# Chapter 3 Transport Layer

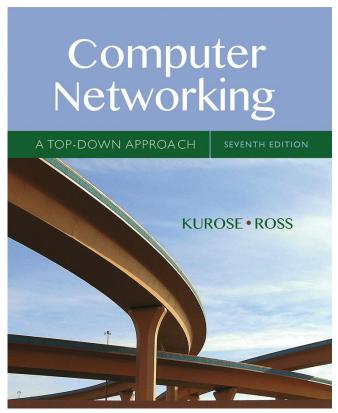
#### A note on the use of these Powerpoint slides:

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

© All material copyright 1996-2016 J.F Kurose and K.W. Ross, All Rights Reserved



#### Computer Networking: A Top Down Approach

7<sup>th</sup> edition Jim Kurose, Keith Ross Pearson/Addison Wesley April 2016

# Chapter 3: Transport Layer

#### our 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

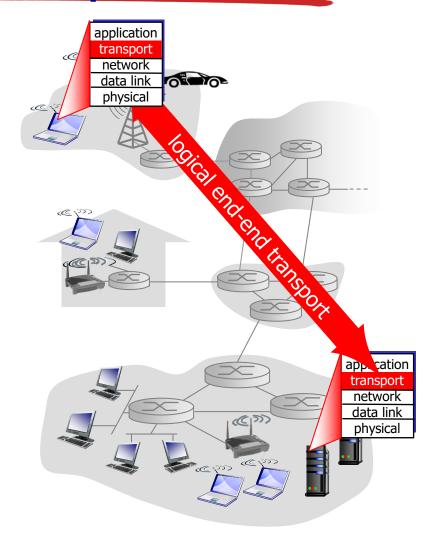
# Chapter 3 outline

- 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

# Transport services and protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



# Transport vs. network layer

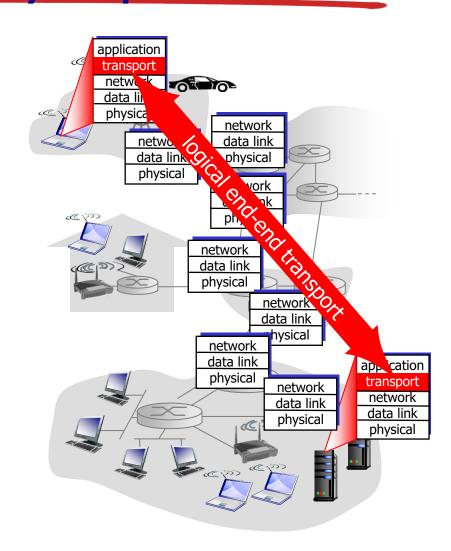
- network layer: logical communication between hosts
- transport layer: logical communication between processes
  - relies on, enhances, network layer services

#### 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 inhouse siblings
- network-layer protocol = postal service

# Internet transport-layer protocols

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

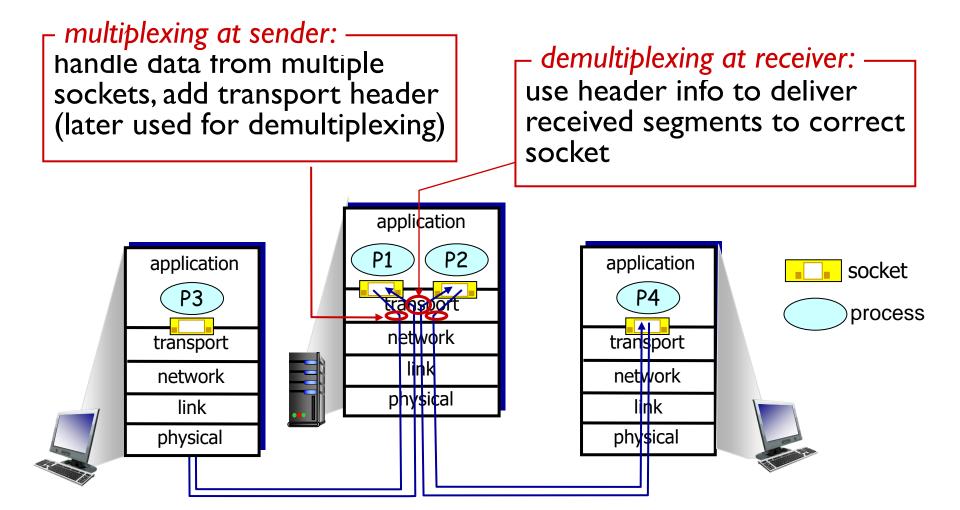


# Chapter 3 outline

- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer

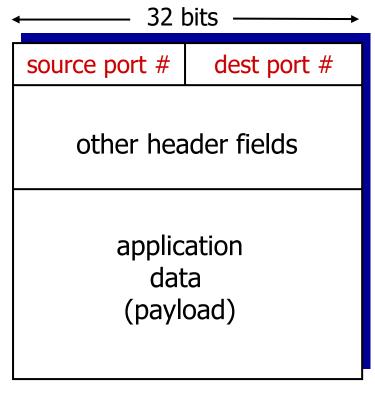
- 3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

# Multiplexing/demultiplexing



### How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

# Connectionless demultiplexing

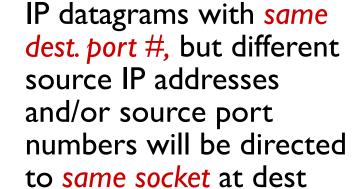
recall: created socket has host-local port #:

DatagramSocket mySocket1

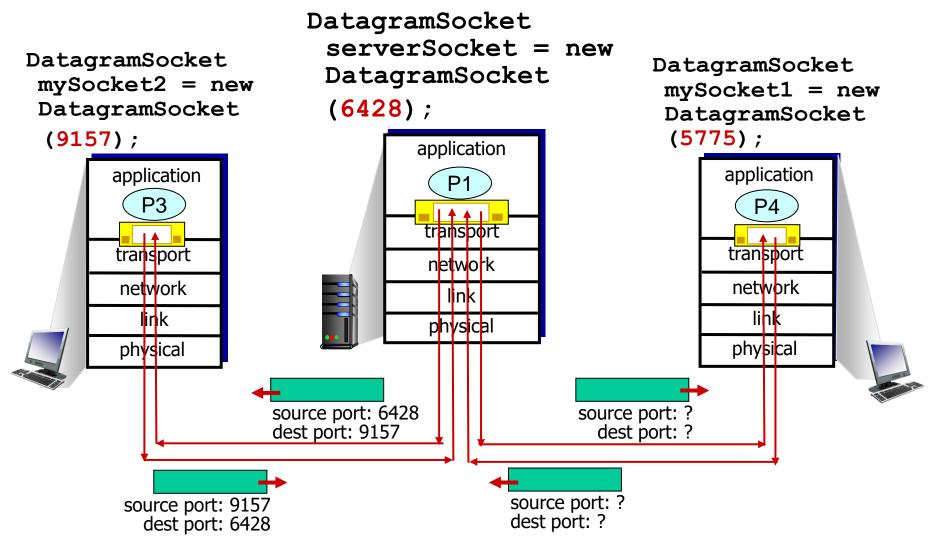
= new DatagramSocket(12534);

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

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #



# Connectionless demux: example

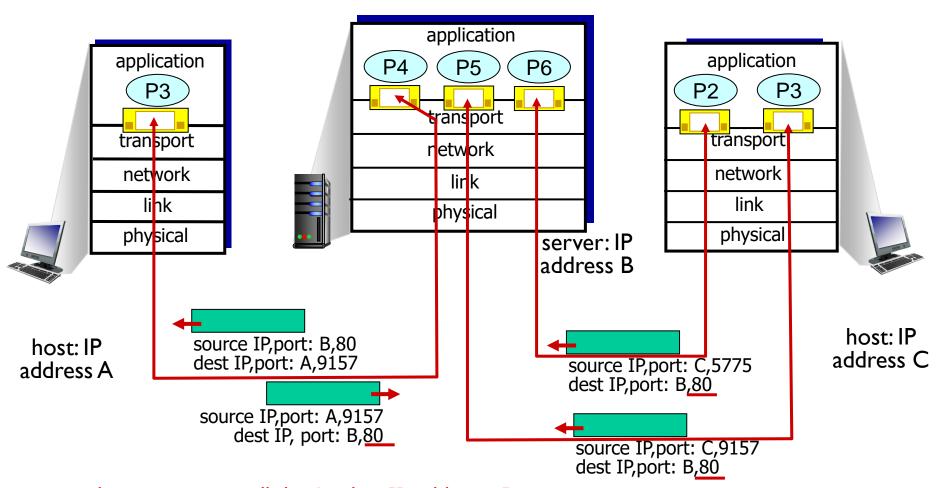


#### Connection-oriented demux

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

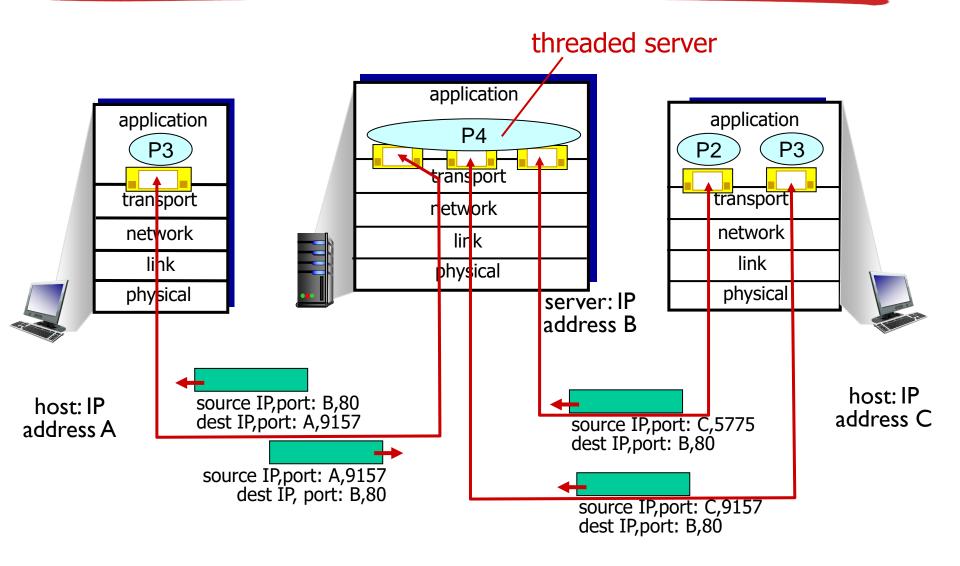
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

