

# Transport Layer

## Chapter 3

# 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

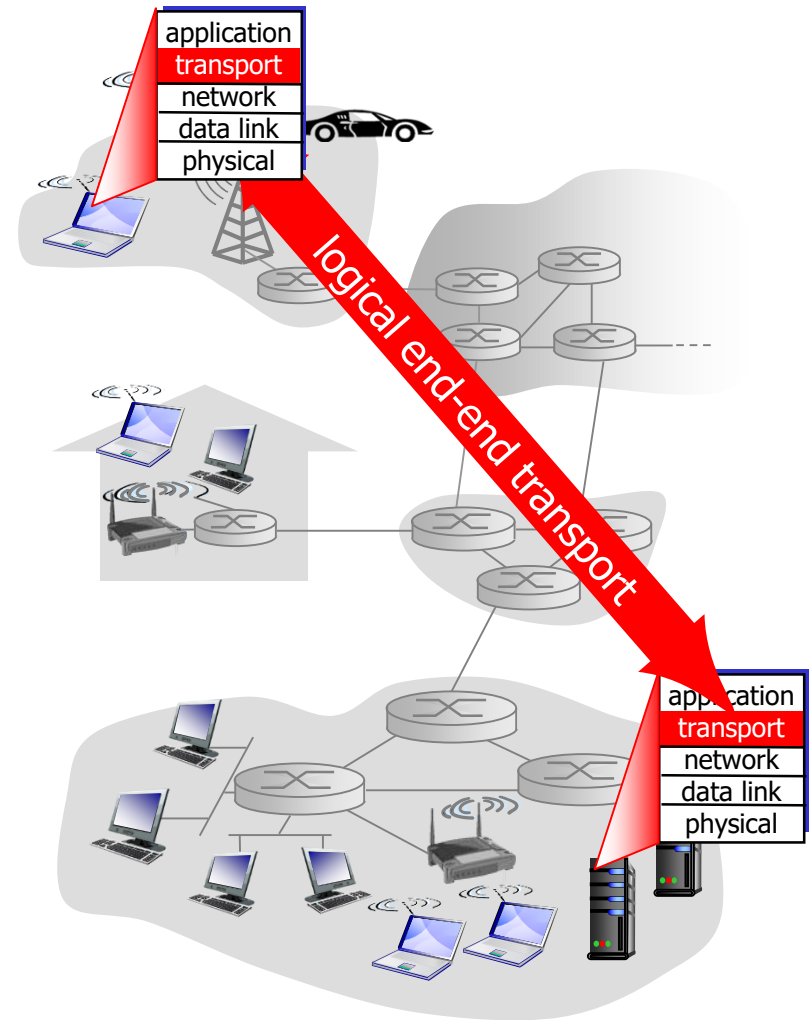
- segment structure
- connection management
- reliable data transfer
- flow control

## 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

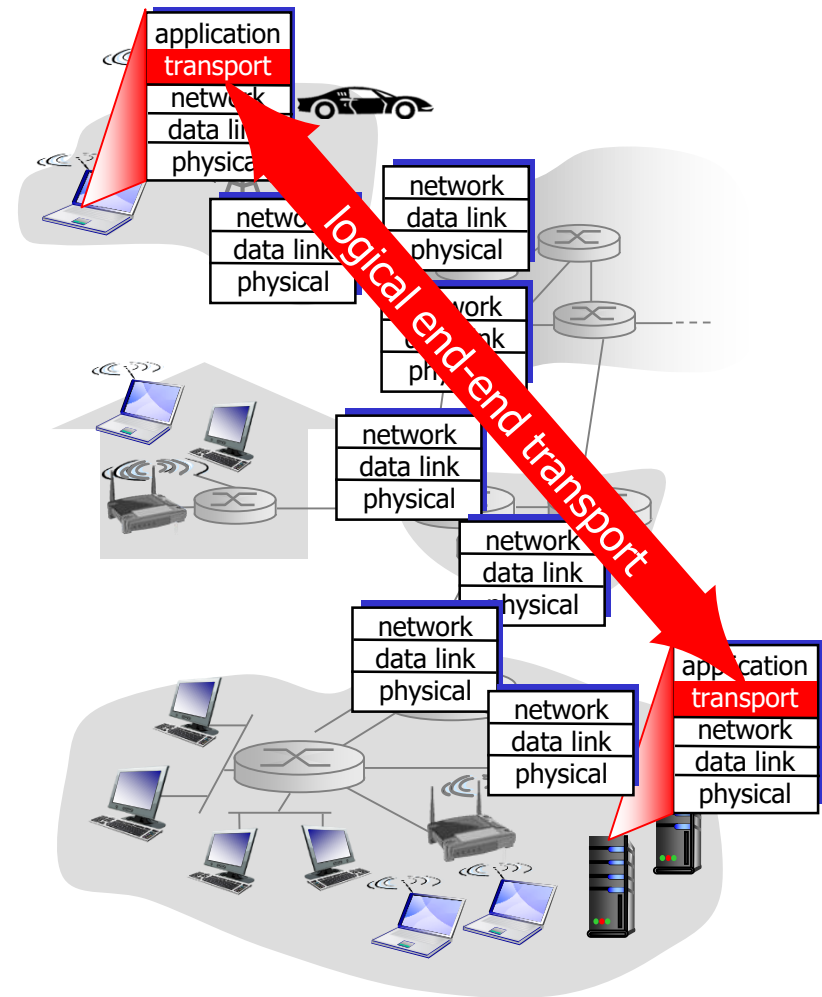
## *household analogy:*

*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 in-house 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
- connection management
- reliable data transfer
- flow control

3.6 principles of congestion control

3.7 TCP congestion control

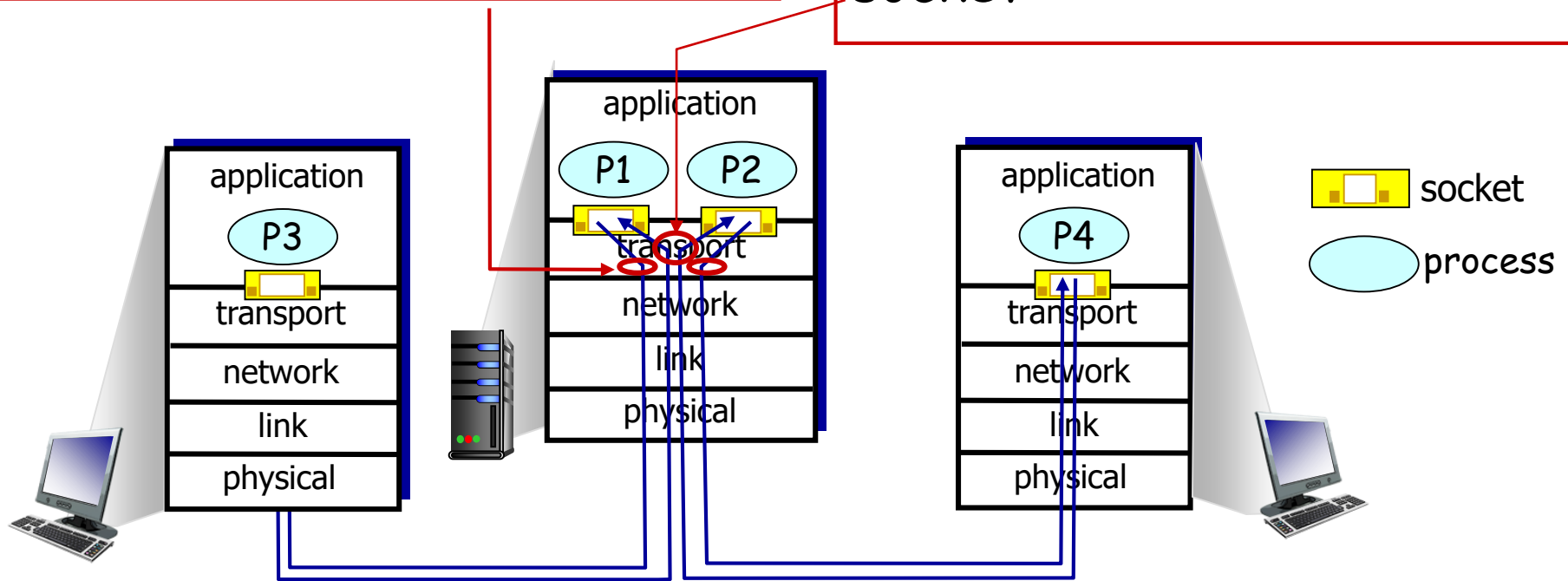
# Multiplexing/demultiplexing

*multiplexing at sender:*

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

*demultiplexing at receiver:*

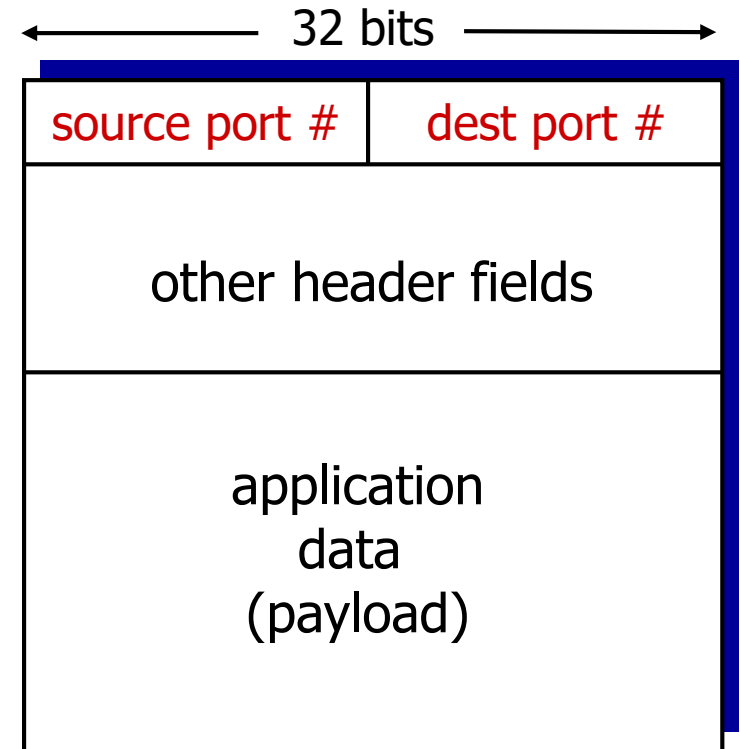
use header info to deliver received segments to correct socket





# How demultiplexing works

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



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 #



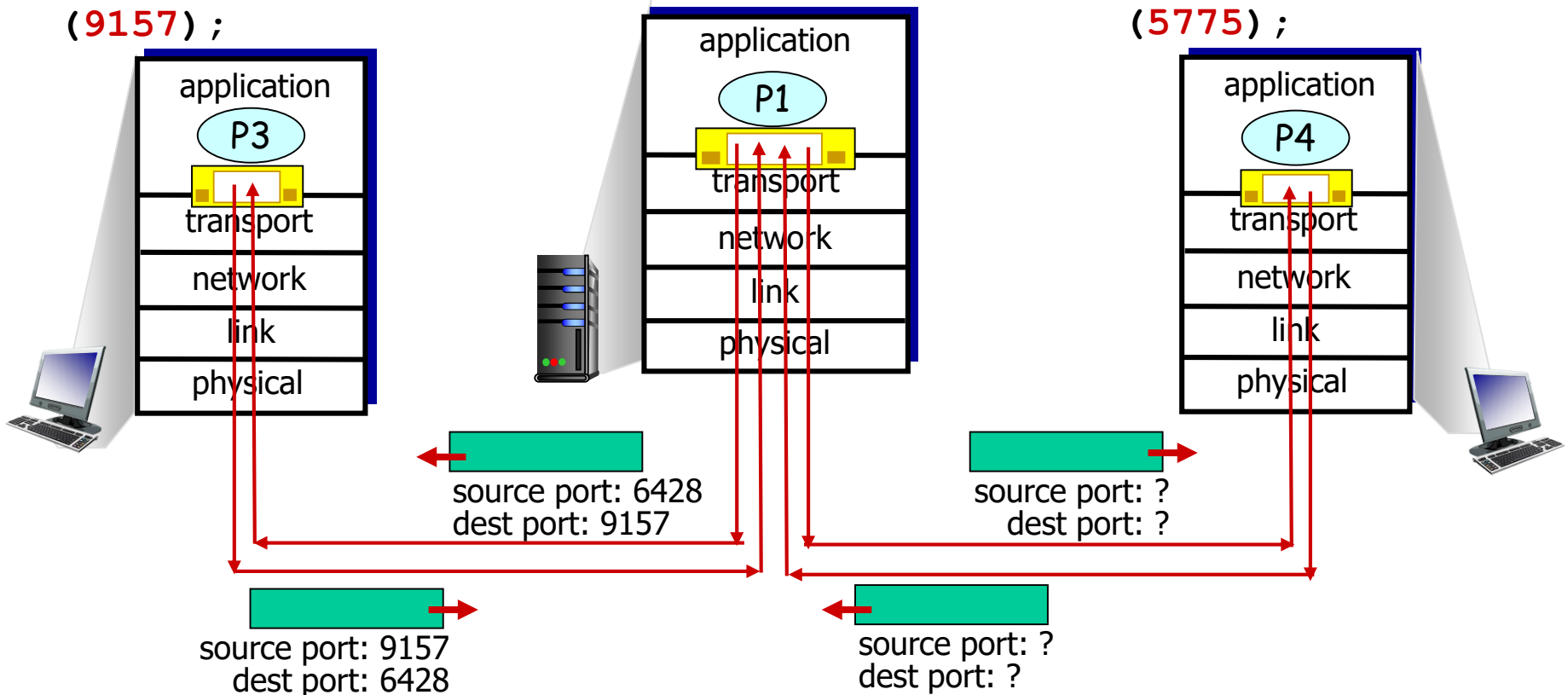
IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

# Connectionless demux: example

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

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

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

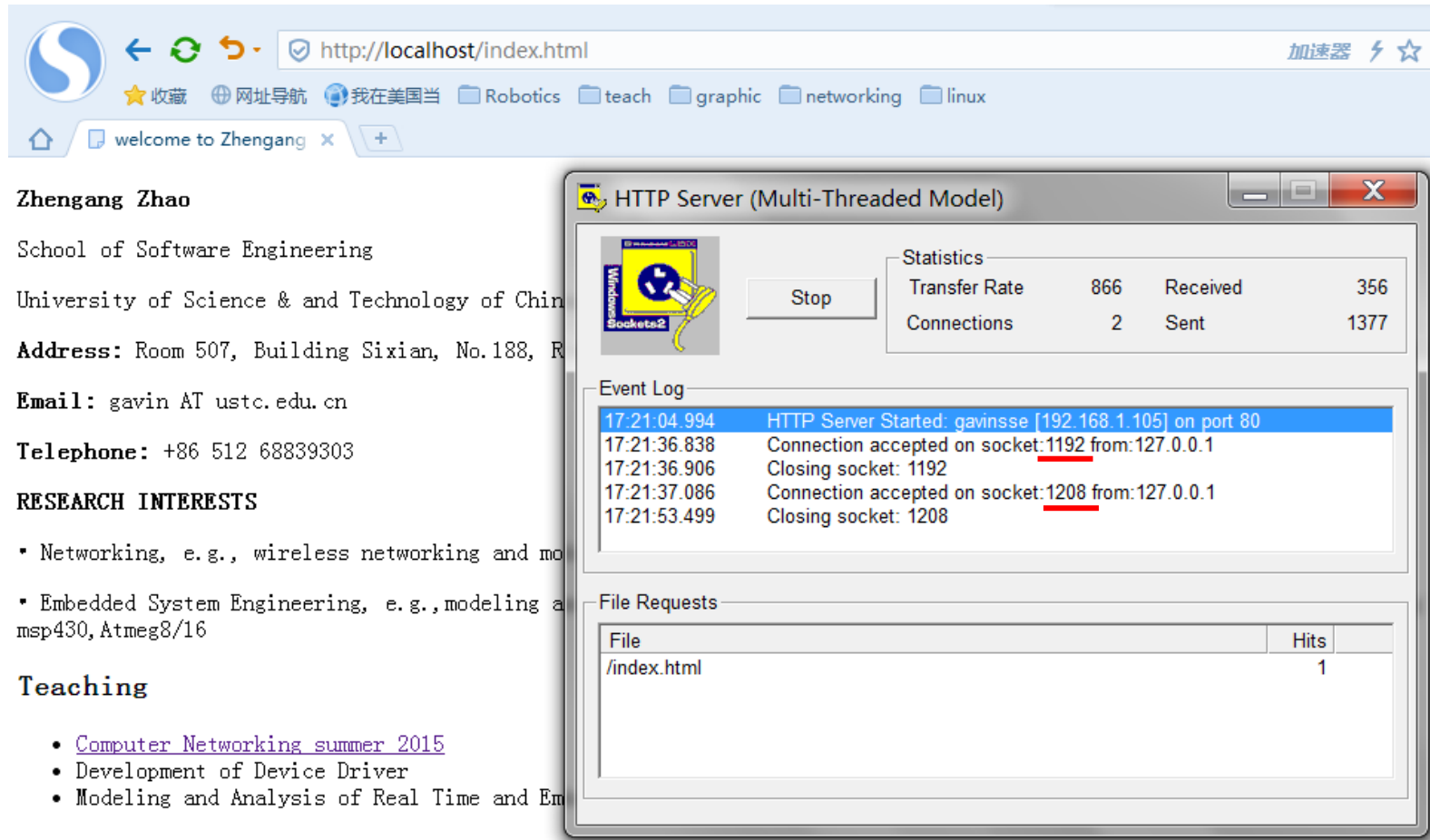


# Connection-oriented demux

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

❖ recall:



The screenshot shows a web browser window displaying a personal page for Zhengang Zhao. The browser's address bar shows 'http://localhost/index.html'. The page content includes contact information, research interests, and teaching topics. Overlaid on the bottom right is a window titled 'HTTP Server (Multi-Threaded Model)'. This window displays server statistics and an event log. The statistics show a transfer rate of 866, 356 received bytes, 2 connections, and 1377 sent bytes. The event log shows the server starting on port 80, accepting connections on sockets 1192 and 1208, and closing them. A file requests table at the bottom shows one hit for '/index.html'.

