

# CS 305: Computer Networks

## Fall 2023

### **Lecture 8: Transport Layer**

**Ming Tang**

Department of Computer Science and Engineering  
Southern University of Science and Technology (SUSTech)

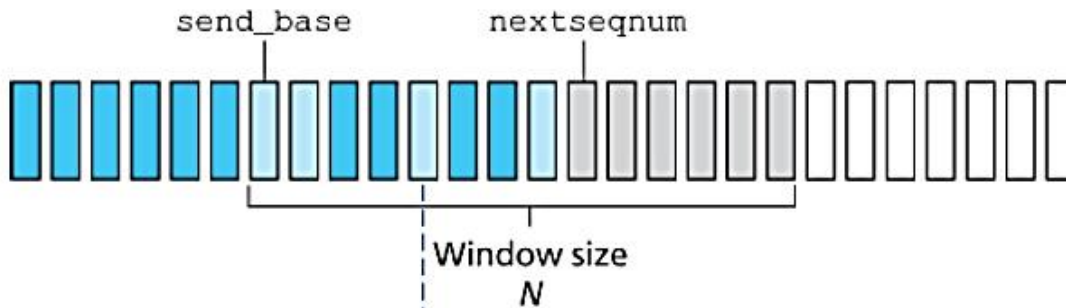
# Pipelined Protocols

- ❖ Go-Back-N
  - Timer for the oldest unACKed packet
  - Cumulative ACK
  - Retransmit all packets in the window
- ❖ Selective repeat
  - Timer for each packet in window
  - Individual ACK for each correctly received packets
  - Retransmit only those packets that might be lost or corrupted

# Selective repeat





- ❖ receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
  - Receiver needs to keep track of the **out-of-packets**
- ❖ sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

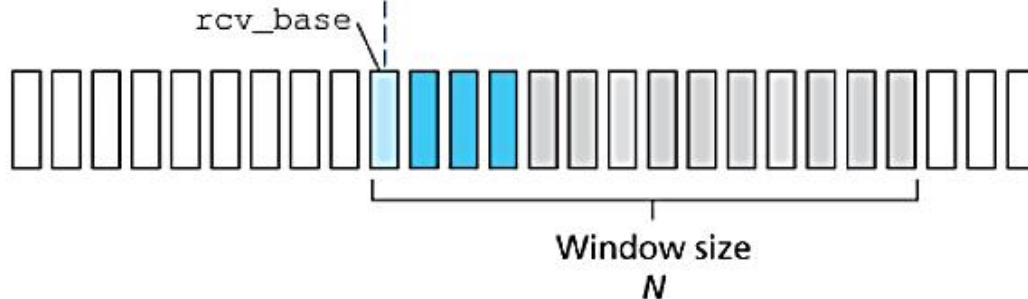
# Selective repeat: sender, receiver windows



a. Sender view of sequence numbers





Key:

 Already ACK'd	 Usable, not yet sent
 Sent, not yet ACK'd	 Not usable



b. Receiver view of sequence numbers

Key:

 Out of order (buffered) but already ACK'd	 Acceptable (within window)
 Expected, not yet received	 Not usable

# Selective repeat

## sender

### data from above:

- ❖ if next available seq # in window, send pkt

### timeout( $n$ ):

- ❖ resend pkt  $n$ , restart timer

### ACK( $n$ ) in $[\text{sendbase}, \text{sendbase}+N]$ :

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

## receiver

### pkt $n$ in $[\text{rcvbase}, \text{rcvbase}+N-1]$

- ❖ 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 $[\text{rcvbase}-N, \text{rcvbase}-1]$

- ❖ ACK( $n$ )

### otherwise:

- ❖ ignore

# Selective repeat

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



*pkt 2 timeout*

send pkt2

record ack4 arrived

record ack5 arrived

receiver

have buffer,  
individual ACK

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,  
send ack3

receive pkt4, buffer,  
send ack4

receive pkt5, buffer,  
send ack5

rcv pkt2; deliver pkt2,  
pkt3, pkt4, pkt5; send ack2

*X loss*

Only retransmit the  
unacked pkt (SR)

*what happens when ack2 arrives?*

# Selective repeat: dilemma

Example:

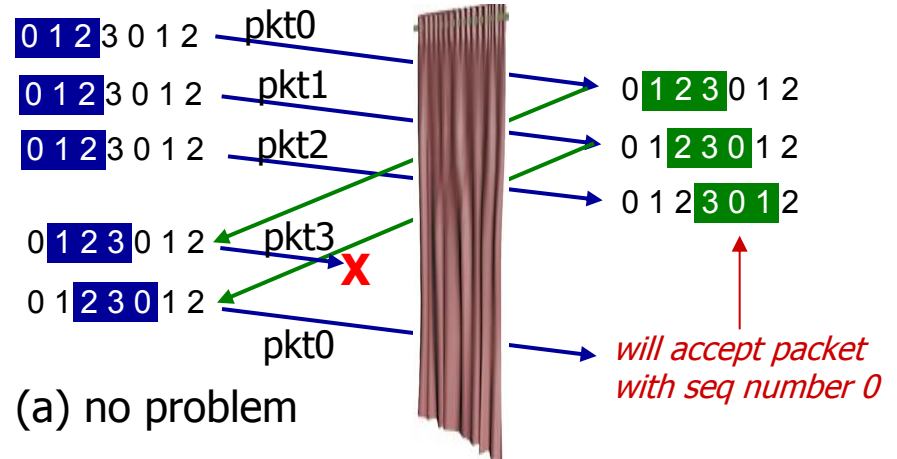
- ❖ seq #'s: 0, 1, 2, 3
- ❖ window size=3
- ❖ receiver sees no difference in two scenarios!
- ❖ duplicate data accepted as new in (b)

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

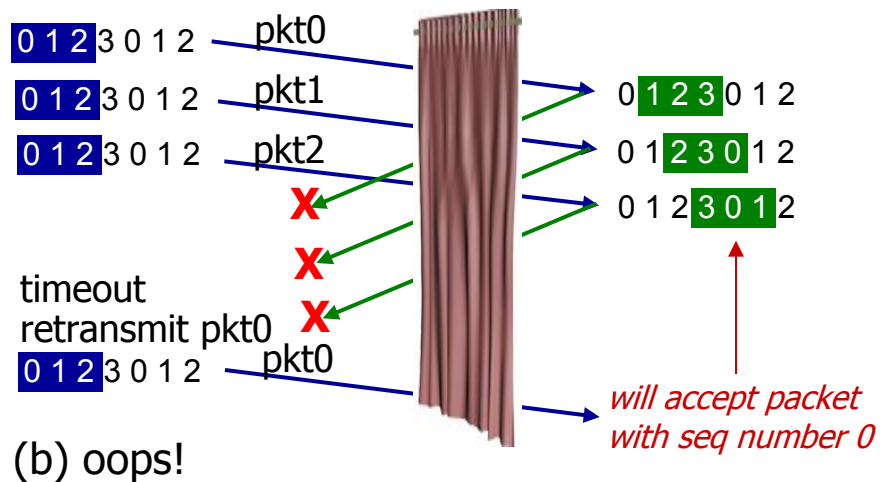
The window size must be less than or equal to half the size of the sequence number space for SR protocols.

sender window  
(after receipt)

receiver window  
(after receipt)



*receiver can't see sender side.  
receiver behavior identical in both cases!  
something's (very) wrong!*



# GBN and SR comparison

## 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  $N$  unack'ed 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



# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 **connection-oriented transport: TCP**

- segment structure, RTT measurement
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

---

## ❖ point-to-point:

- one sender, one receiver
- No buffers or variables are allocated to network elements between hosts

## ❖ reliable, in-order byte stream:

- no “message boundaries”
- Seq # and Ack # are in unit of byte, rather than pkt

## ❖ 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:

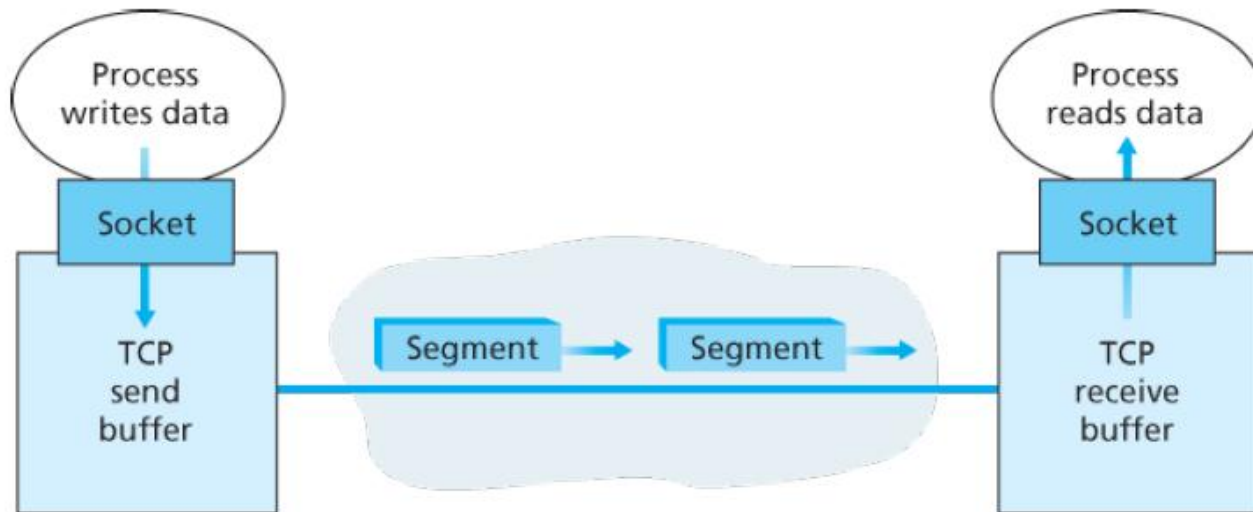
- handshaking (exchange of control msgs) initiates sender and receiver state before data exchange

