Transport Layer – Part II

Computer Networks, Winter 2020/2021
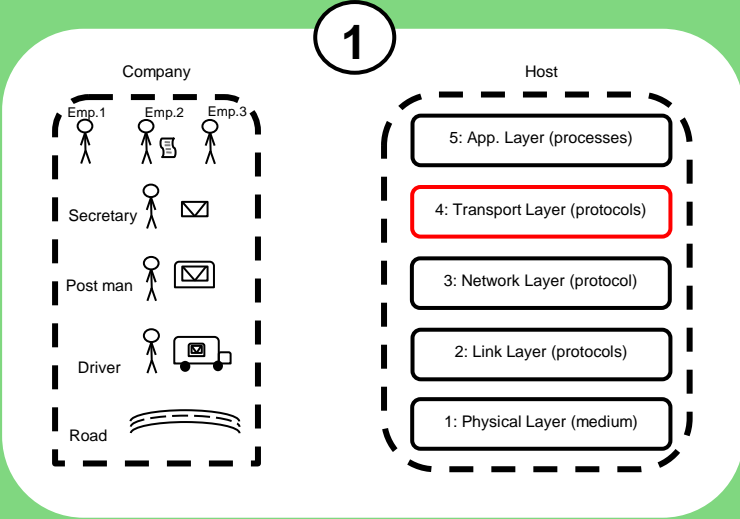
Lecturer: Prof. Xiaoming Fu

Assistants: Yachao Shao (MSc),
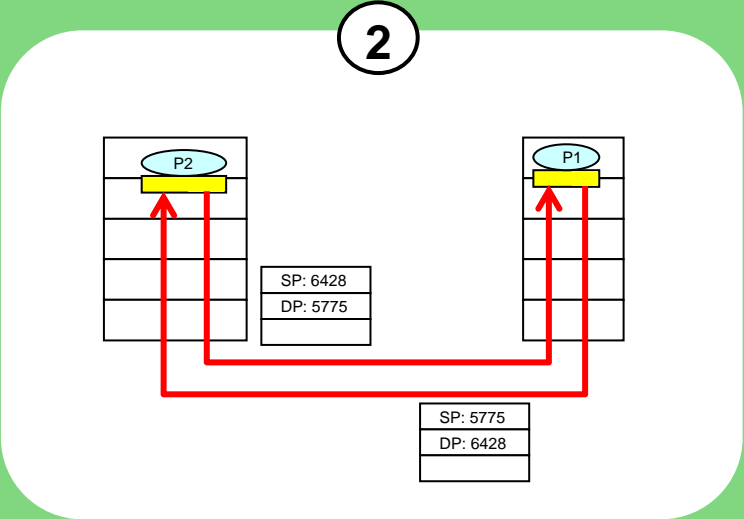
Fabian Wölk (MSc)

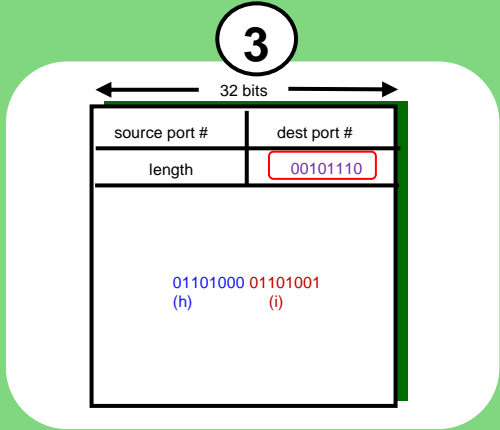# Last Session

1 — Company / Host

Company:
- Emp.1, Emp.2, Emp.3
- Secretary
- Post man
- Driver
- Road

Host:
- 5: App. Layer (processes)
- 4: Transport Layer (protocols)
- 3: Network Layer (protocol)
- 2: Link Layer (protocols)
- 1: Physical Layer (medium)

2

P2   P1

SP: 6428
DP: 5775

SP: 5775
DP: 6428

3

32 bits

source port #	dest port #
length	00101110

01101000 01101001
(h)   (i)

4

0176
OK
1234
Not OK, repeat please
1234
OK

You   Friend

5

3.0   ...   .

Chapter 3 outline

Pipelining Protocols

Go-back-N: big picture:

- Sender can have up to N unacked packets in pipeline

- Rcvr only sends cumulative acks
  - Doesn't ack packet if there's a gap

- Sender has timer for oldest unacked packet
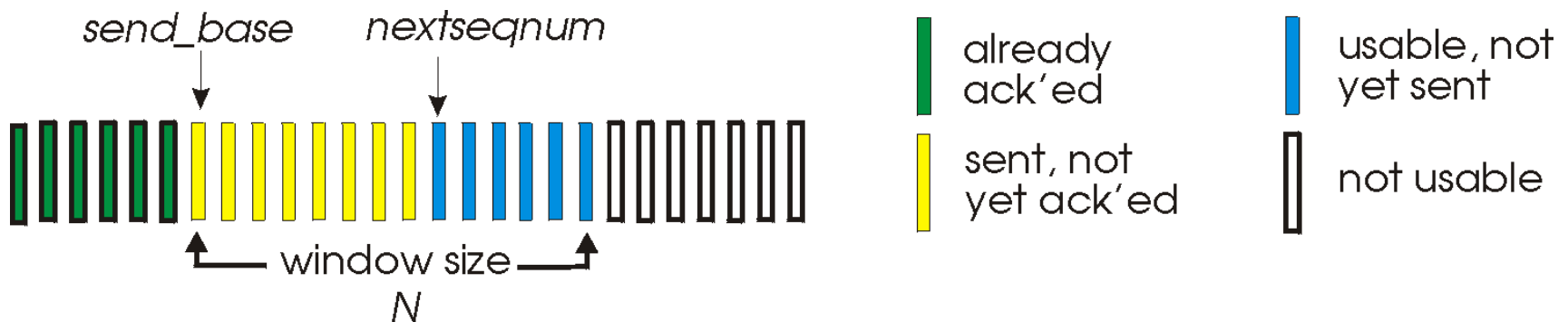 - If timer expires, retransmit all unacked packets

Selective Repeat: big pic

- Sender can have up to N unacked packets in pipeline

- Rcvr acks individual packets

- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet
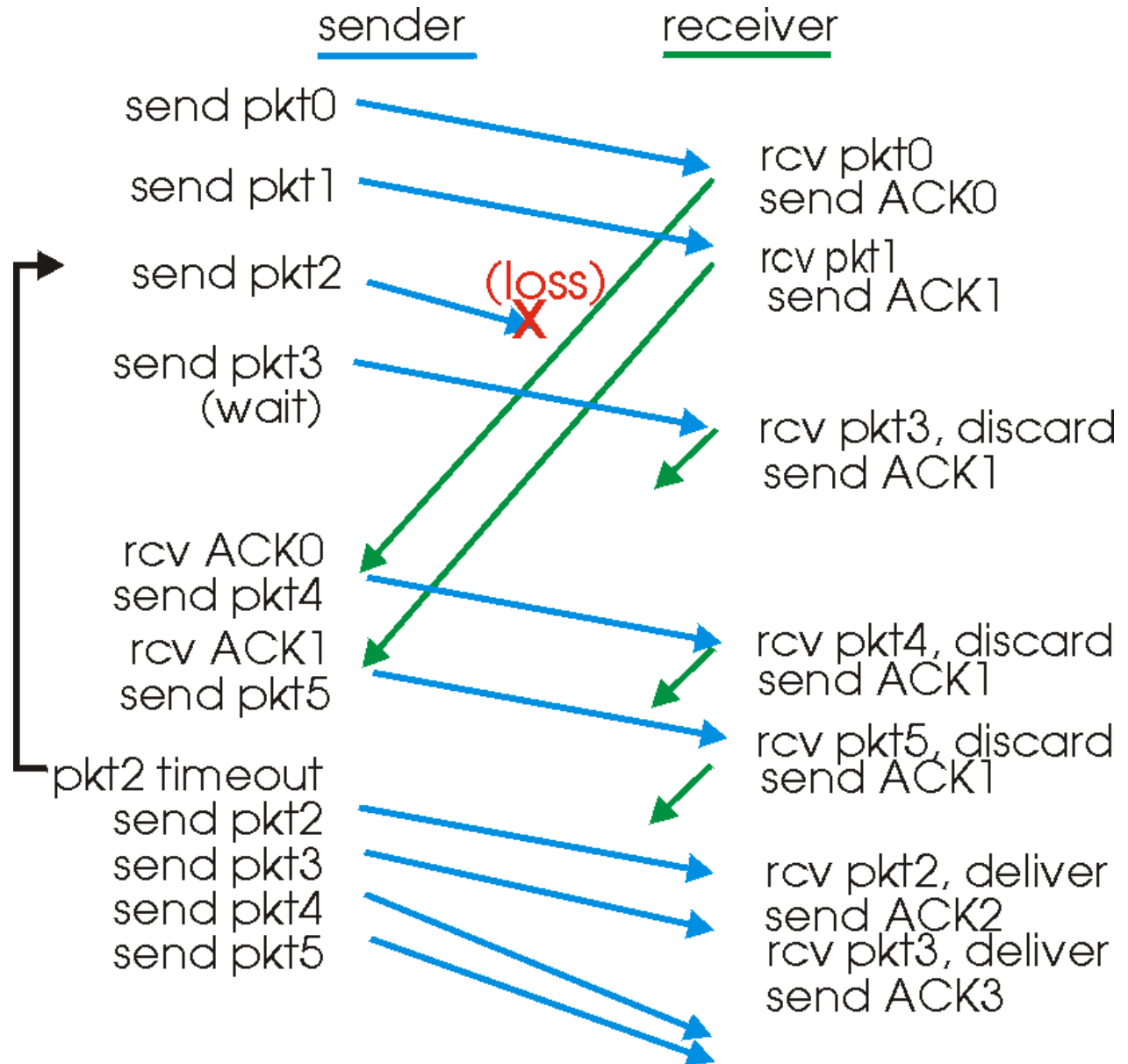
Go-Back-N

