

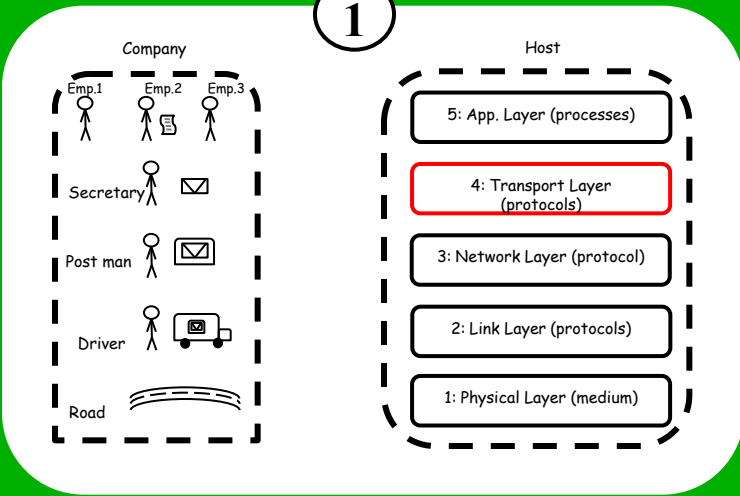
# Transport Layer – Part II

Computer Networks, Winter 2010/2011

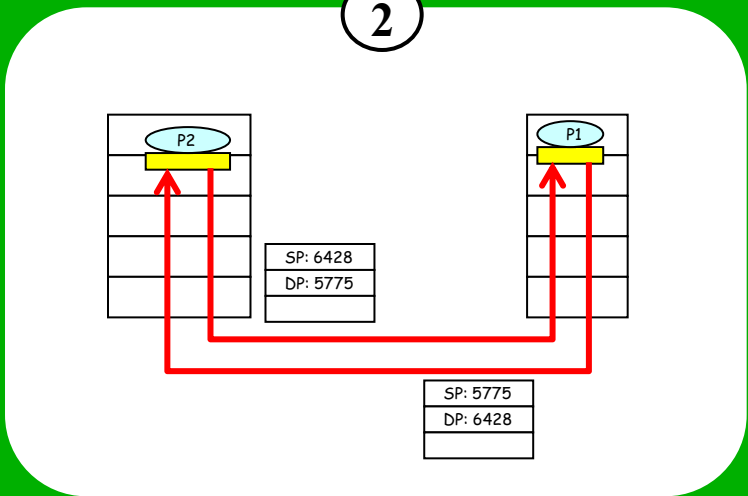


# Last Session

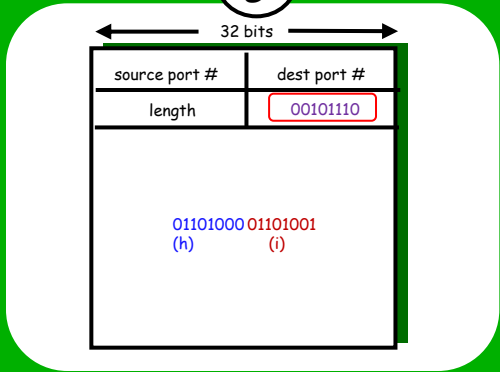
1



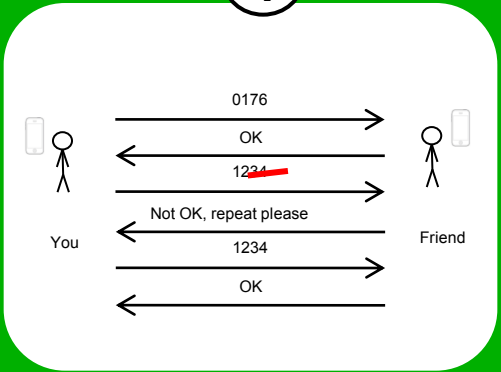
2



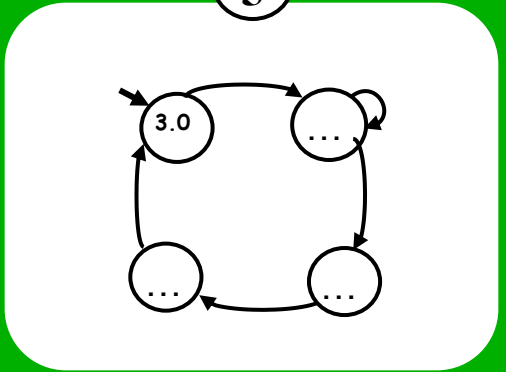
3



4



5



# Chapter 4 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

# 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

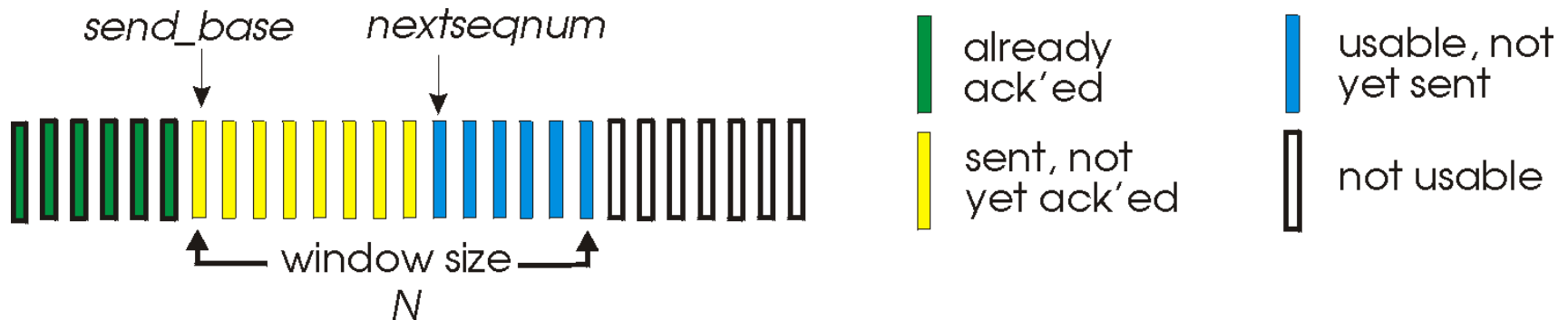
## Selective Repeat: big pic

- 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:

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

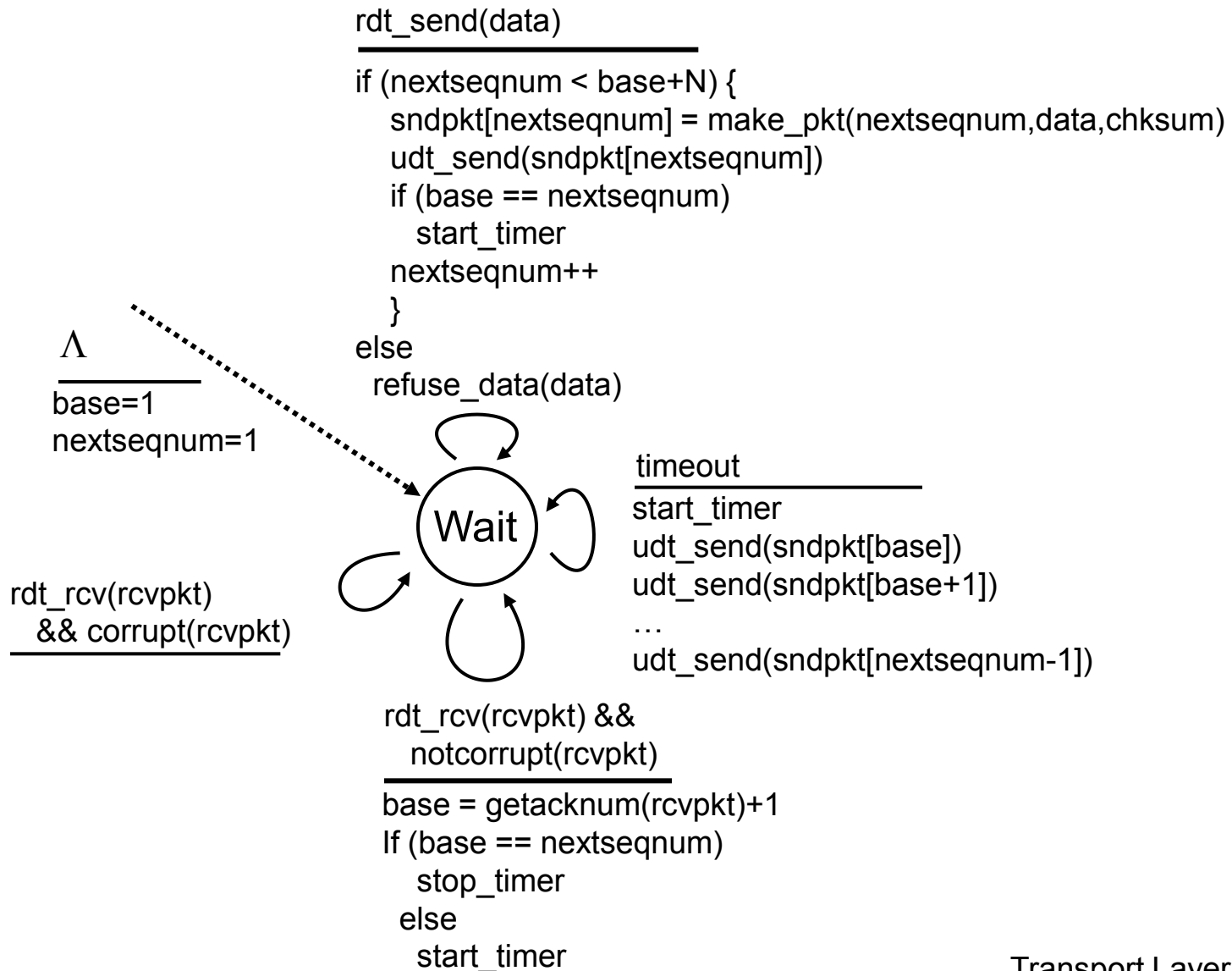


- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

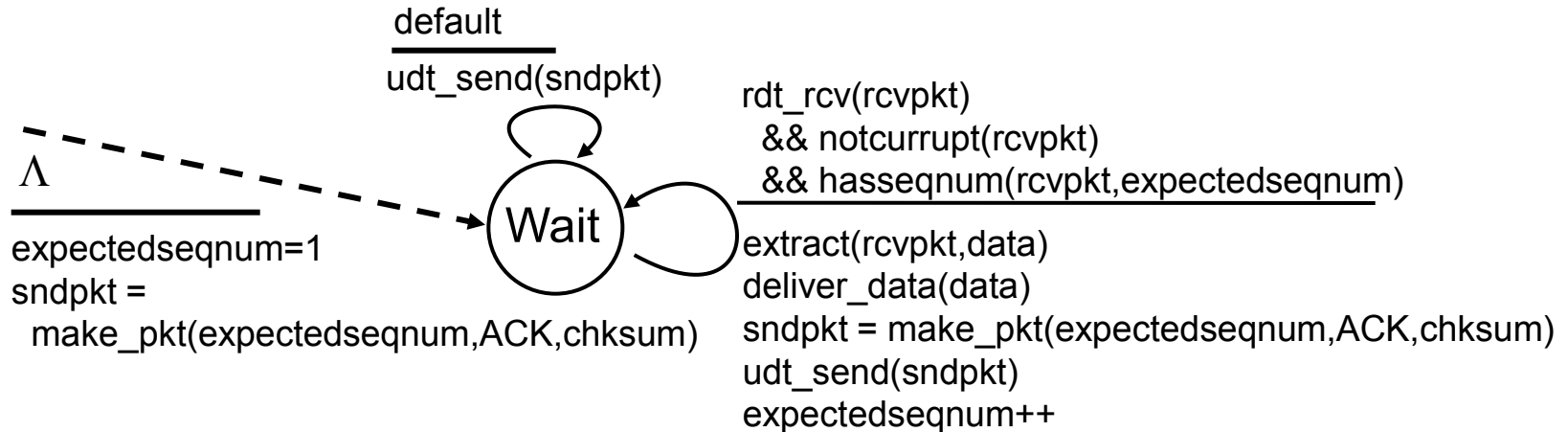
# Applet Demo

- [http://media.pearsoncmg.com/aw/aw\\_kurose\\_network\\_2/applets/go-back-n/go-back-n.html](http://media.pearsoncmg.com/aw/aw_kurose_network_2/applets/go-back-n/go-back-n.html)
- [http://media.pearsoncmg.com/aw/aw\\_kurose\\_network\\_3/applets/SelectRepeat/SR.html](http://media.pearsoncmg.com/aw/aw_kurose_network_3/applets/SelectRepeat/SR.html)
- (Self Study)