**Zhengang Zhao**  
School of Software Engineering  
University of Science & Technology of China  
**Address:** Room 507, Building Sixian, No.188, R  
**Email:** gavin AT ustc.edu.cn  
**Telephone:** +86 512 68839303

**RESEARCH INTERESTS**

- Networking, e.g., wireless networking and mo
- Embedded System Engineering, e.g., modeling a msp430, Atmeg8/16

**Teaching**

- [Computer Networking summer 2015](#)
- Development of Device Driver
- Modeling and Analysis of Real Time and Em

**HTTP Server (Multi-Threaded Model)**

Stop

**Statistics**

Transfer Rate	866	Received	356
Connections	2	Sent	1377

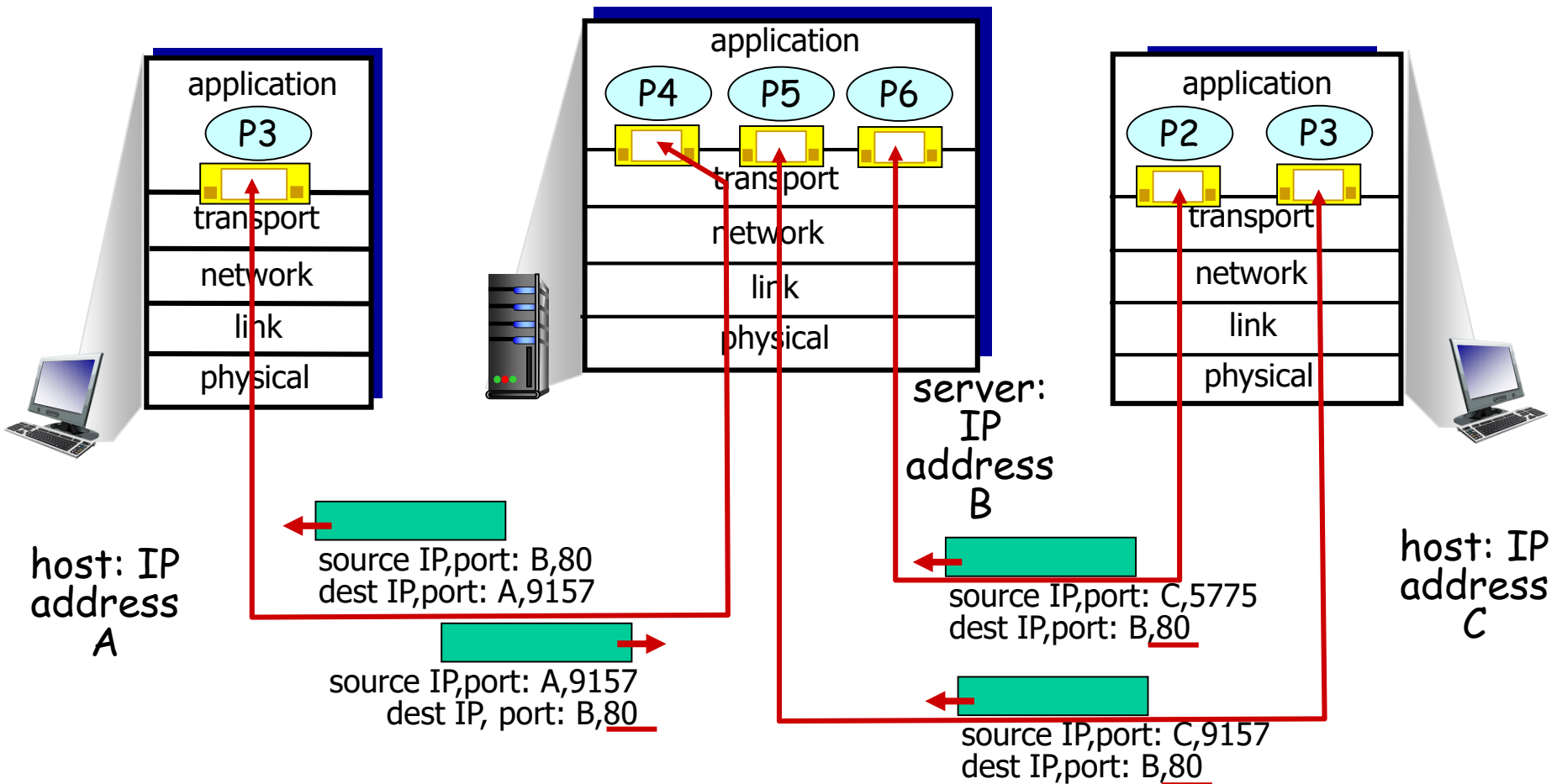
**Event Log**

17:21:04.994	HTTP Server Started: gavinssse [192.168.1.105] on port 80
17:21:36.838	Connection accepted on socket: 1192 from: 127.0.0.1
17:21:36.906	Closing socket: 1192
17:21:37.086	Connection accepted on socket: 1208 from: 127.0.0.1
17:21:53.499	Closing socket: 1208

**File Requests**

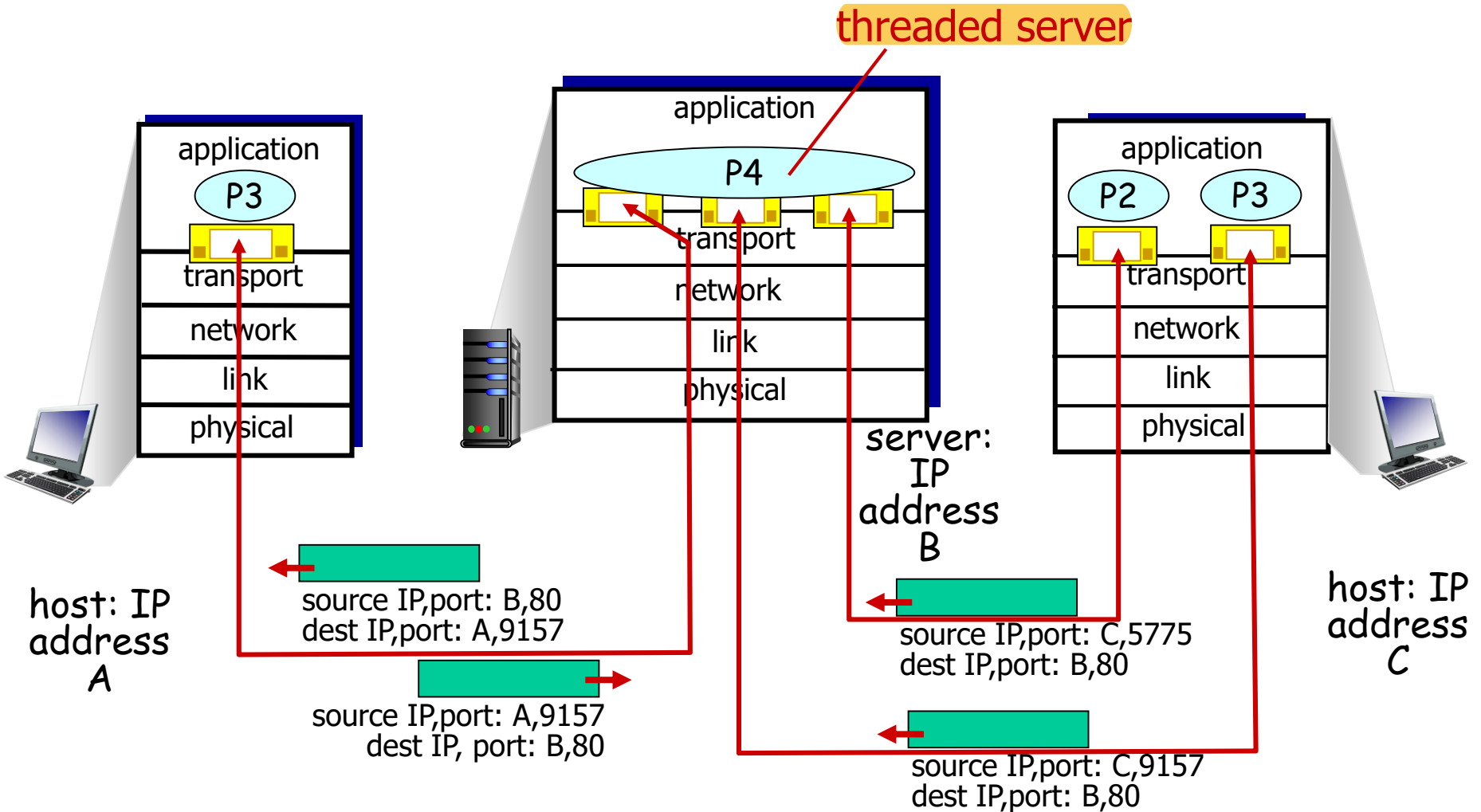
File	Hits
/index.html	1

# 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
- connection management
- reliable data transfer
- flow control

3.6 principles of congestion control

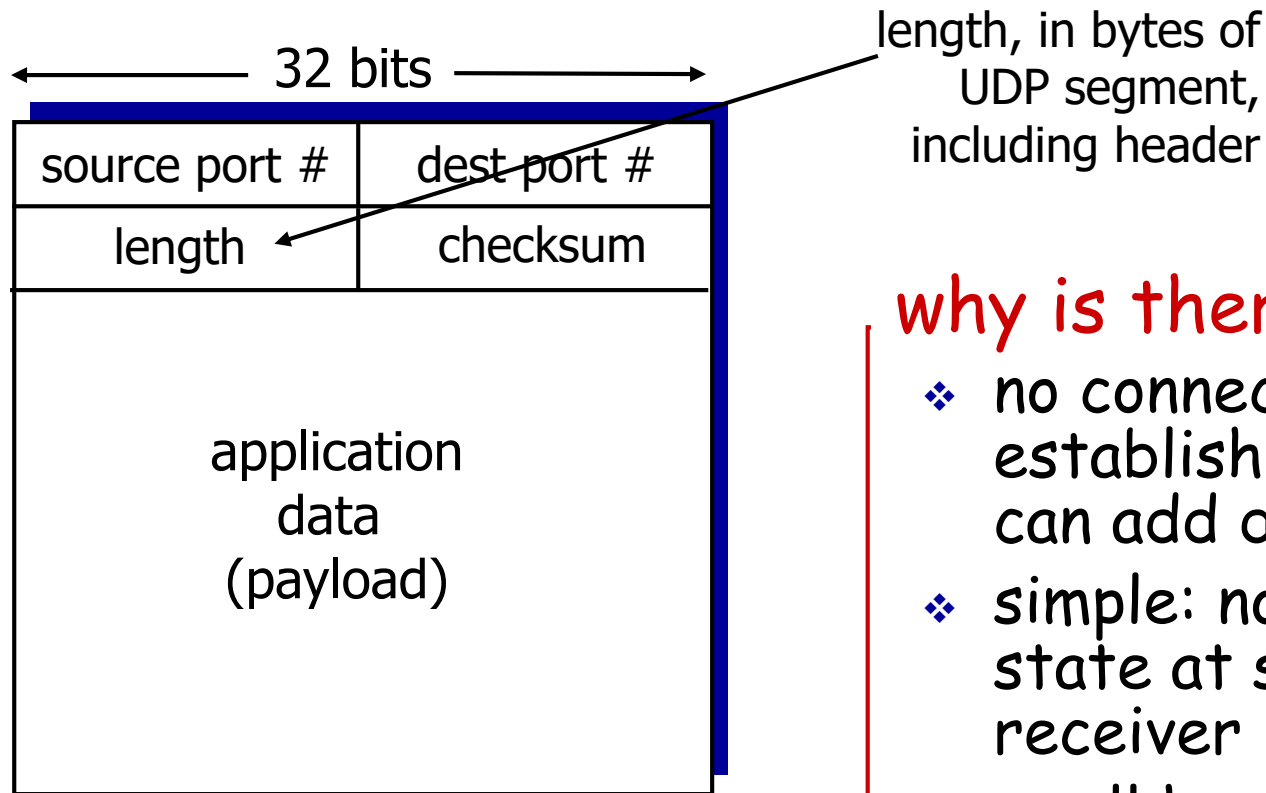
3.7 TCP congestion control



# 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

## 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 ....*

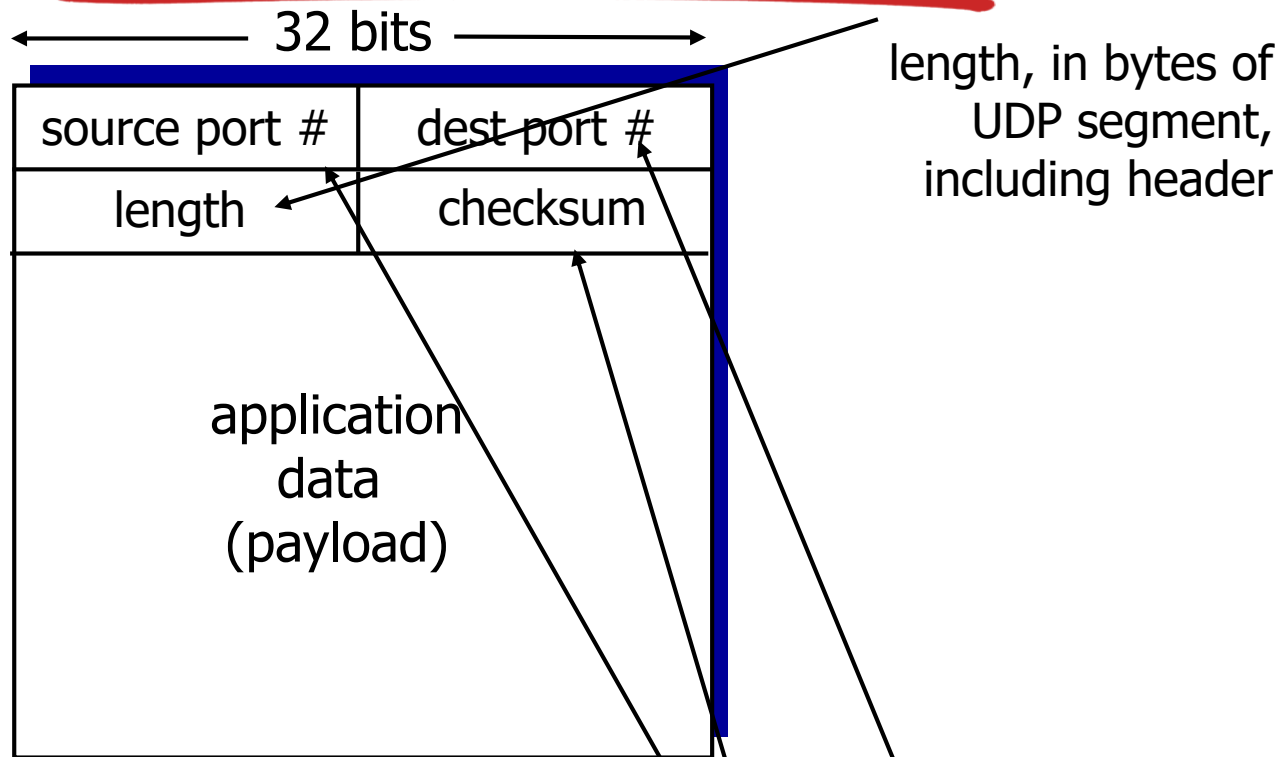
# Internet checksum: example

example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1

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

# UDP: example



UDP segment

```
Frame 1: 73 bytes on wire (584 bits), 73 bytes captured (584 bits) on interface 0
Ethernet II, Src: CompalCo_b8:59:b1 (00:16:d4:b8:59:b1), Dst: Cisco_4b:c8:39:78
Internet Protocol Version 4, Src: 192.168.100.138 (192.168.100.138), Dst: 192.168.100.1
User Datagram Protocol, Src Port: polestar (1060), Dst Port: domain (53)
  Source port: polestar (1060)
  Destination port: domain (53)
  Length: 39
  Checksum: 0x6d5a [validation disabled]
  Domain Name System (query)
```

# 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

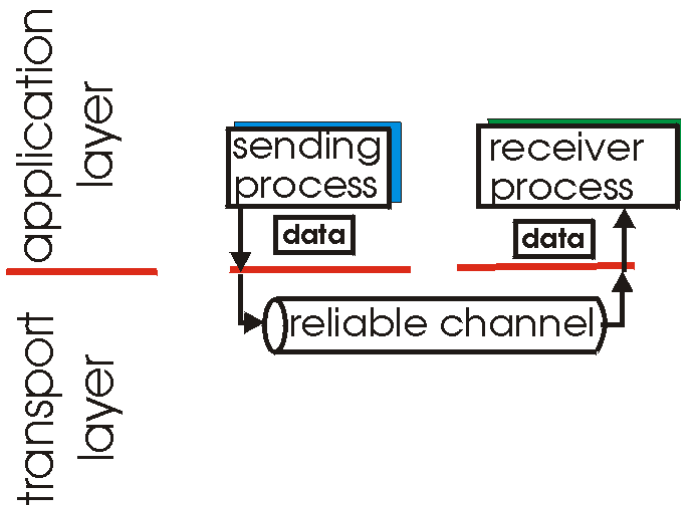
- segment structure
- reliable data transfer
- flow control
- connection management

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!