Sender:

o k-bit seq # in pkt header

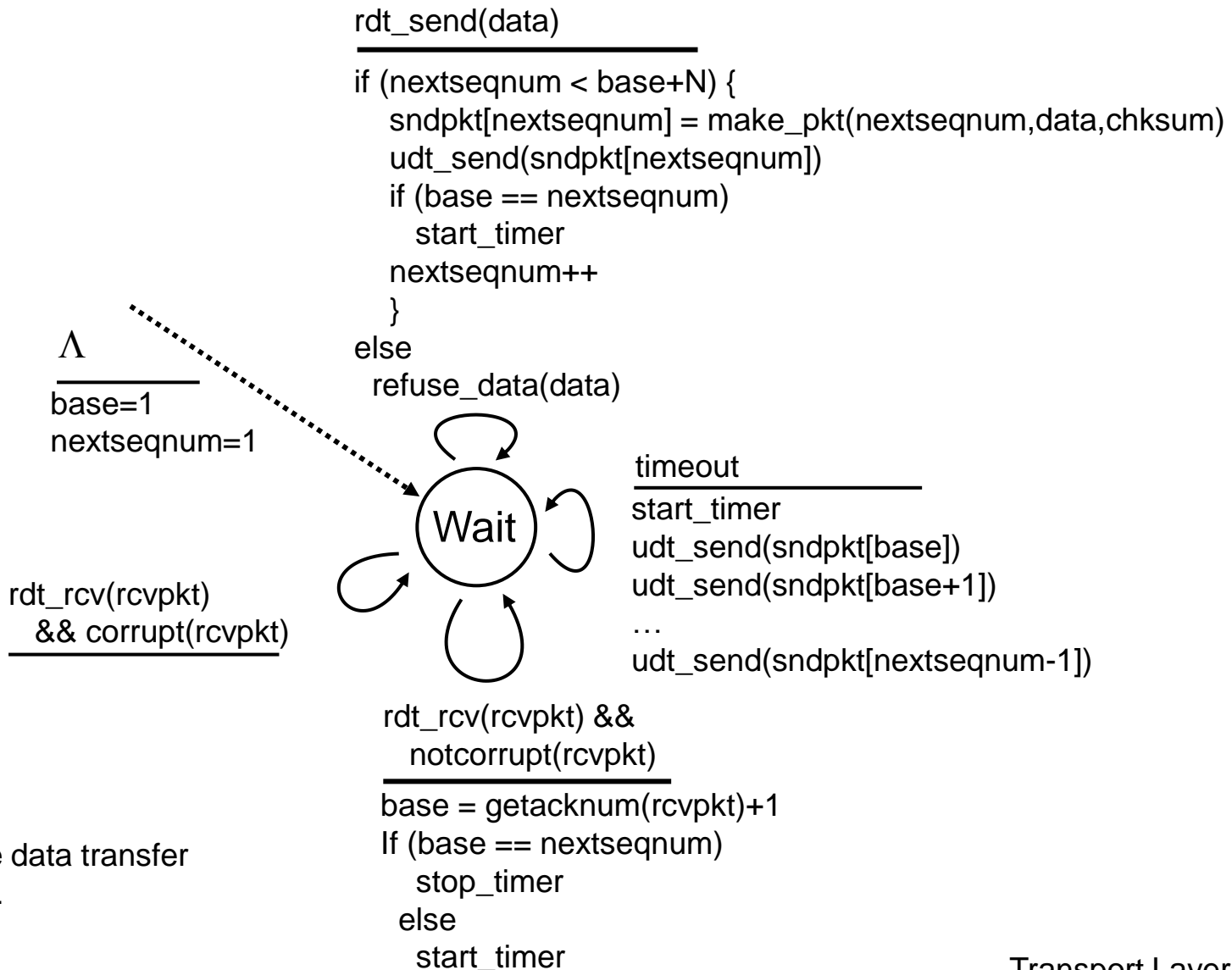o "window" of up to N, consecutive unack'ed pkts allowed

o ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"

  o may receive duplicate ACKs (see receiver)

o timer for each in-flight pkt

o *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN in action

Go back n: 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
    refuse_data(data)
```

Λ
_____
base=1
nextseqnum=1

Wait

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
_____

timeout
_____
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
_____

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

udt = unreliable data transfer
Rdt = reliable…

GBN: receiver extended FSM

default
_____
udt_send(sndpkt)

Λ
_____
expectedseqnum=1
sndpkt =
  make_pkt(expectedseqnum,ACK,chksum)

Wait

rdt_rcv(rcvpkt)
  && notcurrupt(rcvpkt)
  && hasseqnum(rcvpkt,expectedseqnum)
_____
extract(rcvpkt,data)
deliver_data(data)
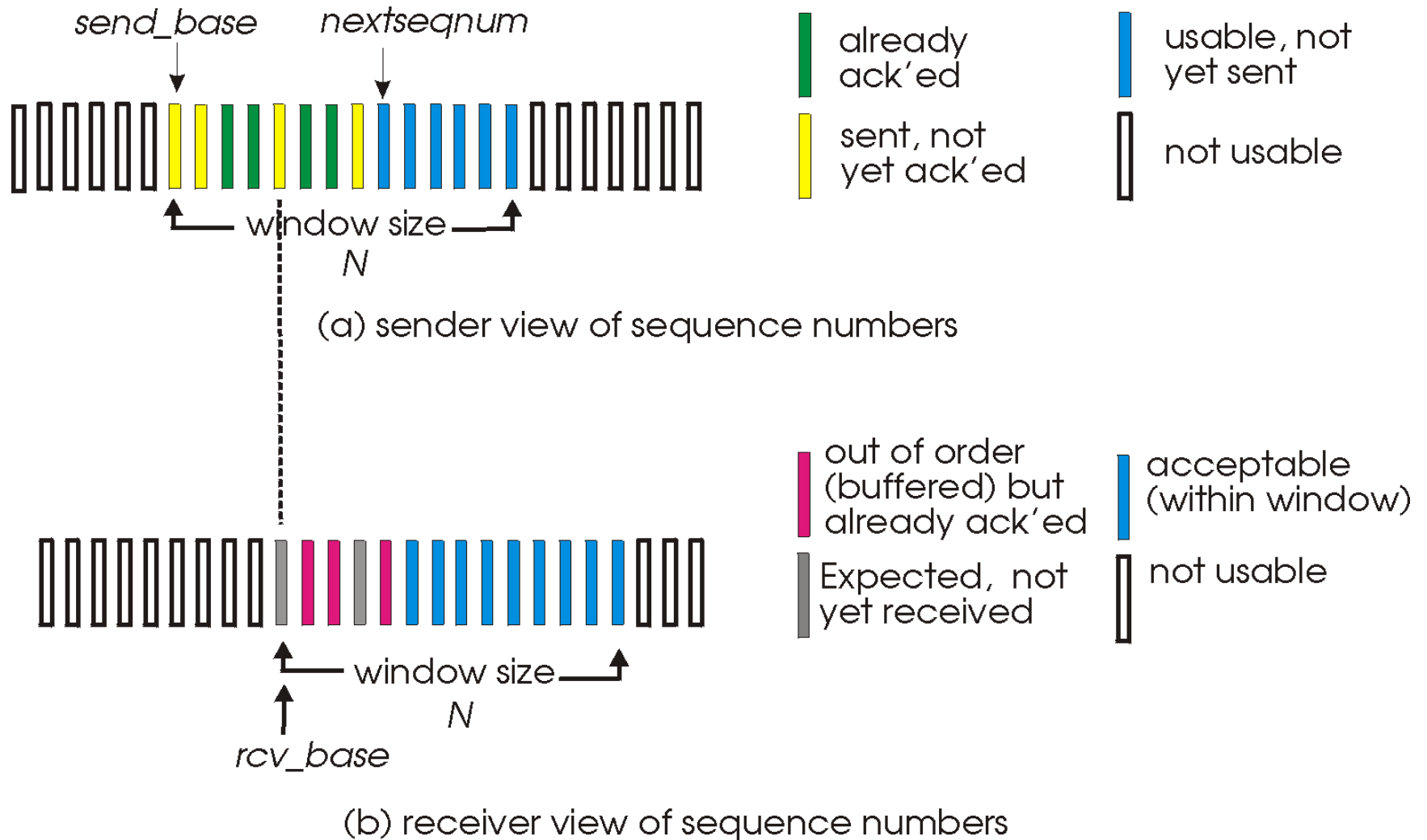sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++
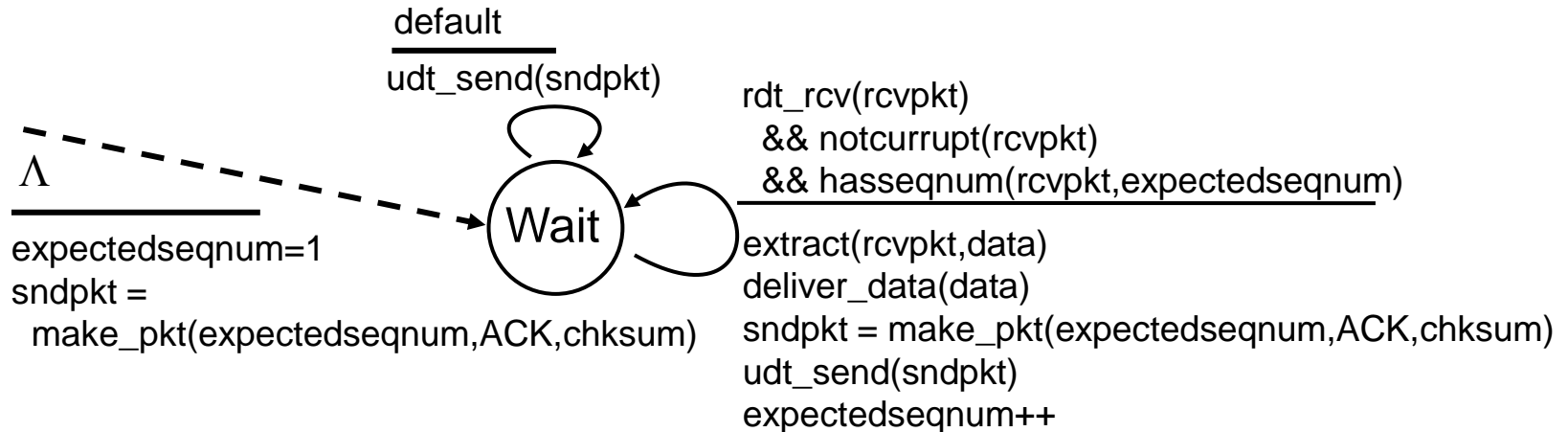
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `expectedseqnum`

o out-of-order pkt:

- discard (don't buffer) -> no receiver buffering!
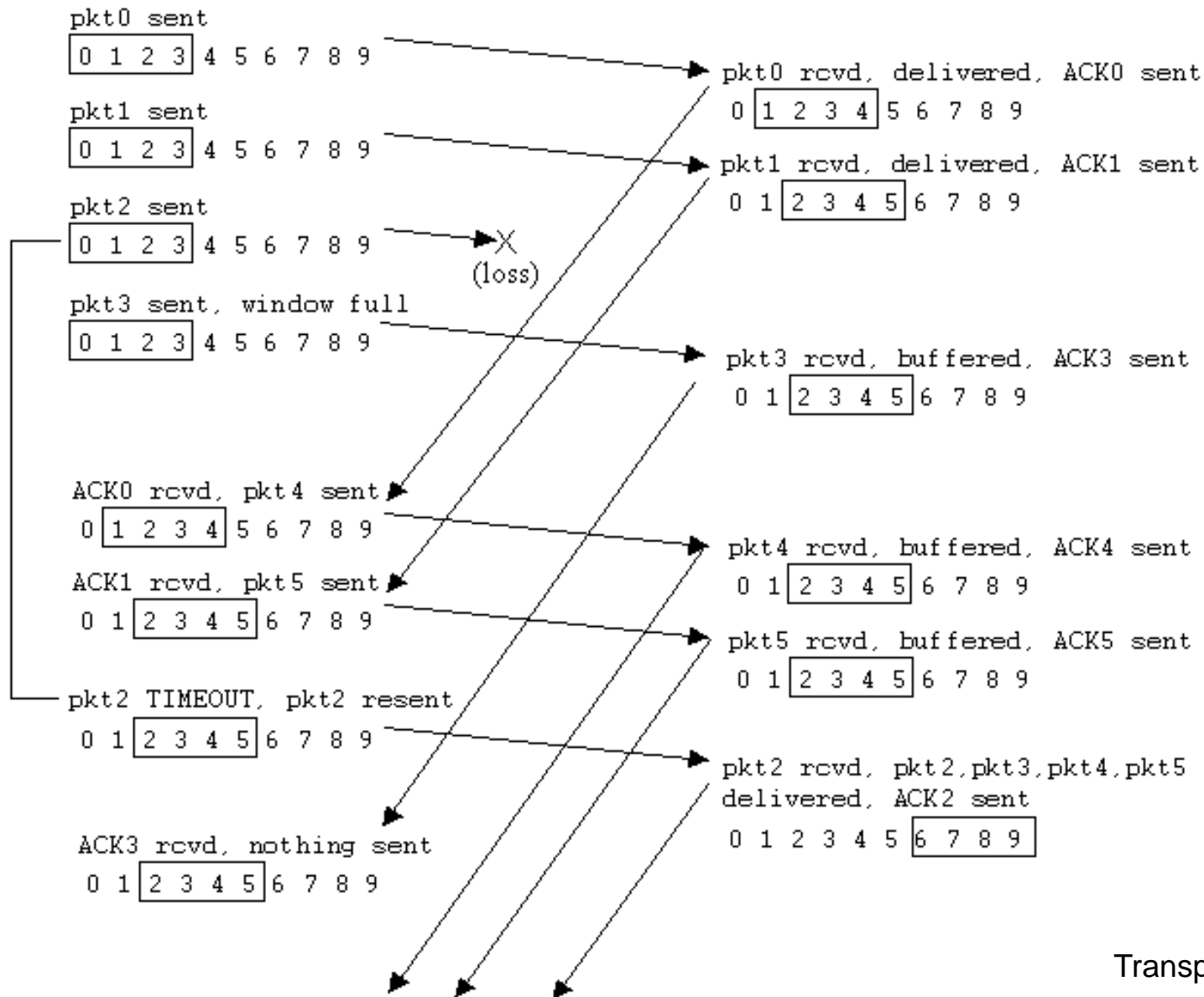- Re-ACK pkt with highest in-order seq #

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

   ○ again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows

send_base *nextseqnum*

▮ already ack'ed		▮ usable, not yet sent
▮ sent, not yet ack'ed		▯ not usable

window size — N

(a) sender view of sequence numbers

▮ out of order (buffered) but already ack'ed		▮ acceptable (within window)
▮ Expected, not yet received		▯ not usable

window size — N

rcv_base

(b) receiver view of sequence numbers

Selective repeat in action

