Welcome to

***CS 3516**:*
Computer Networks

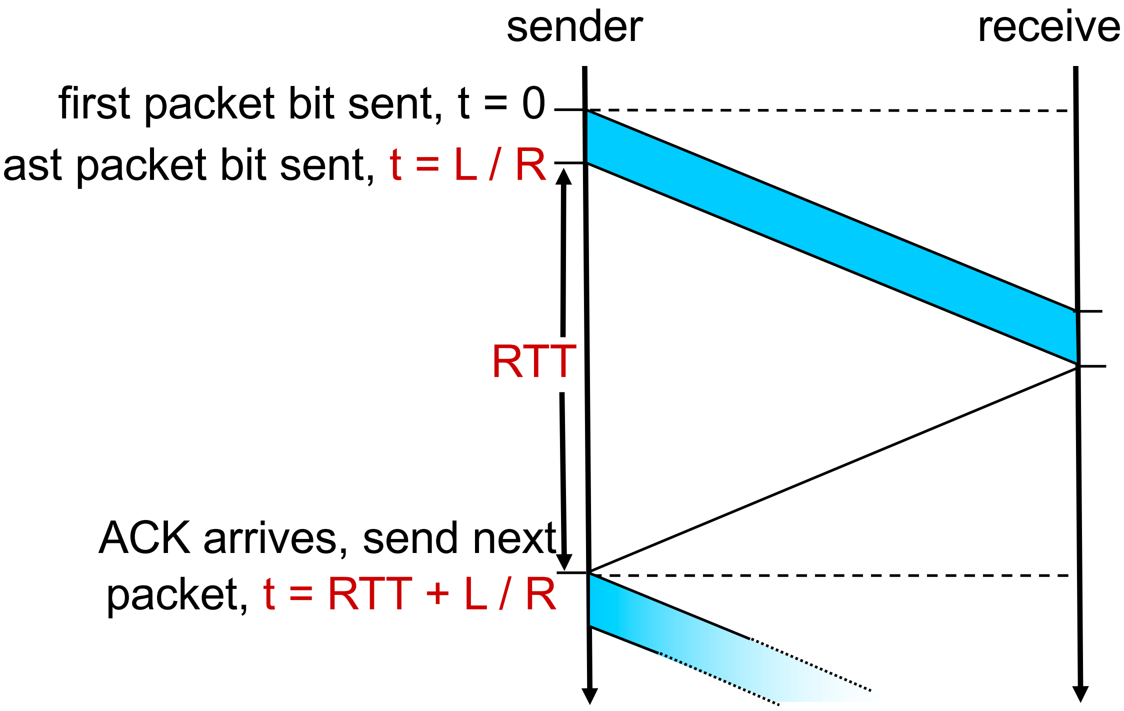
Prof. Yanhua Li

Time: 9:00am –9:50am M, T, R, and F
Location: AK219
Fall 2018 A-term

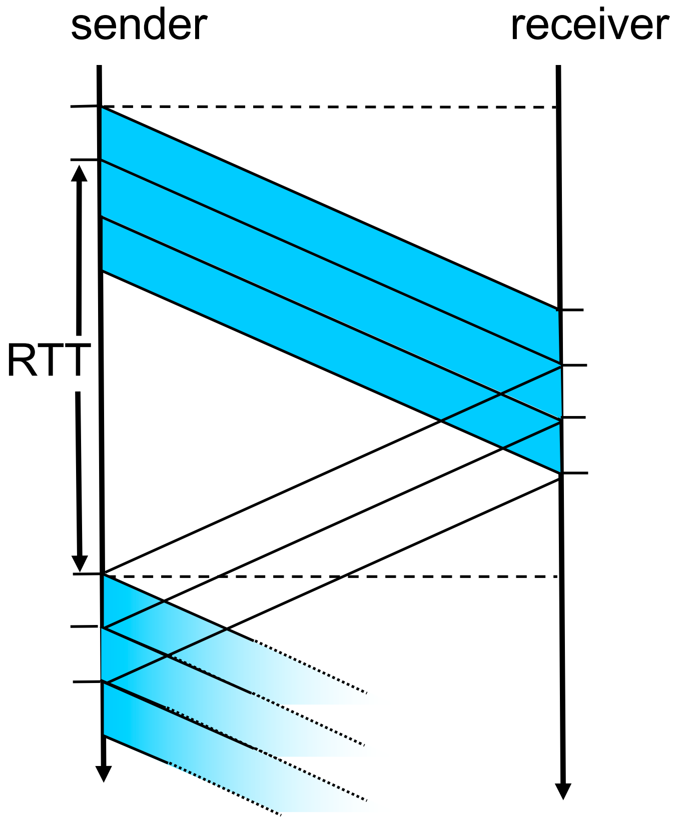
Updates

- Quiz 6
  - Grading by Today
- Mid-term
  - Grading by Wed
- Project 2
  - Due on 9/28 F
  - Extra office hours (TBD)
- Quiz 7:
  - This Thursday
  - *TCP and Network Layer Intro*

rdt3.0 vs Pipelining approach



RDT3.0



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

Pipelining approach

Pipelined protocols: overview

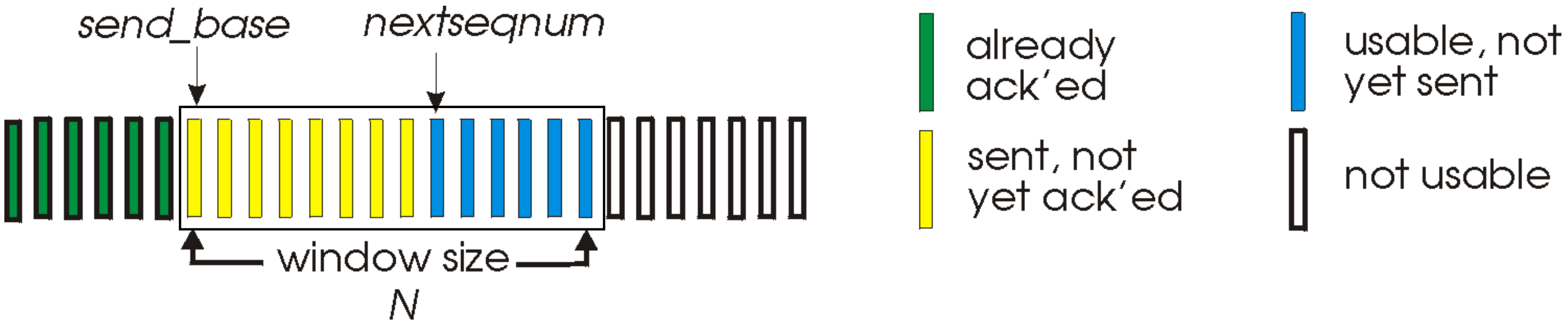
Go-back-N:

- 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

- Extending from
- one unacknowledged pkt (in RDT3.0) to
- multiple unacknowledged pkts (in pipelining)