# GBN: sender extended FSM



# GBN: receiver extended FSM

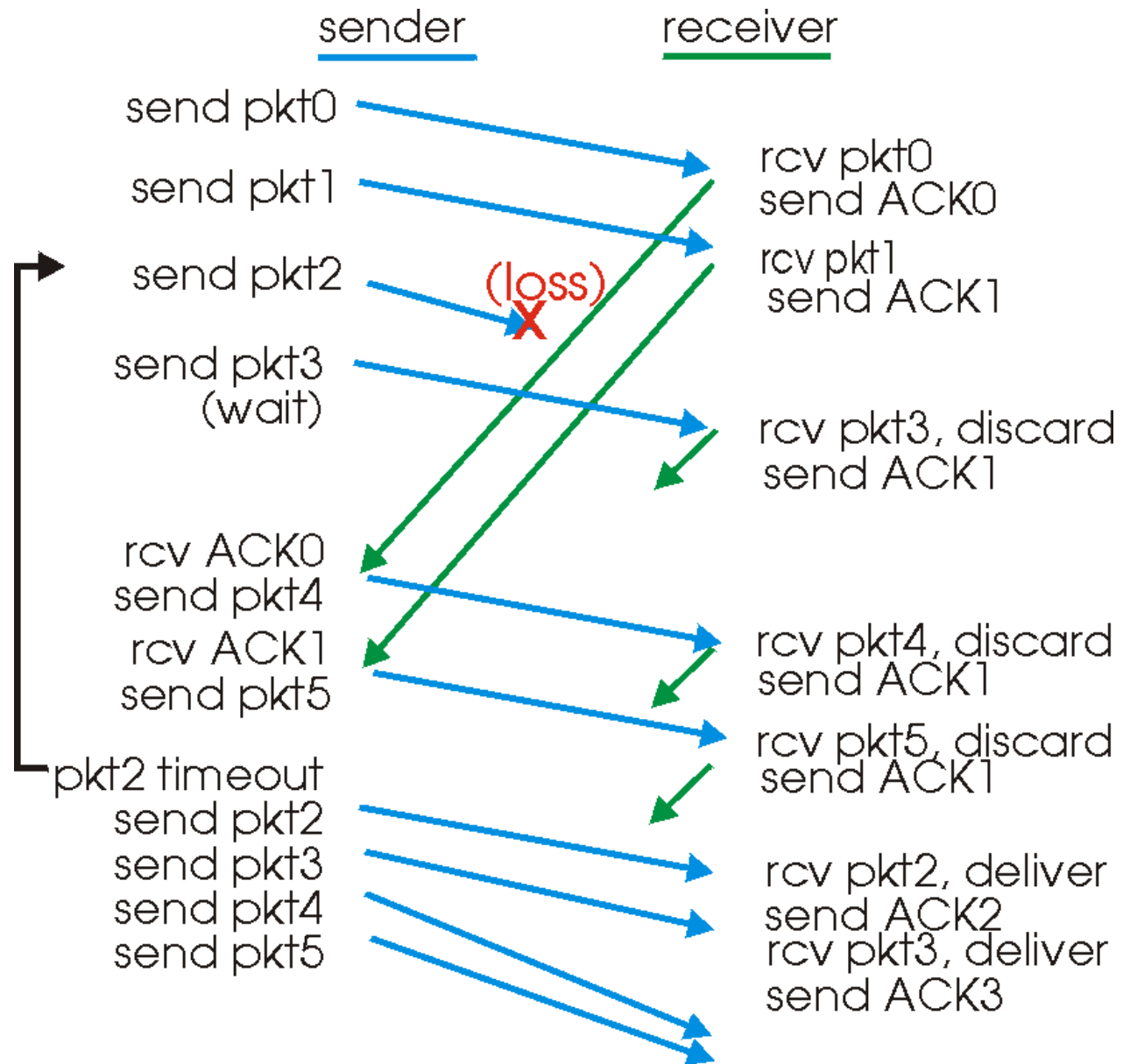


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- 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 #



