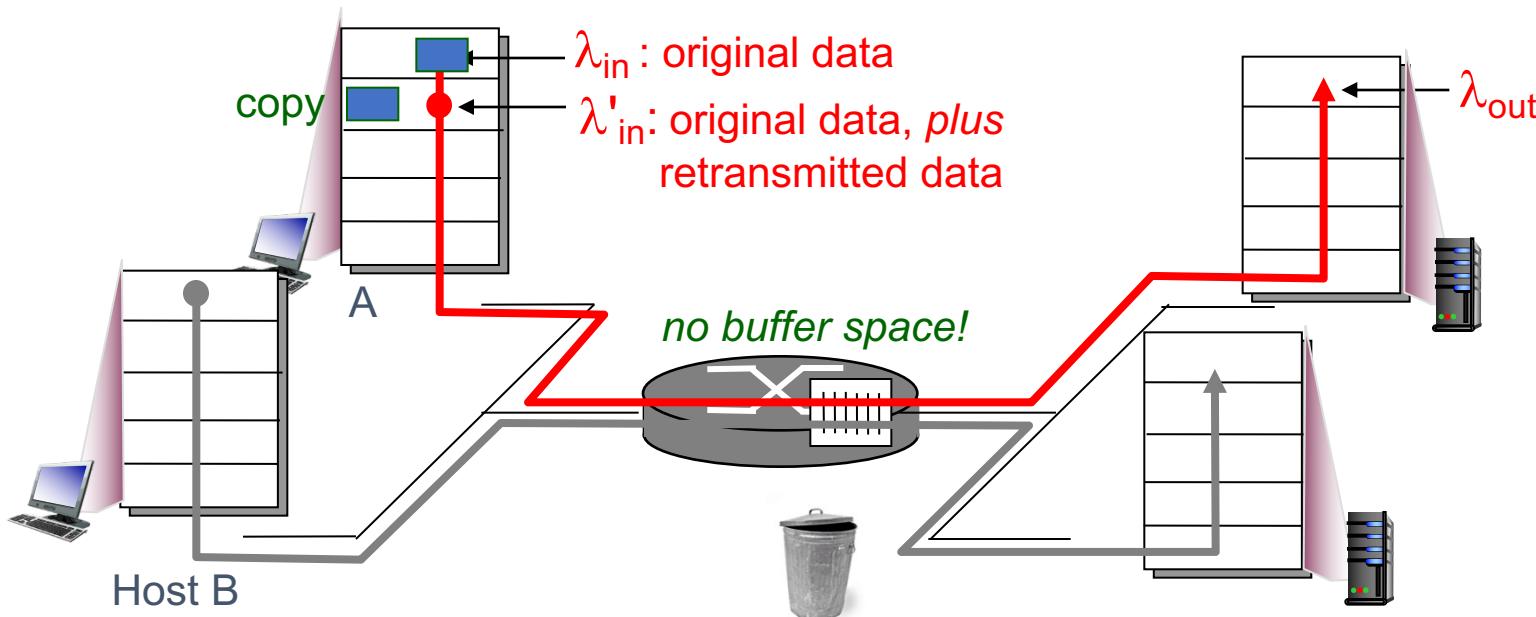


Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

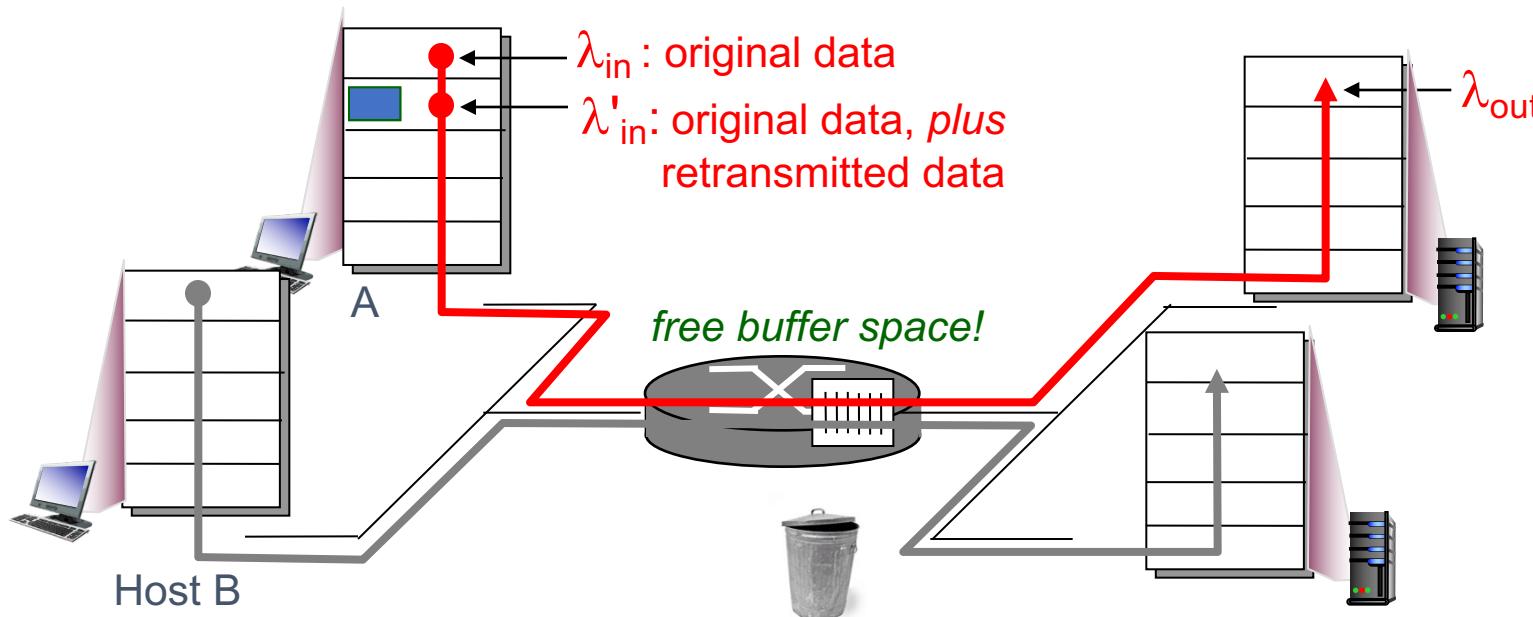