Go-Back-N: sender

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

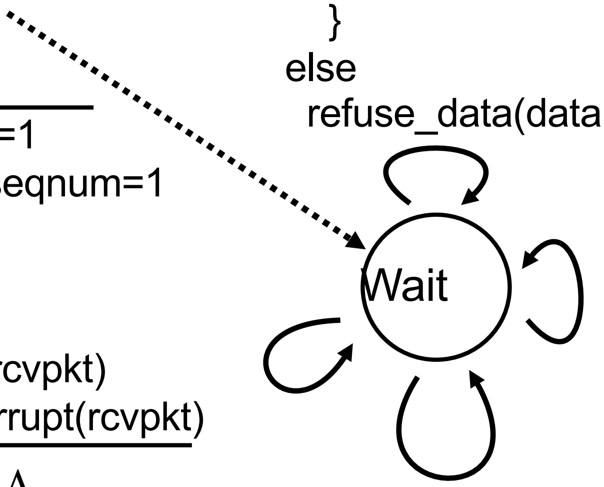


- ACK(n): ACKs all pkts up to, including seq # n - *“cumulative ACK”*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM

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

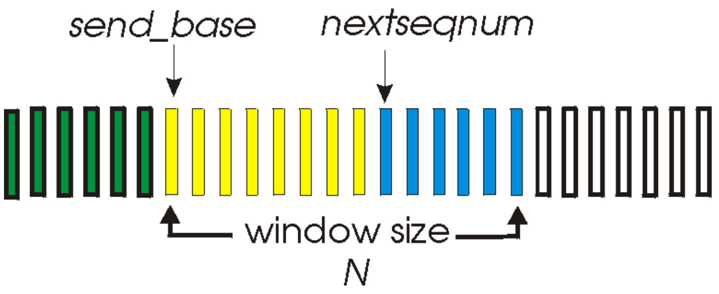
```
Λ
__________
base=1
nextseqnum=1
```



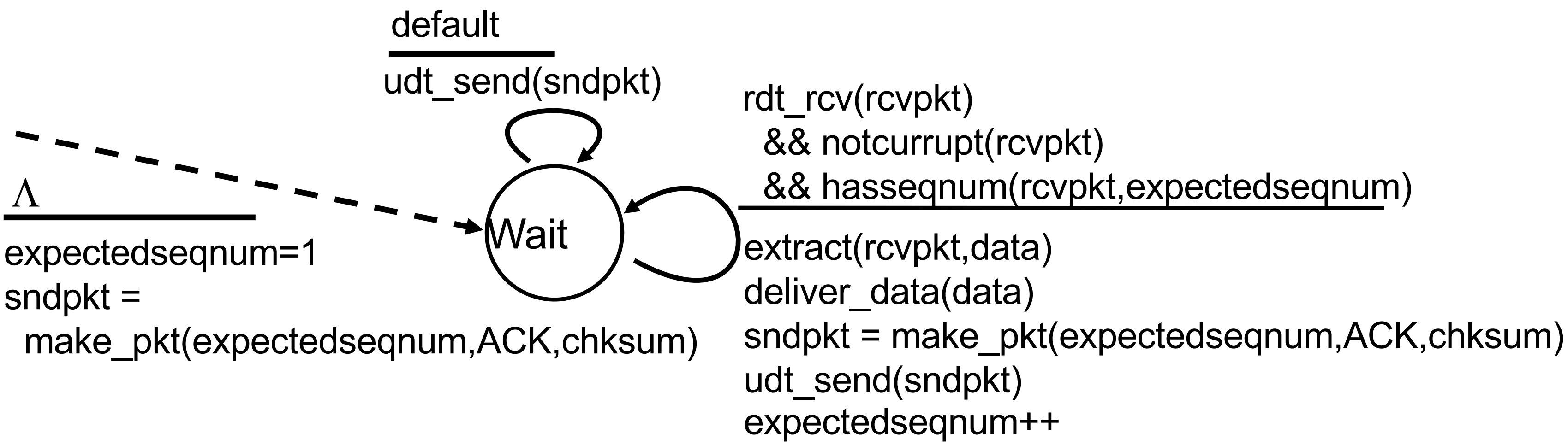
```
timeout
____________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