```
pkt0 sent
0 1 2 3 4 5 6 7 8 9
                                        pkt0 rcvd, delivered, ACK0 sent
pkt1 sent                               0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
                                        pkt1 rcvd, delivered, ACK1 sent
pkt2 sent                               0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9      X
                        (loss)
pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9                      pkt3 rcvd, buffered, ACK3 sent
                                        0 1 2 3 4 5 6 7 8 9


ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9                      pkt4 rcvd, buffered, ACK4 sent
                                        0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9                      pkt5 rcvd, buffered, ACK5 sent
                                        0 1 2 3 4 5 6 7 8 9
pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9                      pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
                                        delivered, ACK2 sent
                                        0 1 2 3 4 5 6 7 8 9
ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9
```

Selective repeat

data from above :

o if next available seq # in window, send pkt

timeout(n):

o resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

o mark pkt n as received

o if n smallest unACKed pkt, advance window base to next unACKed seq #

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

o send ACK(n)

o out-of-order: buffer

o in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
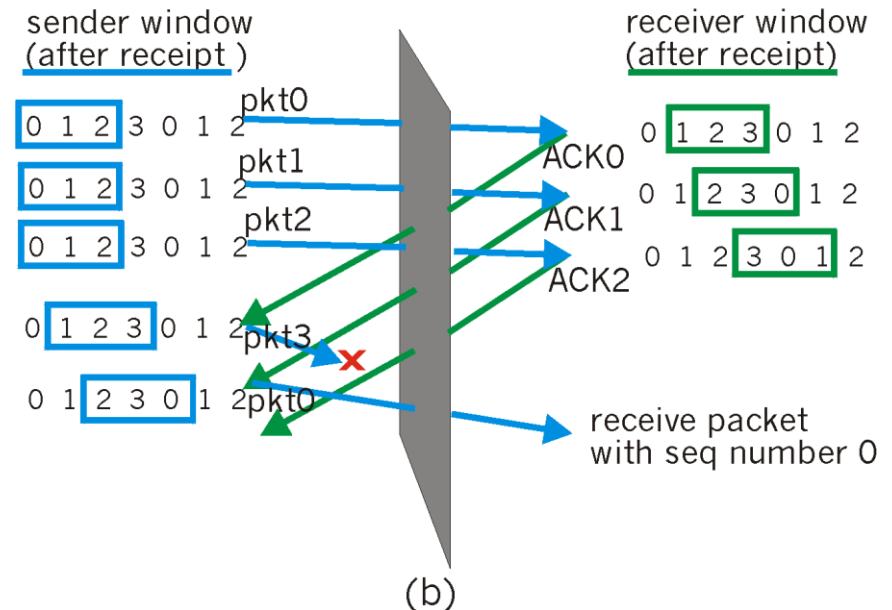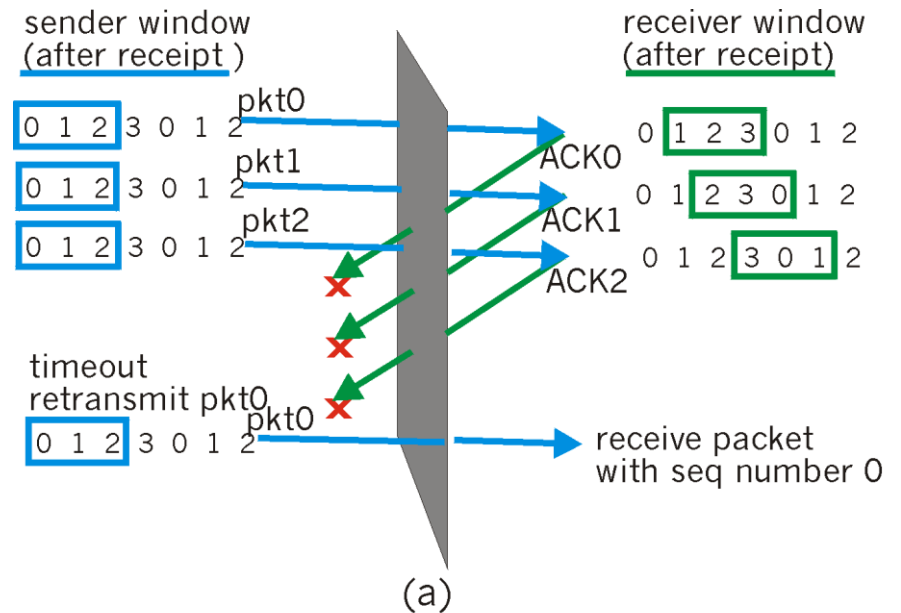
pkt n in [rcvbase-N,rcvbase-1]

o ACK(n)

otherwise:

o ignore

Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Notice: Window size should be not too large, e.g. ½ of sequence range.

(a)

(b)

Chapter 3 outline

TCP: Overview

o point-to-point:

- o one sender, one receiver

o reliable, in-order *byte stream:*

- o no "message boundaries"

o pipelined:

- o TCP congestion and flow control set window size

o *send & receive buffers*

o full duplex data:

- o bi-directional data flow in same connection
- o MSS: maximum segment size

o connection-oriented:

- o handshaking (exchange of control msgs) init's sender, receiver state before data exchange

o flow controlled:

- o sender will not overwhelm receiver

application
writes data