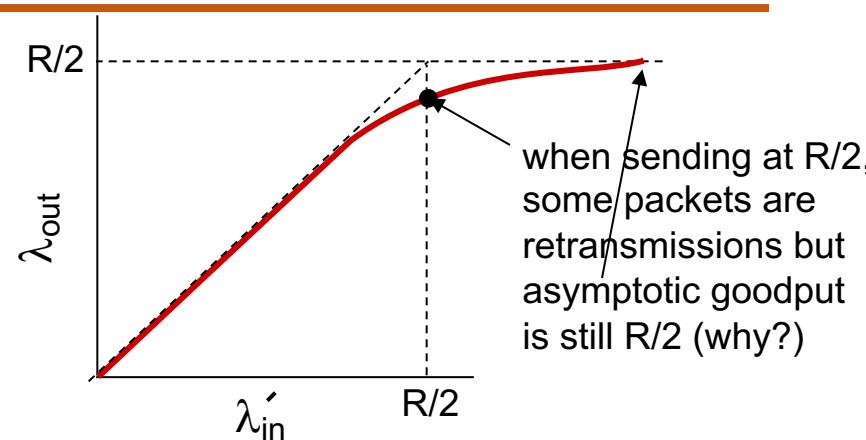


Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost,
dropped at router due to
full buffers

