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

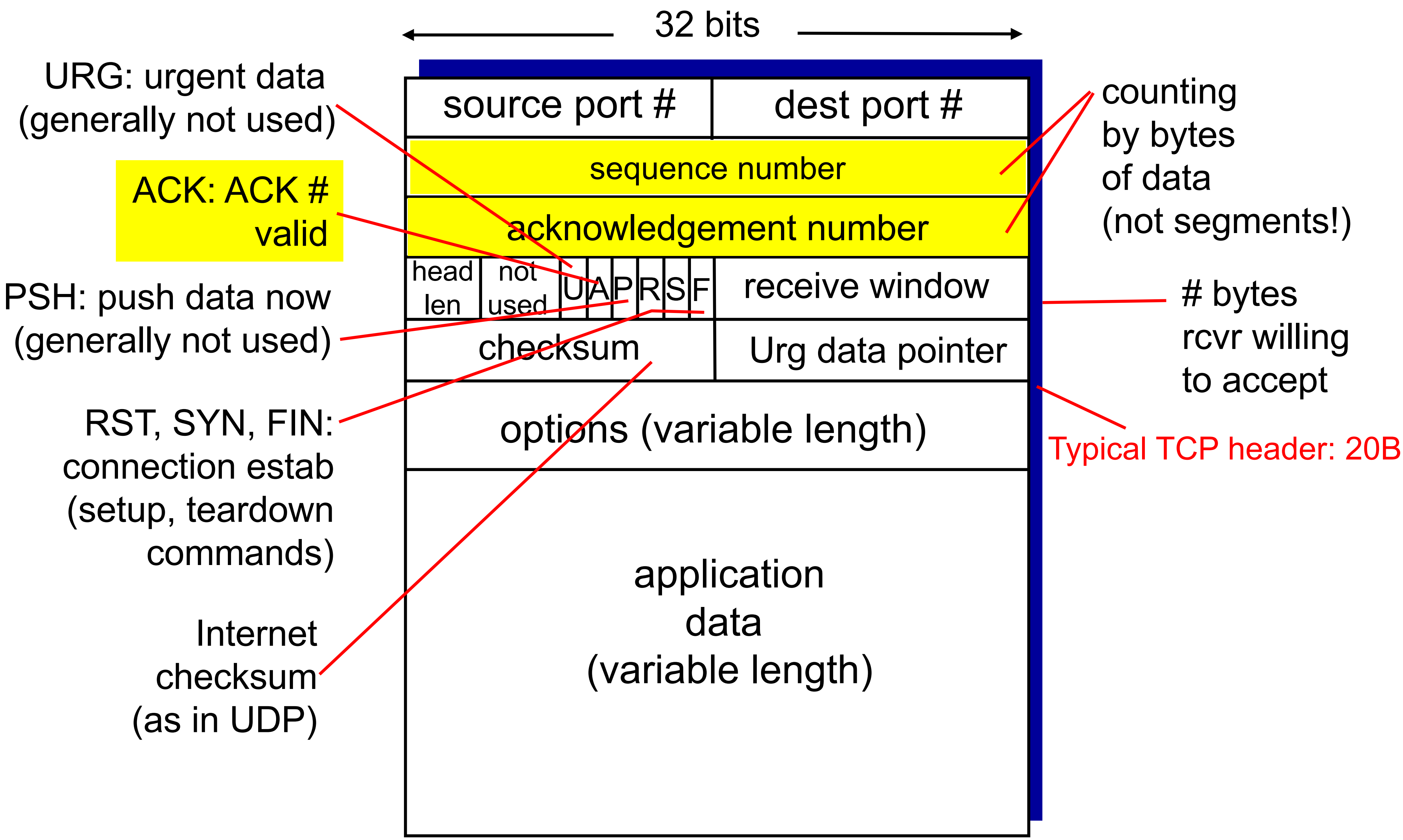
3.6 principles of congestion control

3.7 TCP congestion control

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

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

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

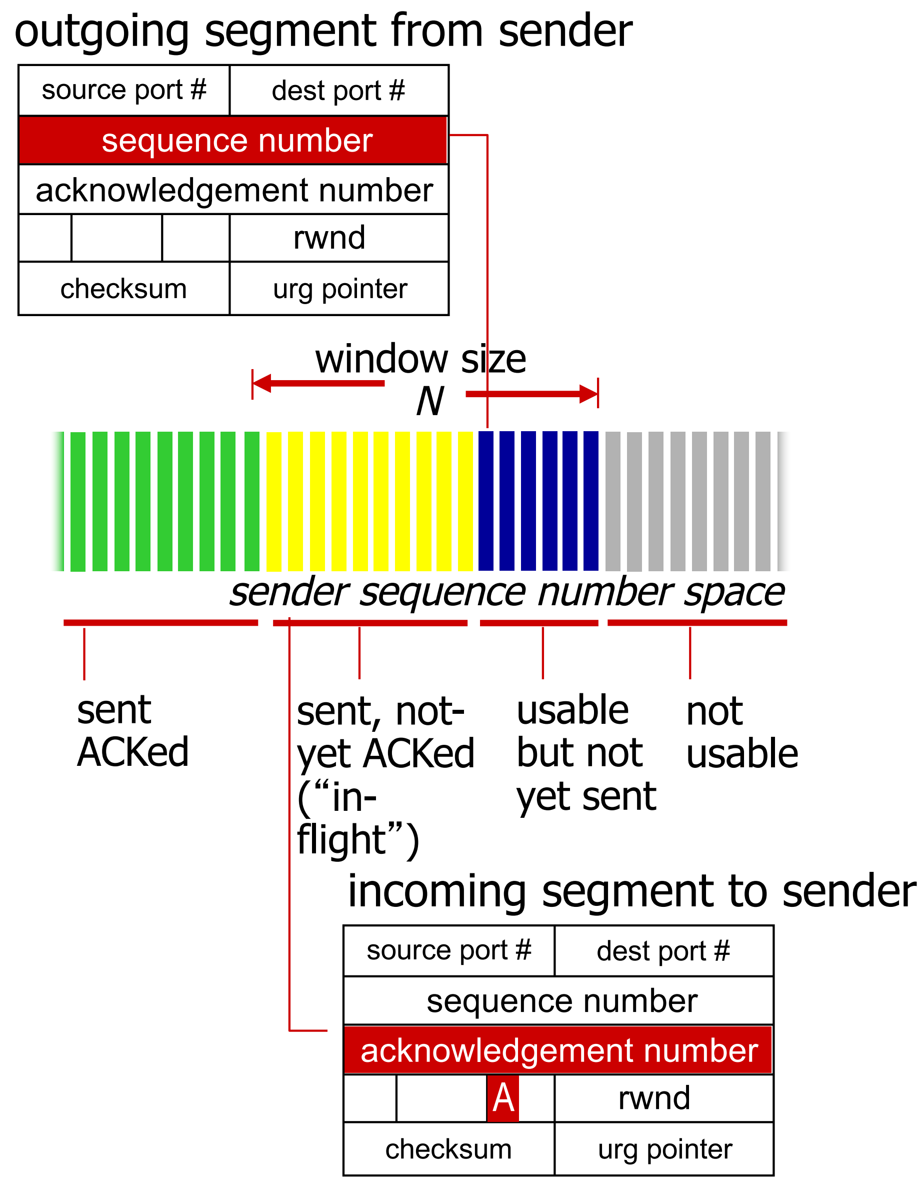
- byte stream "number" of first byte in segment's data
- 32-bit seq
- randomly chosen upon initialization (not 0!)
