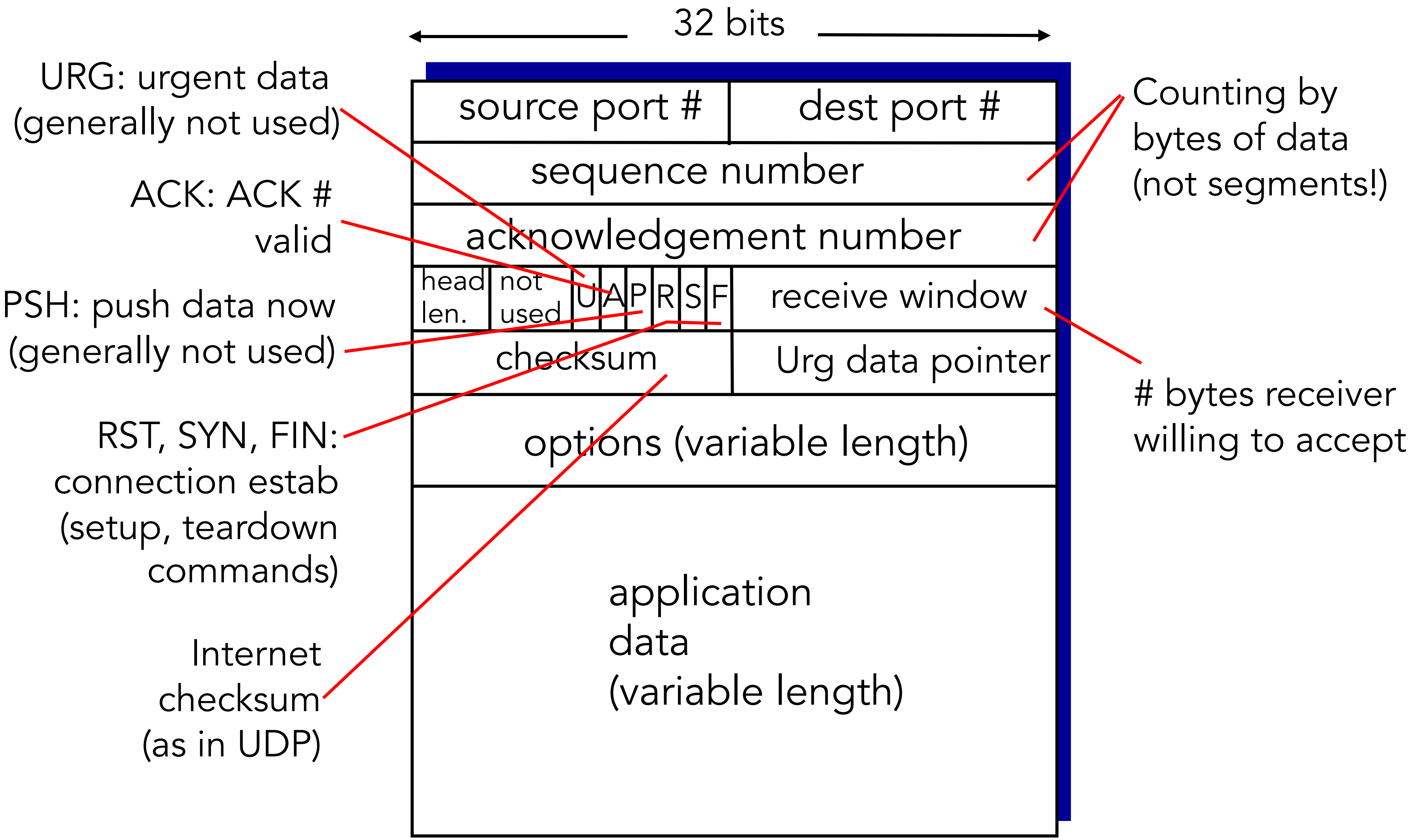
Transport Layer
- TCP Basics -

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

TCP: Overview [RFC 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:

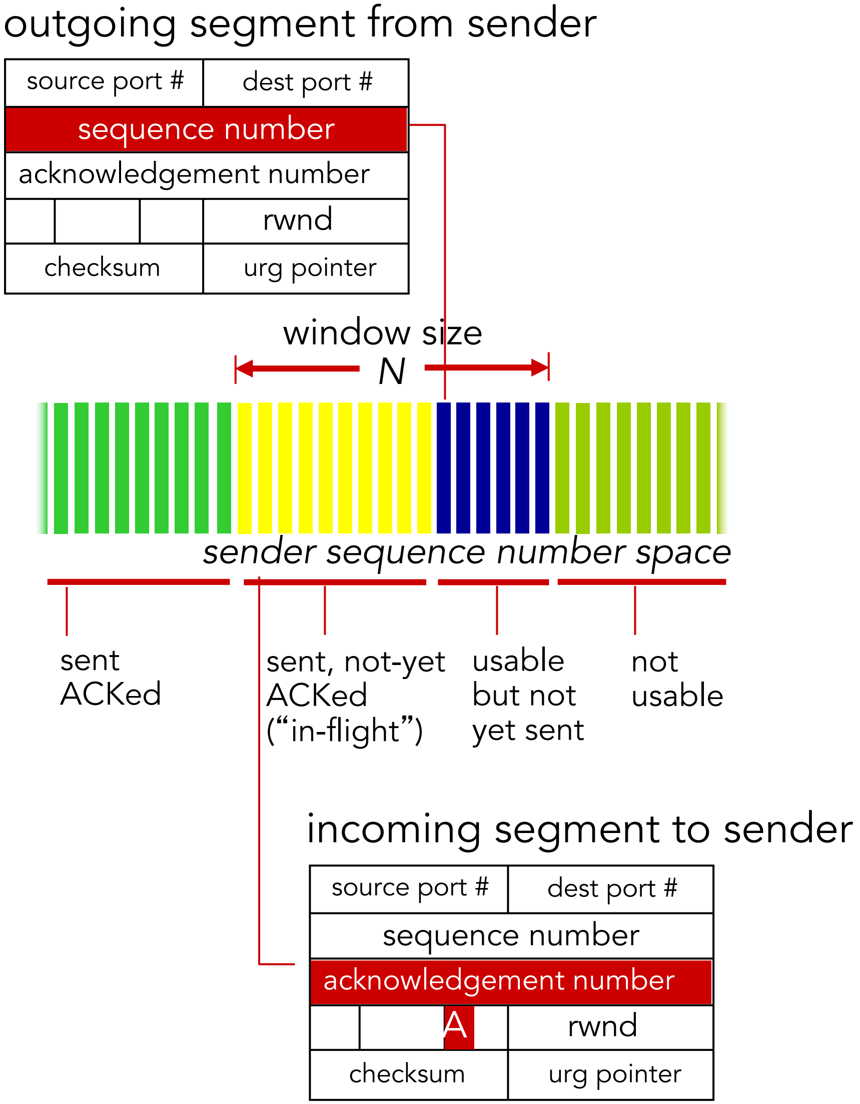
- byte stream "number" of **first byte** in segment's data

