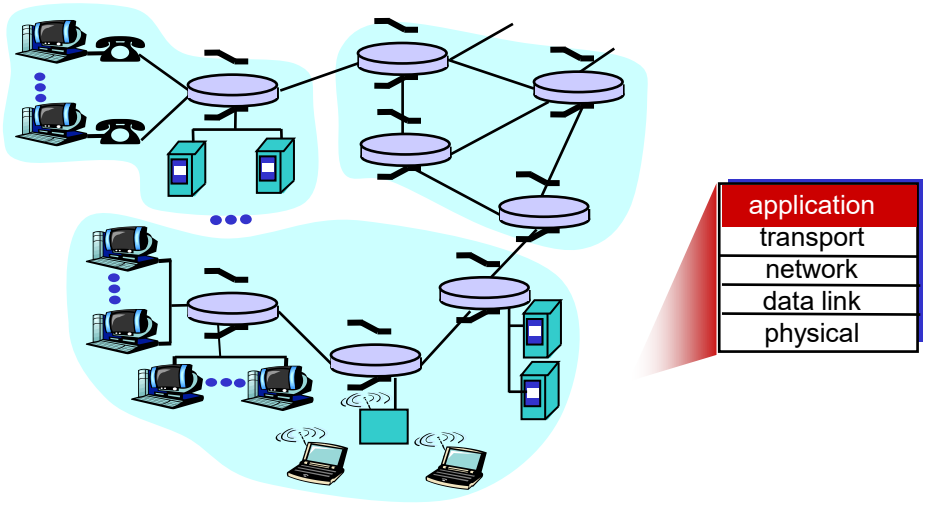
CS 4390
Computer Networks



Transmission Control Protocol (TCP) – An Overview

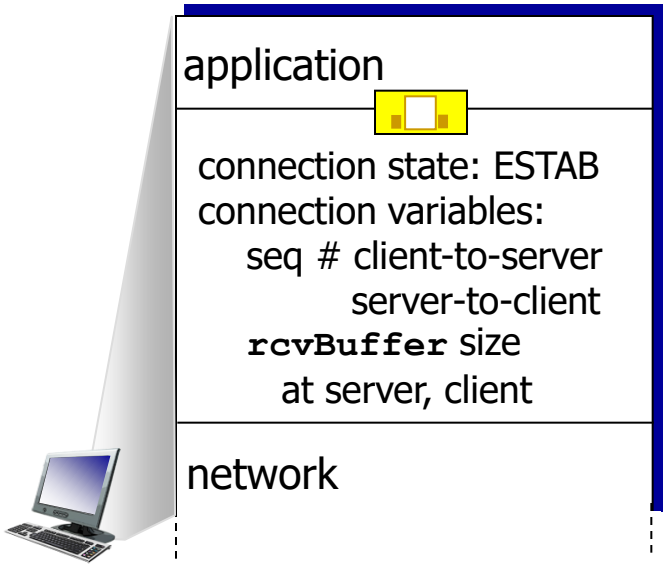
TCP: Overview *RFCs: 793,1122,1323, 2018, 2581*

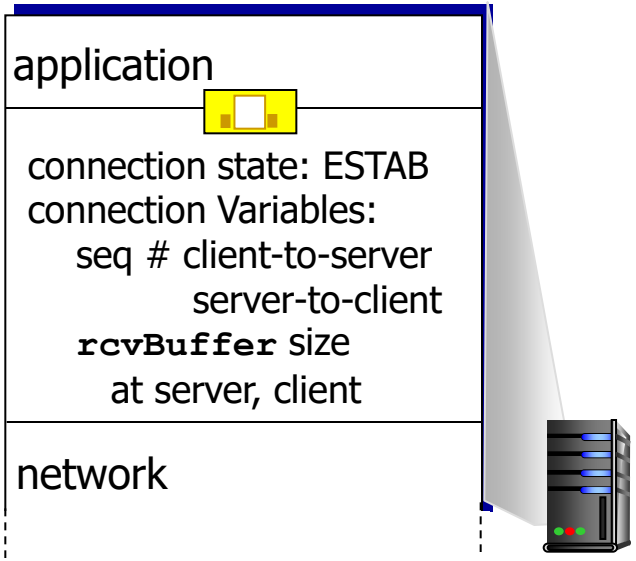
- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP *congestion* and *flow control* set window size
  - *Sliding window!*
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- flow controlled:
  - sender will not overwhelm receiver
- connection-oriented:
  - handshaking (exchange of control msgs) initialized sender, receiver state before data exchange

Connection Setup

before exchanging data, sender and receiver must:

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



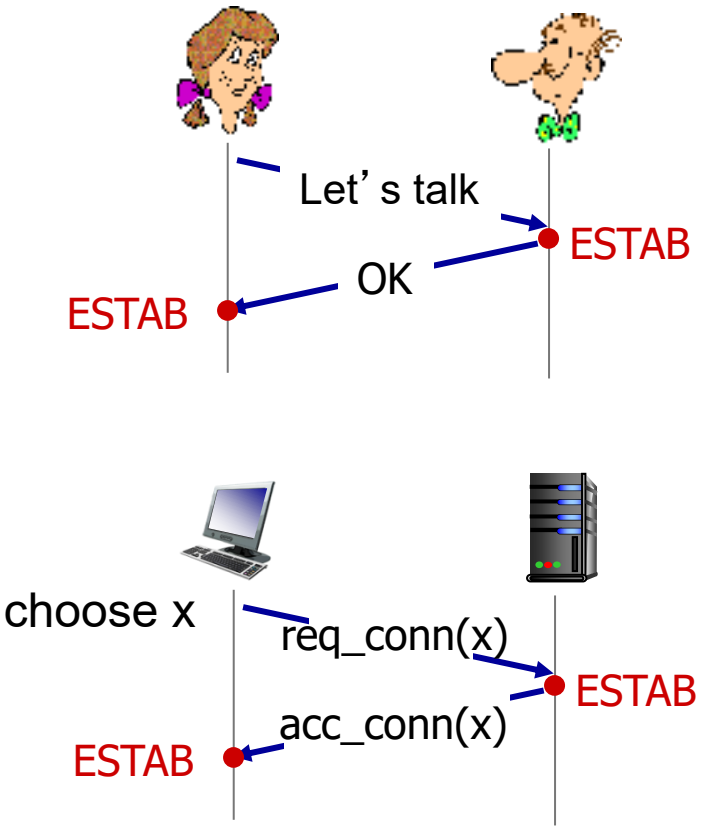



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Agreeing to Establish a Connection

2-way handshake:

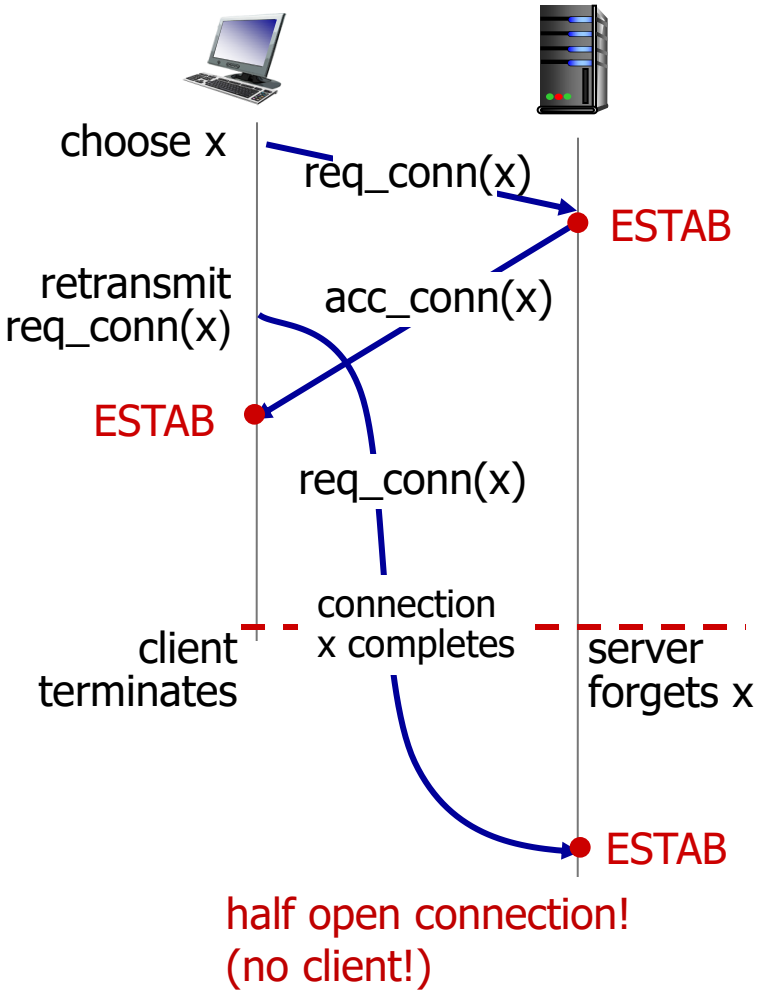


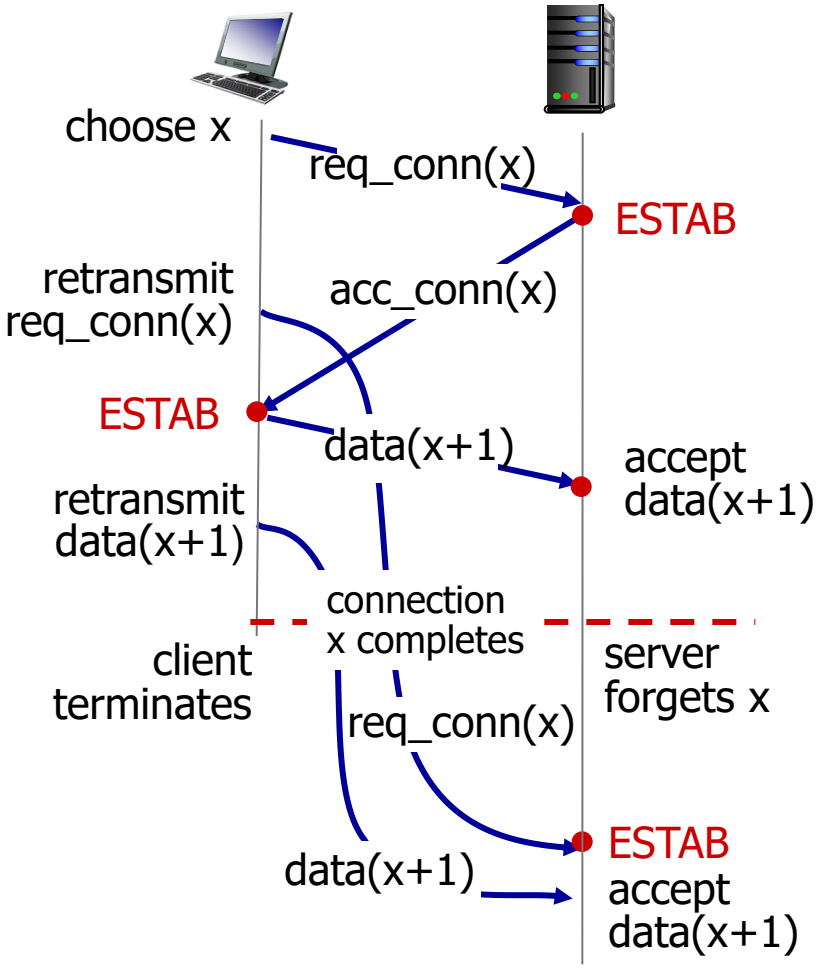
Q: will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