- one per direction

acknowledgements:

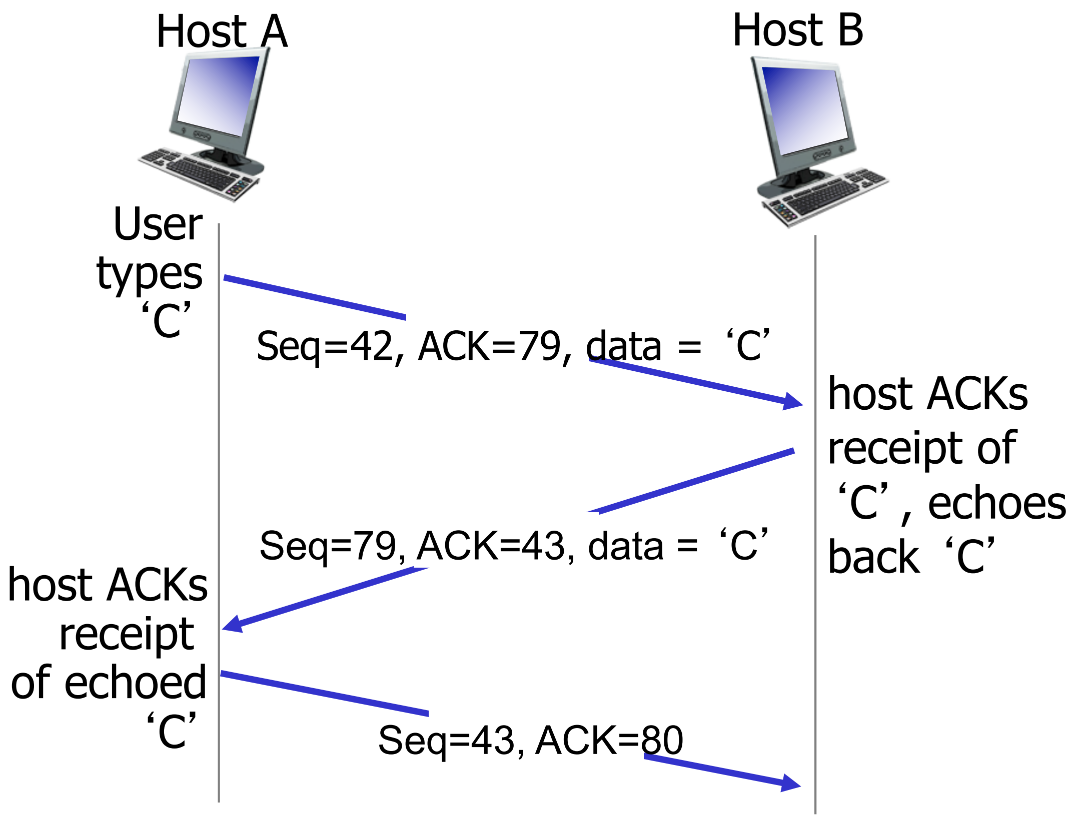
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor
- In practice: typically buffers them and wait to fill up gaps.



TCP seq. numbers, ACKs



simple telnet application scenario (RFC 854)

ACK "piggybacked" on the server-to-client data packet

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
  - window set by flow/congestion control
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

GBN or SR?