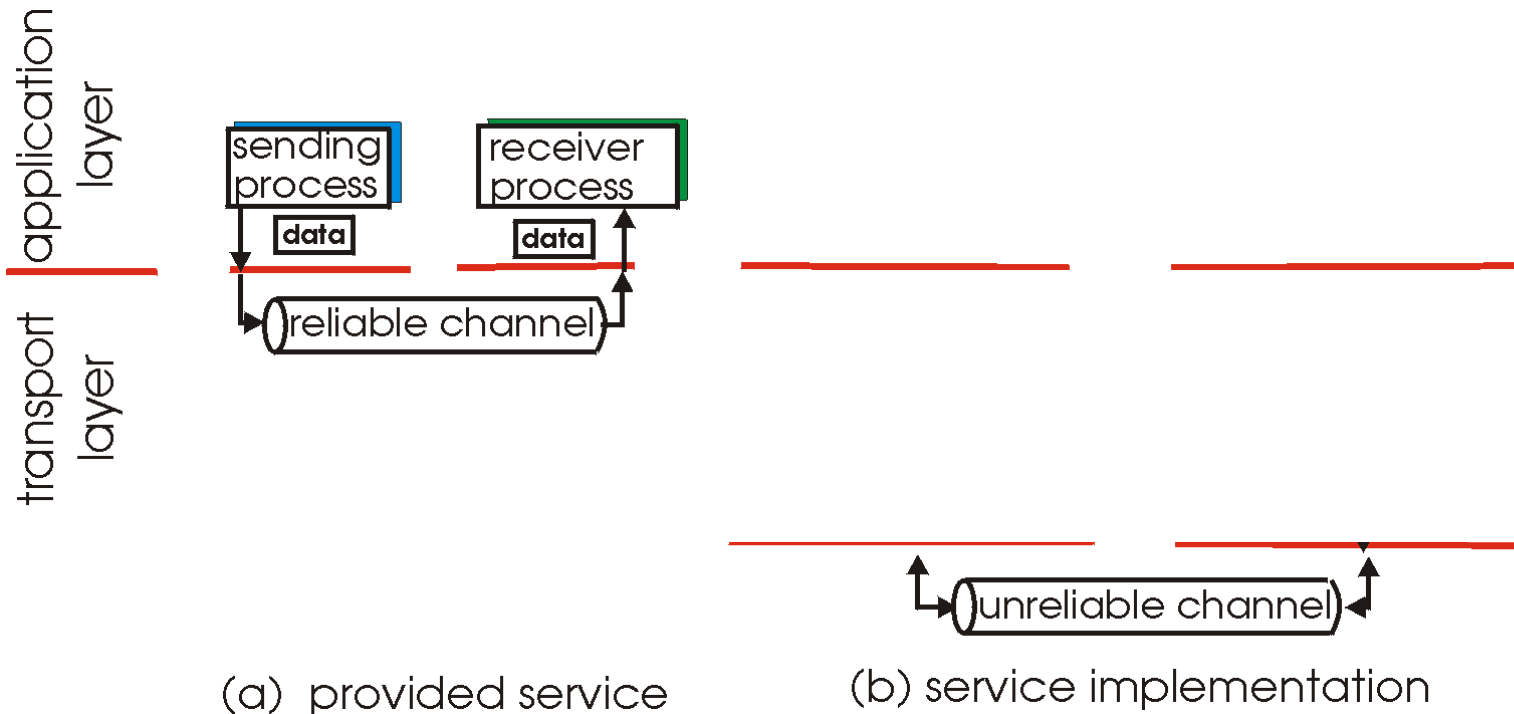


(a) provided service

- ❖ 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!

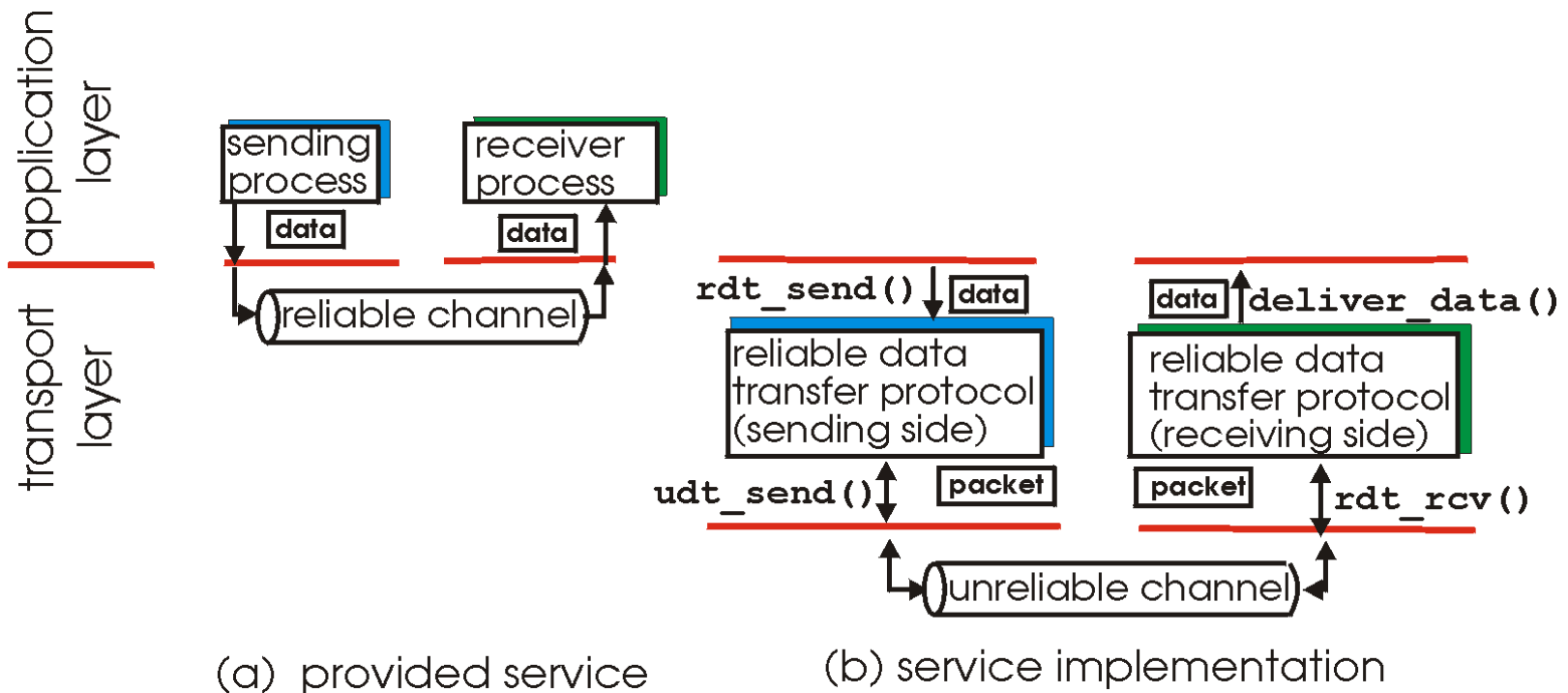


- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)



# Principles of reliable data transfer

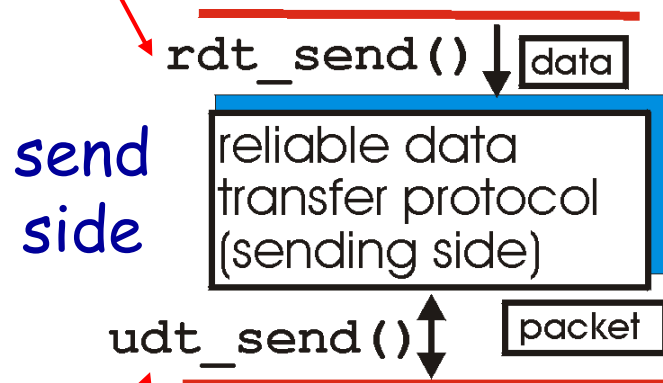
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



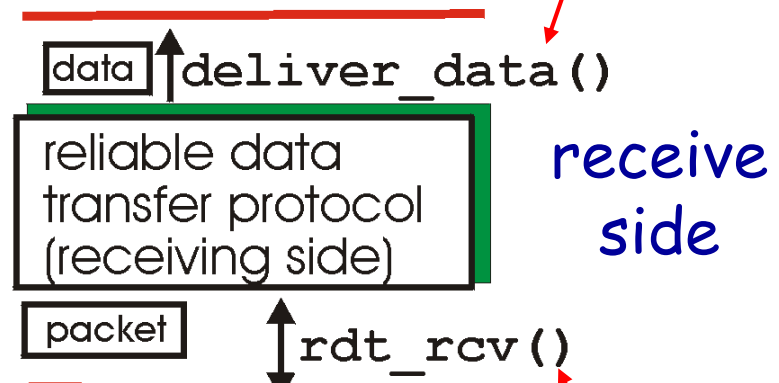
- ❖ characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# Reliable data transfer: getting started

**rdt\_send()** : called from above,  
(e.g., by app.). Passed data to  
deliver to receiver upper layer



**deliver\_data()** : called by  
**rdt** to deliver data to upper



**udt\_send()** : called by rdt,  
to transfer packet over  
unreliable channel to receiver

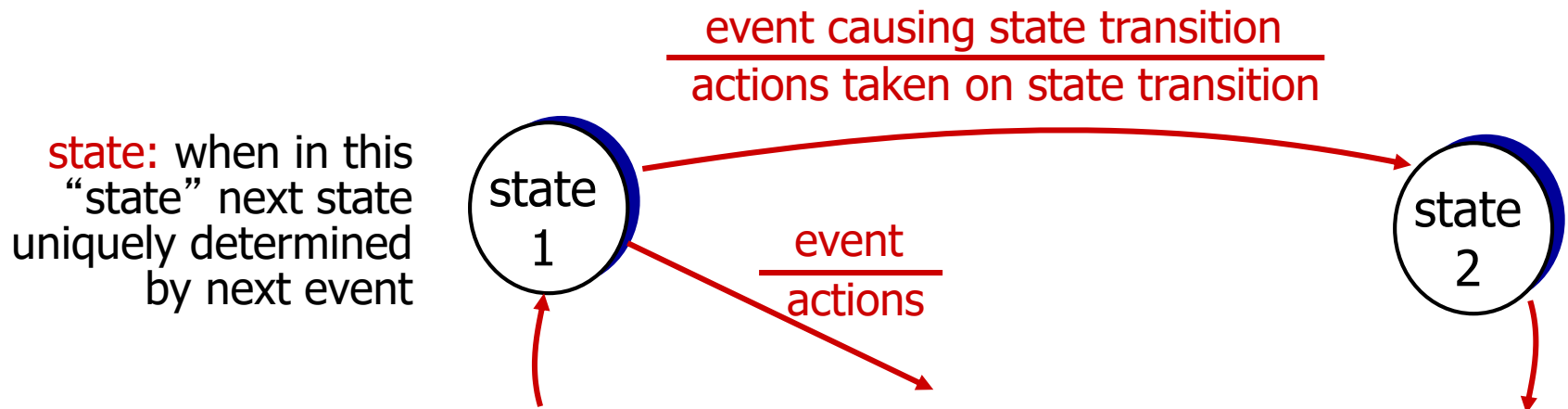
**rdt\_rcv()** : called when packet  
arrives on rcv-side of channel



# Reliable data transfer: getting started

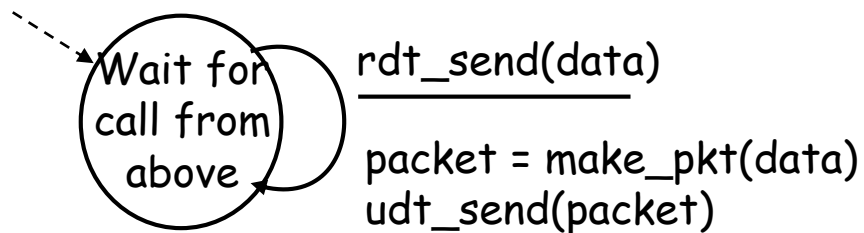
we' ll:

- ❖ 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

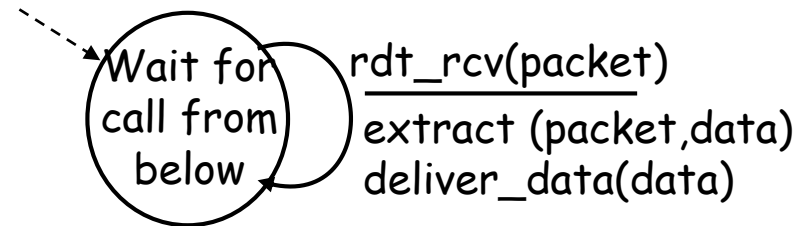


# rdt1.0: reliable transfer over a reliable channel

- ❖ 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



sender



receiver

# rdt2.0: channel with bit errors

- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors in UDP as example
- ❖ 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 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: discussion

## sender:

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

## receiver:

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



## 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*

# rdt3.0: channels with errors and loss

## new assumption:

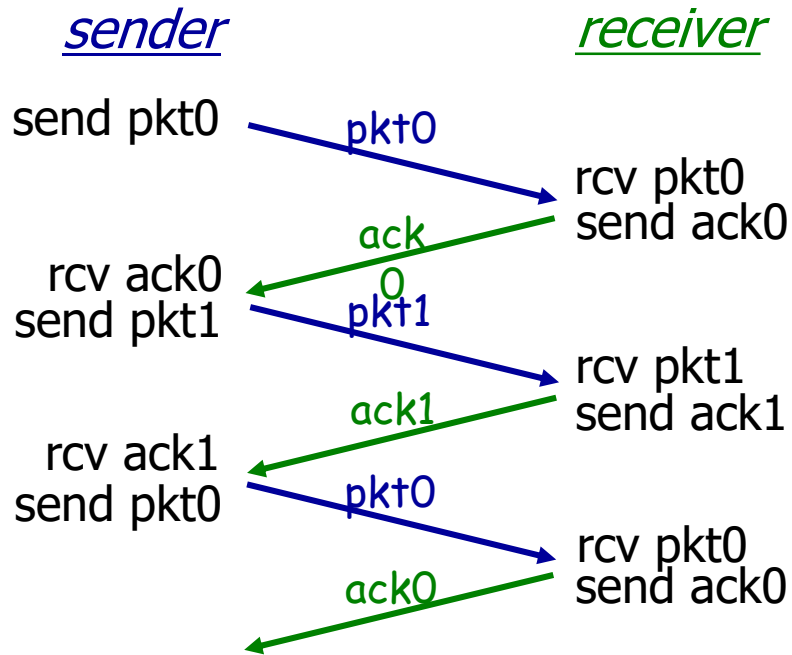
underlying channel  
can also lose  
packets (data,  
ACKs)

