CS 305: Computer Networks
Fall 2024

Lecture 7 : Transport Layer

Tianyue Zheng

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

Chapter 3 outline

Summary

Roadmap:

❖ Perfectly reliable channel : rdt1.0

❖ Channel with bit error:
  - bit error in packet: rdt 2.0
  - bit error in ACK: 2.1
  - NAK-free: 2.2

❖ Lossy channel: rdt 3.0

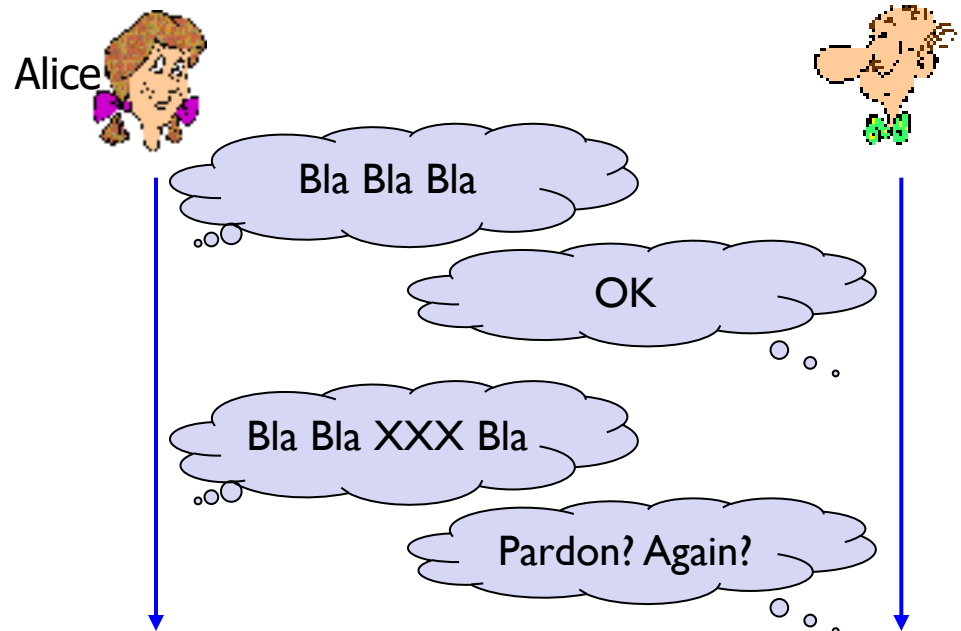**Summary of Techniques**

- Checksum
- Sequence number
- ACK packets
- Retransmission
- Timeout

Alice

Bla Bla Bla

OK
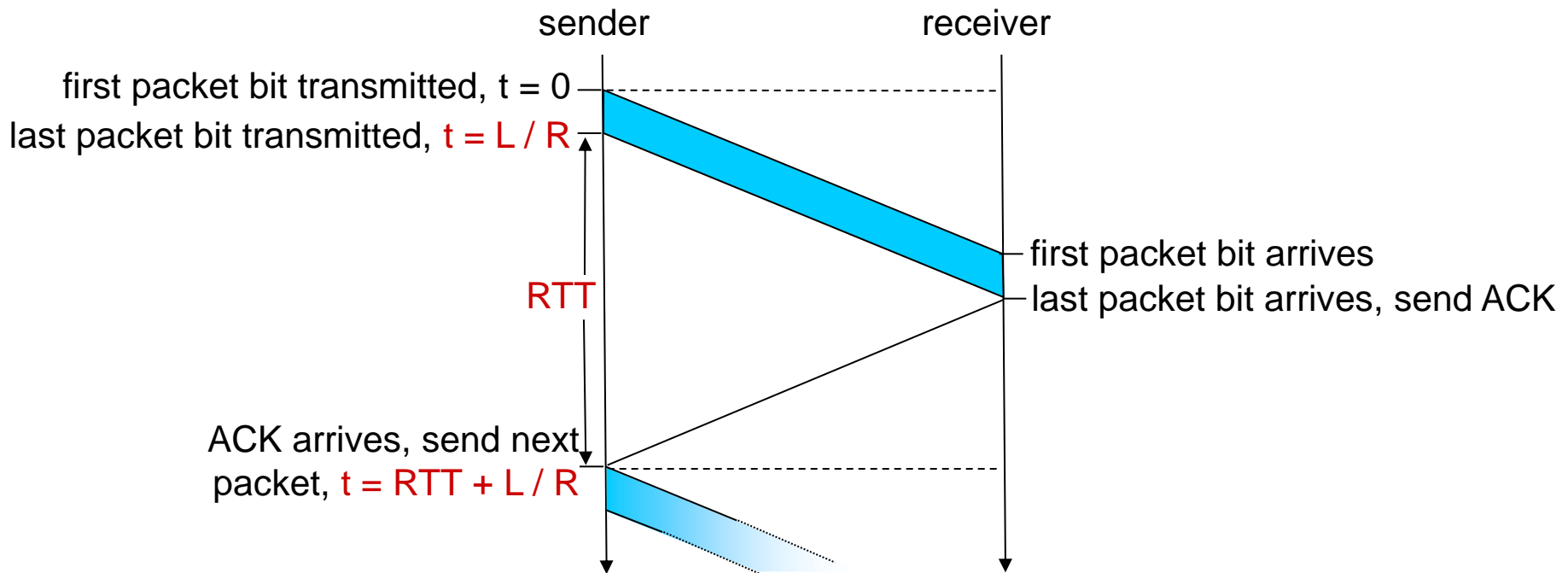
Bla Bla XXX Bla

Pardon? Again?

Stop and wait

Sender sends one packet, then waits for receiver response

Limitations?

Performance of rdt3.0

❖ rdt3.0 is correct, but performance is bad

❖ e.g.: link rate R=1 Gbps, prop. delay T_{pd}=15 ms, packet length L=8000 bit

sender   receiver

first packet bit transmitted, t = 0

last packet bit transmitted, t = L / R

first packet bit arrives

RTT   last packet bit arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

- Calculate utilization U_{sender}: fraction of time sender busy sending

Performance of rdt3.0

❖ link rate R=1 Gbps, prop. delay T_{pd}=15 ms, packet length L=8000 bit

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

▪ utilization U $_{sender}$: fraction of time sender busy sending

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

▪ RTT=30 msec, 1KB pkt every 30 msec:
33kB/sec throughput over 1 Gbps link

❖ network protocol limits use of physical resources!

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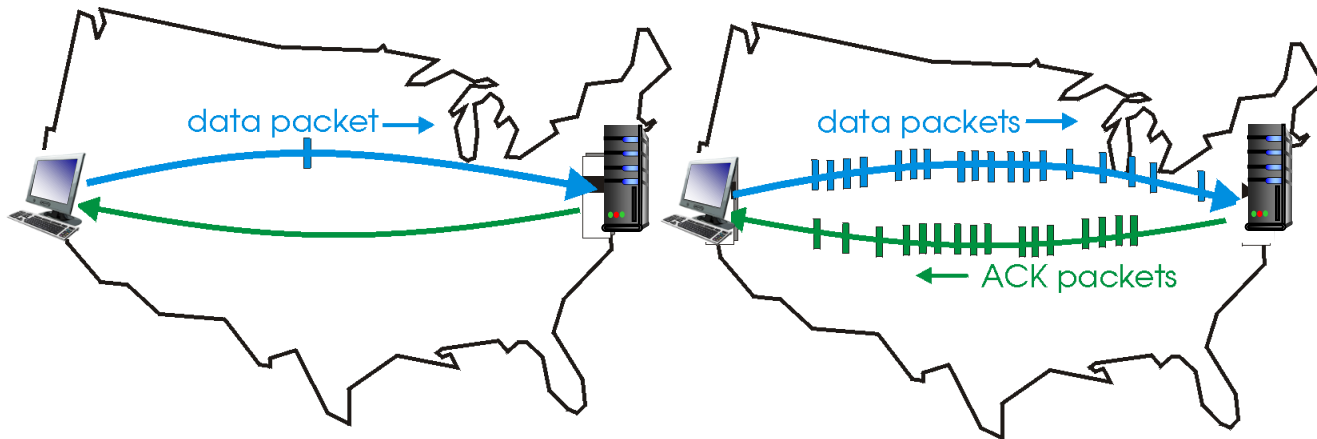
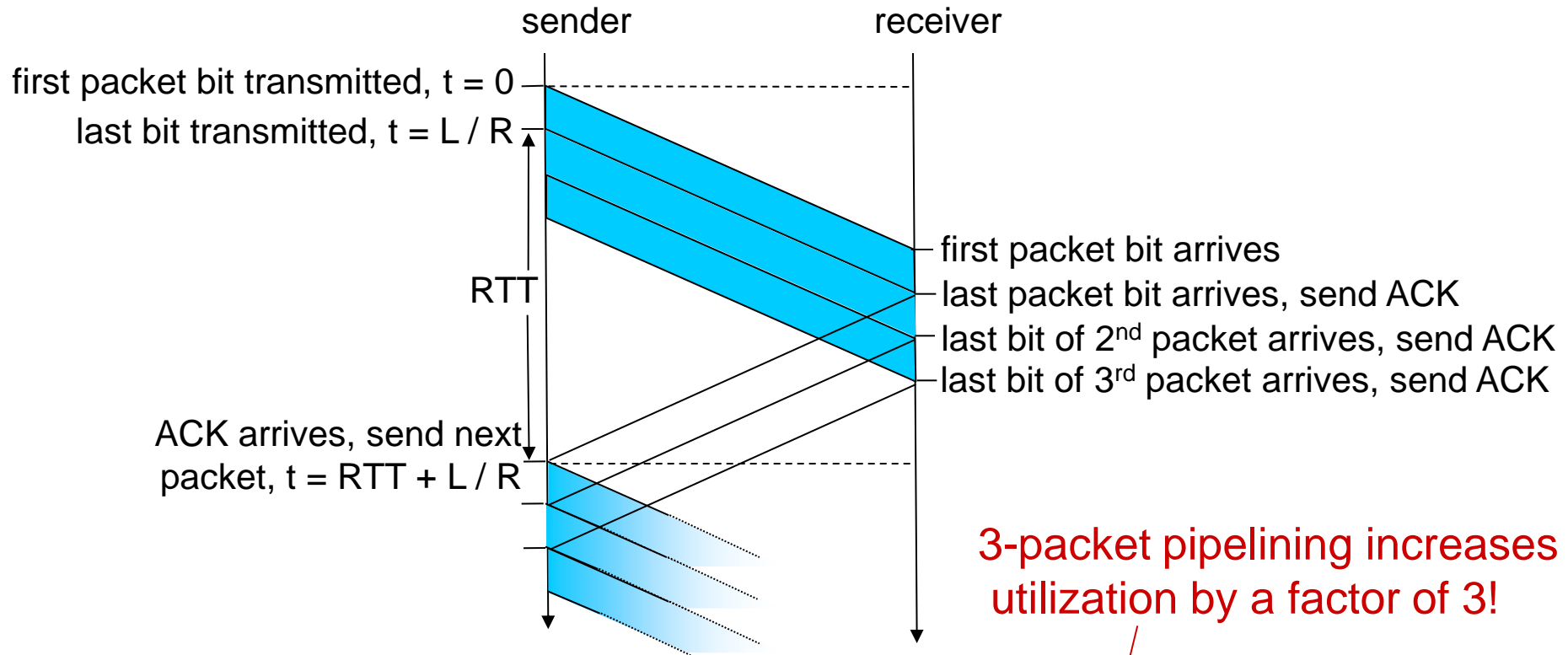(a) a stop-and-wait protocol in operation   (b) a pipelined protocol in operation