```
rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
____________________
        Λ
```

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



GBN: receiver extended FSM



default

udt_send(sndpkt)

Λ

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

Wait

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

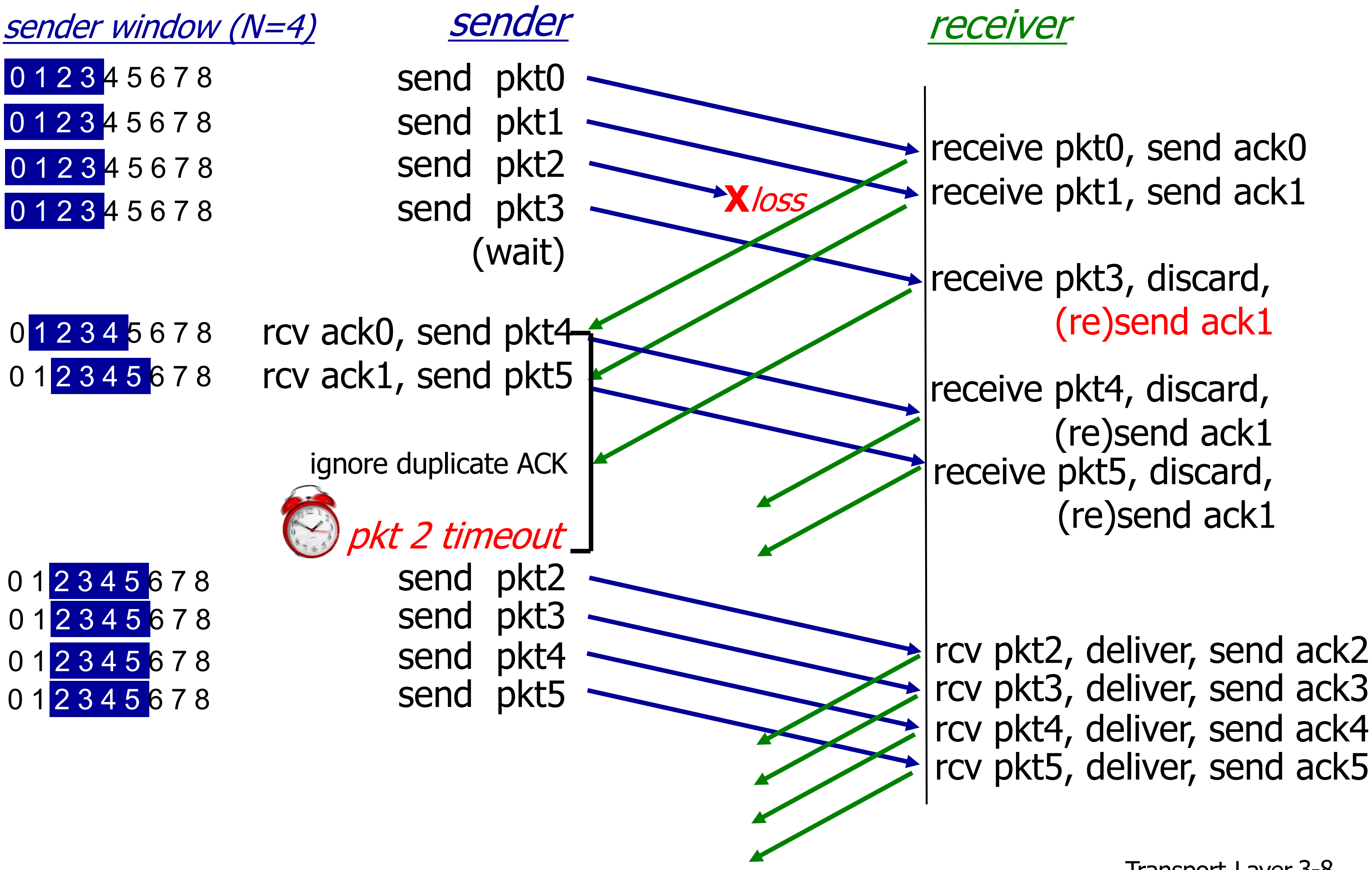
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++

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

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

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

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

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

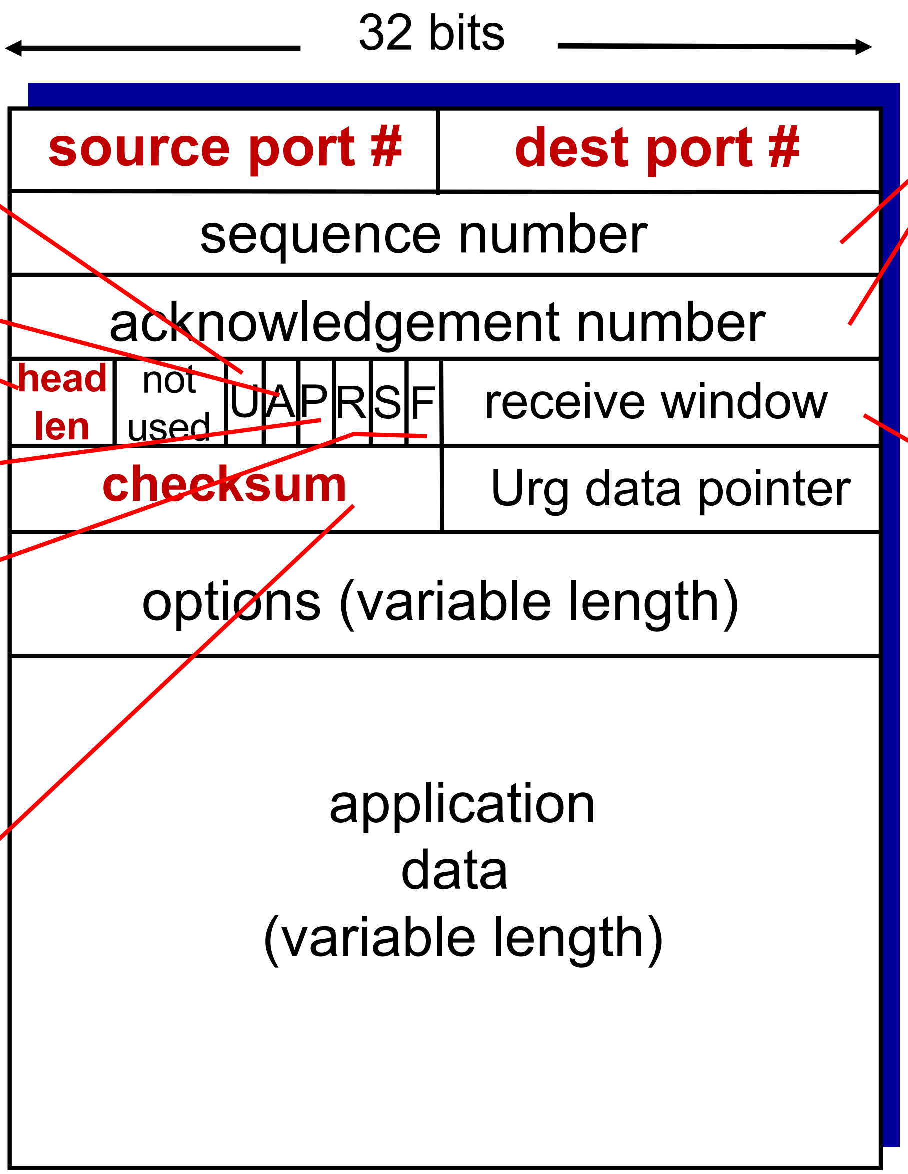
- segment structure
  - Segment Format
  - Sequence #
  - 3 way handshake
  - Timeout value estimation