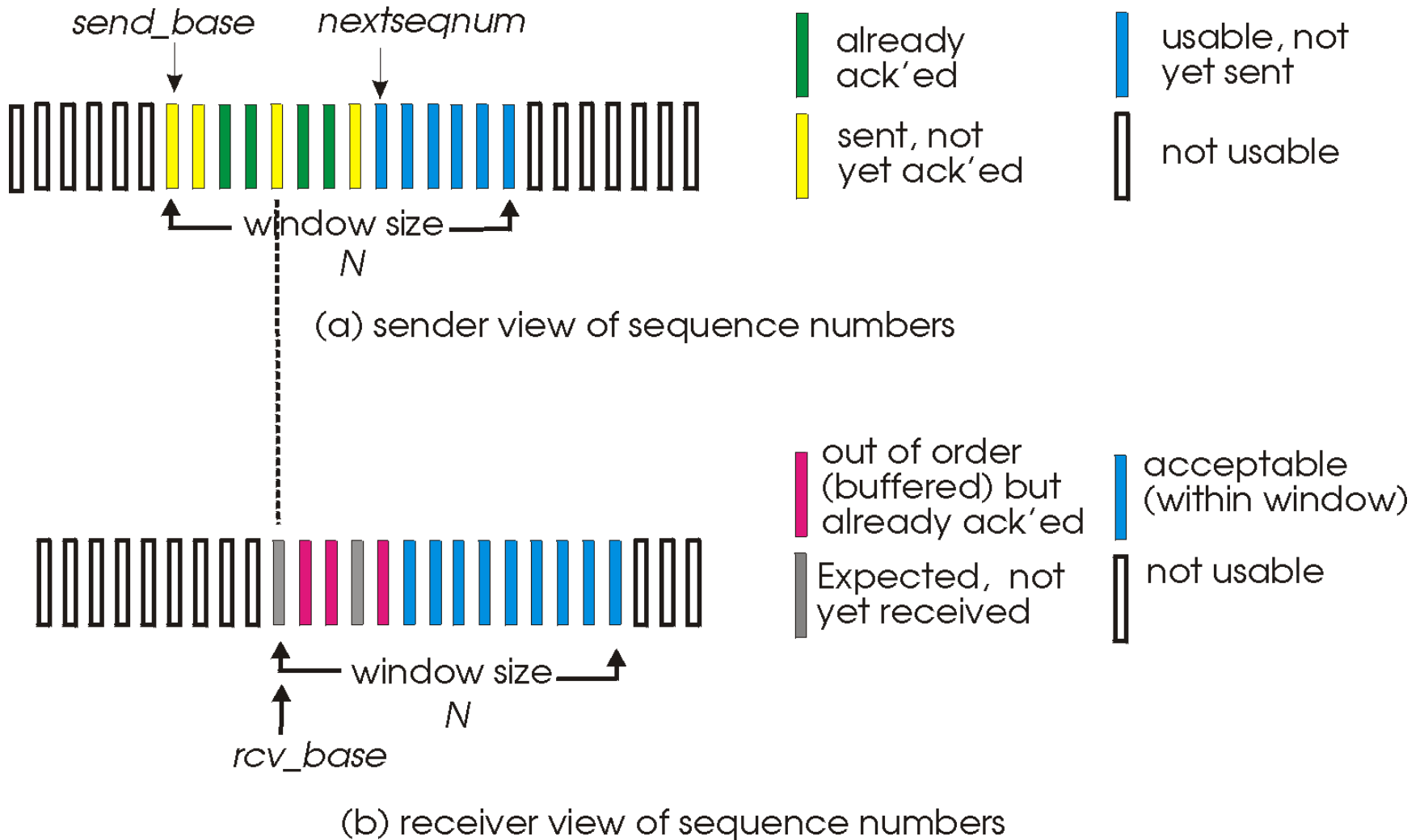
# GBN in action



# 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
  - again limits seq #'s of sent, unACKed pkts

# Selective repeat: sender, receiver windows



# 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

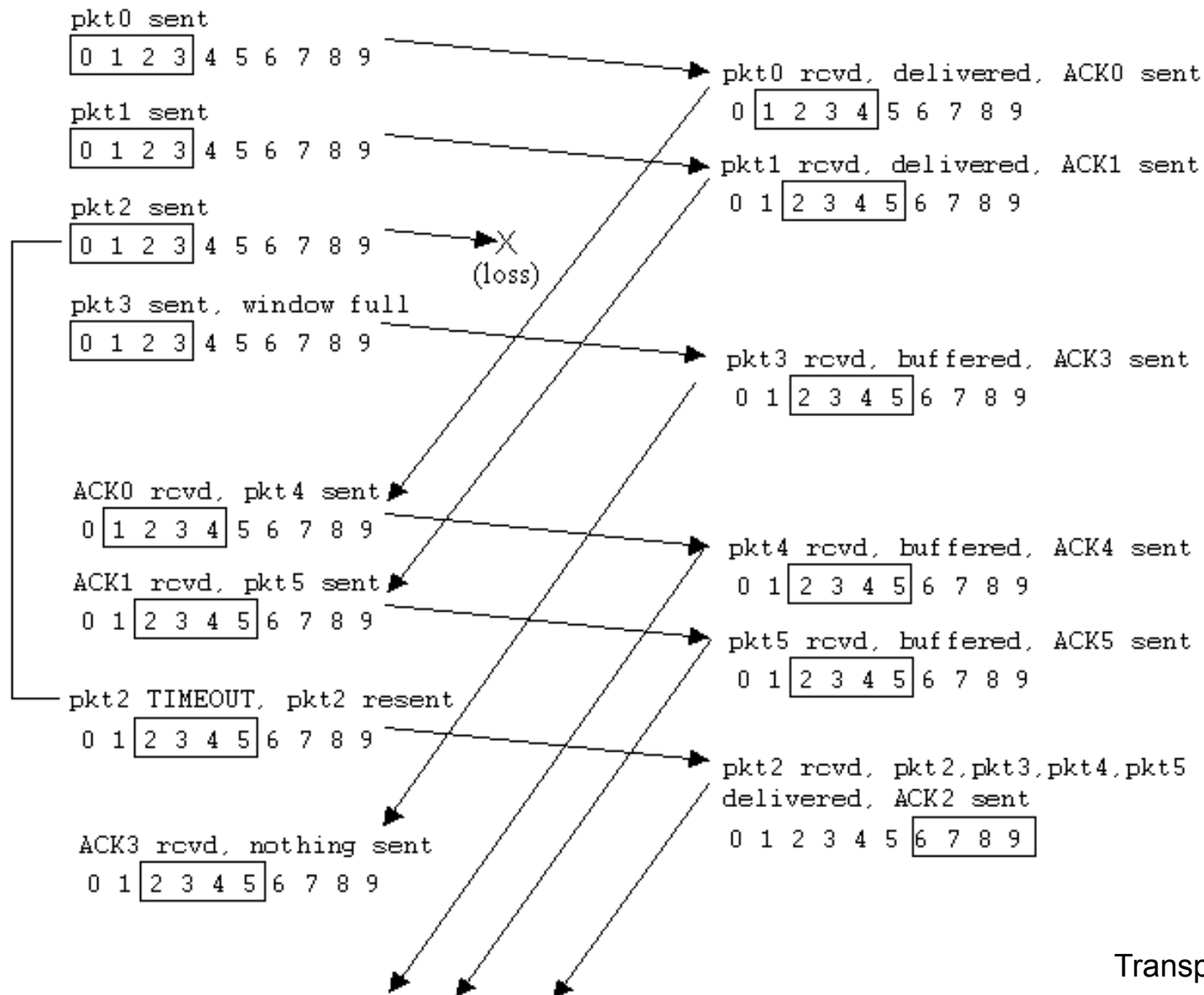
**pkt n in [rcvbase-N,rcvbase-1]**

- ACK(n)

**otherwise:**

- ignore

# Selective repeat in action

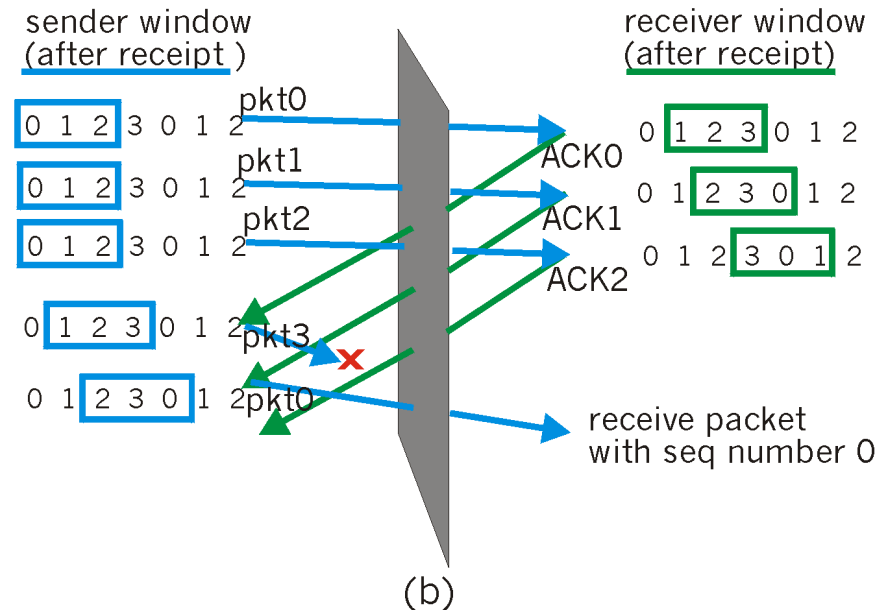
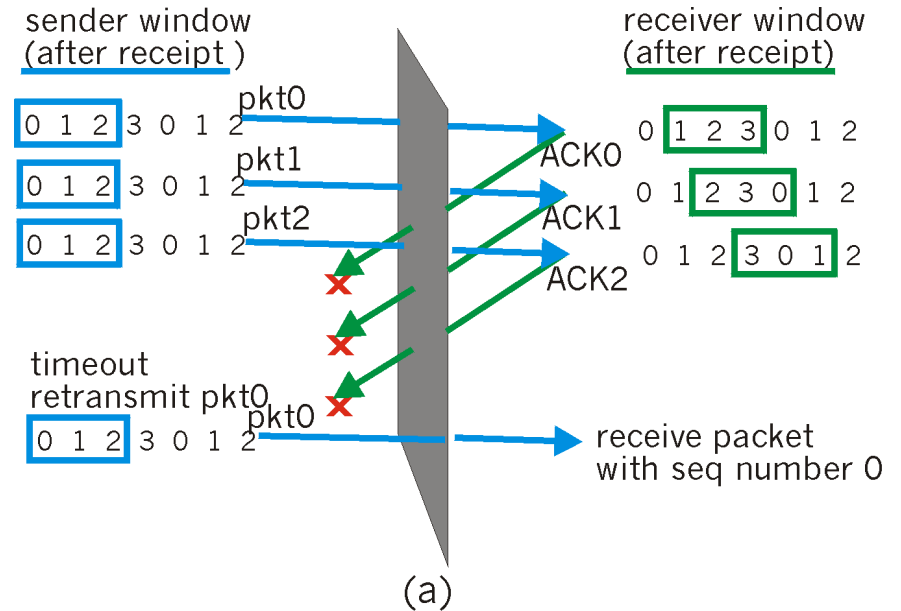


# Selective repeat: dilemma

## Example:

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

**Notice:** Window size should be not too large, e.g.  $\frac{1}{2}$  of sequence range.



# Chapter 4 outline

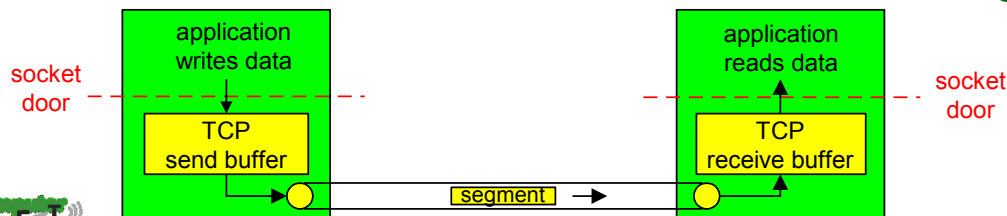
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# TCP: Overview