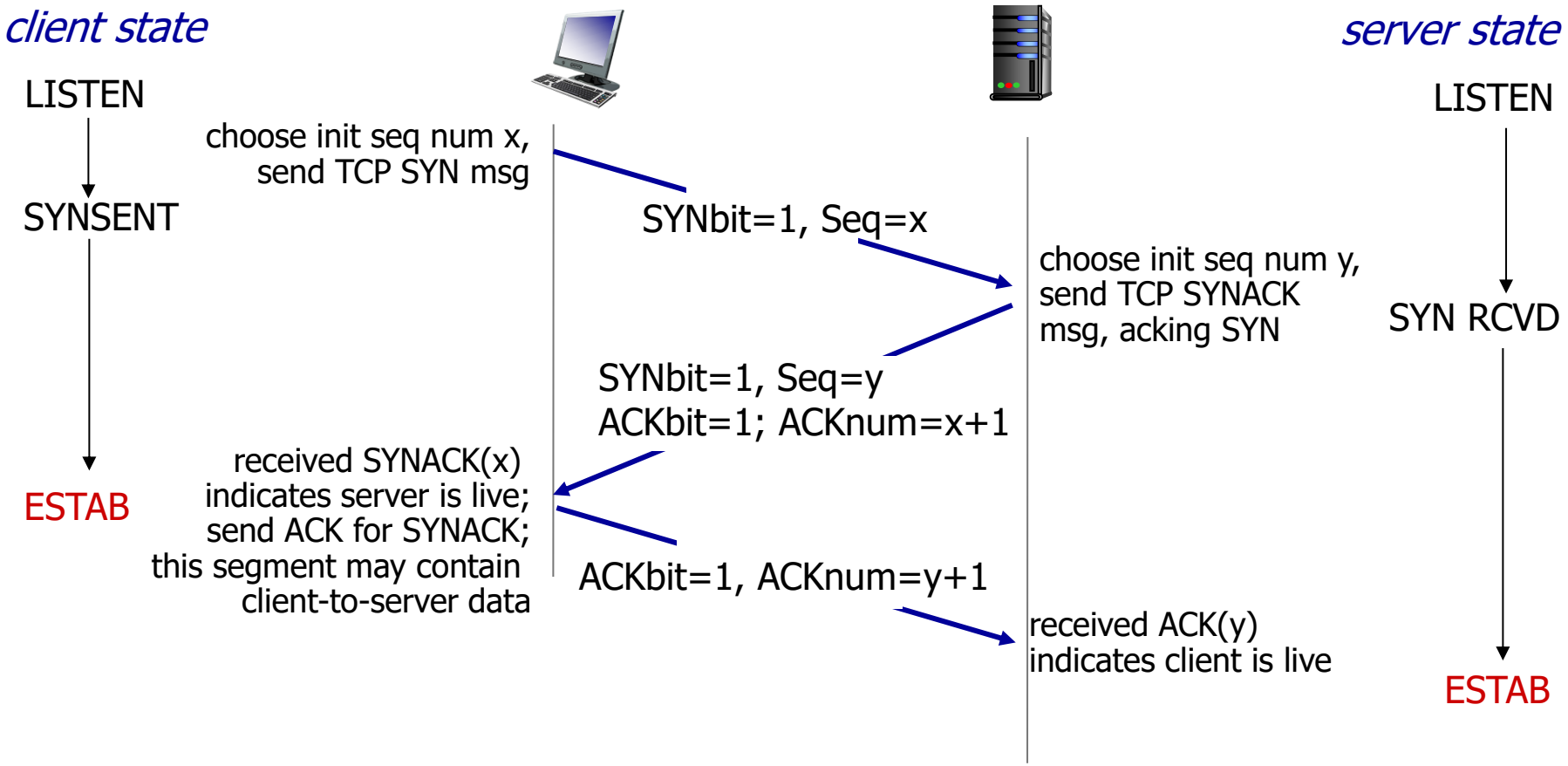
Agreeing to Establish a Connection

2-way handshake failure scenarios:

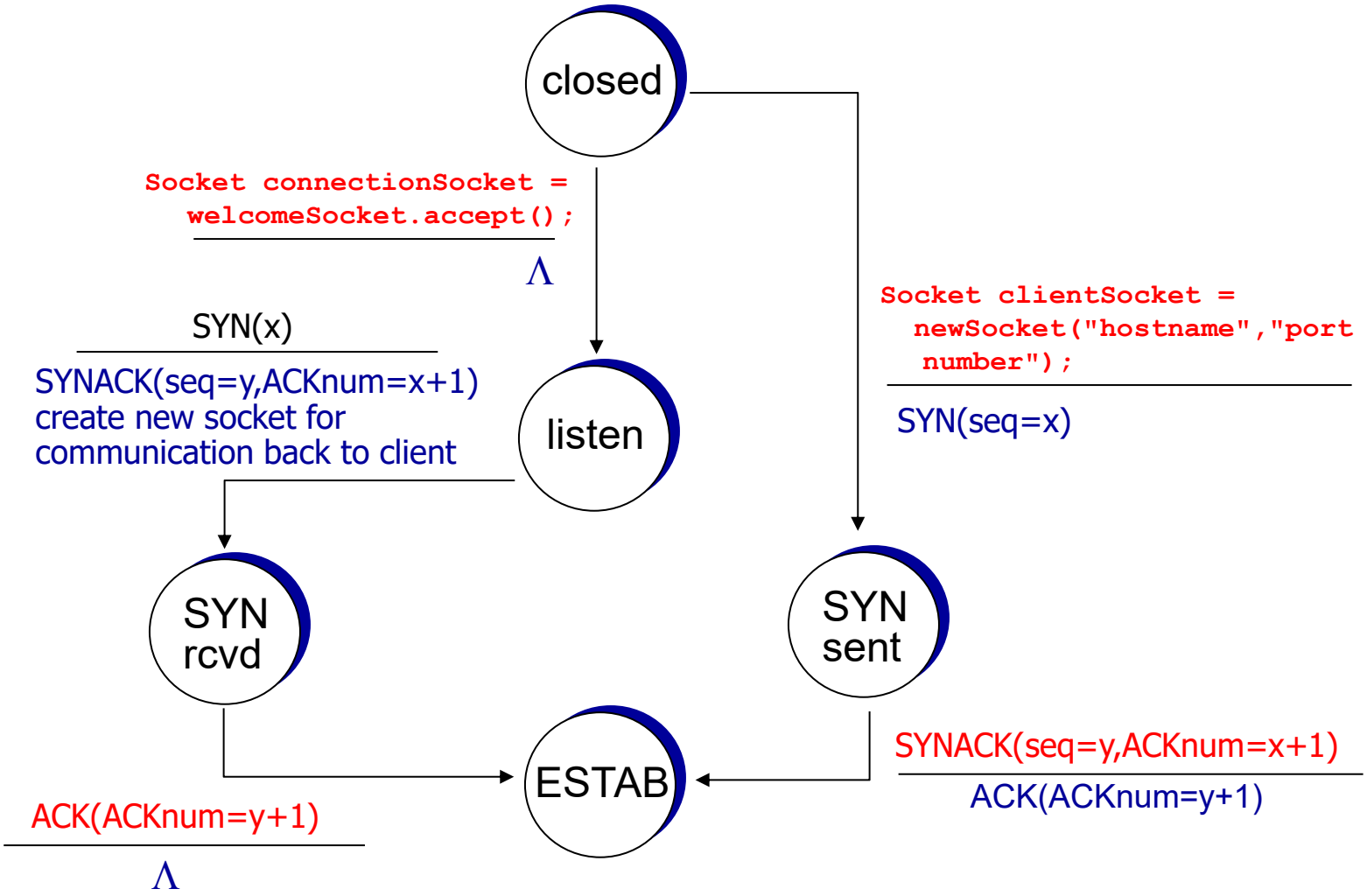




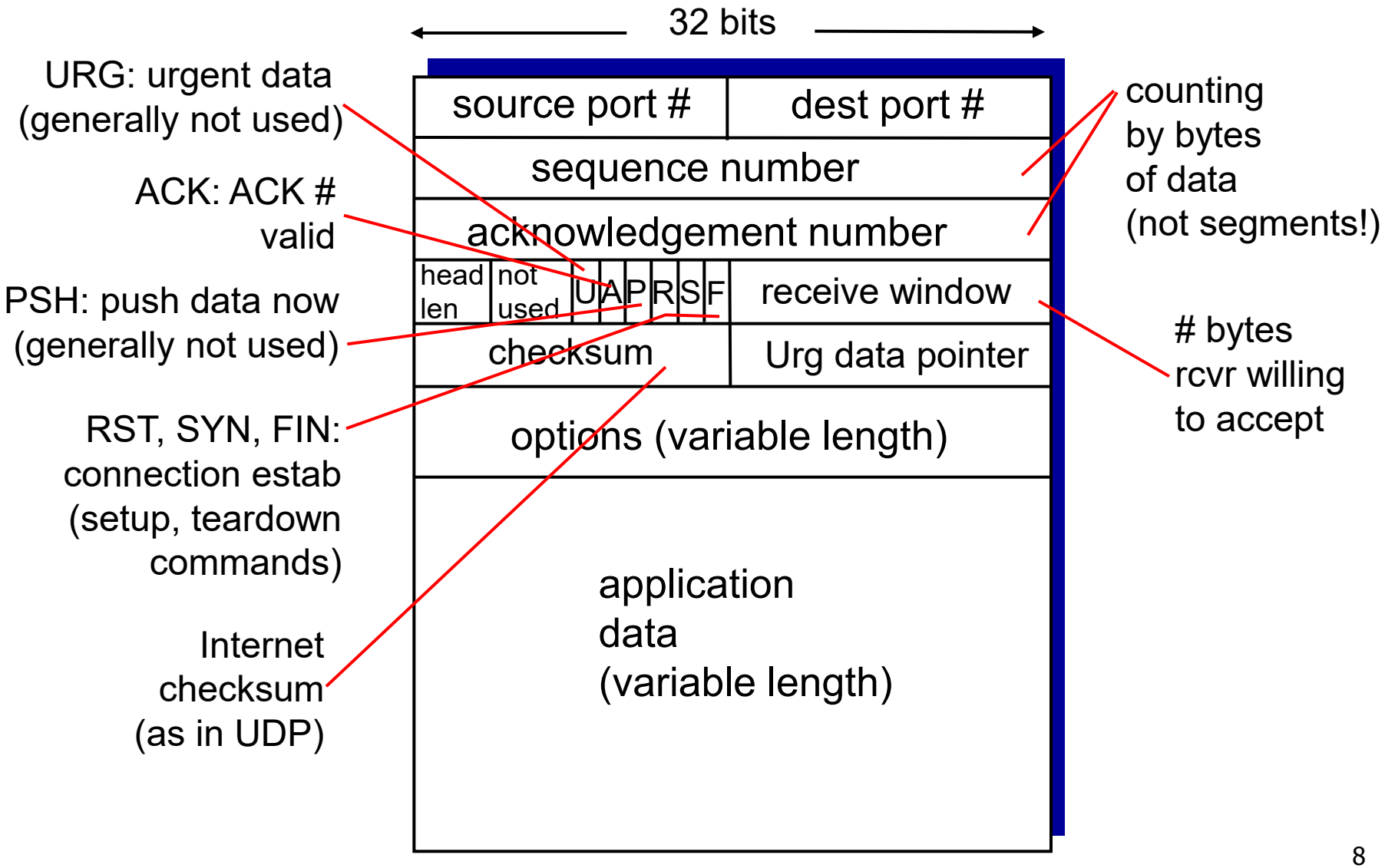
TCP 3-way Handshake



TCP 3-way Handshake: Server FSM



TCP Segment Structure