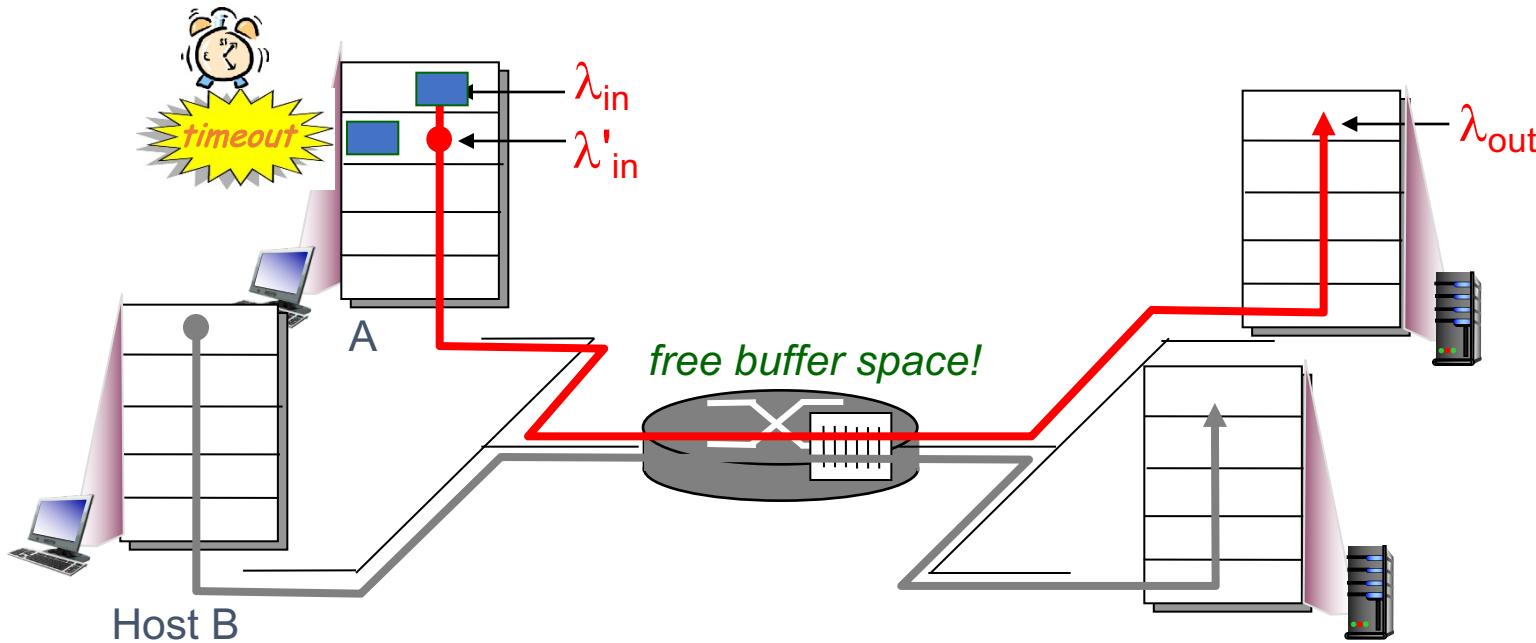
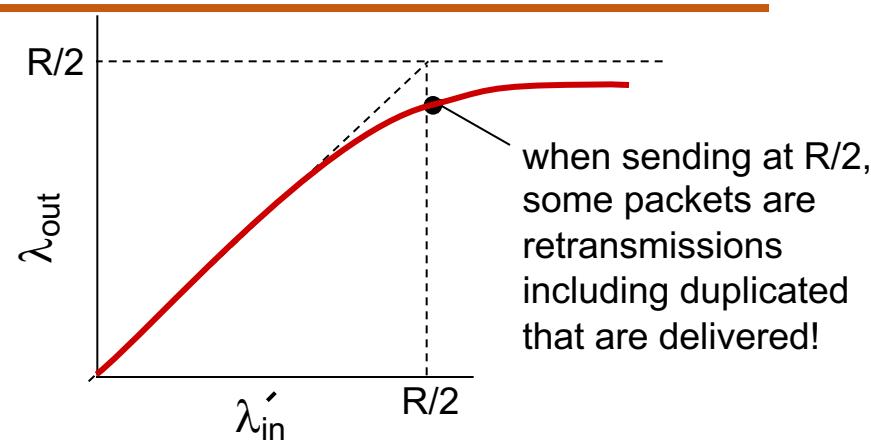
- sender only resends if packet *known* to be lost