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (e.g., 1460B)
  - MTU: layer 3 maximum transmission unit (e.g., 1500B for Ethernet)
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange

TCP segment structure (20+ bytes)



URG: urgent data (generally not used)

ACK: ACK # valid

In 4 bytes = 32 bits

PSH: push data now (generally not used)

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

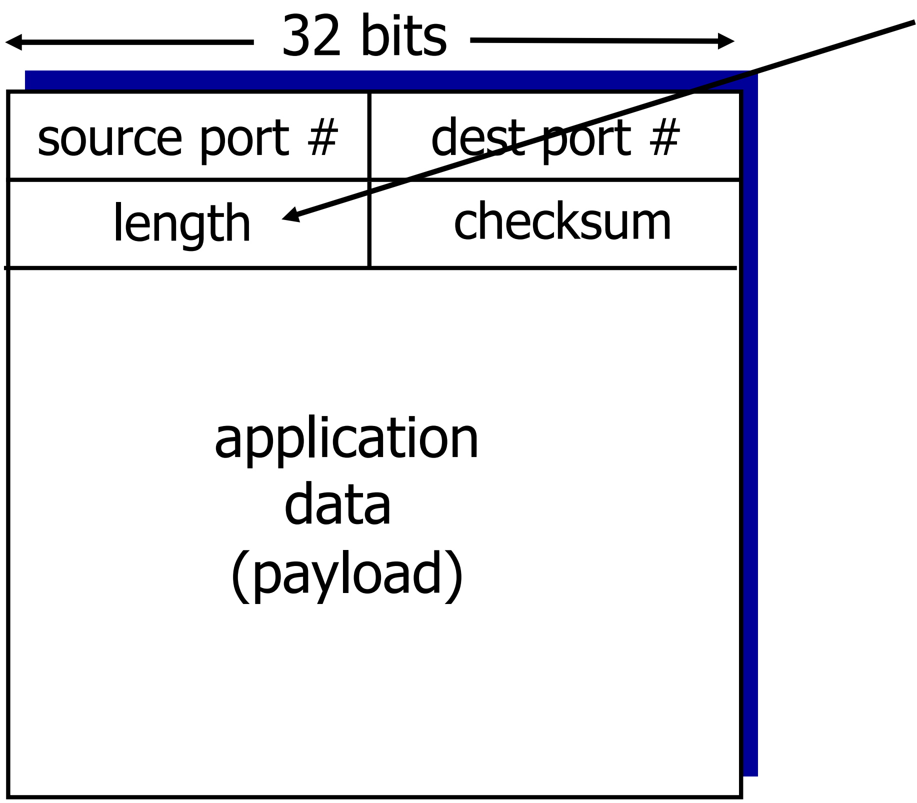
Internet checksum (as in UDP)

counting by bytes of data (not segments!)

# bytes rcvr willing to accept (for flow control)

Length of UDP header?

UDP: segment header (8 bytes)



UDP segment format

length, in bytes of
UDP segment,
including header

TCP seq. numbers, ACKs

sequence numbers:

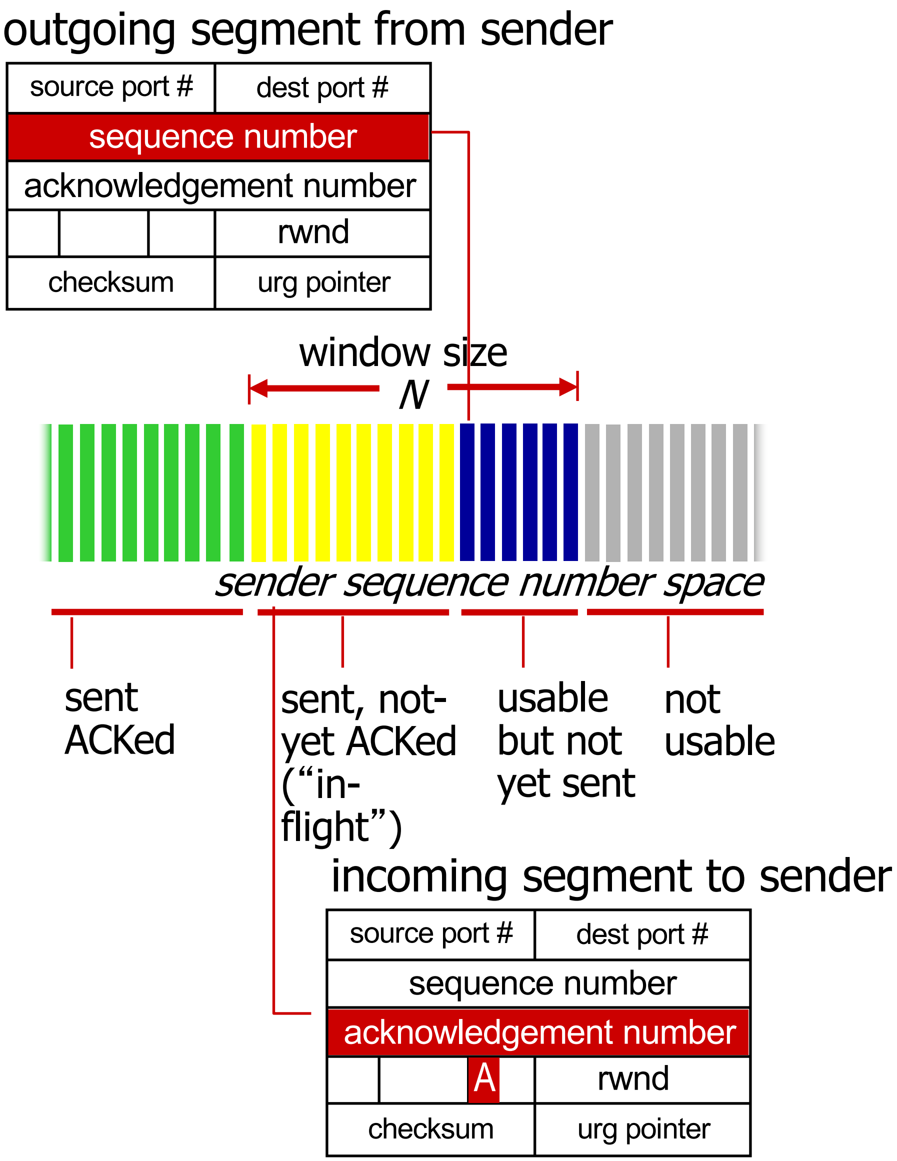
- byte stream "number" of first byte in segment's data

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