### Connection-oriented demux: example



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

### Connection-oriented demux: example



# 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

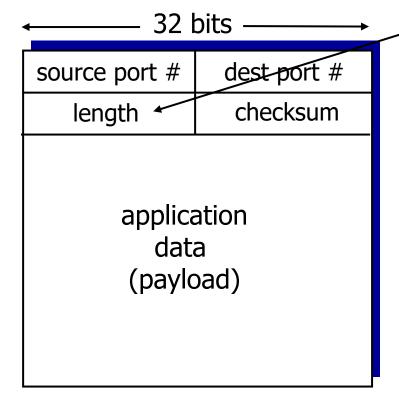
- 3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

### UDP: User Datagram Protocol [RFC 768]

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

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

# **UDP:** segment header



UDP segment format

length, in bytes of UDP segment, including header

#### 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

### **UDP** checksum

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

#### sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

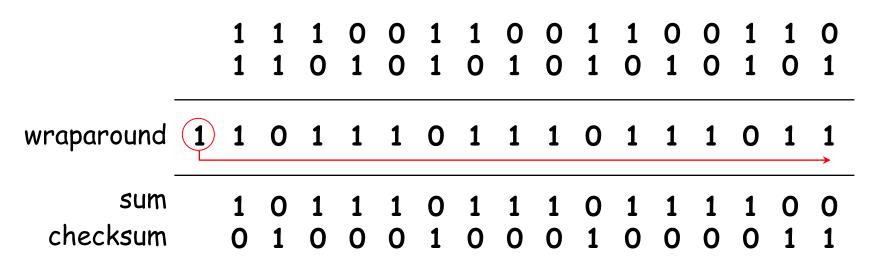
#### receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO error detected
  - YES no error detected.
     But maybe errors
     nonetheless? More later

• • • •

# Internet checksum: example

example: add two 16-bit integers



Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

<sup>\*</sup> Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose\_ross/interactive/

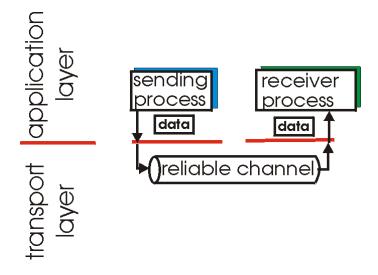
# Chapter 3 outline

- 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 reliable data transfer

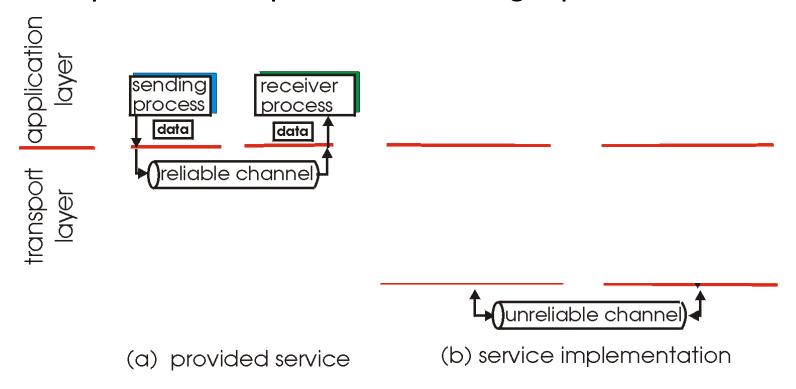
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

# Principles of reliable data transfer

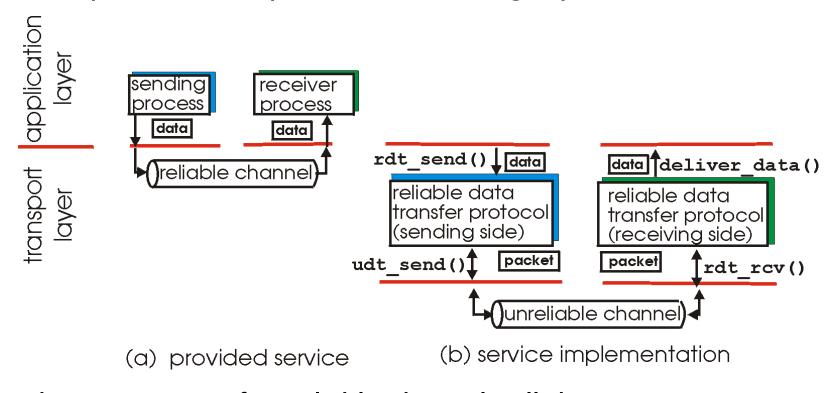
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

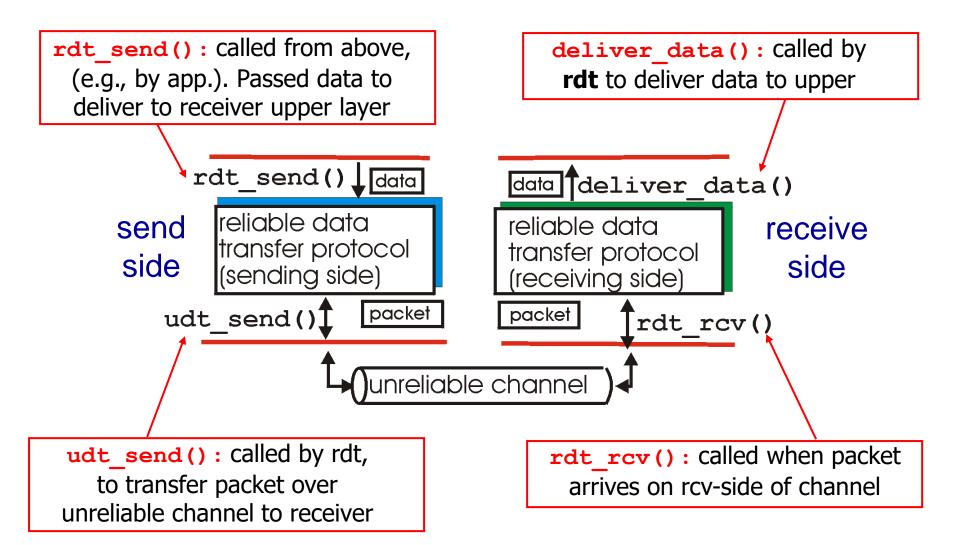
# Principles of reliable data transfer

- 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)

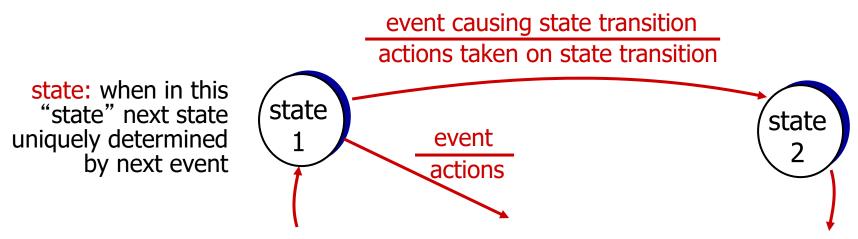
#### Reliable data transfer: getting started



#### Reliable data transfer: getting started

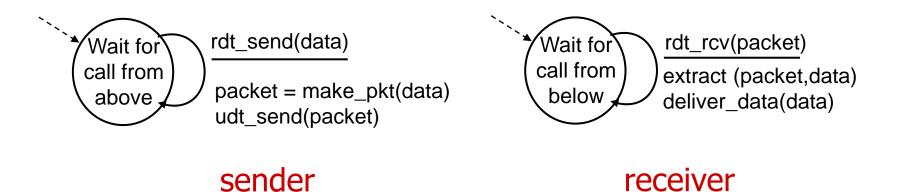
#### we'll:

- 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



#### rdt I.O: reliable transfer over a reliable channel

- 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



### rdt2.0: channel with bit errors

- 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?

### rdt2.0: channel with bit errors

- 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

### rdt2.0: FSM specification

rdt\_send(data)
sndpkt = make\_pkt(data, checksum)
udt\_send(sndpkt)

Wait for
call from
above

rdt\_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt\_send(sndpkt)

rdt\_rcv(rcvpkt) && isACK(rcvpkt)

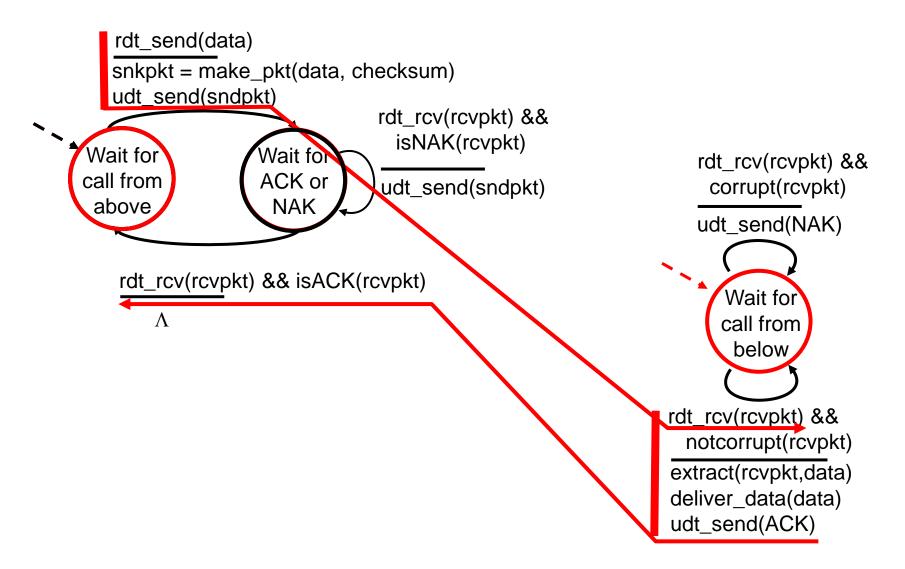
A

sender

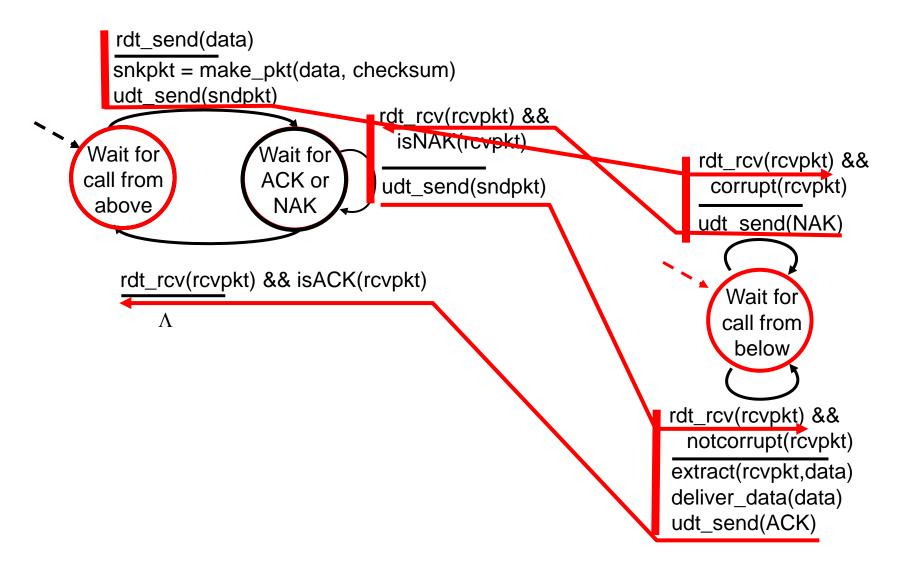
#### receiver

rdt\_rcv(rcvpkt) && corrupt(rcvpkt) udt send(NAK) Wait for call from below rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt) extract(rcvpkt,data) deliver\_data(data) udt\_send(ACK)