Acknowledgements:

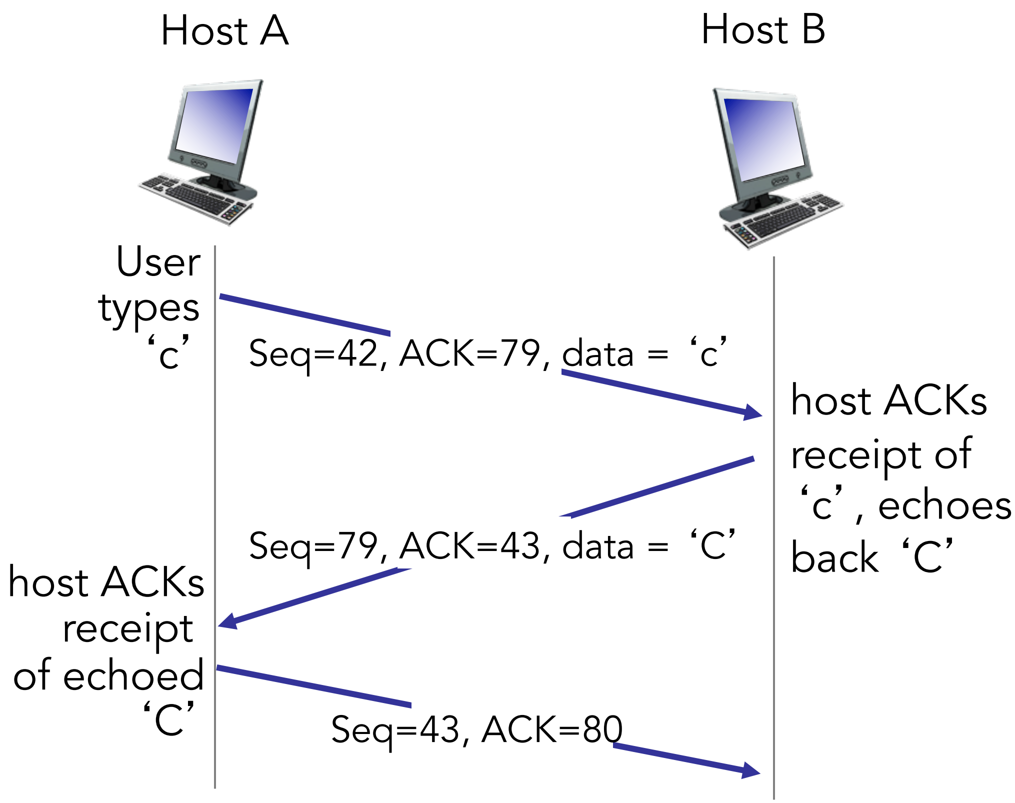
- seq # of **next byte expected** from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, it is up to implementor



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

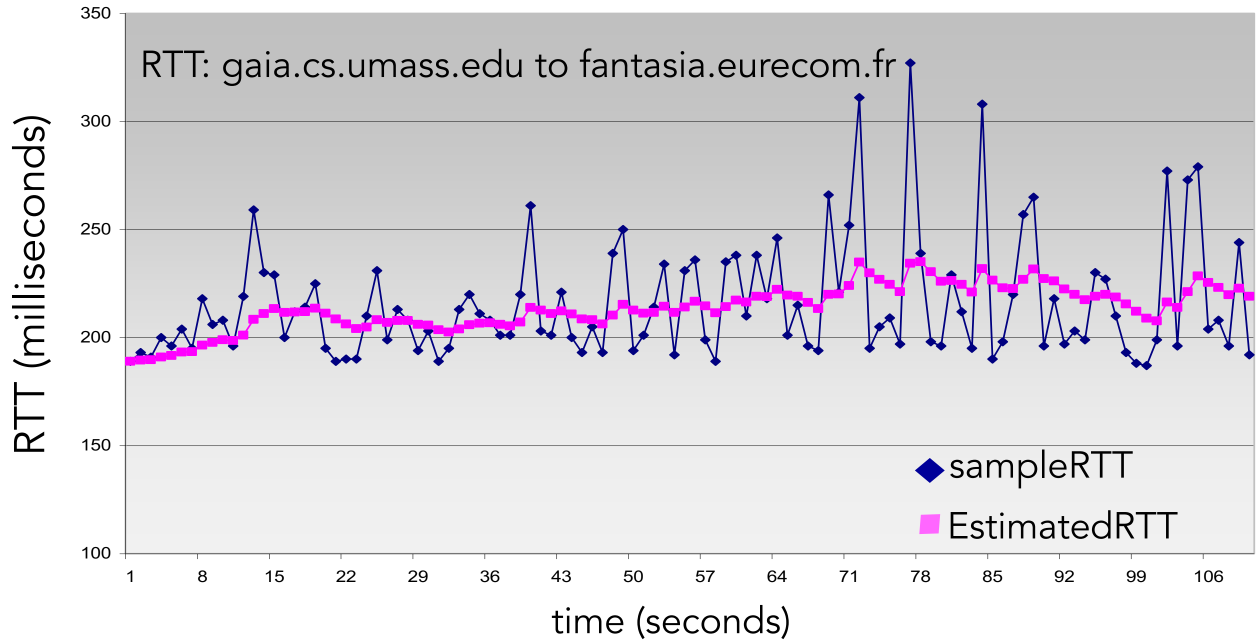
Q: how to estimate RTT?