## ❖ flow controlled:

- sender will not overwhelm receiver

# TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

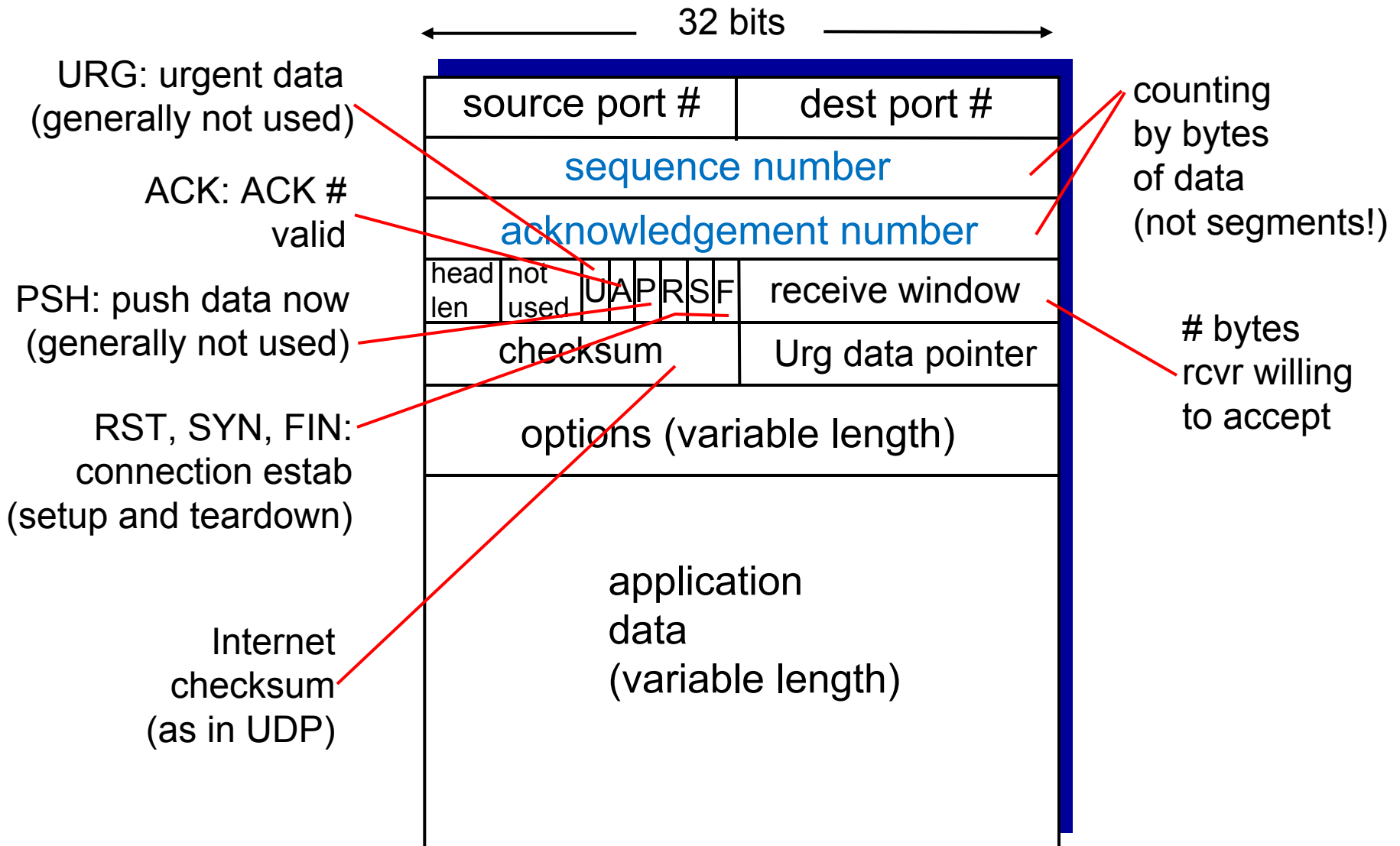


- TCP connection
- TCP grab chunks of data from the sender buffer
  - MSS: maximum segment size, typically 1460 bytes
  - MTU: maximum transmission unit (link-layer frame), typically 1500 bytes
    - Application data + TCP/IP header (typically 40 bytes)
- TCP receives a segment at the other end, place it in receiver buffer
- application reads the stream from the receive buffer

# TCP Reliable Data Transfer

- ❖ Segment structure
  - Segment format
  - Seq. number and ACKs
  - An example
- ❖ Round-trip time estimation
- ❖ Reliable data transfer
- ❖ Flow control
- ❖ Control management

# TCP segment structure



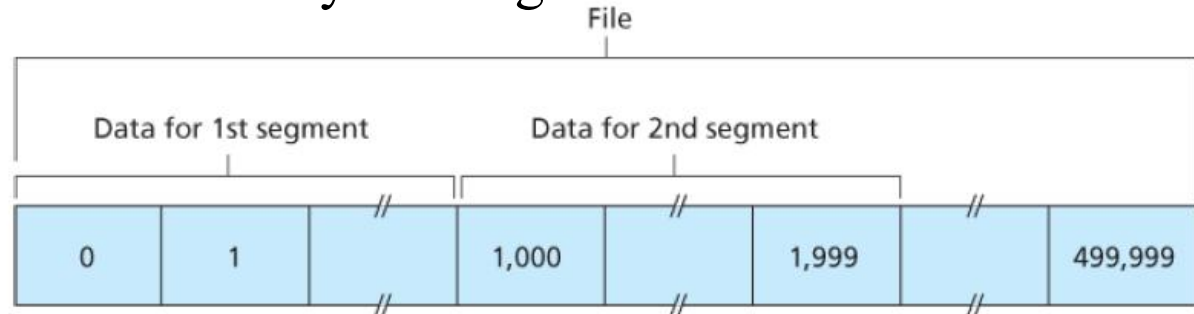
# TCP seq. numbers, ACKs

TCP views data as an unstructured, but ordered, stream of bytes.

- Sequence numbers are over the **stream** of transmitted bytes and *not* over the series of transmitted **segments**

## sequence numbers:

- byte stream “number” of first byte in segment’s data



## acknowledgements:

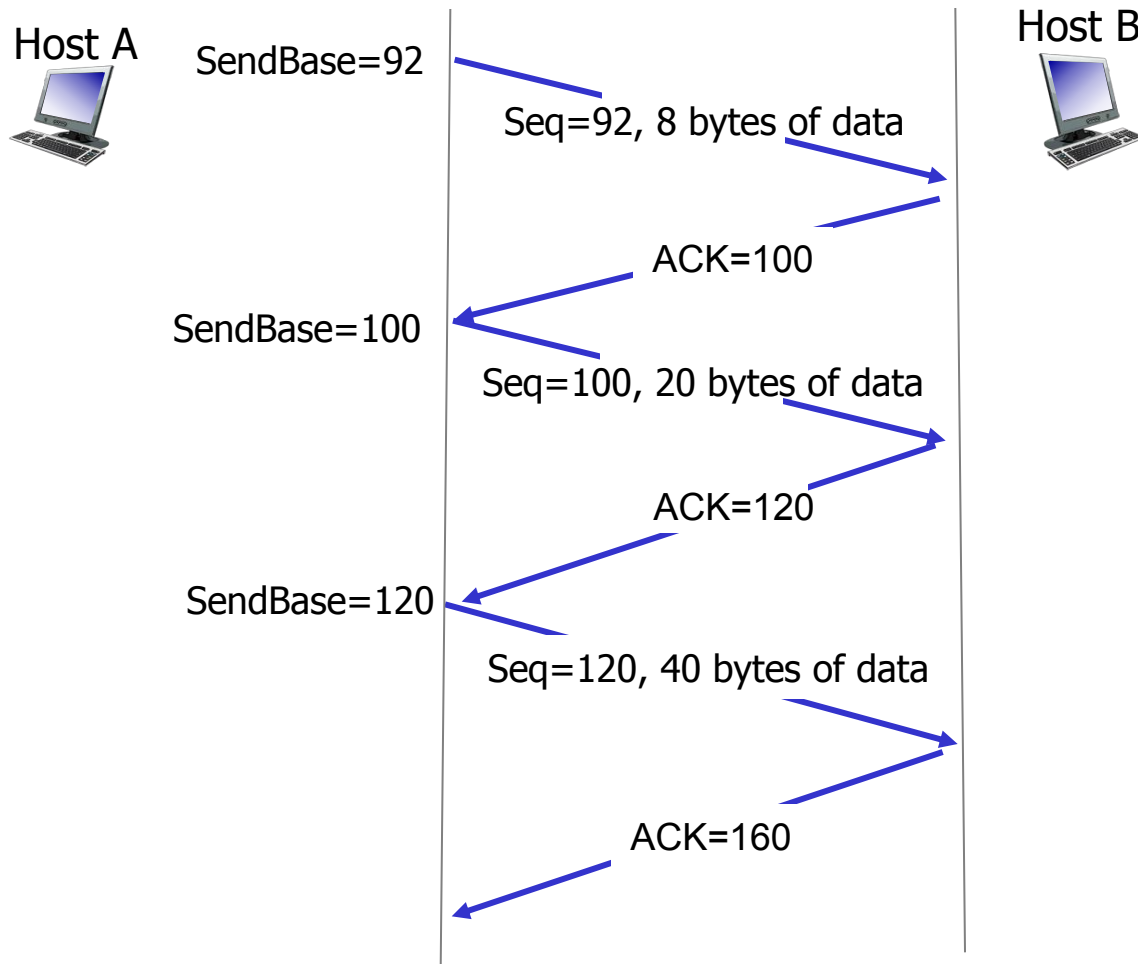
- seq # of next byte expected from other side
  - E.g., receiver has received bytes numbered 0 through 535 and 900 through 1000; then, acknowledgement number is 536.
- cumulative ACK

**Q:** how receiver handles out-of-order segments

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

Initial sequence  
number is randomly  
chosen

# TCP Example

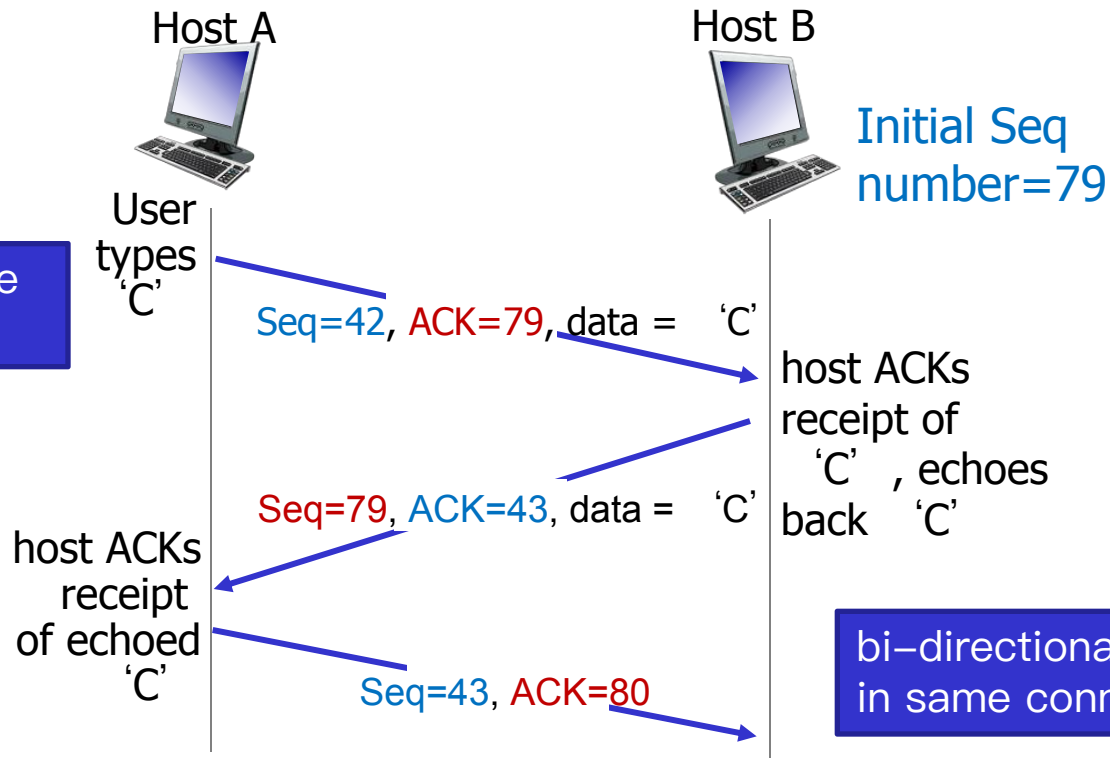


# Telnet Case Study

- User types a character at host A, and host A sends the character to host B
- Host B sends back a copy of the character
- Host A displays the character on user's screen

Initial Seq  
number=42

ACK # is the next byte  
expected from other





# TCP Reliable Data Transfer

- ❖ Segment structure
- ❖ Round-trip time estimation
- ❖ Reliable data transfer
- ❖ Flow control
- ❖ Control management

# 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

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- ❖ exponential weighted moving average
- ❖ influence of past sample decreases exponentially fast
- ❖ typical value:  $\alpha = 0.125$

