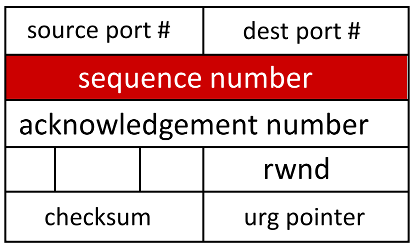
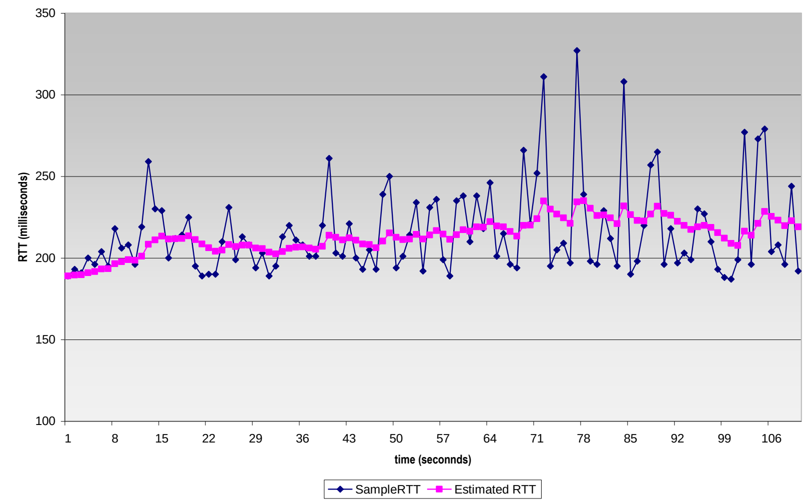
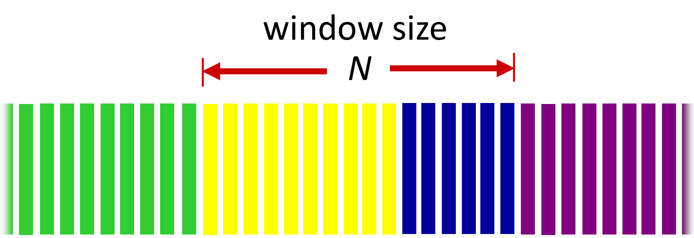
Transmission Control Protocol (TCP)







Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley

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

Transmission Control Protocol (TCP)

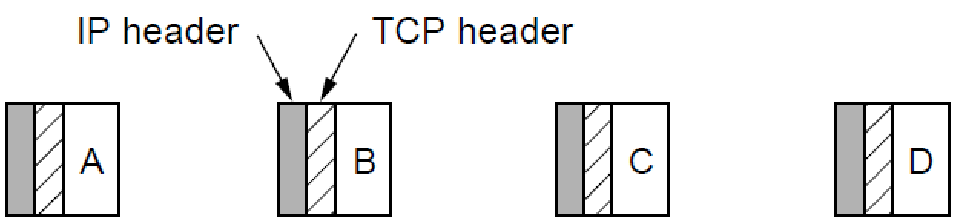
- Stream of bytes
  - Send and receive streams, not messages
- Reliable, in-order delivery
  - Checksums to detect corrupted data
  - Sequence numbers to detect losses and reorder
  - ACKs and retransmission for reliability
- Connection-oriented
  - Explicit setup and teardown of connections
  - Full duplex, two streams one in each direction
- Flow control
  - Prevent overrunning receiver's buffer

Transmission Control Protocol (TCP)

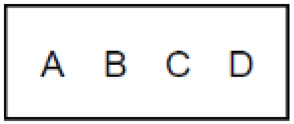
- Congestion control
  - Adapt for the greater good
- History:
  - RFC 793, TCP formally defined, September 1981
  - RFC 1122, clarification and bug fixes
  - RFC 1323, high performance extensions
  - RFC 2018, selective acknowledgements
  - RFC 2581, congestion control
  - RFC 2873, quality of service
  - RFC 2988, improved retransmission timers
  - RFC 3168, congestion notification
  - …
  - RFC 4614, guide to TCP RFCs

TCP service model

- Uses port number abstraction, same as UDP
- Demultiplexing key:
  - <source IP, source port, dest. IP, dest. port>
- Byte steam, no message boundaries
  - No way to know what size chunks given to SEND when other side does RECEIVE