### rdt2.0: operation with no errors



#### rdt2.0: error scenario



## rdt2.0 has a fatal flaw!

# what happens if ACK/NAK corrupted?

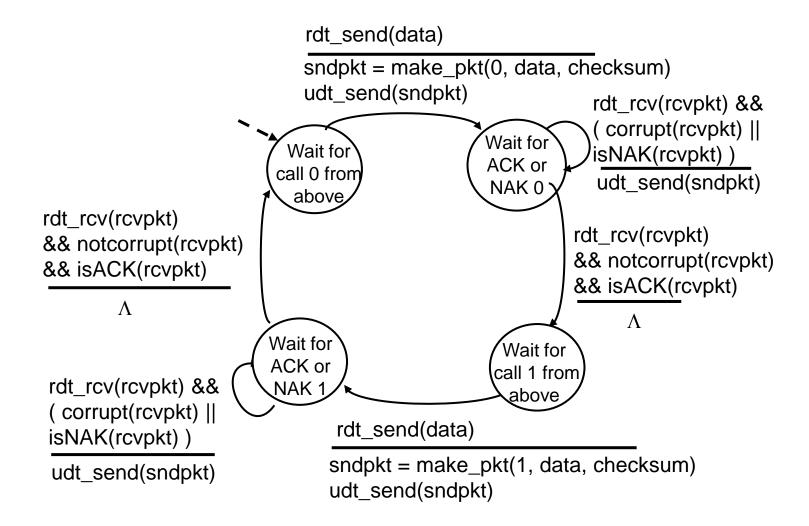
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

#### handling duplicates:

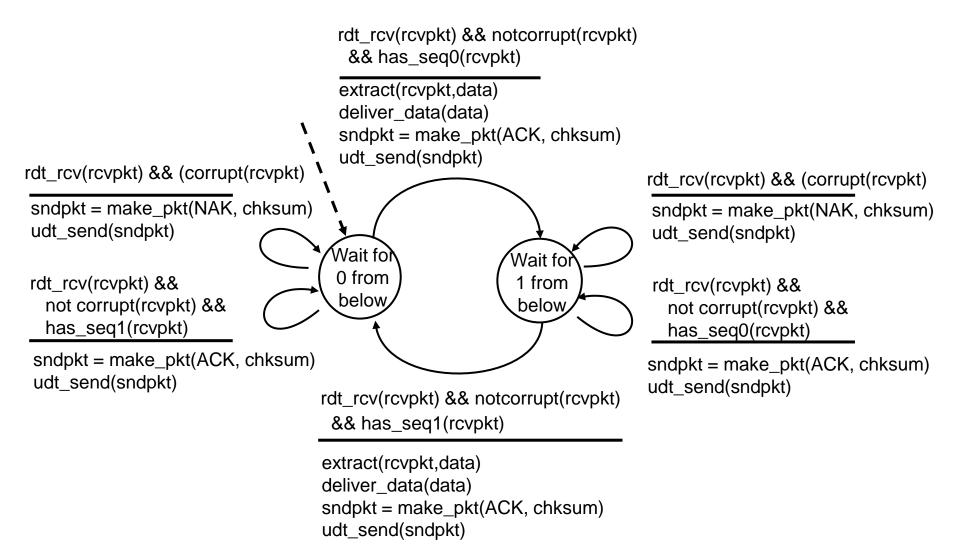
- sender retransmits current pkt if ACK/NAK corrupted
- sender adds sequence number to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

stop and wait sender sends one packet, then waits for receiver response

#### rdt2.1: sender, handles garbled ACK/NAKs



#### rdt2.1: receiver, handles garbled ACK/NAKs



### rdt2.1: discussion

#### 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 I

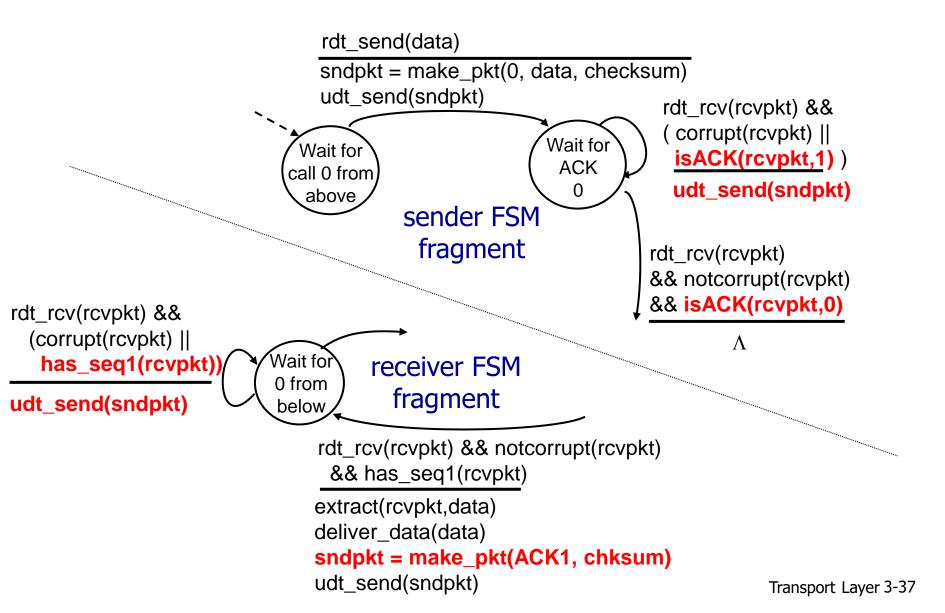
#### receiver:

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

### rdt2.2: a NAK-free protocol

- 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

### rdt2.2: sender, receiver fragments



### rdt3.0: channels with errors and loss

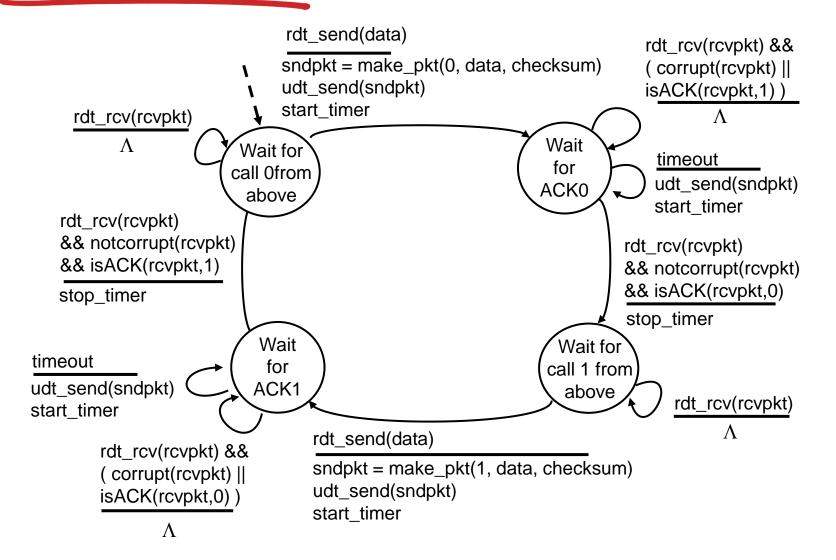
#### new assumption:

underlying channel can also lose packets (data, ACKs)

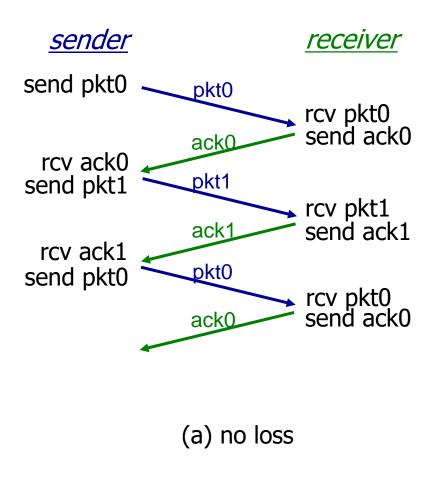
checksum, seq. #,
 ACKs, retransmissions
 will be of help ... but
 not enough

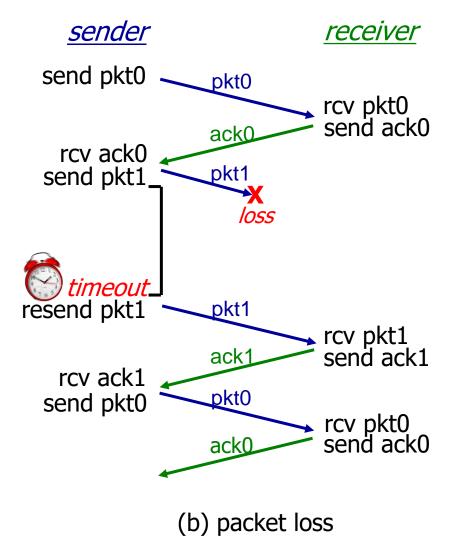
- approach: sender waits
   "reasonable" amount of
   time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