- Mostly GBN
- Elements of SR
  - selective retransmission
  - Acks trigger retransmissions
  - most receiver implementations store out-of-order packets
  - Selective ACK extension

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

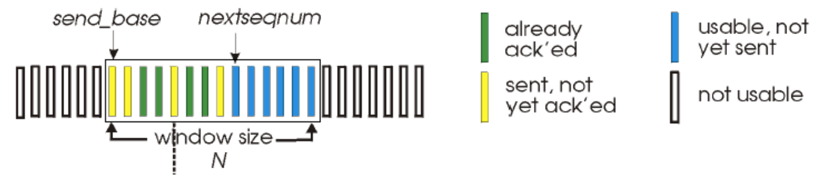
ack rcvd:

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

TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
 /*Assume: app data less than 1MSS, 1 direction, no flow/cong. control   */
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

   event: timer timeout  /* later will add dupACK */
      retransmit not-yet-acknowledged segment with smallest seqno
     /*one that caused timeout */
      restart timer

   event: ACK received, with ACK field value of y /*cumack*/
      if (y > SendBase) { /* SendBase is the oldest unacked byte. SendBase-1 last received */
         SendBase = y
         if (there are currently not-yet-acknowledged segments)
              start timer
         }
 }
```

TCP sender (simplified)



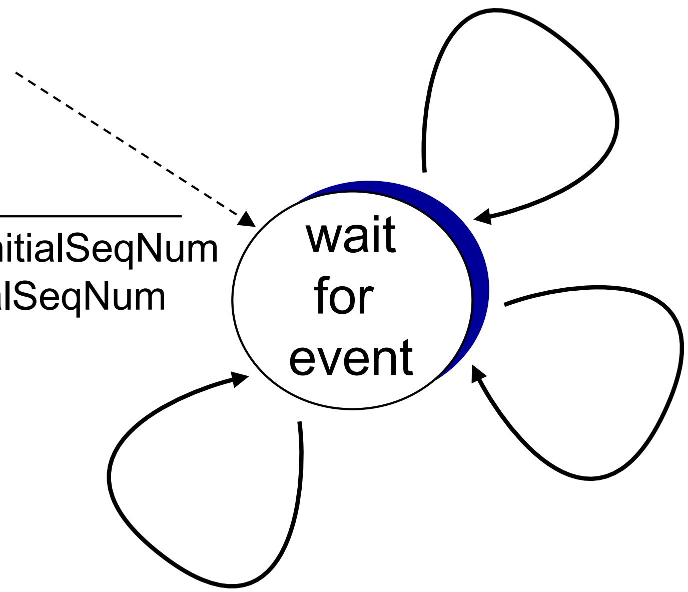
data received from application above

---

create TCP segment with seq# NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
  start timer /*one timer,
      for oldest un-ACKed packet */

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



Timeout /* later will add dupACK */

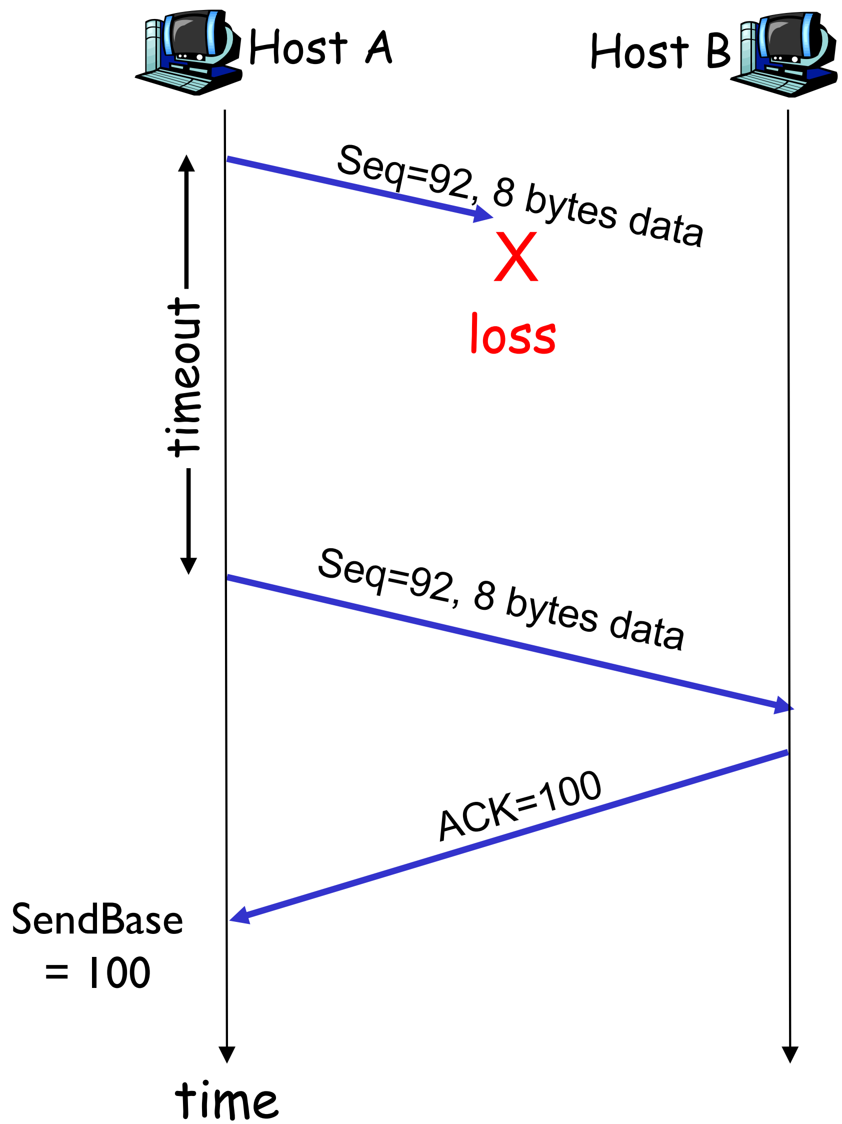
---

retransmit not-yet-acked segment
      with smallest seq. #
/*Just the one that caused the timeout*/
/*NOT the entire window! */
restart timer

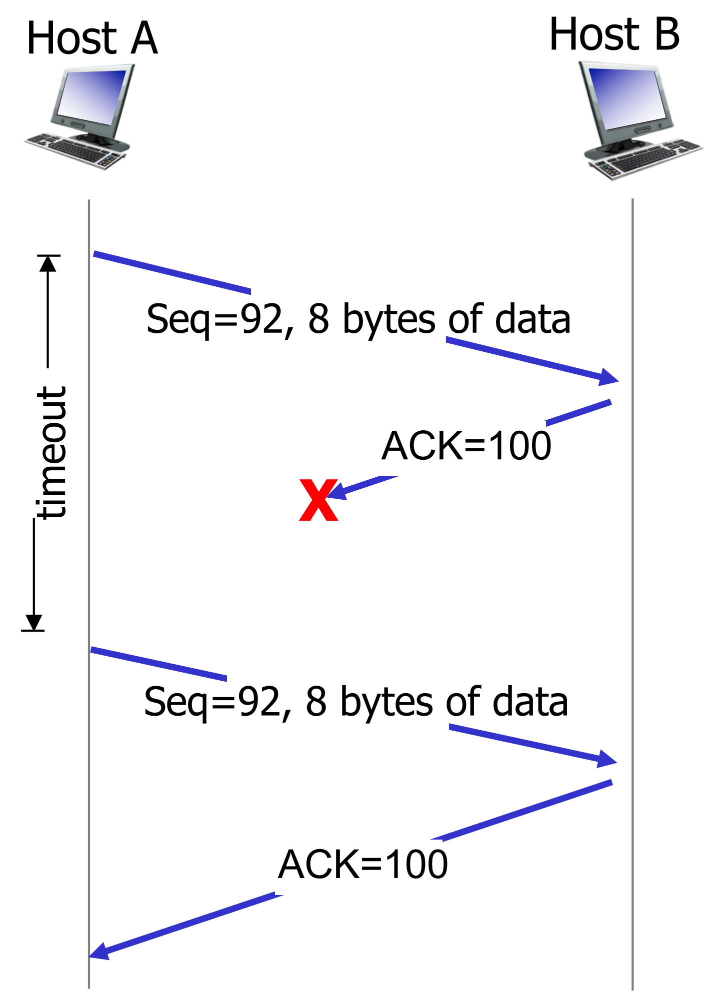
ACK received, with ACK field value y /*cumACK */