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

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next packet, t = RTT + L / R

3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.00081$$

Pipelined Protocols

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

Go-Back-N: Sender

- k-bit seq # in pkt header (not 0 or 1): $[0, 2^k - 1]$
- At most N pkts in flight: window size = N, (N consecutive unacked pkts allowed
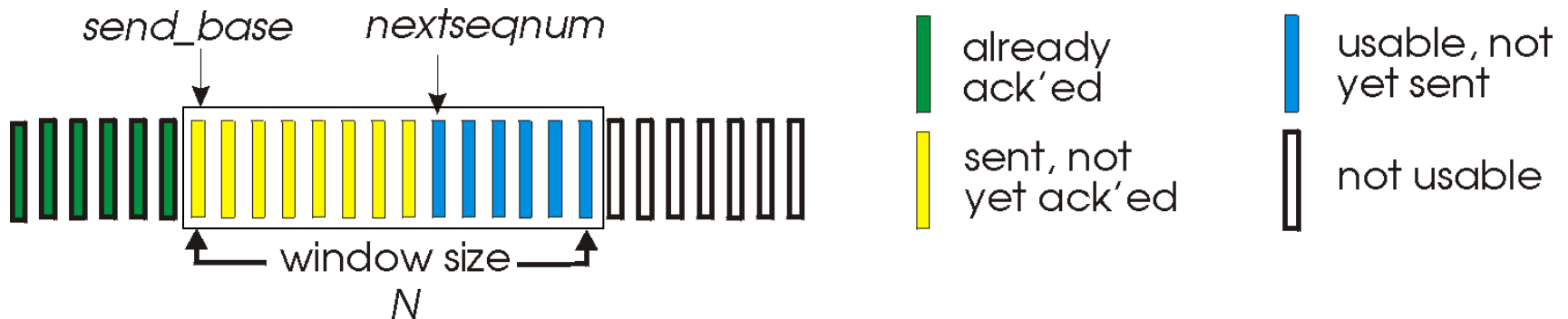
Sender: When rdt_send() is called from above,
- window is not full: a packet is sent, variables are updated.
- window is full: simply returns the data back to the upper layer

A timer for the oldest transmitted but not yet ACKed packet

Go-Back-N: Receiver and Timeout

Receiver: Receipt of an ACK.
- Cumulative acknowledgment (ACK)
- ACK(n): all packets with a sequence # up to and including n have been correctly received at the receiver
- Expect n and receive n: ACK(n)
- Expect n and receive others: previous ACK; discard packet

Sender:
- **timeout** occurs: resends all packets in the window;
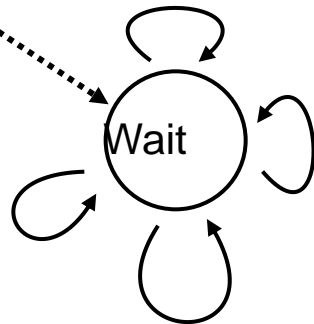- ACK(n): slide window; restart timer
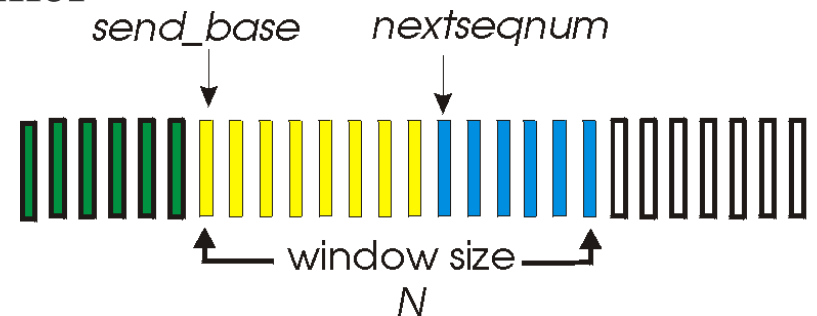
GBN: sender extended FSM

rdt_send(data)
_____

```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
    nextseqnum++
    }
else
    refuse_data(data)
```