- rdt 3.0 & GBN & more



outgoing segment from sender

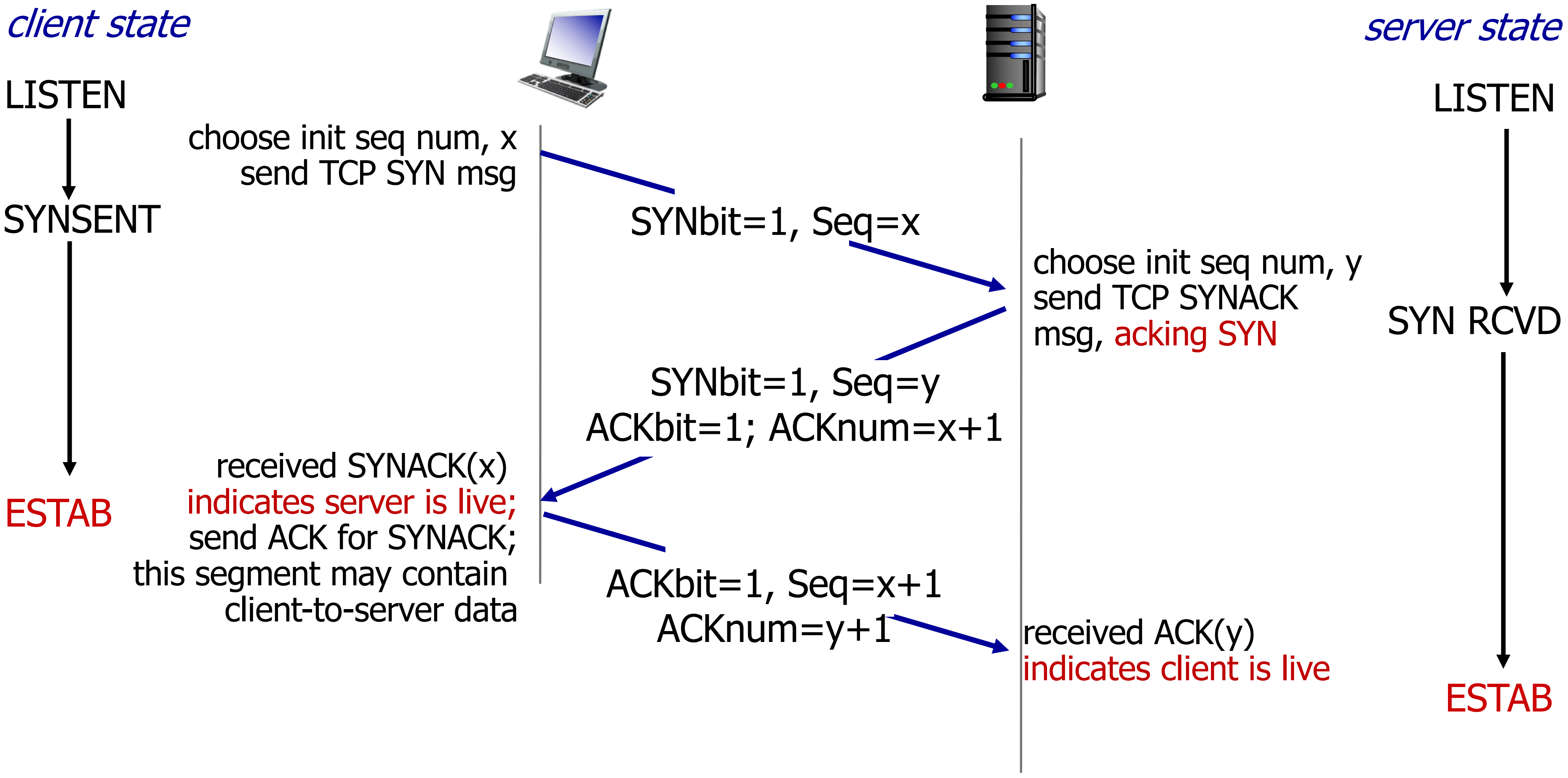
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

window size
N
sender sequence number space
sent ACKed
sent, not-yet ACKed ("in-flight")
usable but not yet sent
not usable

incoming segment to sender

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

TCP 3-way handshake



client state

LISTEN

SYNSENT

ESTAB

server state

LISTEN

SYN RCVD

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

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

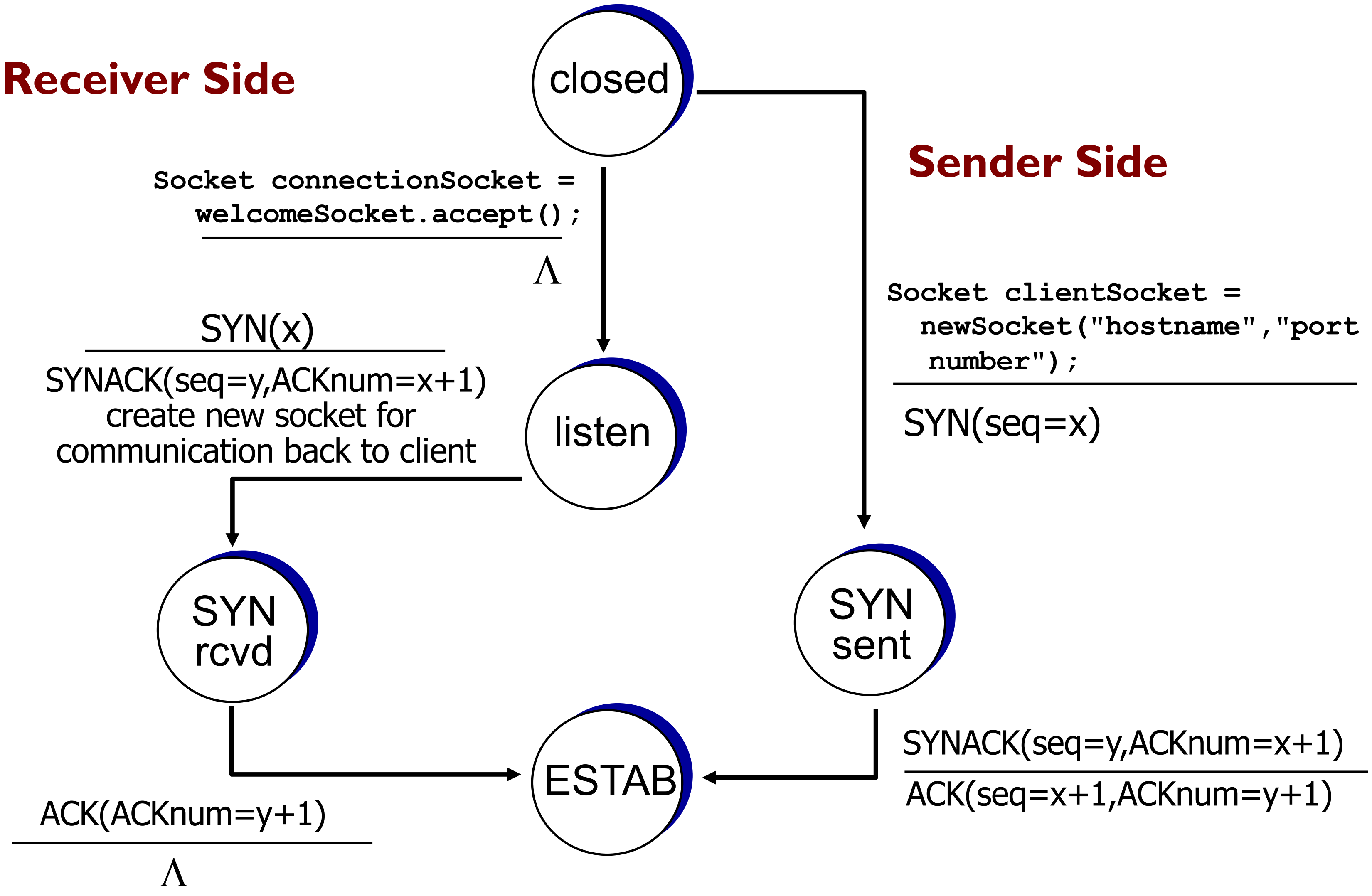
received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

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

received ACK(y)
indicates client is live

Random initial seq # are generated on both sides

TCP 3-way handshake: FSM



Receiver Side

closed