TCP seq. numbers, ACKs

sequence numbers:

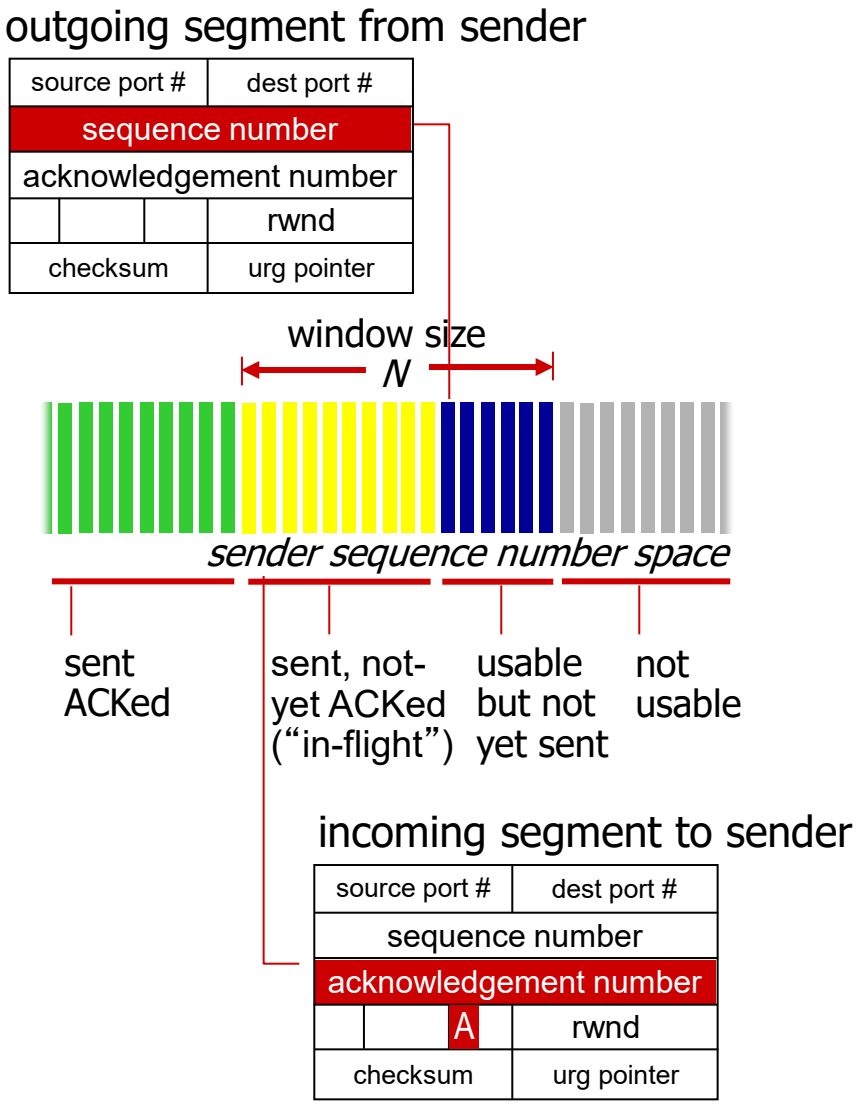
- byte stream "number" of first byte in segment's data

acknowledgements:

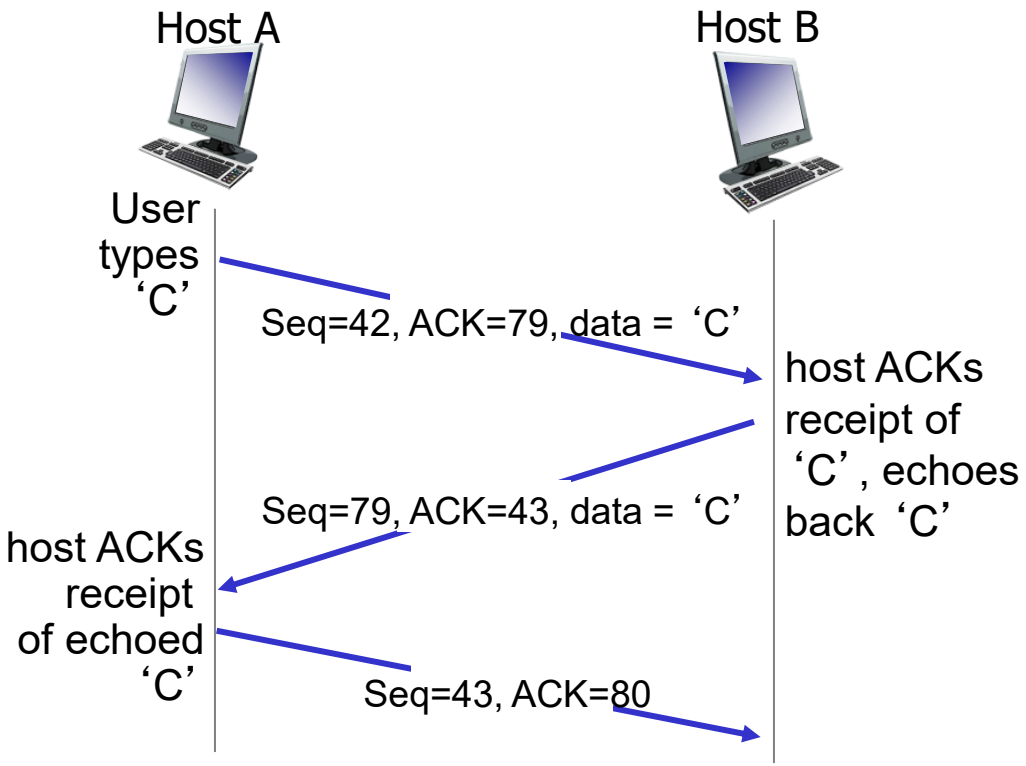
- seq # of next byte expected from other side
- cumulative ACK

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

- A: TCP spec doesn't say, - up to implementation!



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

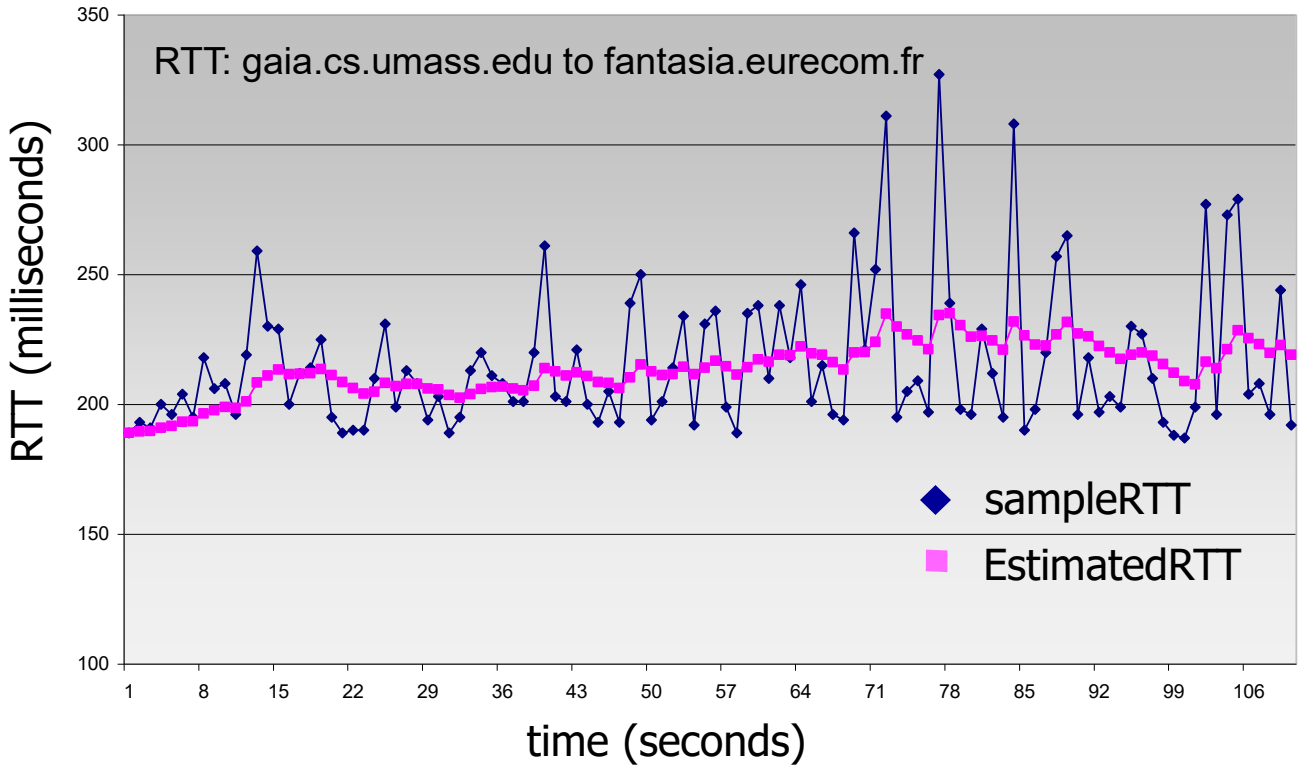
Q: *how to estimate RTT?*

- **`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

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

- *exponential weighted moving average*
- influence of past sample decreases exponentially fast
- typical value: α = 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`**:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT}-\texttt{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