application
reads data

socket
door

socket
door

TCP
send buffer

TCP
receive buffer

segment

TCP segment structure

32 bits

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)

source port #	dest port #
sequence number	
acknowledgement number	
head len / not used / UAPRSF	Receive window
checksum	Urg data pnter
Options (variable length)	
application data (variable length)	

counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept

Transport Layer   3-16

TCP seq. #'s and ACKs

Seq. #'s:
- byte stream "number" of first byte in segment's data

ACKs:
- 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

Host A   Host B

User types 'C'

Seq=42, ACK=79, data = 'C'

→ host ACKs receipt of 'C', echoes back 'C'

Seq=79, ACK=43, data = 'C'

host ACKs receipt of echoed 'C'

Seq=43, ACK=80

time

simple telnet scenario
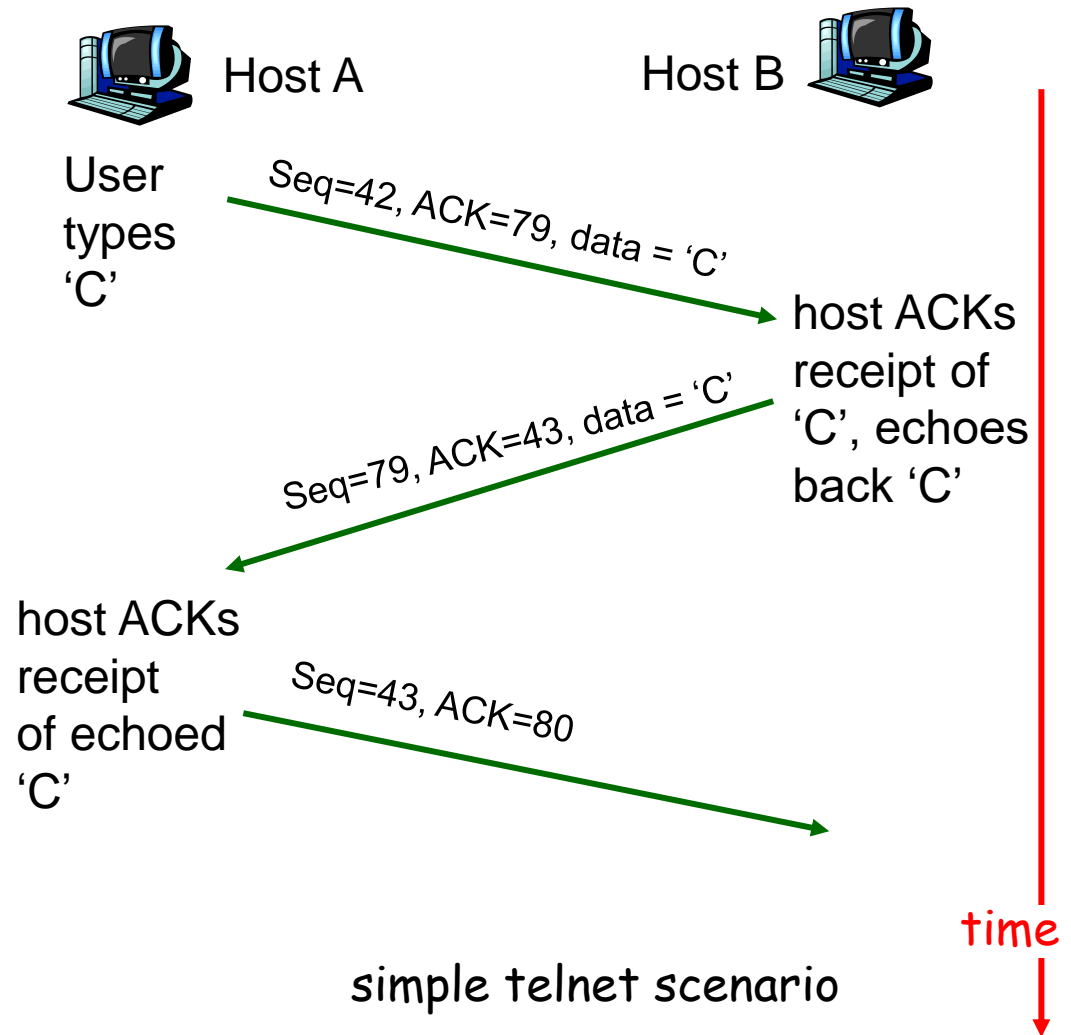
TCP Round Trip Time and Timeout

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 and Timeout

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

- ❑ Exponential weighted moving average
- ❑ influence of past sample decreases exponentially fast
- ❑ typical value: α **=** 0.125

Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr
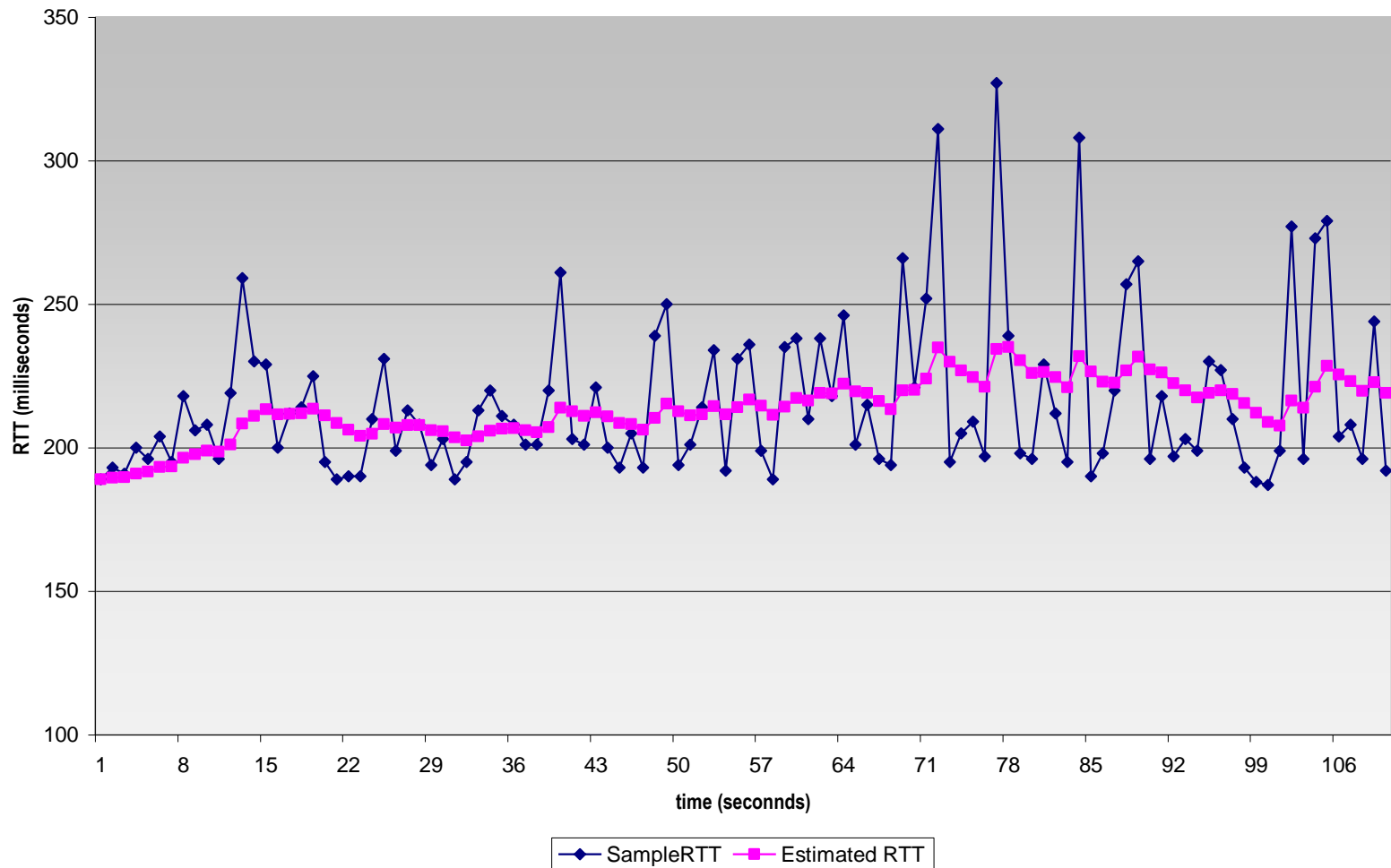
TCP Round Trip Time and Timeout

Setting the timeout

- **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT ->** larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, β = 0.25)

Then set timeout interval:

TimeoutInterval = EstimatedRTT + 4*DevRTT

Chapter 3 outline

TCP reliable data transfer

o TCP creates rdt service on top of IP's unreliable service

o Pipelined segments

o Cumulative acks

o TCP uses single retransmission timer

o Retransmissions are triggered by:

  o timeout events

  o duplicate acks

o Initially consider simplified TCP sender:

  o ignore duplicate acks

  o 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 acknowledges previously unacked segments

  - update what is known to be acked

  - start timer if there are outstanding segments

TCP: retransmission scenarios

Host A   Host B

Seq=92, 8 bytes data

ACK=100

X
loss

timeout

Seq=92, 8 bytes data

ACK=100

SendBase
= 100

time   **lost ACK scenario**

Host A   Host B

Seq=92, 8 bytes data

Seq=100, 20 bytes data

ACK=100

ACK=120

Seq=92, 8 bytes data

Seq=92 timeout

Seq=92 timeout

Sendbase
= 100

SendBase
= 120

ACK=120

SendBase
= 120

time   **premature timeout**