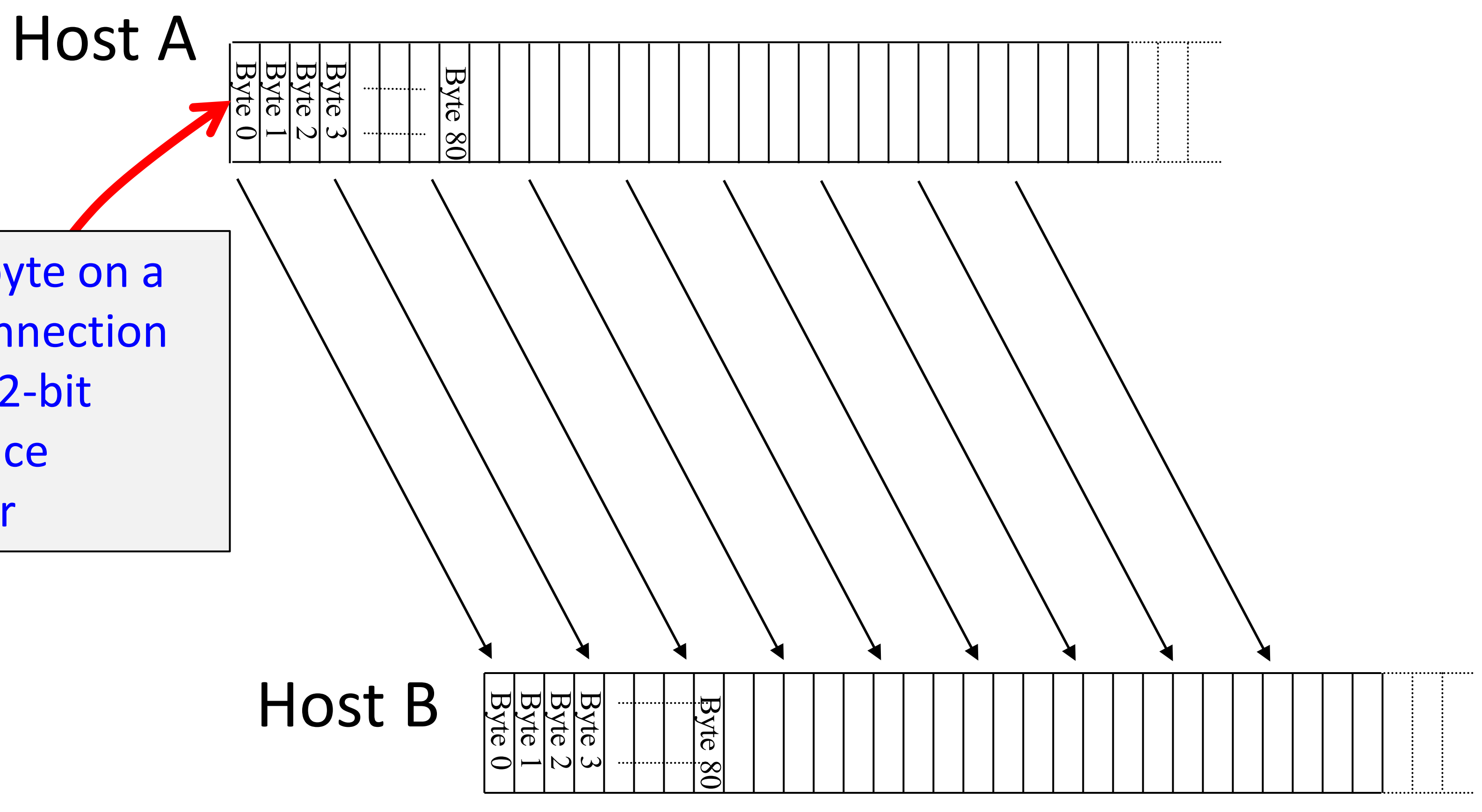


Four 512-byte segments sent as separate IP datagrams.



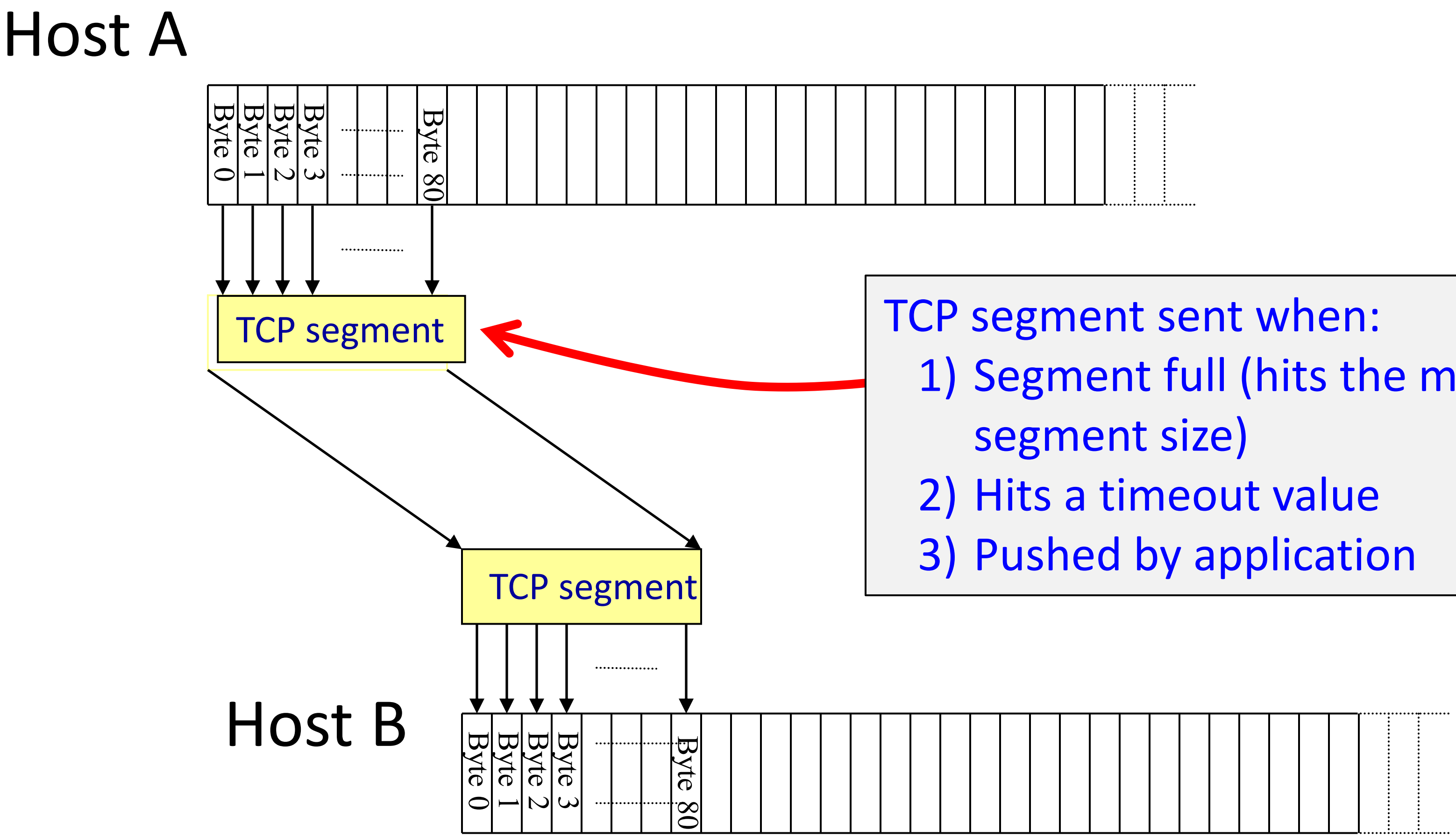
2048 bytes of data delivery to application in single READ call