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

Λ

listen

SYN(x)

SYNACK(seq=y,ACKnum=x+1)
create new socket for
communication back to client

SYN rcvd

ACK(ACKnum=y+1)

Λ

ESTAB

Sender Side

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

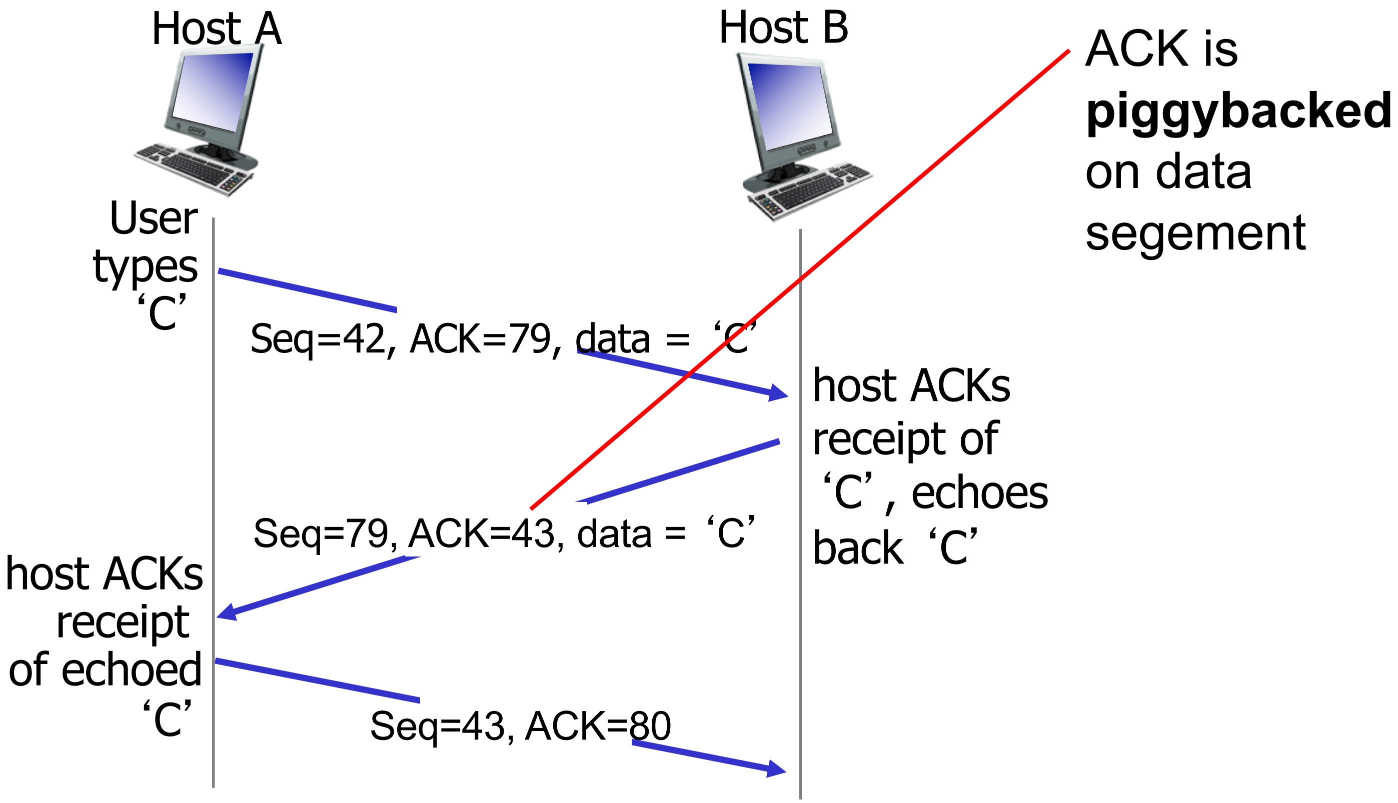
SYN(seq=x)

SYN sent

SYNACK(seq=y,ACKnum=x+1)

ACK(seq=x+1,ACKnum=y+1)

TCP seq. numbers, ACKs


Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80
ACK is **piggybacked** on data segement

simple telnet scenario

How to determine a packet loss?

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

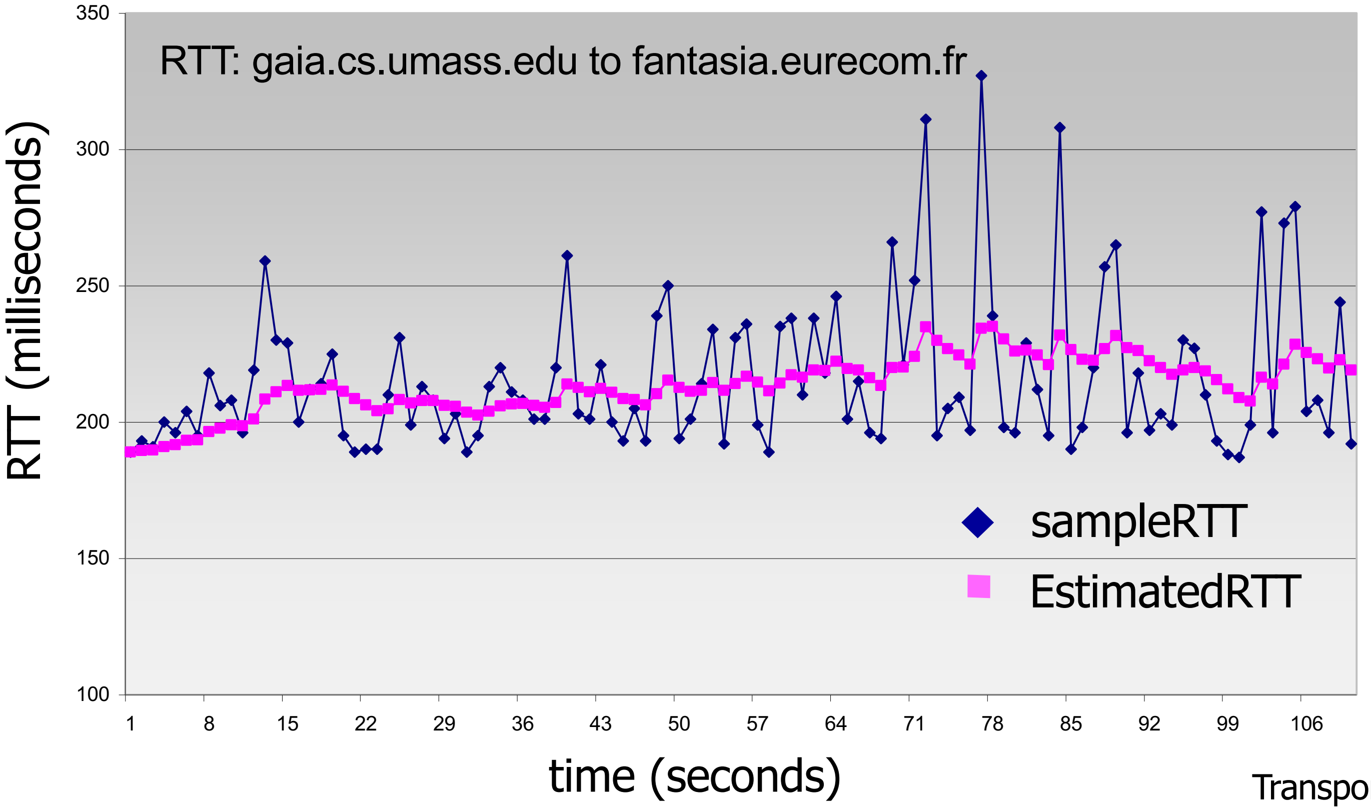
Q: how to estimate RTT?

- **`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: $\alpha = 0.125$



TCP round trip time, timeout

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

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