Causes/costs of congestion: scenario 2

Realistic: duplicates

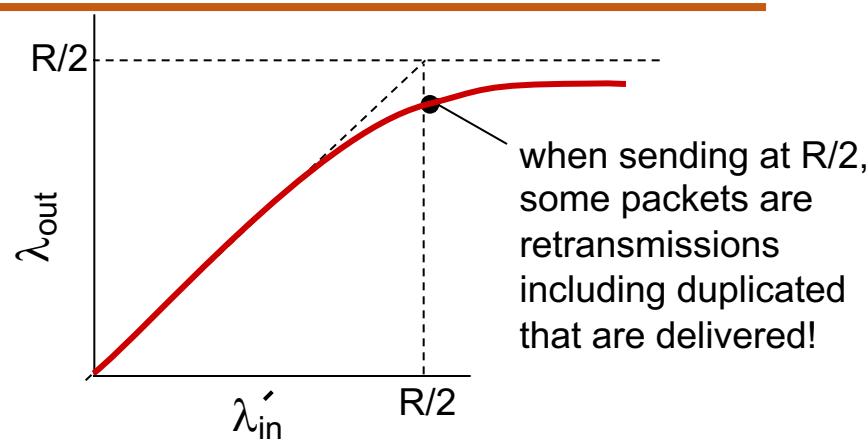
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending **two** copies, both of which are delivered



Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending **two** copies, both of which are delivered



“costs” of congestion:

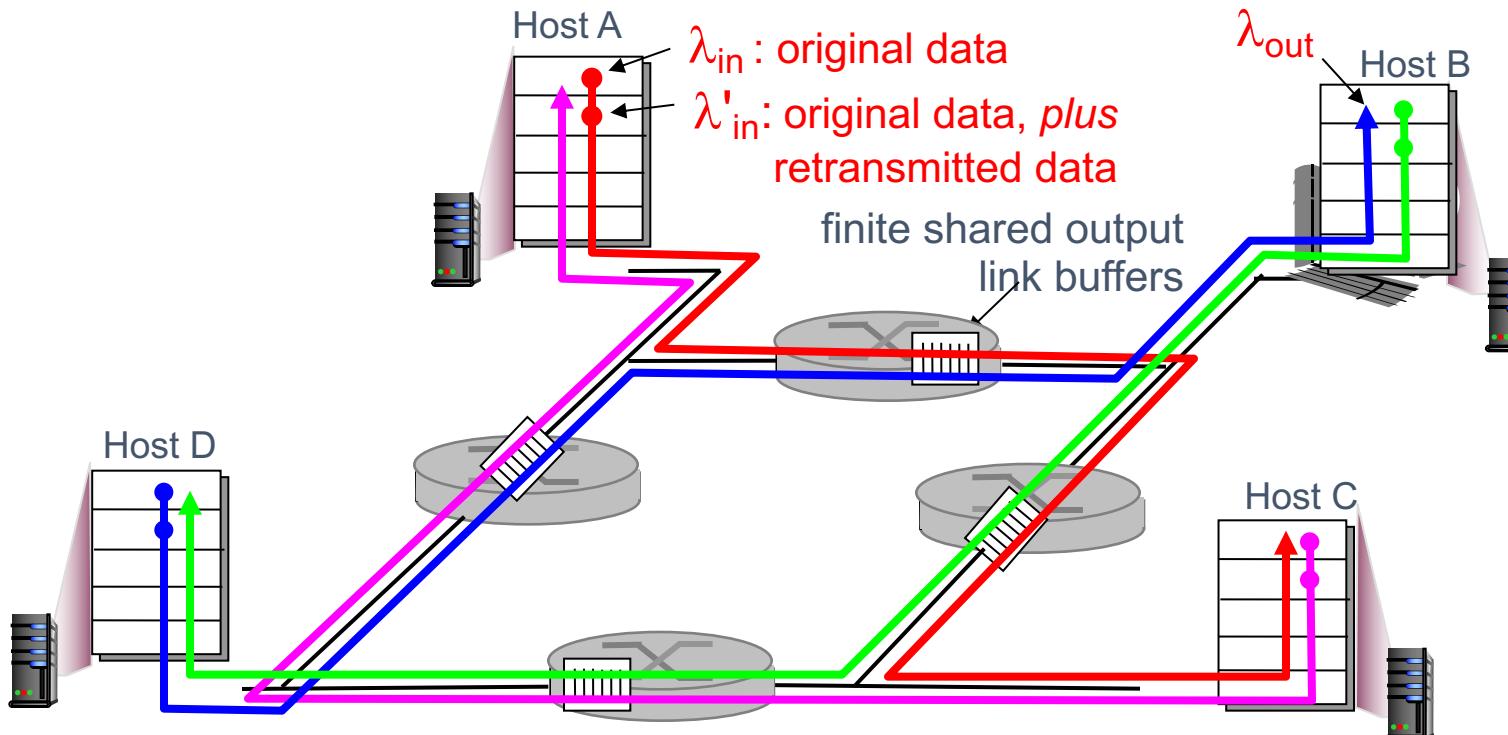
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
 - decreasing goodput