## Example:

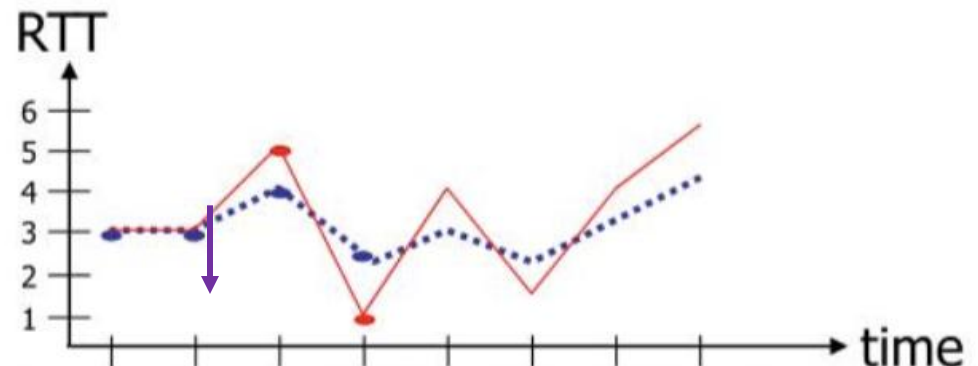
Suppose  $\alpha = 0.5$

EstimatedRTT = 3

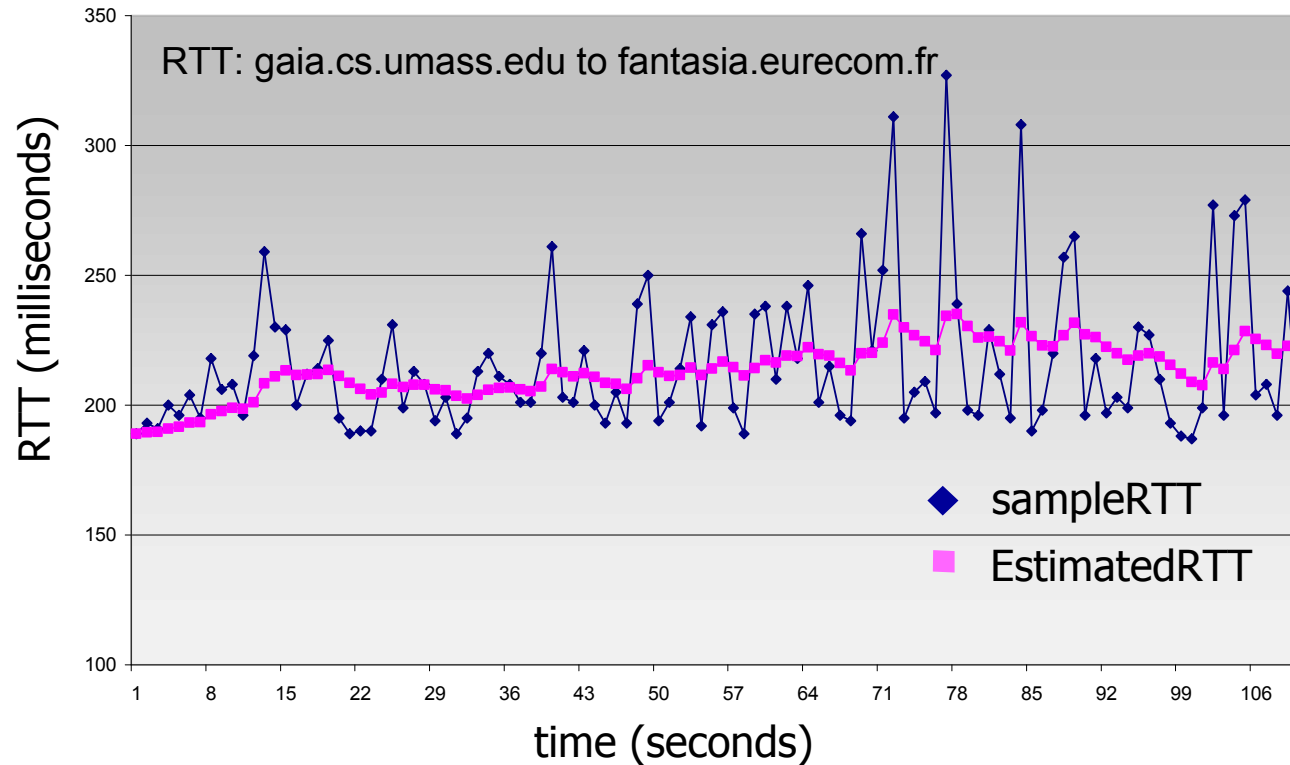
2) EstimatedRTT =  $.5 * 3 + .5 * 3 = 3$

3) EstimatedRTT =  $.5 * 3 + .5 * 5 = 4$

4) EstimatedRTT =  $.5 * 4 + .5 * 1 = 2.5$



# TCP round trip time, timeout



# TCP round trip time, timeout


**Variability** of the RTT: how much **SampleRTT** deviates from **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

**TCP timeout interval:** **EstimatedRTT** plus “safety margin”  
large variation in **EstimatedRTT** -> larger safety margin

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

 ↑  
estimated RTT ↑  
“safety margin”

# TCP Reliable Data Transfer

- ❖ Segment structure
- ❖ Round-trip time estimation
- ❖ **Reliable data transfer**
- ❖ Flow control
- ❖ Control management

# TCP reliable data transfer

- ❖ TCP creates rdt service on top of IP's unreliable service
  - pipelined segments: window size, SendBase
  - **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

# TCP sender events:

## *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 sender events:

```
NextSeqNum=InitialSeqNumber  
SendBase=InitialSeqNumber
```

```
loop (forever) {  
    switch(event)
```

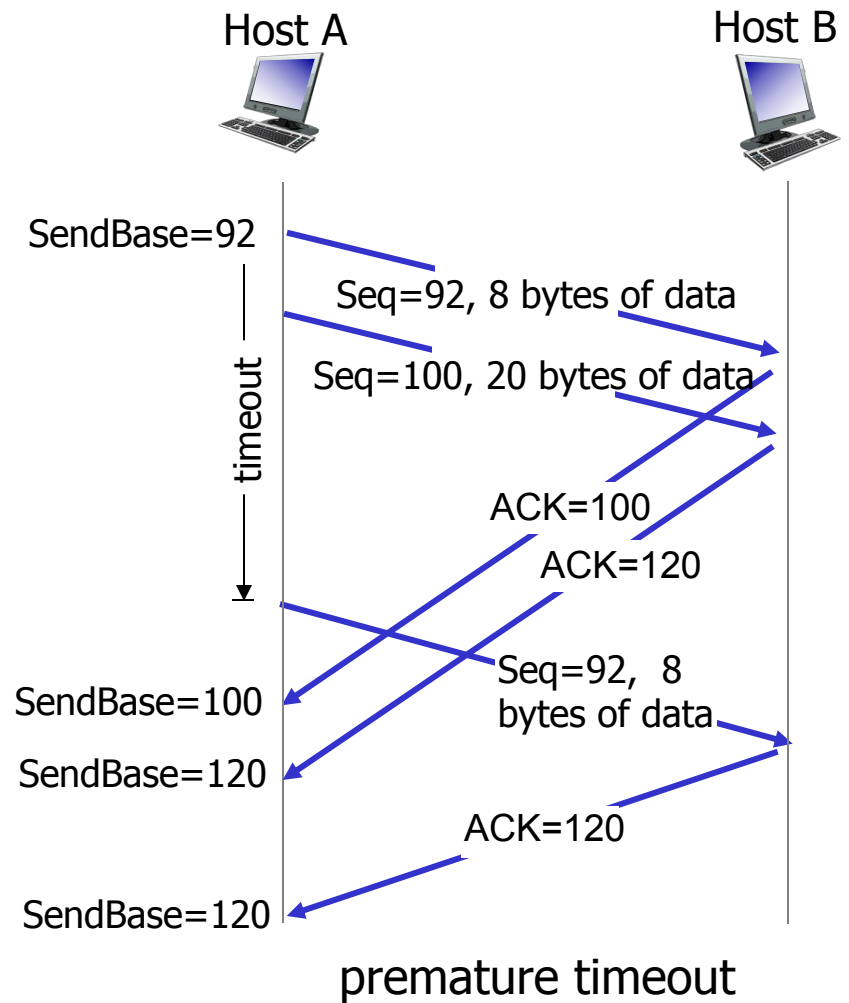
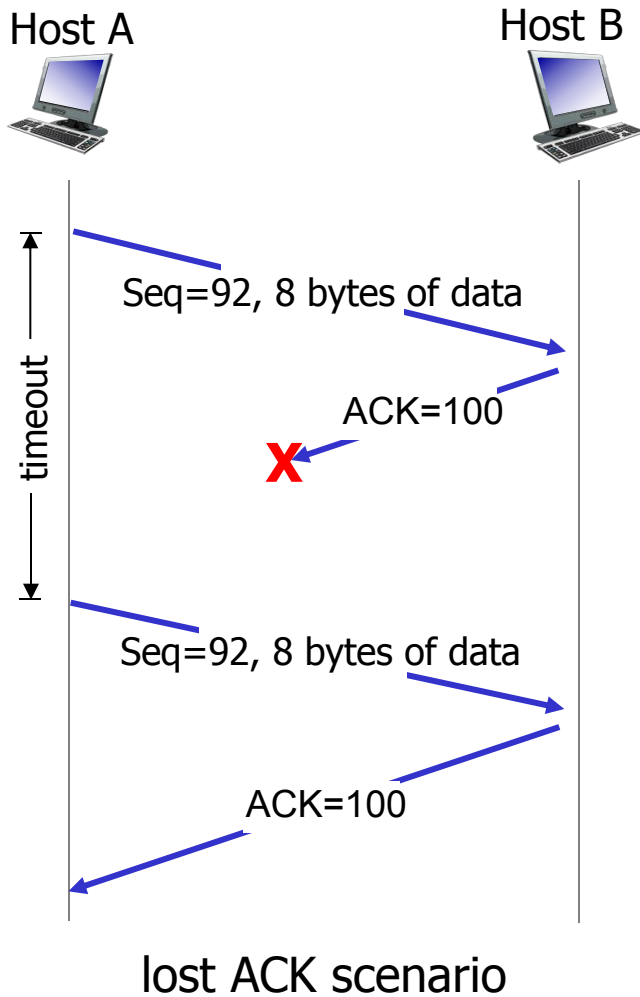
```
event: data received from application above  
    create TCP segment with sequence number NextSeqNum  
    if (timer currently not running)  
        start timer  
    pass segment to IP  
    NextSeqNum=NextSeqNum+length(data)  
    break;
```

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

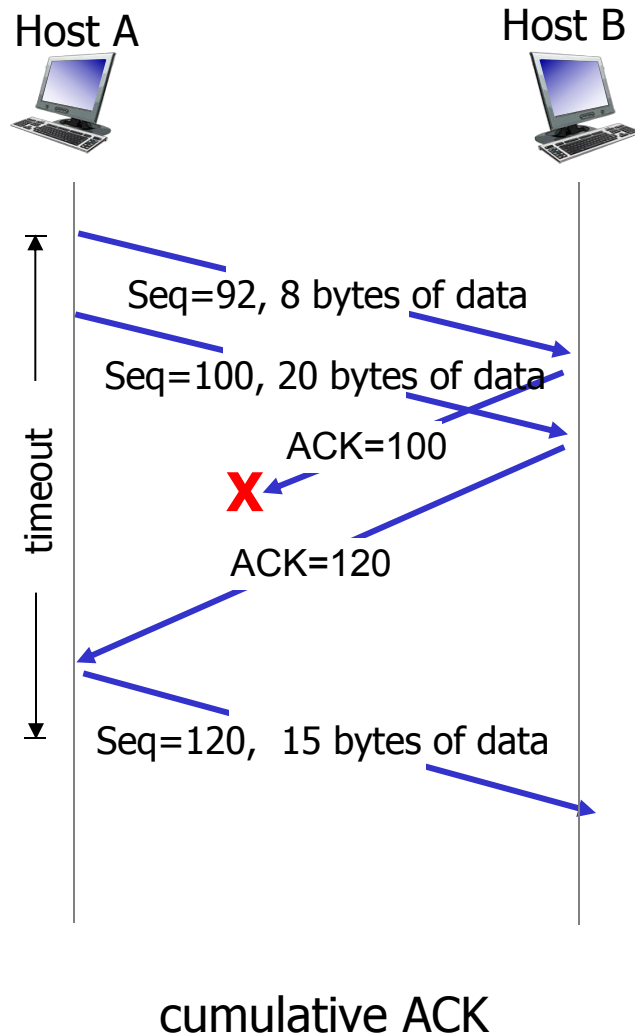
```
event: ACK received, with ACK field value of y  
    if (y > SendBase) {  
        SendBase=y  
        if (there are currently any not-yet-acknowledged segments)  
            start timer  
    }  
    break;
```

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

# TCP: retransmission scenarios



# TCP: retransmission scenarios



# TCP receiver [RFC 1122, RFC 2581]

<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

# Double Timeout Interval

Each time TCP retransmits, it sets the **next timeout interval to twice** the previous value

- ❖ Suppose *TimeoutInterval* associated with the oldest not yet acknowledged segment is 0.75 sec.
- ❖ when the timer first expires, TCP will then retransmit this segment and set the new expiration time to 1.5 sec.
- ❖ If the timer expires again, TCP will again retransmit this segment, now setting the expiration time to 3.0 sec.