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

estimated RTT

"safety margin"

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 unack'ed segment
  - expiration interval: **`TimeOutInterval`**

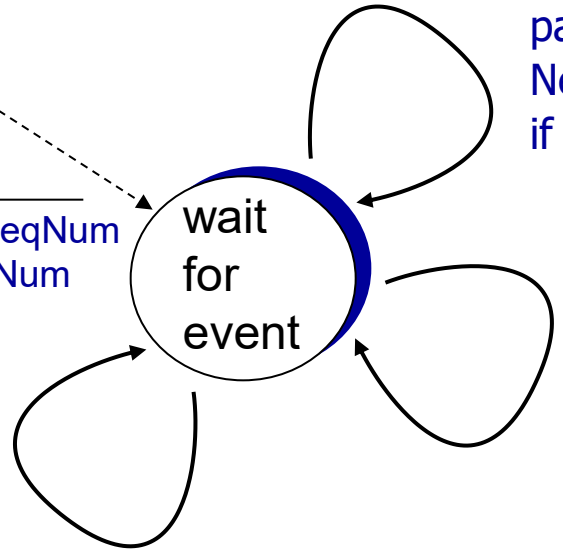
timeout:

- retransmit segment that caused timeout
- restart timer

ACK rcvd:

- if ack acknowledges previously unack'ed 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