Causes/costs of congestion: scenario 3

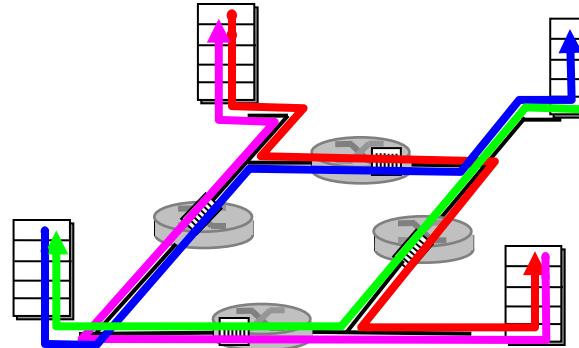
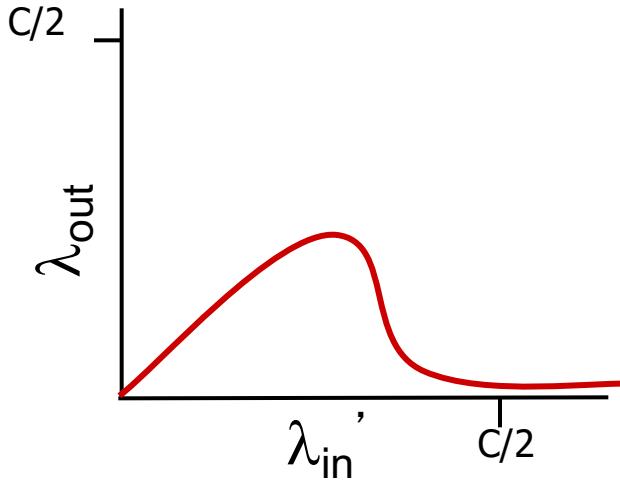
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?

A: as red λ_{in} increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow 0$



Causes/costs of congestion: scenario 3



another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet was wasted!

COMPUTER NETWORKS

Transport Layer

Ashwini M Joshi

Department of Computer Science & Engineering

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**

COMPUTER NETWORKS

TCP Congestion Control

Ashwini M Joshi

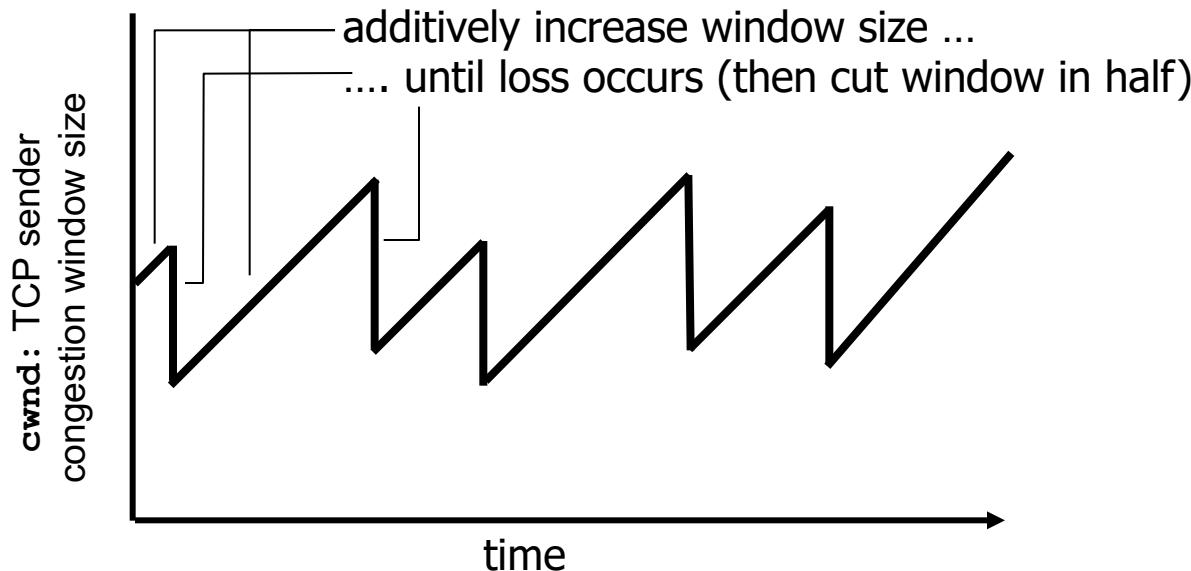
Department of Computer Science & Engineering