When **the timer is started** after either of the two other events (that is, data received from application above, and ACK received):

- ❖ the *TimeoutInterval* is derived from the most recent values of *EstimatedRTT* and *DevRTT*.

Part of congestion control: being polite

# TCP fast retransmit

- ❖ 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 duplicate ACKs** for same data  
( “triple duplicate ACKs” ), resend unacked segment with smallest seq #
- likely that unacked segment lost, so don't wait for timeout

# TCP fast retransmit

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

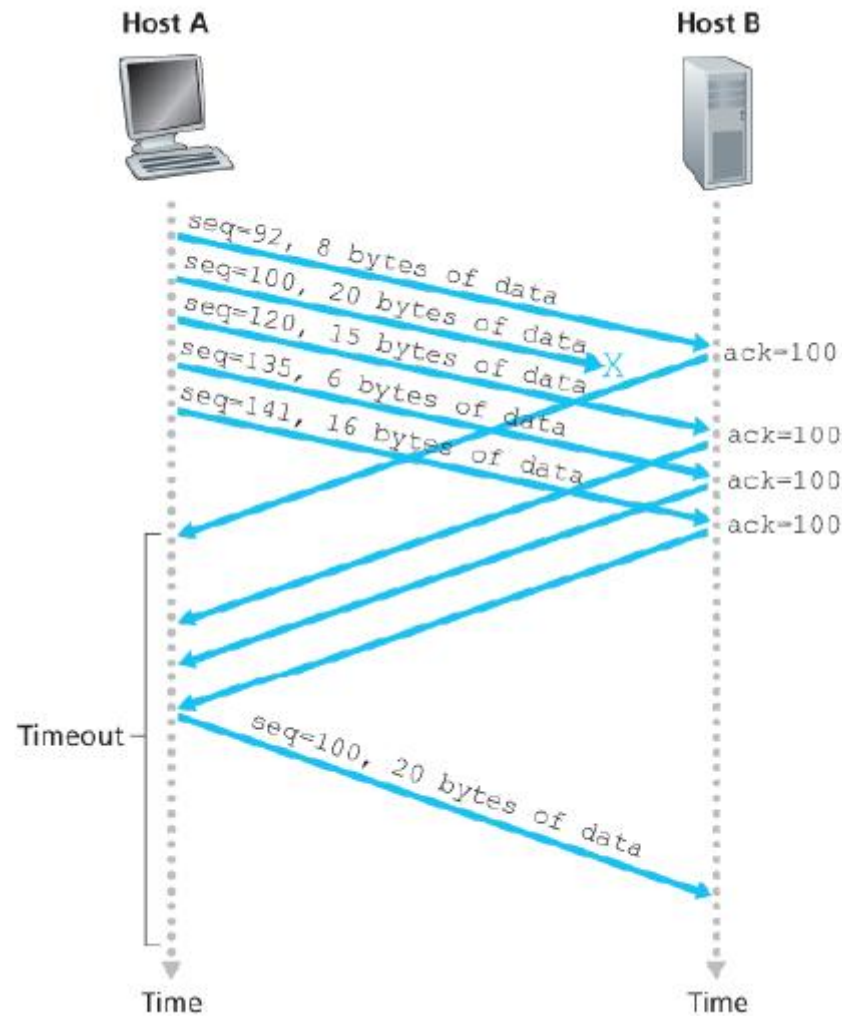
        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase=y
                if (there are currently any not yet
                    acknowledged segments)
                    start timer
            }
            break;

        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;

        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;

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

# TCP fast retransmit



fast retransmit after sender  
receipt of triple duplicate ACK

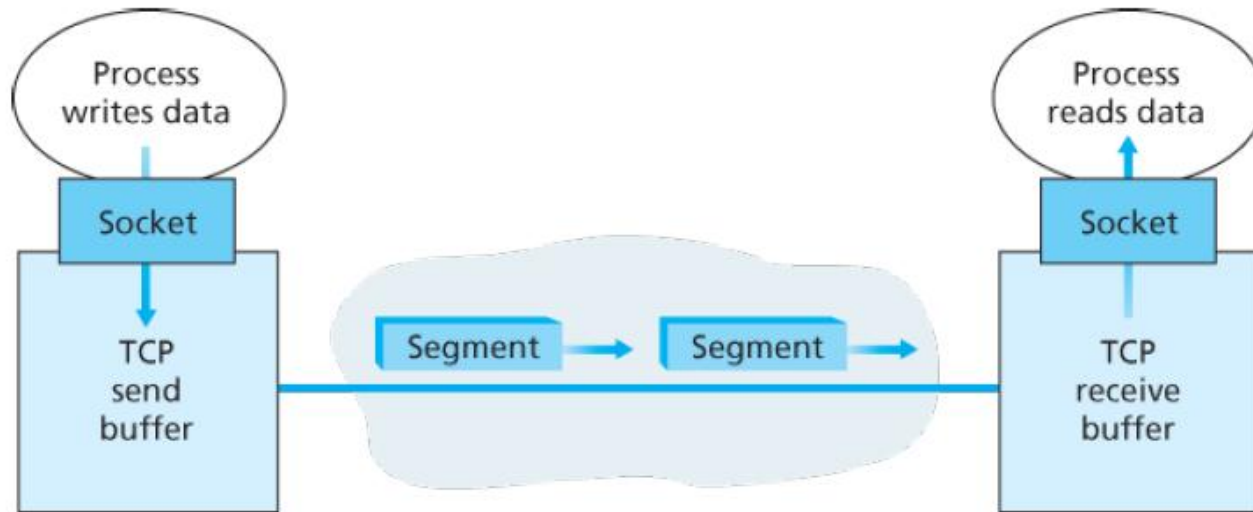


# TCP Reliable Data Transfer

- ❖ Segment structure
- ❖ Round-trip time estimation
- ❖ Reliable data transfer
- ❖ **Flow control**
- ❖ Control management

# TCP: Overview

---



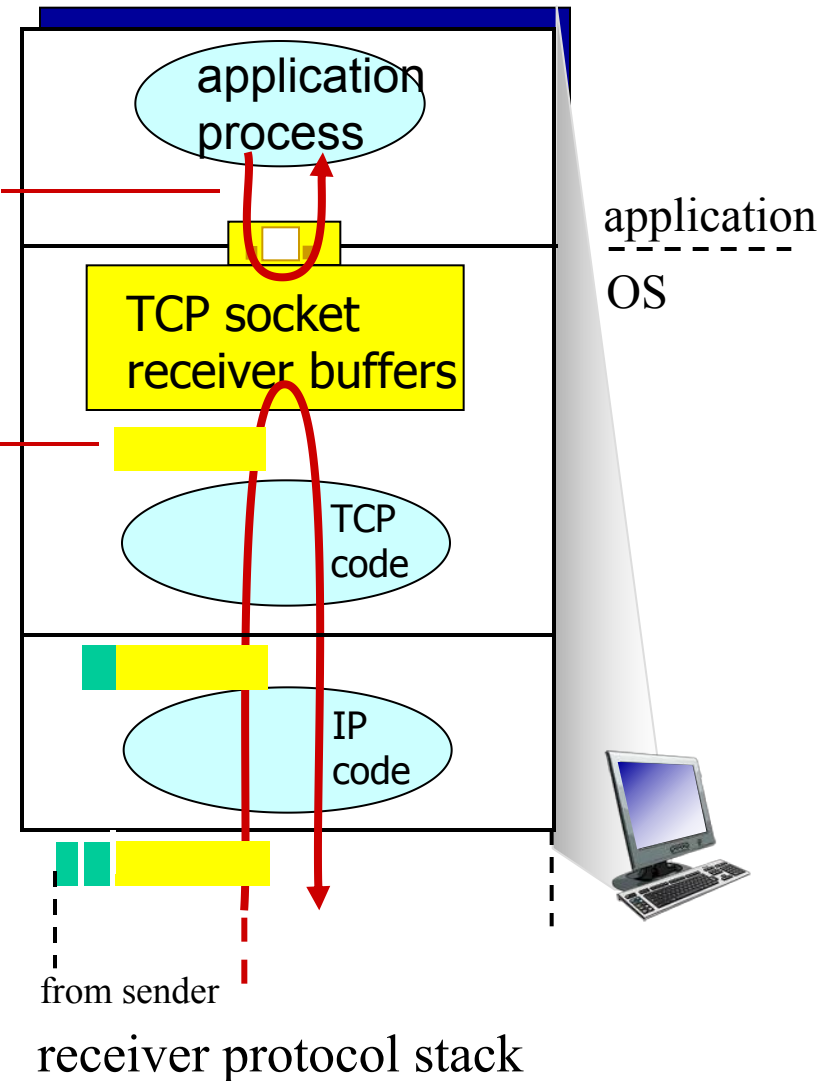
- TCP connection
- TCP grab chunks of data from the sender buffer
- TCP receives a segment at the other end, place it in receiver buffer
- application reads the stream from the receive buffer

# TCP flow control

application may  
remove data from  
TCP socket buffers ....

... slower than TCP  
receiver is  
delivering  
(sender is sending)

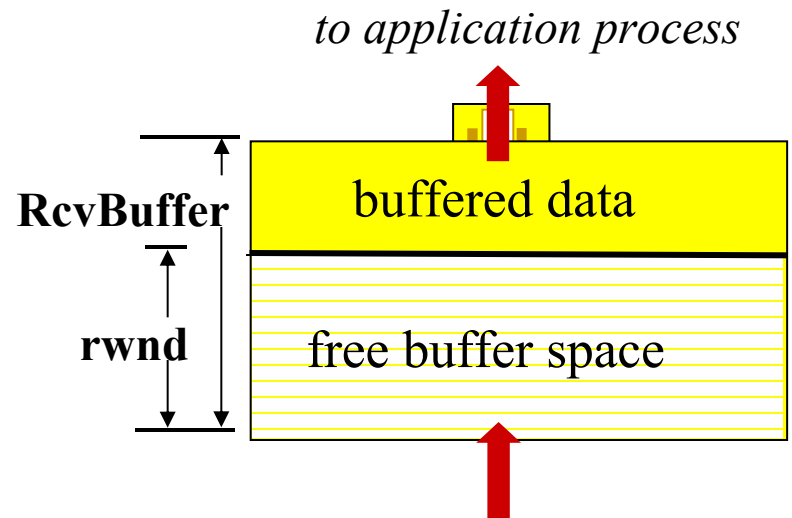
*flow control*  
Receiver controls sender, so  
sender won't overflow receiver's  
buffer by transmitting too much,  
too fast



# TCP flow control

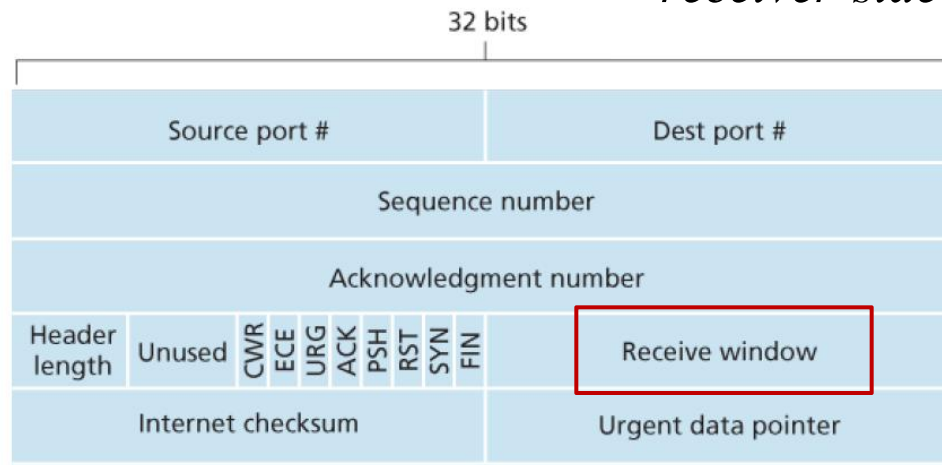
**Receiver** “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments

- ❖ **RcvBuffer** size set via socket options (typical default is 4096 bytes)
- ❖ many operating systems autoadjust **RcvBuffer**



*TCP segment payloads*  
*receiver-side buffering*

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

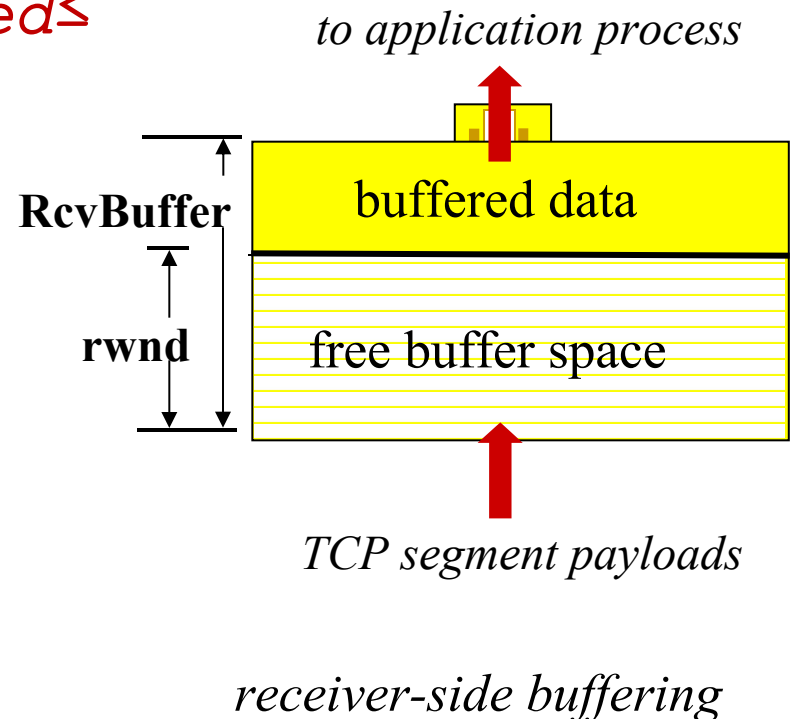


# TCP flow control

**Sender** limits amount of unacked ( “in-flight” ) data to receiver's **rwnd** value

*$LastByteSent - LastByteAcked \leq rwnd$*

Guarantees receive buffer will not overflow



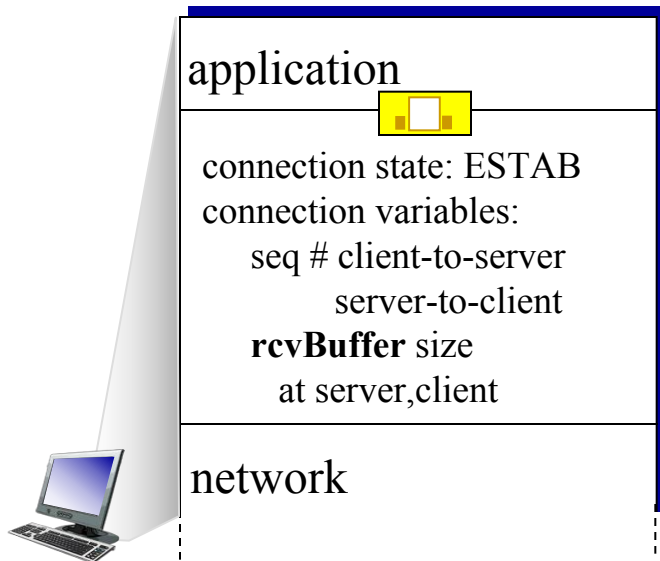
# TCP Reliable Data Transfer

- ❖ Segment structure
- ❖ Round-trip time estimation
- ❖ Reliable data transfer
- ❖ Flow control
- ❖ Control management

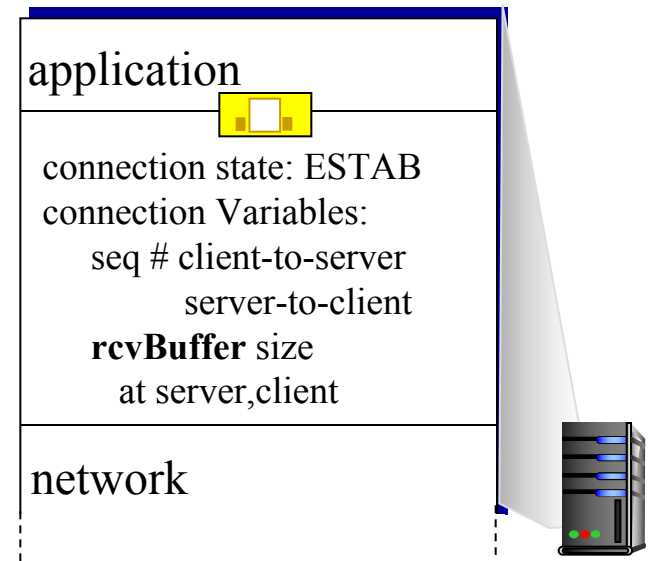
# Connection Management

before exchanging data, sender/receiver “handshake” :

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



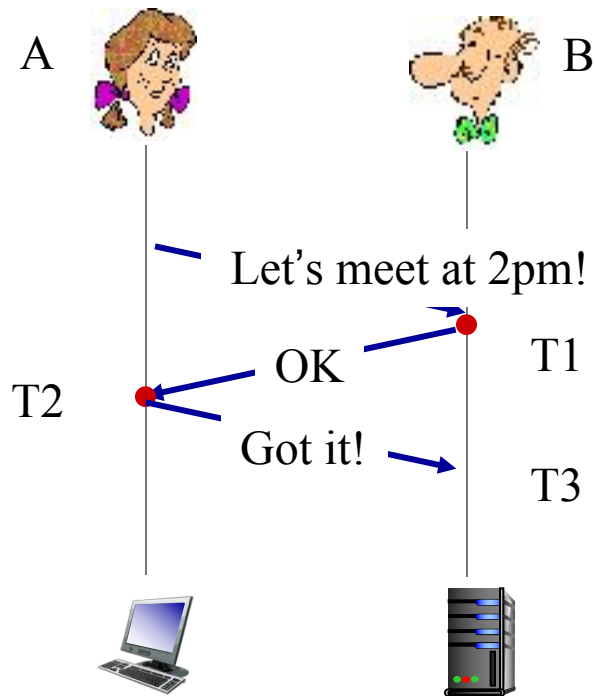
```
clientSocket = socket(AF_INET, SOCK_STREAM);  
clientSocket.connect((hostname,port number));
```



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

# Agreeing to establish a connection

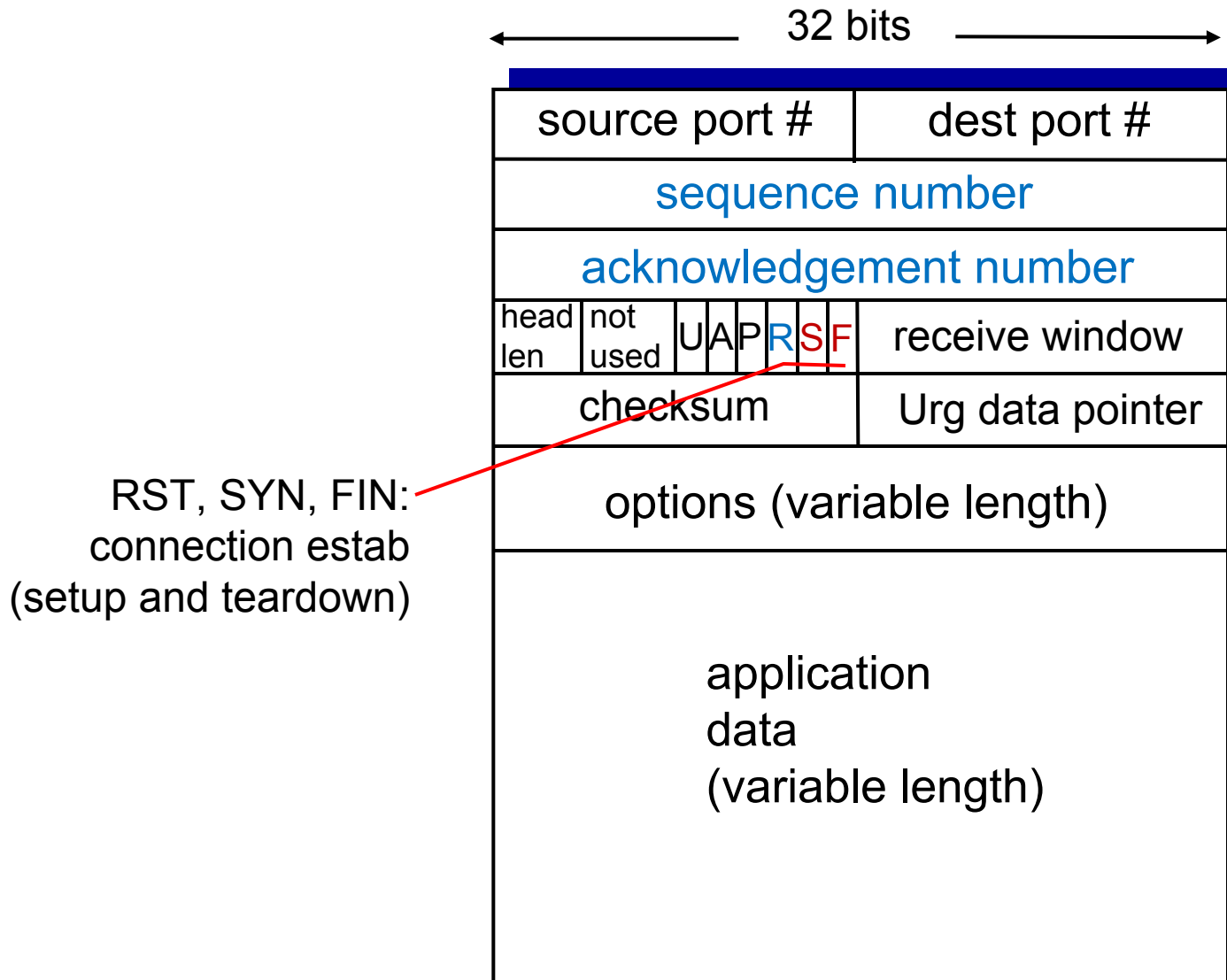
3-way handshake:



- ❖ T1: B knows A's transmitter and B's receiver is OK
- ❖ T2: A knows A's transceiver and B's transceiver is OK, B has no more information than T1
- ❖ T3: Both A and B know their transceiver are OK, they can start the communication!



# TCP segment structure



# TCP 3-way handshake

## client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x  
send TCP SYN msg

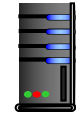


SYNbit=1, Seq=x

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

SYNbit=0  
ACKbit=1, ACKnum=y+1



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

received ACK(y)  
indicates client is live

## server state

LISTEN

SYN RCVD

ESTAB

Once these three steps have been completed, the client and server hosts can send segments containing data to each other.