```
if (y > SendBase) {{
   SendBase = y  /*update what is known to be ACKed */
   /* SendBase–1: last cumulatively ACKed byte. SendBase is the oldest unacked byte */
   if (there are still currently not-yet-acked segments)
       start timer
     else stop timer /*empty sender window*/
  }
```

TCP: retransmission scenarios

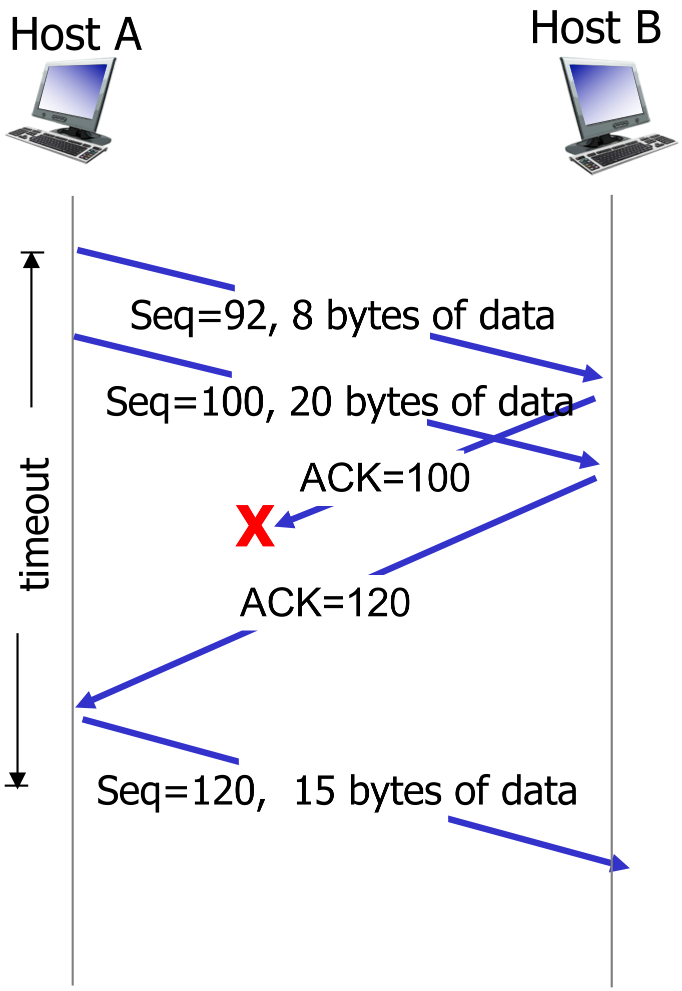


lost data scenario

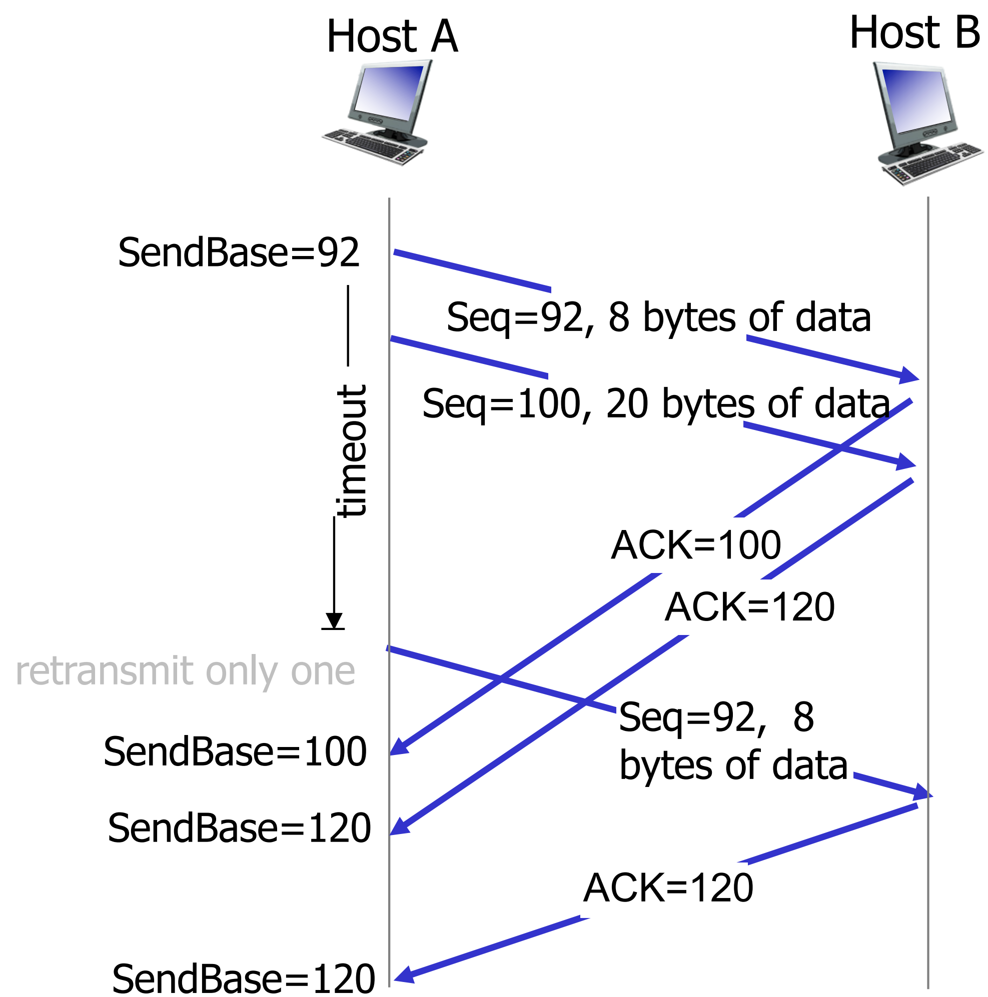


lost ACK scenario

TCP: retransmission scenarios



cumulative ACK



premature timeout

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

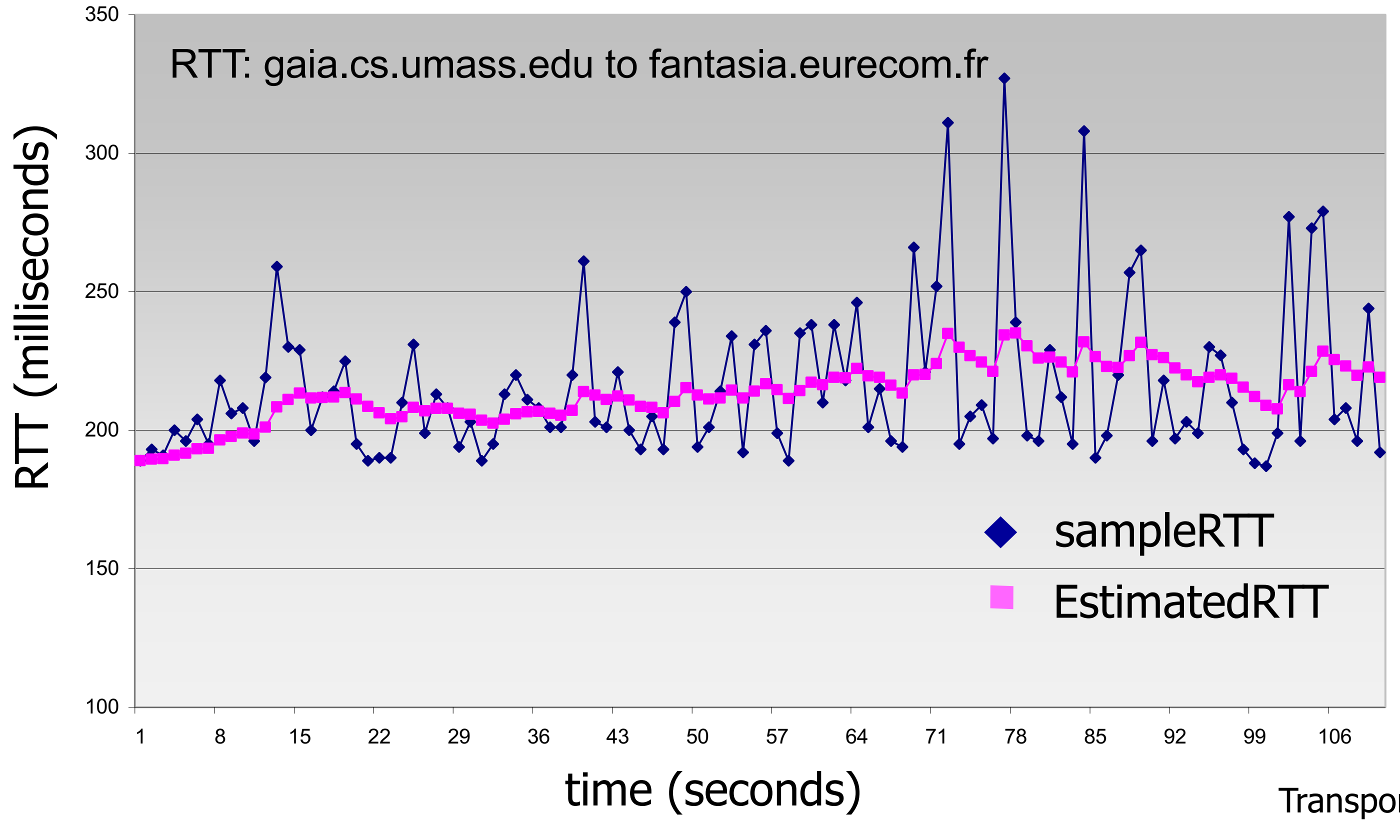
Q: how to estimate RTT?

- **`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
  - Why? https://en.wikipedia.org/wiki/Exponential_smoothing
- 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"

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Fine tuning of Timeout

- Normal operation

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

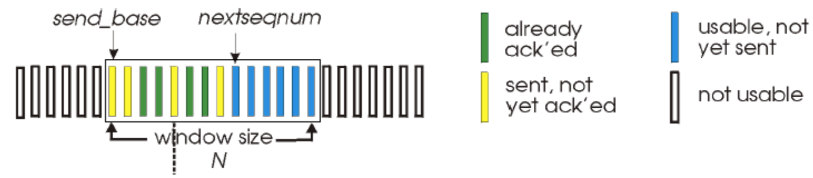
- After a retransmission:
  - Double the timeout →grows exponentially with repeated retransmissions
  - Rationale: long timeout→ indication of congestion → backoff (limited form of congestion control)

- When the timer is restarted