## 2581

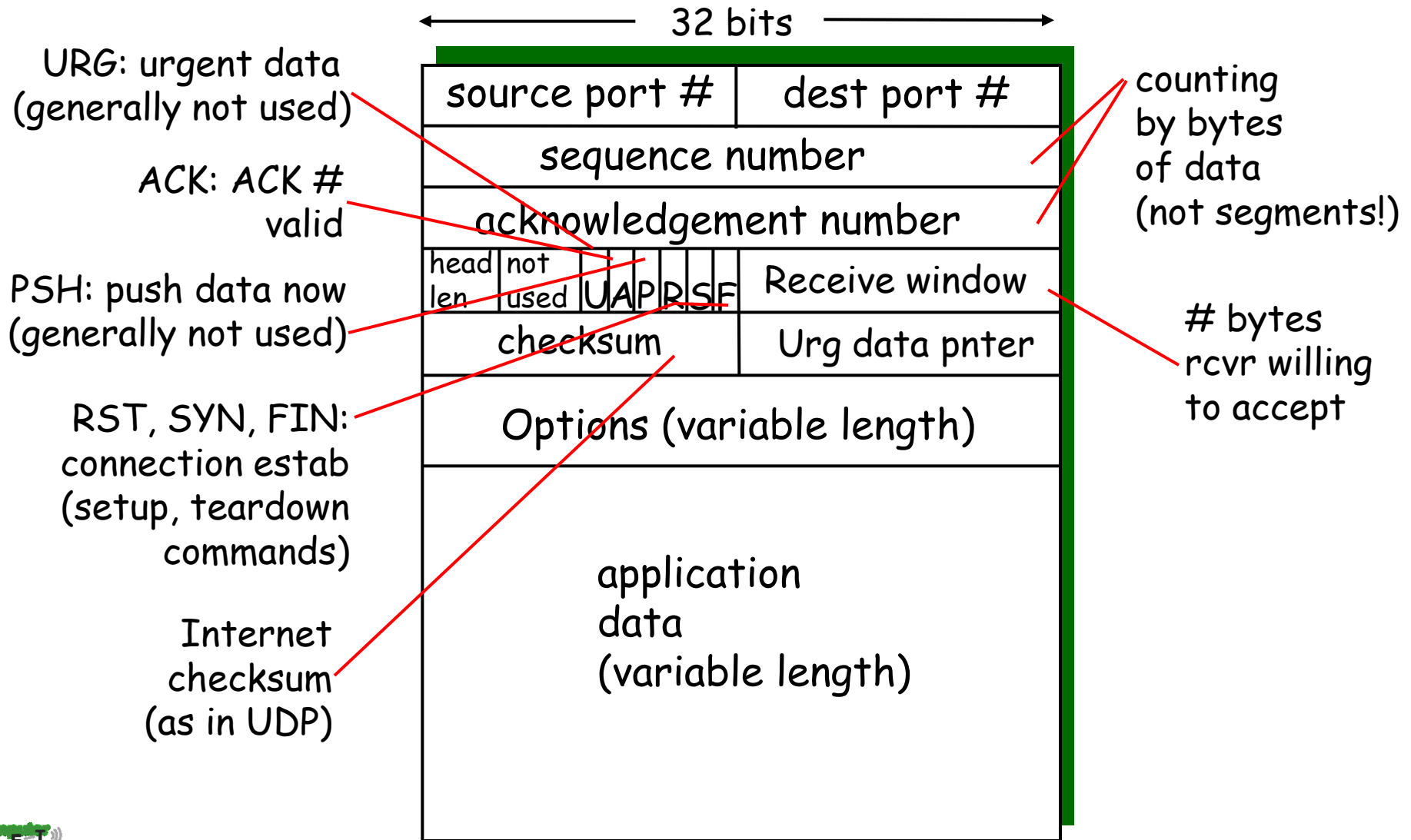
RFCs: 793, 1122, 1323, 2018,

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





# TCP segment structure



# TCP seq. #'s and ACKs

## Seq. #'s:

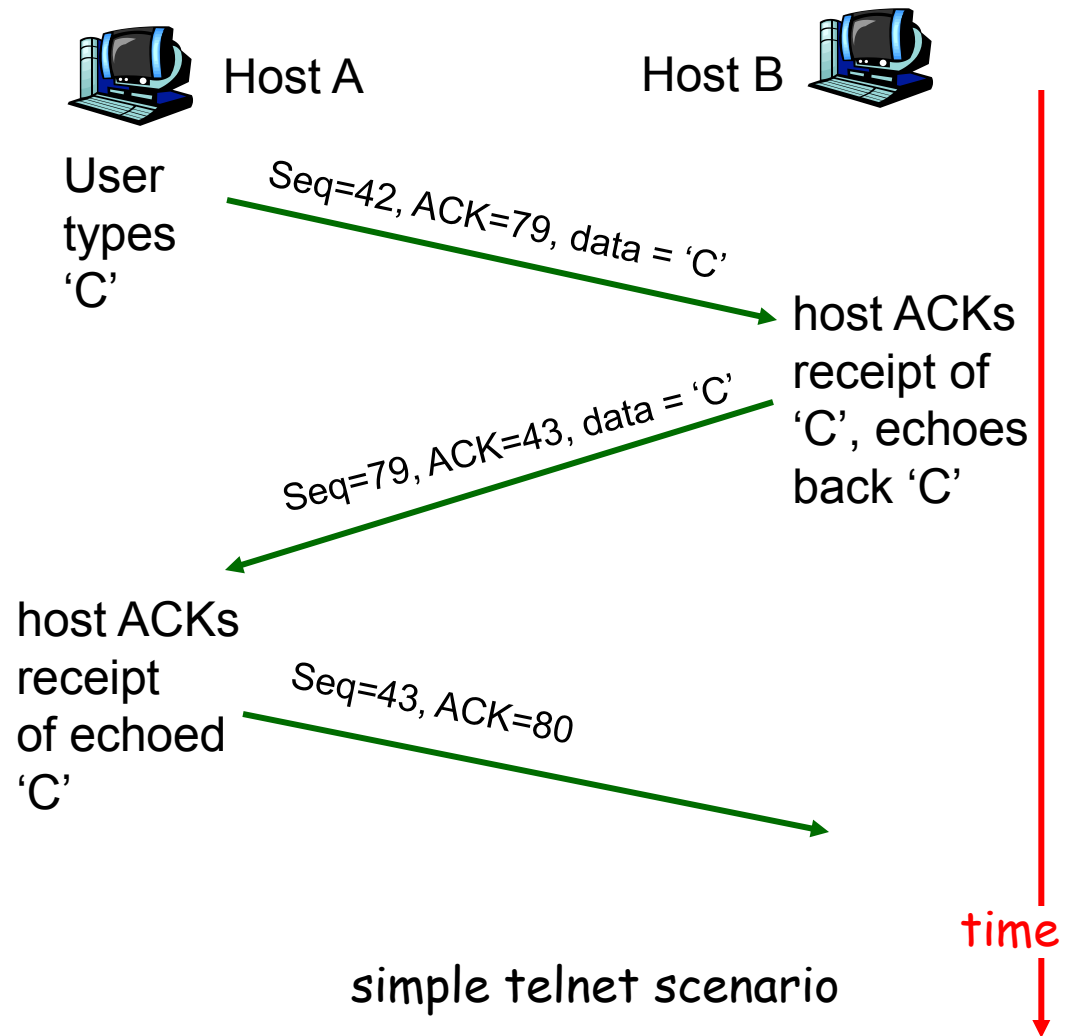
- byte stream  
“number” of first byte in segment's data

## 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



# 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**

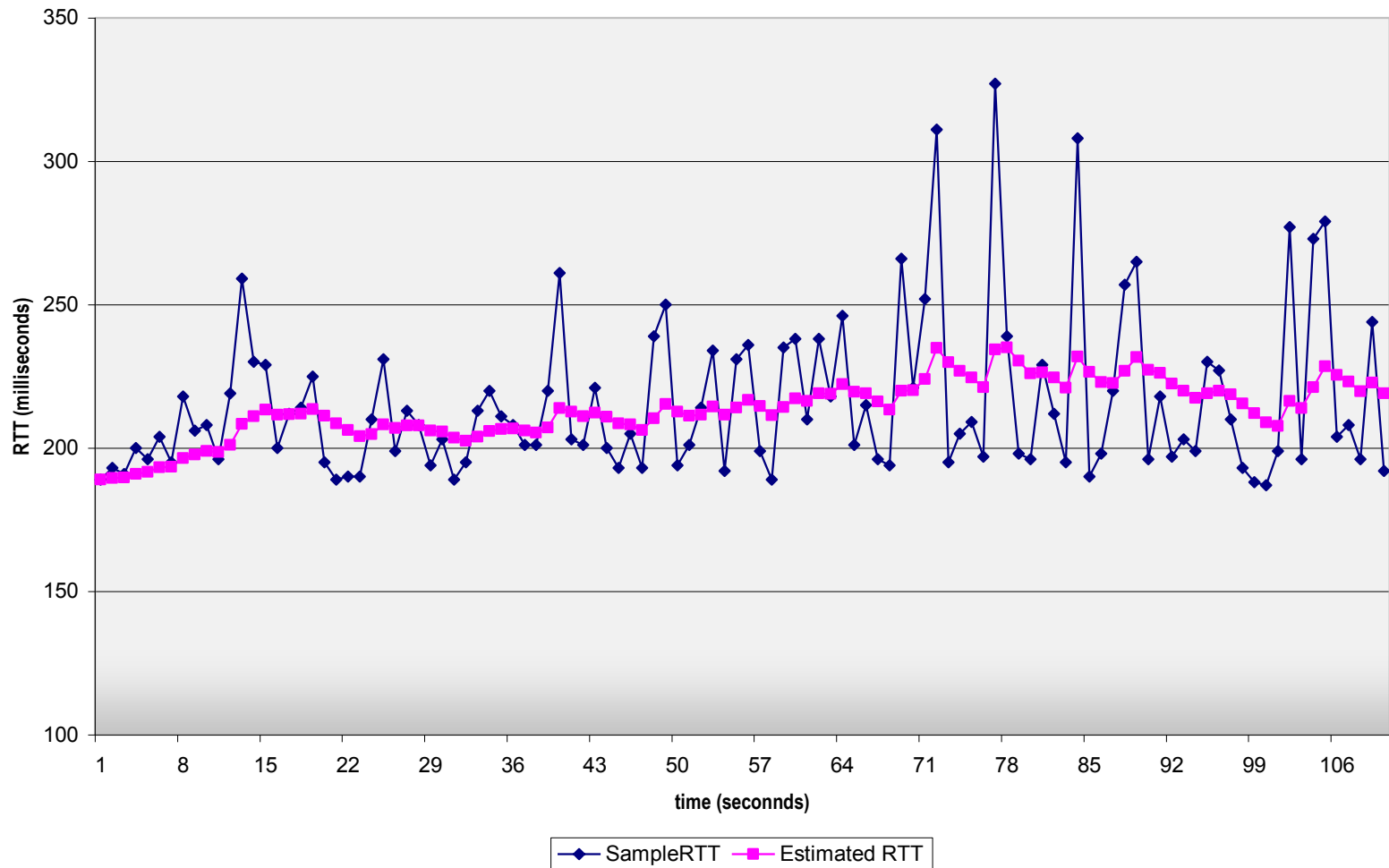
# TCP Round Trip Time and Timeout

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

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

# Example 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**:

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

(typically,  $\beta = 0.25$ )

Then set timeout interval:

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

# Chapter 4 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
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- 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 acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

```
NextSeqNum = InitialSeqNum
```

```
SendBase = InitialSeqNum
```

```
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)
```

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

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

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

# TCP sender (simplified)

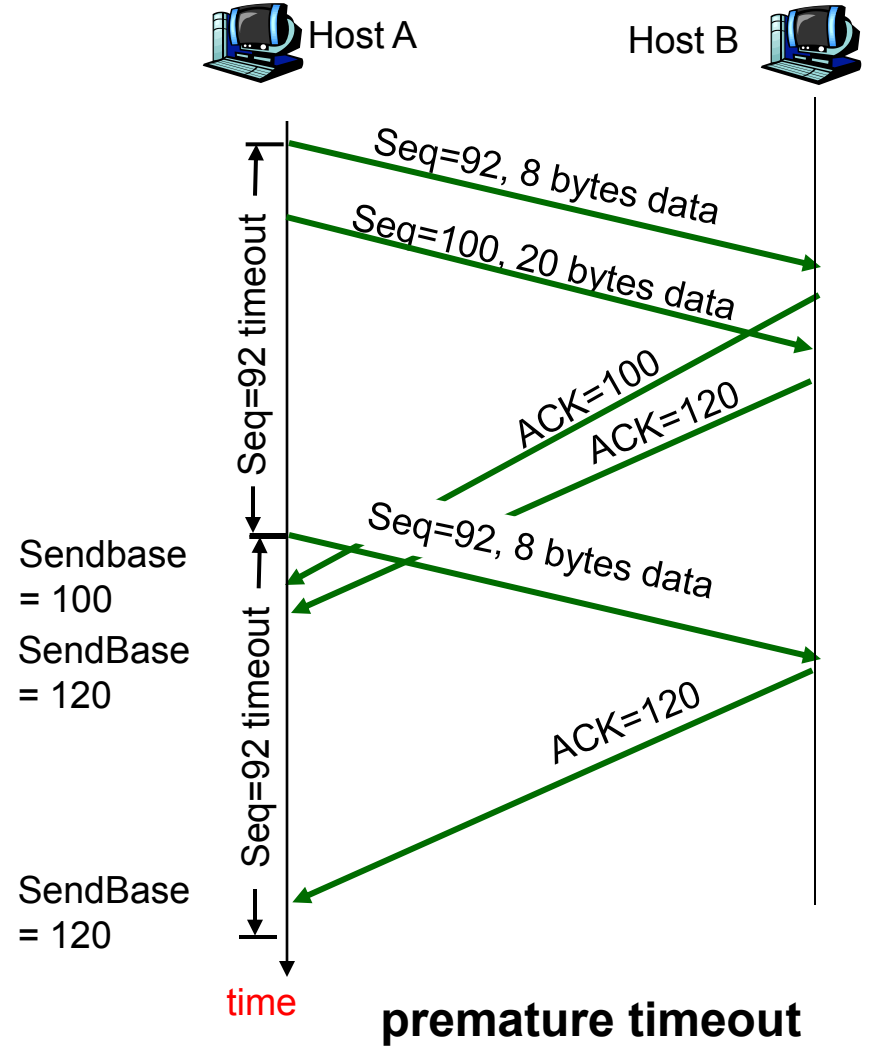
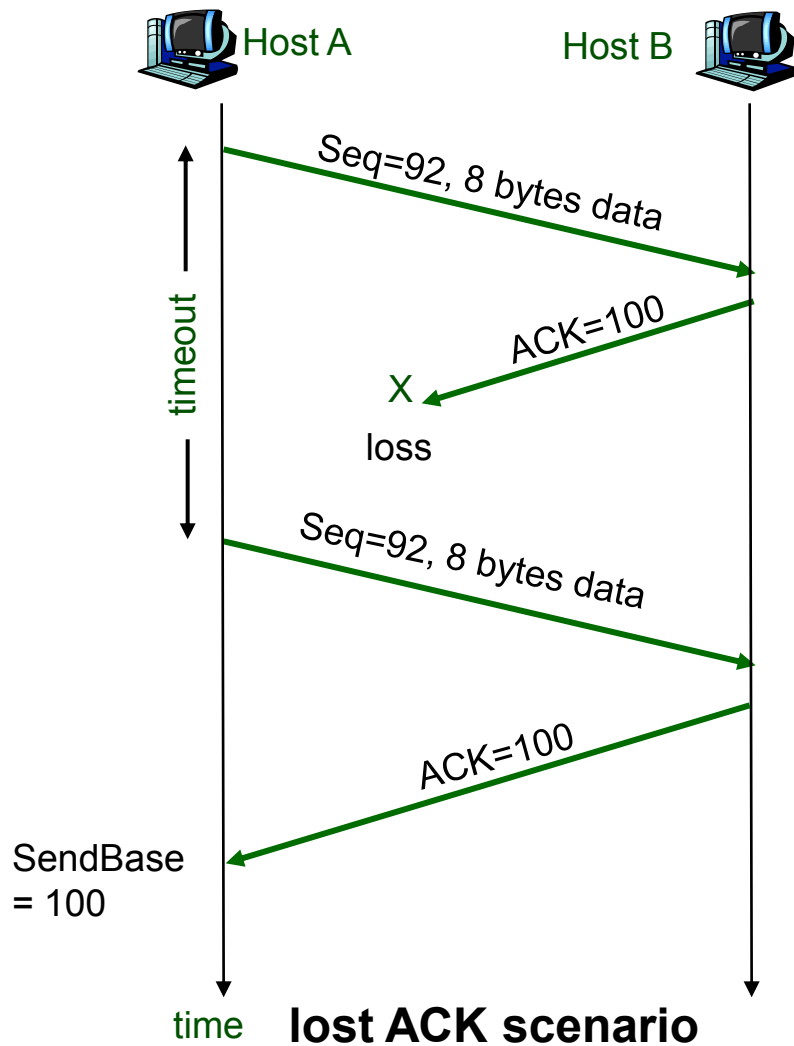
## Comment:

- SendBase-1: last cumulatively ack'ed byte

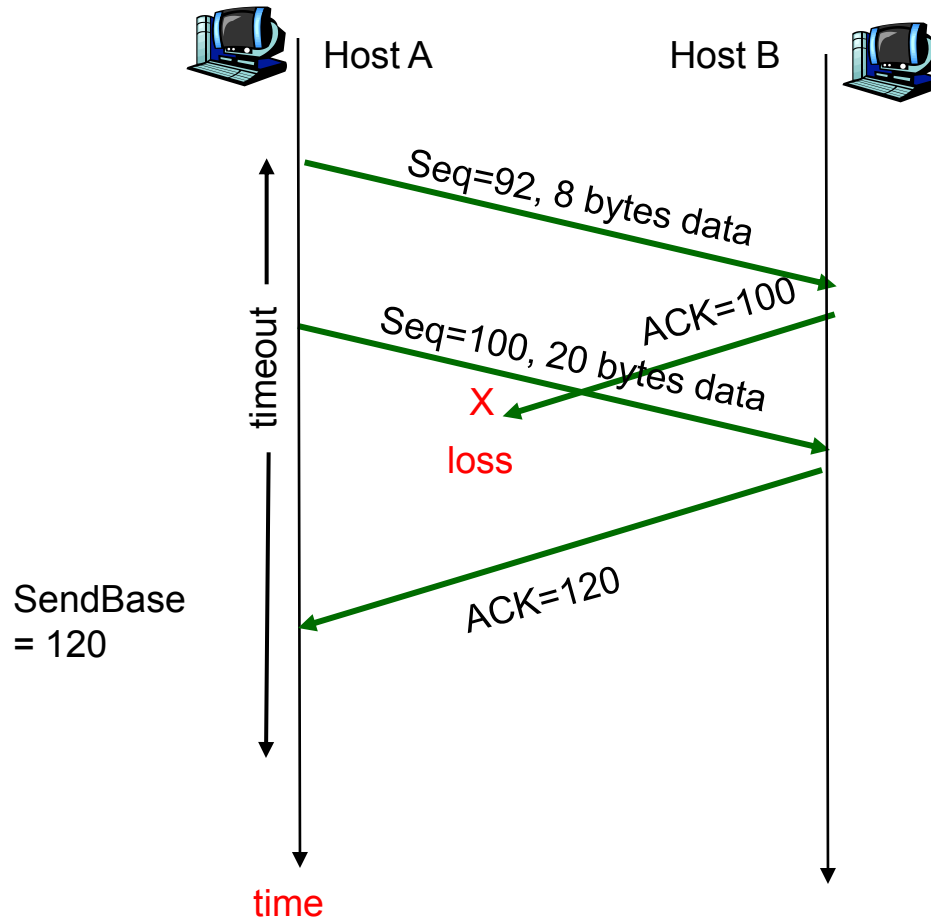
## Example:

- SendBase-1 = 71;  
y = 73, so the rcvr wants 73+ ;  
y > SendBase, so that new data is acked

# TCP: retransmission scenarios



# TCP retransmission scenarios (more)



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-expected 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

- 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.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