TCP retransmission scenarios (more)

Host A   Host B

timeout

Seq=92, 8 bytes data

ACK=100

Seq=100, 20 bytes data

X

loss

ACK=120

SendBase
= 120
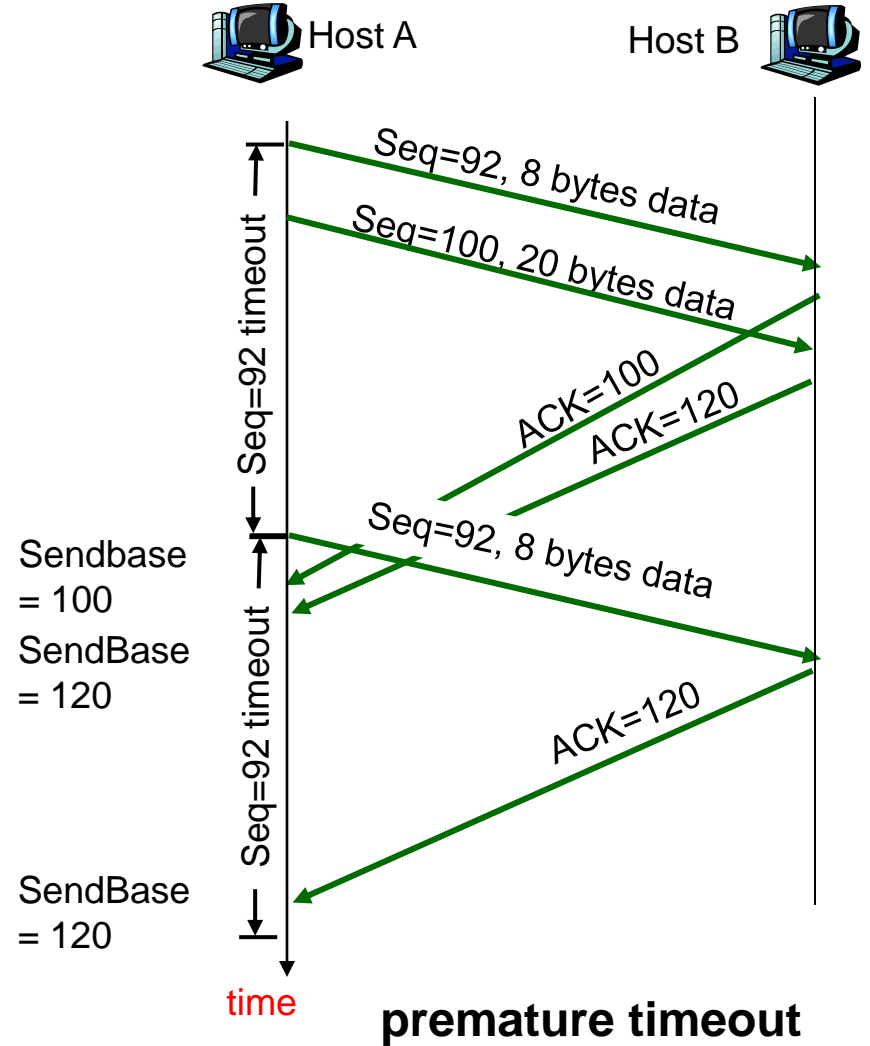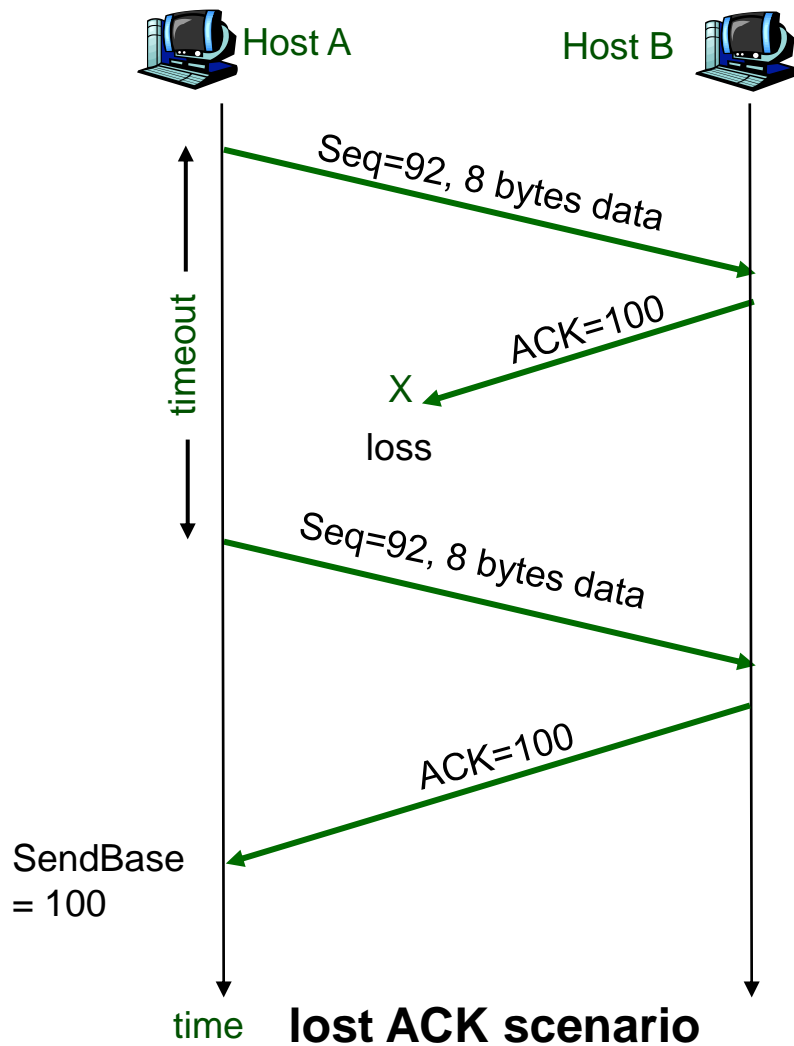
time
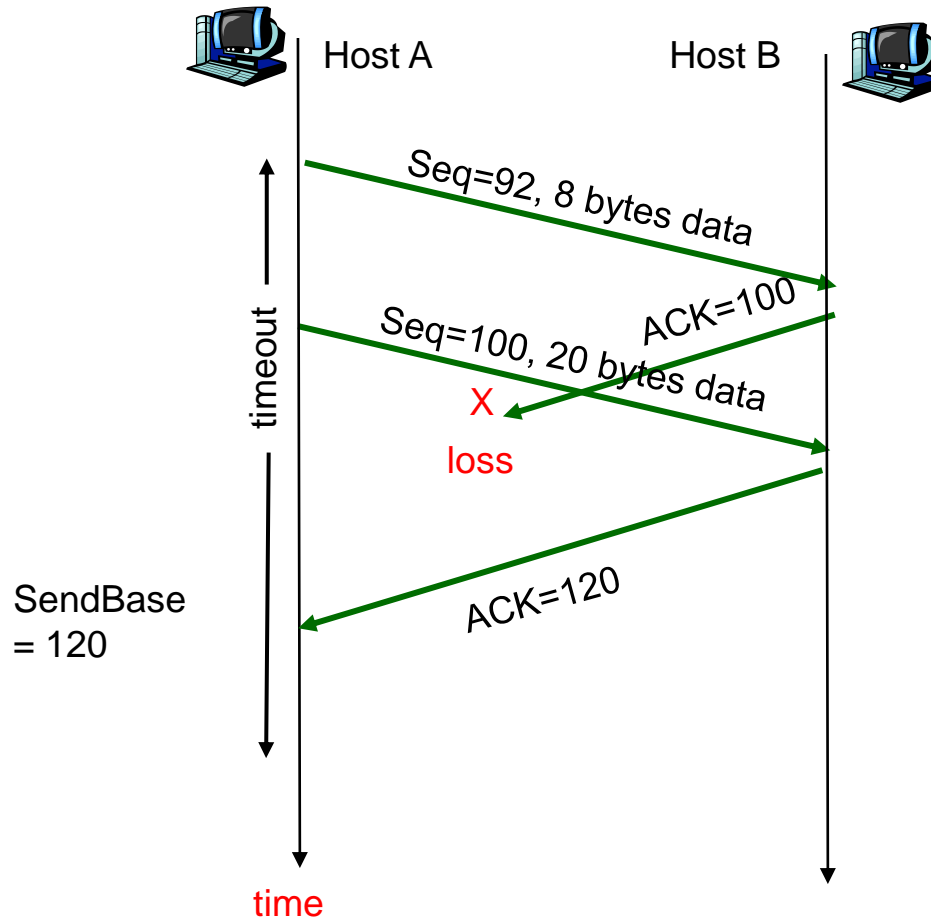
Cumulative ACK scenario

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

Fast Retransmit

o Time-out period often relatively long:

    o long delay before resending lost packet

o Detect lost segments via duplicate ACKs.

    o Sender often sends many segments back-to-back

    o If segment is lost, there will likely be many duplicate ACKs.

o If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:

    o fast retransmit: resend segment before timer expires

Host A   Host B

timeout

X

resend 2ⁿᵈ segment

time
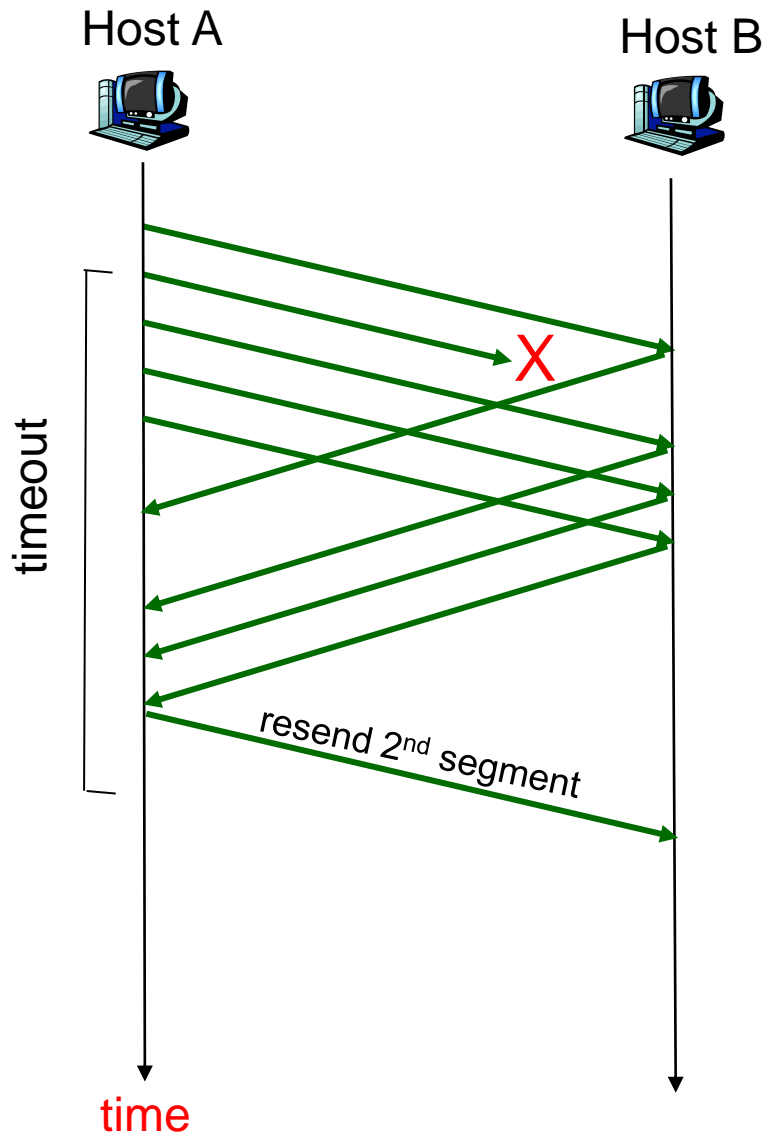
Figure 3.37 Resending a segment after triple duplicate ACK

Chapter 3 outline

Analogy: Flow Control

o **Assumptions:**

- o Secretary delivers mail at rate of 4 letters/h

- o Employee Bill processes mail at 1 letter/h.

- o Table has place for 10 letters, more will drop on floor.

o After half a day his table overflows, letters get lost.

o Sender needs to decrease sending rate.

time	Mail read	Mail on table
9:00	0	4
10:00	1	7
11:00	2	10
12:00	3	**13 !**

Bill
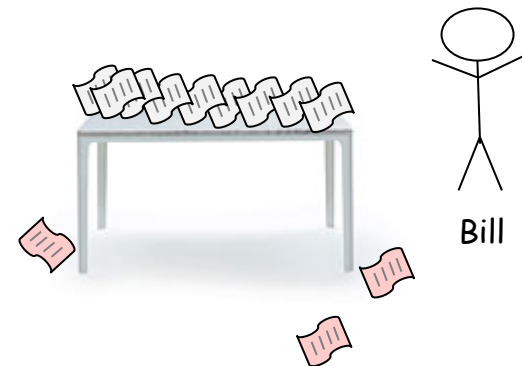
TCP Flow Control

o receive side of TCP connection has a receive buffer:
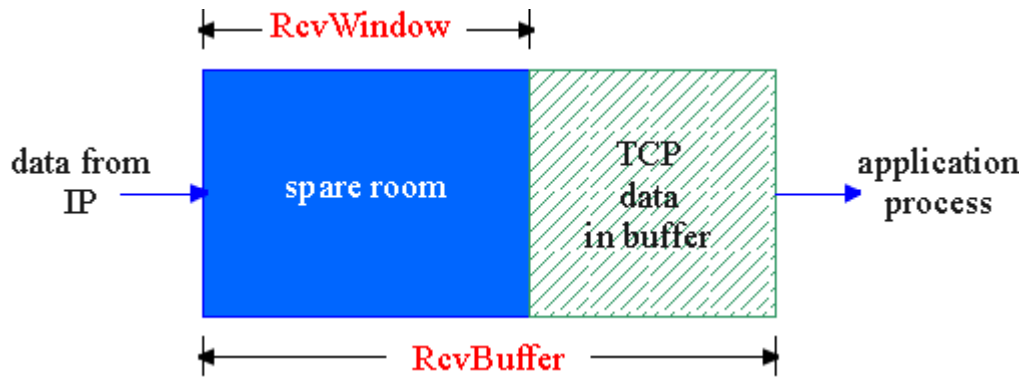
o app process may be slow at reading from buffer

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

TCP Flow control: how it works
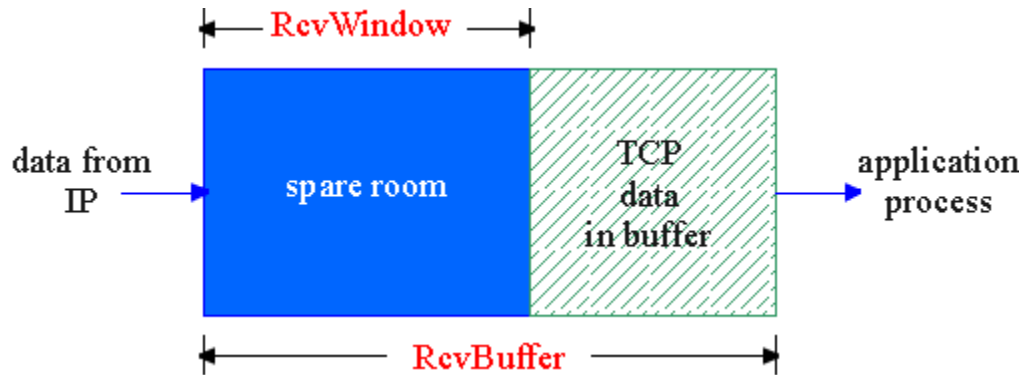
(Suppose TCP receiver
  discards out-of-order
  segments)

o spare room in buffer

`= RcvWindow`

`= RcvBuffer-[LastByteRcvd -`
  `LastByteRead]`

o Rcvr advertises spare
  room by including value
  of `RcvWindow` in
  segments

o Sender limits unACKed
  data to `RcvWindow`

  o guarantees receive buffer
    doesn't overflow

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 Connection Management

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

- ○ initialize TCP variables:
    - ○ seq. #s
    - ○ buffers, flow control info (e.g. `RcvWindow`)
- ○ *client:* connection initiator

    ```
    Socket clientSocket = new
    Socket("hostname","port

    number");
    ```

- ○ *server:* contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- ○ specifies initial seq #
- ○ no data