- In each of these future segments, SYNbit=0

# TCP: closing a connection

## client state

ESTAB

clientSocket.close()

FIN\_WAIT\_1

can no longer  
send but can  
receive data

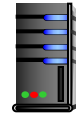
FIN\_WAIT\_2

wait for server  
close

TIMED\_WAIT

timed wait

CLOSED



FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

can still  
send data

can no longer  
send data

## server state

ESTAB

CLOSE\_WAIT

LAST\_ACK

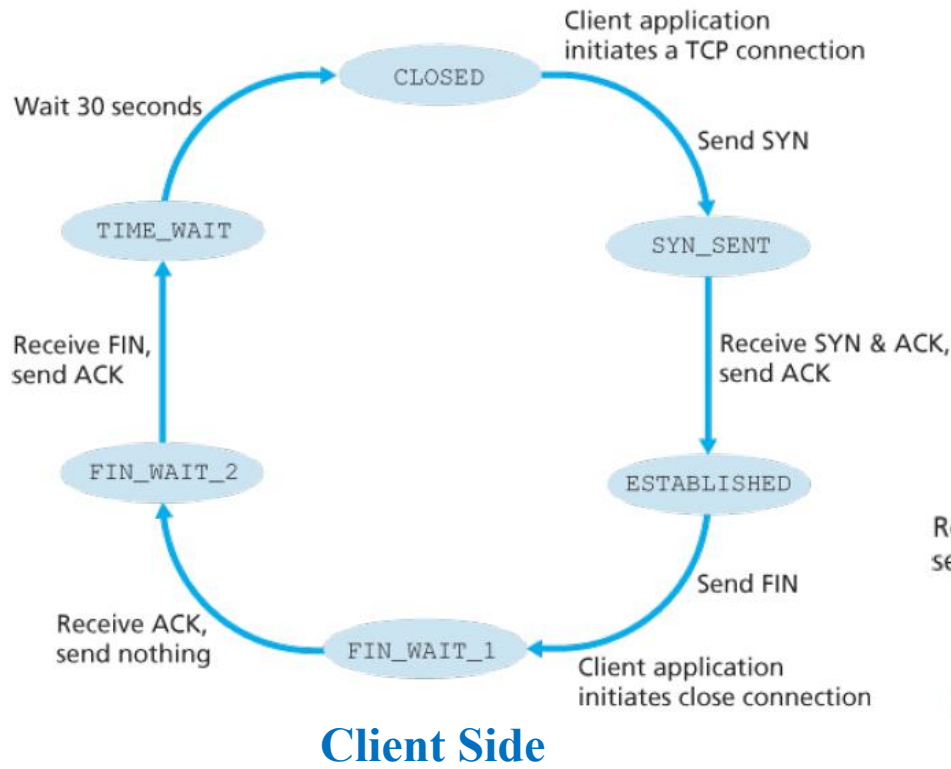
CLOSED

The TIME\_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost.

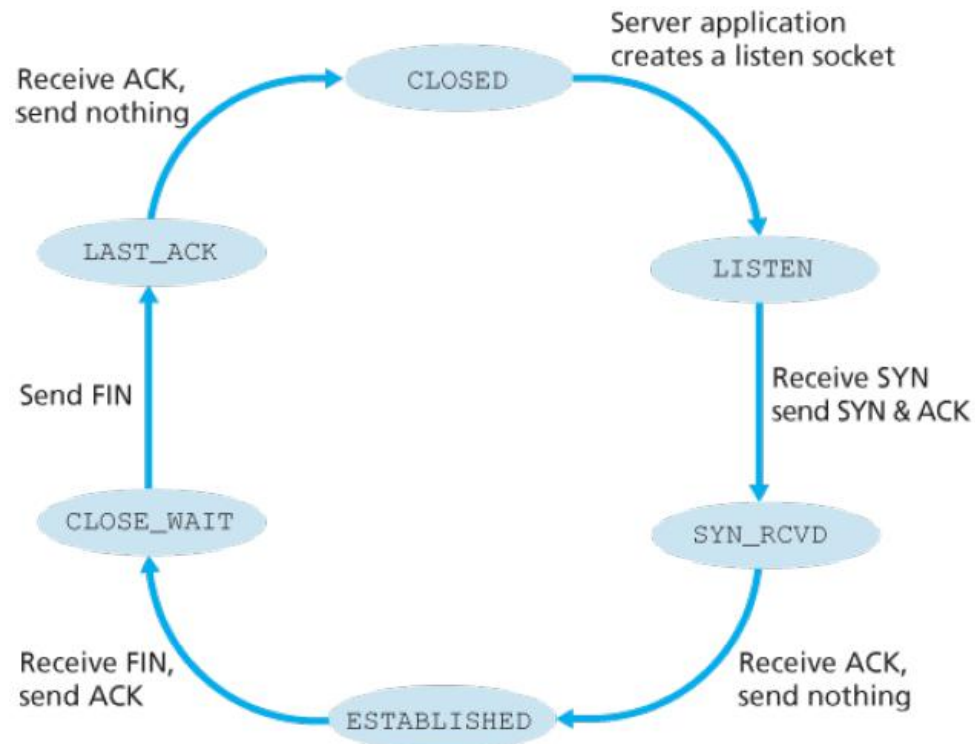
# TCP: closing a connection

- ❖ Four-way handshaking
  - Either of the two processes participating in a TCP connection can end the connection.
- ❖ client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- ❖ respond to received FIN with ACK
- ❖ Why FIN and ACK can not be sent in one msg as SYNACK in connection establishment?
  - The other side may still have packets need to be sent. It can not send FIN until the transmission is finished.

# TCP States



## Server Side



# Reset Segment

---

When a host receives a TCP segment whose port numbers or source IP address do not match with any of the ongoing sockets.

- ❖ Then the host will send a special reset segment to the source. RST flag bit is set to 1.
- ❖ “I don’t have a socket for that segment. Please do not resend the segment.”

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Principles of congestion control

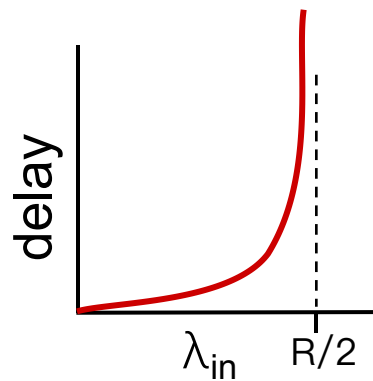
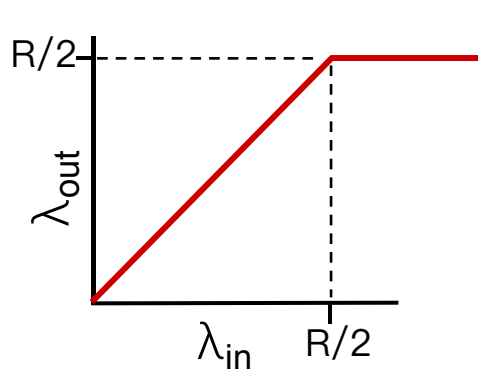
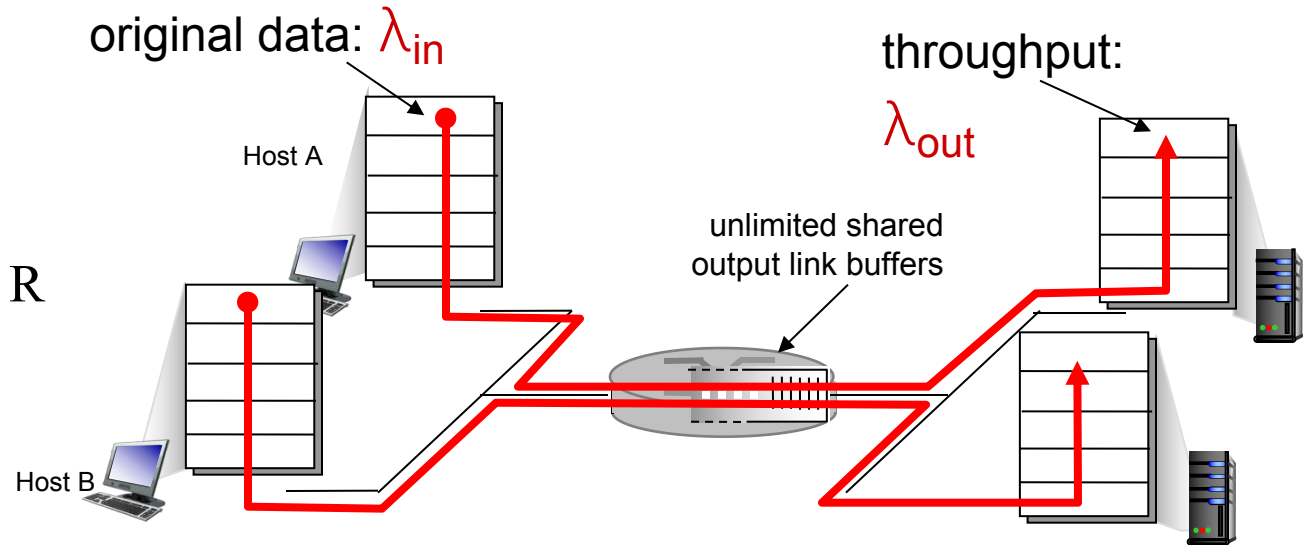
## Congestion:

- ❖ 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

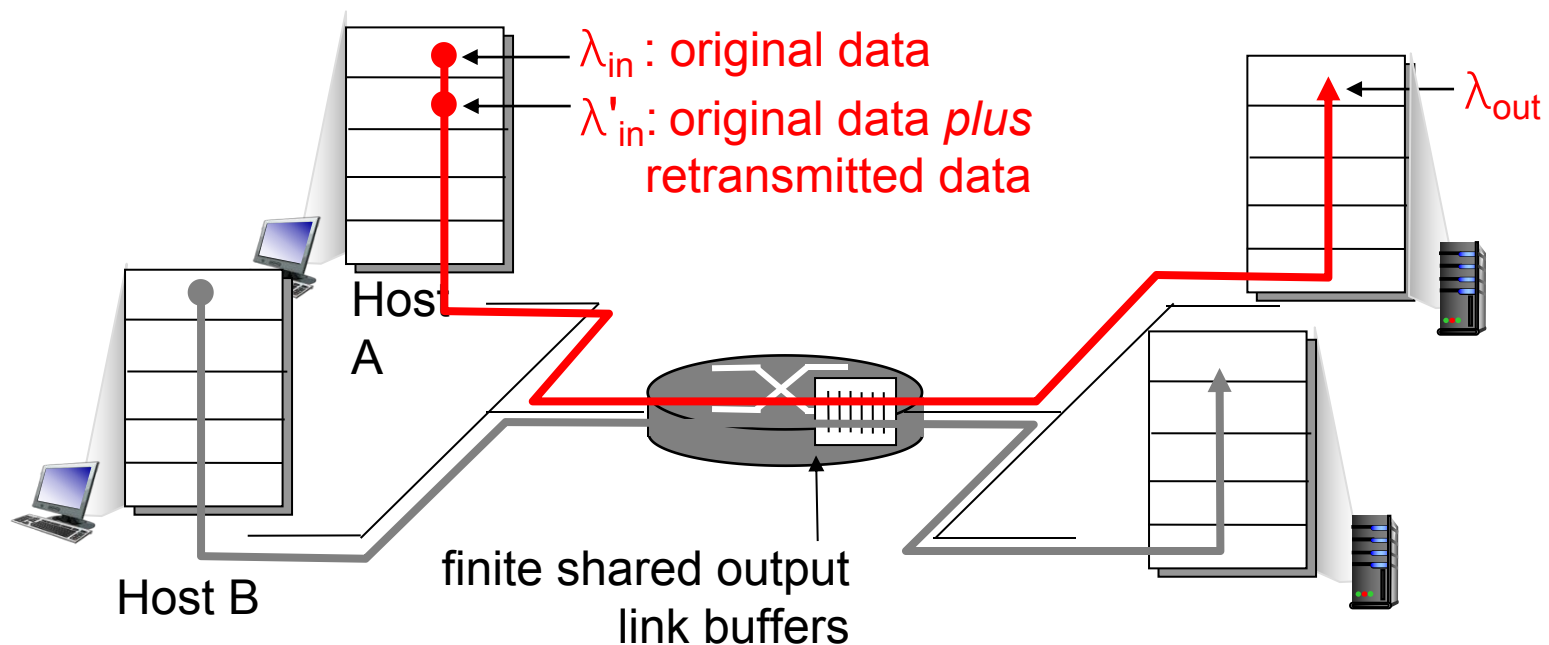


*large queuing delays are experienced as the packet-arrival rate nears the link capacity.*