- 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

- EWMA: Exponentially 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:

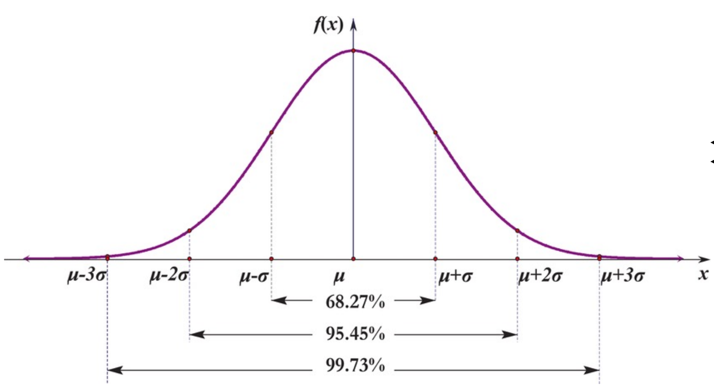
```
DevRTT = (1-β)*DevRTT +
               β*|SampleRTT-EstimatedRTT|
```

(typically, $\beta = 0.25$)

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

estimated RTT

"safety margin"



Assuming a normal distribution

$\mu \pm 4\sigma$: 99.9937%
$\mu \pm 5\sigma$: 99.99994%
$\mu \pm 6\sigma$: 99.9999998%

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 unacked segment
  - expiration interval: TimeOutInterval

timeout:

- retransmit segment that caused timeout
- restart timer

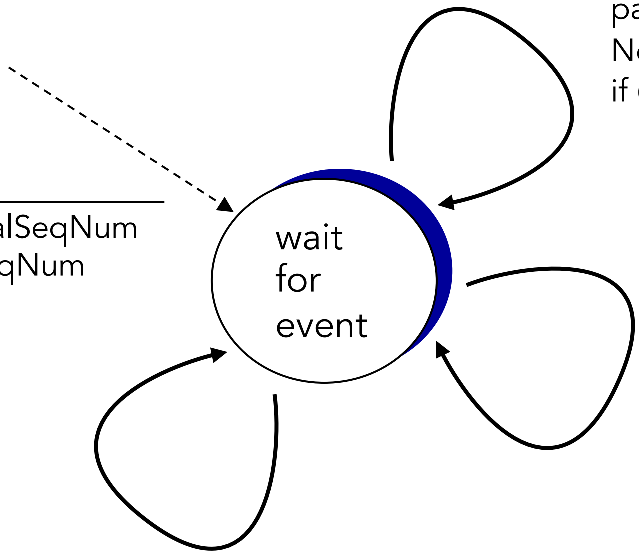
ack rcvd:

- 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

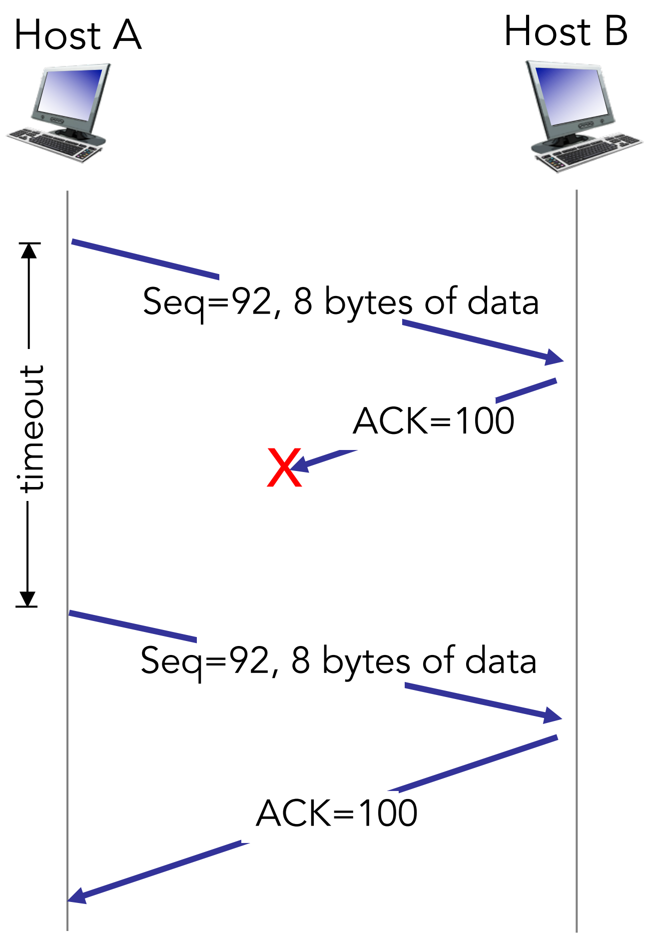
timeout

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

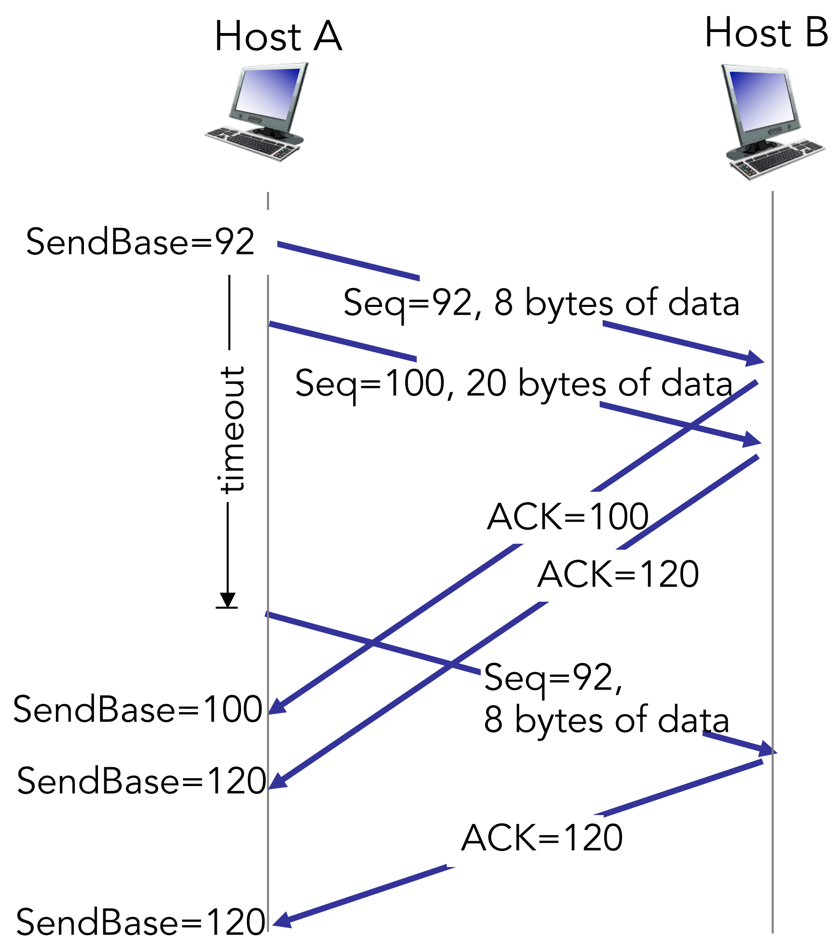
ACK received, with ACK field value y

```
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 (check seq#)

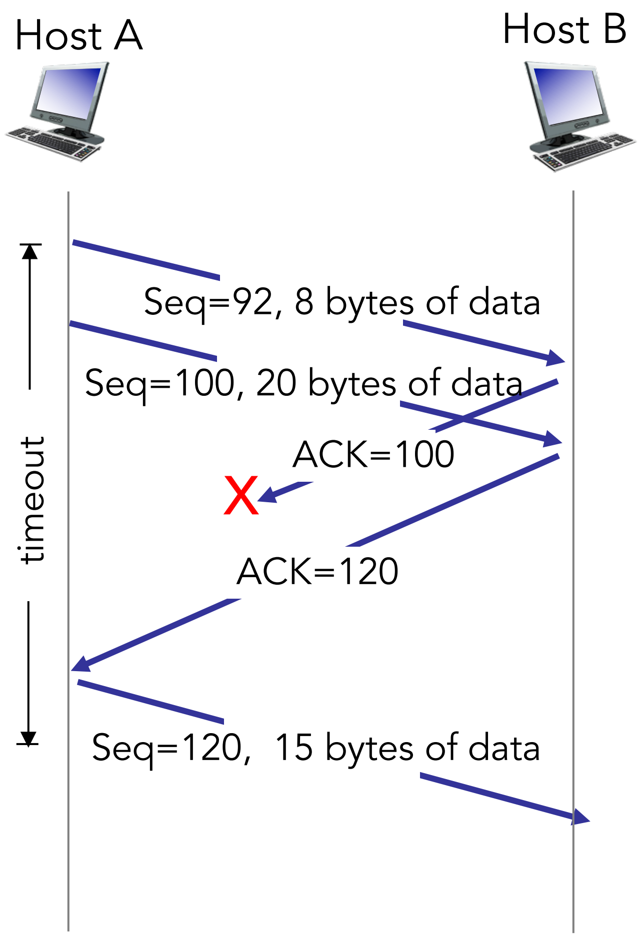


lost ACK scenario



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 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.

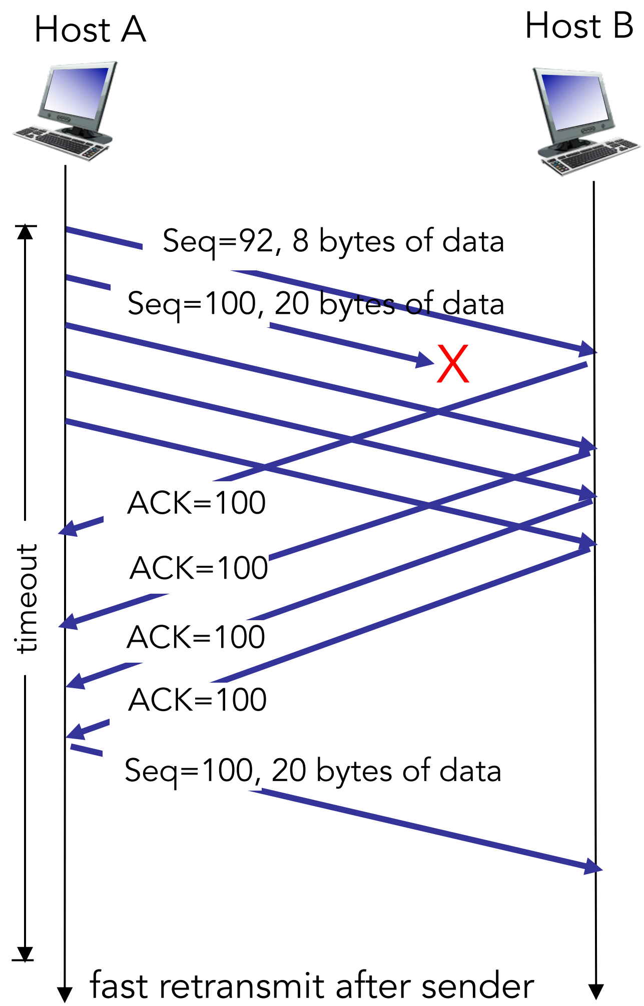
TCP fast retransmit

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

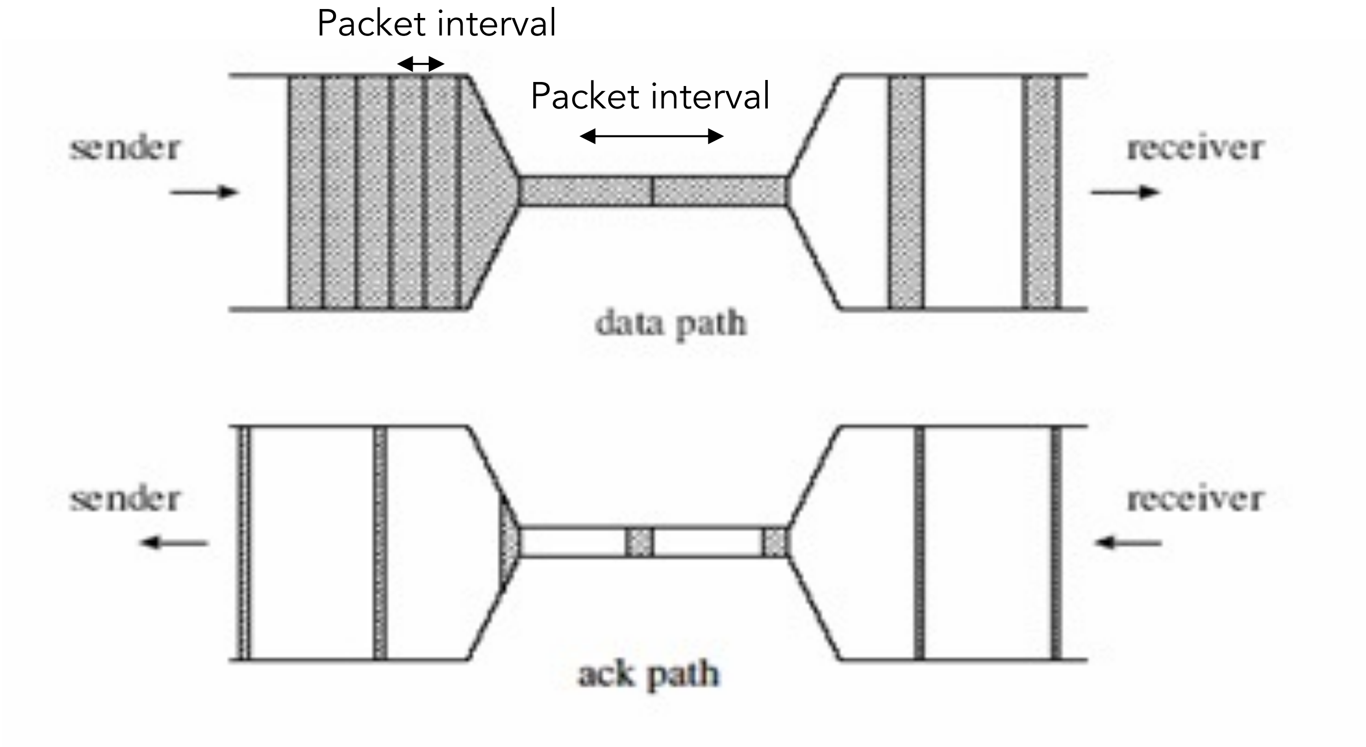
TCP Fast Retransmit



fast retransmit after sender receipt of triple duplicate ACK

Congestion control in TCP

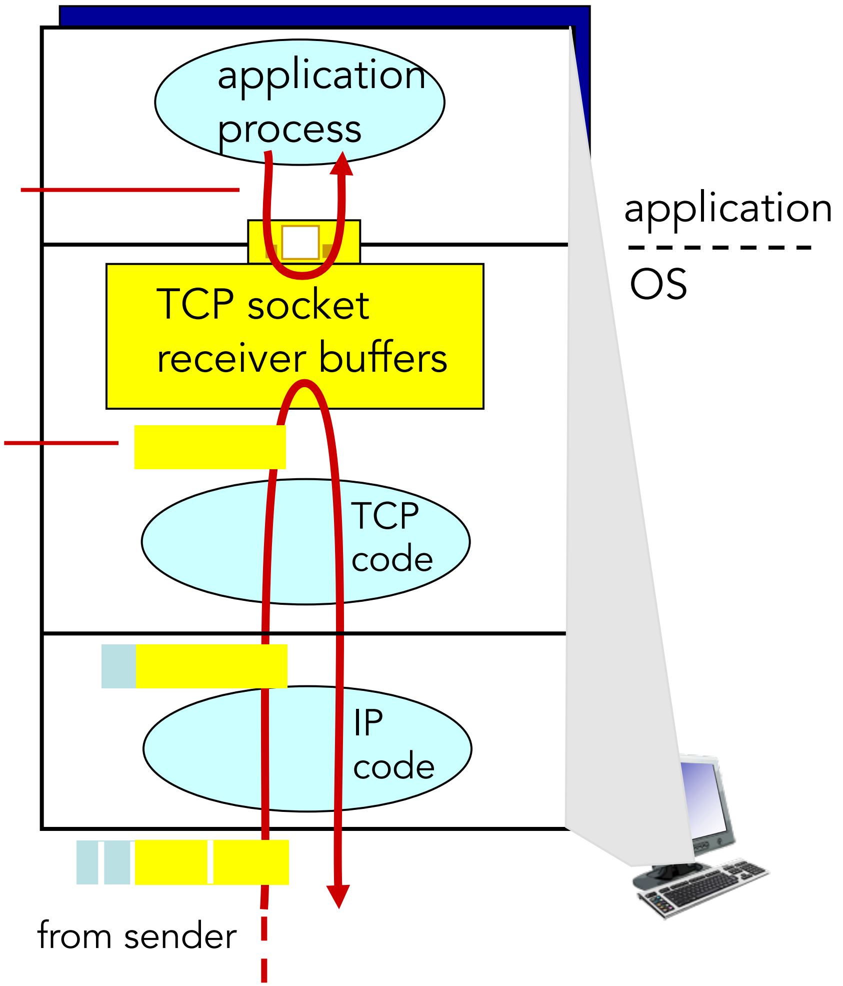