Step 2: server host receives SYN, replies with SYNACK segment
- ○ server allocates buffers
- ○ specifies server initial seq. #

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

TCP Connection Management (cont.)

Closing a connection:

client closes socket:
    `clientSocket.close();`

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

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

client   server

close ——— FIN ———→

       ←——— ACK ———   close

       ←——— FIN ———

timed wait ——— ACK ———→
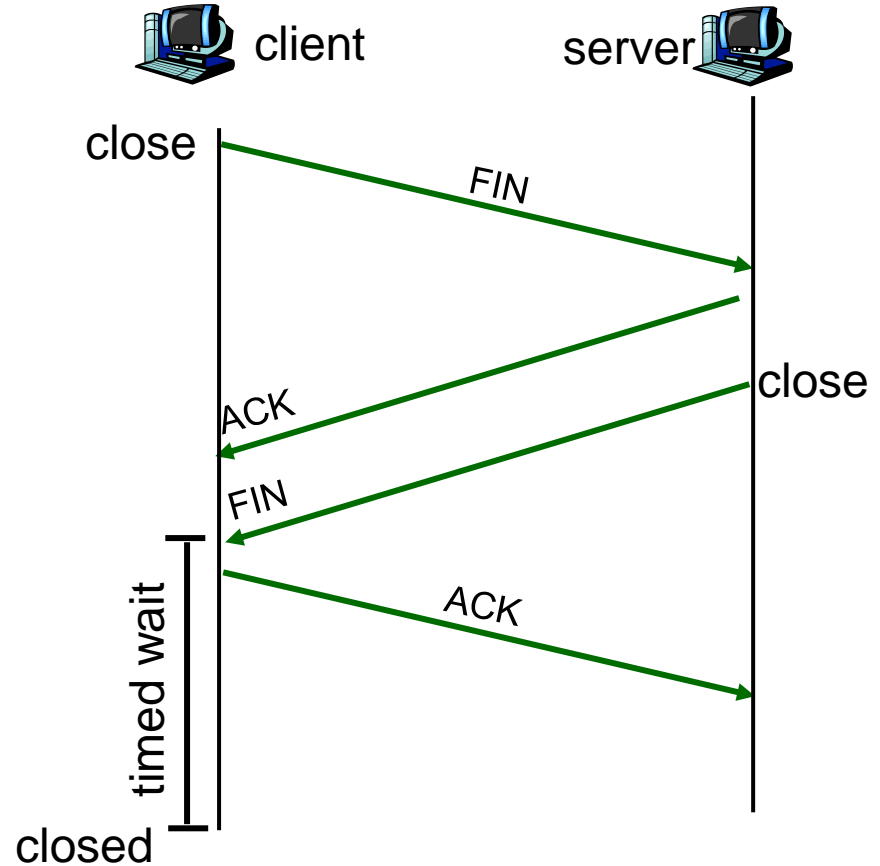
closed

TCP Connection Management (cont.)

<u>Step 3:</u> client receives FIN, replies with ACK.

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

<u>Step 4:</u> server, receives ACK. Connection closed.

<u>Note:</u> with small modification, can handle simultaneous FINs.
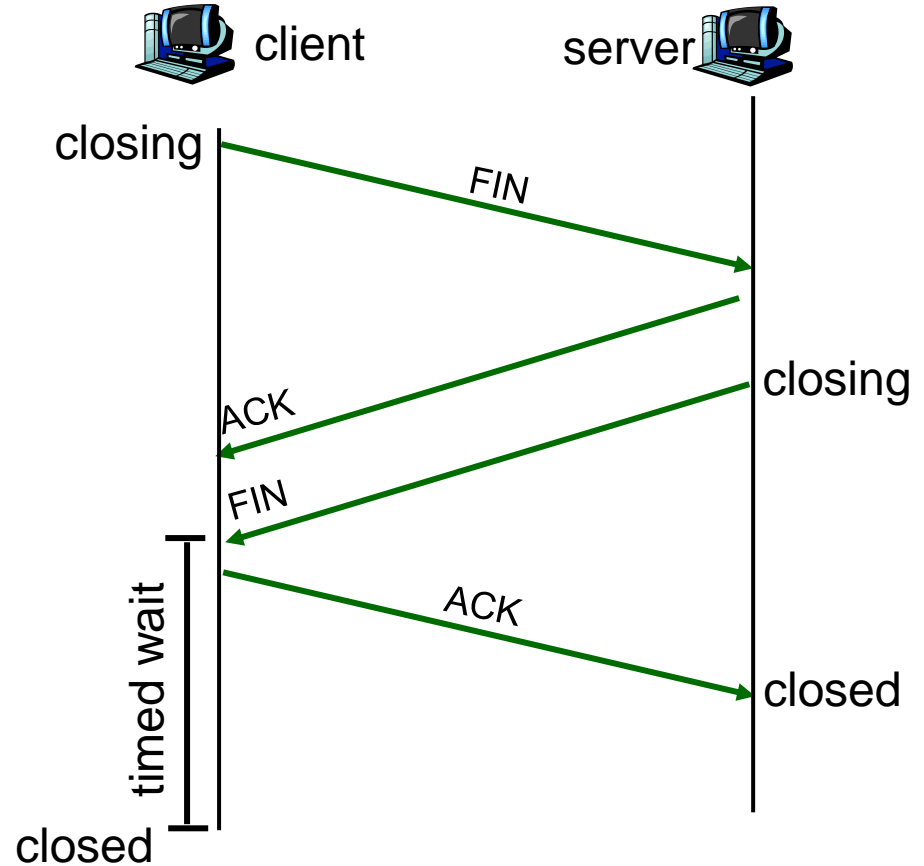
TCP Connection Management (cont)

TCP server lifecycle
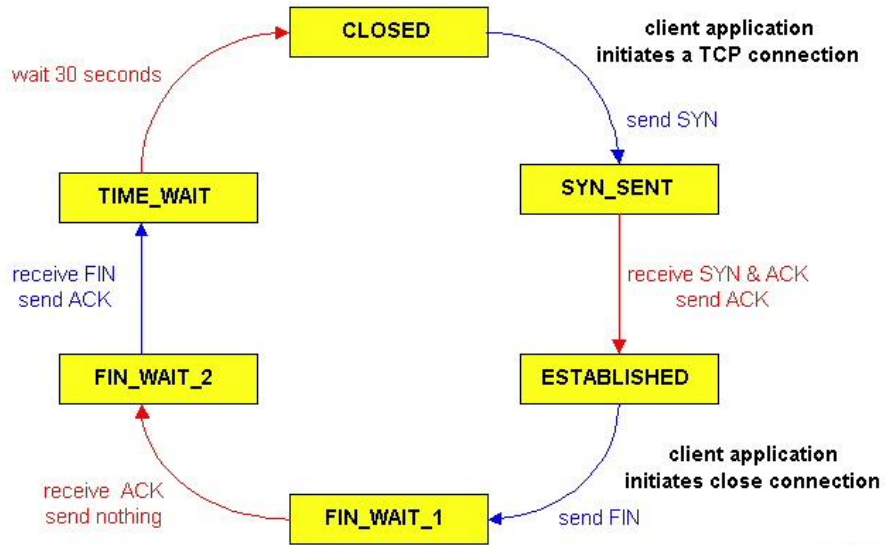
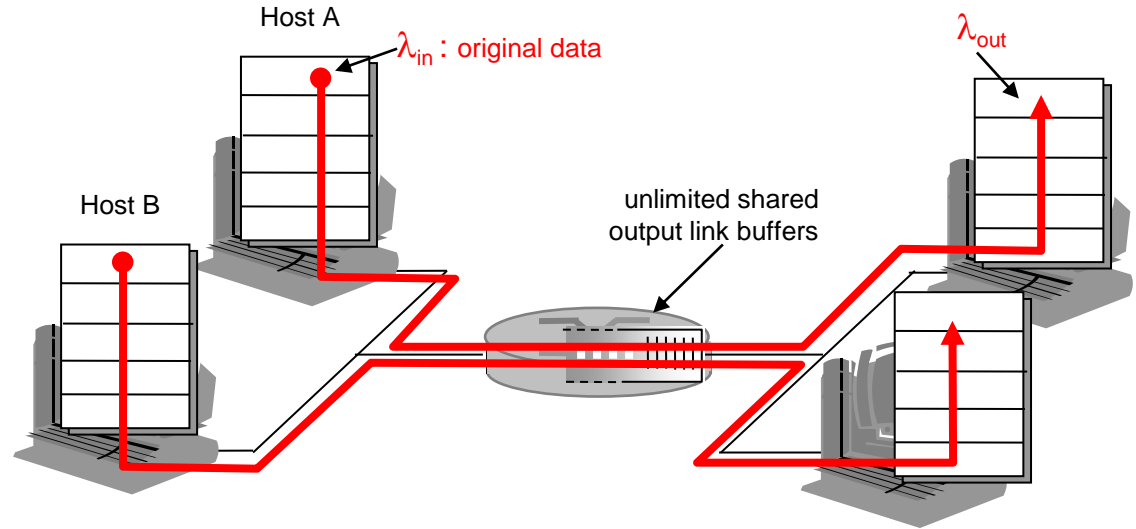TCP client lifecycle

Chapter 3 outline

Principles of Congestion Control

Congestion:

o informally: "too many sources sending too much data too fast for *network* to handle"

o different from flow control! (overflow at receiver v.s. overflow on path routers)

o manifestations:

   o lost packets (buffer overflow at routers)

   o long delays (queueing in router buffers)

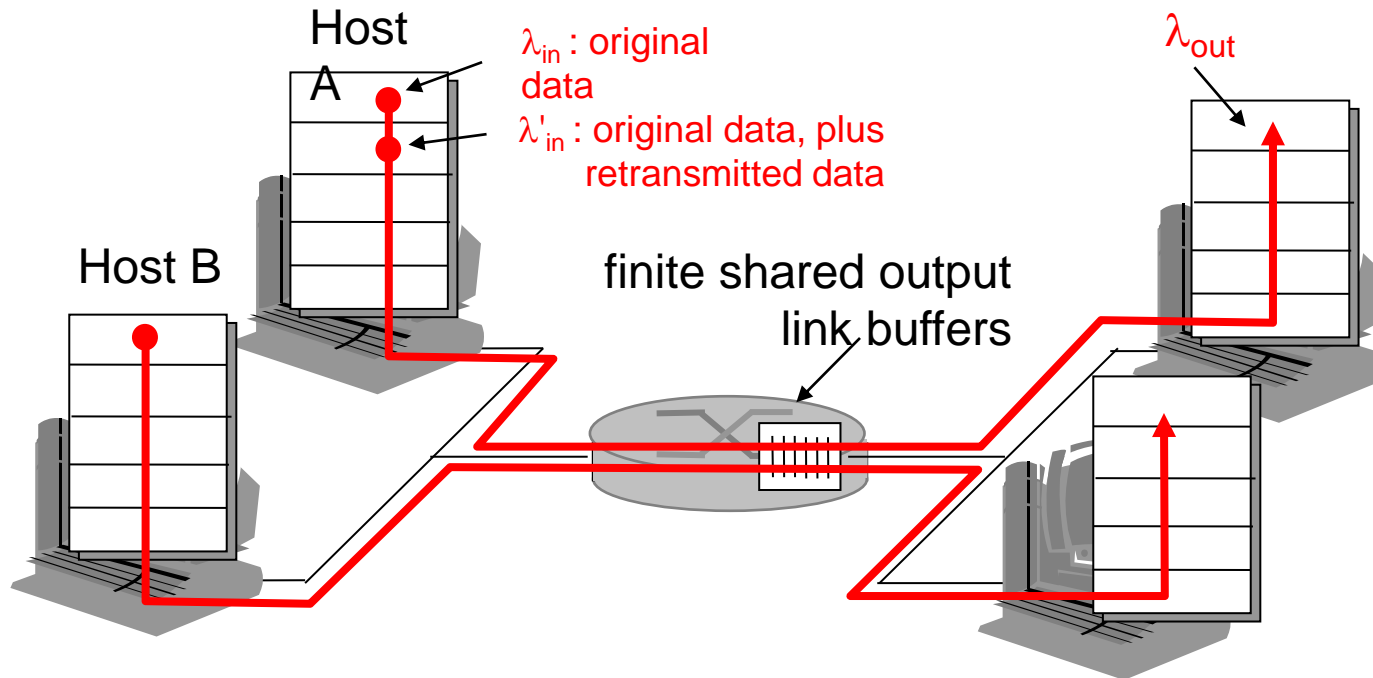o a top-10 problem!

Causes/costs of congestion: scenario 1

o **two senders, two receivers**

o **one router, infinite buffers**

o **no retransmission**

Host A

λ_{in} : original data

λ_{out}

Host B

unlimited shared output link buffers
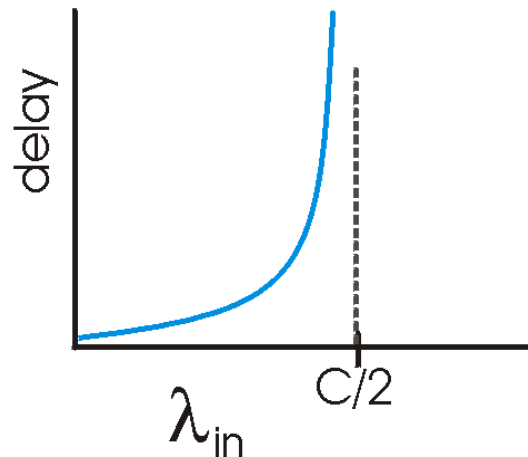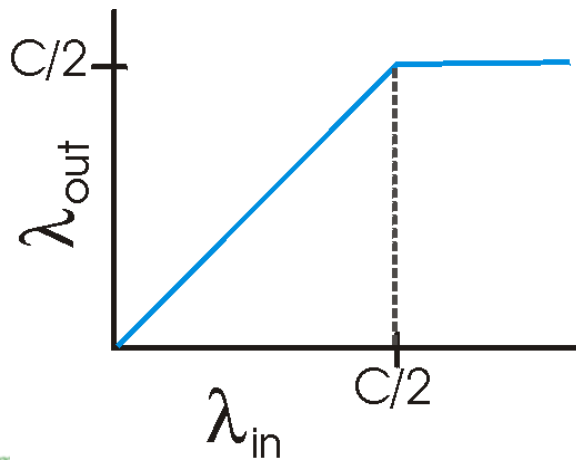
o **large delays when congested**

o **maximum achievable throughput**

Causes/costs of congestion: scenario 2

○ one router, *finite* buffers

○ sender retransmission of lost packet

Host A

λ_{in} : original data

λ'_{in} : original data, plus retransmitted data

λ_{out}

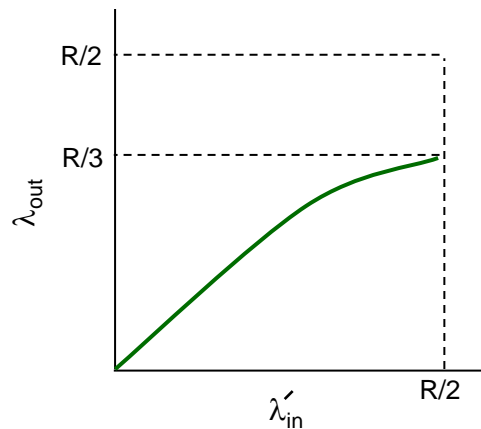Host B

finite shared output link buffers
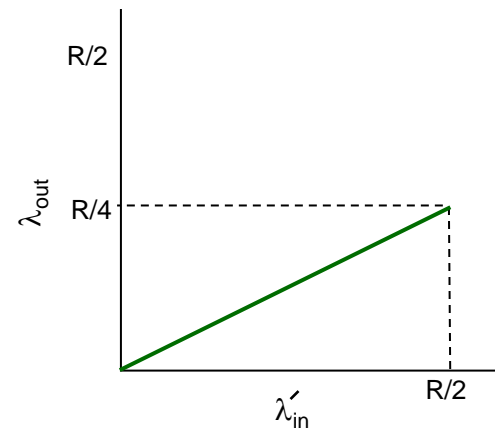
Causes/costs of congestion: scenario 2

○ always: $\lambda_{in} = \lambda_{out}$ (goodput)

○ "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$

○ retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}
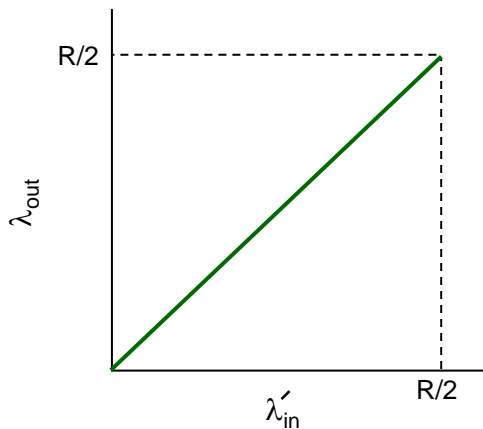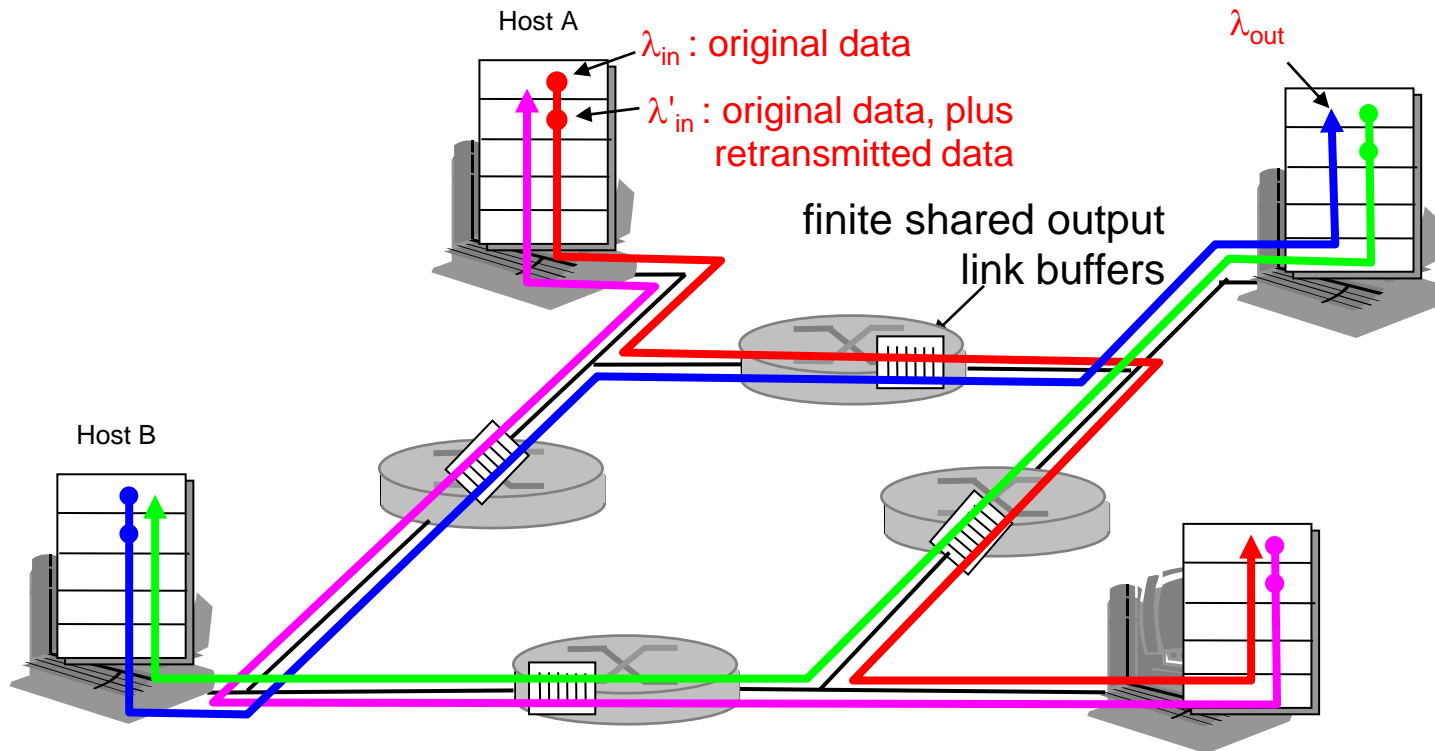

a.   b.   c.