```
TimeoutInterval = EstimatedRTT + 4*DevRTT,
```

  - using most recent estimates

TCP sender - revisited



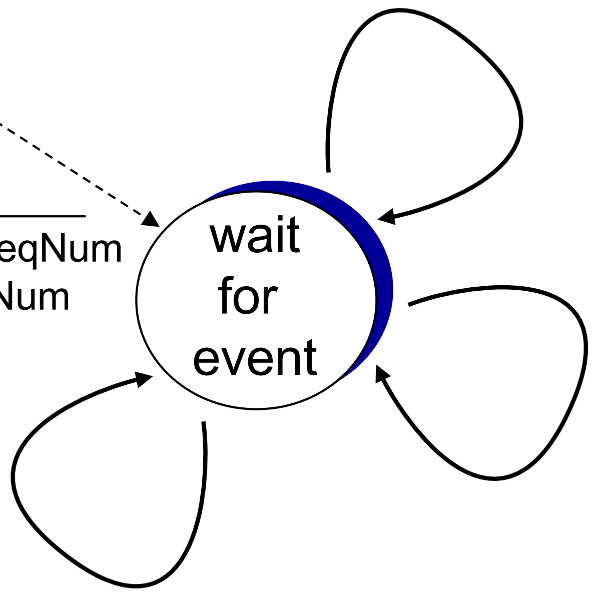
data received from application above

---

create TCP segment with seq# NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer /*one timer,
        for oldest un-ACKed packet */

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



Timeout /* later will add dupACK */

---

retransmit not-yet-acked segment
    with smallest seq. #
/*Just the one that caused the timeout*/
/*NOT the entire window! */
restart timer

ACK received, with ACK field value y /*cumACK */