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

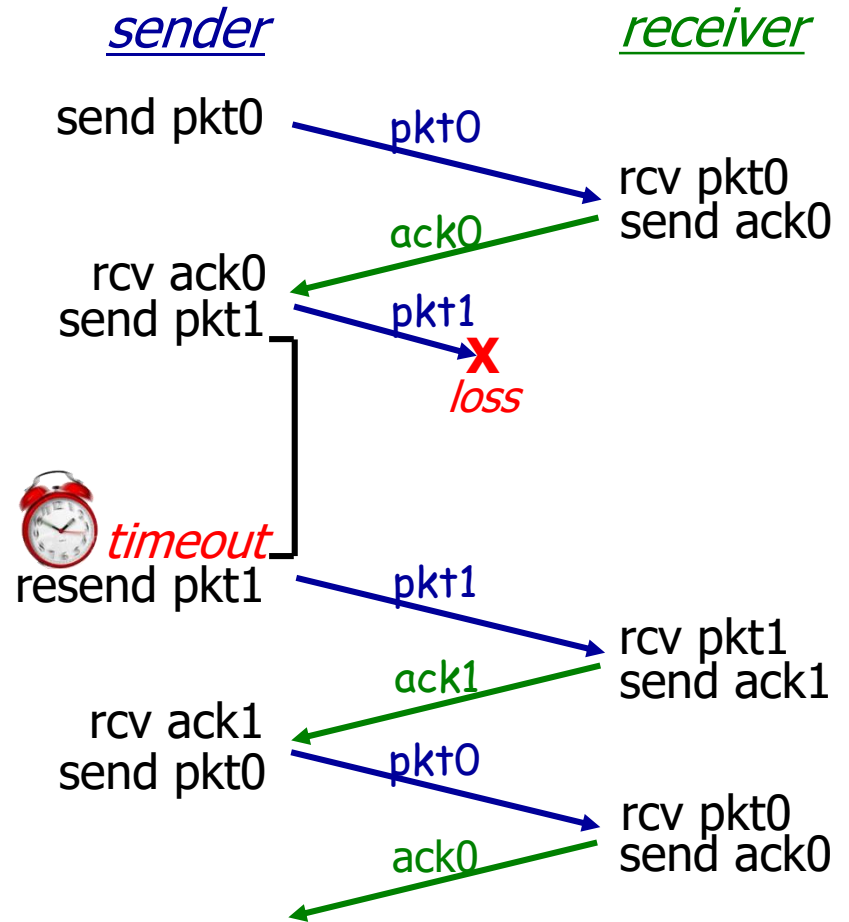
## approach: sender waits “reasonable” amount of time for ACK

- ❖ 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 in action

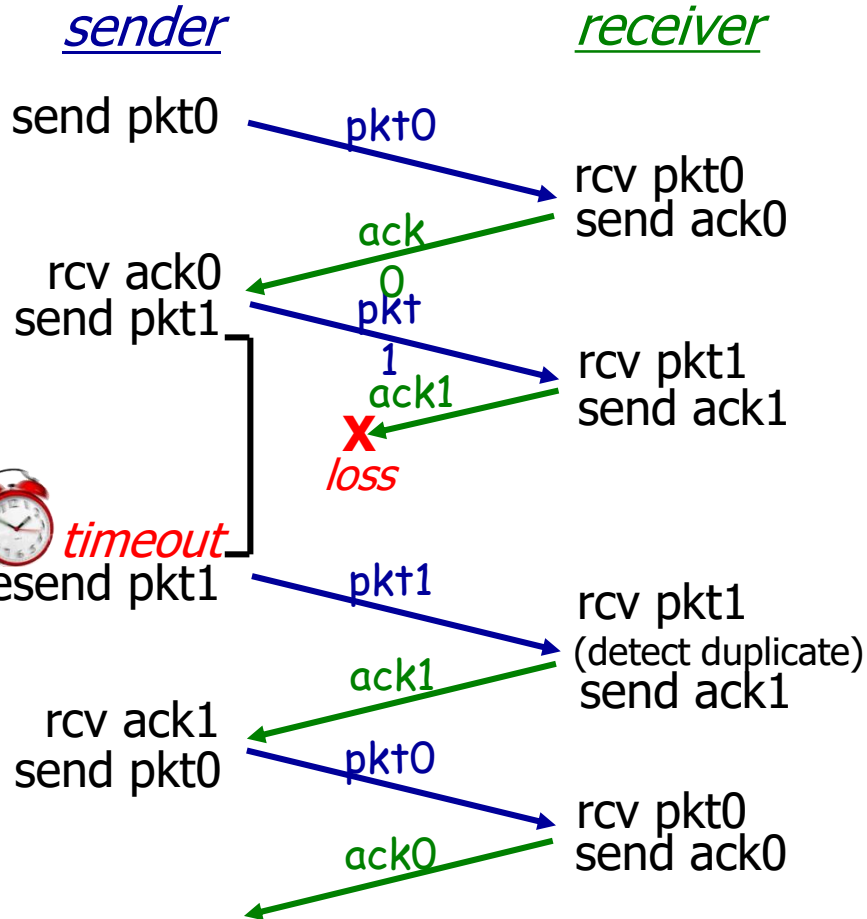


(a) no loss

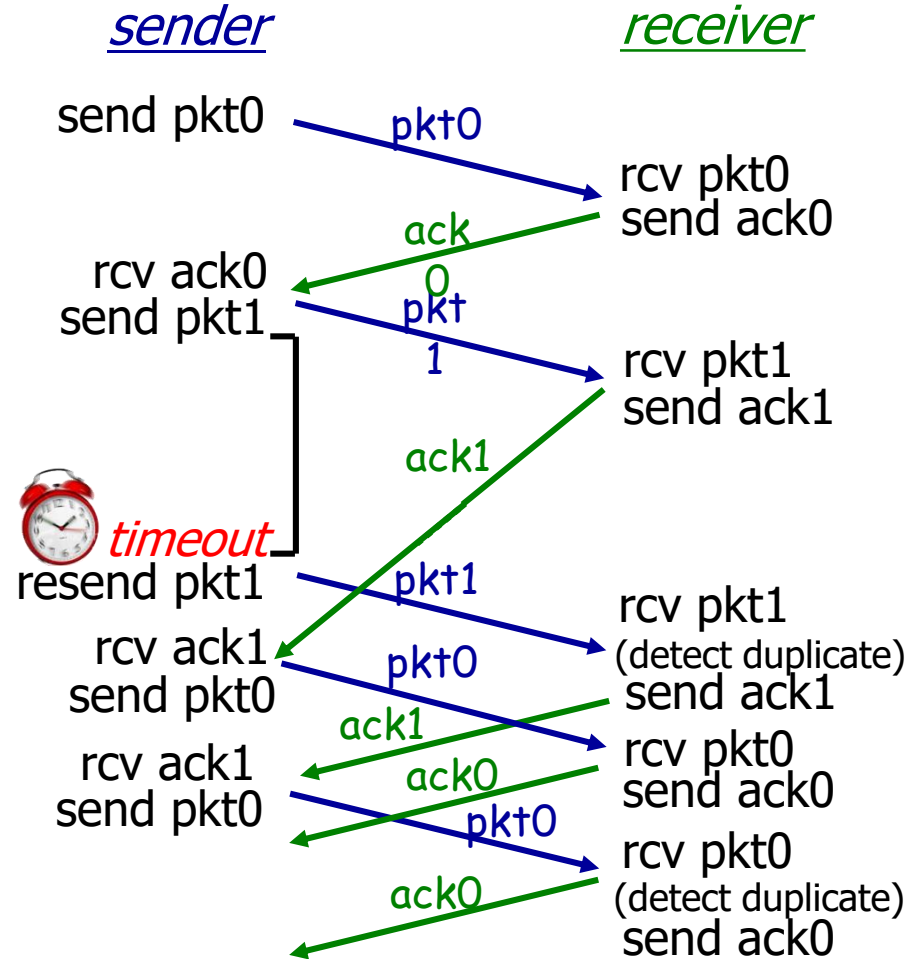


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Performance of rdt3.0

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

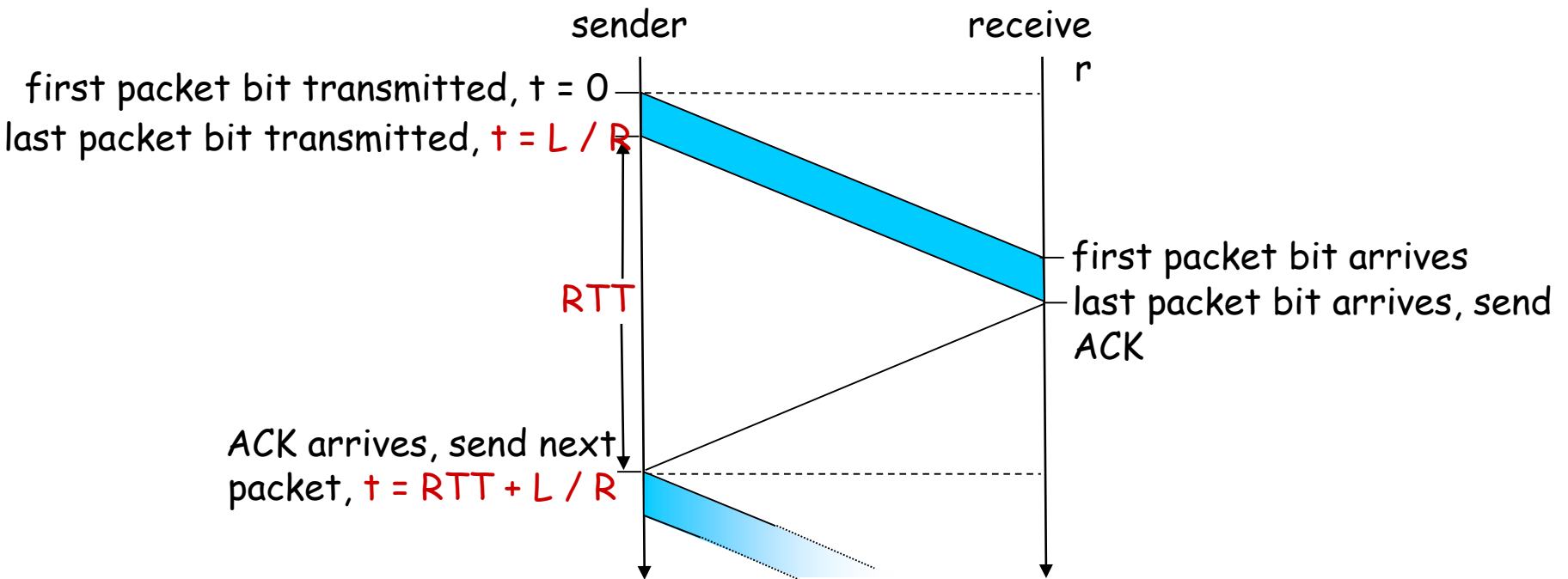
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- $U_{\text{sender sending}}$ : **utilization** - fraction of time sender busy

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

- if RTT=30 msec, 1KB pkt every 30 msec:  
267kbp/sec thrupt over 1 Gbps link
- ❖ network protocol limits use of physical resources!

# rdt3.0: stop-and-wait operation

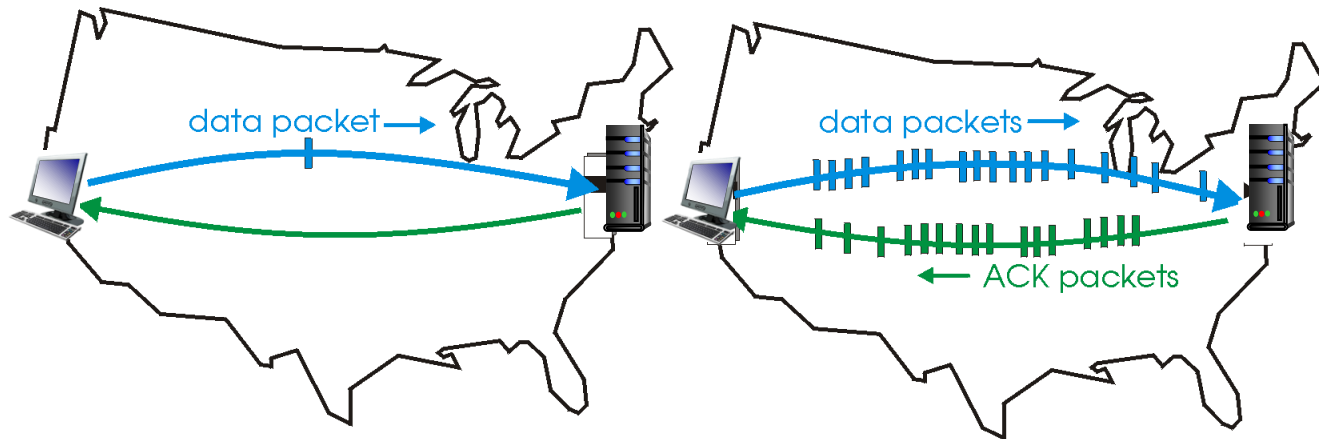


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

# Pipelined protocols

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

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

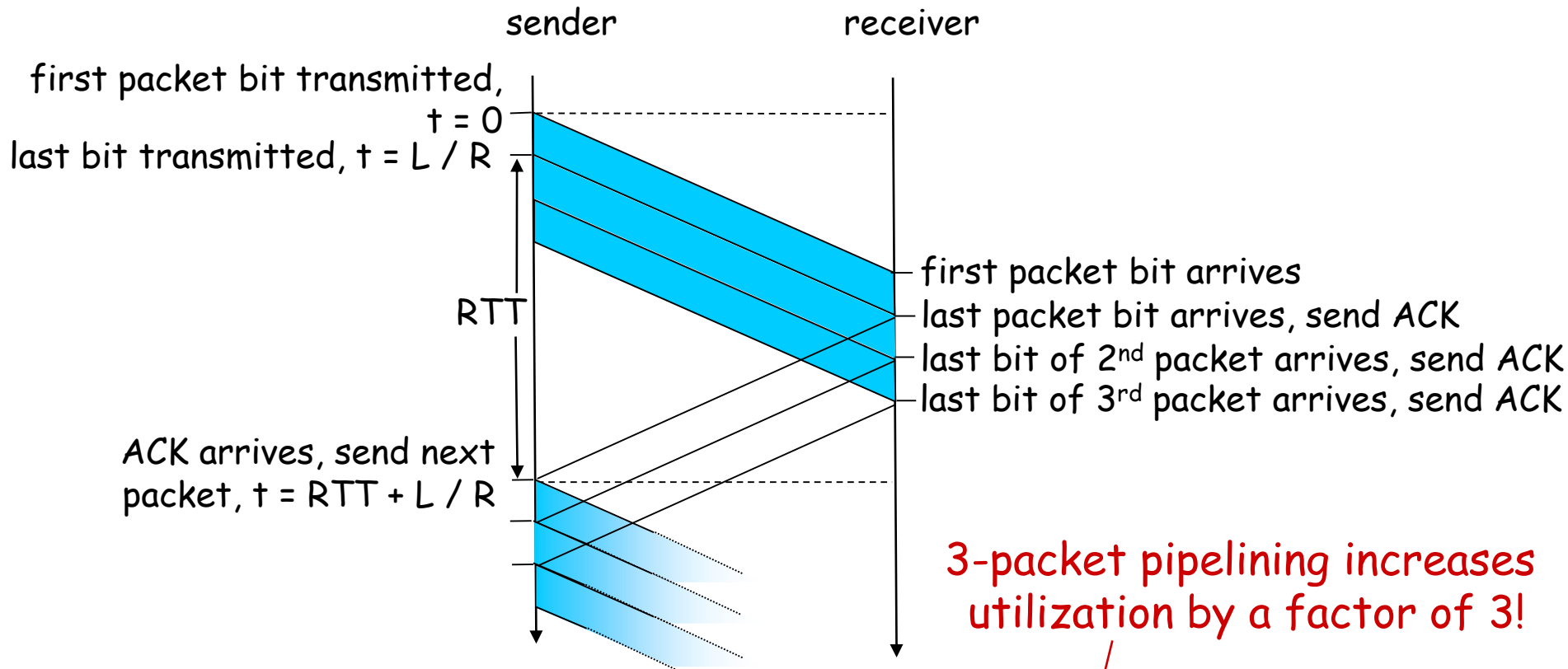


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

# Pipelining: increased utilization



$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$



# Pipelined protocols: overview

## Go-back-N:

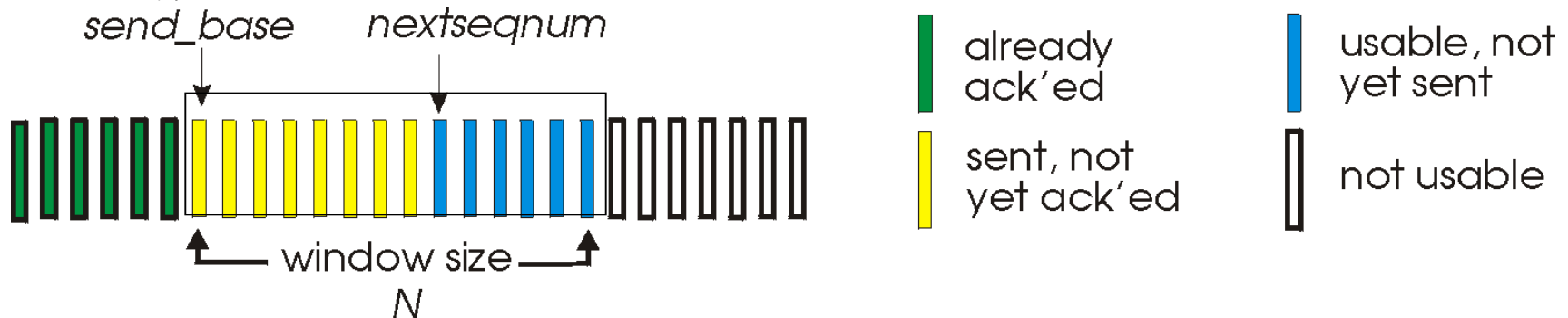
- ❖ sender can have up to N unacked packets in pipeline
- ❖ receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- ❖ sender has timer for *oldest unacked packet*
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