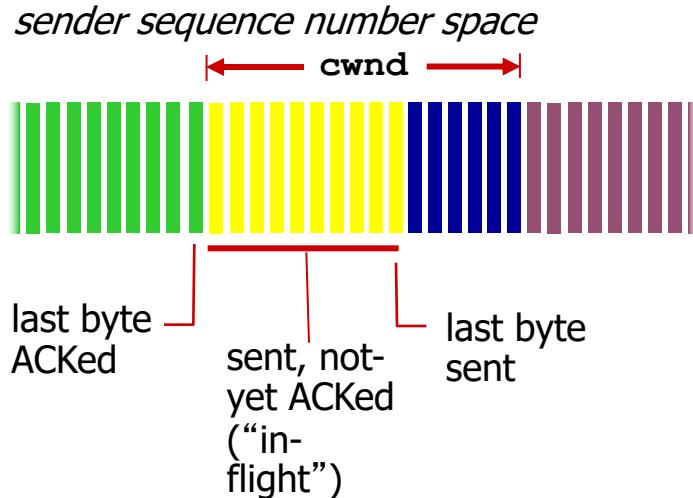
In this segment

- TCP congestion control: additive increase multiplicative decrease
- TCP Congestion Control: details
- TCP Slow Start
- TCP: detecting, reacting to loss
- TCP: switching from slow start to CA
- Summary: TCP Congestion Control
- TCP throughput
- TCP Futures: TCP over “long, fat pipes”
- TCP Fairness
- Why is TCP fair?
- Explicit Congestion Notification (ECN)

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
 - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
 - *multiplicative decrease:* cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth





- sender limits transmission:

$$\frac{\text{LastByteSent} - \text{LastByteAcked}}{\text{cwnd}} \leq \text{cwnd}$$

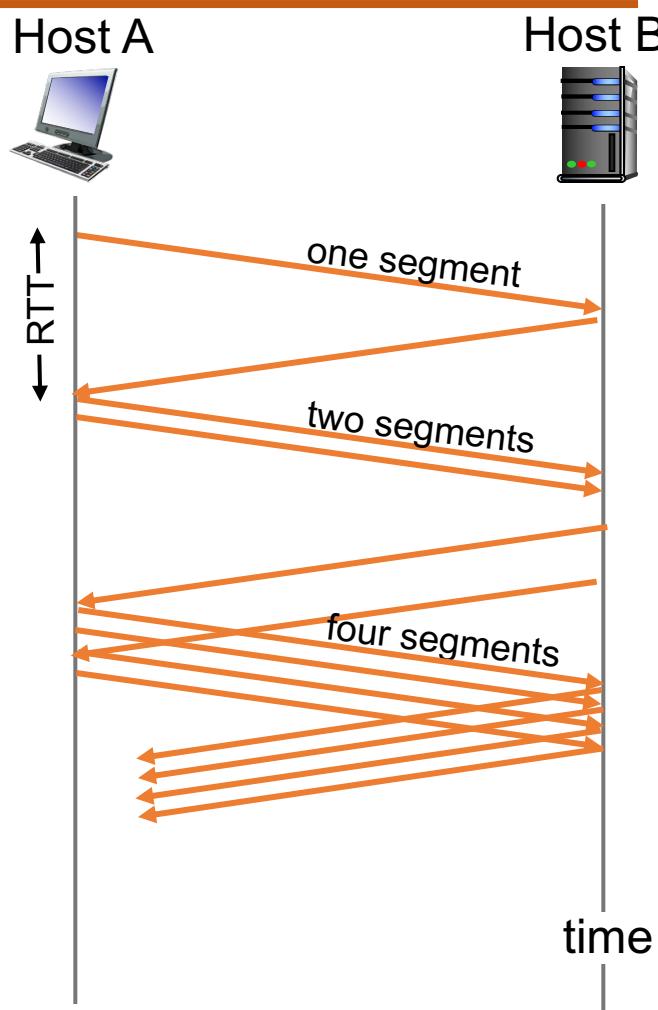
- **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