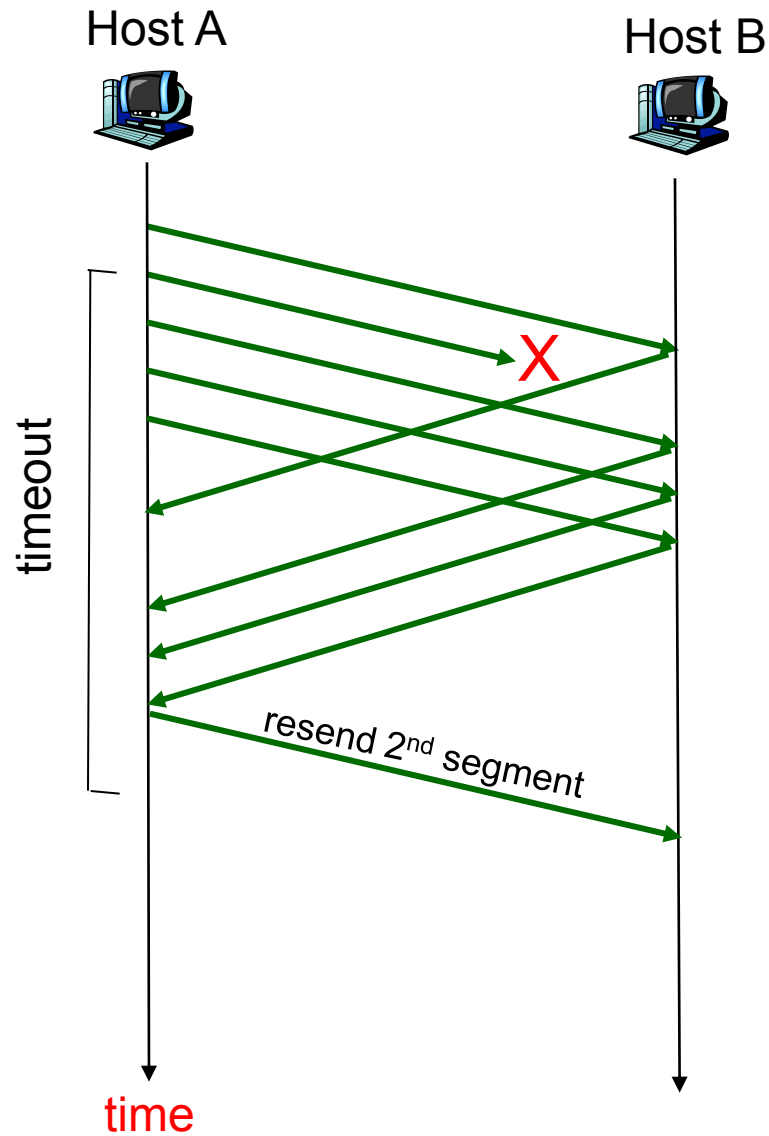


Figure 3.37 Resending a segment after triple duplicate ACK

# Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
    }
    else {
        increment count of dup ACKs received for y
        if (count of dup ACKs received for y = 3) {
            resend segment with sequence number y
        }
    }
```

a duplicate ACK for  
already ACKed segment

fast retransmit



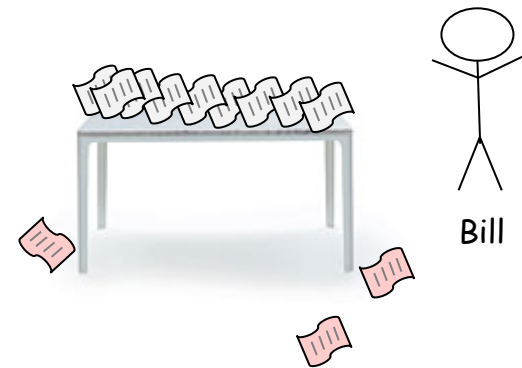
# Chapter 4 outline

- 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

# Analogy: Flow Control

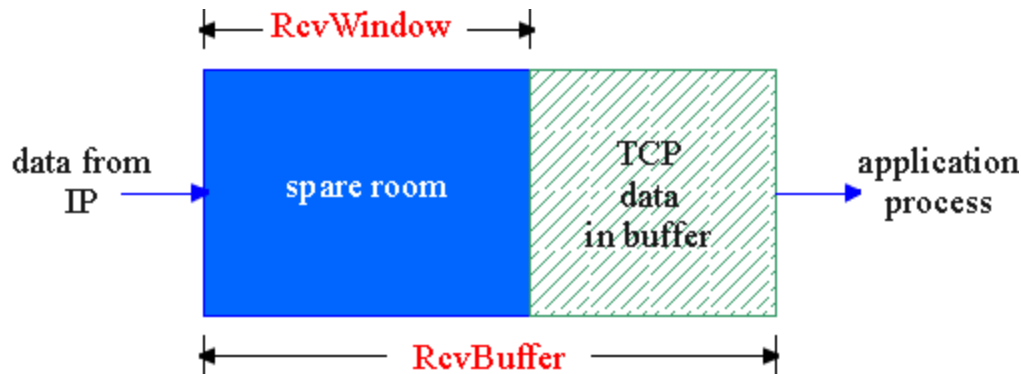
- Assumptions:
  - Secretary delivers mail at rate of 4 letters/h
  - Employee Bill processes mail at 1 letter/h.
  - Table has place for 10 letters, more will drop on floor.
- After half a day his table overflows, letters get lost.
- Sender needs to decrease sending rate.

time	Mail read	Mail on table
9:00	0	4
10:00	1	7
11:00	2	10
12:00	3	<b>13 !</b>



# TCP Flow Control

- receive side of TCP connection has a receive buffer:



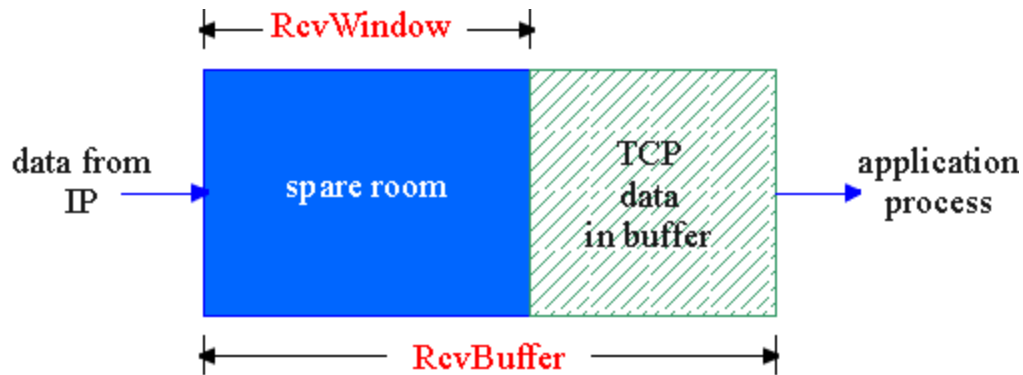
## flow control

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

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

- app process may be slow at reading from buffer

# TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
- =  $RcvWindow$
- =  $RcvBuffer - [LastByteRcvd - LastByteRead]$

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
  - guarantees receive buffer doesn't overflow

# Chapter 4 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.)

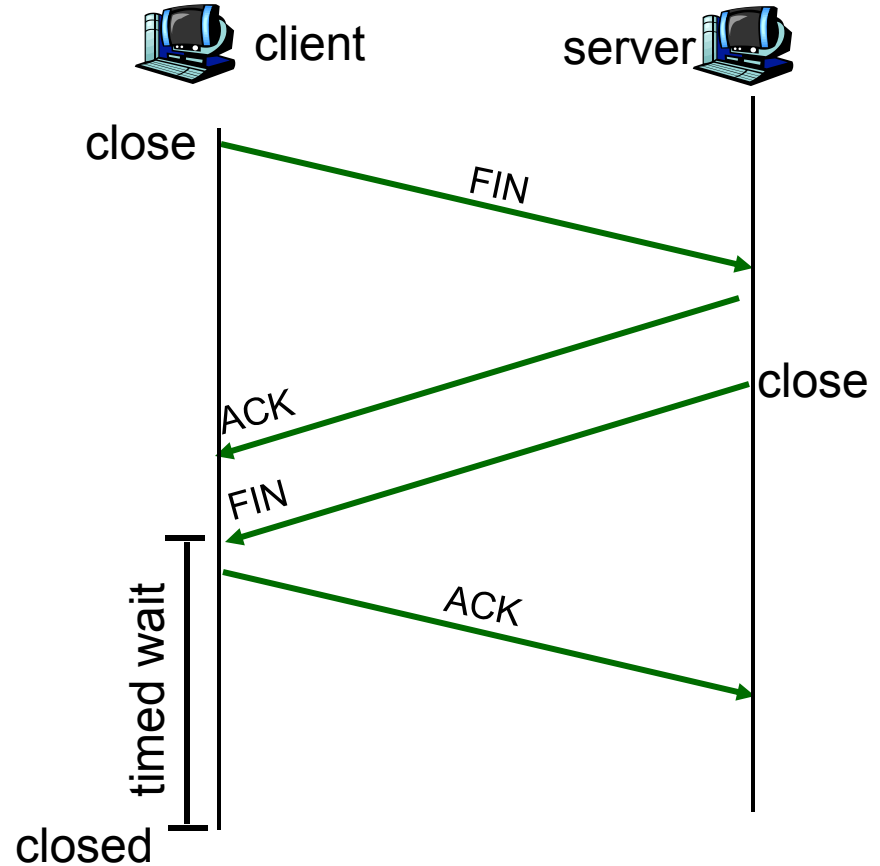
## Closing a connection:

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.



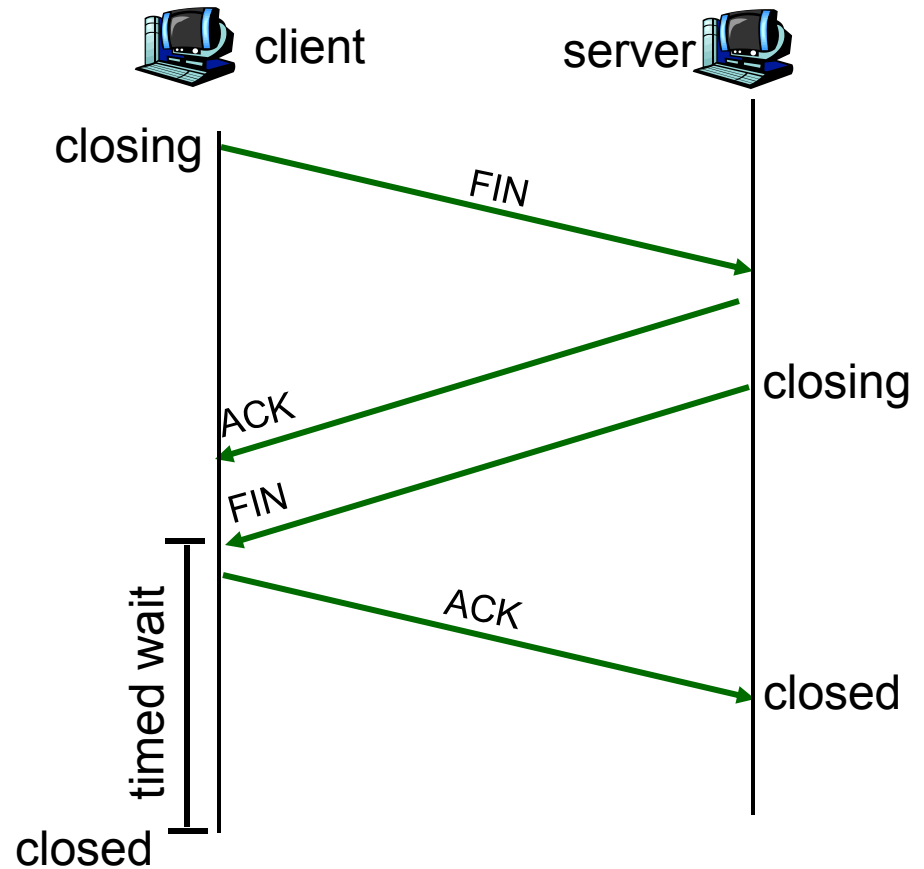
# TCP Connection Management (cont.)

**Step 3:** client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

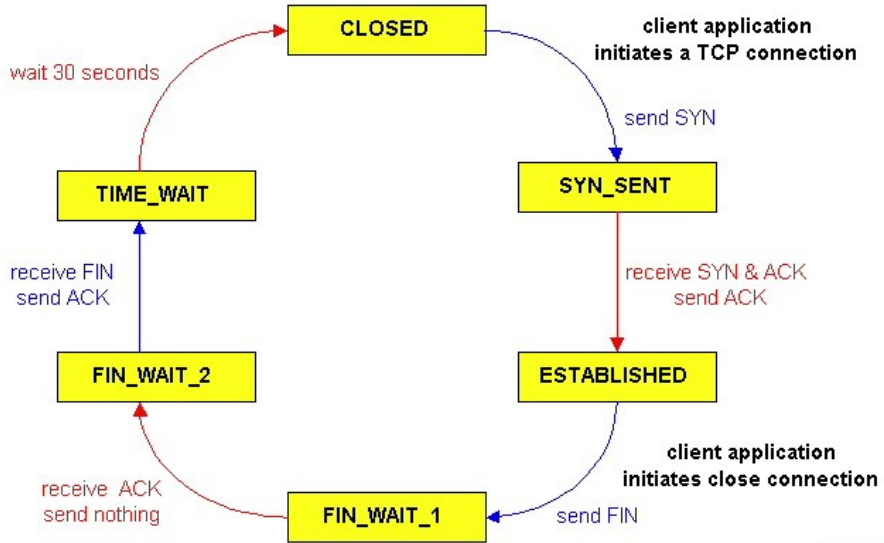
**Step 4:** server, receives ACK. Connection closed.

**Note:** with small modification, can handle simultaneous FINs.



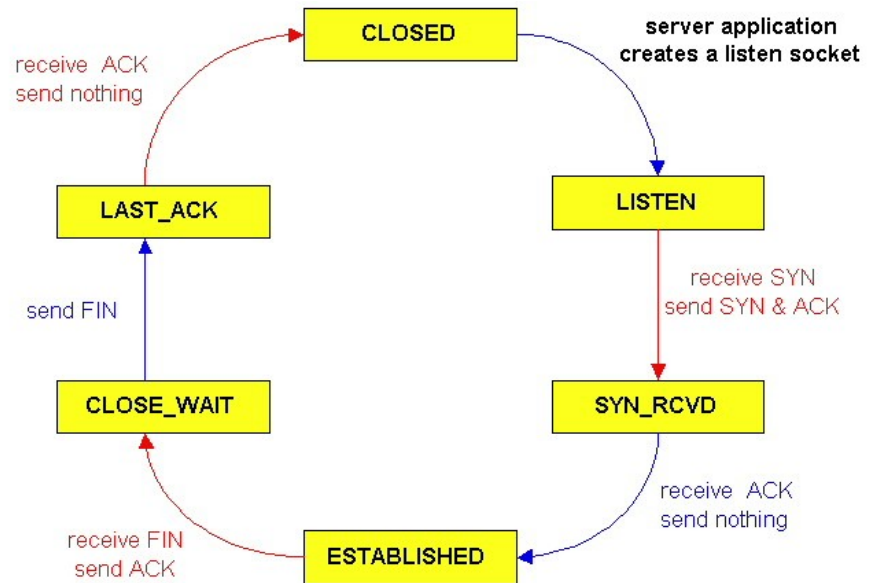


# TCP Connection Management (cont)



TCP client lifecycle

## TCP server lifecycle



# Chapter 4 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

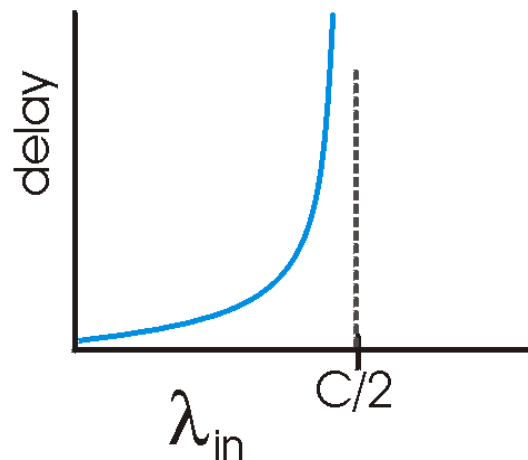
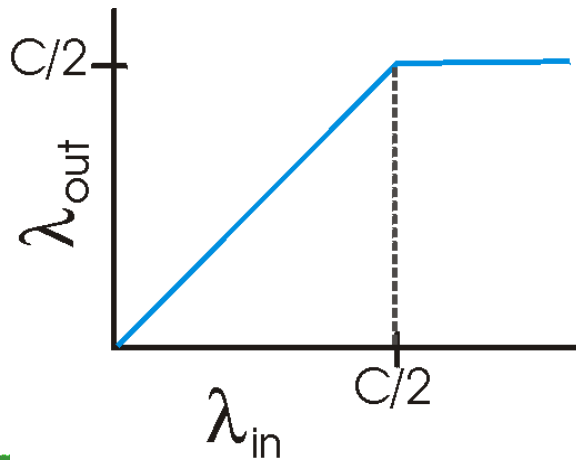
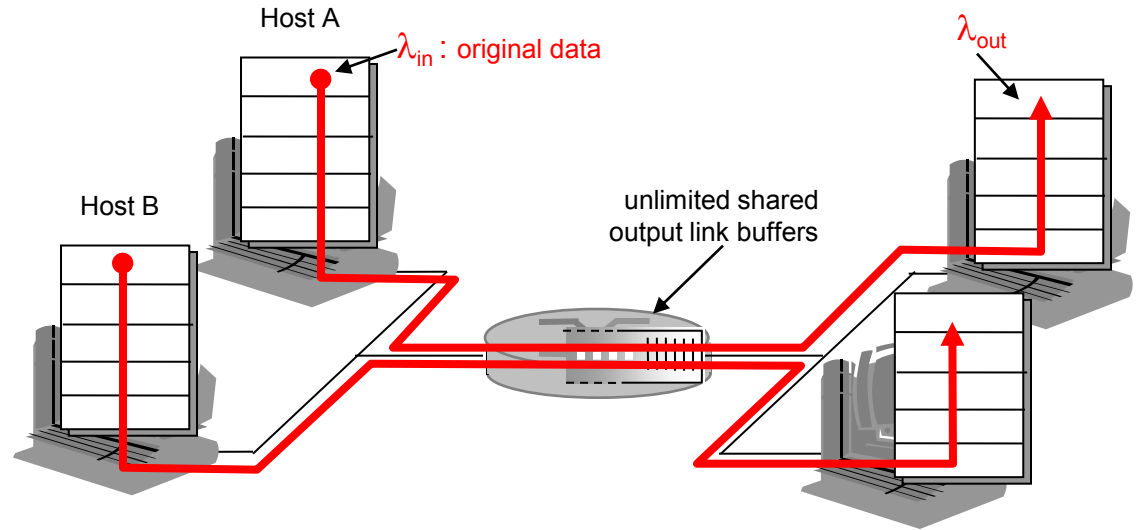