TCP "stream of bytes" service



Every byte on a TCP connection has a 32-bit sequence number

Emulating a byte stream



TCP segment sent when:

1) Segment full (hits the max segment size)
2) Hits a timeout value
3) Pushed by application

TCP buffering

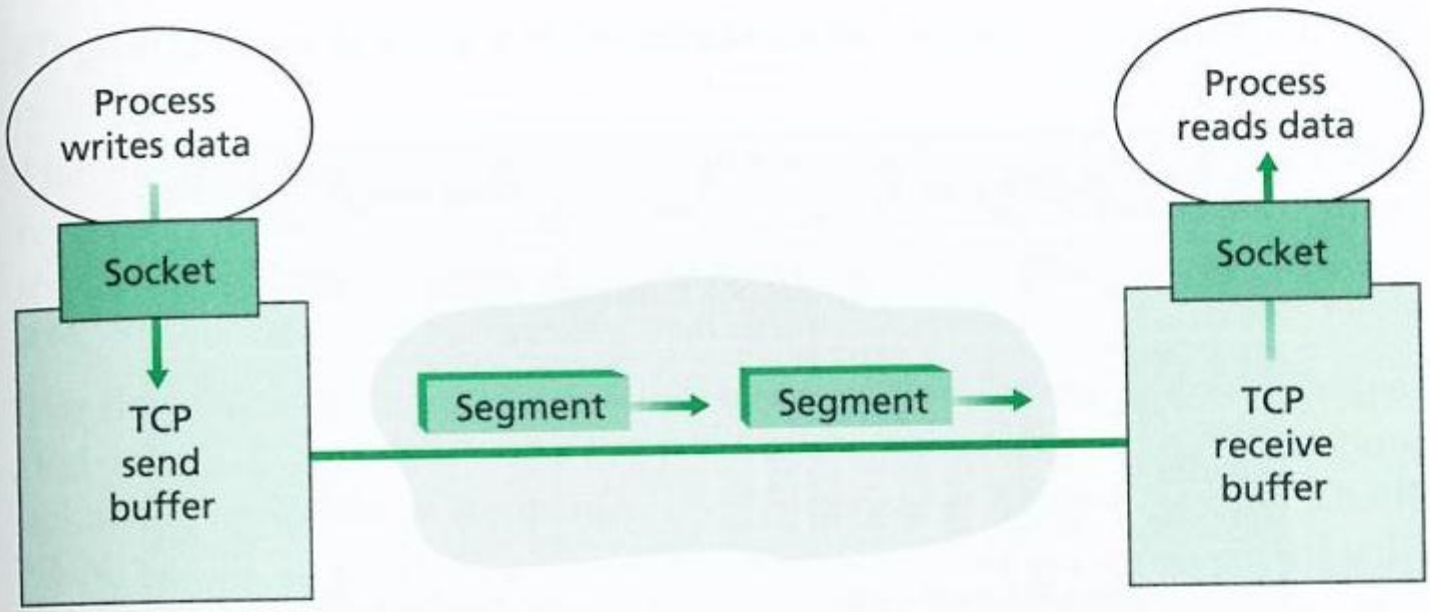
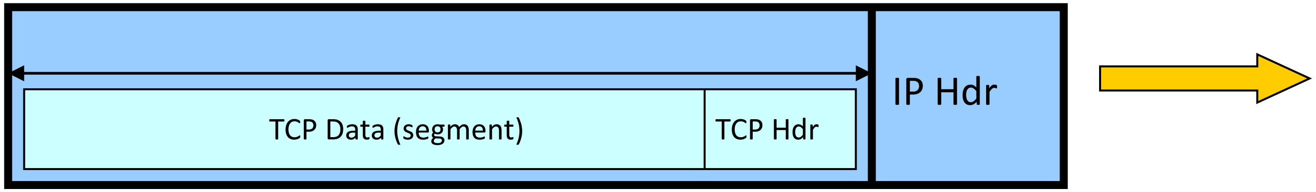