Λ
_____
base=0
nextseqnum=0

rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
_____
Λ

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

Wait

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
_____
```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

send_base   *nextseqnum*

window size
N
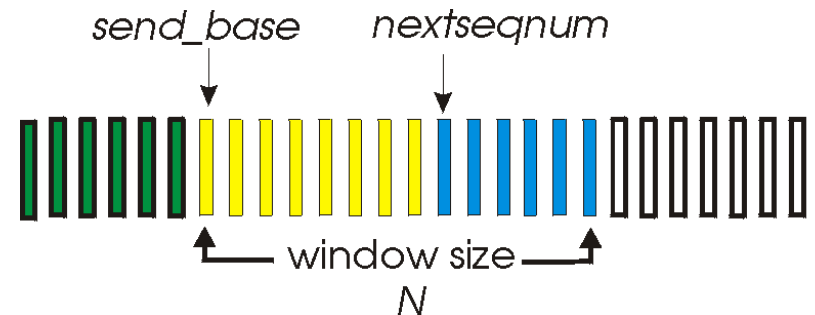
GBN: receiver extended FSM

default
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcurrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)

Λ
expectedseqnum=0
sndpkt =
make_pkt(expectedseqnum,ACK,chksum)

Wait

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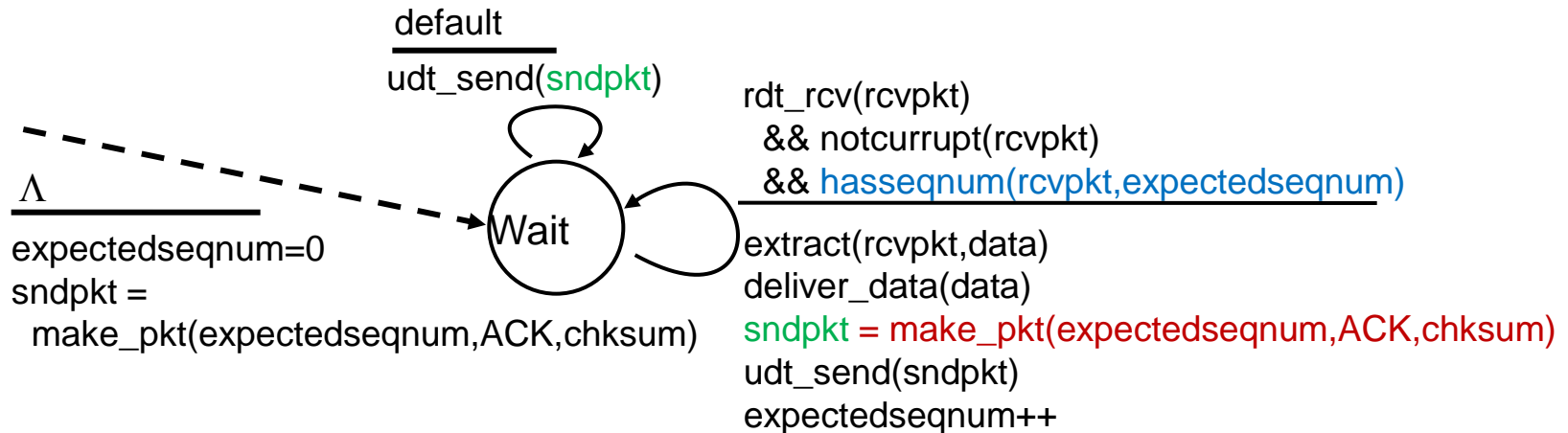
Cumulative ACK: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs

Out-of-order pkt:

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

Go-Back-N Recall

sender window (N=4)

sender		receiver	No buffer, Cumulative ACK

0 1 2 3 4 5 6 7 8   send pkt0

0 1 2 3 4 5 6 7 8   send pkt1

0 1 2 3 4 5 6 7 8   send pkt2   receive pkt0, send ack0

0 1 2 3 4 5 6 7 8   send pkt3   **X** *loss*   receive pkt1, send ack1

(wait)

receive pkt3, discard,
(re)send ack1

0 1 2 3 4 5 6 7 8   rcv ack0, send pkt4

0 1 2 3 4 5 6 7 8   rcv ack1, send pkt5   receive pkt4, discard,
(re)send ack1

ignore duplicate ACK   receive pkt5, discard,
(re)send ack1

🕐 *pkt 2 timeout*

0 1 2 3 4 5 6 7 8   send pkt2

0 1 2 3 4 5 6 7 8   send pkt3

0 1 2 3 4 5 6 7 8   send pkt4   rcv pkt2, deliver, send ack2

0 1 2 3 4 5 6 7 8   send pkt5   rcv pkt3, deliver, send ack3

rcv pkt4, deliver, send ack4

rcv pkt5, deliver, send ack5

Retransmit all pkts upon
pkt loss or error (GBN)
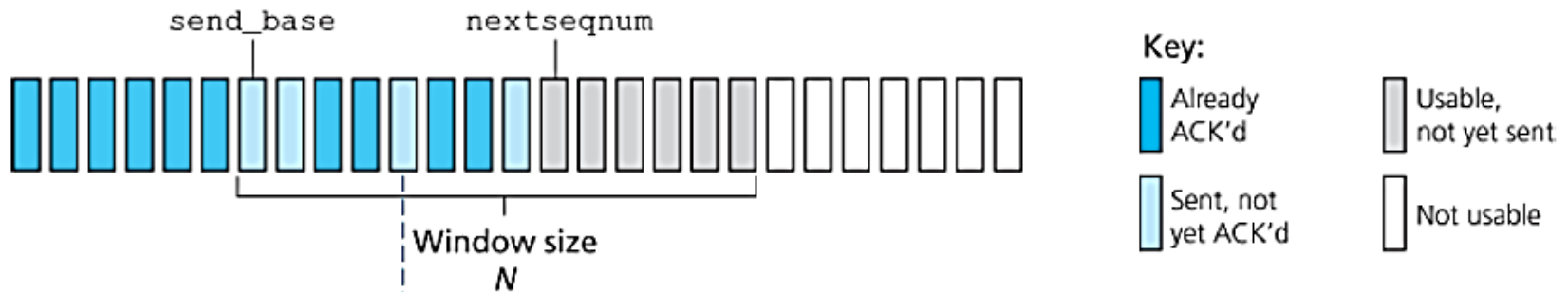
Pipelined Protocols

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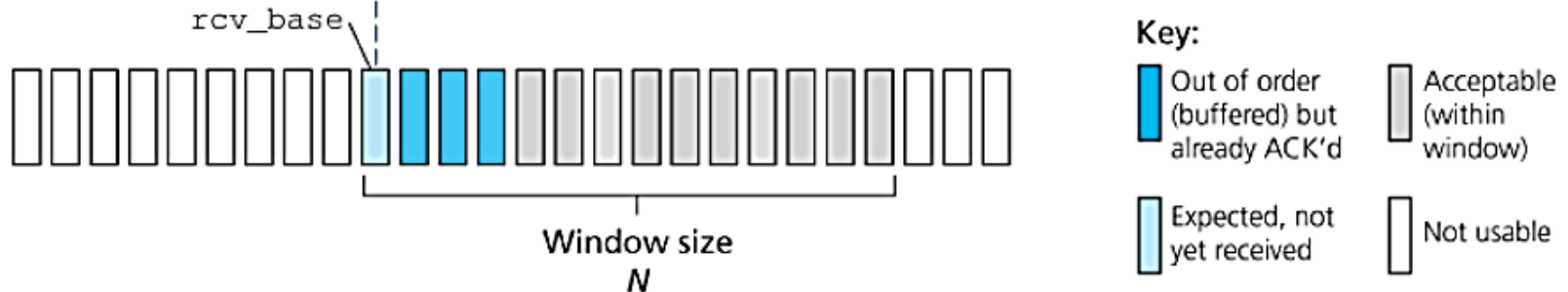   - ■ Retransmit only those packets that might be lost or corrupted

Selective repeat

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

Selective repeat: sender, receiver windows

a. Sender view of sequence numbers

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

b. Receiver view of sequence numbers

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

Selective repeat

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 #

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

sender window (N=4)

sender

receiver   have buffer, individual ACK

0 1 2 3 4 5 6 7 8   send pkt0
0 1 2 3 4 5 6 7 8   send pkt1
0 1 2 3 4 5 6 7 8   send pkt2   receive pkt0, send ack0
0 1 2 3 4 5 6 7 8   send pkt3   **X** loss   receive pkt1, send ack1
                          (wait)

                                                        receive pkt3, buffer,
                                                            send ack3

0 **1 2 3 4** 5 6 7 8   rcv ack0, send pkt4
0 1 **2 3 4 5** 6 7 8   rcv ack1, send pkt5   receive pkt4, buffer,
                                                             send ack4
                          record ack3 arrived   receive pkt5, buffer,
                                                             send ack5

                          pkt 2 timeout

0 1 **2 3 4 5** 6 7 8   send pkt2
0 1 **2 3 4 5** 6 7 8   record ack4 arrived
0 1 **2 3 4 5** 6 7 8   record ack5 arrived   rcv pkt2; deliver pkt2,