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

# 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 in action

sender window (N=4)

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

sender

send pkt0  
send pkt1  
send pkt2  
send pkt3  
(wait)

rcv ack0, send pkt4  
rcv ack1, send pkt5

ignore duplicate ACK



*pkt 2 timeout*

send pkt2  
send pkt3  
send pkt4  
send pkt5

receiver

receive pkt0, send ack0  
receive pkt1, send ack1

receive pkt3, discard,  
(re)send ack1

receive pkt4, discard,  
(re)send ack1

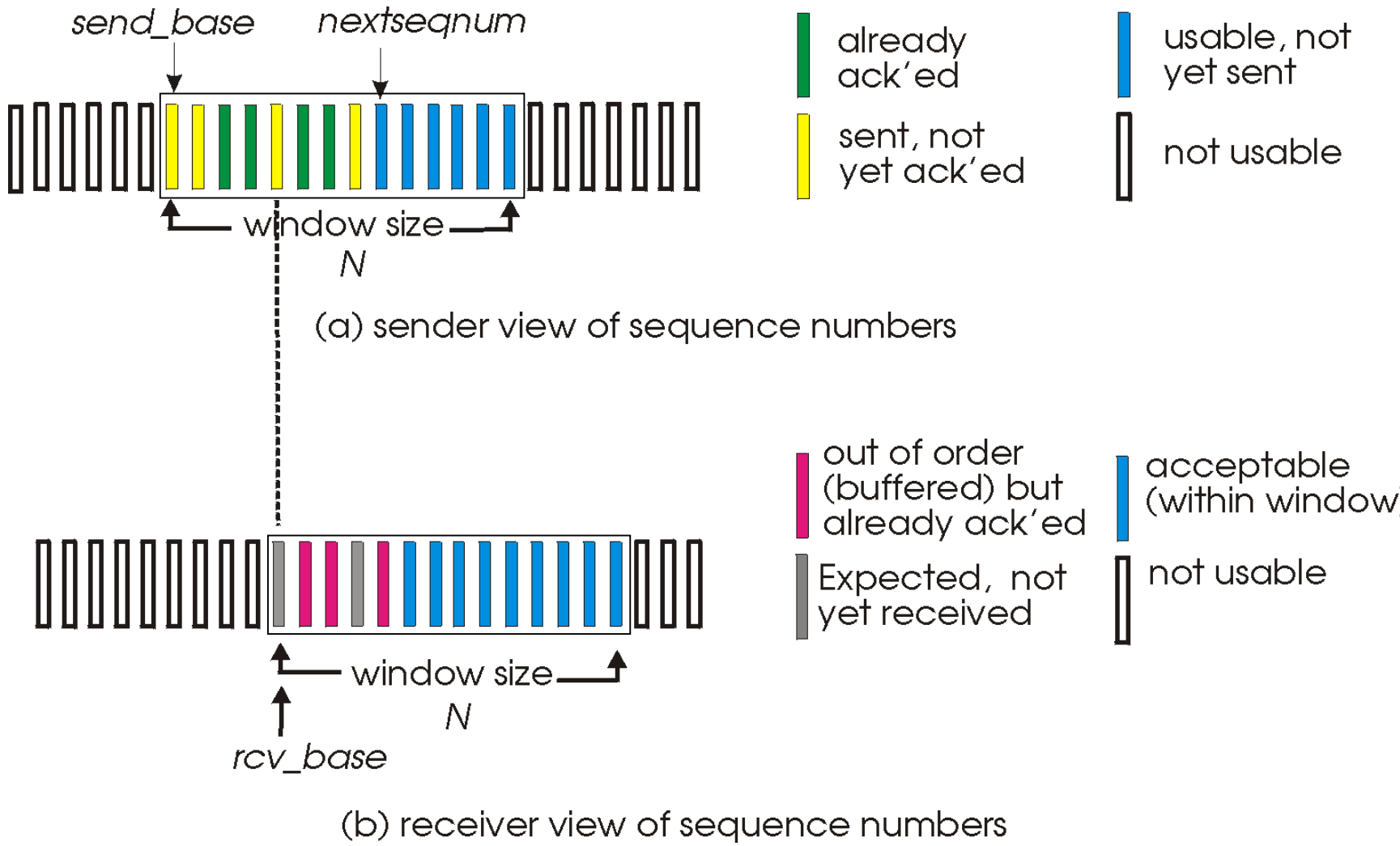
receive pkt5, discard,  
(re)send ack1

rcv pkt2, deliver, send ack2  
rcv pkt3, deliver, send ack3  
rcv pkt4, deliver, send ack4  
rcv pkt5, deliver, send ack5

# SR: 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



# 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

sender window (N=4)

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
[ ]

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8  
0 1 2 3 4 5 6 7 8

sender

send pkt0  
send pkt1  
send pkt2  
send pkt3  
(wait)

rcv ack0, send pkt4  
rcv ack1, send pkt5

record ack3 arrived



*pkt 2 timeout*

send pkt2  
record ack4 arrived  
record ack5 arrived

receiver

receive pkt0, send ack0  
receive pkt1, send ack1

receive pkt3, buffer,  
send ack3

receive pkt4, buffer,  
send ack4

receive pkt5, buffer,  
send ack5

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

*X loss*

*Q: what happens when ack2 arrives?*

# Selective repeat: dilemma

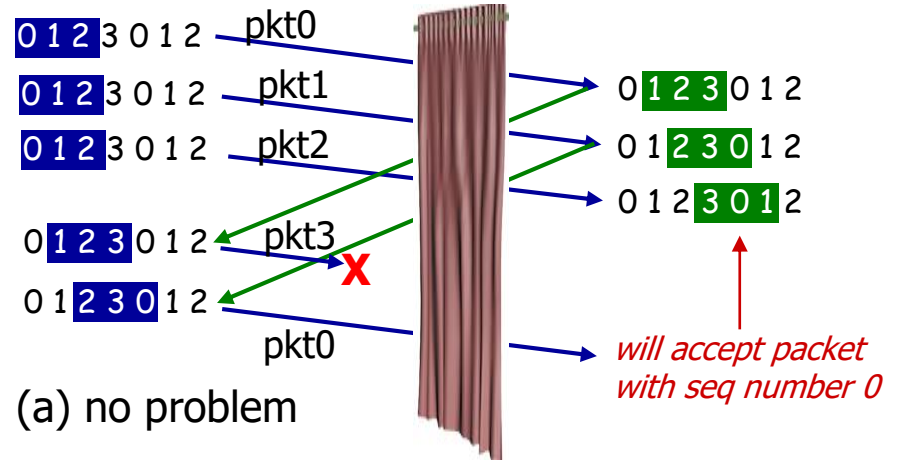
example:

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

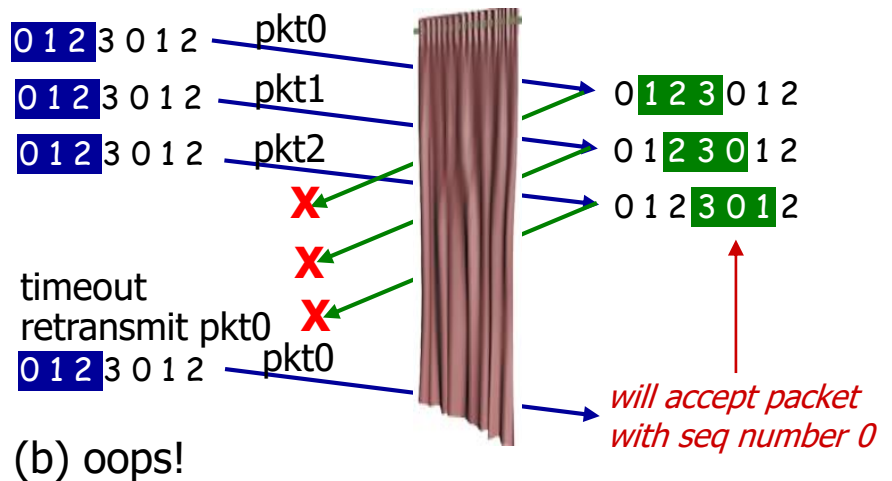
Q: what relationship between seq # size and window size to avoid problem in (b)?

sender window  
(after receipt)

receiver window  
(after receipt)



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





# Summary of reliable data transfer mechanisms and their use

Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.

# 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
- connection management
- reliable data transfer
- flow control

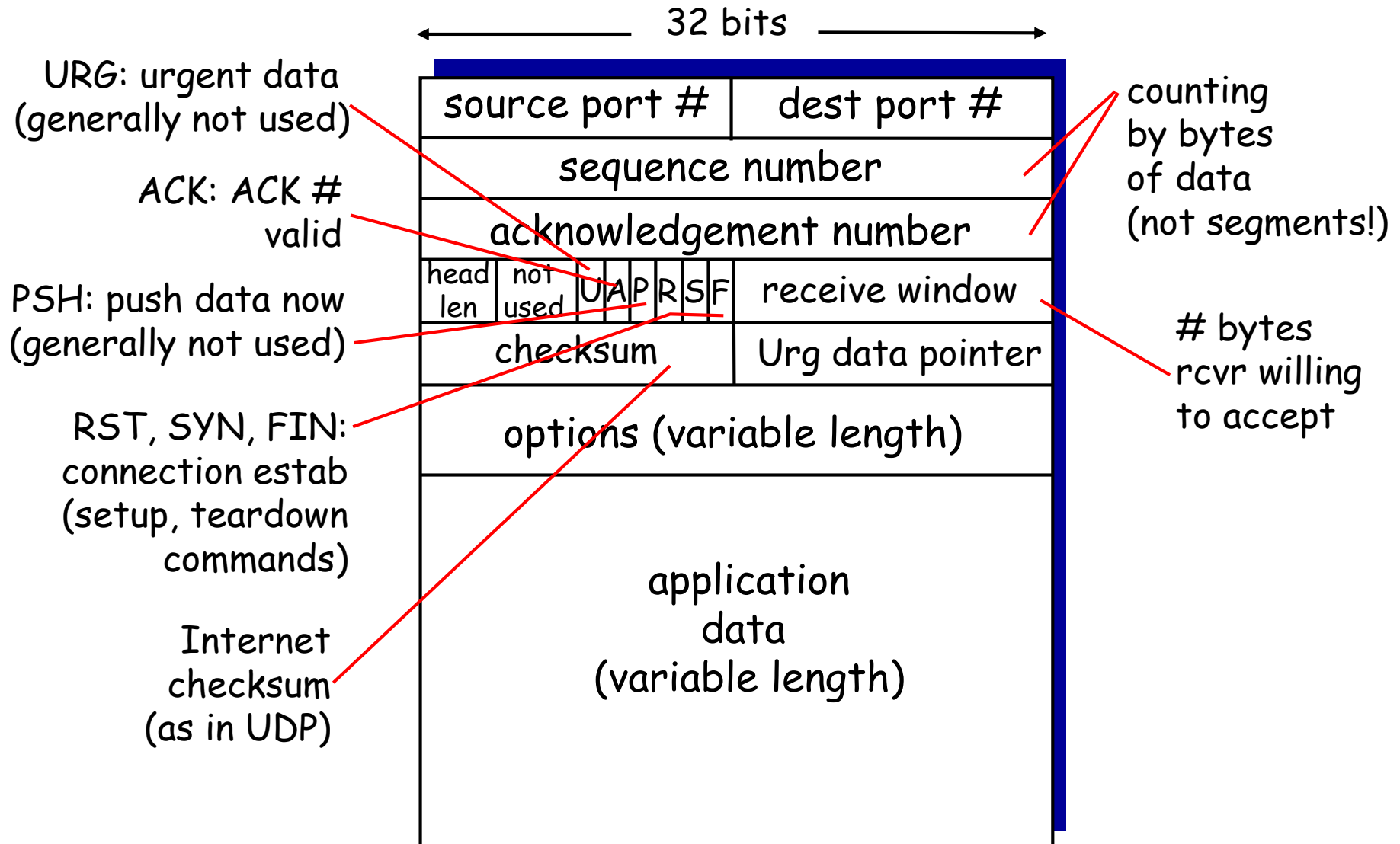
3.6 principles of congestion control

3.7 TCP congestion control

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

- ❖ **point-to-point:**
  - one sender, one receiver
- ❖ **reliable, in-order byte stream:**
  - 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) initializes sender, receiver state before data exchange
- ❖ **flow controlled:**
  - sender will not overwhelm receiver

# TCP segment structure



# TCP seq. numbers, ACKs

## sequence numbers:

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

## acknowledgements:

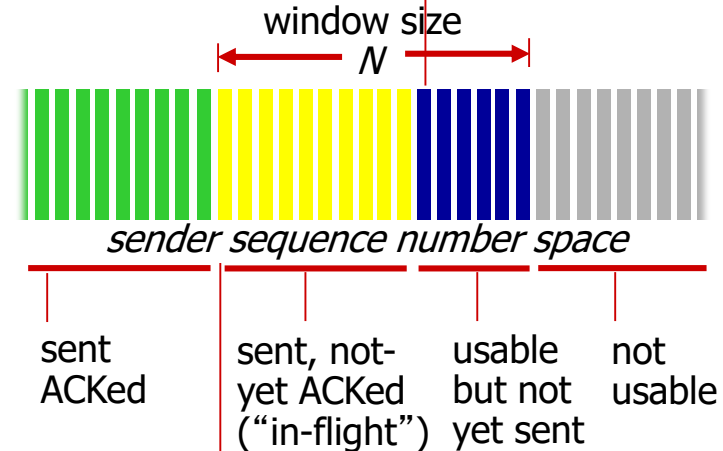
- seq # of next byte expected from other side
- cumulative ACK

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

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

outgoing segment from sender

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



incoming segment to sender

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

❖ *The value in the sequence number field of a segment defines the number assigned to the first data byte contained in that segment.*

*The value of the acknowledgment field in a segment defines the number of the next byte a party expects to receive.  
The acknowledgment number is cumulative.*

**Example1: 0 to 535 → First segment**

**Rx: Sends an Ack 536.**

**Example2: : 0 to 535 → First segment**

**& Rxer Rxes a segment 900 to 1000**

**Meaning? ( 536 to 899 ?)**

# Seq no continued..

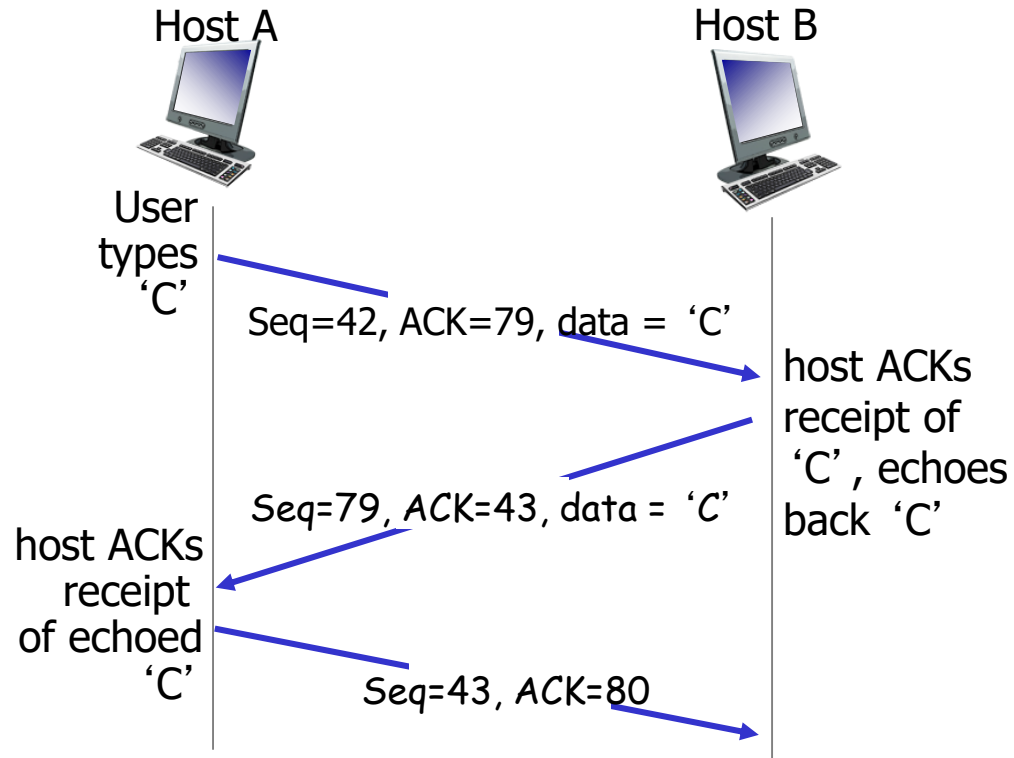
- ❖ Seq no for a segment : the byte stream number i.e first byte in the segment.