- **CWND** and Ack
  - "ACK clocking": the sender transmits a data packet upon an ACK reception



TCP Flow Control

application may
remove data from
TCP socket buffers ....

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



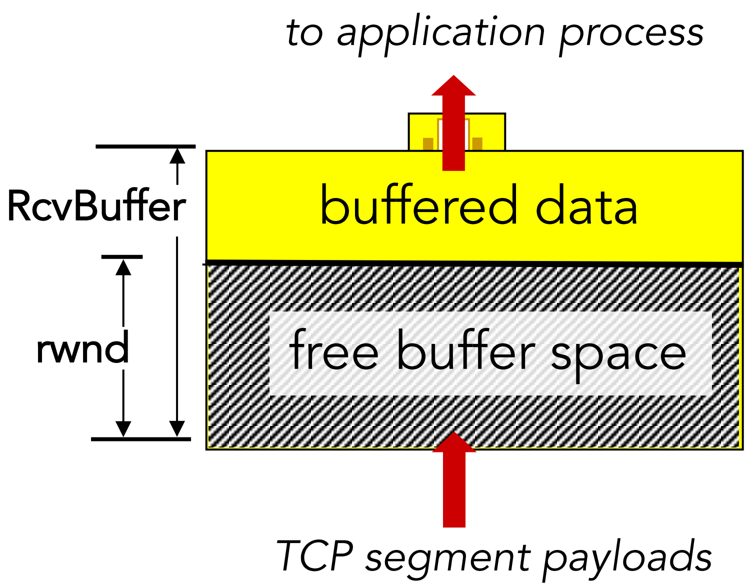
receiver protocol stack

flow control

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

TCP Flow Control

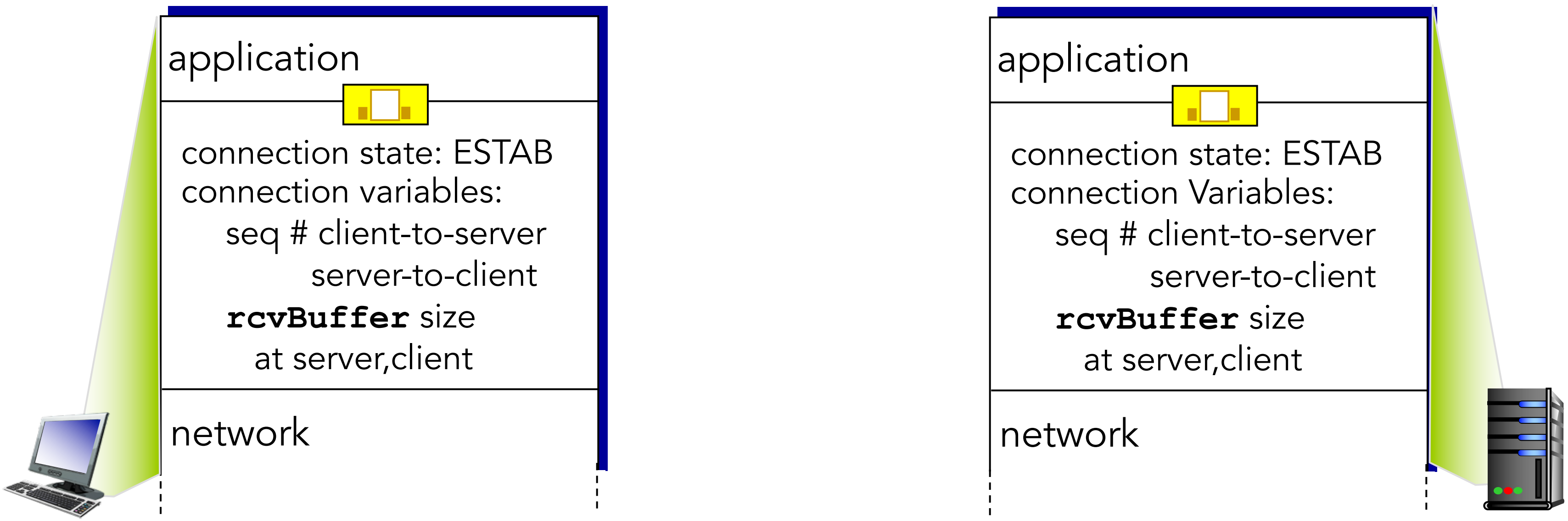
- Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto adjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



receiver-side buffering

Connection Management

Before exchanging data, sender/receiver handshake:
- agree to establish connection
- agree on connection parameters

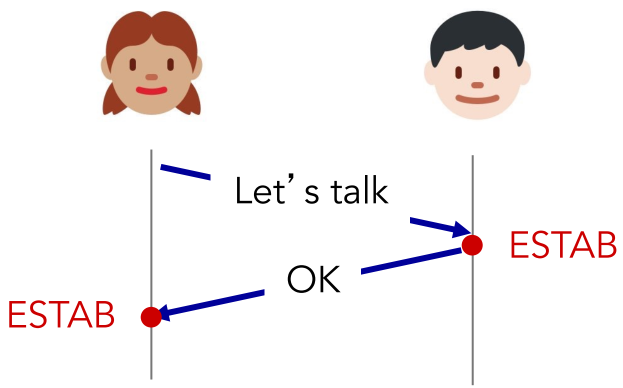


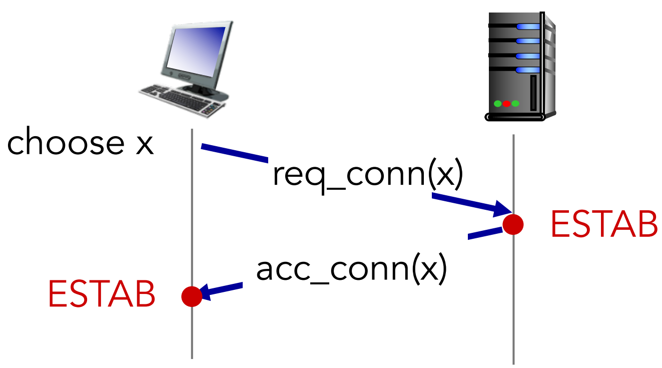