### rdt3.0 sender

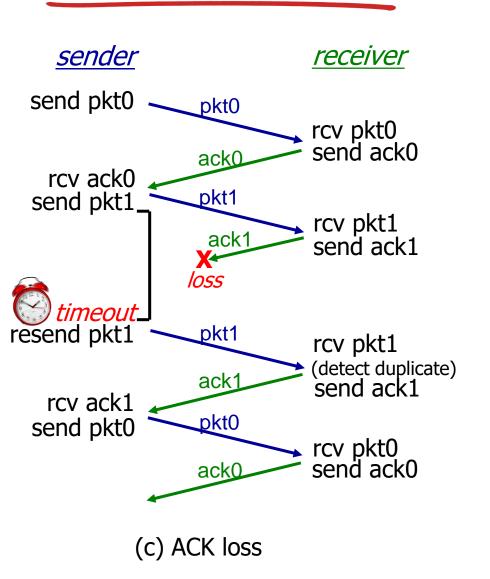


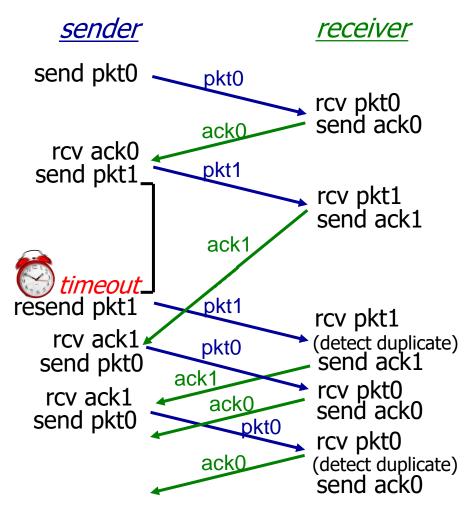
# rdt3.0 in action





### rdt3.0 in action





(d) premature timeout/ delayed ACK

### Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: I Gbps link, I5 ms prop. delay, 8000 bit packet:

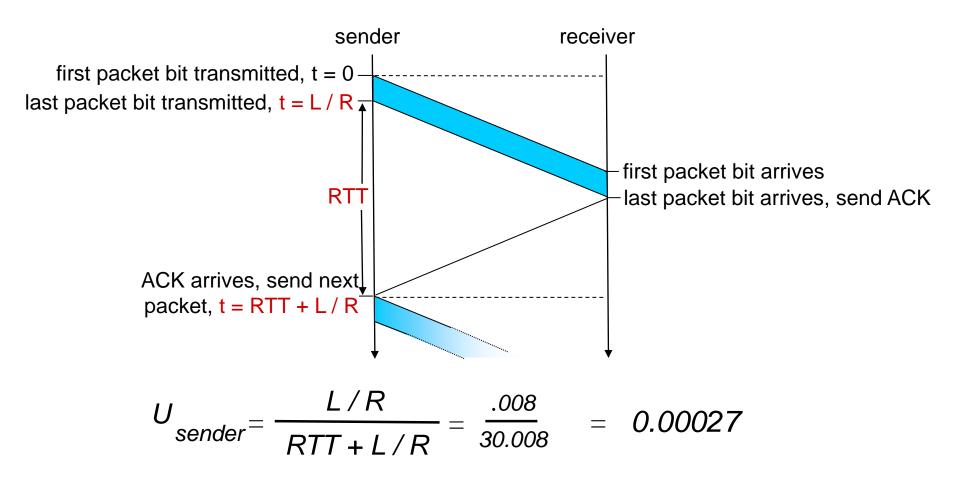
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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, IKB pkt every 30 msec: 33kB/sec thruput over I Gbps link
- network protocol limits use of physical resources!

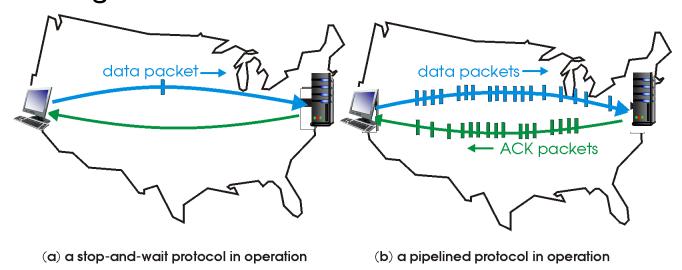
# rdt3.0: stop-and-wait operation



### Pipelined protocols

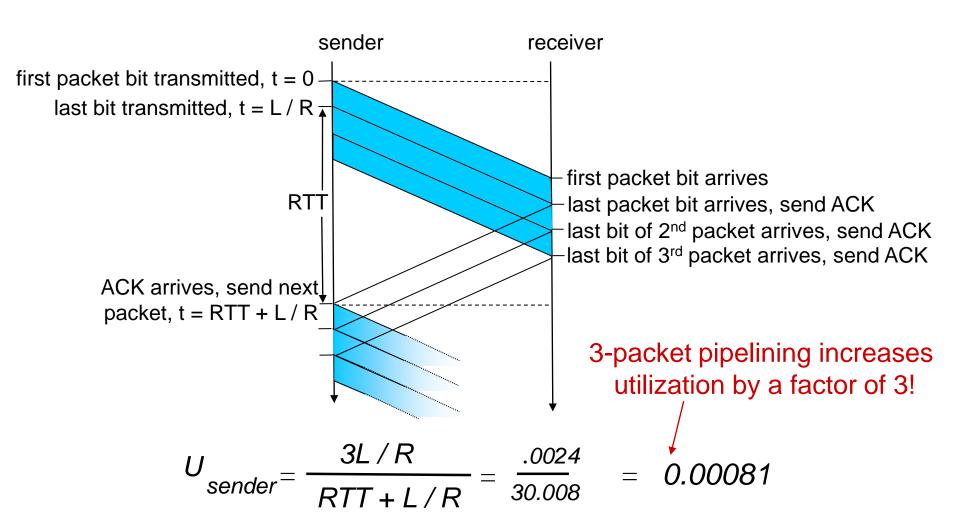
pipelining: sender allows multiple, "in-flight", yetto-be-acknowledged pkts

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



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

## Pipelining: increased utilization



# Pipelined protocols: overview

#### 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

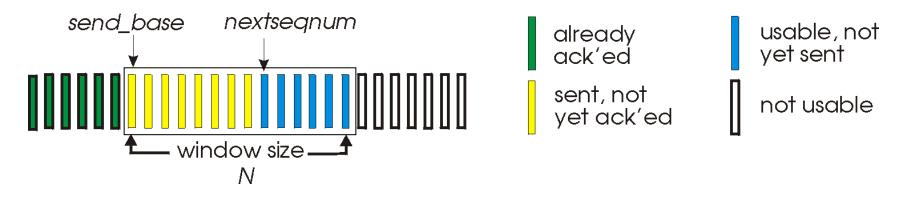
#### Selective Repeat:

- 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

### Go-Back-N: sender

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

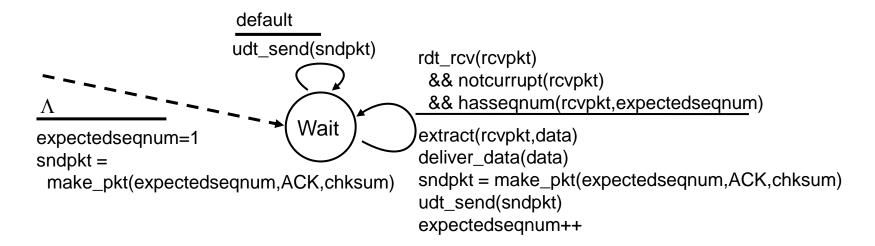


- ACK(n):ACKs all pkts up to, including seq # n "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

#### GBN: sender extended FSM

```
rdt send(data)
                       if (nextseqnum < base+N) {
                          sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
                          udt_send(sndpkt[nextseqnum])
                          if (base == nextseqnum)
                            start_timer
                          nextseqnum++
                       else
   Λ
                         refuse_data(data)
  base=1
  nextseqnum=1
                                           timeout
                                           start timer
                             Wait
                                           udt_send(sndpkt[base])
                                           udt send(sndpkt[base+1])
rdt_rcv(rcvpkt)
 && corrupt(rcvpkt)
                                           udt_send(sndpkt[nextsegnum-1])
                         rdt_rcv(rcvpkt) &&
                           notcorrupt(rcvpkt)
                         base = getacknum(rcvpkt)+1
                         If (base == nextseqnum)
                           stop_timer
                          else
                            start_timer
```

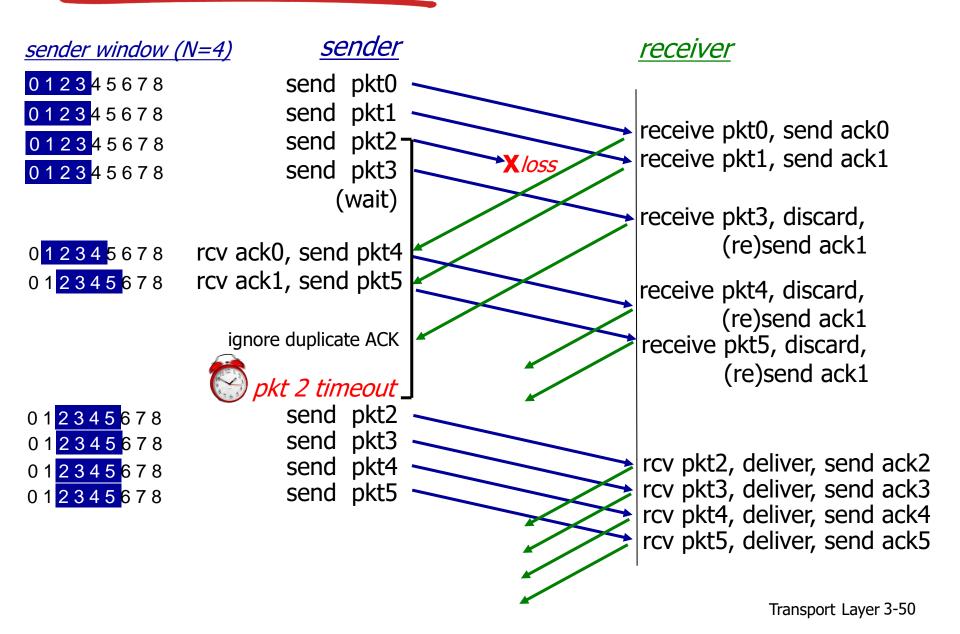
### GBN: receiver extended FSM



# ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember expectedseqnum
- out-of-order pkt:
  - discard (don't buffer): no receiver buffering!
  - re-ACK pkt with highest in-order seq #