# Principles of Congestion Control

## Congestion:

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

# Causes/costs of congestion: scenario 1

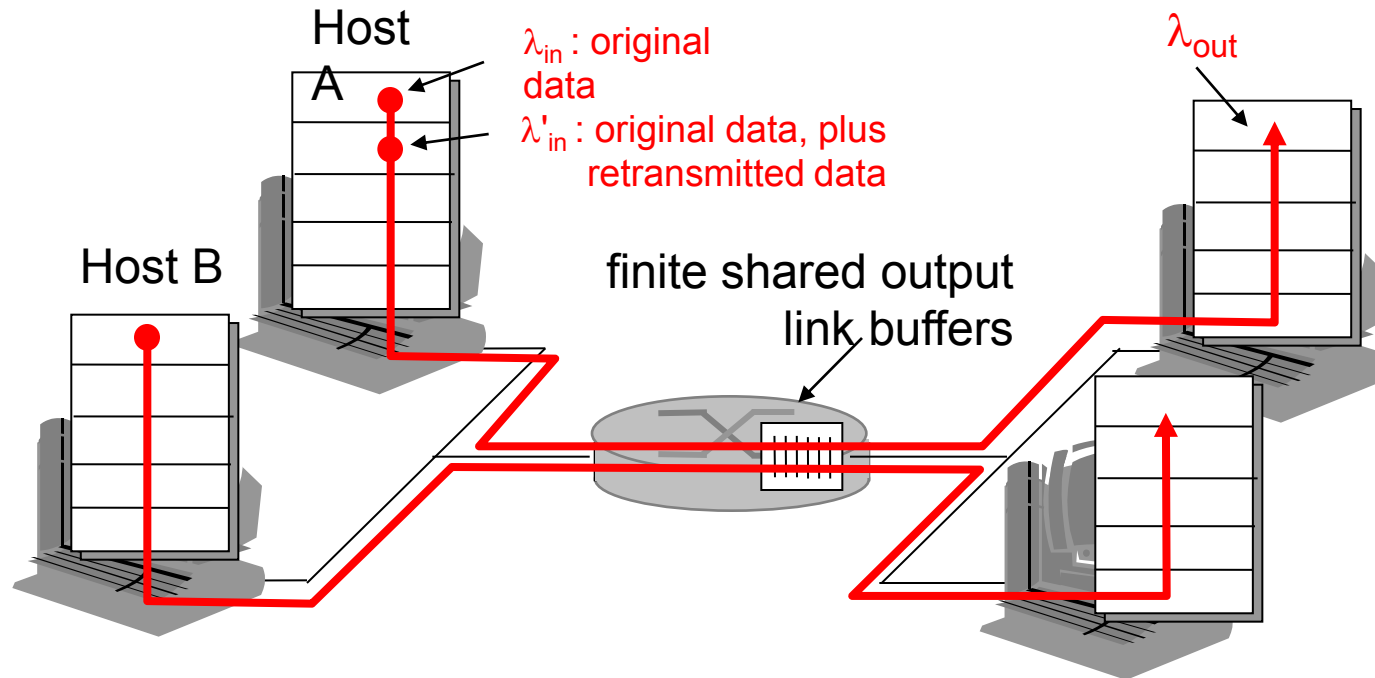
- two senders, two receivers
- one router, infinite buffers
- no retransmission



- large delays when congested
- maximum achievable throughput

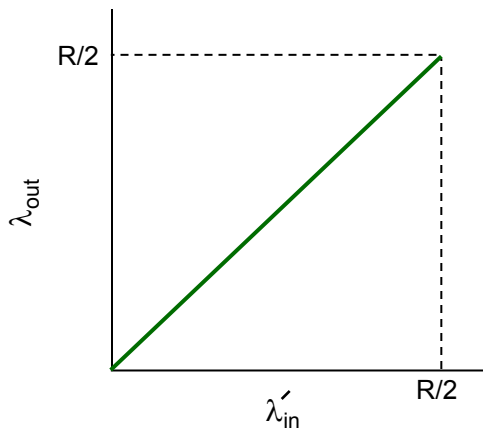
# Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet

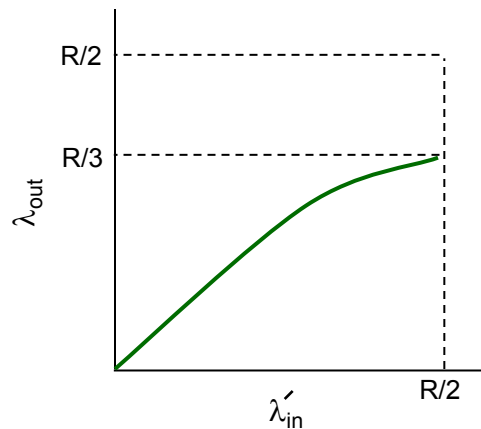


# Causes/costs of congestion: scenario 2

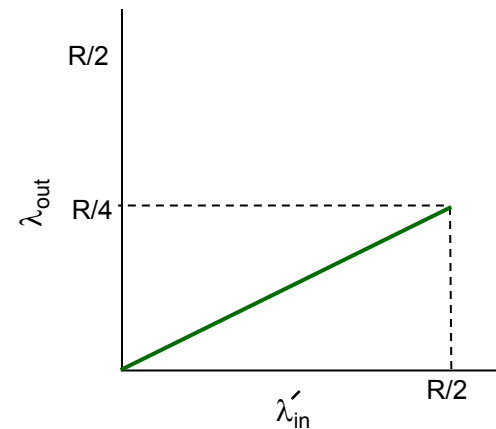
- always:  $\lambda_{in} = \lambda_{out}$  (goodput)
- “perfect” retransmission only when loss:  $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes  $\lambda'_{in}$  larger (than perfect case) for same  $\lambda_{out}$



a.



b.



c.

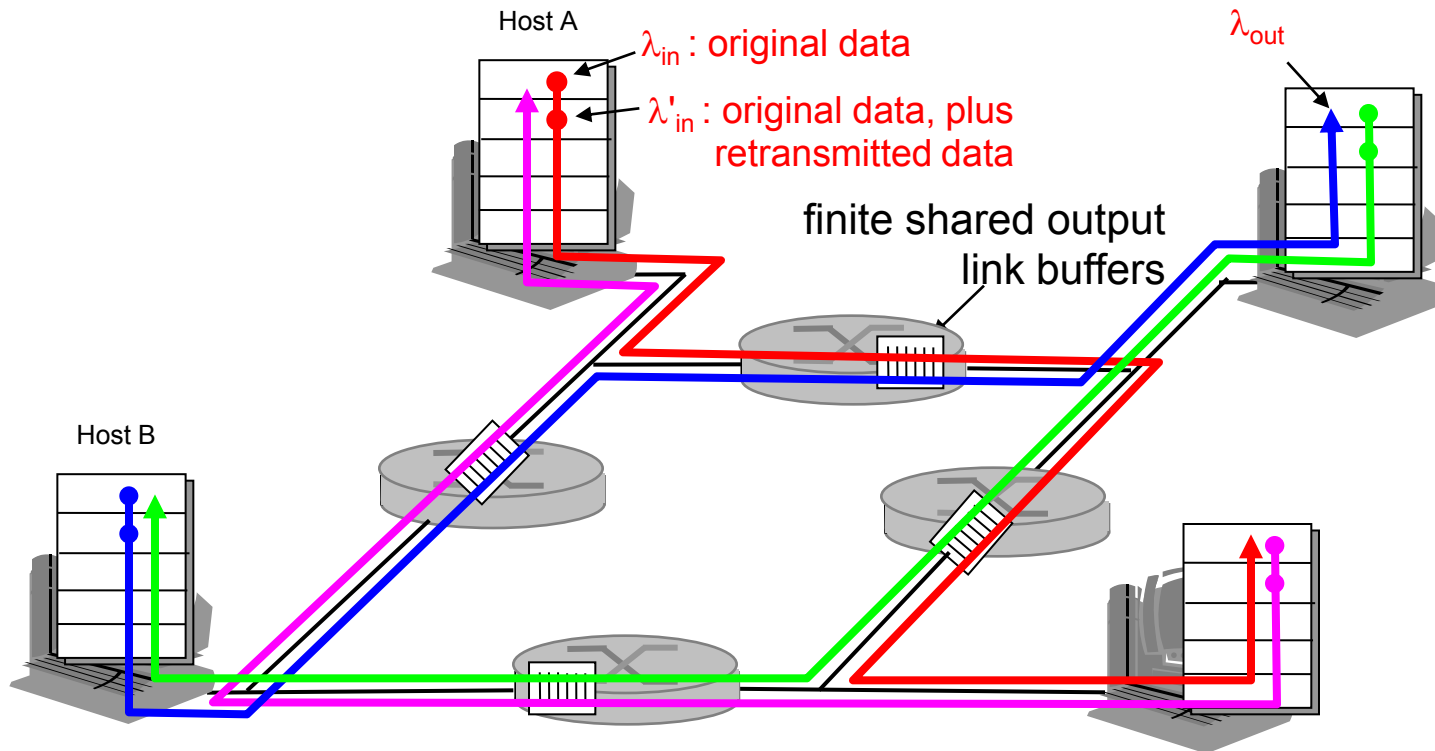
## “costs” of congestion:

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

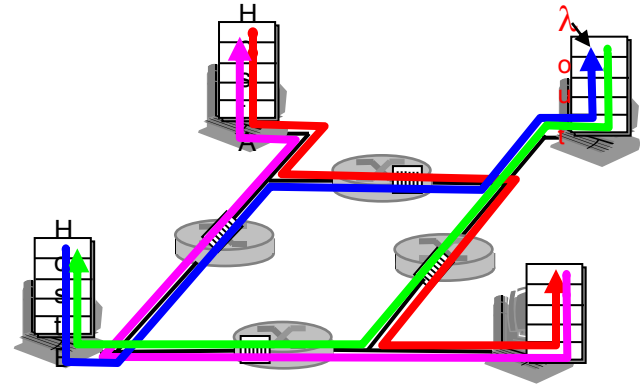
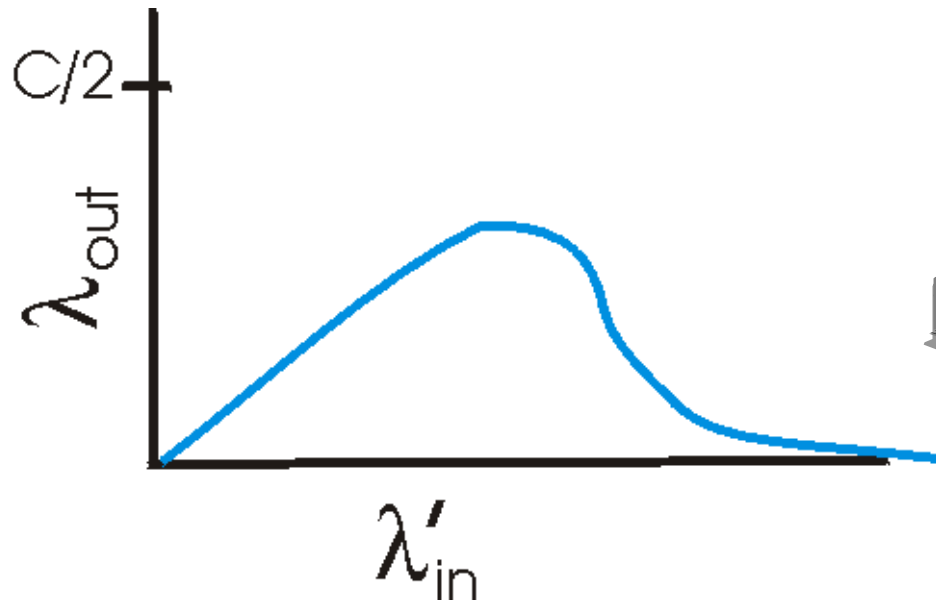
# Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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



## 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:

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

## Network-assisted congestion control:

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

# Case study: ATM ABR congestion control

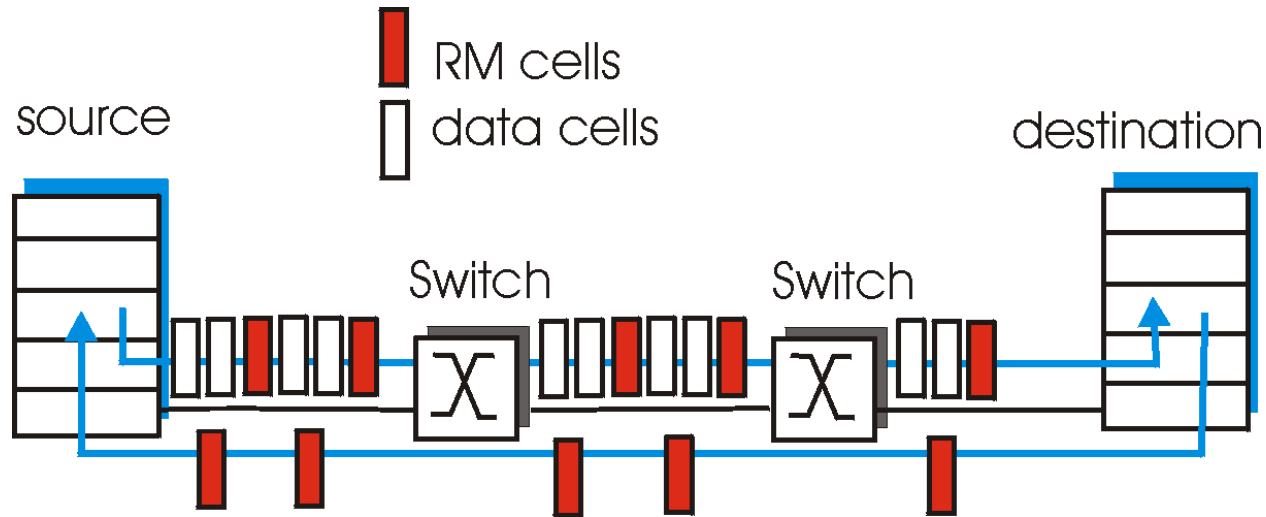
## ABR: available bit rate:

- “elastic service”
- if sender’s path “underloaded”:
  - sender should use available bandwidth
- if sender’s path congested:
  - sender throttled to minimum guaranteed rate

## RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches (“*network-assisted*”)
  - **NI bit**: no increase in rate (mild congestion)
  - **CI bit**: congestion indication
- RM cells returned to sender by receiver, with bits intact

# Case study: ATM ABR congestion control



- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender' send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

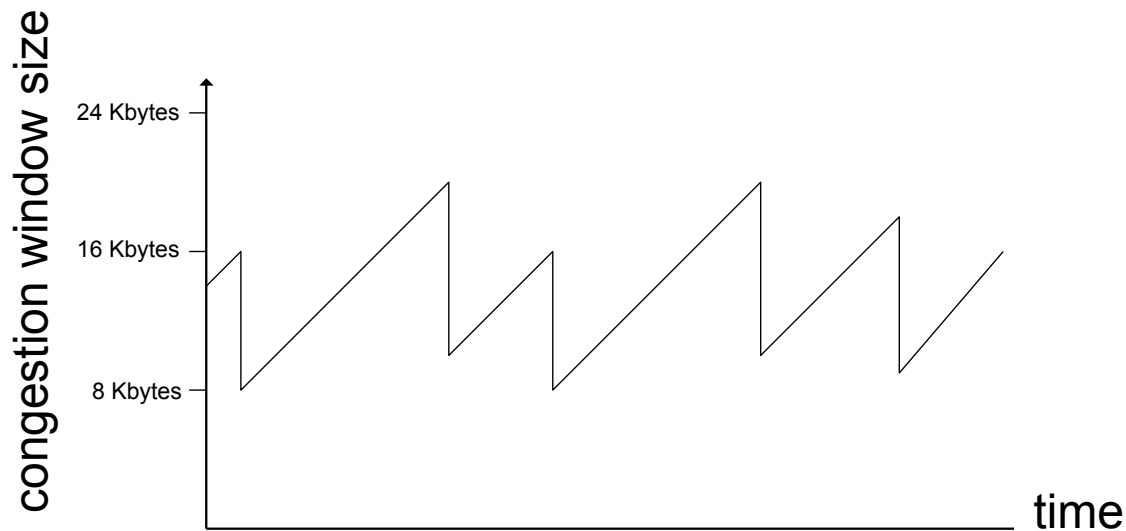
# Chapter 4 outline

- 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*: 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:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

- Roughly,

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