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

# Causes/costs of congestion: scenario 2

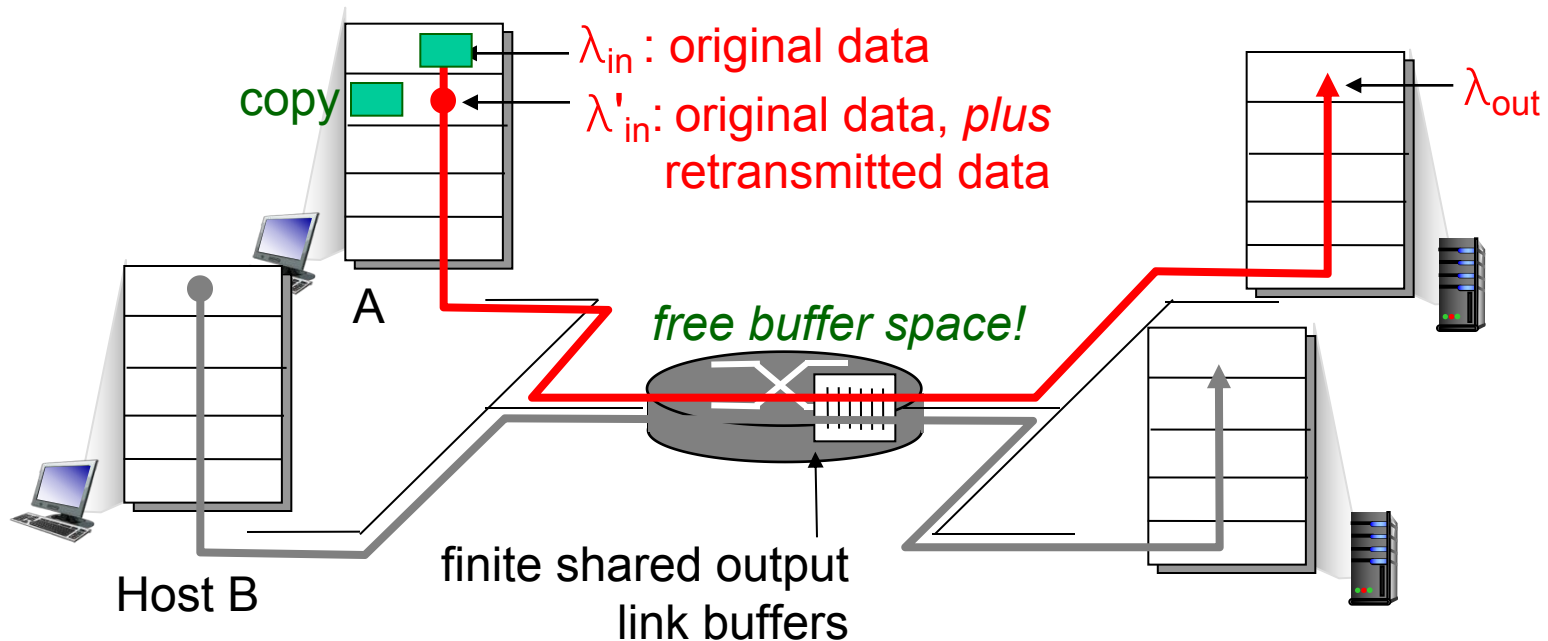
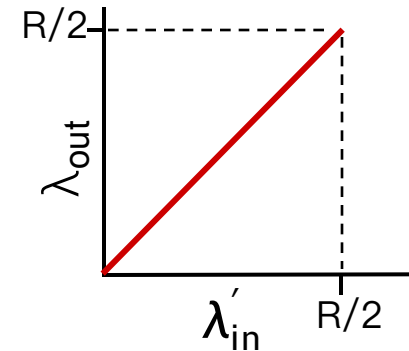
- ❖ one router, *finite* buffers
- ❖ sender retransmission of timed-out packet
  - application-layer input = application-layer output:  $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* :  $\lambda'_{in} \geq \lambda_{in}$



# Causes/costs of congestion: scenario 2

Idealization: perfect knowledge

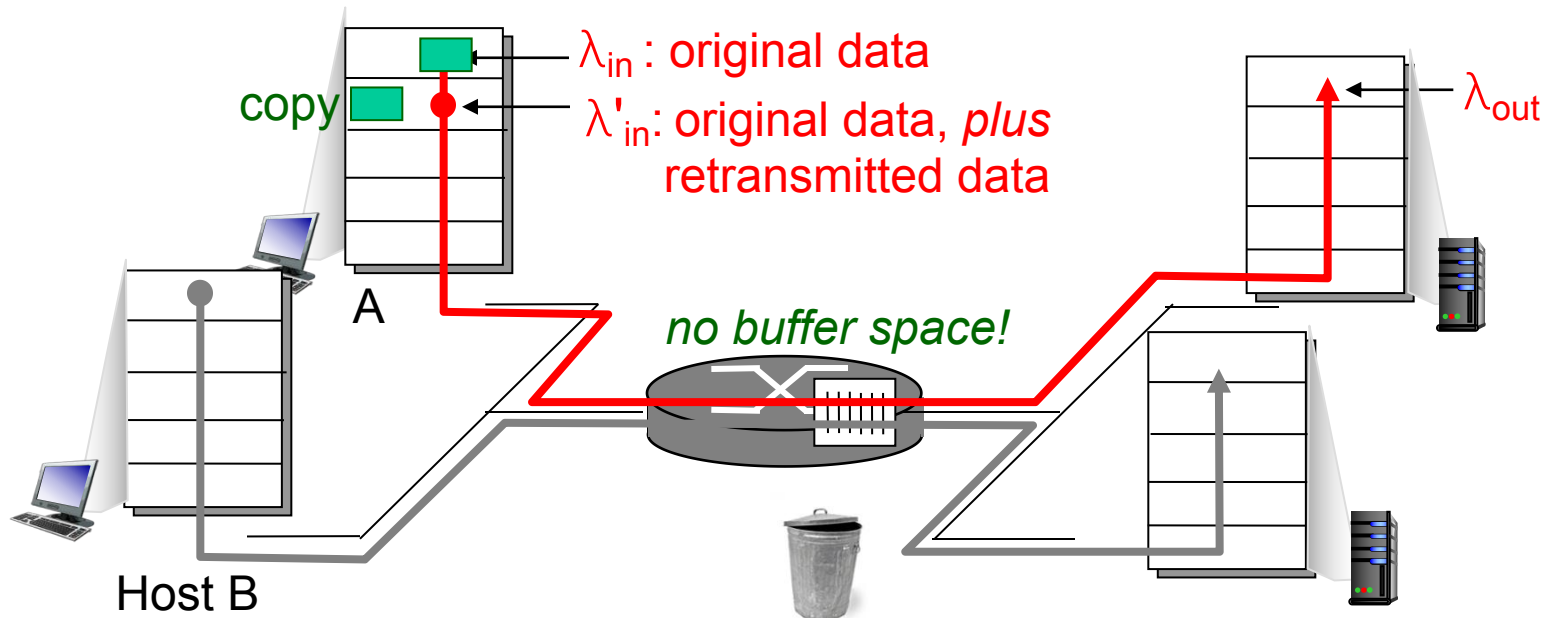
- ❖ Sender sends only when router buffers available
- ❖ No loss occurs:  $\lambda'_{in} = \lambda_{in}$



## 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
- ❖  $\lambda'_{in} \geq \lambda_{in}$

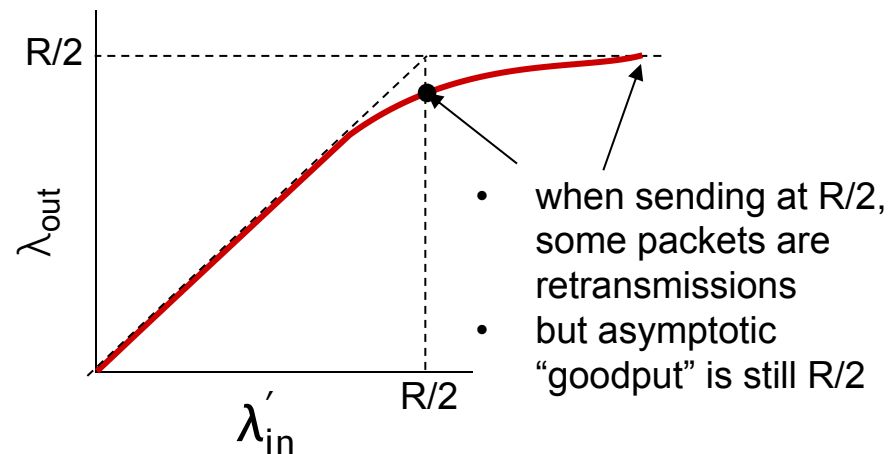


# Causes/costs of congestion: scenario 2

*Idealization: known loss* packets can be lost, dropped at router due to full buffers

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

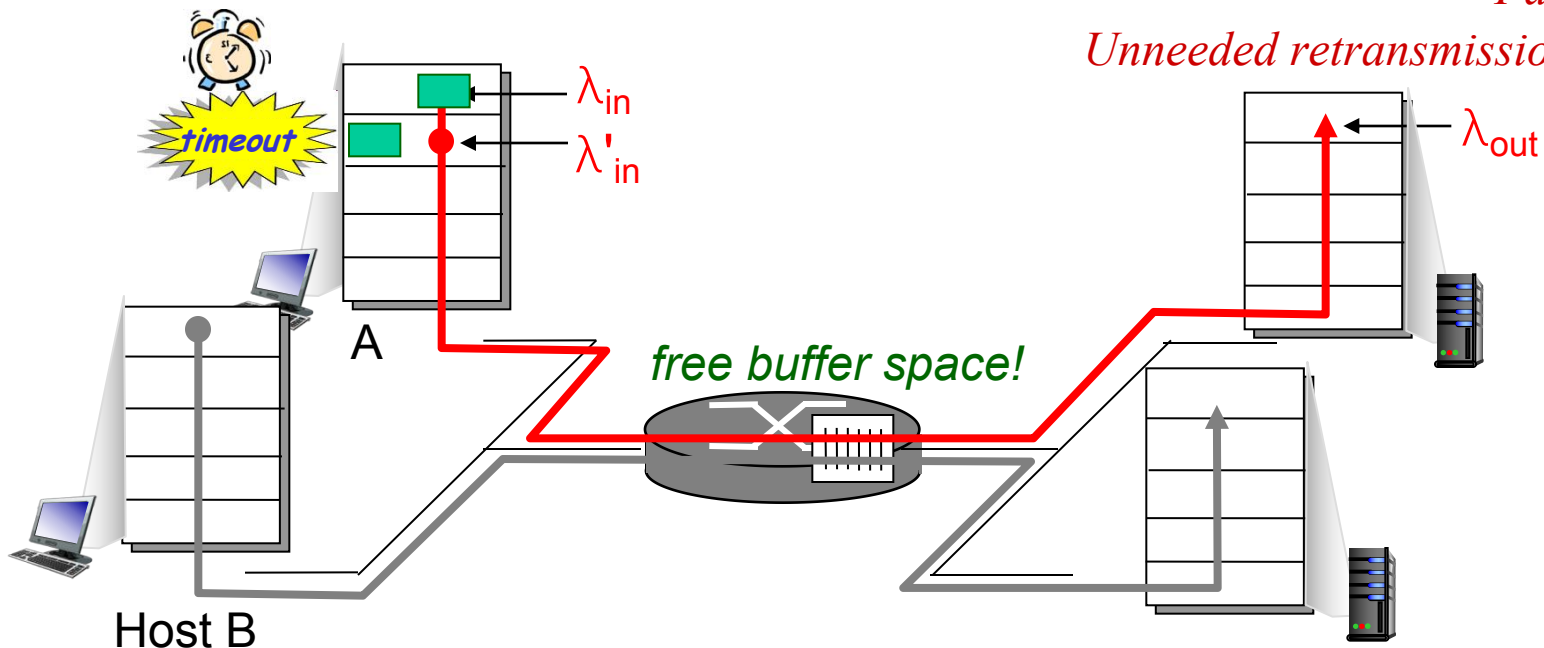
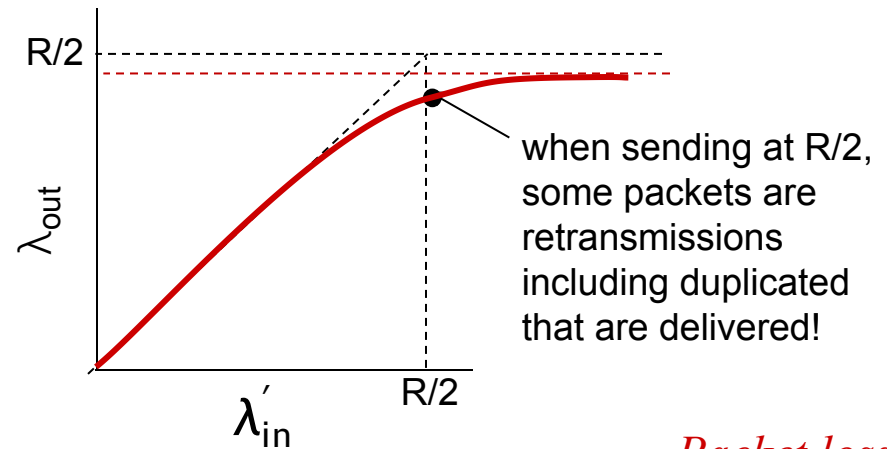
❖  $\lambda'_{in} \geq \lambda_{in}$