Ex: Suppose a TCP connection is transferring a file of 5,000 bytes. The first byte is numbered 10,001. What are the sequence numbers for each segment if data are sent in five segments, each carrying 1,000 bytes?

Soln:

Segment 1	→	Sequence Number:	10,001	<b>Range:</b>	10,001	to	11,000
Segment 2	→	Sequence Number:	11,001	<b>Range:</b>	11,001	to	12,000
Segment 3	→	Sequence Number:	12,001	<b>Range:</b>	12,001	to	13,000
Segment 4	→	Sequence Number:	13,001	<b>Range:</b>	13,001	to	14,000
Segment 5	→	Sequence Number:	14,001	<b>Range:</b>	14,001	to	15,000

# 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

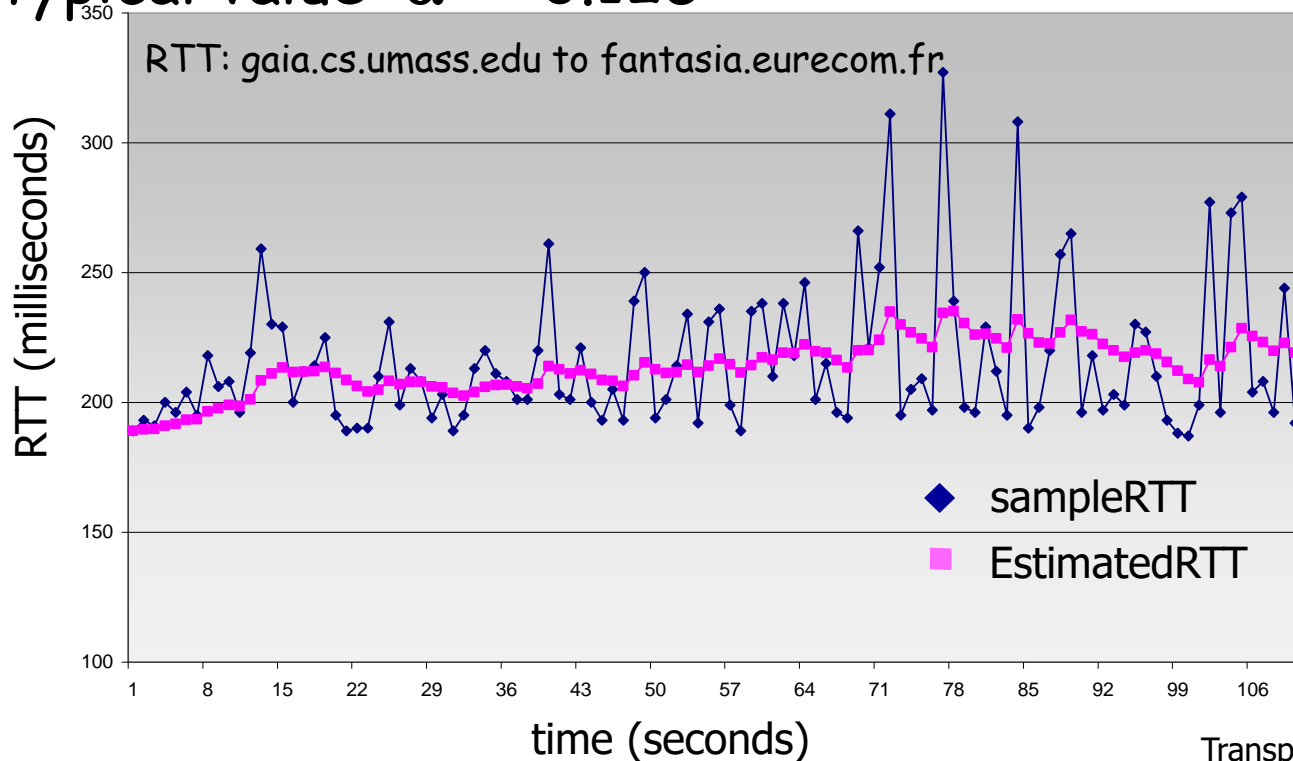
Q: how to estimate RTT?

- ❖ **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

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

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



# TCP round trip time, timeout

- ❖ **timeout interval:** EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin
- ❖ **estimate SampleRTT deviation from EstimatedRTT:**  
$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically,  $\beta = 0.25$ )

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



↑  
estimated RTT

↑  
“safety margin”

# 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

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

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

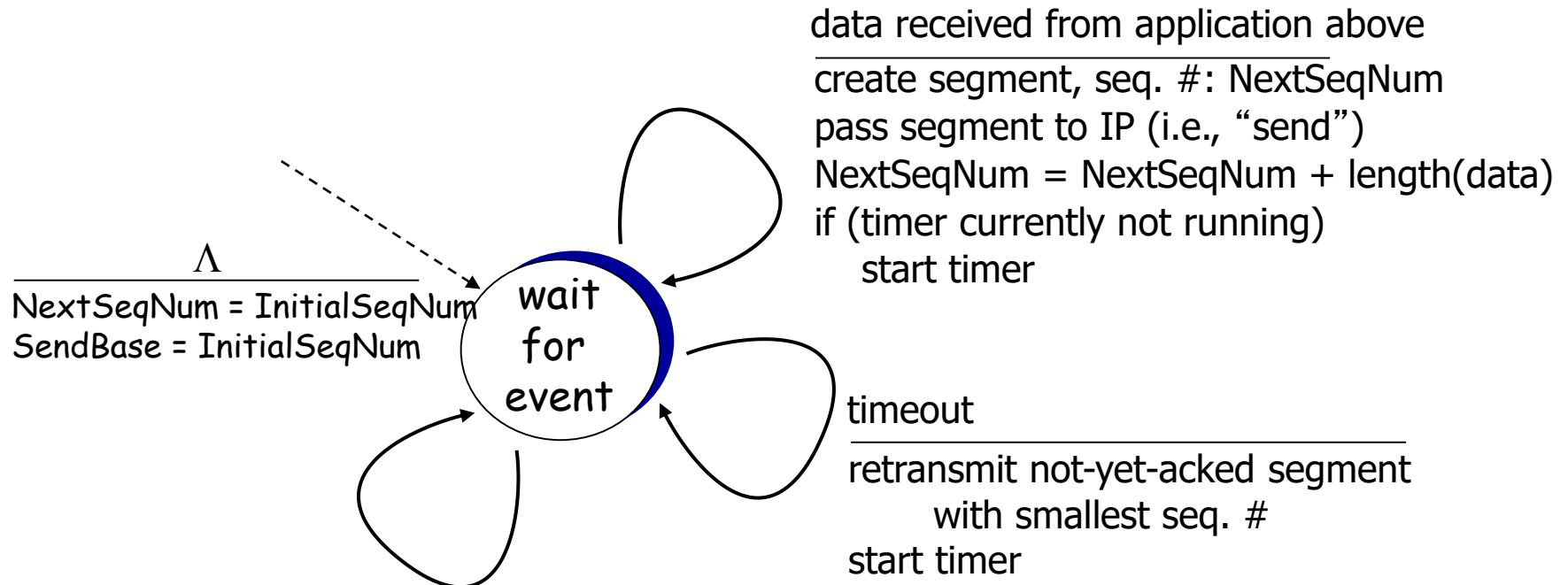
```
    event: data received from application above  
        create TCP segment with sequence number NextSeqNum  
        if (timer currently not running)  
            start timer  
        pass segment to IP  
        NextSeqNum=NextSeqNum+length(data)  
        break;
```

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

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

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

# TCP sender (simplified)

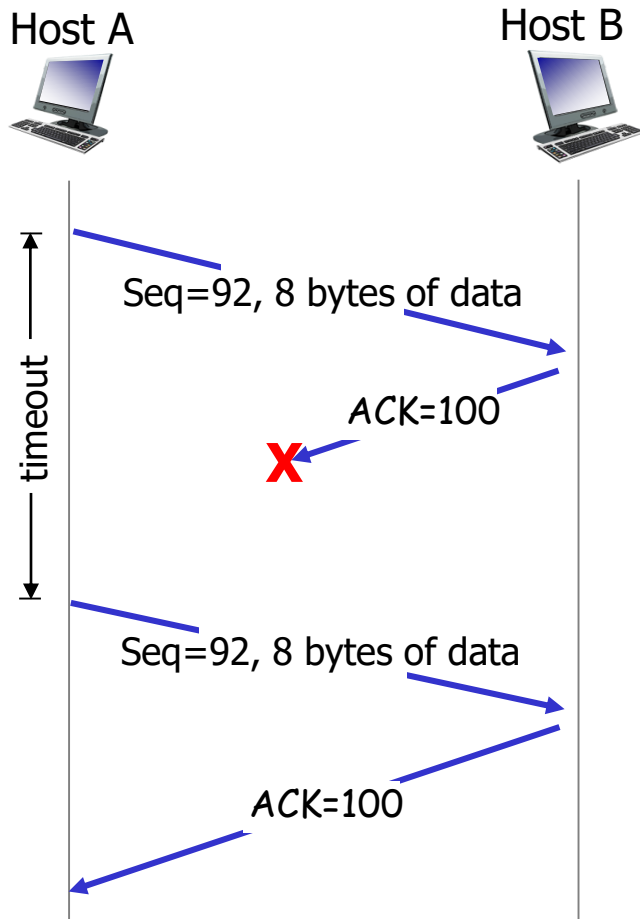


ACK received, with ACK field value y

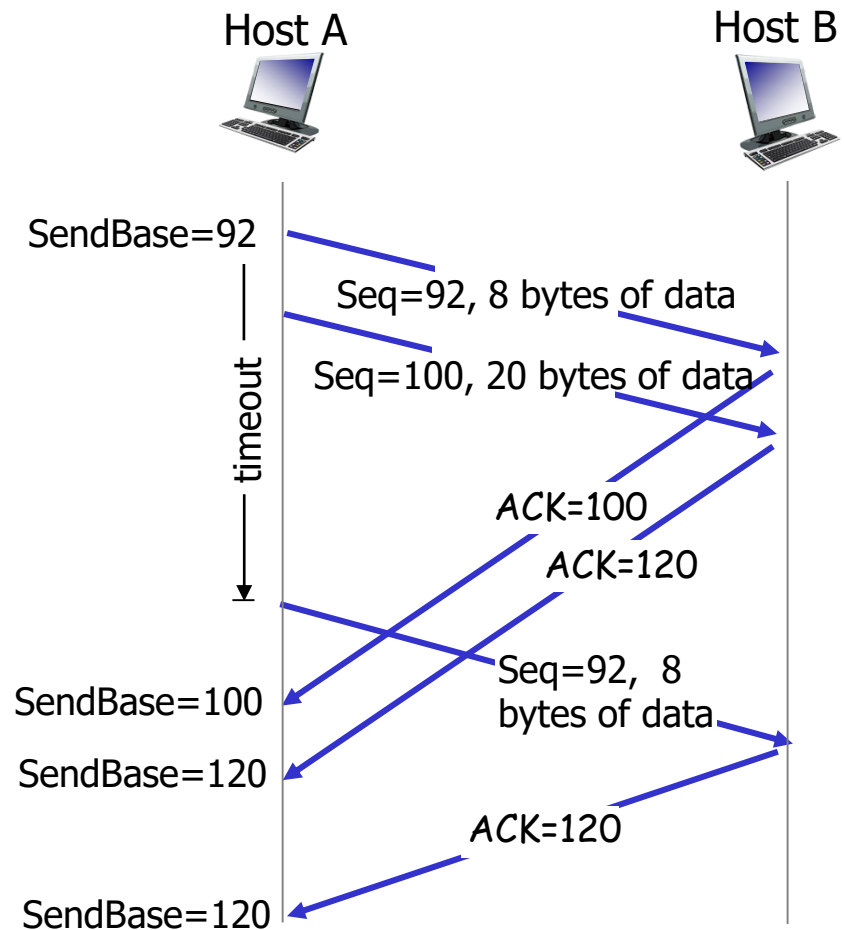
```
if (y > SendBase) {  
    SendBase = y  
    /* SendBase-1: last cumulatively ACKed byte */  
    if (there are currently not-yet-acked segments)  
        start timer  
    else stop timer  
}
```



# TCP: retransmission scenarios

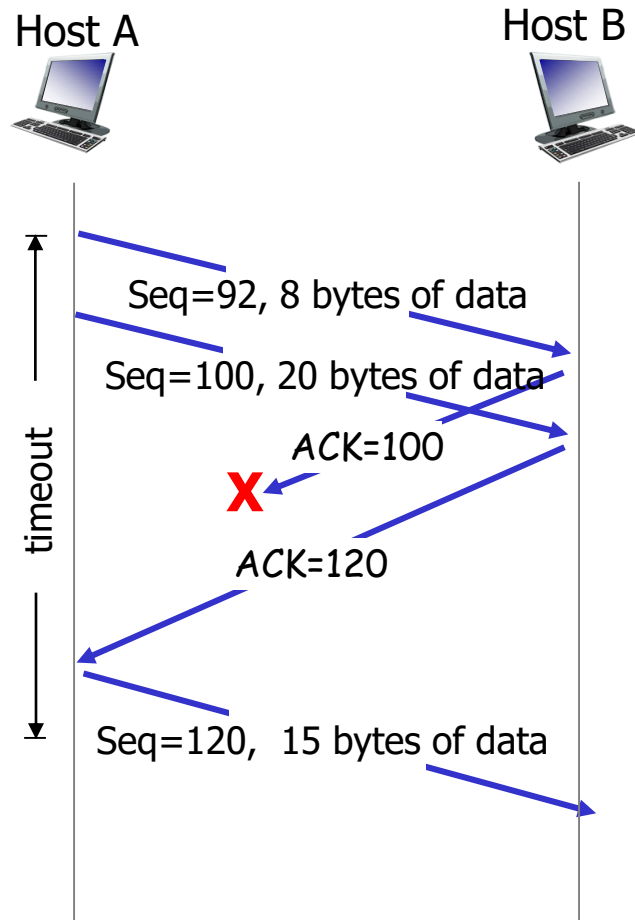


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



cumulative ACK

# TCP ACK generation [RFC 1122, RFC 2581]

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

# TCP fast retransmit

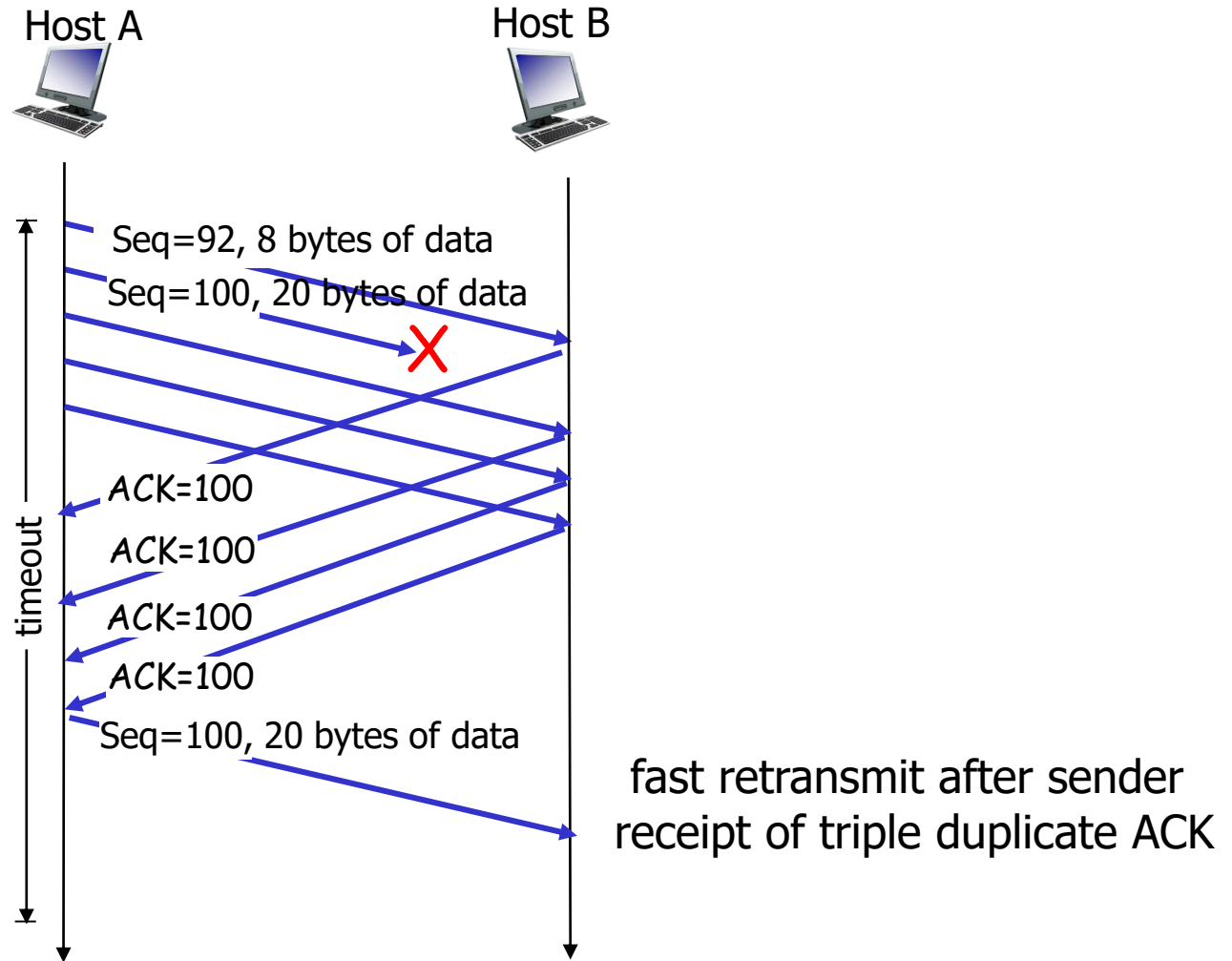
- ❖ time-out period often relatively long:
  - long delay before resending lost packet
- ❖ detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*TCP fast retransmit*