```
if (y > SendBase) {{
   SendBase = y  /*update what is known to be ACKed */
   /* SendBase–1: last cumulatively ACKed byte. SendBase is the oldest unacked byte */
   if (there are still currently not-yet-acked segments)
       start timer
     else stop timer /*empty sender window*/
   }
```

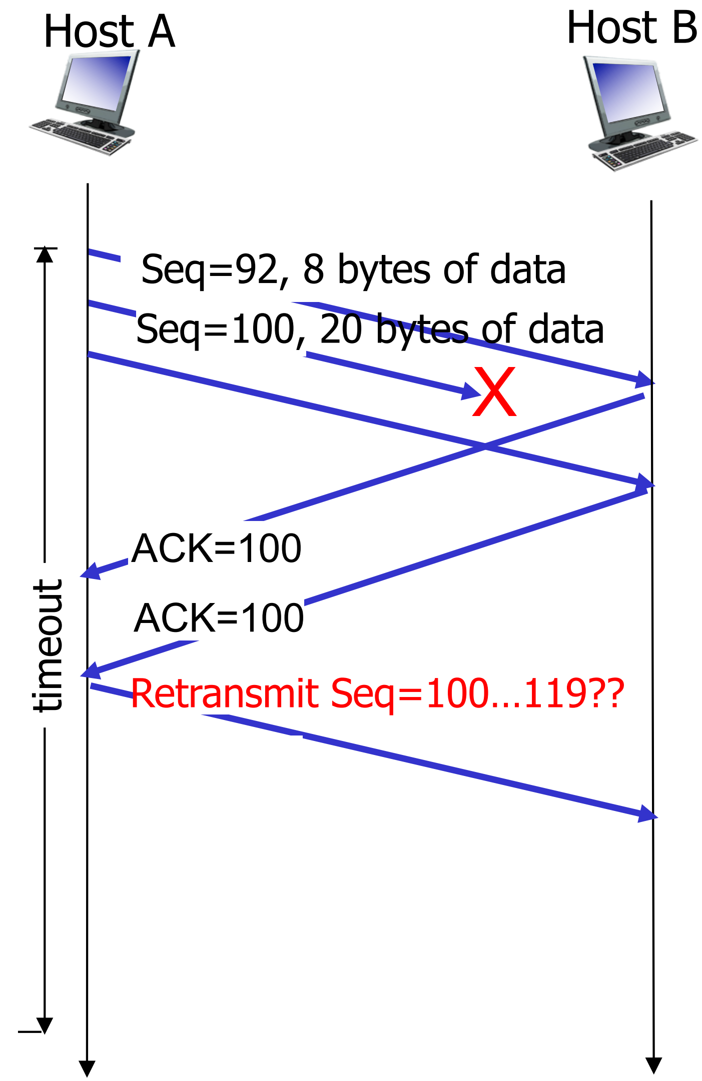
TCP ACK generation(receiver side) [RFC 1122, RFC 2581]

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 Most implementations store the segment
arrival of segment that partially or completely fills gap	immediately send ACK, provided that segment starts at lower end of gap

These are the differences from classical go-back-N

A duplicate ACK indicates loss (sender side)

- Motivation: time-out period often 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.



Difference from classical Go-back-N
(Go-back-N uses only Timeout to detect loss)

TCP fast retransmit

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend one unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

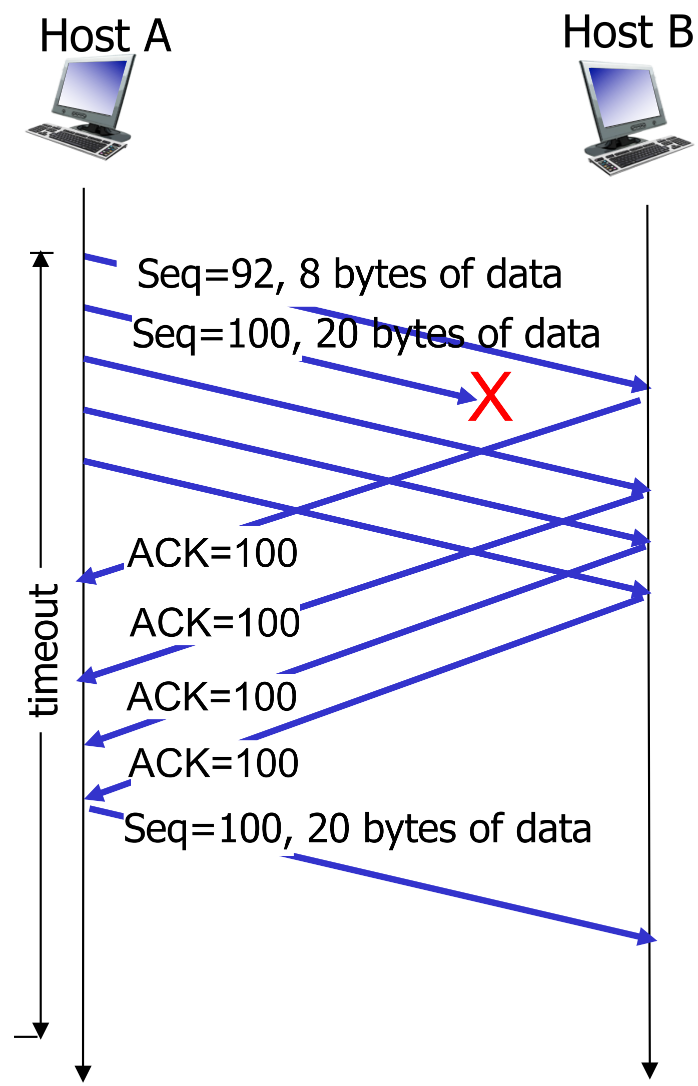
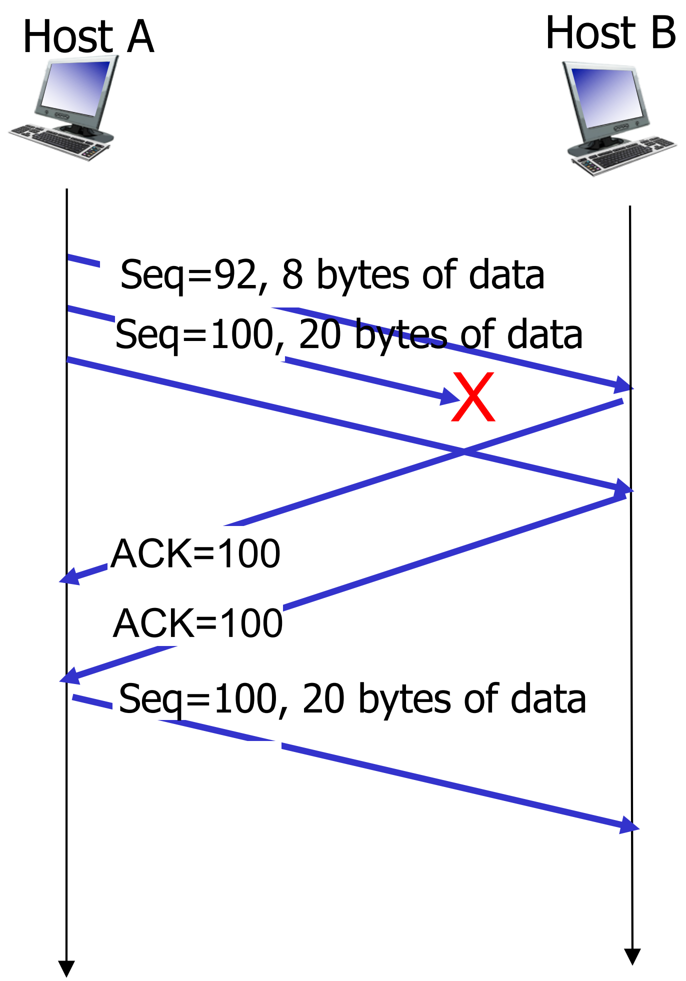


Fig. 3.37: fast retransmit after sender receipt of triple duplicate ACK

Fast retransmit algorithm (at sender):

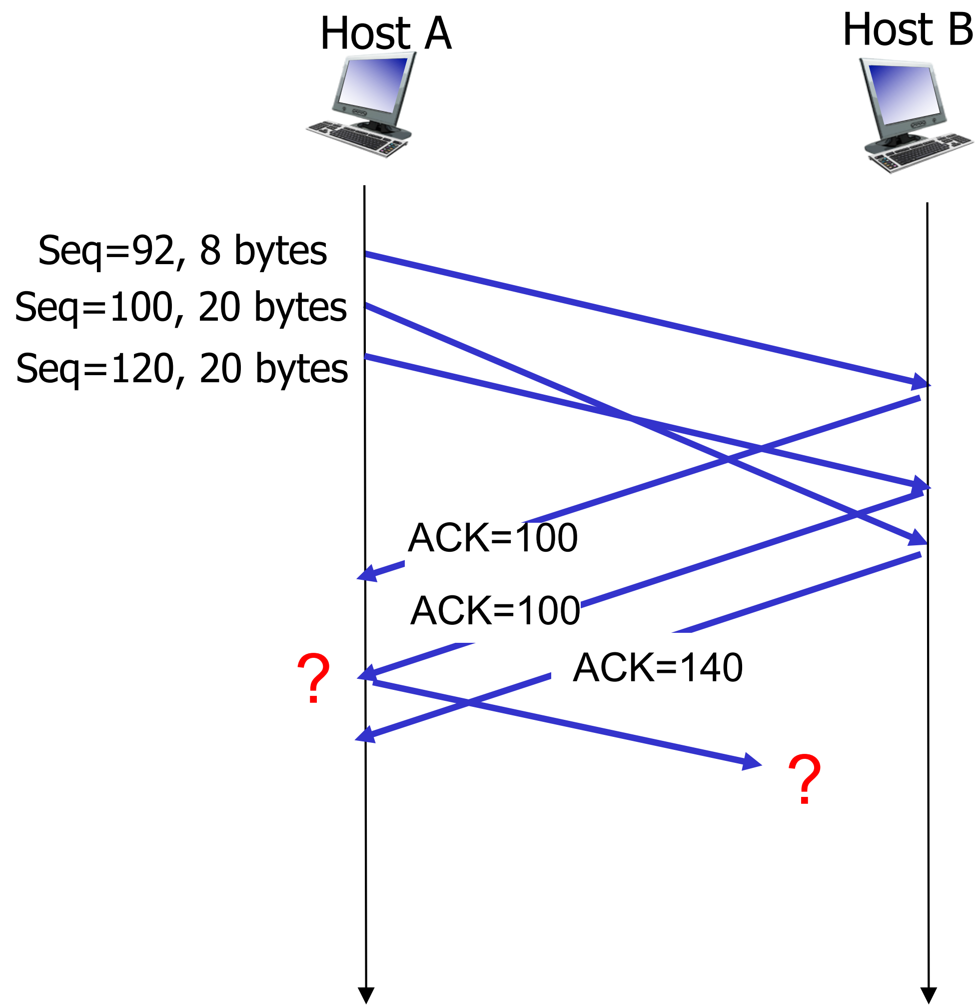
```
event: ACK received, with ACK field value of y
           if (y > SendBase) { /* first ACK for not-yet-ACKed segment */
               SendBase = y
               if (there are currently not-yet-acknowledged segments)
                   start timer
             }
           else { /* duplicate ACK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

Why 3 dup ACKs instead of 2?



Duplicate ACKs due to loss

- We want to recover fast
- Likely to receive many dup ACKs



Duplicate ACKs due to reordering

- We want to not react prematurely
- Likely to receive many dup ACKs

TCP reliable data transfer - Summary

- TCP creates rdt service on top of unreliable IP
  - pipelined segments [window set by flow & congestion control]
  - cumulative acks
  - single retransmission timer [setting of timeouts based on RTT]
- Retransmissions
  - triggered by:
    - timeout events
    - dup ACKs
    - three duplicate acks → "fast retransmit"
  - at most one packet at a time (not entire window)
- Mostly GBN with some SR elements:
  - selective retransmission
  - most receiver implementations store out-of-order packets
  - selective ACK extension [not covered here]
- Other differences from idealized protocols
  - seq# refer to bytes,
  - ACKs are piggy-backed on data
  - delayed ACKs