# 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



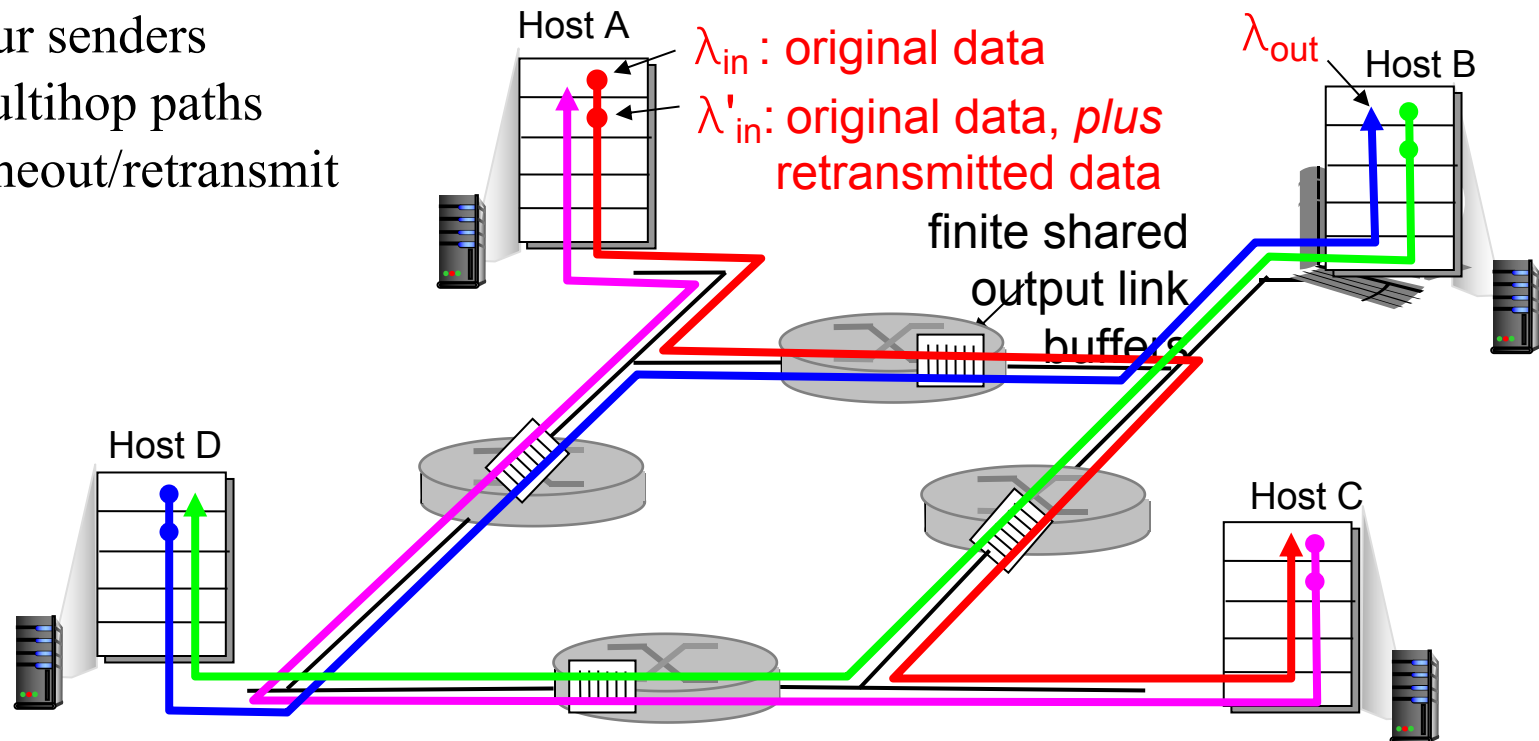
# Causes/costs of congestion: scenario 3

For small values of  $\lambda_{in}^-$ :

- buffer overflows are rare
- the throughput  $\lambda_{out}$  approximately equals the offered load

$$\lambda'_{in} = \lambda_{in}.$$

- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit

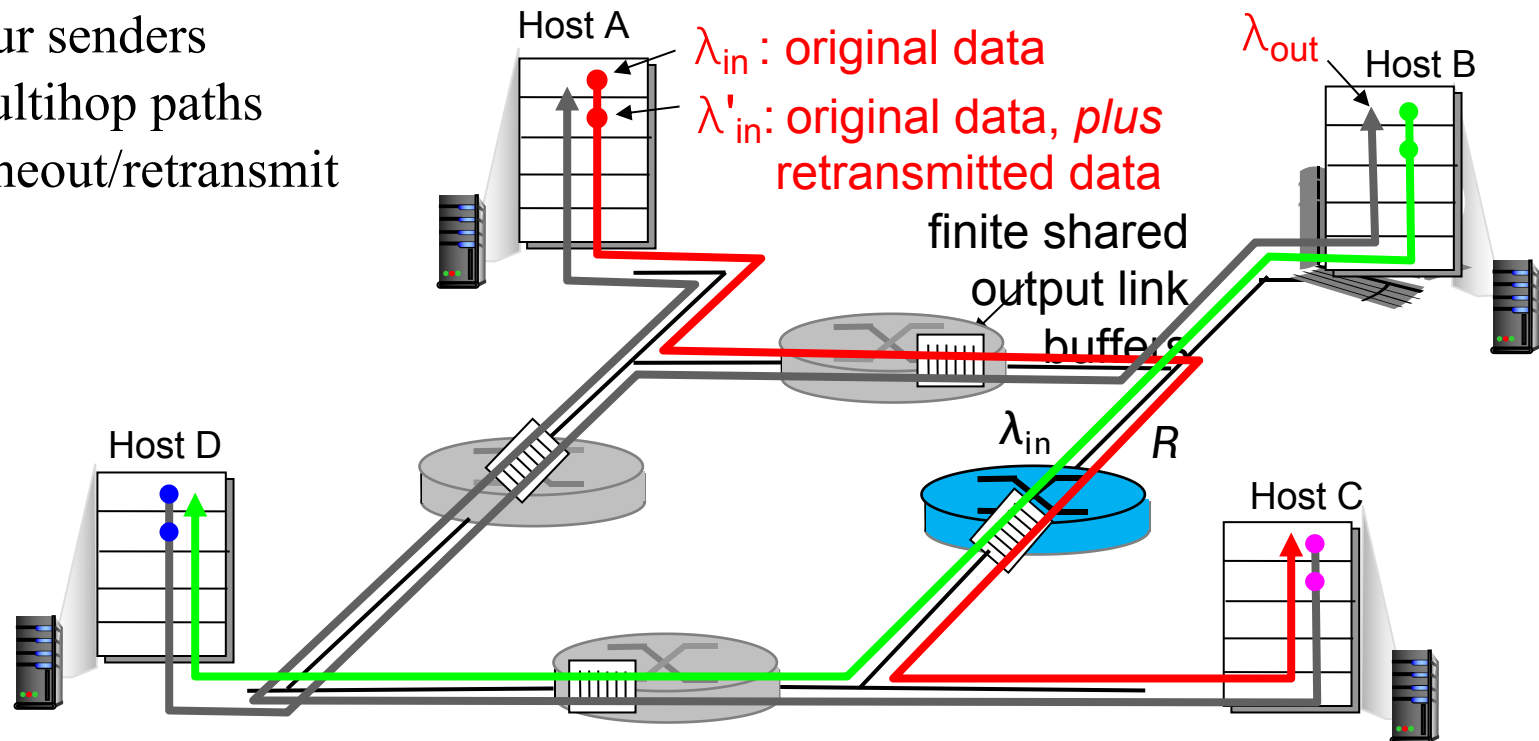


# Causes/costs of congestion: scenario 3

Q: what happens as  $\lambda_{in}$  increases ?

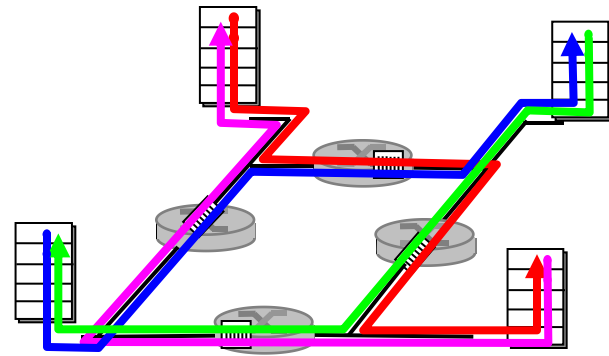
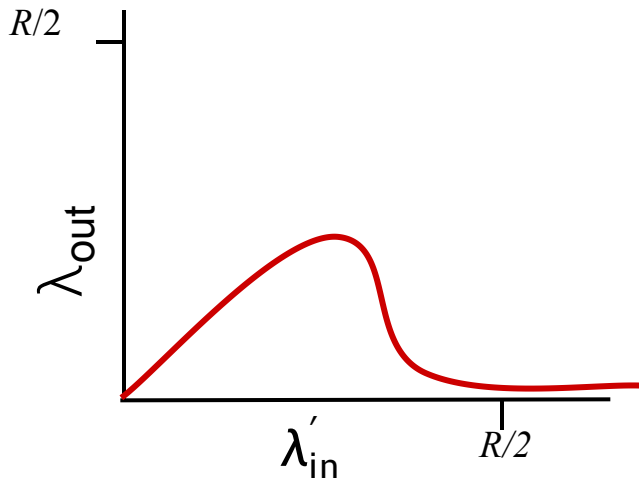
A: as green  $\lambda_{in}$  increases, all arriving red pkts at upper queue are dropped, red throughput goes 0

- ❖ four senders
- ❖ multihop paths
- ❖ timeout/retransmit





# Causes/costs of congestion: scenario 3



another “cost” of congestion:

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

# Cause and Cost of Congestion

---

## **Cause**

- Shared link; limited link capacity
- Sending at a high rate

## **Cost of Congestion**

- Delay
- Packet lost and retransmission
- Unneeded retransmission: waste
- “upstream” transmission capacity was wasted

# Approaches to Congestion Control

## **End-to-end congestion control:**

- TCP segment loss or round-trip segment delay
- TCP decreases its **window size** accordingly

## **Network-assisted congestion control:**

- **routers provide feedback** to the sender and/or receiver
- a single bit indicating congestion at a link; the maximum host sending rate the router can support

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control