Figure 3.28 ♦ TCP send and receive buffers

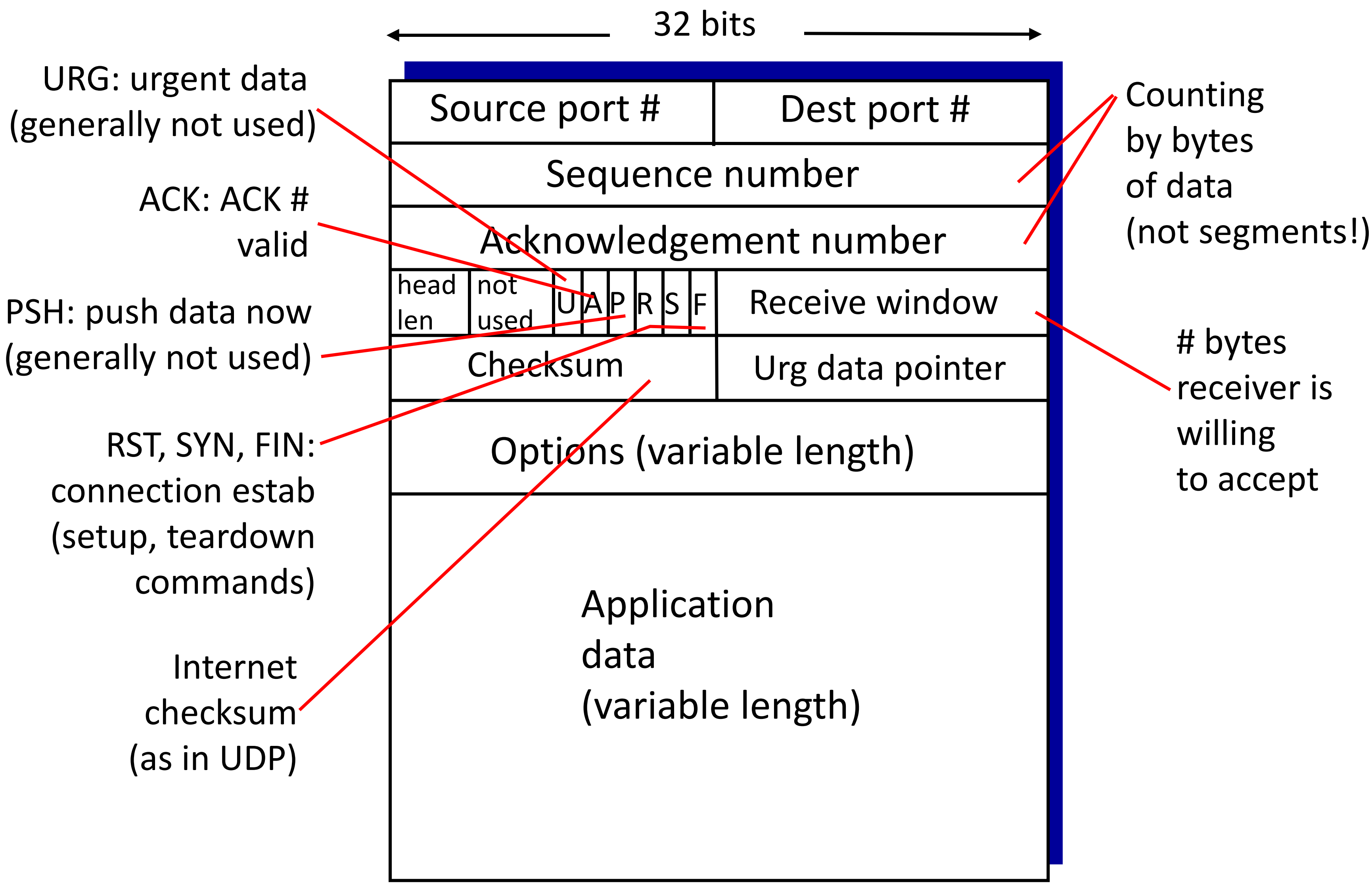
- Data sent by socket gets put in TCP send buffer
  - RFC 793: "send that data in segments at its own convenience"
  - Send side can request PUSH by sitting bit in TCP header

Determining MSS

- Maximum Segment Size (MSS)
  - Default size:
    - Nodes must support min IP MTU of 576 bytes
    - 536 bytes = 576 – 20 (IP header) – 20 (TCP header)
    - Usually doesn't fragment, unless IP/TCP options used
  - Nodes specify MSS during connection setup
    - Done via MSS option field of TCP segment header
    - Could be different in each direction



TCP segment structure



TCP sequence numbers, ACKs

<u>Sequence numbers:</u>

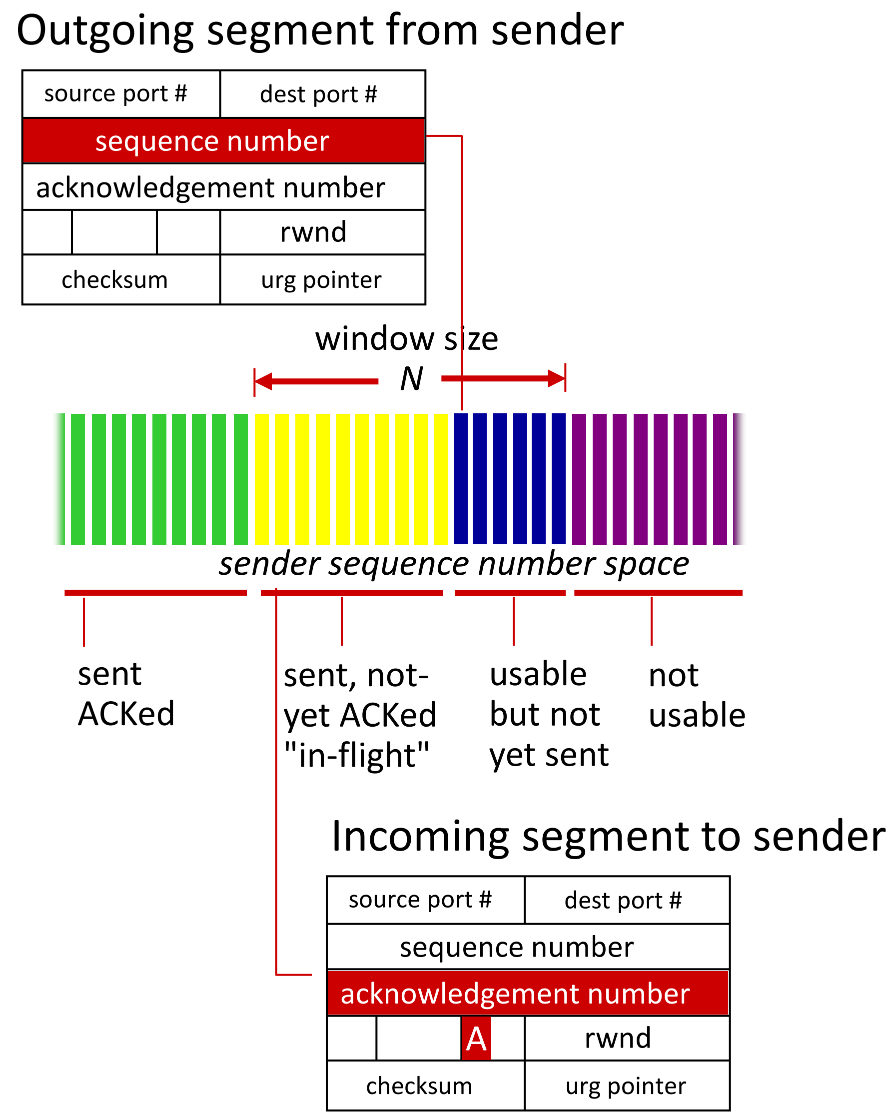
- Byte stream number of *first byte* in segment's data