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

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

- 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



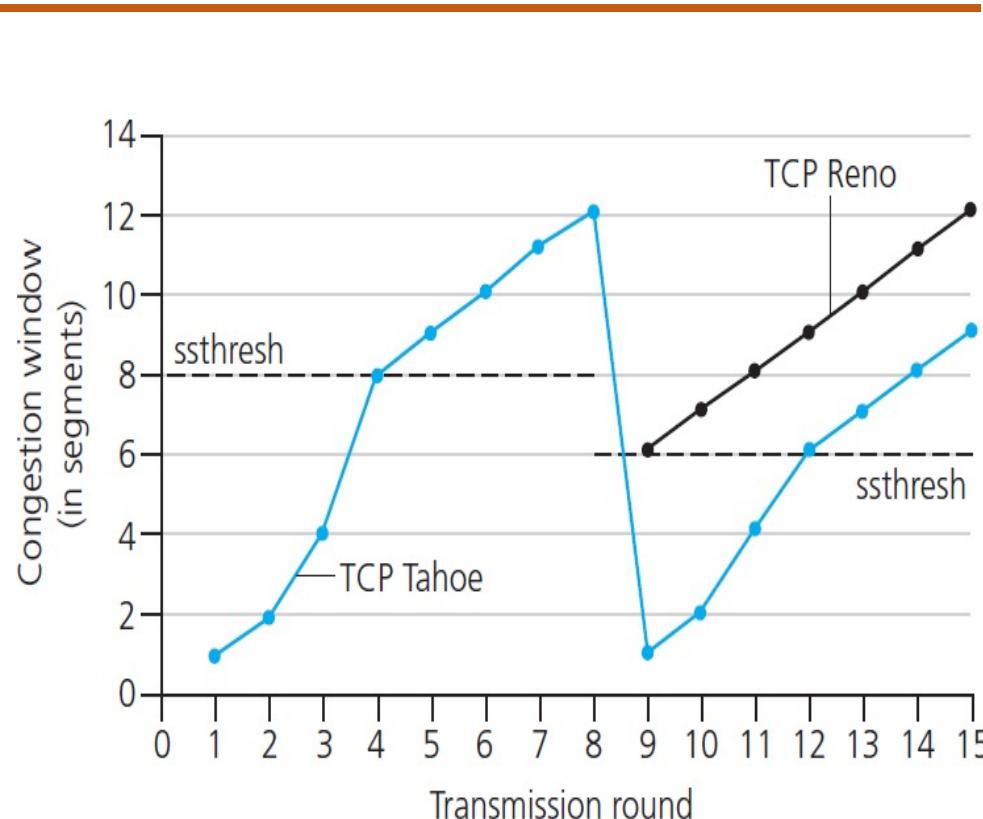
- loss indicated by timeout:
 - **cwnd** set to 1 MSS;
 - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: **TCP RENO**
 - dup ACKs indicate network capable of delivering some segments
 - **cwnd** is cut in half window then grows linearly
- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

Q: when should the exponential increase switch to linear?