if sender receives 3 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

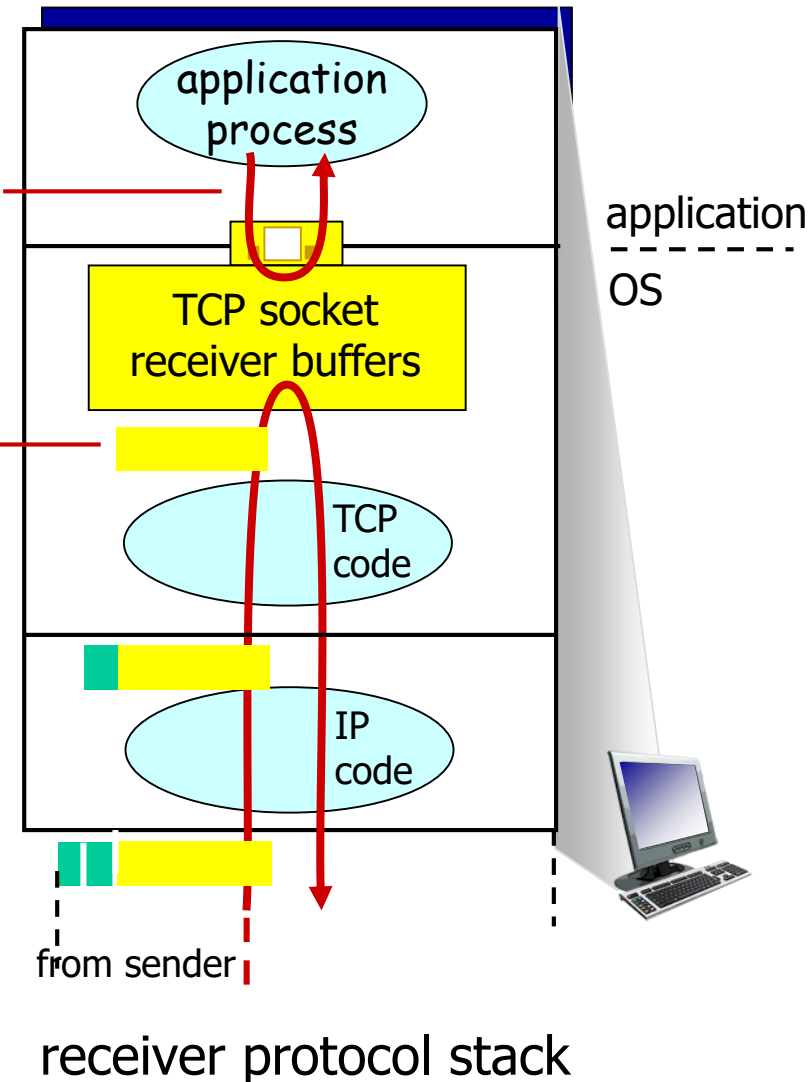
3.7 TCP congestion control

# TCP flow control

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

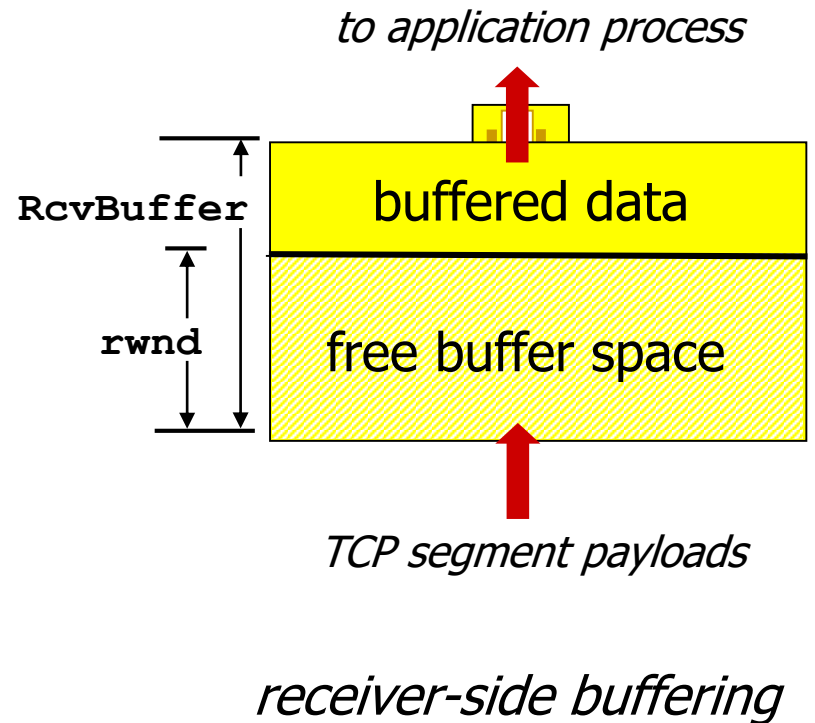
... slower than TCP  
receiver is delivering  
(sender is sending)

*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 auto adjust `RcvBuffer`
- ❖ sender limits amount of unacked (“in-flight”) data to receiver’s `rwnd` value
- ❖ guarantees receive buffer will not overflow





# Flow Control...

- ❖ **LastByteRead**: the number of the last byte in the data stream read from the buffer by the application process in B
- ❖ **LastByteRcvd**: the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.
- ❖ Because TCP is not permitted to overflow the allocated buffer, we must have
$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$
- ❖ The receive window, denoted **rwnd** is set to the amount of spare room in the buffer:
- ❖  $\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$ .
- ❖ **Sender** :  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$

# 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

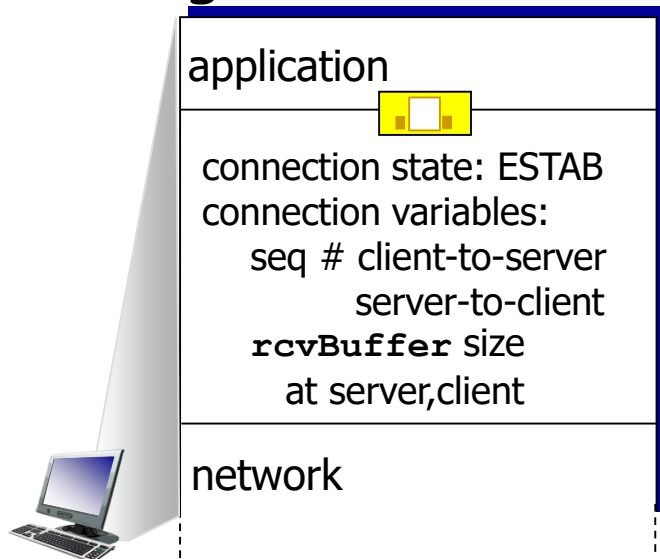
3.6 principles of congestion control

3.7 TCP congestion control

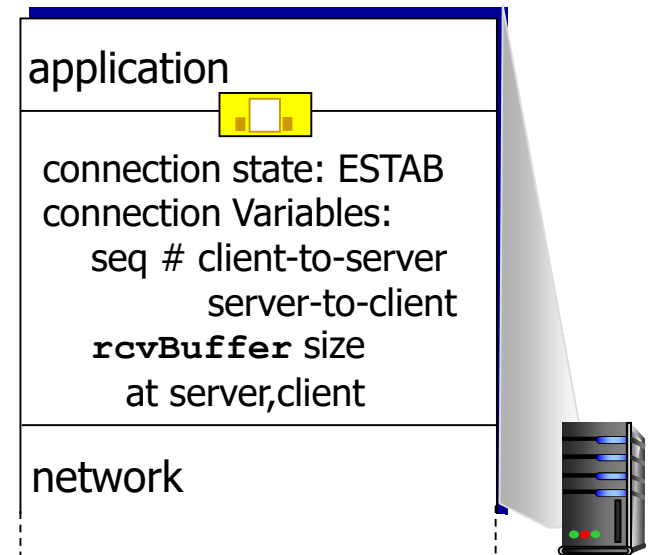
# Connection Management

before exchanging data, sender/receiver  
“handshake”:

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



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



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

# TCP 3-way handshake

*client state*

LISTEN

SYNSENT

ESTAB

choose init seq num, x  
send TCP SYN msg



SYNbit=1, Seq=x



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

*server state*

LISTEN

SYN RCVD

ESTAB

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

received SYNACK(x)  
indicates server is live;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data

ACKbit=1, ACKnum=y+1

received ACK(y)  
indicates client is live

# TCP: closing a connection

- ❖ client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- ❖ respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- ❖ simultaneous FIN exchanges can be handled