<u>Acknowledgements:</u>

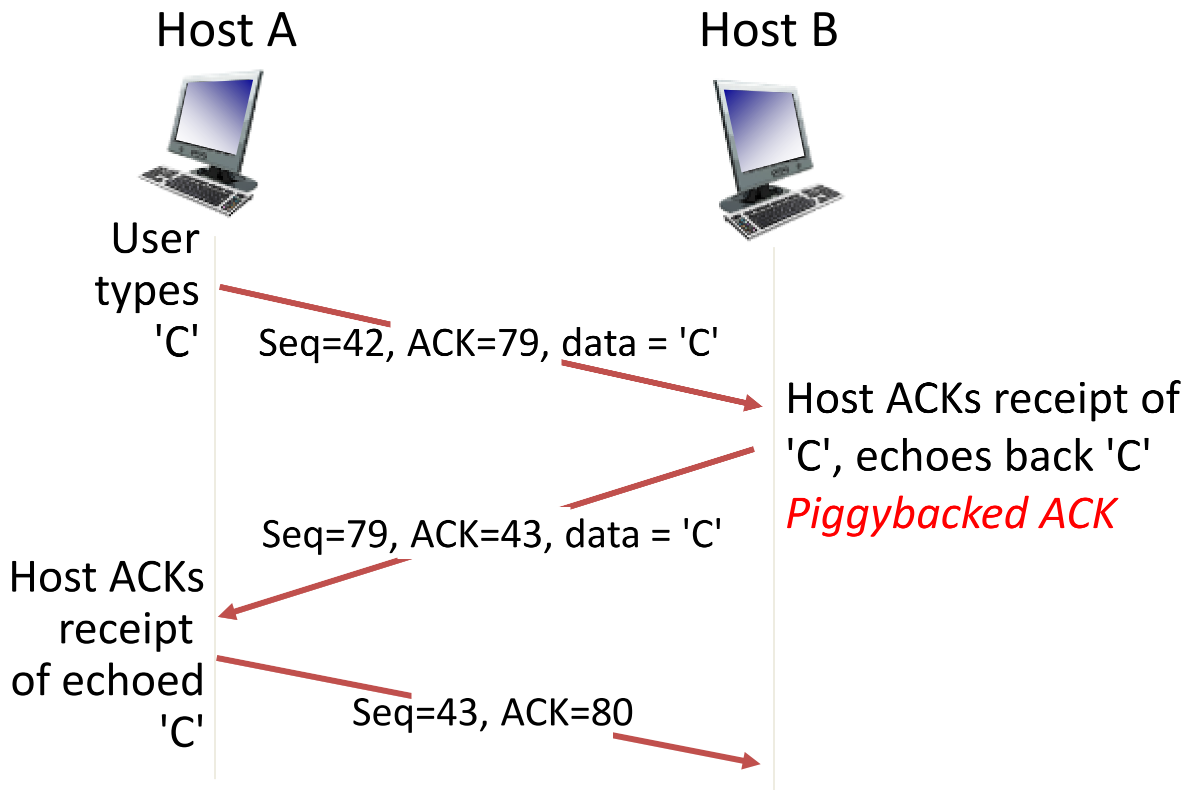
- Sequence # of *next byte* expected from other side
- Cumulative ACK
  - If segment(s) out of order, ACK last next expect in-order byte

<u>**Q:**</u> How does receiver handle out-of-order segments?

<u>**A:**</u> TCP spec doesn't say, up to implementation



TCP sequence numbers, ACKs



Simple telnet scenario

TCP RTT, timeout

Q: How to set TCP timeout value?