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

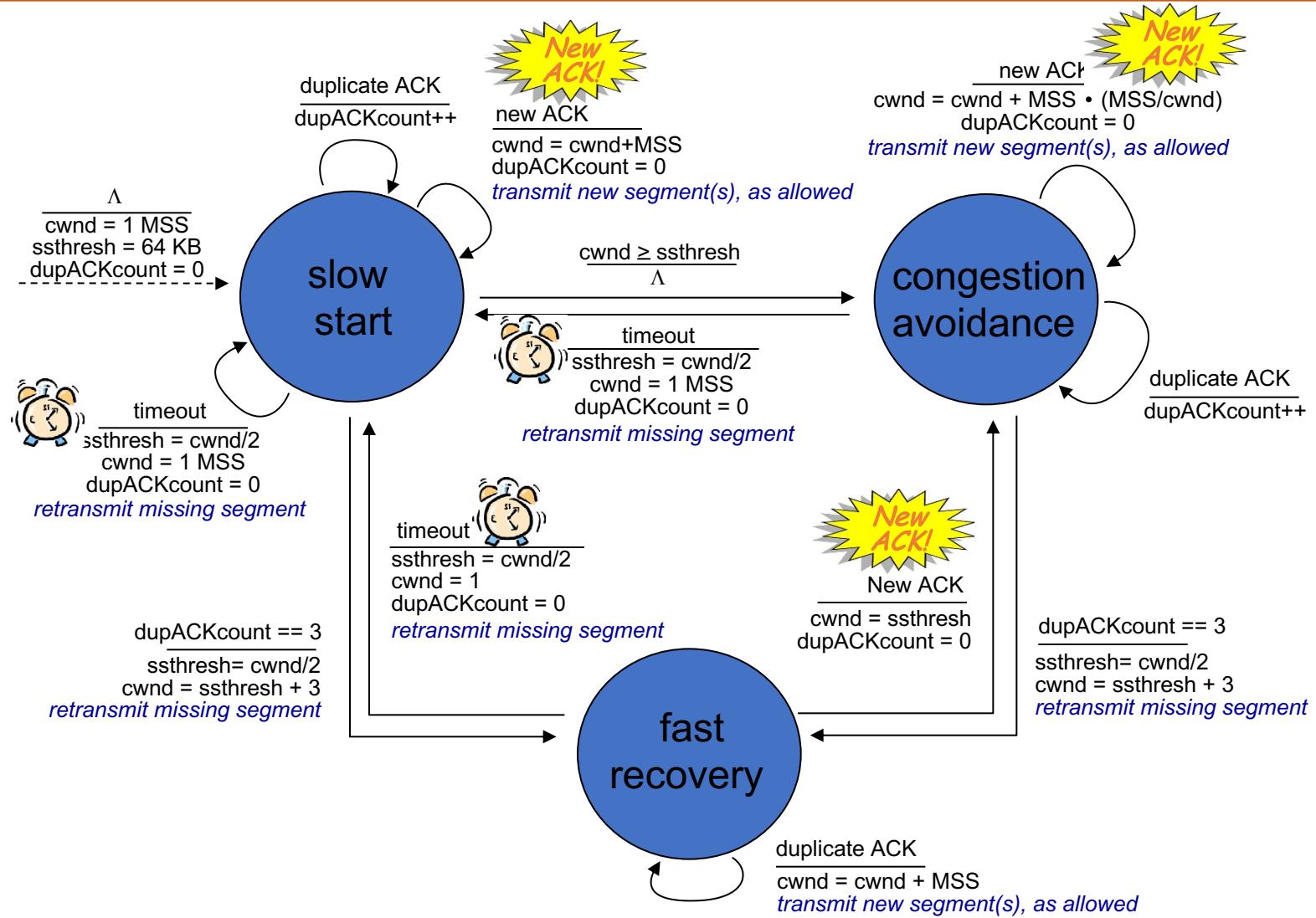
Implementation:

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



COMPUTER NETWORKS

Summary: TCP Congestion Control



Exercise Problem

In GB3 if every 5th packet that is transmitted is lost and if we have to send 10 packets. Then how many transitions are required?

In GB3 if every 5th packet that is transmitted is lost and if we have to send 10 packets. Then how many transitions are required?

1 2 3 4 5 6 7 | 8 9 10 ^ \$ (packet no. 5 lost)

1 2 3 4 5 6 7 5 6 7 8 9 | 10

* ^ \$

1 2 3 4 5 6 7 5 6 7 8 9 7 8 9 10

* ^ \$

1 2 3 4 5 6 7 5 6 7 8 9 7 8 9 10 9 10

(count starts from * till ^)

(from ^ to \$ retransmission is done)

From the Last packet is window size to lost pocket we resend the entire window.

Total no. of transmissions = 18



THANK YOU

Ashwini M Joshi

Department of Computer Science & Engineering

ashwinimjoshi@pes.edu