"costs" of congestion:

❑ more work (retrans) for given "goodput"

❑ unneeded retransmissions: link carries multiple copies of pkt
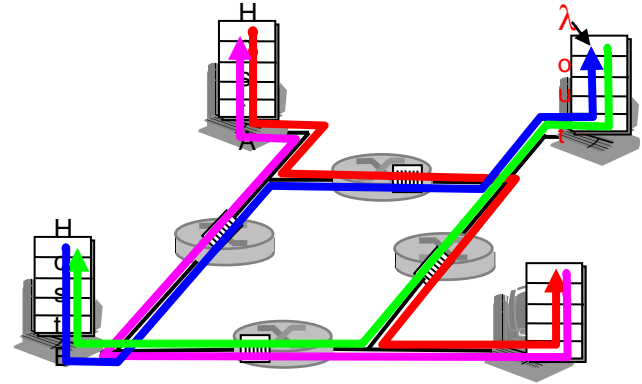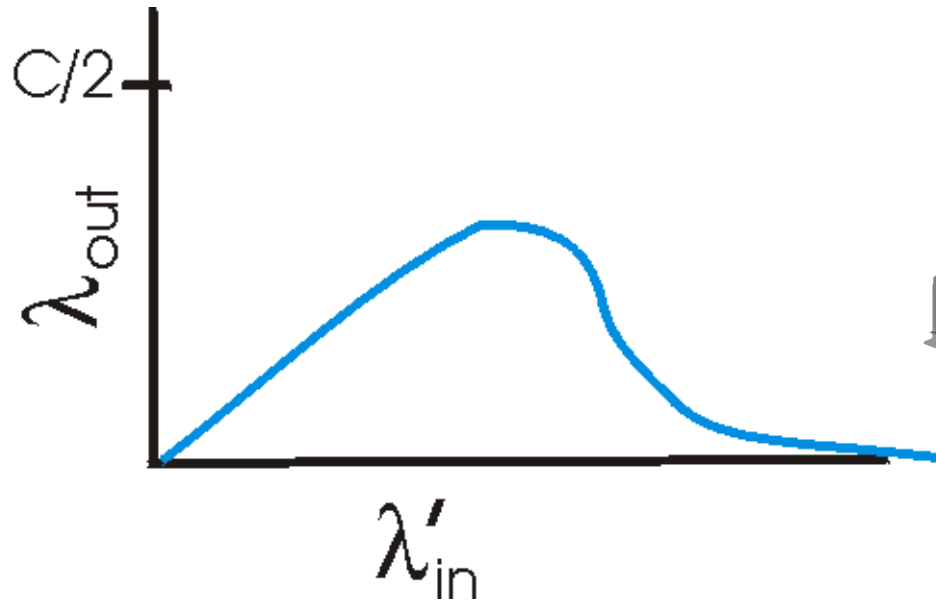
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

Host A

λ_{in} : original data

λ'_{in} : original data, plus retransmitted data

λ_{out}

finite shared output link buffers

Host B

Causes/costs of congestion: scenario 3

Another "cost" of congestion:

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

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- o no explicit feedback from network
- o congestion inferred from end-system observed loss, delay
- o approach taken by TCP

Network-assisted congestion control:

- o routers provide feedback to end systems
  - o single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - o explicit rate sender should send at
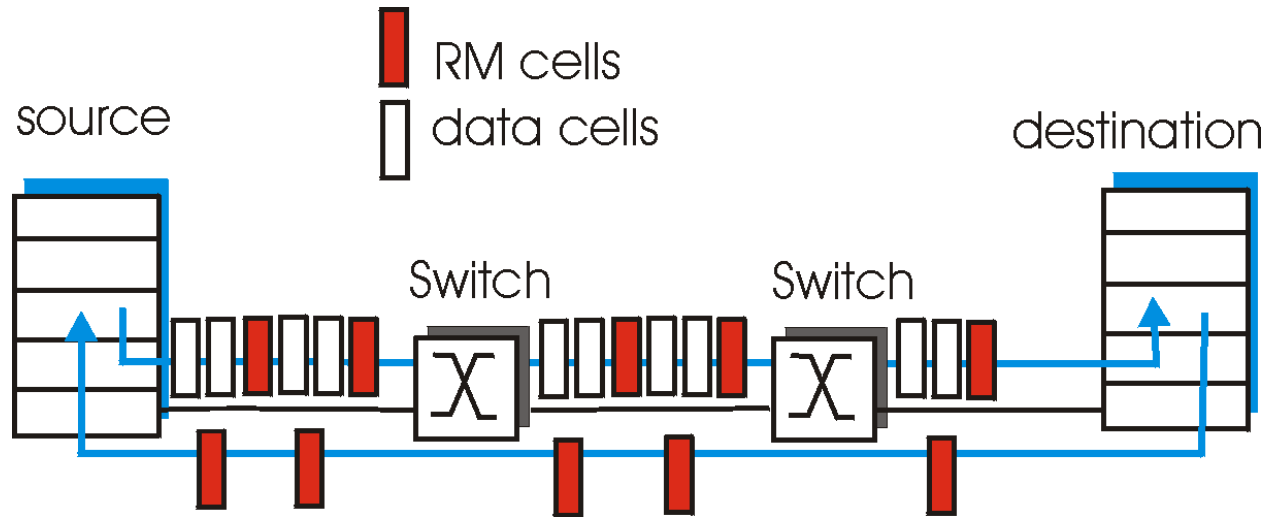
Case study: ATM ABR congestion control

ABR: available bit rate:

o "elastic service"

o if sender's path "underloaded":

    o sender should use available bandwidth

o if sender's path congested:

    o sender throttled to minimum guaranteed rate

RM (resource management) cells:

o sent by sender, interspersed with data cells

o bits in RM cell set by switches ("*network-assisted*")

    o NI bit: no increase in rate (mild congestion)

    o CI bit: congestion indication

o RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control

o two-byte ER (explicit rate) field in RM cell

  o congested switch may lower ER value in cell

  o sender' send rate thus maximum supportable rate on path

o EFCI bit in data cells: set to 1 in congested switch

 o if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

Chapter 3 outline

TCP congestion control: additive increase, multiplicative decrease

☐ *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs

  ○ *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected

  ○ *multiplicative decrease*: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth
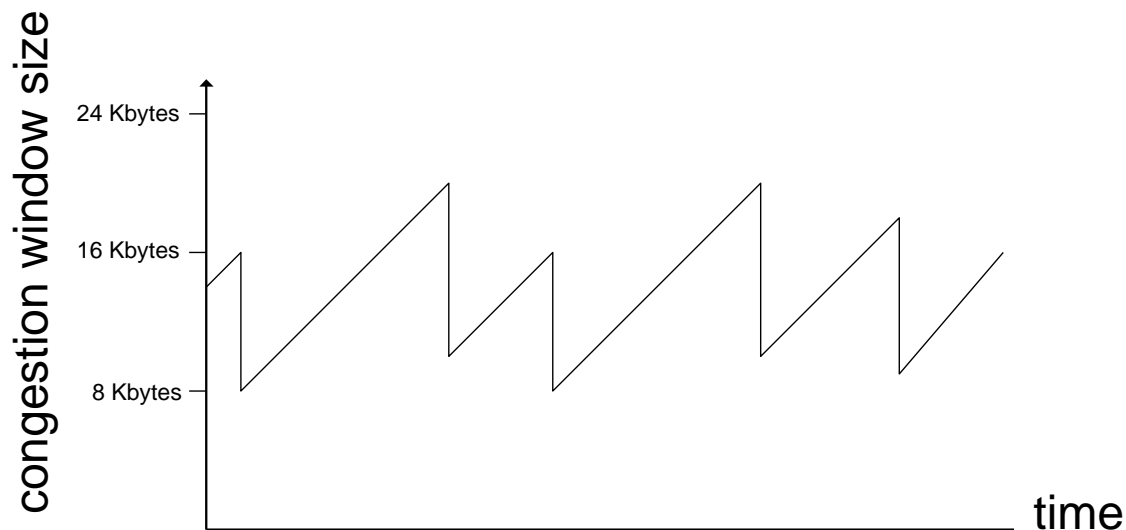
TCP Congestion Control: details

- sender limits transmission:

  LastByteSent-LastByteAcked

  $$\leq \texttt{CongWin}$$

- Roughly,

  $$\text{rate} = \frac{\texttt{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

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

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

- AIMD
- slow start