- **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:

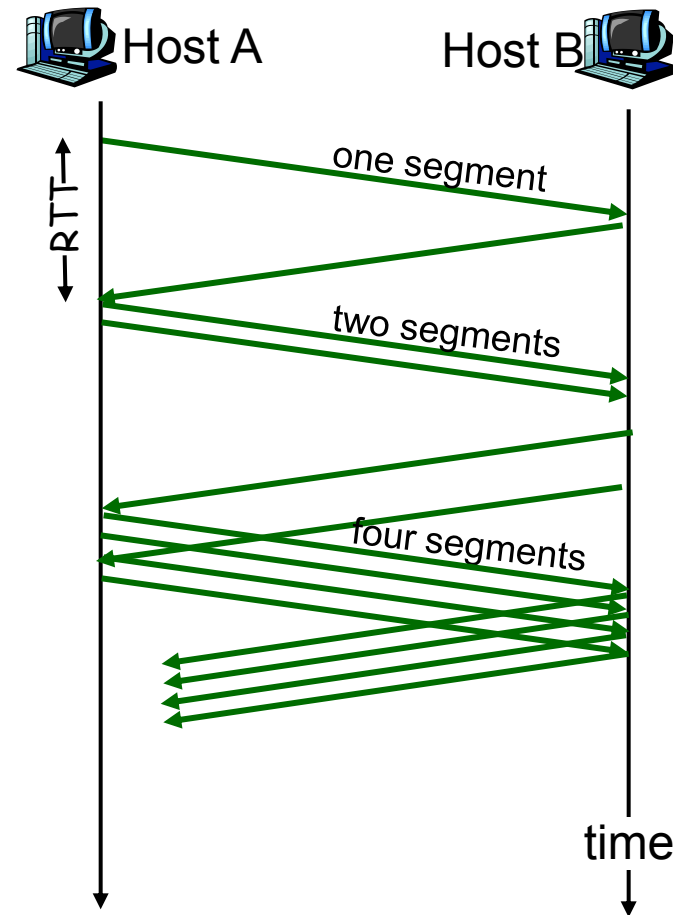
- AIMD
- slow start
- conservative after timeout events

# TCP Slow Start

- When connection begins, **CongWin = 1 MSS**
  - Example: MSS = 500 bytes & RTT = 1000 msec (1sec)
  - initial rate = 500 bytes/s
- available bandwidth may be  $\gg$  MSS/RTT
  - 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
- **Summary:** initial rate is slow but ramps up exponentially fast





# Refinement: inferring loss

- After 3 dup ACKs:
  - **CongWin** is cut in half
  - window then grows linearly
- But after timeout event:
  - **CongWin** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

## Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario

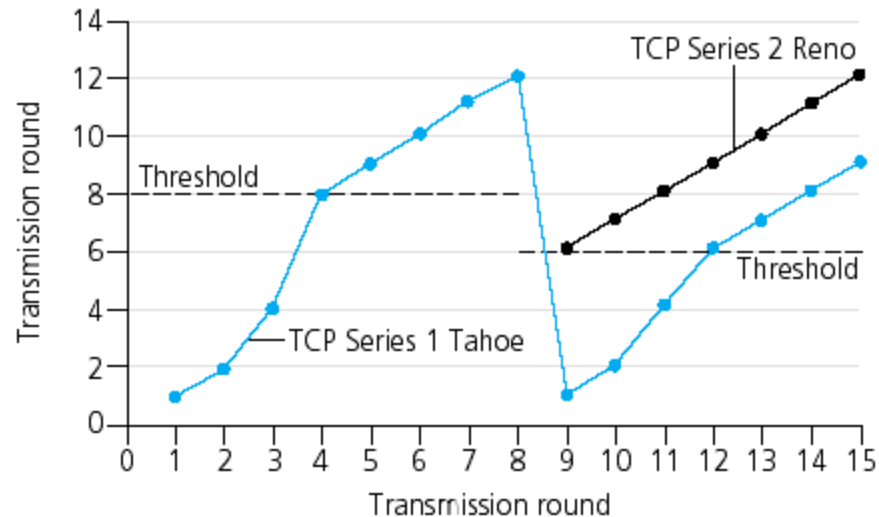
# Refinement

**Q:** When should the exponential increase switch to linear?

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

## Implementation:

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



# Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in **slow-start** phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in **congestion-avoidance** phase, window grows linearly.
- When a **triple duplicate ACK** occurs, **Threshold** set to **CongWin/2** and **CongWin** set to **Threshold**.
- 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	$\text{CongWin} = \text{CongWin} + \text{MSS}$ , If ( $\text{CongWin} > \text{Threshold}$ ) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	$\text{CongWin} = \text{CongWin} + \text{MSS} * (\text{MSS} / \text{CongWin})$	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	$\text{Threshold} = \text{CongWin} / 2$ , $\text{CongWin} = \text{Threshold}$ , Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	$\text{Threshold} = \text{CongWin} / 2$ , $\text{CongWin} = 1 \text{ MSS}$ , 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

- What's the average throughput of TCP as a function of window size and RTT?
  - Ignore slow start
- Let  $W$  be the window size when loss occurs.
- When window is  $W$ , throughput is  $W/RTT$
- Just after loss, window drops to  $W/2$ , throughput to  $W/2RTT$ .
- Average throughput:  $.75 W/RTT$

# Chapter 4: Summary

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

## Next:

- Networked Multimedia

# Thank you

Any questions?