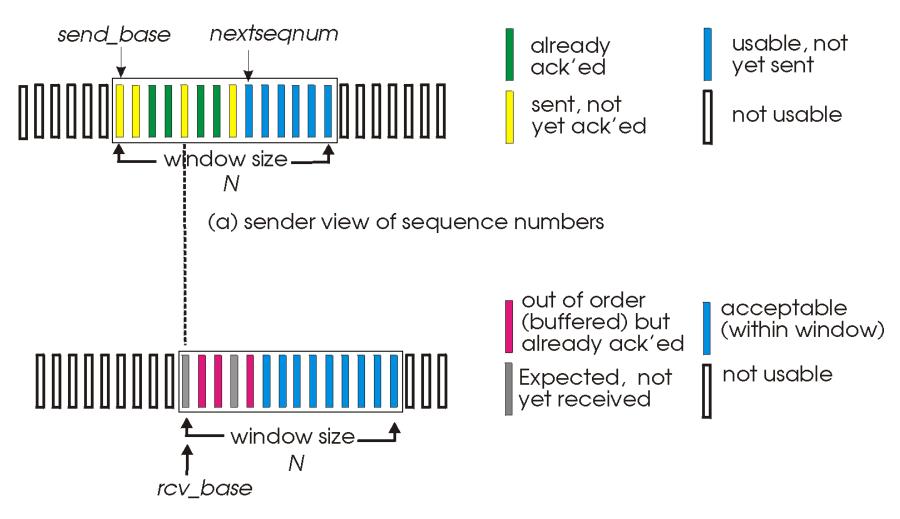
### GBN in action



### Selective repeat

- 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



(b) receiver view of sequence numbers

# Selective repeat

#### 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

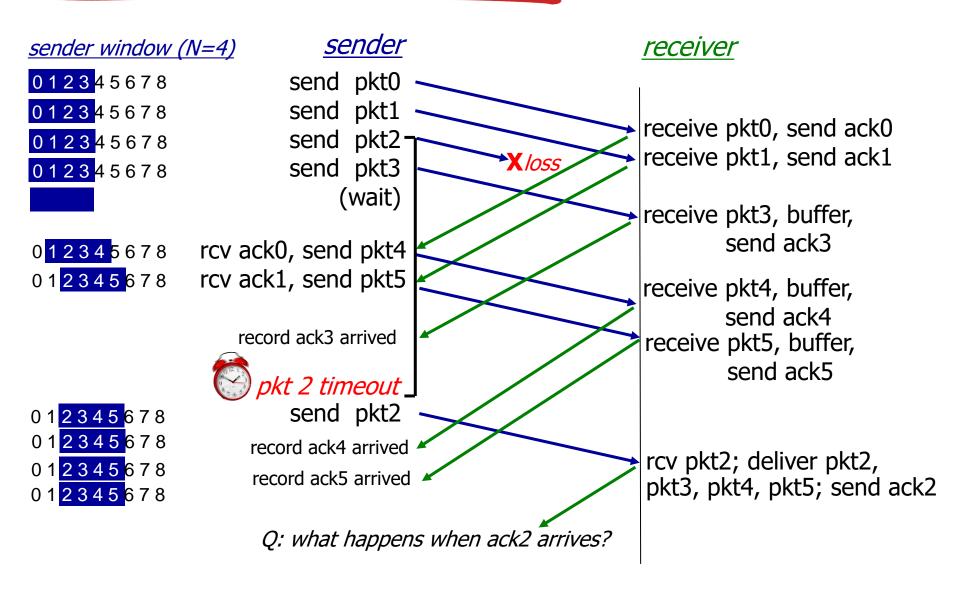
#### pkt n in [rcvbase-N,rcvbase-I]

ACK(n)

#### otherwise:

ignore

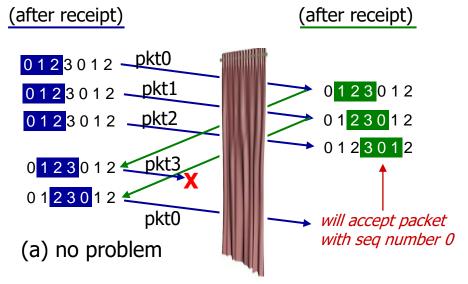
### Selective repeat in action



# Selective repeat: dilemma

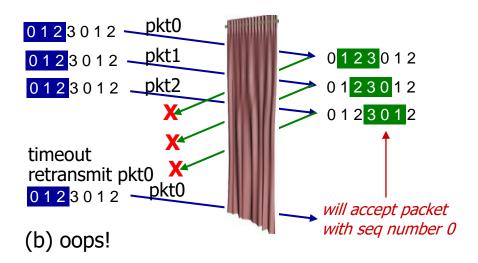
#### example:

- 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)?



sender window

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



receiver window

# 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

## TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- 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) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

### TCP segment structure

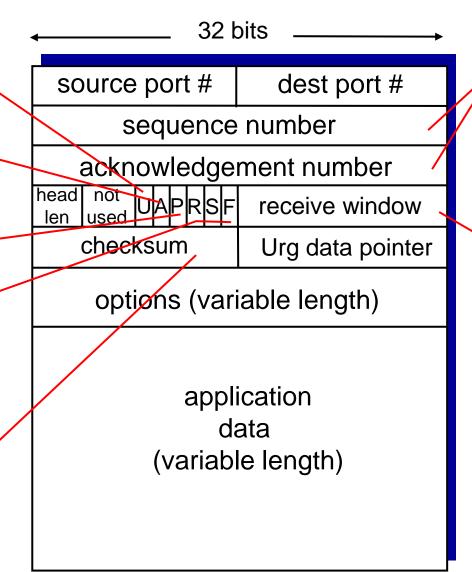
URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

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

> Internet checksum' (as in UDP)



counting by bytes of data (not segments!)

> # bytes rcvr willing to accept

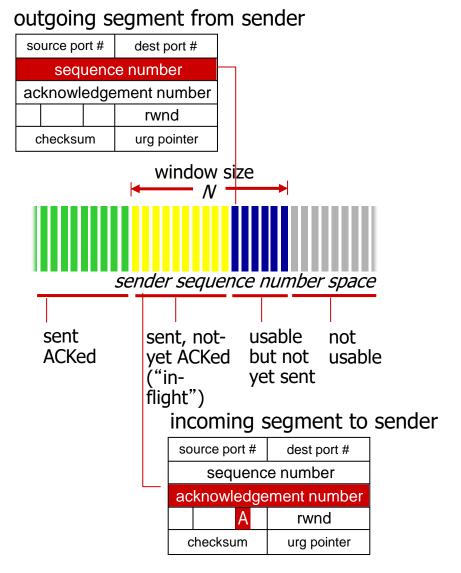
# TCP seq. numbers, ACKs

#### sequence numbers:

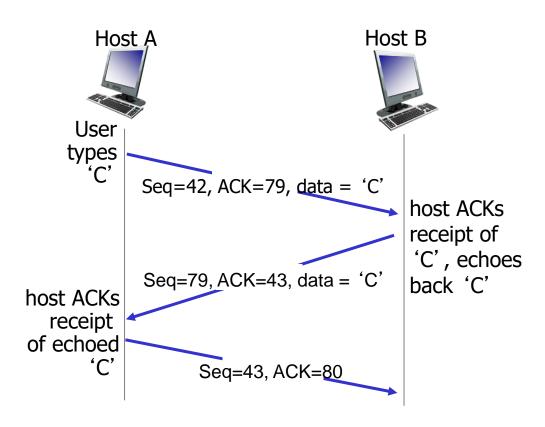
 byte stream "number" of first byte in segment's data

#### acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK
- Q: how receiver handles out-of-order segments
  - A: TCP spec doesn't say,
    - up to implementor



# TCP seq. numbers, ACKs



simple telnet scenario

# TCP round trip time, timeout

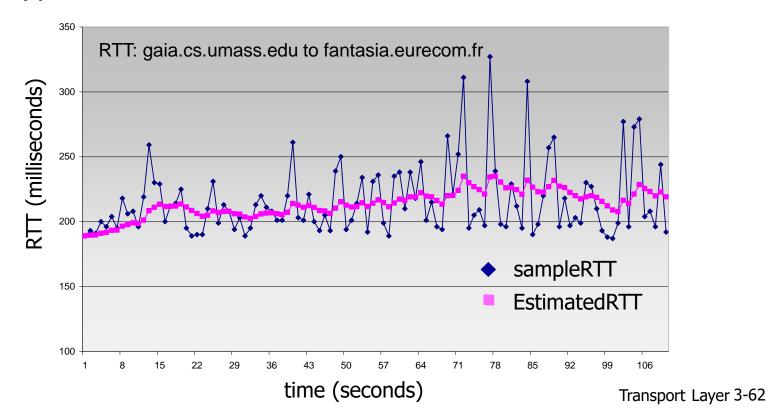
- Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