0 1 **2 3 4 5** 6 7 8   pkt3, pkt4, pkt5; send ack2

Only retransmit the
unacked pkt (SR)   *what happens when ack2 arrives?*
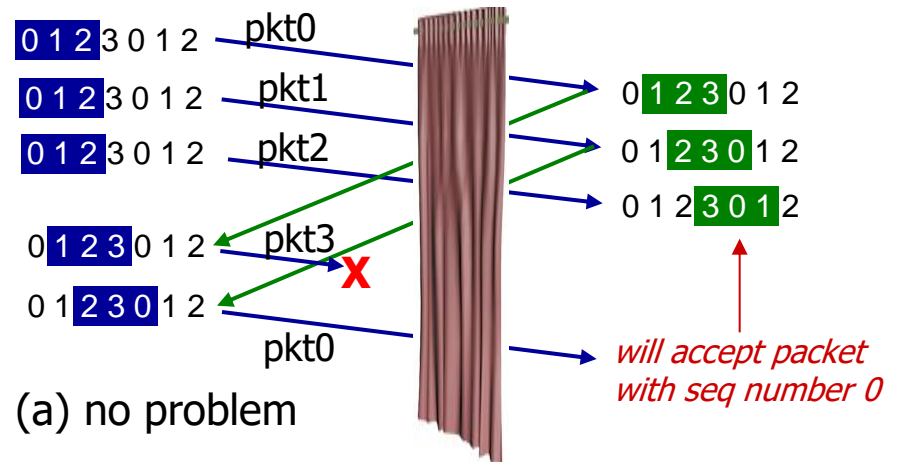
Selective repeat: dilemma

Example:

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

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

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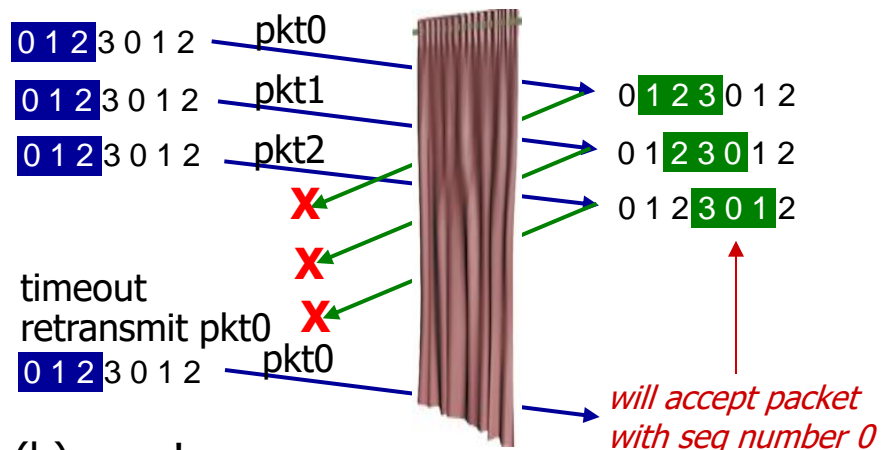

sender window
(after receipt)

receiver window
(after receipt)

0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt2   0 1 2 3 0 1 2
                           0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt3
0 1 2 3 0 1 2
                pkt0

will accept packet
with seq number 0

(a) no problem

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

0 1 2 3 0 1 2   pkt0
0 1 2 3 0 1 2   pkt1   0 1 2 3 0 1 2
0 1 2 3 0 1 2   pkt2   0 1 2 3 0 1 2
                           0 1 2 3 0 1 2

timeout
retransmit pkt0
0 1 2 3 0 1 2   pkt0

will accept packet
with seq number 0

(b) oops!

GBN and SR comparison

Go-back-N:

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

Selective Repeat:

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

Chapter 3 outline

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

- ❖ point-to-point:
  - one sender, one receiver
  - No buffers or variables are allocated to network elements between hosts
- ❖ reliable, in-order *byte stream:*
  - no "message boundaries"
  - Seq # and Ack # are in unit of byte, rather than pkt
- ❖ pipelined:
  - TCP congestion and flow control set window size

- ❖ full duplex data:
  - bi-directional data flow in same connection
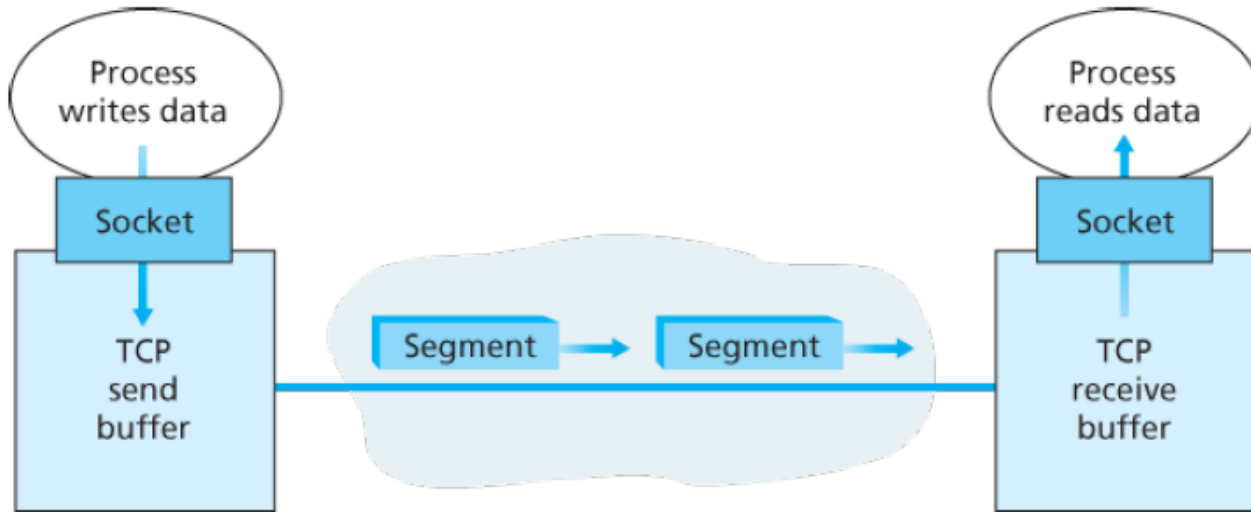 - MSS: maximum segment size
- ❖ connection-oriented:
  - handshaking (exchange of control msgs) initiates sender and receiver state before data exchange
- ❖ flow controlled:
  - sender will not overwhelm receiver
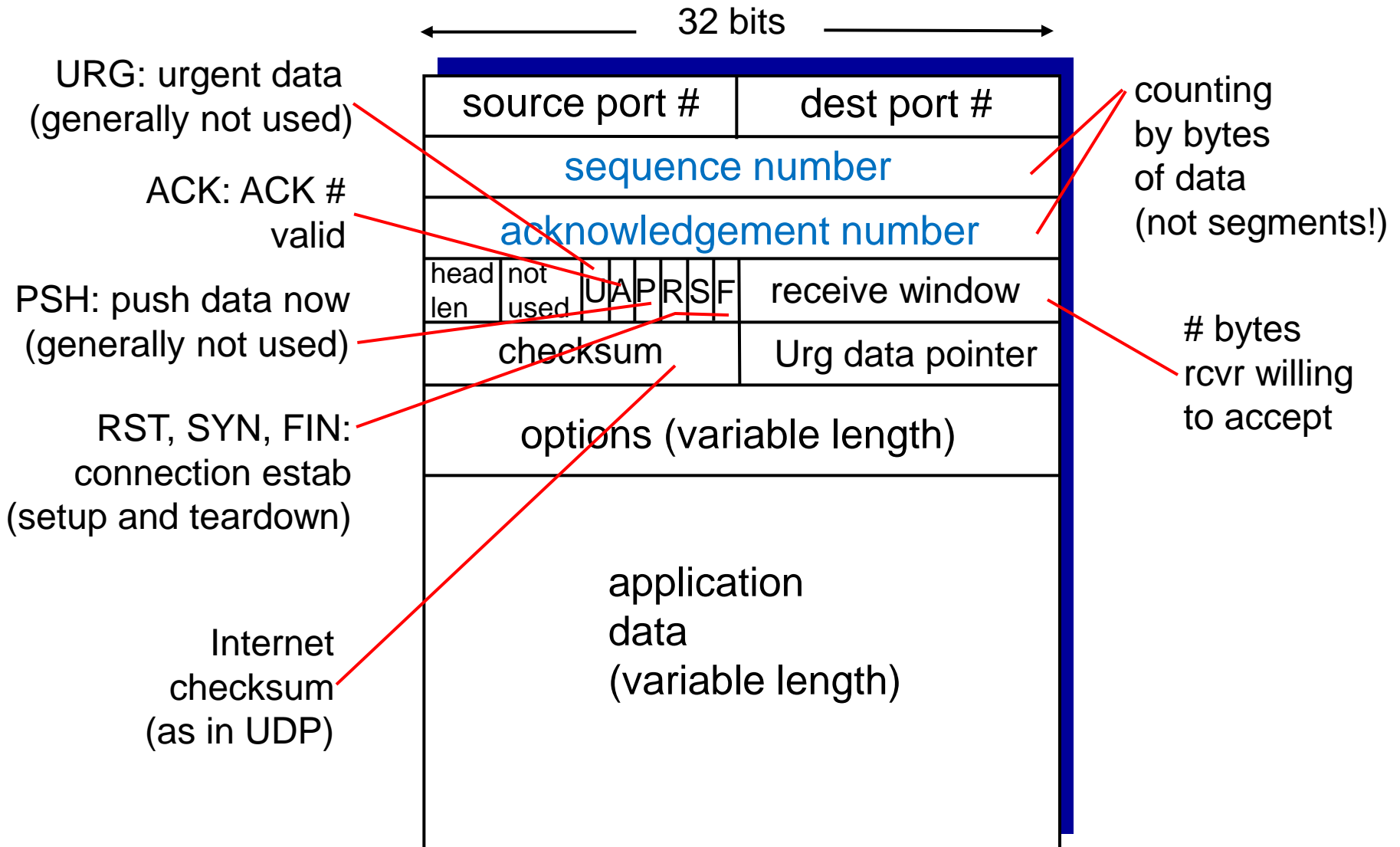
TCP: Overview

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

TCP Reliable Data Transfer

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

TCP segment structure

32 bits

URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

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

Internet
checksum
(as in UDP)

source port #	dest port #
sequence number	
acknowledgement number	

head len	not used	U A P R S F	receive window

checksum	Urg data pointer

options (variable length)

application
data
(variable length)

counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept

TCP seq. numbers, ACKs

TCP views data as an unstructured, but ordered, stream of bytes.
- Sequence numbers are over the stream of transmitted bytes and *not* over the series of transmitted segments

sequence numbers:
- byte stream "number" of first byte in segment's data

File

Data for 1st segment   Data for 2nd segment

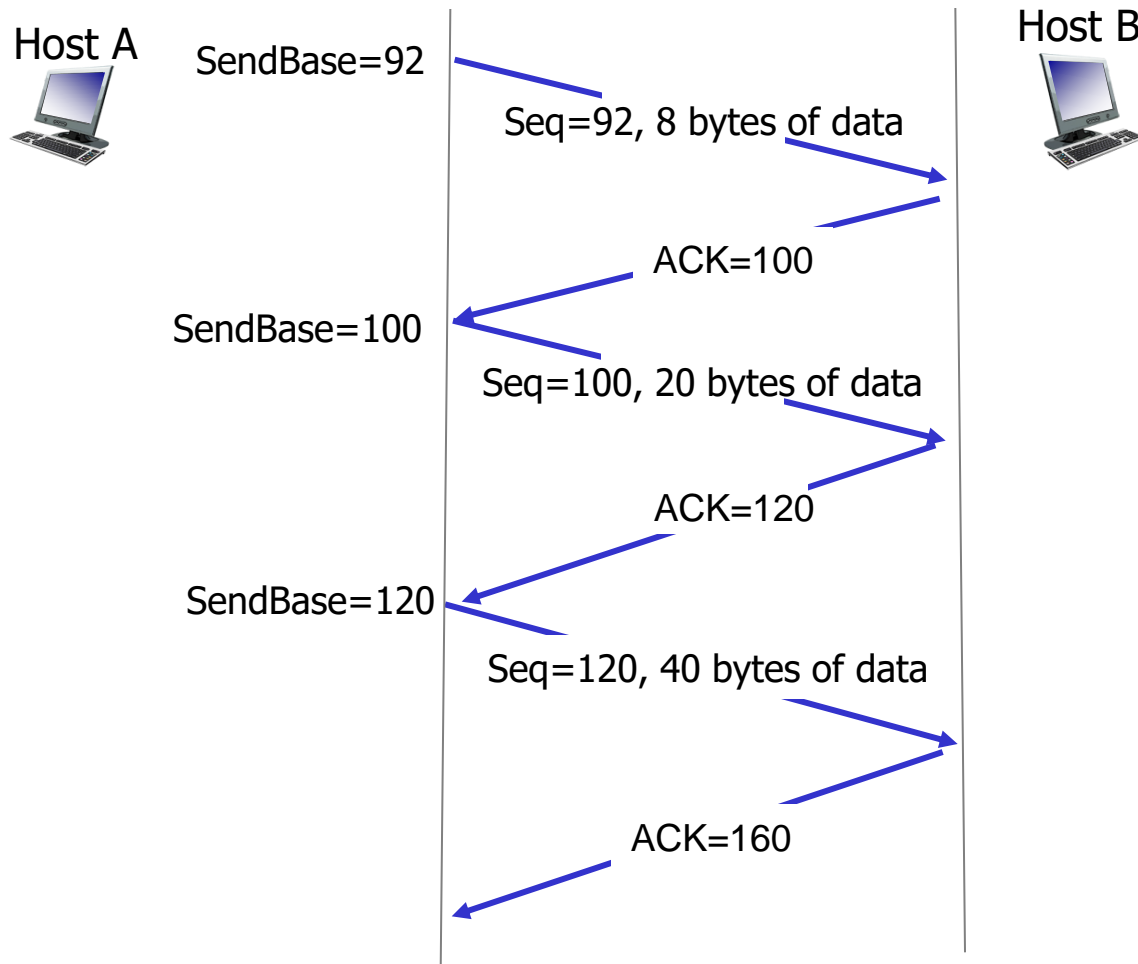| 0 | 1 | // | 1,000 | // | 1,999 | // | 499,999 |
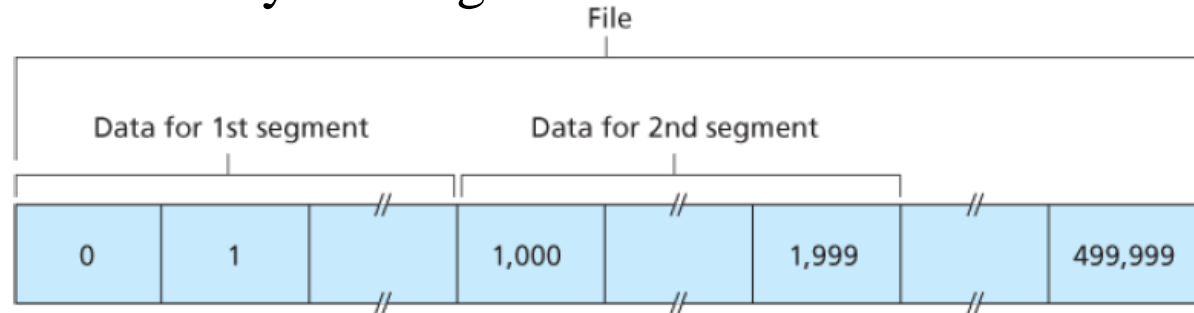
acknowledgements:
- seq # of next byte expected from other side
    - E.g., receiver has received bytes numbered 0 through 535 and 900 through 1000; then, acknowledgement number is 536.