```
TimeoutInterval = EstimatedRTT + 4*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
  - Segment Format
  - Sequence #
  - 3 way handshake
  - Timeout value estimation
- reliable data transfer
  - FSM
  - TCP Retransmissions

TCP reliable data transfer (Sim. GBN)

- 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

TCP sender events:

data rcvd from app:

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

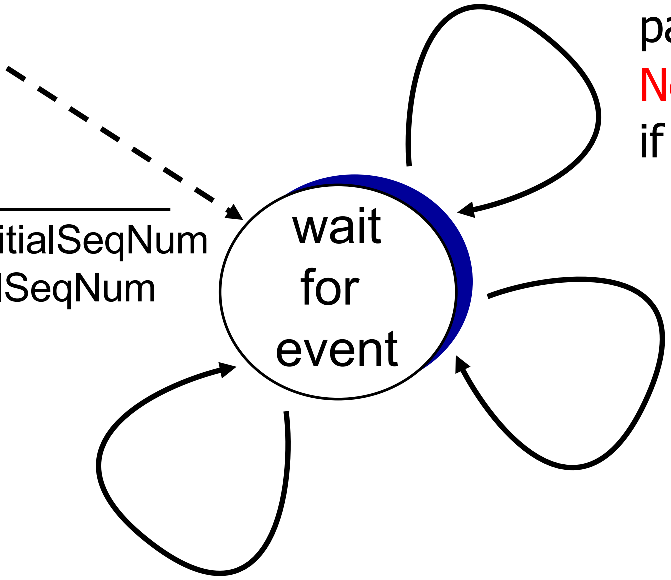
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

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

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

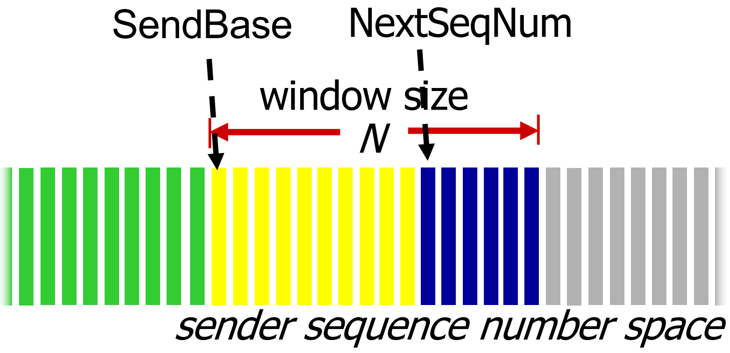
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

timeout

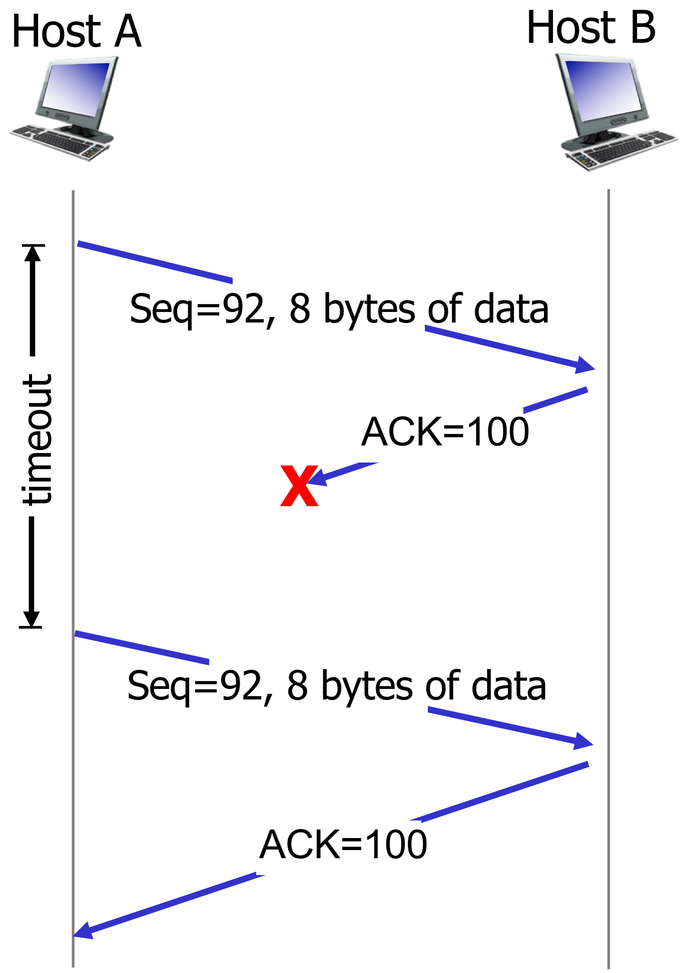
retransmit not-yet-acked segment
    with smallest seq. #
start timer