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

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



# TCP round trip time, timeout

- timeout interval: EstimatedRTT plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT| (typically, \beta = 0.25)
```

TimeoutInterval = EstimatedRTT + 4\*DevRTT



estimated RTT

safety margin"

<sup>\*</sup> Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose\_ross/interactive/

# 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

### TCP reliable data transfer

- 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

### 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

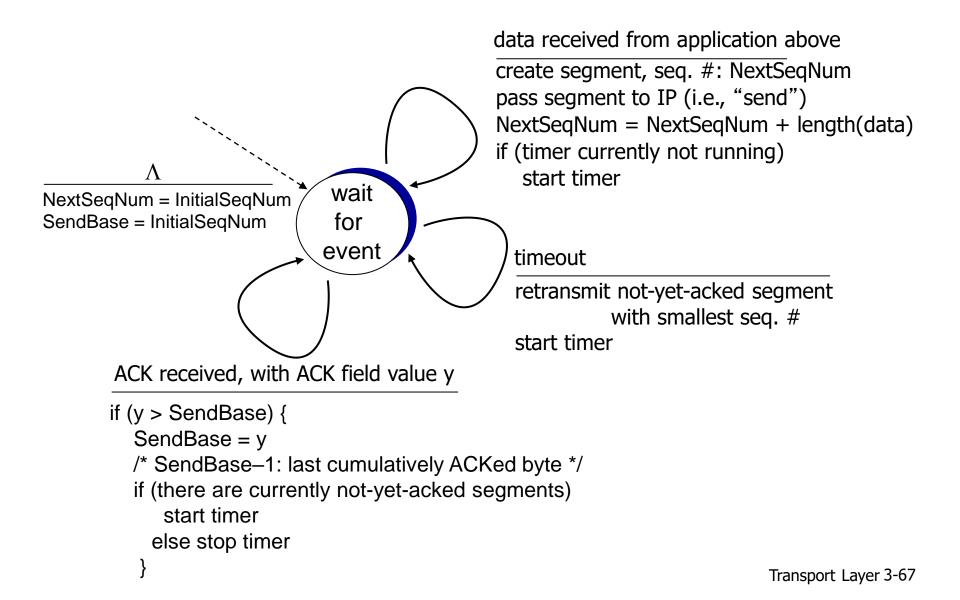
#### timeout:

- retransmit segment that caused timeout
- restart timer

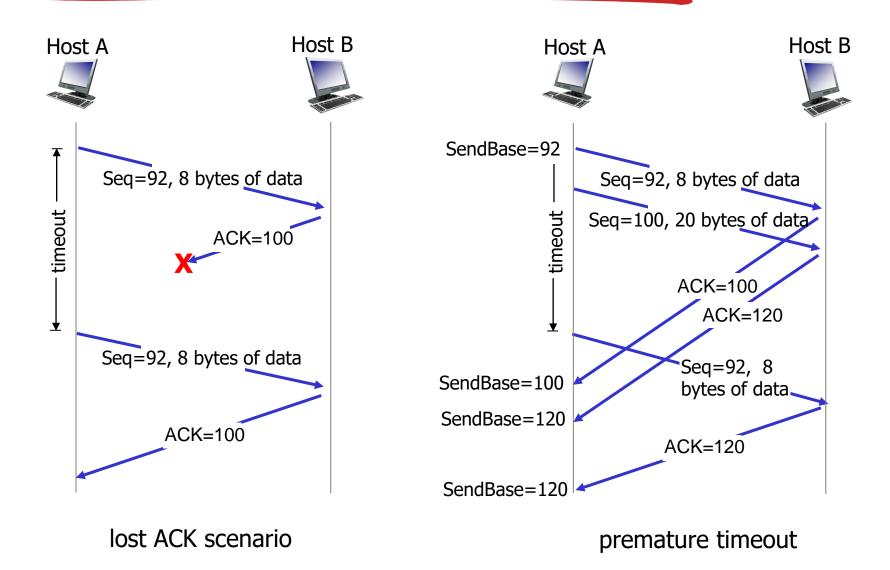
#### ack rcvd:

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

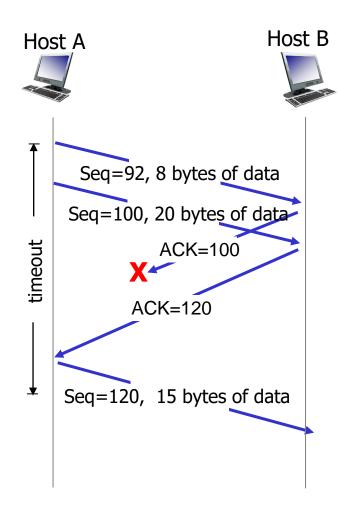
# TCP sender (simplified)



### TCP: retransmission scenarios



### TCP: retransmission scenarios



cumulative ACK

# TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	TCP receiver action
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 duplicate ACK, 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

### TCP fast retransmit

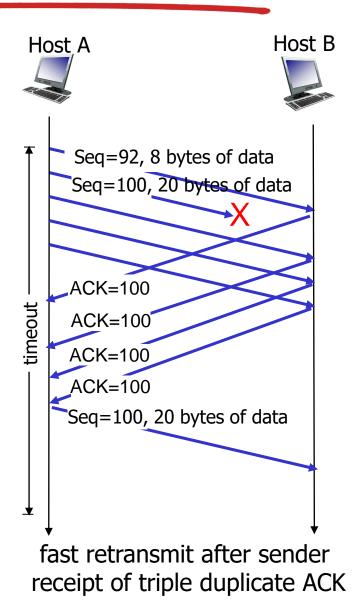
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments backto-back
  - if segment is lost, there will likely be many duplicate ACKs.

#### 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

### TCP fast retransmit



### 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

### TCP flow control

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

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

### application process application OS TCP socket receiver buffers TCP code ĬΡ code from sender

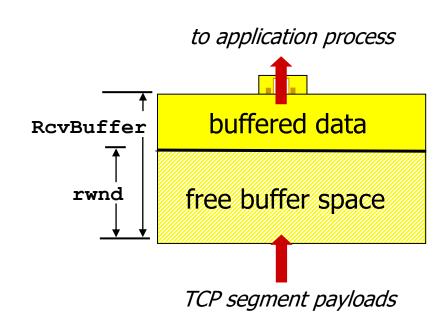
receiver protocol stack

#### 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
- sender limits amount of unacked ("in-flight") data to receiver's rwnd value
- guarantees receive buffer will not overflow



receiver-side buffering

### Chapter 3 outline

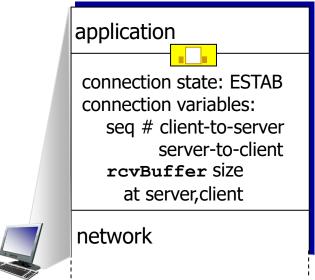
- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer

- 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

### Connection Management

before exchanging data, sender/receiver "handshake":

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



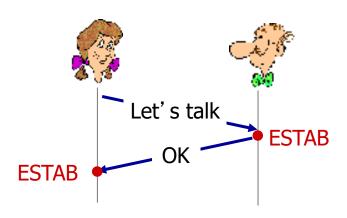
```
application
connection state: ESTAB
connection Variables:
   seq # client-to-server
          server-to-client
   rcvBuffer Size
     at server, client
network
```

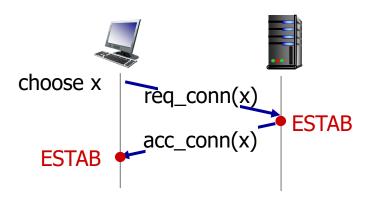
```
Socket clientSocket =
 newSocket("hostname", "port
  number");
```

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

### Agreeing to establish a connection

#### 2-way handshake:

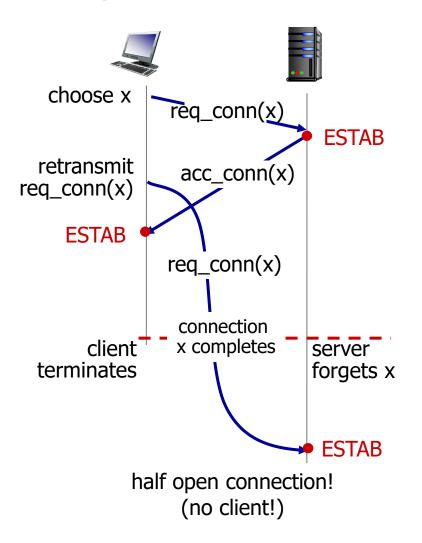