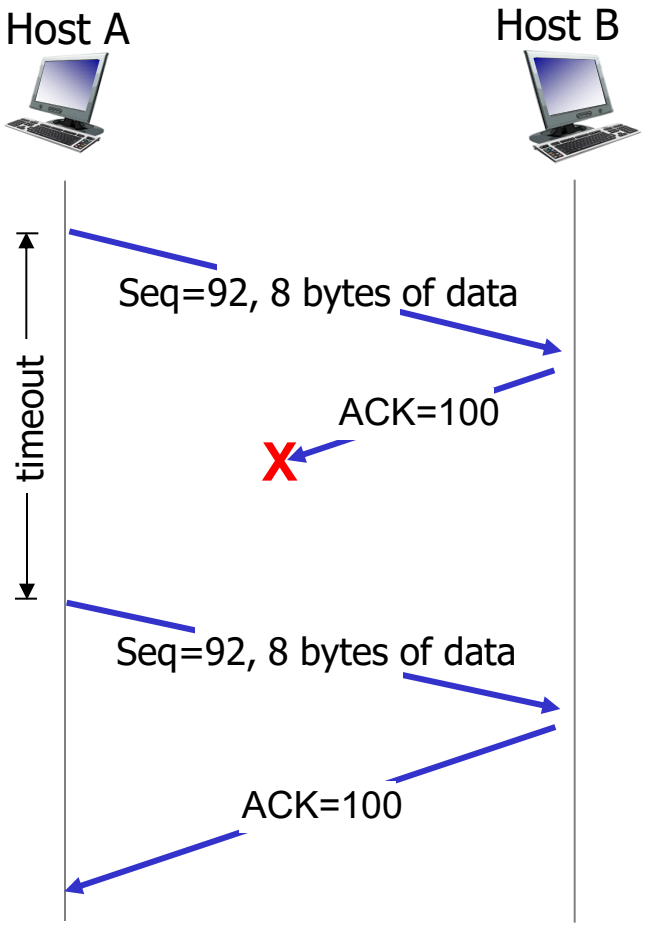
timeout

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

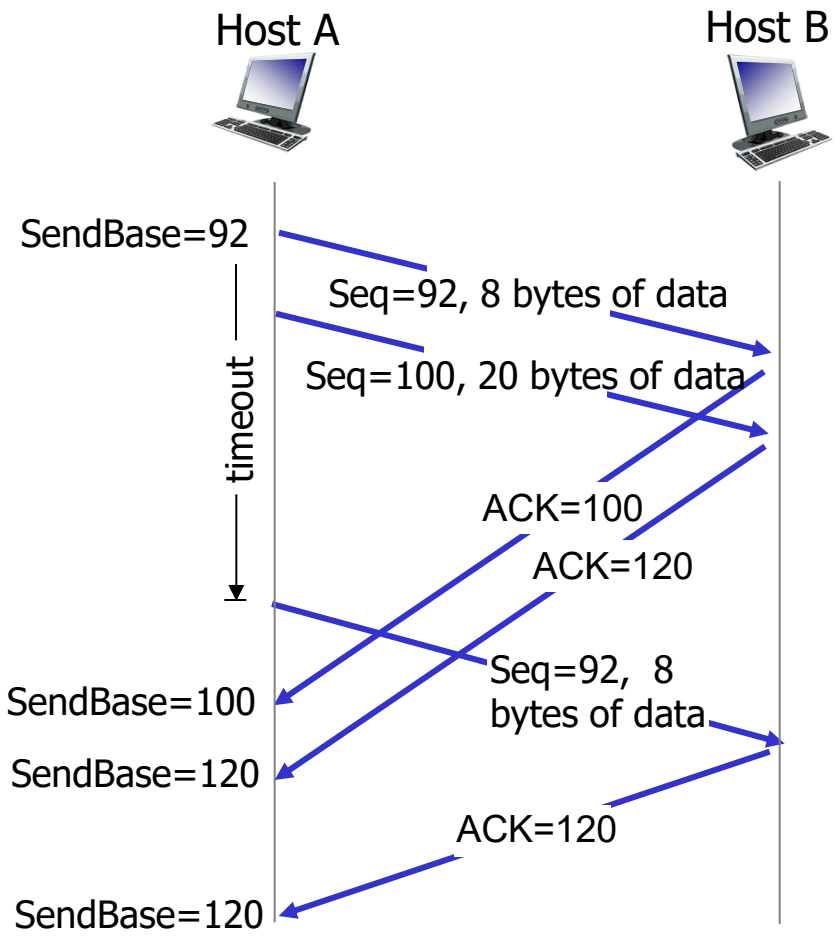
ACK received, with ACK field value y

```
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

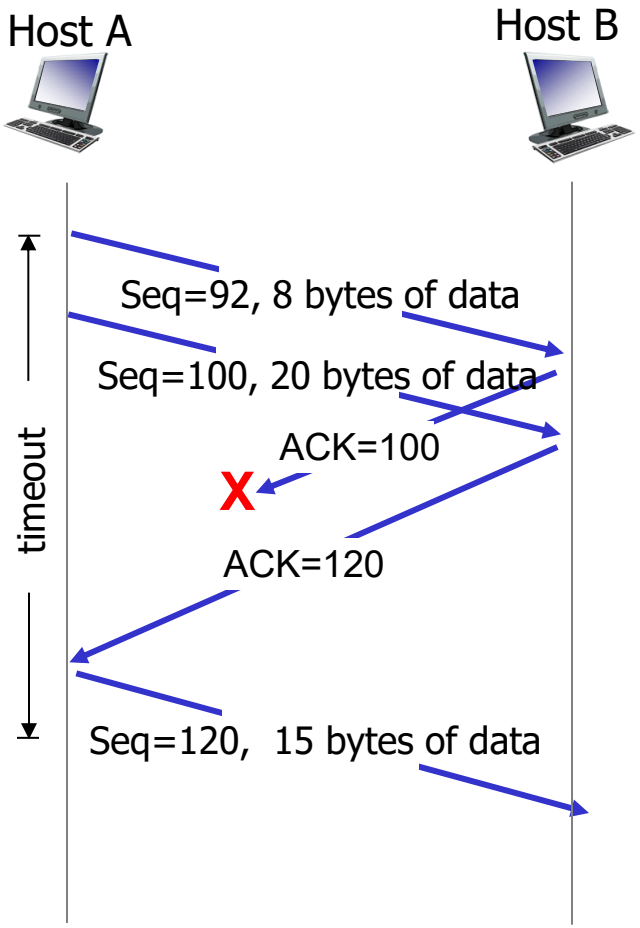


lost ACK scenario



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	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.

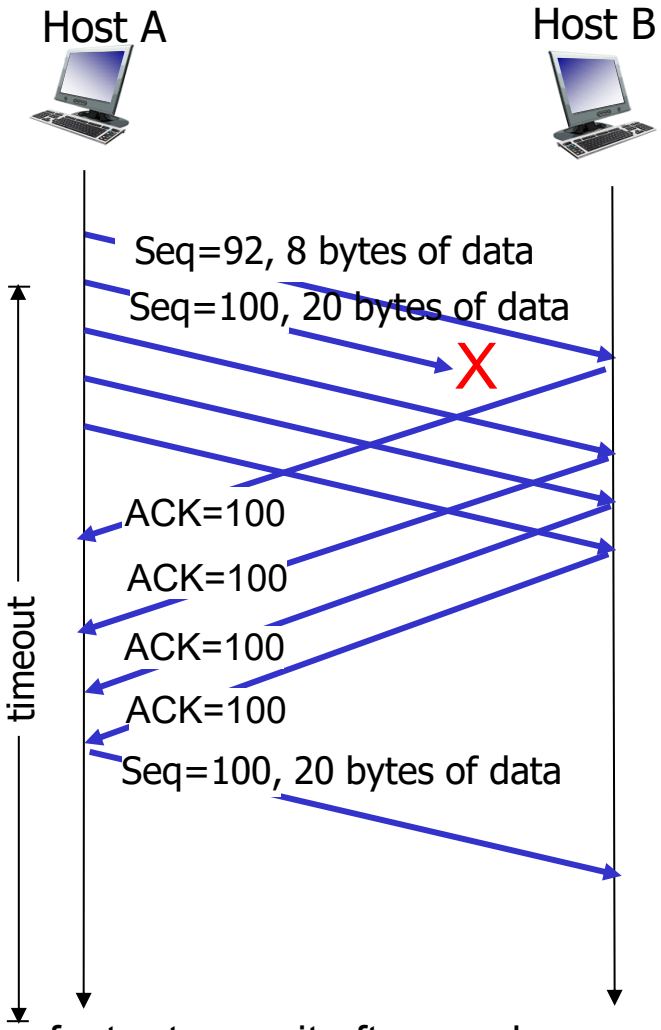
TCP fast retransmit

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



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