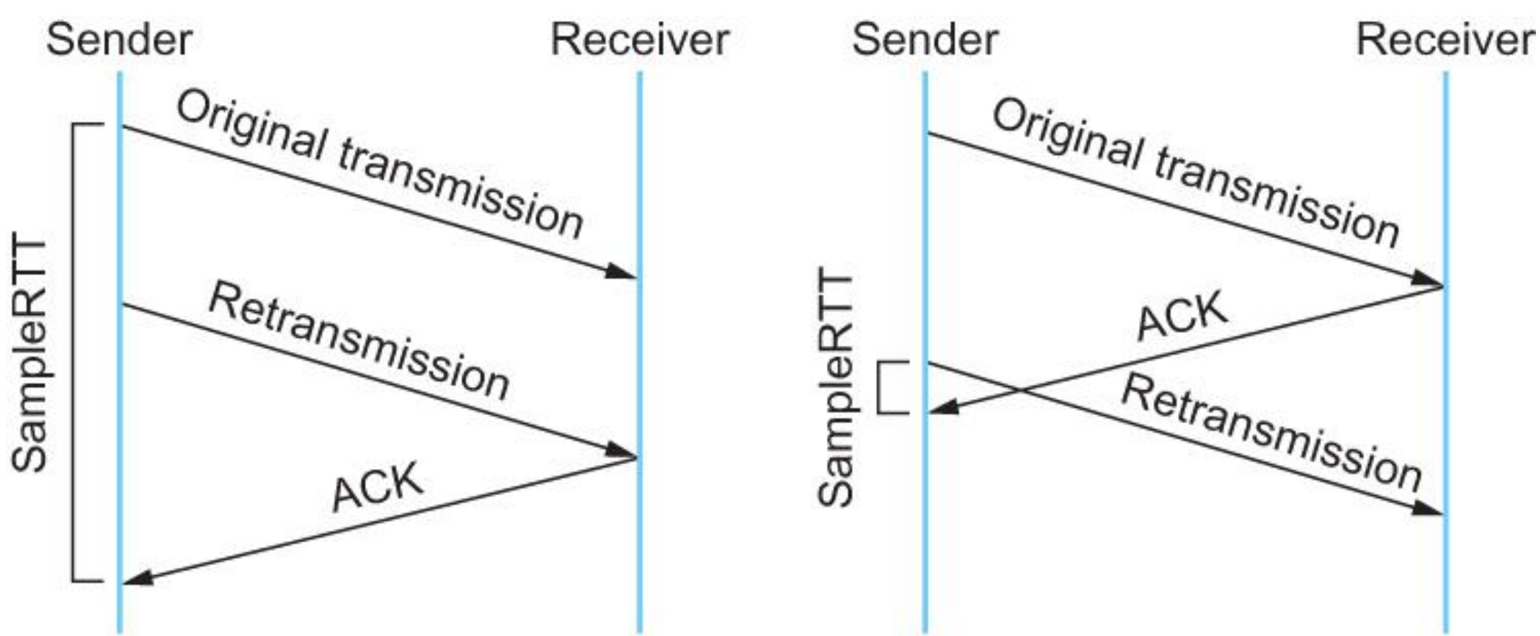
- Longer than RTT
  - But RTT varies
- *Too short:* premature timeout, unnecessary retransmissions
- *Too long:* slow reaction to segment loss

Q: How to estimate RTT?

- **SampleRTT**: Measured time from segment transmission until ACK receipt
  - Ignore retransmissions
  - Will vary, want estimated RTT smoother
  - Average several *recent* measurements, not just current sample RTT

Adaptive timeout

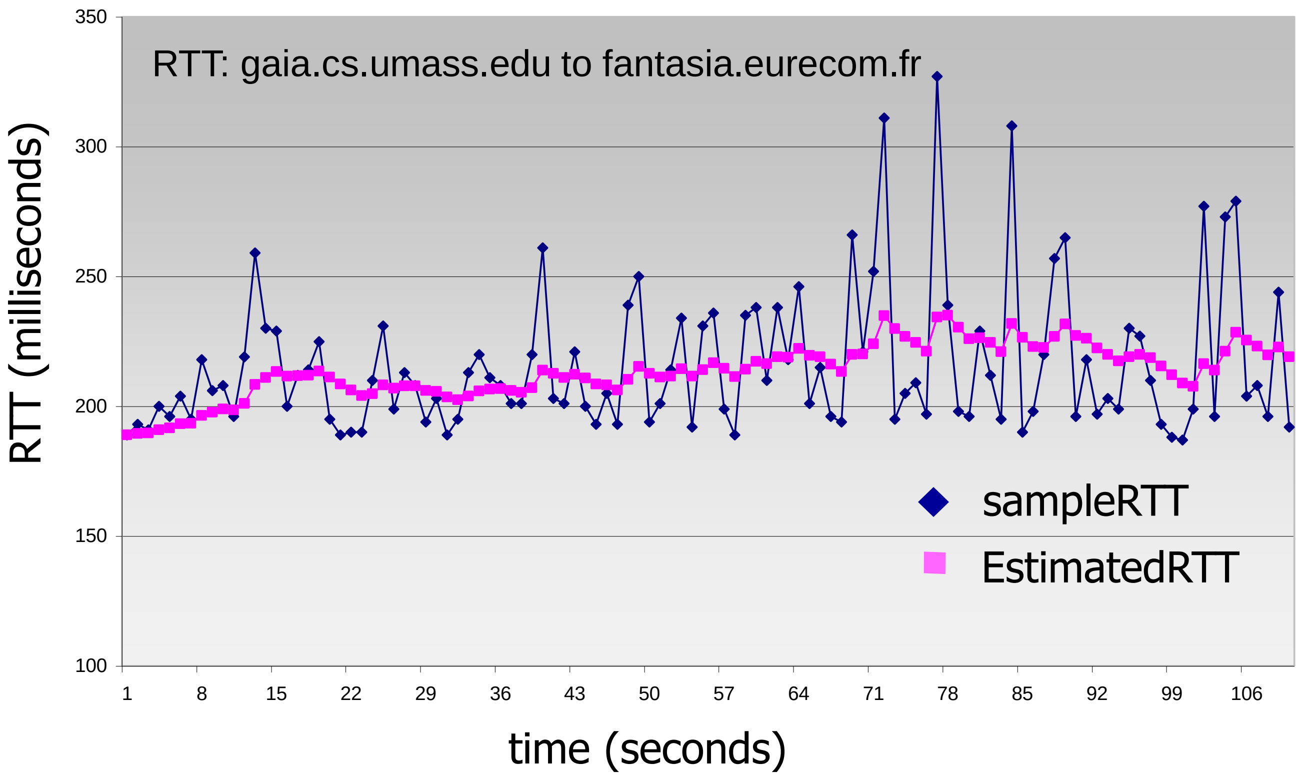
- Don't update SampleRTT on retransmitted frames
  - Karn/Partidge algorithm, 1987
  - Ignore RTT's of packet that were retransmitted
  - Double timeout value on retransmission
    - Exponential backoff