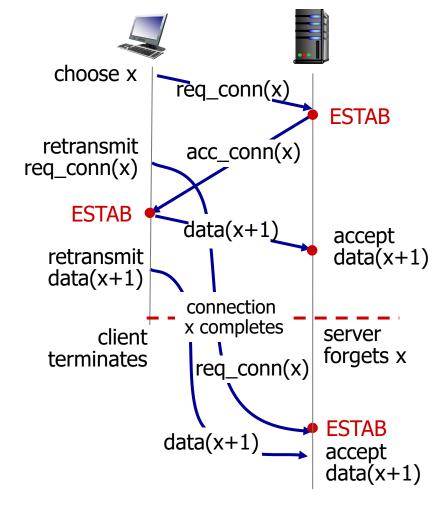


- Q: will 2-way handshake always work in network?
- variable delays
- retransmitted messages (e.g. req\_conn(x)) due to message loss
- message reordering
- can't "see" other side

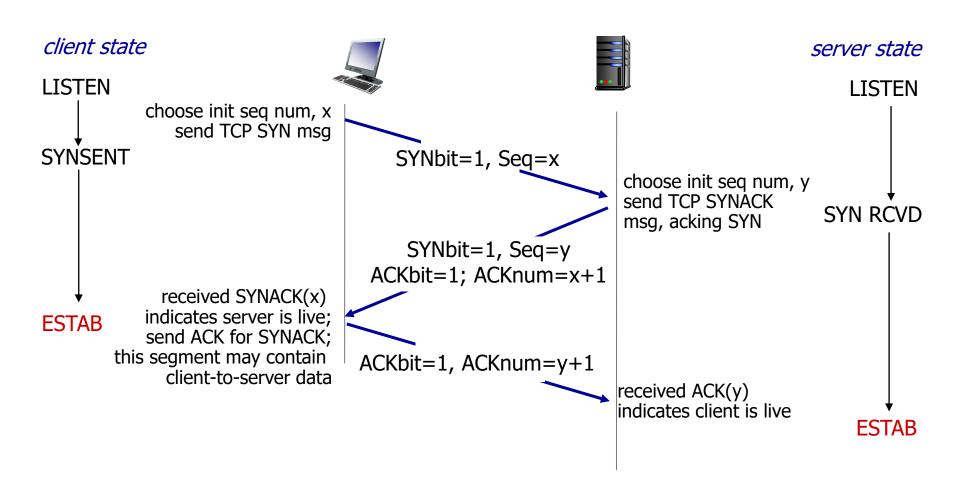
### Agreeing to establish a connection

#### 2-way handshake failure scenarios:

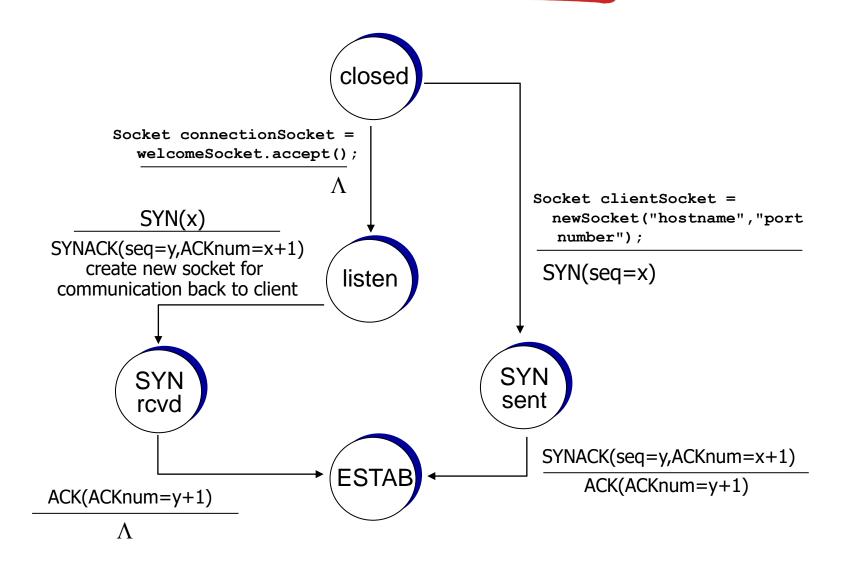




### TCP 3-way handshake



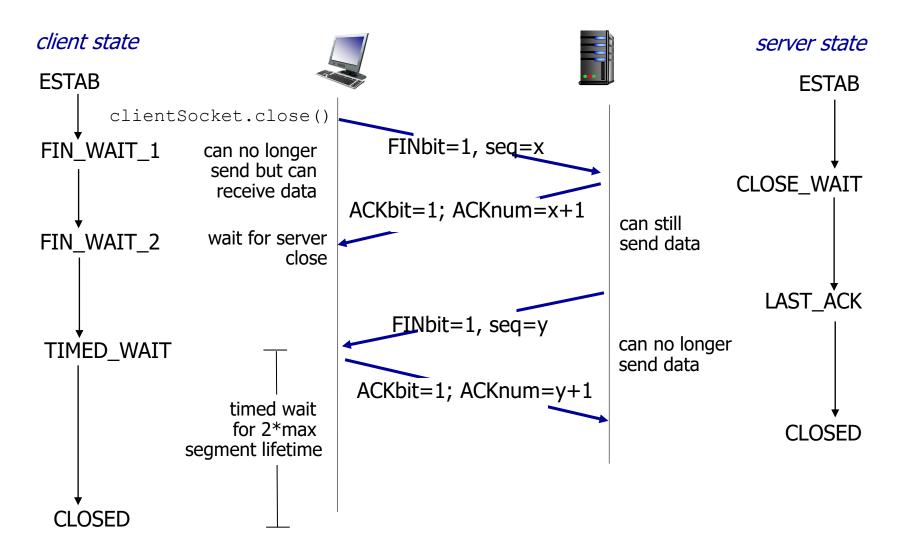
### TCP 3-way handshake: FSM



### TCP: closing a connection

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

### TCP: closing a connection



### Chapter 3 outline

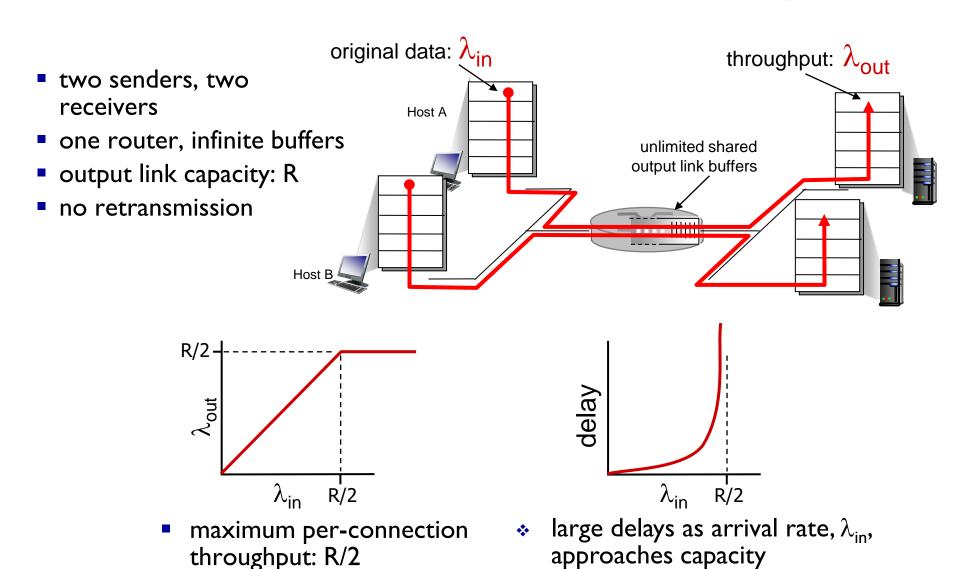
- 3.1 transport-layer services
- 3.2 multiplexing and demultiplexing
- 3.3 connectionless transport: UDP
- 3.4 principles of reliable data transfer

- 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

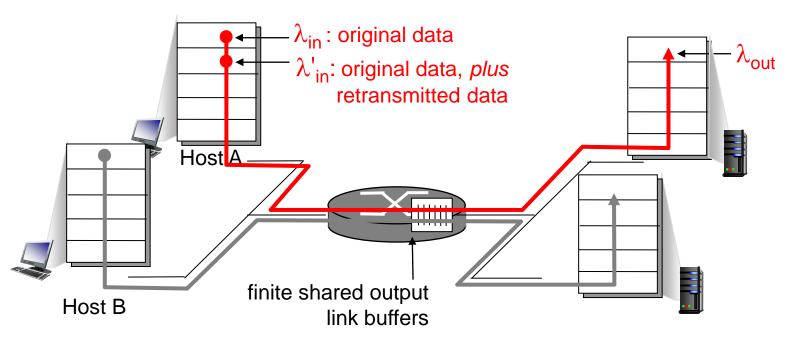
### Principles of congestion control

#### 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!

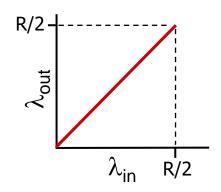


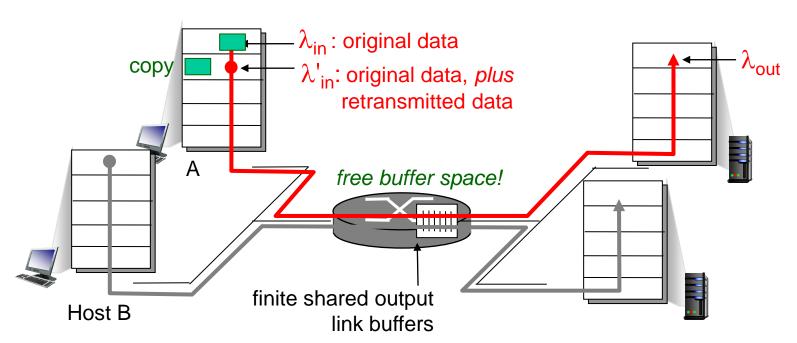
- one router, finite buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output:  $\lambda_{\text{in}}$  =  $\lambda_{\text{out}}$
  - transport-layer input includes retransmissions :  $\lambda_{in} \ge \lambda_{in}$



## idealization: perfect knowledge

sender sends only when router buffers available

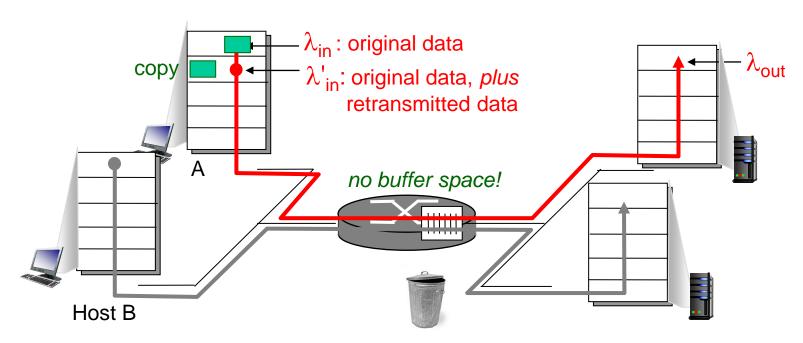




#### Idealization: known loss

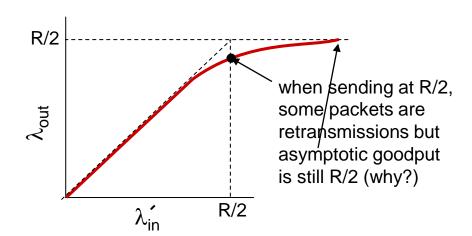
packets can be lost, dropped at router due to full buffers

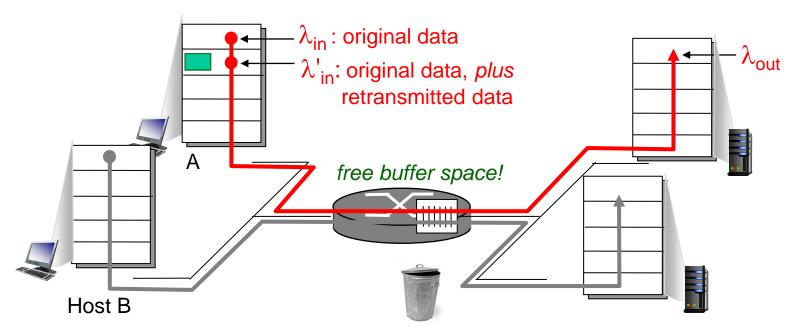
 sender only resends if packet known to be lost



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

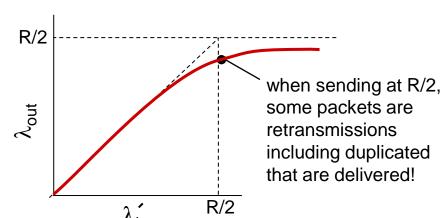
sender only resends if packet known to be lost

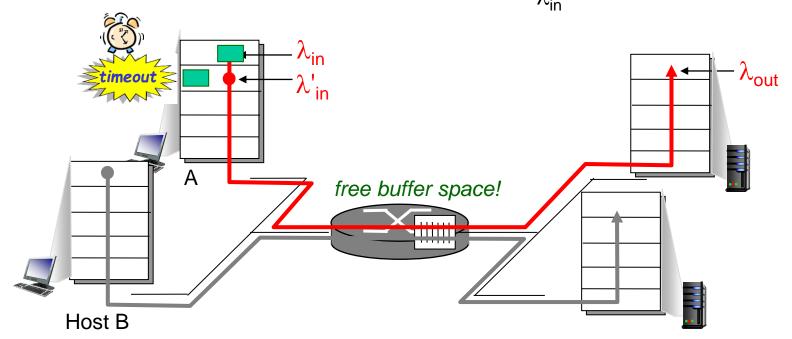




#### Realistic: duplicates

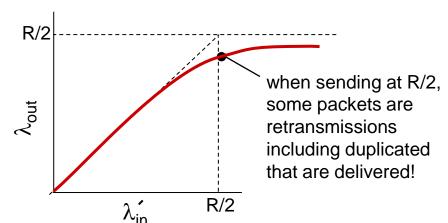
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered





#### 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



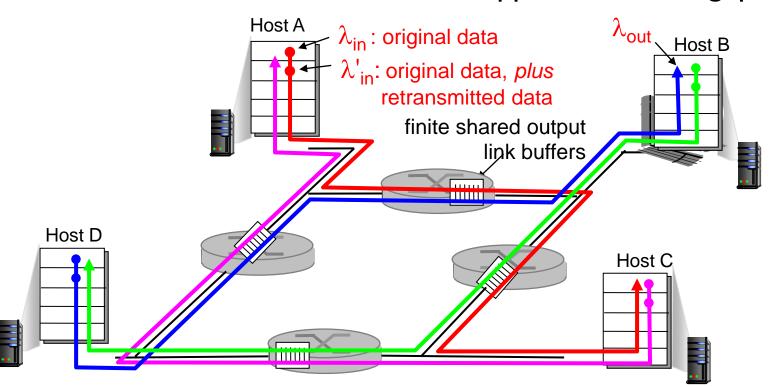
#### "costs" of congestion:

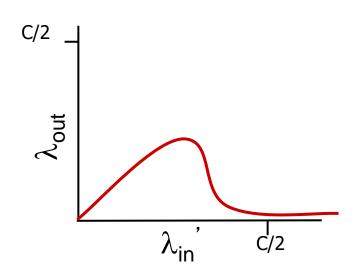
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

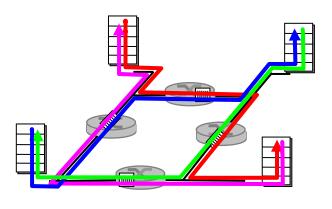
- four senders
- multihop paths
- timeout/retransmit

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

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







#### another "cost" of congestion:

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

### 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

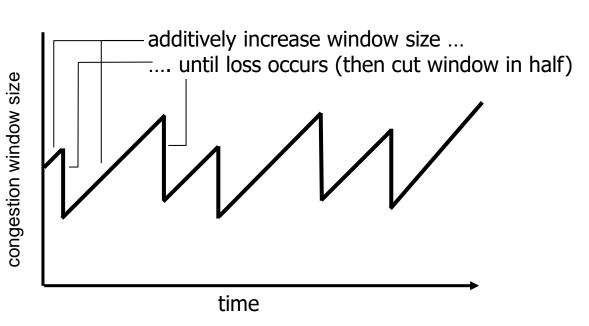
- 3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 principles of congestion control
- 3.7 TCP congestion control

# TCP congestion control: additive increase multiplicative decrease

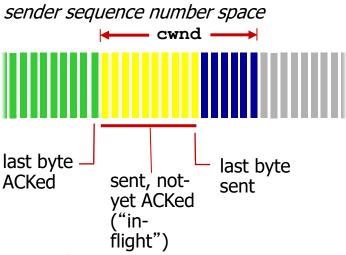
- approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase: increase cwnd by I MSS every RTT until loss detected
  - multiplicative decrease: cut cwnd in half after loss

AIMD saw tooth behavior: probing for bandwidth

cwnd: TCP sender



### TCP Congestion Control: details



sender limits transmission:

 cwnd is dynamic, function of perceived network congestion

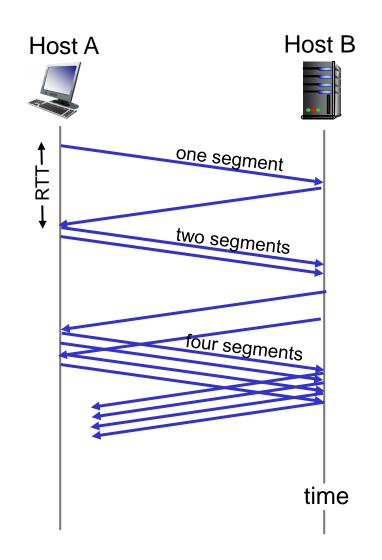
#### TCP sending rate:

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

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

### **TCP Slow Start**

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



### TCP: detecting, reacting to loss

- loss indicated by timeout:
  - cwnd set to I 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 I (timeout or 3 duplicate acks)

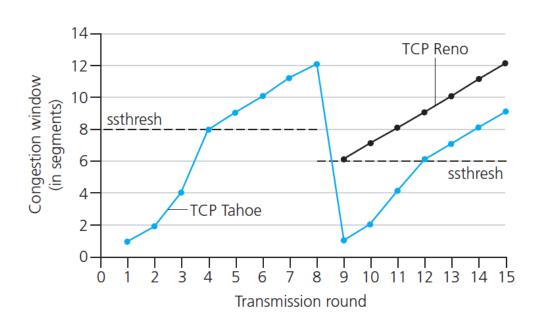
### TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

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

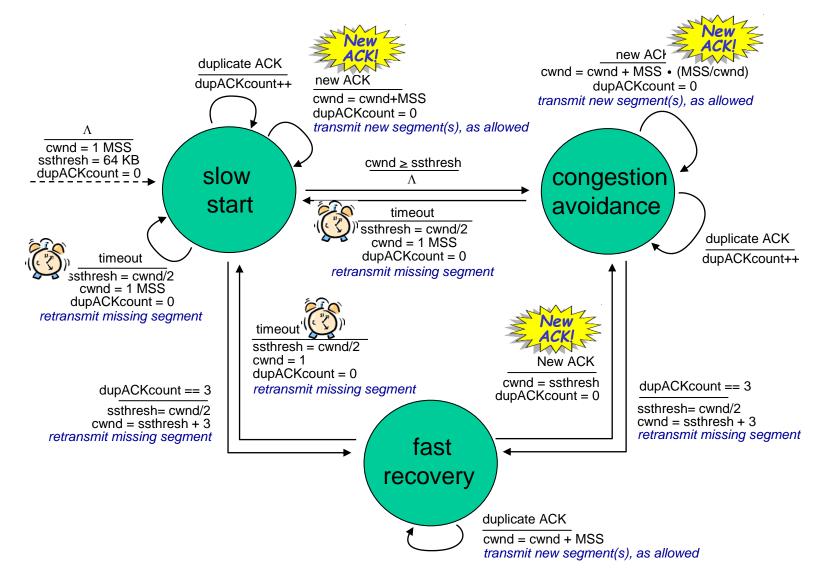


- variable ssthresh
- on loss event, ssthresh is set to 1/2 of cwnd just before loss event



<sup>\*</sup> Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose\_ross/interactive/

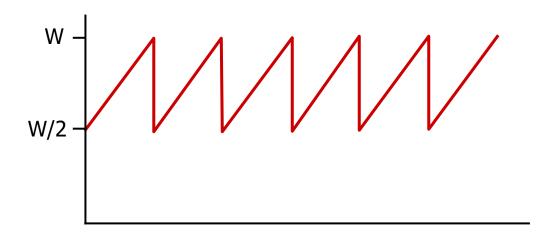
### Summary: TCP Congestion Control



### TCP throughput

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is 3/4 W
  - avg. thruput is 3/4W per RTT

avg TCP thruput = 
$$\frac{3}{4} \frac{W}{RTT}$$
 bytes/sec



### TCP Futures: TCP over "long, fat pipes"

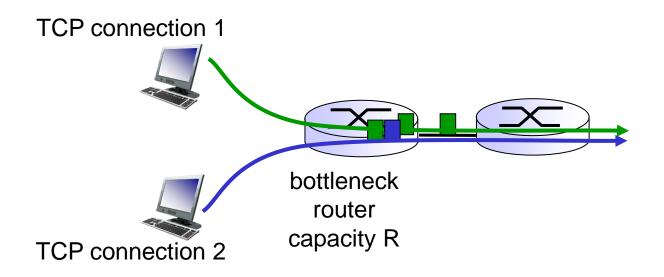
- example: I500 byte segments, I00ms RTT, wantI0 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

TCP throughput = 
$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

- ⇒ to achieve 10 Gbps throughput, need a loss rate of L =  $2 \cdot 10^{-10}$  a very small loss rate!
- new versions of TCP for high-speed

### **TCP Fairness**

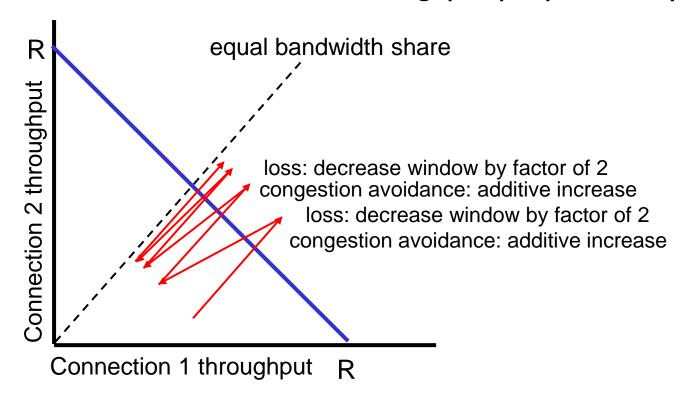
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



### Why is TCP fair?

#### two competing sessions:

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



### Fairness (more)

#### 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

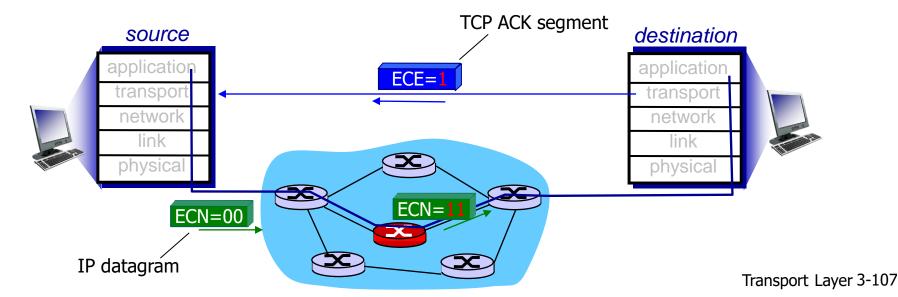
# Fairness, parallel TCP connections

- 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 I TCP, gets rate R/I0
  - new app asks for 11 TCPs, gets R/2

### Explicit Congestion Notification (ECN)

#### network-assisted congestion control:

- two bits in IP header (ToS field) marked by network router to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram) )
   sets ECE bit on receiver-to-sender ACK segment to
   notify sender of congestion



### Chapter 3: summary

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

#### next:

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