```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Agreeing to establish a connection

2-way handshake:



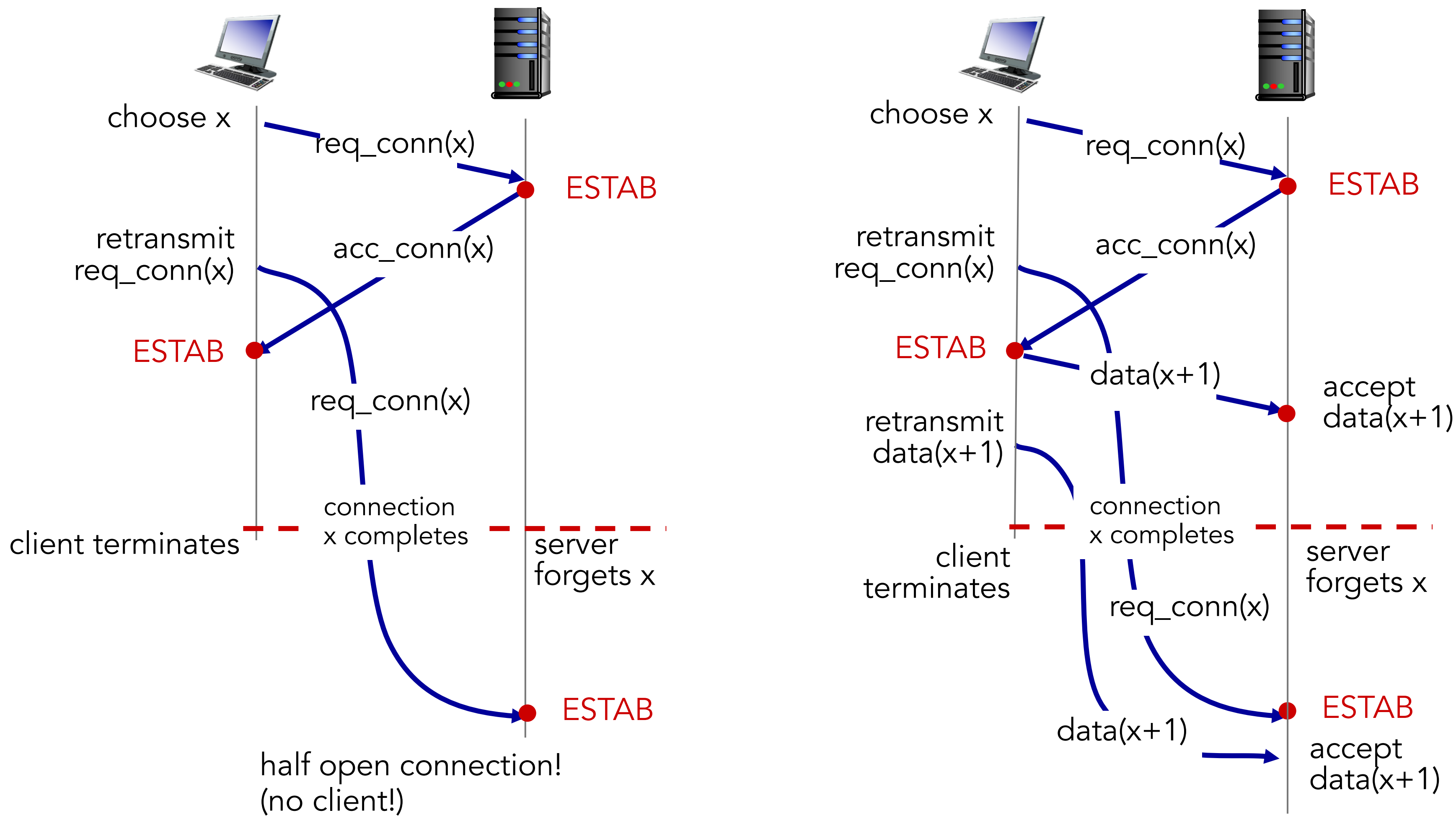


Q: will 2-way handshake always work in network?