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



Host A

Host B

timeout

Seq=92, 8 bytes of data

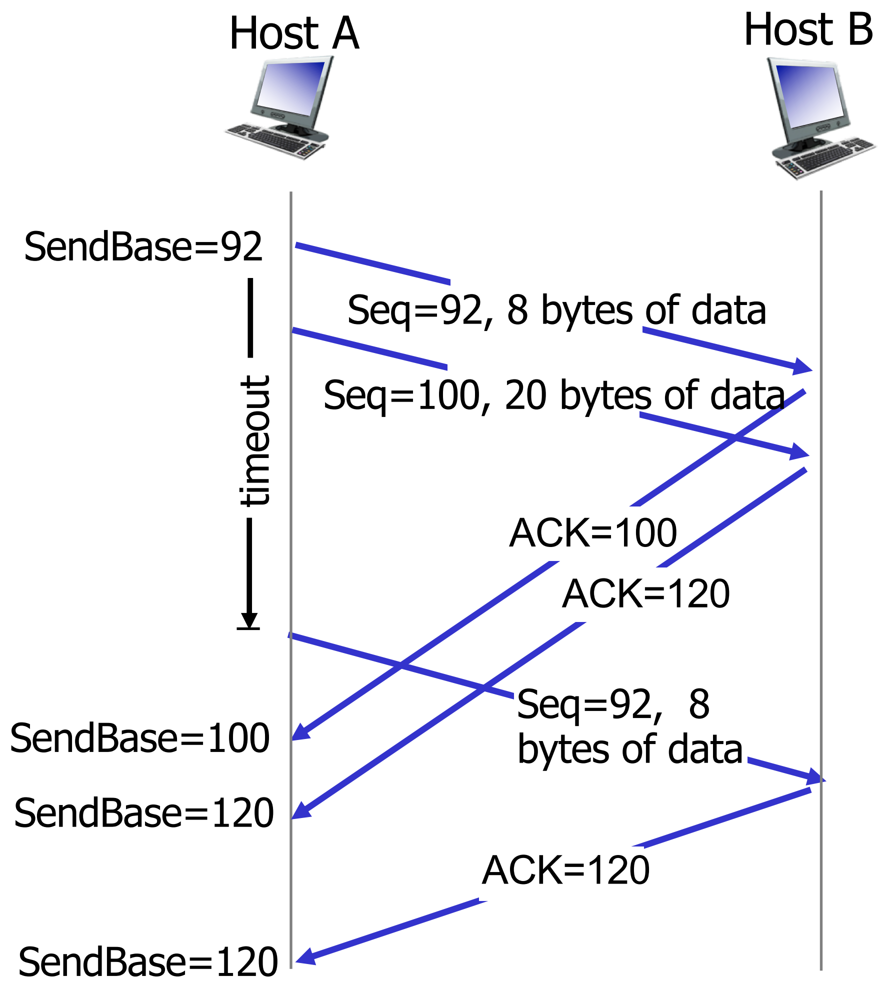
ACK=100

X

Seq=92, 8 bytes of data

ACK=100

lost ACK scenario



Host A

Host B

SendBase=92

timeout

Seq=92, 8 bytes of data

Seq=100, 20 bytes of data

ACK=100

ACK=120

SendBase=100

SendBase=120

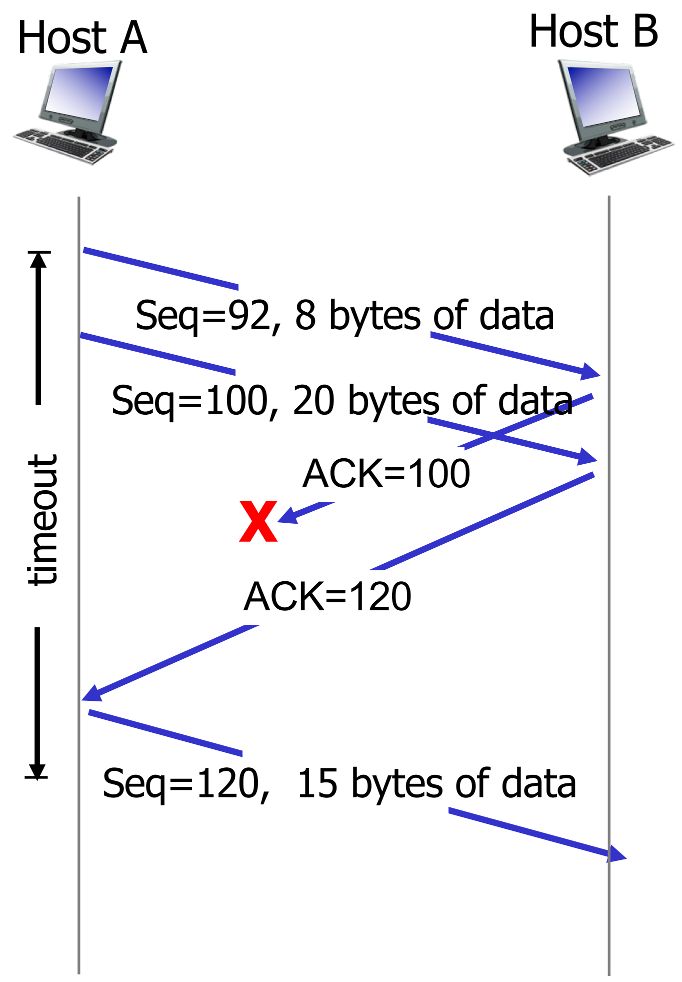
Seq=92, 8 bytes of data

ACK=120

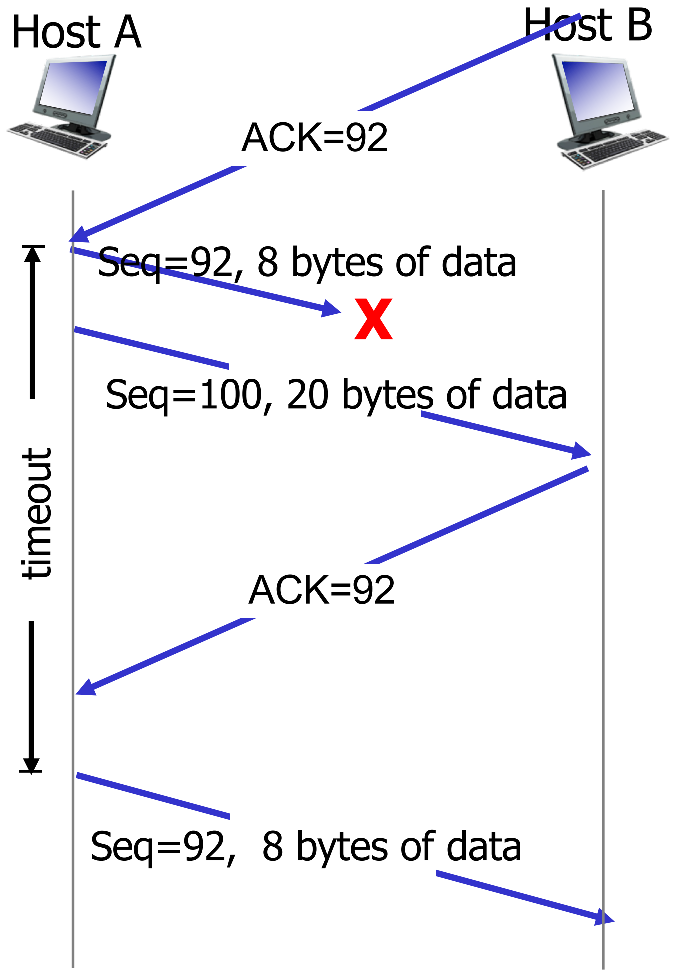
SendBase=120

premature timeout

TCP: retransmission scenarios



cumulative ACK



duplicate ACK

TCP ACK generation [RFC 1122, RFC 2581]

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

TCP fast retransmit

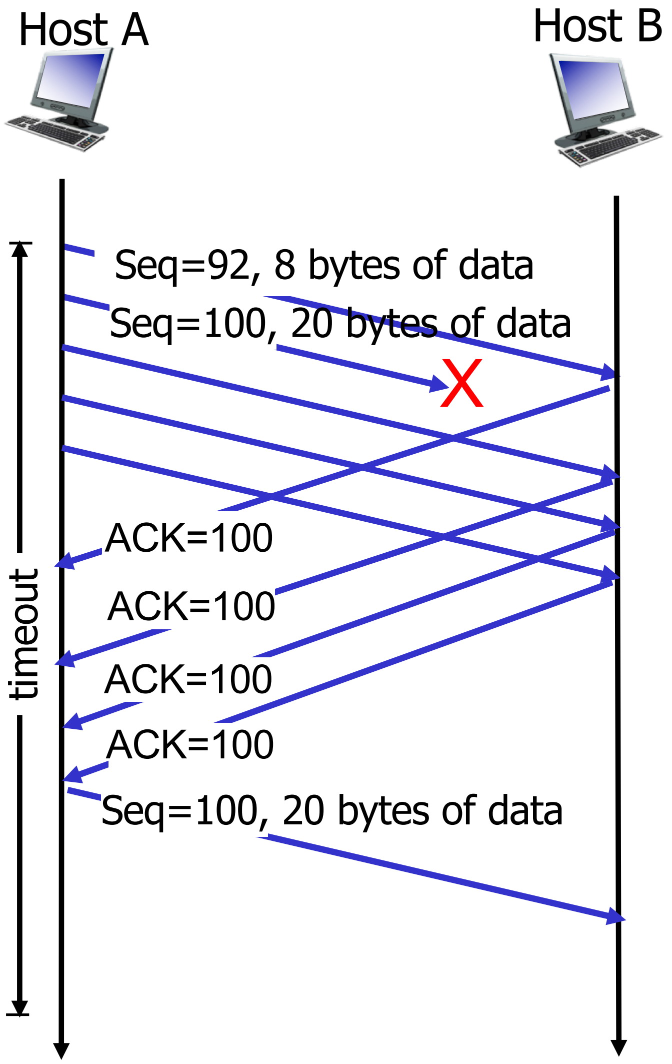
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 dup 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



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3: summary

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

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"

Questions?