TCP RTT, timeout

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$



TCP RTT, 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

TCP reliable data transfer

- TCP creates reliable 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
- Ignore congestion control

TCP sender events

Data received from app:

- Create segment with sequence #
- Sequence # 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

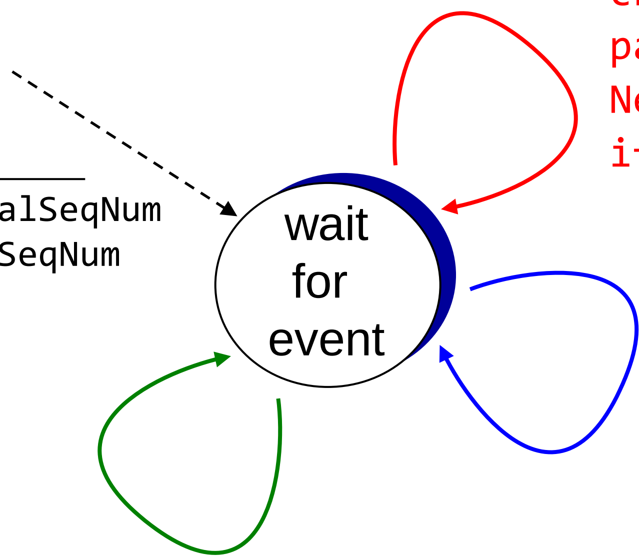
ACK received:

- 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 (send it)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
   start timer
```

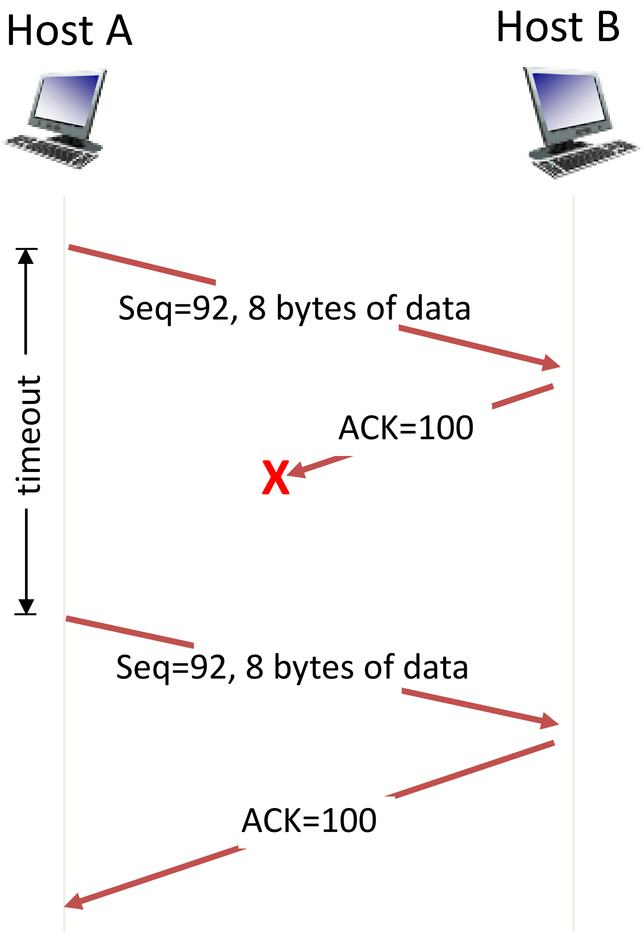
timeout

```
retransmit not-yet-ACKed
segment with smallest seq #
start timer
```

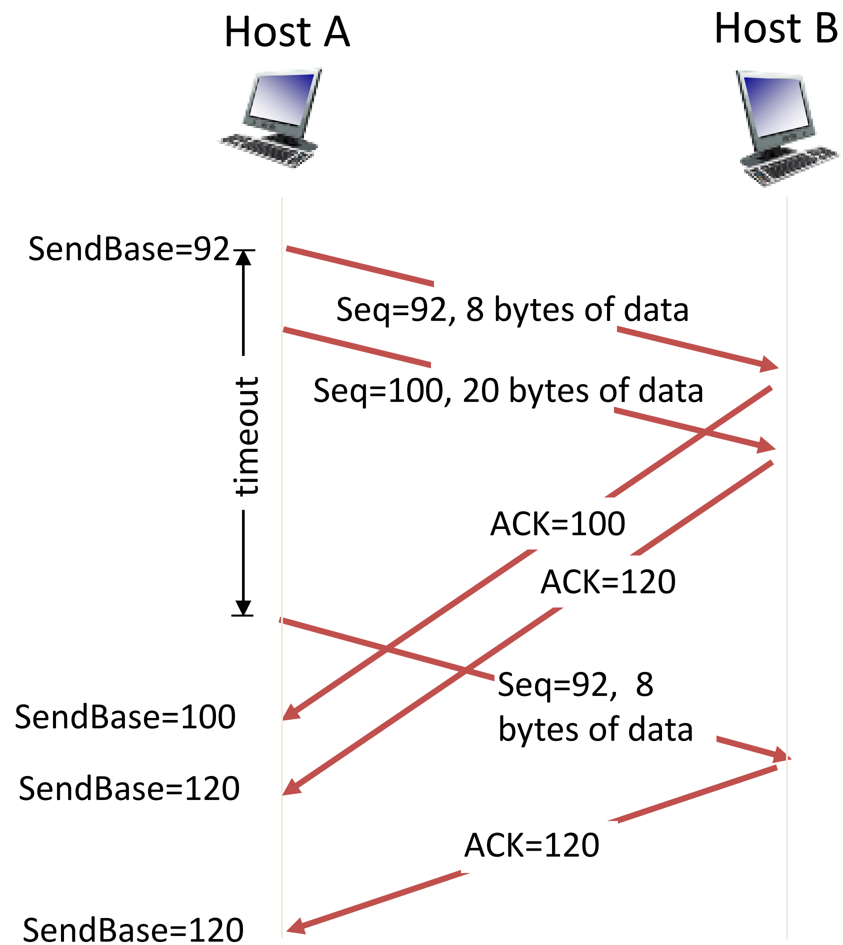
ACK received, with ACK field value y