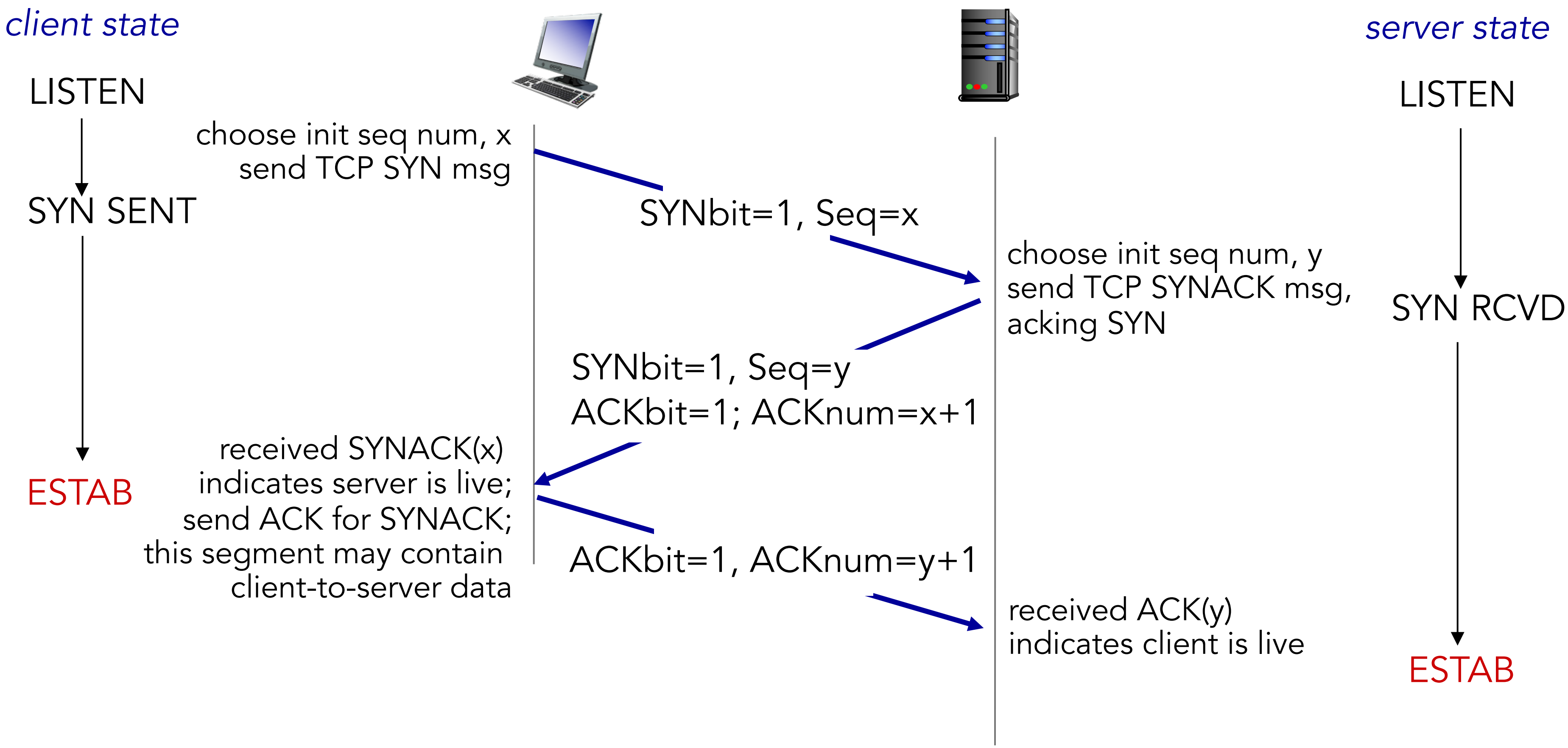
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

Agreeing to establish a connection

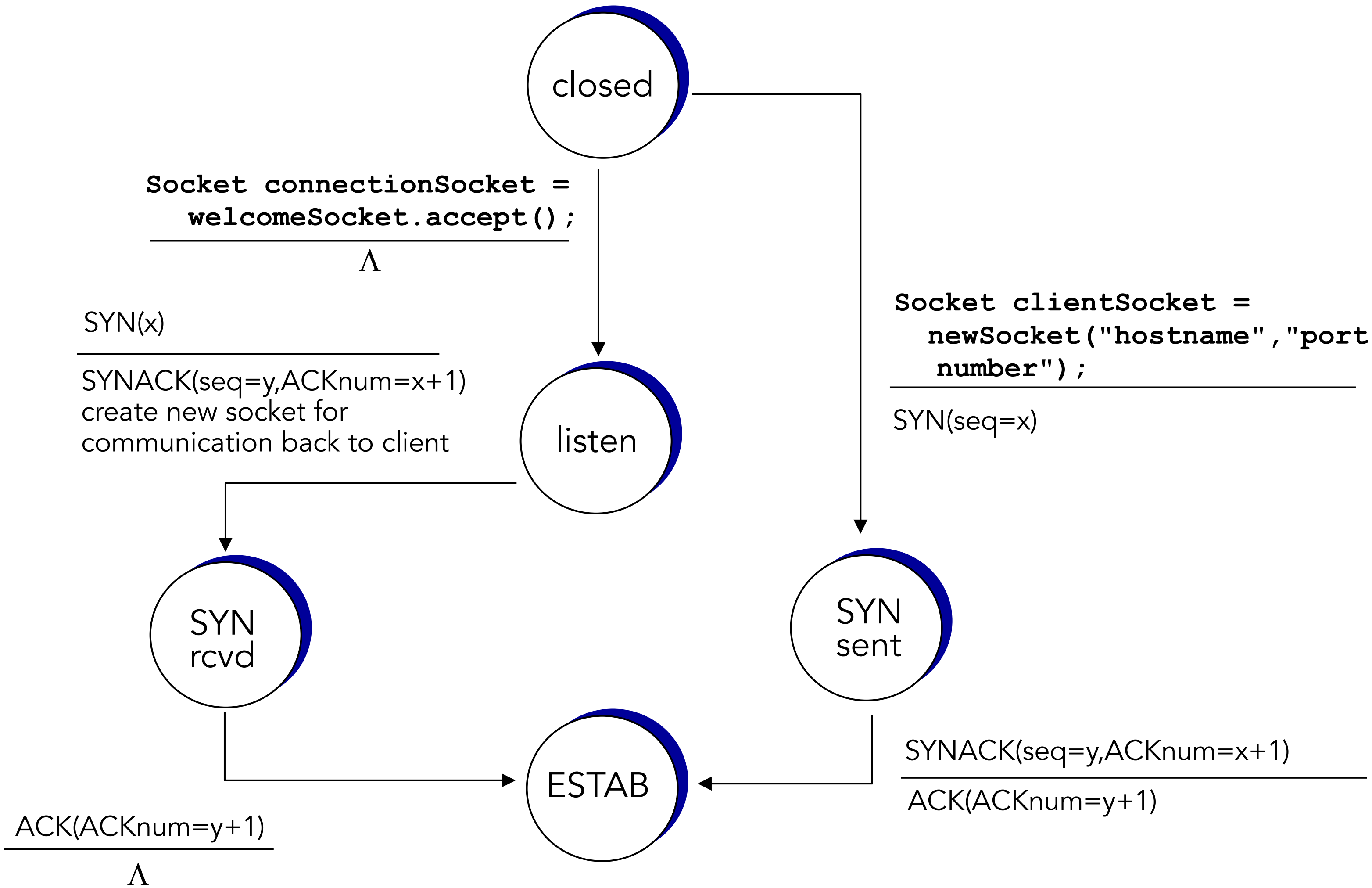
2-way handshake failure scenarios:



TCP 3-way handshake



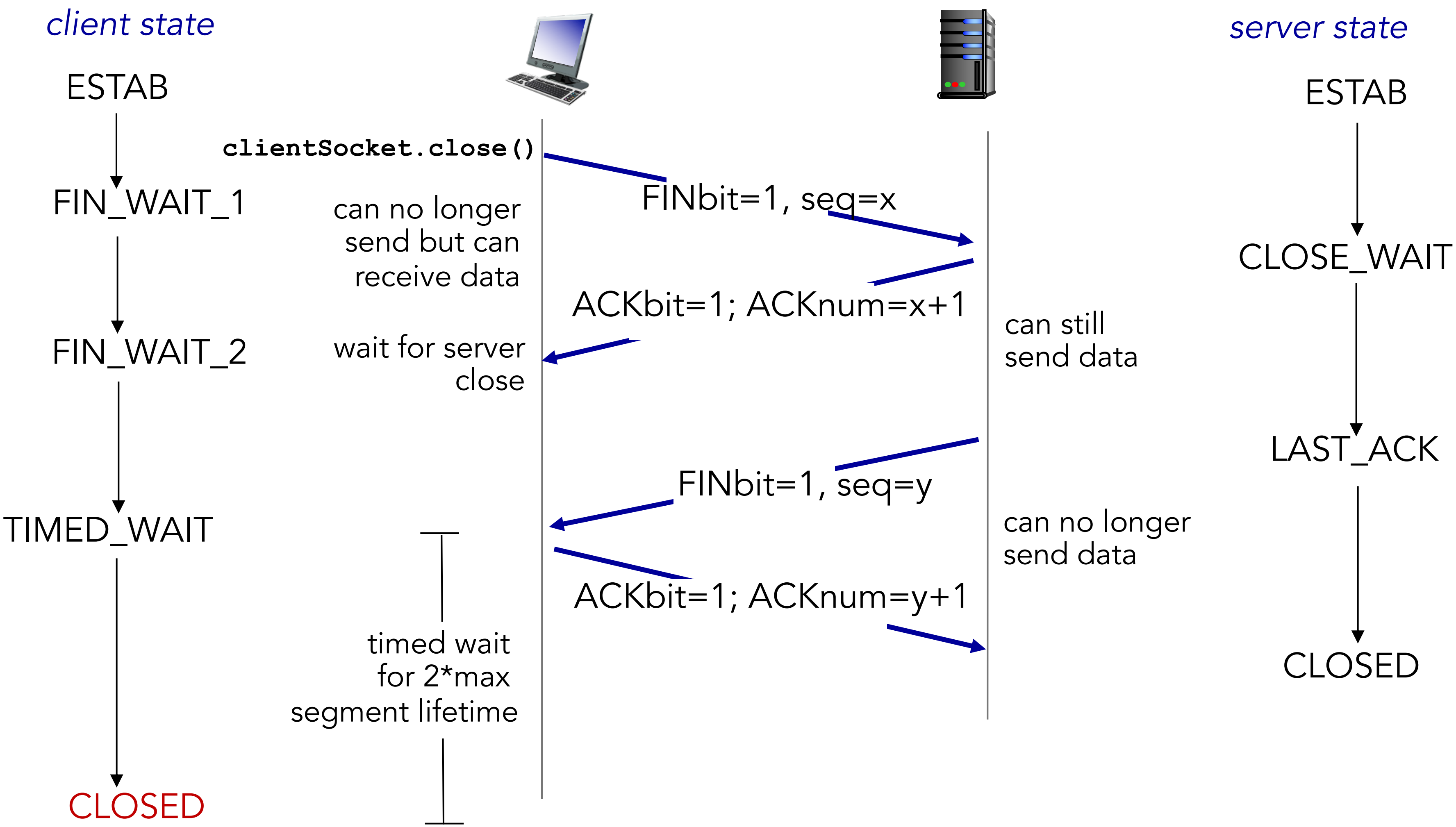
TCP 3-way handshake: FSM



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



TCP SYN, FIN

TCP Initial: SYN, SYN-ACK, ACK

Time	Source	Destination	Protocol	Info
0.0000	130.207.228.23	199.77.227.200	TCP	51845 > smtp [SYN] Seq=0 Ack=0 Win=65535 [CHEC
0.0005	199.77.227.200	130.207.228.23	TCP	smtp > 51845 [SYN, ACK] Seq=0 Ack=1 Win=24616
0.0005	130.207.228.23	199.77.227.200	TCP	51845 > smtp [ACK] Seq=1 Ack=1 Win=65535 [CHEC

TCP Final: FIN, ACK, FIN-ACK, ACK

Time	Source	Destination	Protocol	Info
116.29	130.207.228.23	199.77.227.200	SMTP	Command: QUIT
116.29	199.77.227.200	130.207.228.23	SMTP	Response: 221 2.0.0 mail.ece.gatech.edu clo
116.29	199.77.227.200	130.207.228.23	TCP	smtp > 51845 [FIN, ACK] Seq=261 Ack=20
116.29	130.207.228.23	199.77.227.200	TCP	51845 > smtp [ACK] Seq=20 Ack=262 Win=
116.29	130.207.228.23	199.77.227.200	TCP	51845 > smtp [FIN, ACK] Seq=20 Ack=262
116.29	199.77.227.200	130.207.228.23	TCP	smtp > 51845 [ACK] Seq=262 Ack=21 Win=