- cumulative ACK

Q: how receiver handles out-of-order segments
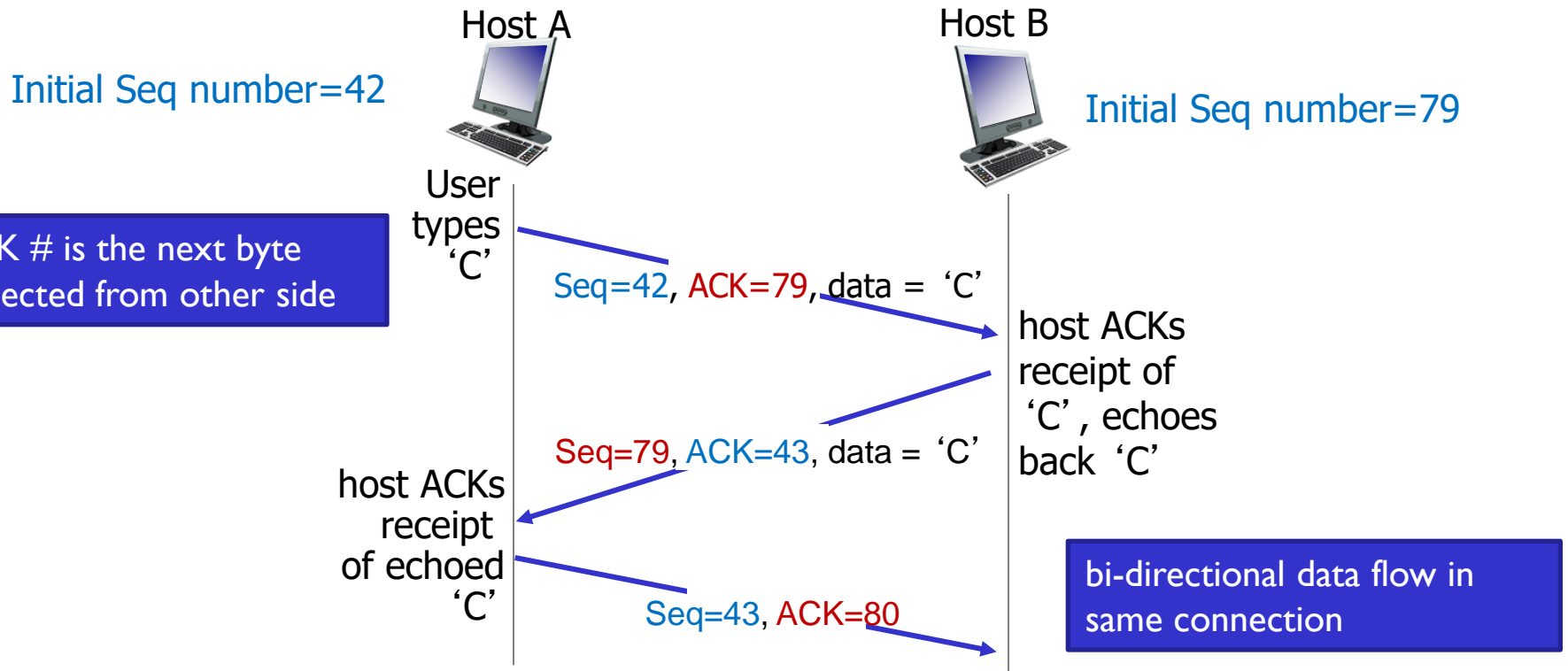- A: TCP spec doesn't say, - up to implementor

Initial sequence number is randomly chosen

TCP Example

Host A

Host B

SendBase=92

Seq=92, 8 bytes of data

ACK=100

SendBase=100

Seq=100, 20 bytes of data

ACK=120

SendBase=120

Seq=120, 40 bytes of data

ACK=160

Telnet Case Study

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

Host A

Host B

Initial Seq number=42

Initial Seq number=79

User types 'C'

ACK # is the next byte expected from other side

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

bi-directional data flow in same connection

TCP Reliable Data Transfer

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

TCP round trip time, timeout

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

Q: How to estimate RTT?
❖ **SampleRTT**: measured time from segment transmission until ACK receipt
  ▪ ignore retransmissions
❖ **SampleRTT** will vary, want estimated RTT "smoother"
  ▪ average several *recent* measurements, not just current **SampleRTT**

TCP round trip time, timeout

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

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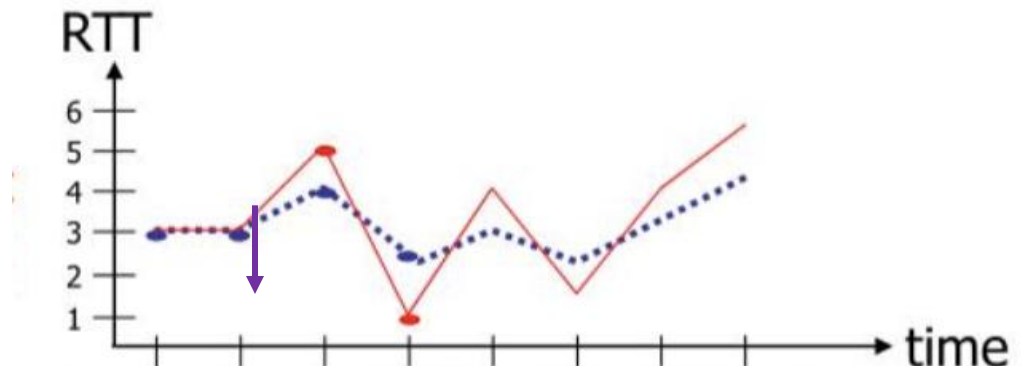- ❖ typical value: $\alpha = 0.125$
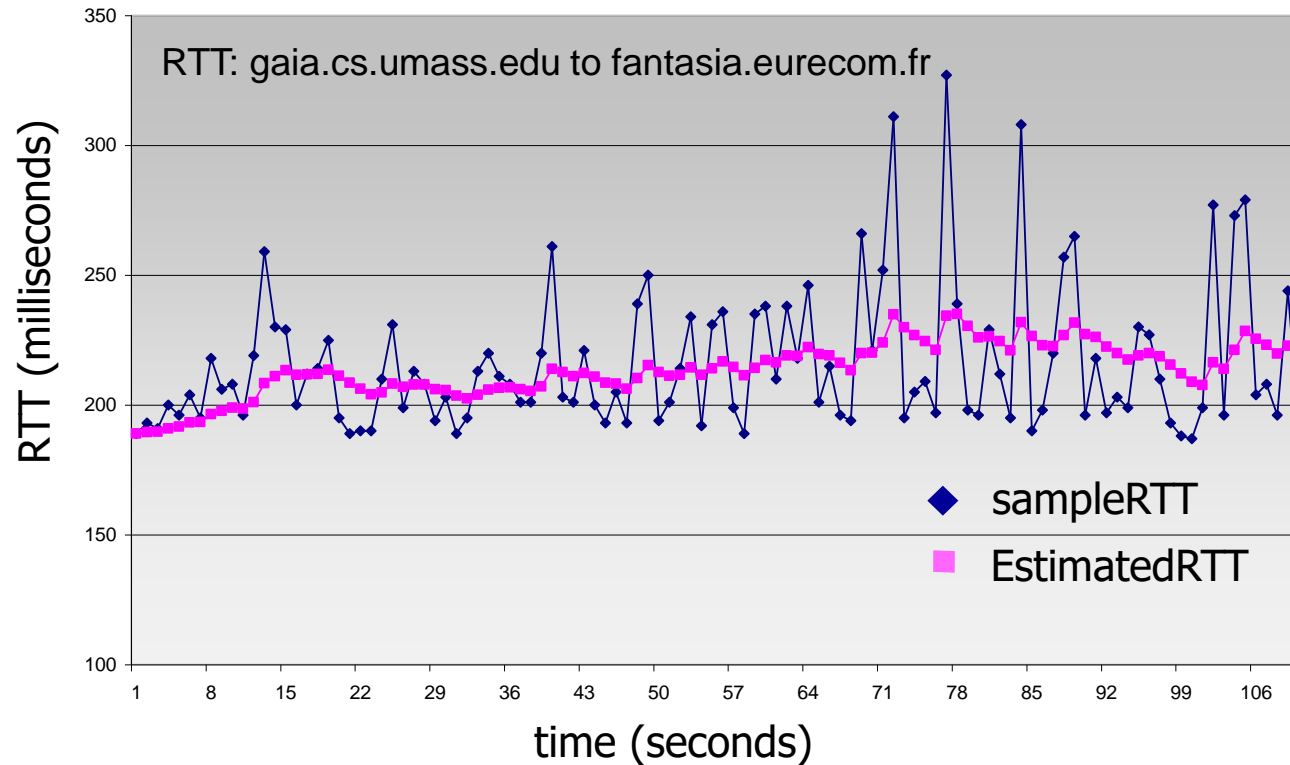
Example:

Suppose $\alpha = 0.5$
EstimatedRTT = 3

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

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

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

TCP round trip time, timeout

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

RTT (milliseconds)

time (seconds)

◆ sampleRTT

■ EstimatedRTT

TCP round trip time, timeout

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

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

TCP timeout interval: `EstimatedRTT` plus "safety margin"
  large variation in `EstimatedRTT` -> larger safety margin

TimeoutInterval = EstimatedRTT + 4*DevRTT

estimated RTT   "safety margin"

TCP Reliable Data Transfer

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

TCP reliable data transfer

❖ TCP creates rdt service on top of IP's unreliable service
  ▪ pipelined segments: window size, SendBase
  ▪ cumulative acks
  ▪ single retransmission timer
❖ retransmissions triggered by:
  ▪ timeout events
  ▪ duplicate acks

Let's initially consider simplified TCP sender:
  ▪ ignore duplicate acks
  ▪ ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- ❖ create segment with seq #
- ❖ seq # is byte-stream number of first data byte in segment
- ❖ start timer if not already running
  - ▪ think of timer as for oldest unacked segment
  - ▪ expiration interval: **TimeoutInterval**

timeout:

- ❖ retransmit segment that caused timeout
- ❖ restart timer

ack rcvd:

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

TCP sender events:

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

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

} /* end of loop forever */
```
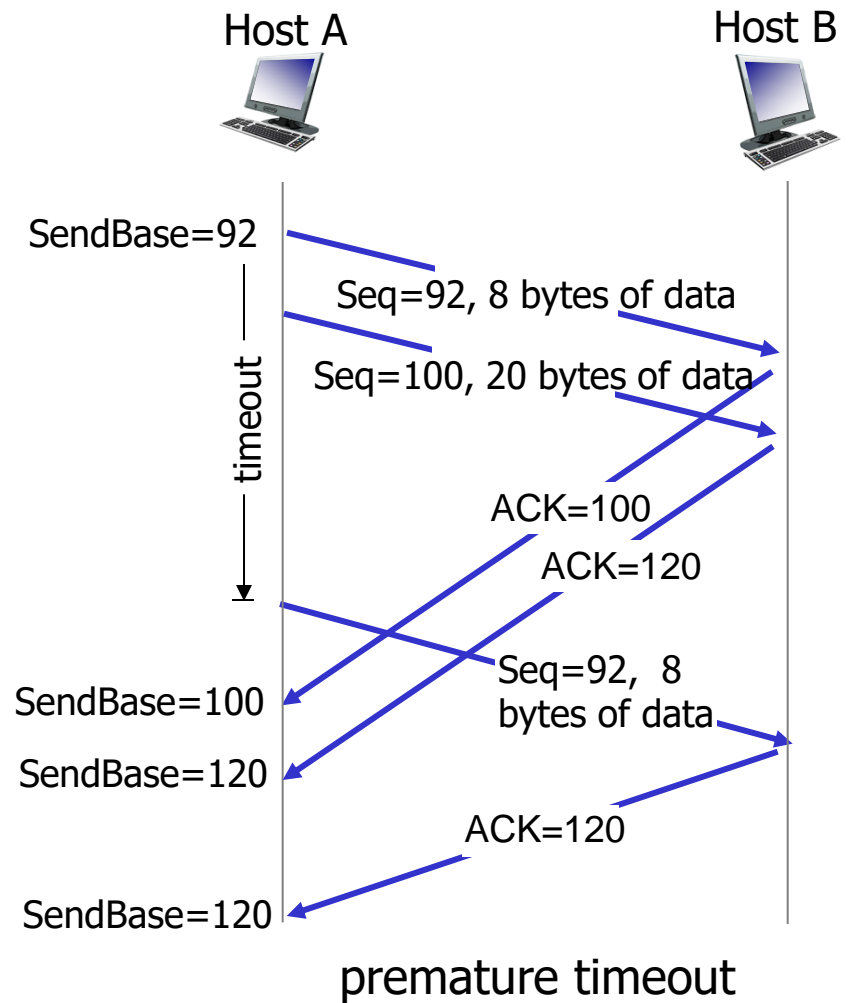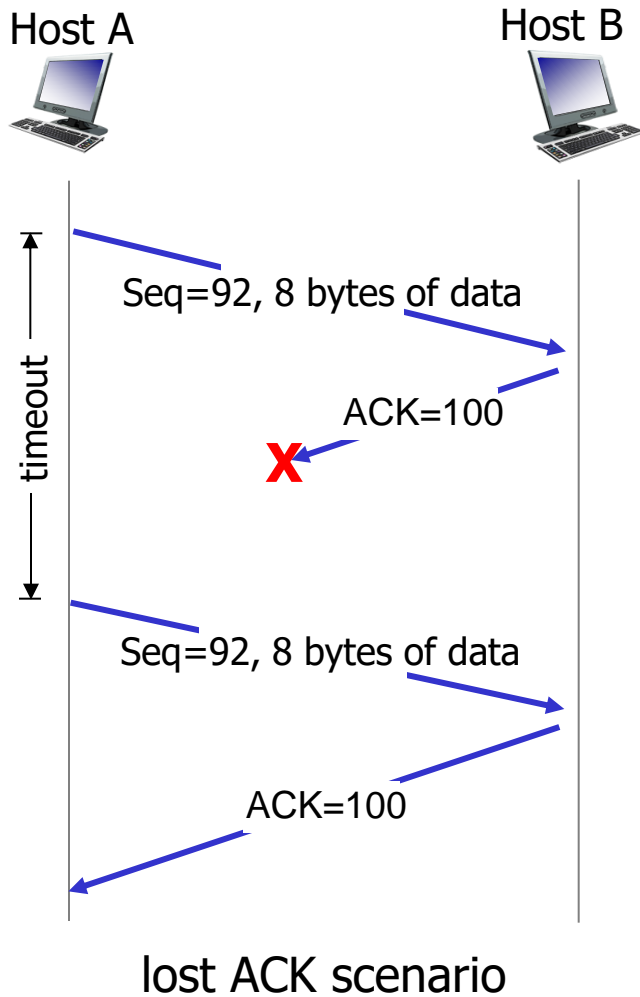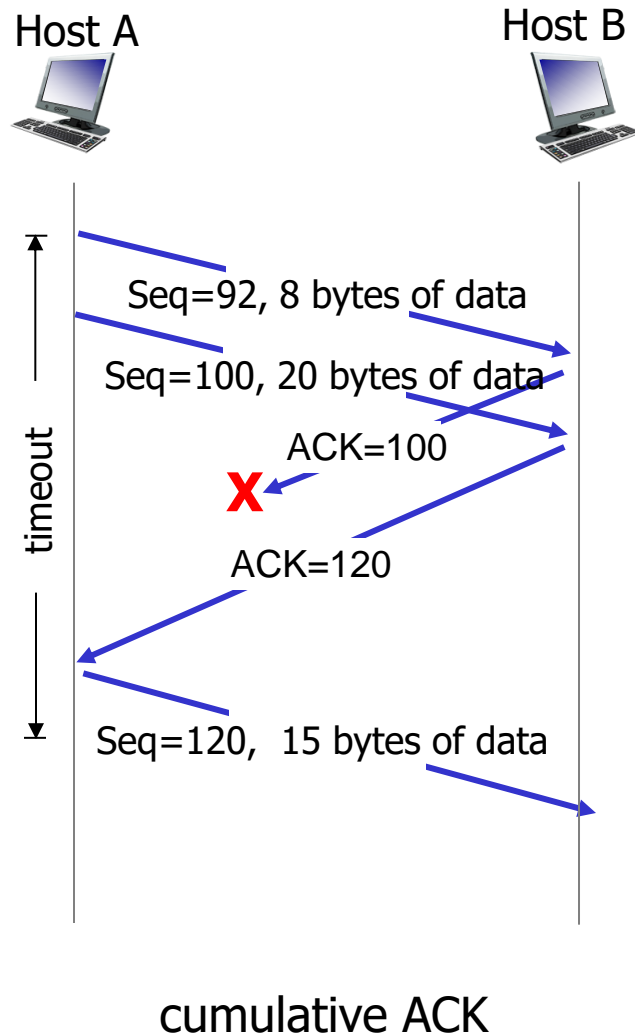
TCP: retransmission scenarios