```
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

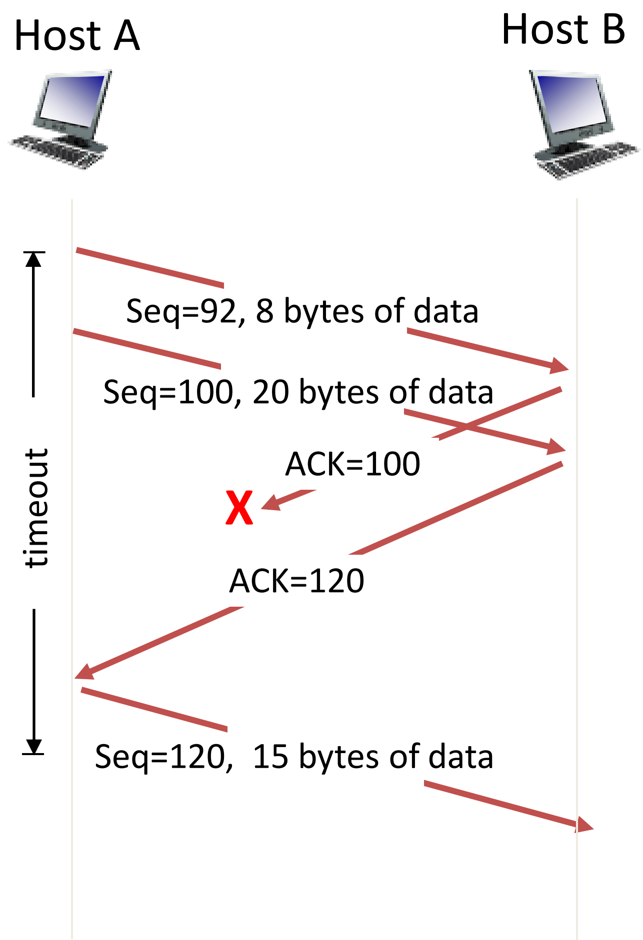


Lost ACK scenario



Premature timeout

TCP: retransmission scenarios



Cumulative 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	Immediately send ACK, provided that segment starts at lower end of gap

TCP fast transmit

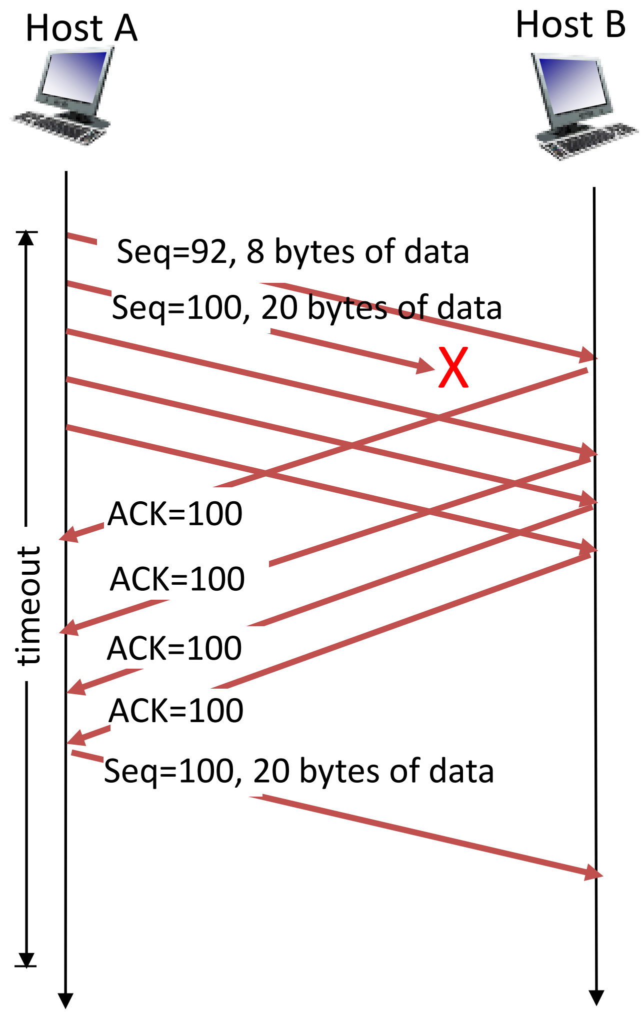
- Time-out period often relatively long:
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs

TCP fast retransmit

If sender receives 3 ACKs for same data, resend unACKed segment with smallest sequence #

- Likely that unACKed segment lost, so don't wait for timeout

TCP fast retransmit



Fast retransmit after sender receipt of triple duplicate ACK

Staying Alive

- TCP keep-alive timer
  - If connection is idle > timeout, send a frame with no data to see if other side still alive
  - Checking for dead peer
  - Prevent disconnection due to inactivity
    - NAT box might drop your state if you don't communicate once in awhile

Summary

- Transmission Control Protocol (TCP)
  - Provides reliable byte-stream
  - Sequence number
    - Number of first byte in segment's data
  - Acknowledgement number
    - Next expected byte from other side
  - Estimating timeout value
  - Reliable transport in TCP
  - Fast transmit
    - Avoid waiting for timeout
    - Happens on triple duplicate ACK