# TCP: closing a connection

*client state*

ESTAB

`clientSocket.close()`

FIN\_WAIT\_1

can no longer  
send but can  
receive data

FIN\_WAIT\_2

wait for server  
close

TIMED\_WAIT

timed wait  
for  $2 * \text{max}$   
segment lifetime

CLOSED



FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FINbit=1, seq=y

ACKbit=1; ACKnum=y+1

can still  
send data

can no longer  
send data

*server state*

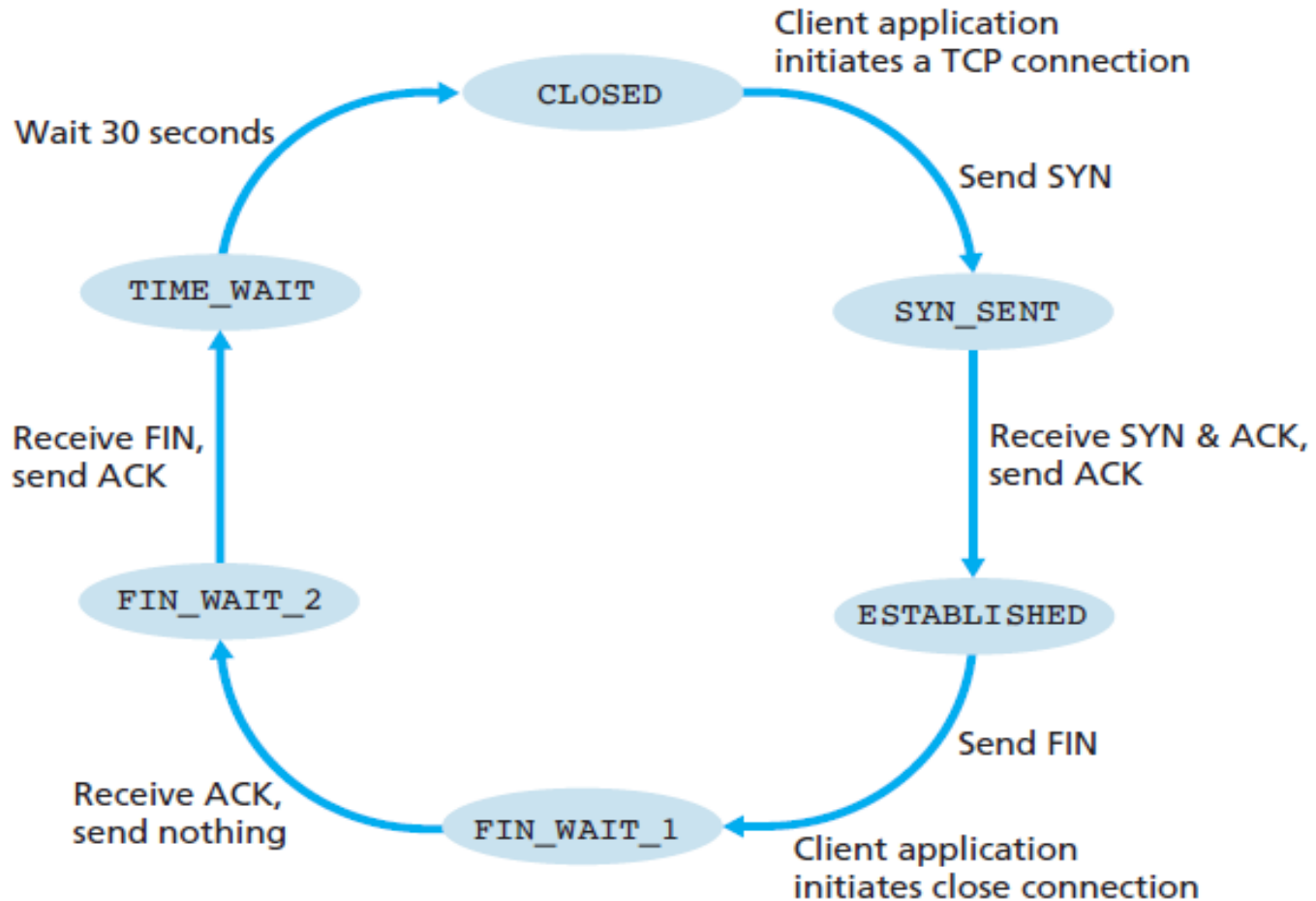
ESTAB

CLOSE\_WAIT

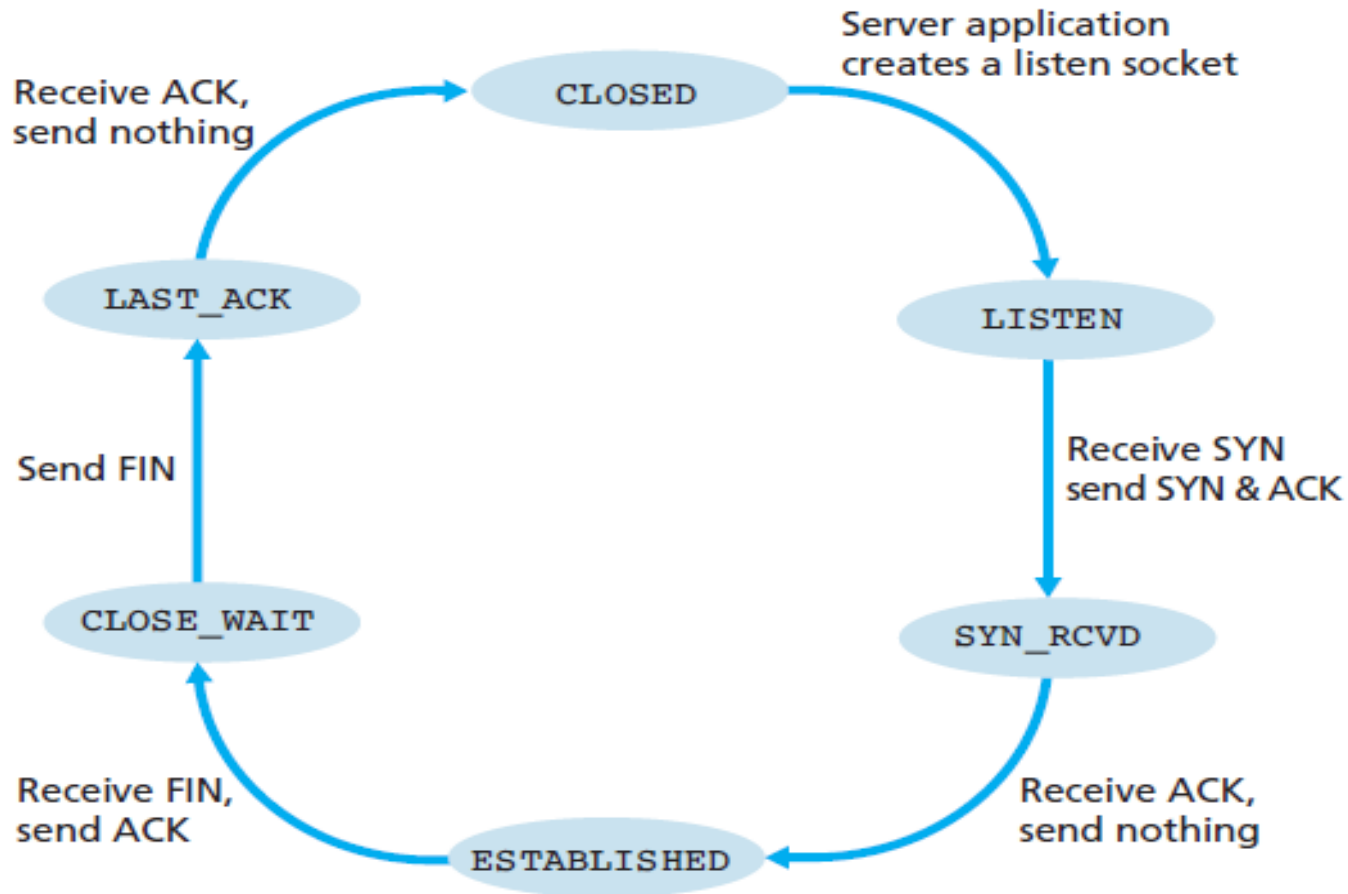
LAST\_ACK

CLOSED

# A typical sequence of TCP states visited by a client TCP



# A typical sequence of TCP states visited by a server-side TCP





# 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!

# 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, ATM)
  - explicit rate for sender to send at
  - Choke packets

# 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

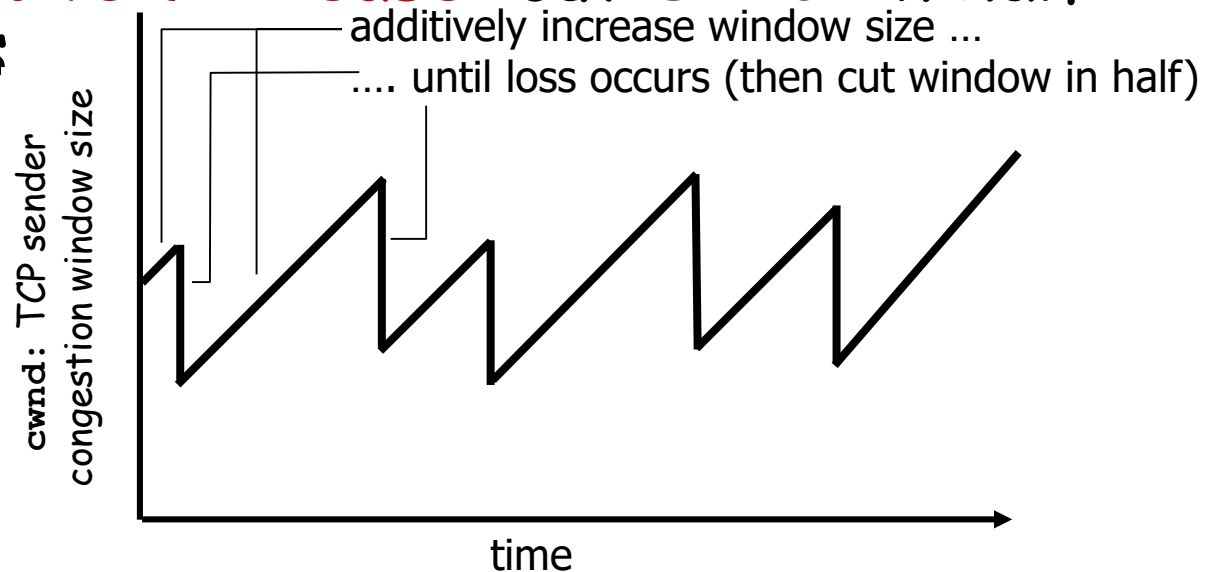
3.6 principles of congestion control

3.7 TCP congestion control

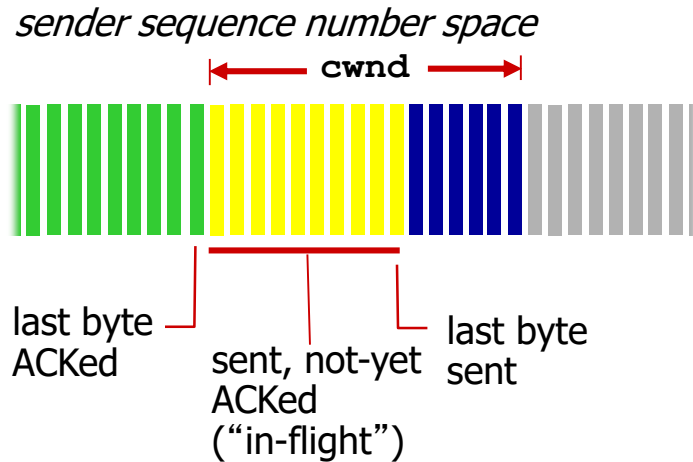
# TCP congestion control: additive increase multiplicative decrease

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

AIMD saw tooth behavior: probing for bandwidth



# TCP Congestion Control: details



*TCP sending rate:*

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

- ❖ sender limits transmission:

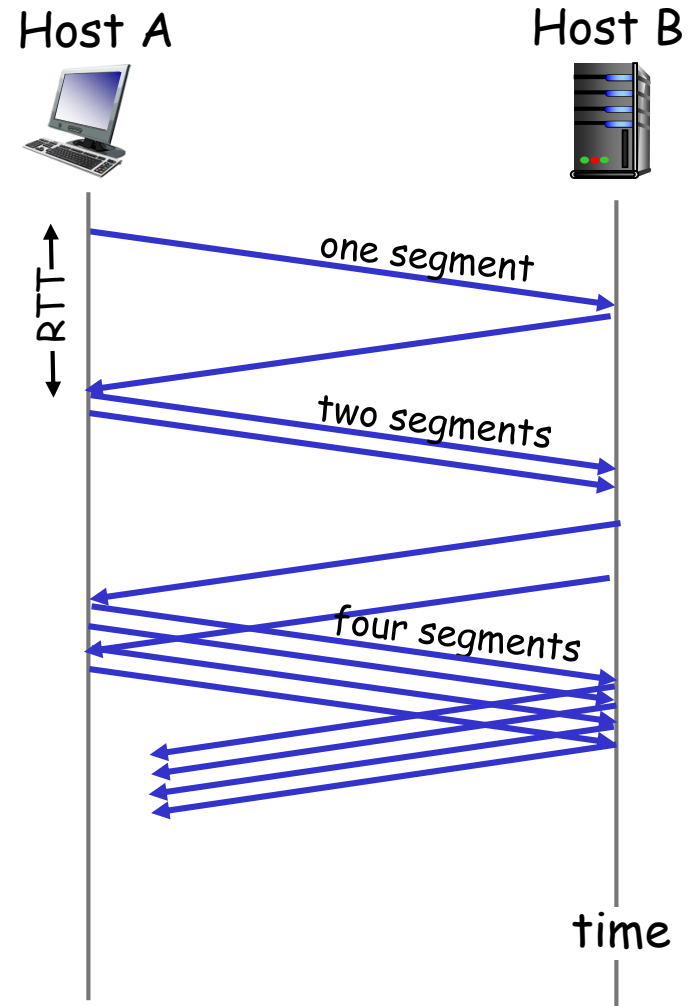
$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

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

- ❖ cwnd is dynamic, function of perceived network congestion

# TCP Slow Start

- ❖ when connection begins, increase rate exponentially until first loss event:
  - initially  $cwnd = 1 \text{ MSS}$
  - 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 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- ❖ loss indicated by 3 duplicate ACKs: **TCP RENO**
  - dup ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly
- ❖ **TCP Tahoe** always sets cwnd to 1 (timeout or 3 duplicate acks)



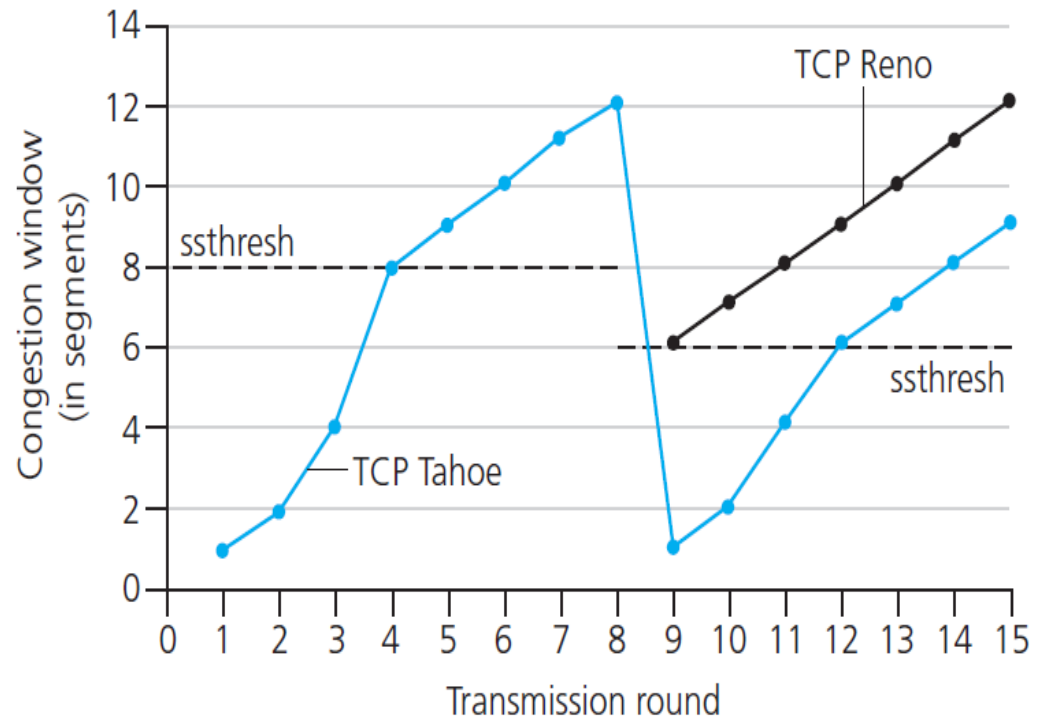
# TCP: switching from slow start to CA

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

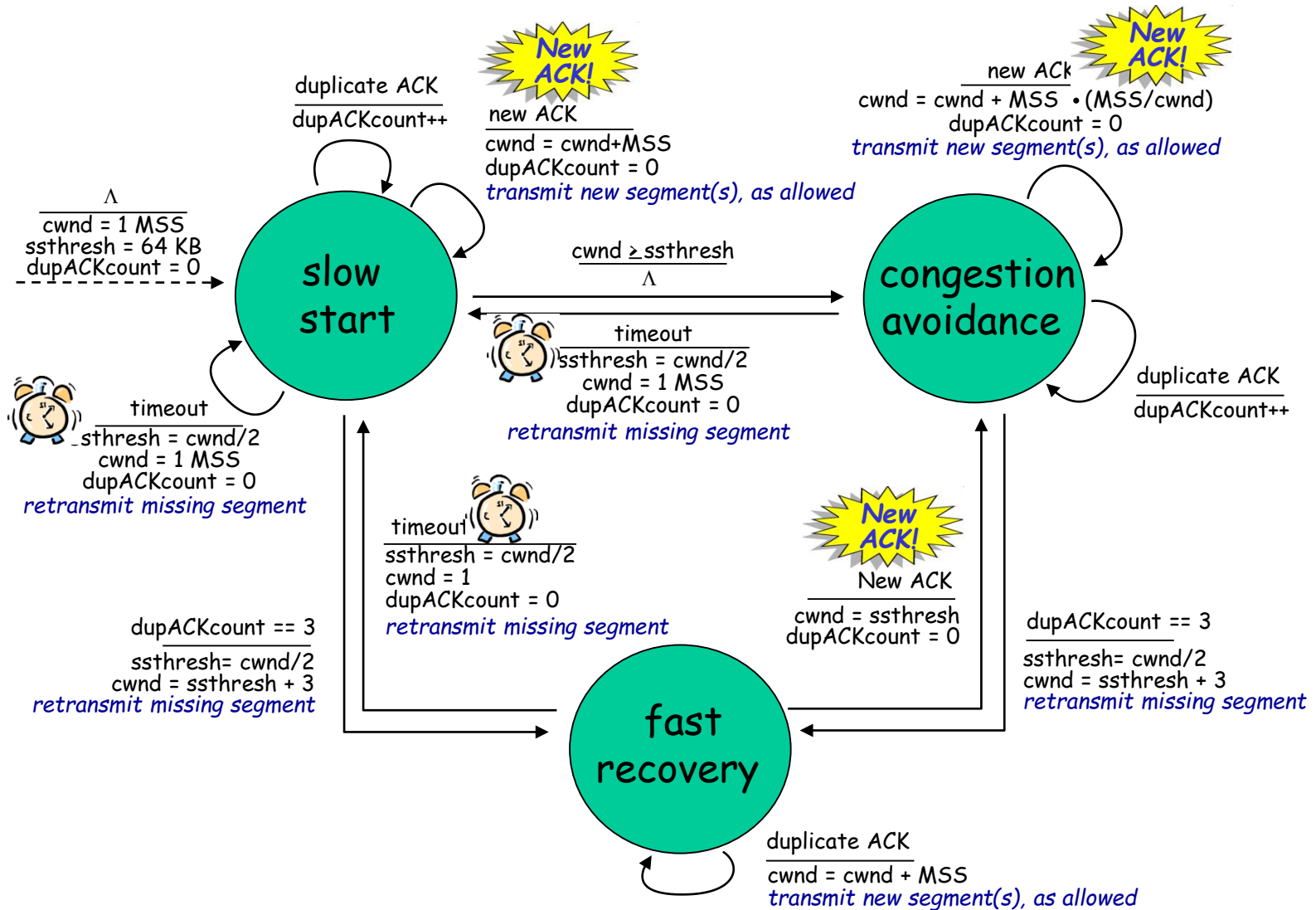
**A:** when cwnd gets to 1/2 of its value before timeout.

## Implementation:

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



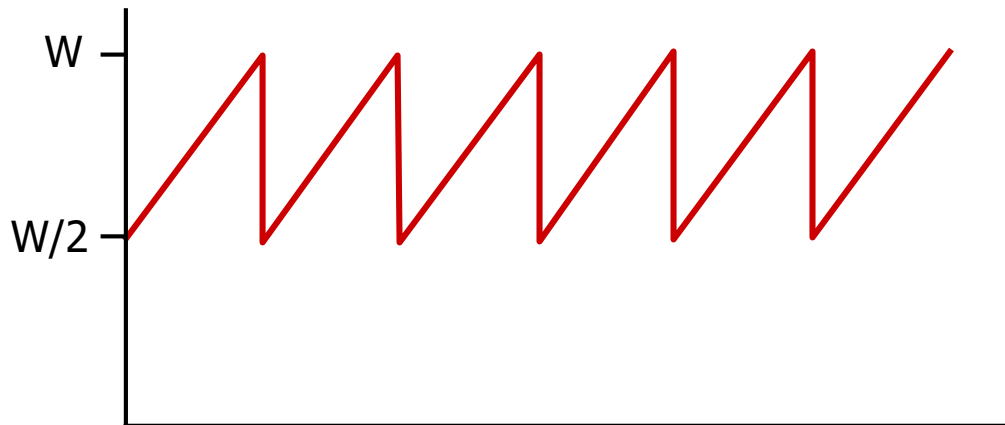
# Summary: TCP Congestion Control



# TCP throughput

- ❖ avg. TCP thrupt 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  $\frac{3}{4} W$
  - avg. thrupt is  $3/4W$  per RTT

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



# 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”

# Exercise

R8. Suppose that a Web server runs in Host C on port 80. Suppose this Web server uses persistent connections, and is currently receiving requests from two different Hosts, A and B. Are all of the requests being sent through the same socket at Host C? If they are being passed through different sockets, do both of the sockets have port 80? Discuss and explain.

# Solution

- ❖ For each persistent connection, the Web server creates a separate "connection socket".
- ❖ Each connection socket is identified with a four-tuple: (source IP address, source port number, destination IP address, destination port number).
- ❖ When host *C* receives an IP datagram, it examines these four fields in the datagram/segment to determine to which socket it should pass the payload of the TCP segment. Thus, the requests from *A* and *B* pass through different sockets. The identifier for both of these sockets has 80 for the destination port; however, the identifiers for these sockets have different values for source IP addresses.