lost ACK scenario

premature timeout

TCP: retransmission scenarios

Host A   Host B

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

X

ACK=120

timeout

Seq=120, 15 bytes of data

cumulative ACK

TCP receiver [RFC 1122, RFC 2581]

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

TCP fast retransmit

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

> **TCP fast retransmit**
>
> if sender receives 3 duplicate ACKs for same data
>
> ("triple duplicate ACKs"), resend unacked segment with smallest seq #
>
> - likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

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

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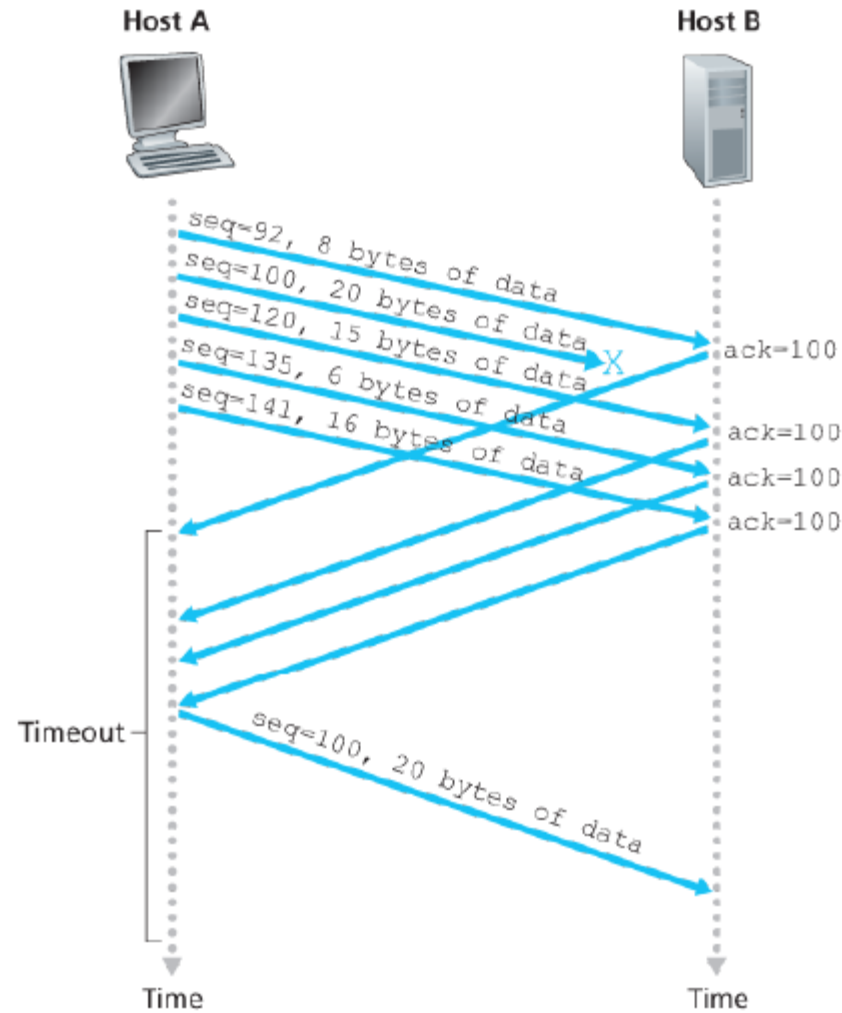
```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
            SendBase=y
            if (there are currently any not yet
                    acknowledged segments)
                start timer

            }
        else {/* a duplicate ACK for already ACKed
                segment */
            increment number of duplicate ACKs
                received for y
            if (number of duplicate ACKS received
                for y==3)
                /* TCP fast retransmit */
                resend segment with sequence number y
            }
        break;
```

TCP fast retransmit

Host A

Host B

seq=92, 8 bytes of data
seq=100, 20 bytes of data
seq=120, 15 bytes of data
seq=135, 6 bytes of data
seq=141, 16 bytes of data

X

ack=100

ack=100

ack=100
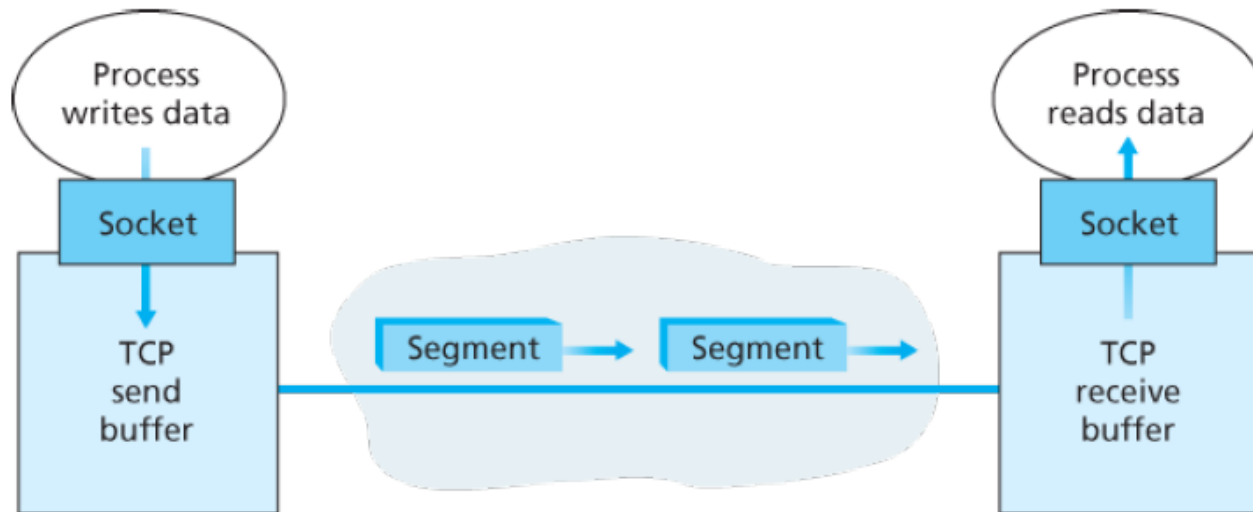
ack=100

Timeout

seq=100, 20 bytes of data

Time

Time

fast retransmit after sender
receipt of triple duplicate ACK

TCP Reliable Data Transfer

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

TCP: Overview

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

TCP flow control

application may
remove data from
TCP socket buffers ….

application
process

TCP socket
receiver buffers

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

application

- - - - - -

OS

TCP
code

IP
code

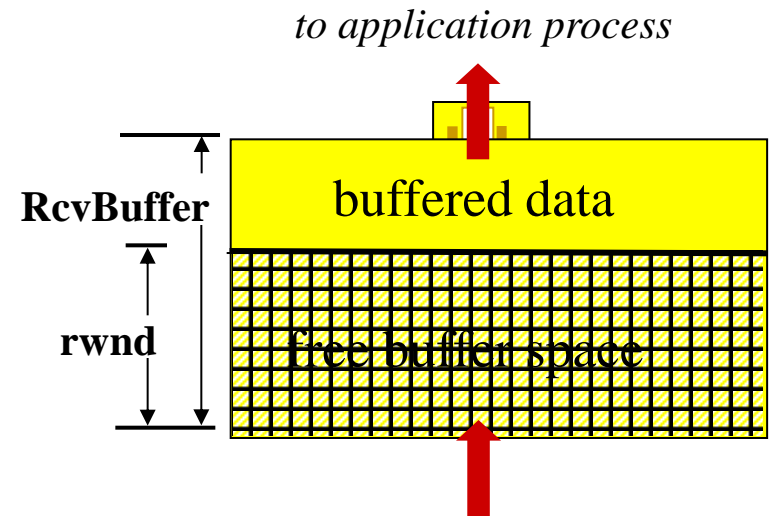*flow control*

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

from sender

receiver protocol stack
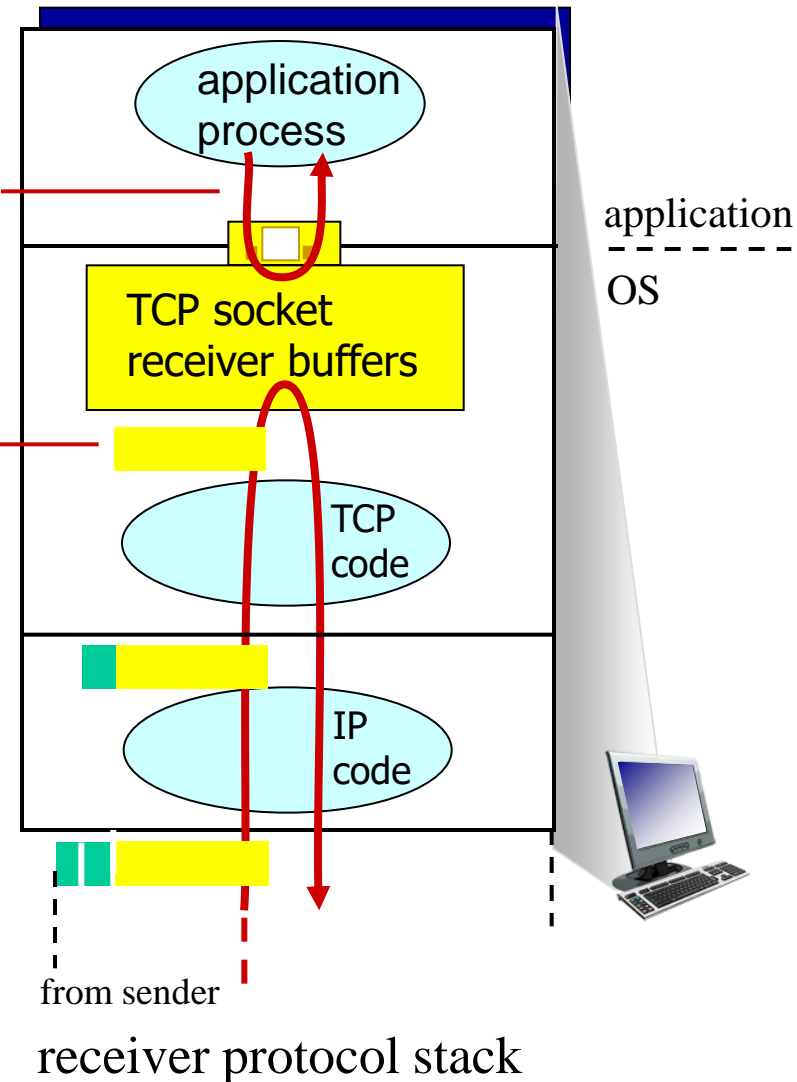
TCP flow control

Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
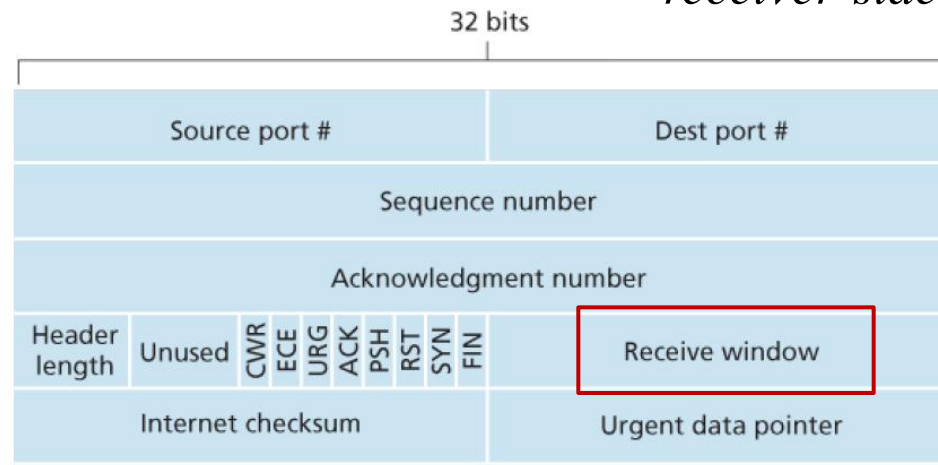
- ❖ **RcvBuffer** size set via socket options (typical default is 4096 bytes)
- ❖ many operating systems autoadjust **RcvBuffer**

$$rwnd=RcvBuffer−[LastByteRcvd−LastByteRead]$$

to application process

RcvBuffer

buffered data

rwnd

free buffer space

TCP segment payloads

receiver-side buffering

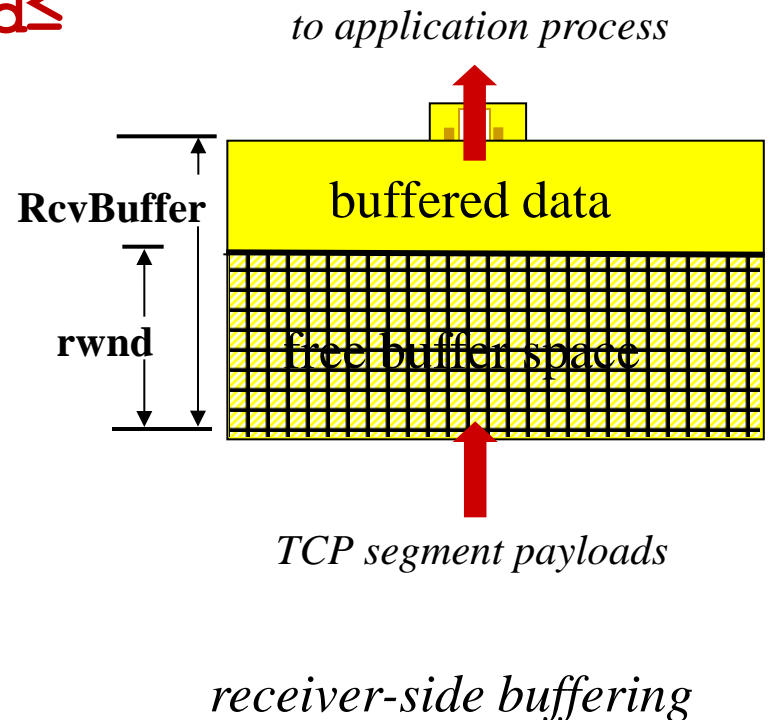| 32 bits	
Source port #	Dest port #
Sequence number	
Acknowledgment number	
Header length / Unused / CWR ECE URG ACK PSH RST SYN FIN	Receive window
Internet checksum	Urgent data pointer

TCP flow control

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

LastByteSent−LastByteAcked≤ rwnd

Guarantees receive buffer will not overflow

to application process

RcvBuffer

buffered data

rwnd

free buffer space

TCP segment payloads
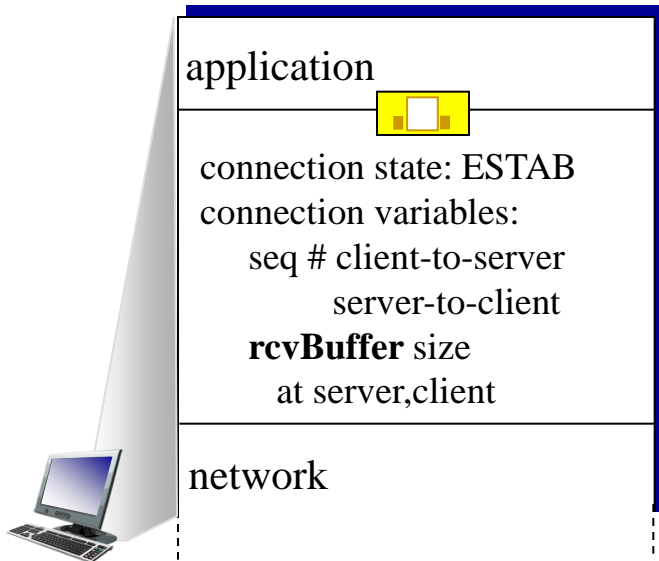
receiver-side buffering

TCP Reliable Data Transfer

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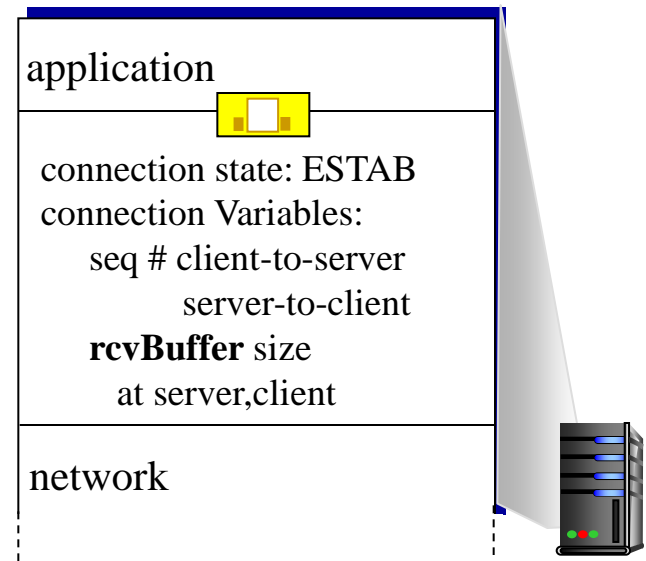- ❖ Reliable data transfer
- ❖ Flow control
- ❖ Control management

Connection Management

before exchanging data, sender/receiver "handshake":

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

❖ agree on connection parameters

Client side (left):

application

connection state: ESTAB
connection variables:
 seq # client-to-server
  server-to-client
 rcvBuffer size
  at server,client

network

Server side (right):

application

connection state: ESTAB
connection Variables:
 seq # client-to-server
  server-to-client
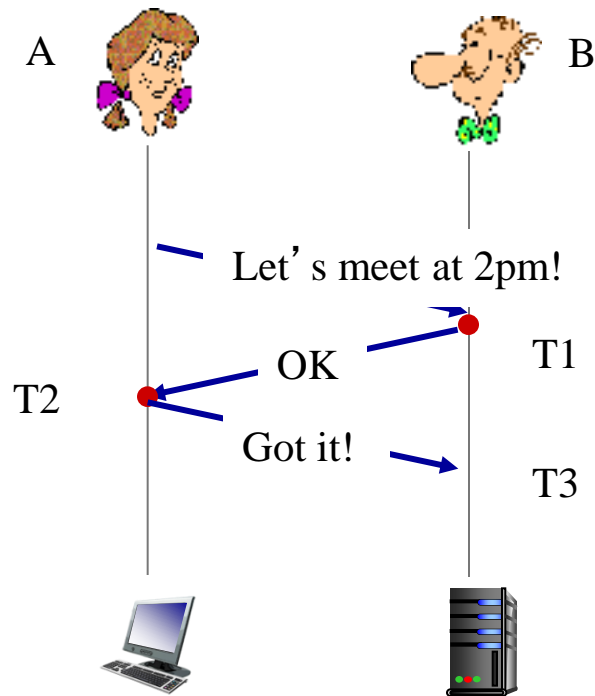 **rcvBuffer** size
  at server,client