- conservative after timeout events

TCP Slow Start

o When connection begins, **`CongWin`** = 1 MSS
  - o Example: MSS = 500 bytes & RTT = 1000 msec (1sec)
  - o initial rate = 500 bytes/s

o available bandwidth may be >> MSS/RTT
  - o desirable to quickly ramp up to respectable rate

❑ When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

○ When connection begins, increase rate exponentially until first loss event:
  ○ double `CongWin` every RTT
  ○ done by incrementing `CongWin` for every ACK received

○ <u>Summary:</u> initial rate is slow but ramps up exponentially fast

Host A   Host B

RTT

one segment

two segments

four segments
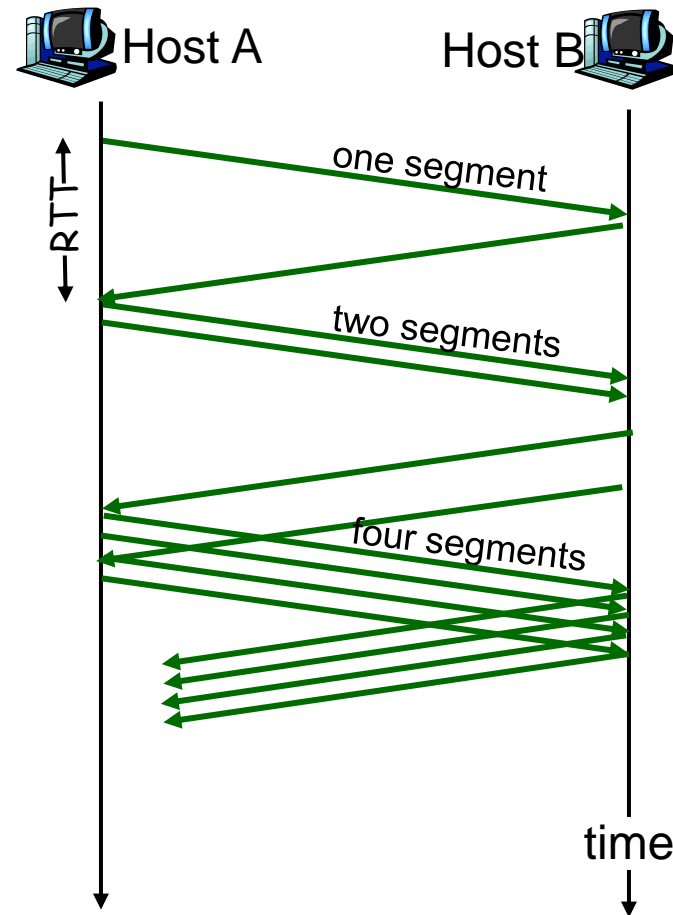
time

Refinement: inferring loss

o After 3 dup ACKs:

- o `CongWin` is cut in half

- o window then grows linearly

o <u>But</u> after timeout event:

- o `CongWin` instead set to 1 MSS;

- o window then grows exponentially

- o to a threshold, then grows linearly

Philosophy:

❑ 3 dup ACKs indicates network capable of delivering some segments

❑ timeout indicates a "more alarming" congestion scenario
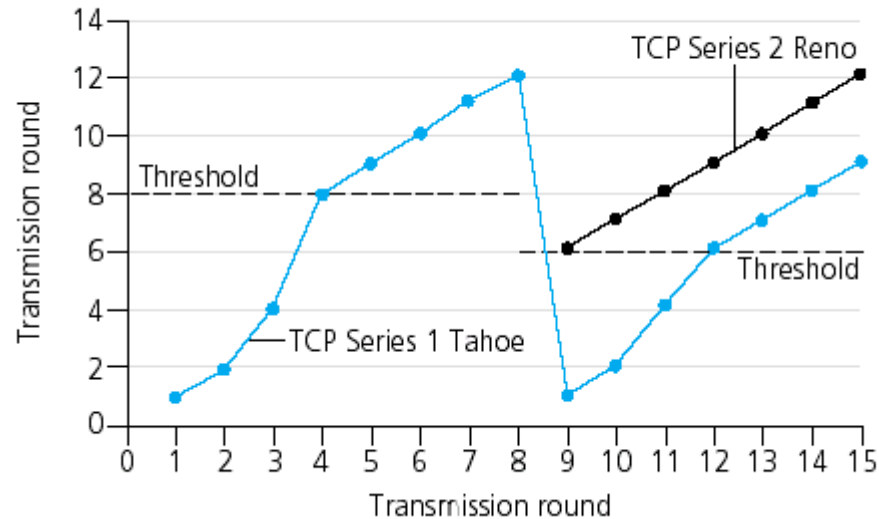
Refinement

Q: When should the exponential increase switch to linear?

A: When `CongWin` gets to 1/2 of its value before timeout.

Implementation:

o   Variable Threshold

o   At loss event, Threshold is set to 1/2 of CongWin just before loss event

Summary: TCP Congestion Control

o When **CongWin** is below **Threshold**, sender in slow-start phase, window grows exponentially.

o When **CongWin** is above **Threshold**, sender is in congestion-avoidance phase, window grows linearly.

o When a triple duplicate ACK occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.

o When timeout occurs, **Threshold** set to **CongWin/2** and **CongWin** is set to 1 MSS.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

TCP throughput

o What's the average throughout of TCP as a function of window size and RTT?

    o Ignore slow start

o Let W be the window size when loss occurs.

o When window is W, throughput is W/RTT

o Just after loss, window drops to W/2, throughput to W/2RTT.

o Average throughout: .75 W/RTT

Chapter 3: Summary

o principles behind transport layer services:

  o multiplexing, demultiplexing

  o reliable data transfer

  o flow control

  o congestion control

o instantiation and implementation in the Internet

  o UDP

  o TCP

Next:

o Network Layer

Thank you

Any questions?