network

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

```
connectionSocket = welcomeSocket.accept();
```
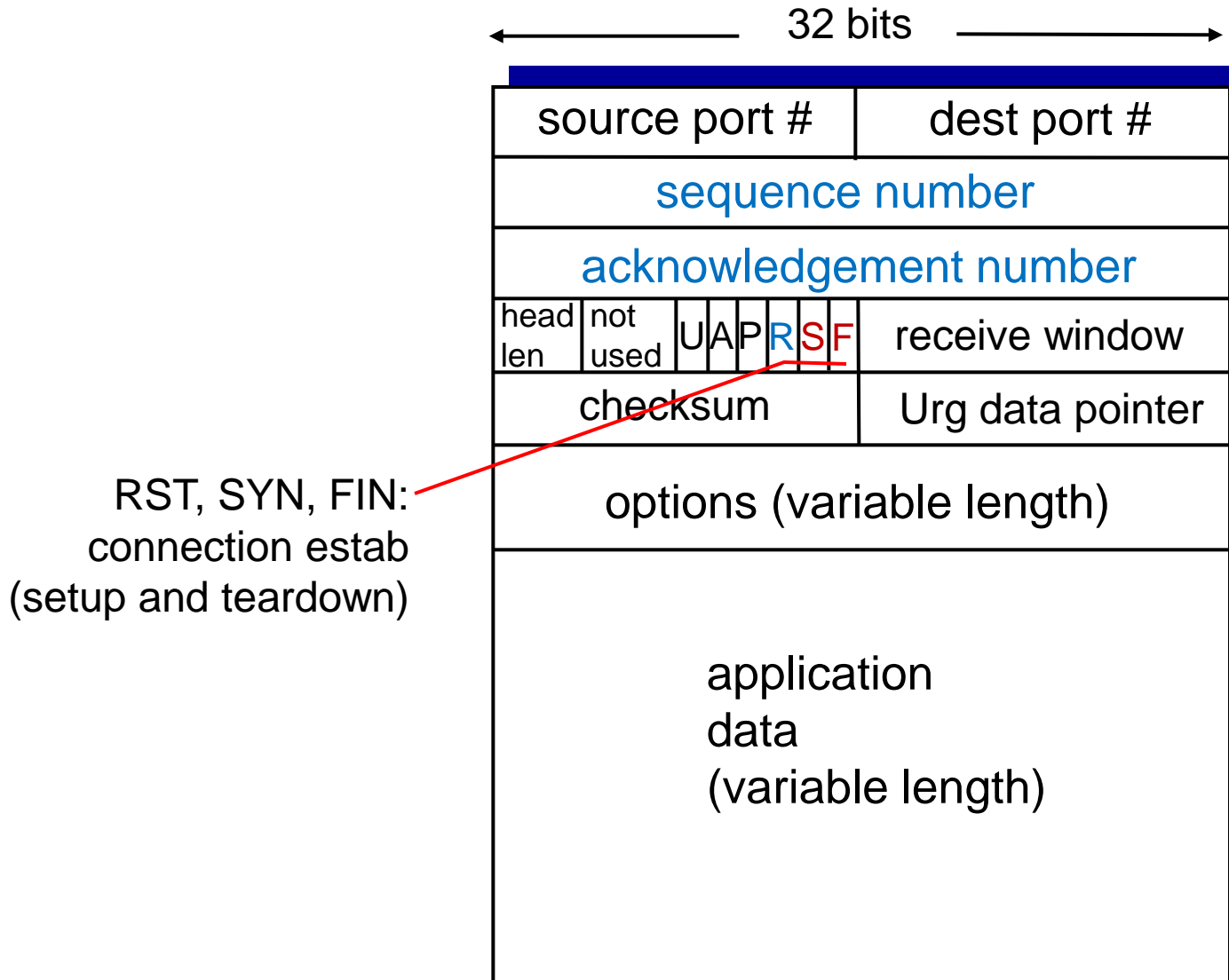
Agreeing to establish a connection

3-way handshake:

- ❖ T1 : B knows A's transmitter and B's receiver is OK

- ❖ T2: A knows A's transceiver and B's transceiver is OK, B has no more information than T1

- ❖ T3: Both A and B know their transceiver are OK, they can start the communication!

TCP segment structure

32 bits

source port #	dest port #
sequence number	
acknowledgement number	

head len	not used	U	A	P	R	S	F	receive window

checksum	Urg data pointer

options (variable length)

application
data
(variable length)

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

TCP 3-way handshake

client state

LISTEN

SYNSENT

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

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

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

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

server state

LISTEN

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

SYN RCVD

received ACK(y)
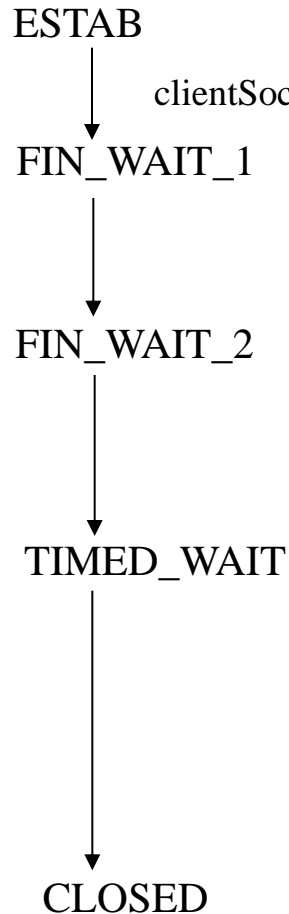indicates client is live

ESTAB

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

- In each of these future segments, SYNbit=0

TCP: closing a connection

client state

ESTAB

clientSocket.close()

FIN_WAIT_1   can no longer send but can receive data

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

FIN_WAIT_2   wait for server close

FINbit=1, seq=y

TIMED_WAIT

ACKbit=1; ACKnum=y+1

timed wait

CLOSED

server state

ESTAB

CLOSE_WAIT   can still send data

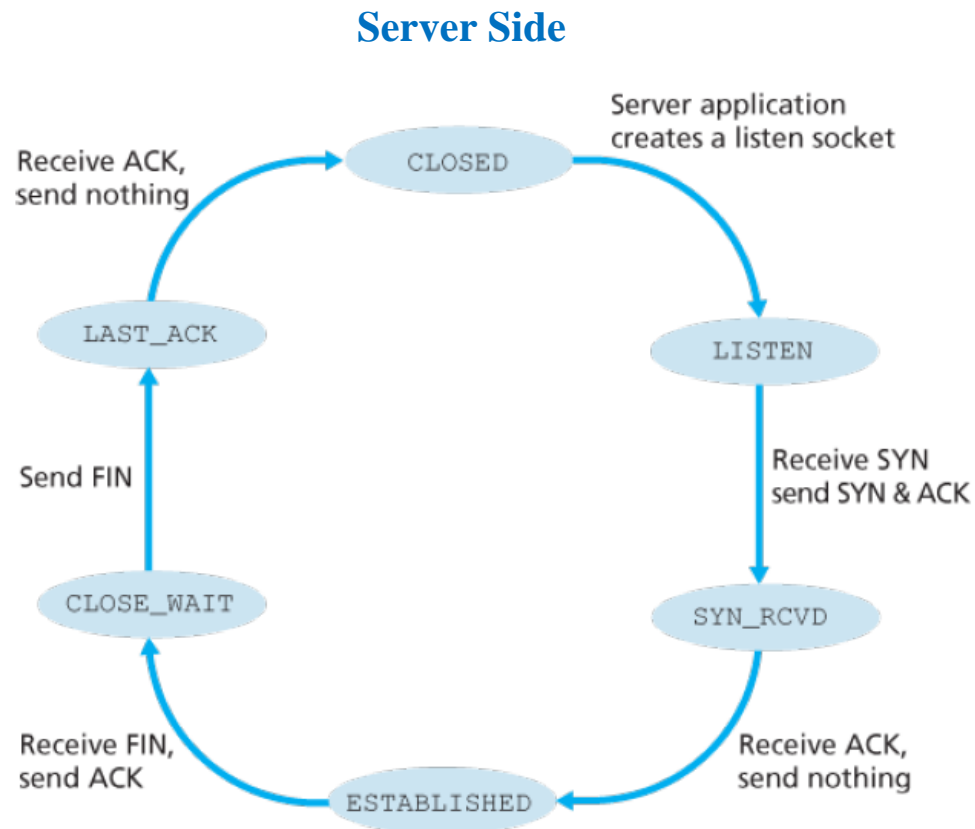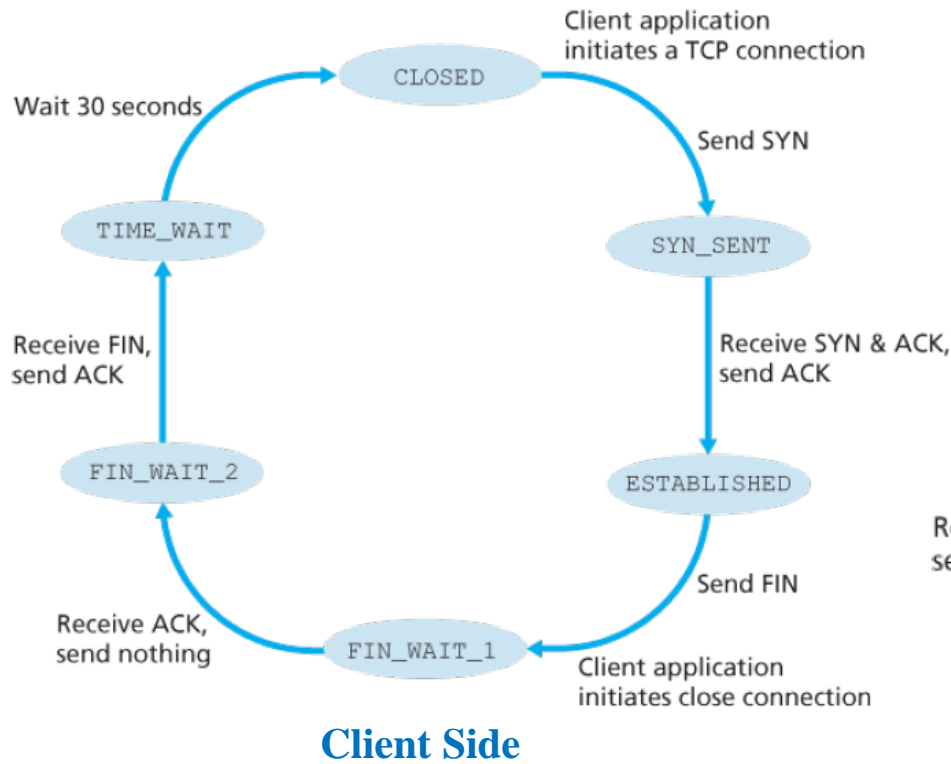LAST_ACK   can no longer send data

CLOSED
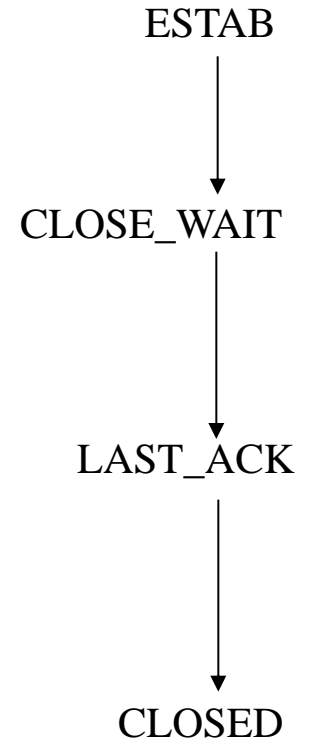
The TIME_WAIT state lets the TCP client resend the final acknowledgment in case the ACK is lost.

TCP: closing a connection

- ❖ Four-way handshaking
  - Either of the two processes participating in a TCP connection can end the connection.

- ❖ client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- ❖ respond to received FIN with ACK

- ❖ Why FIN and ACK can not be sent in one msg as SYNACK in connection establishment?
  - The other side may still have packets need to be sent. It can not send FIN until the transmission is finished.

TCP States

Client application
initiates a TCP connection

CLOSED

Send SYN

Wait 30 seconds

TIME_WAIT

SYN_SENT

Receive SYN & ACK,
send ACK

Receive FIN,
send ACK

Server Side

FIN_WAIT_2

ESTABLISHED

Server application
creates a listen socket

Receive ACK,
send nothing

CLOSED

Receive ACK,
send nothing

Send FIN

FIN_WAIT_1

Client application
initiates close connection

LAST_ACK

LISTEN

Client Side

Receive SYN
send SYN & ACK

Send FIN

CLOSE_WAIT

SYN_RCVD

Receive FIN,
send ACK

Receive ACK,
send nothing

ESTABLISHED

Reset Segment

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

❖ Then the host will send a special reset segment to the source. RST flag bit is set to 1.

❖ "I don't have a socket for that segment. Please do not resend the segment."

Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

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

3.6 principles of congestion control

3.7 TCP congestion control