Transport Layer – TCP

To do …

- ❑ Overview and structure
- ❑ Sequence numbers and retransmission
- ❑ Flow control
- ❑ 3-way handshake and closing a connection

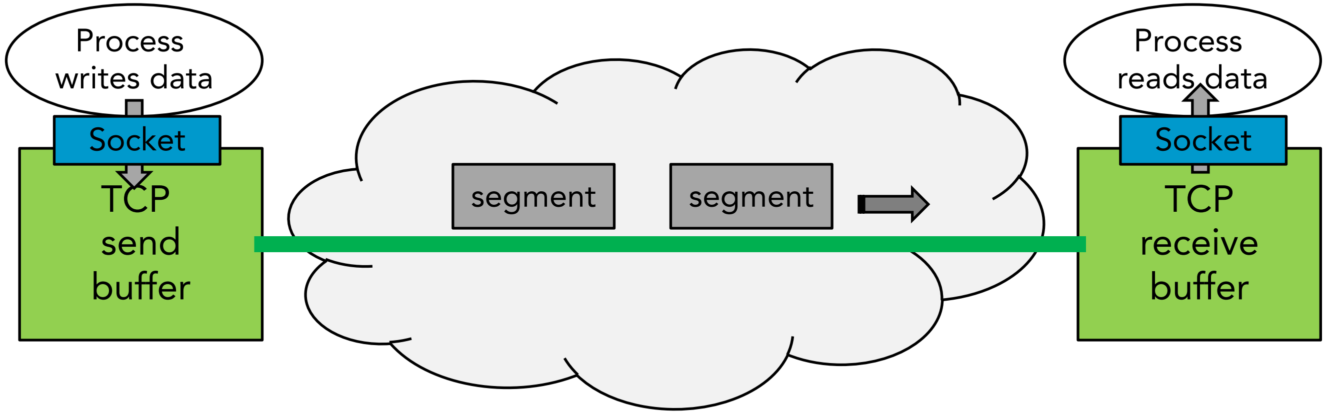
Northwestern

TCP overview

- Connection-oriented – handshaking (exchange of control msgs) to initialize sender and receiver state before data exchange
  - A logical, end-to-end connection, not a TDM or FDM one
- Point-to-point – one sender, one receiver (e.g., no multicasting)
- Full-duplex – Bi-directional data flow in same connection
- Reliable, in-order byte stream – no “message boundaries”
- App data is placed on the buffer (set aside during the handshake), and sent in pieces as segments

TCP overview

- App data is placed on the buffer, sent in pieces as segments
  - Piece at a time as given by Maximum Segment Size (MSS)
  - MSS based on largest link-layer frame or MTU – Maximum Transmission Unit - Typical MTU is 1,500B, minus TCP headers 40B → MSS of 1,460B
  - Segments are buffered on the other end, and the app reads the stream of data from there
- Pipelined – TCP congestion and flow control set window size
- Flow controlled – Sender will not overwhelm receiver



TCP segment structure

Header length in 32-bit words

URG: urgent data (generally not used)

For reliable transfer

ACK: ACK # valid

TCP Header

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer (if URG set)															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															
		Data																															

Internet checksum (as in UDP)

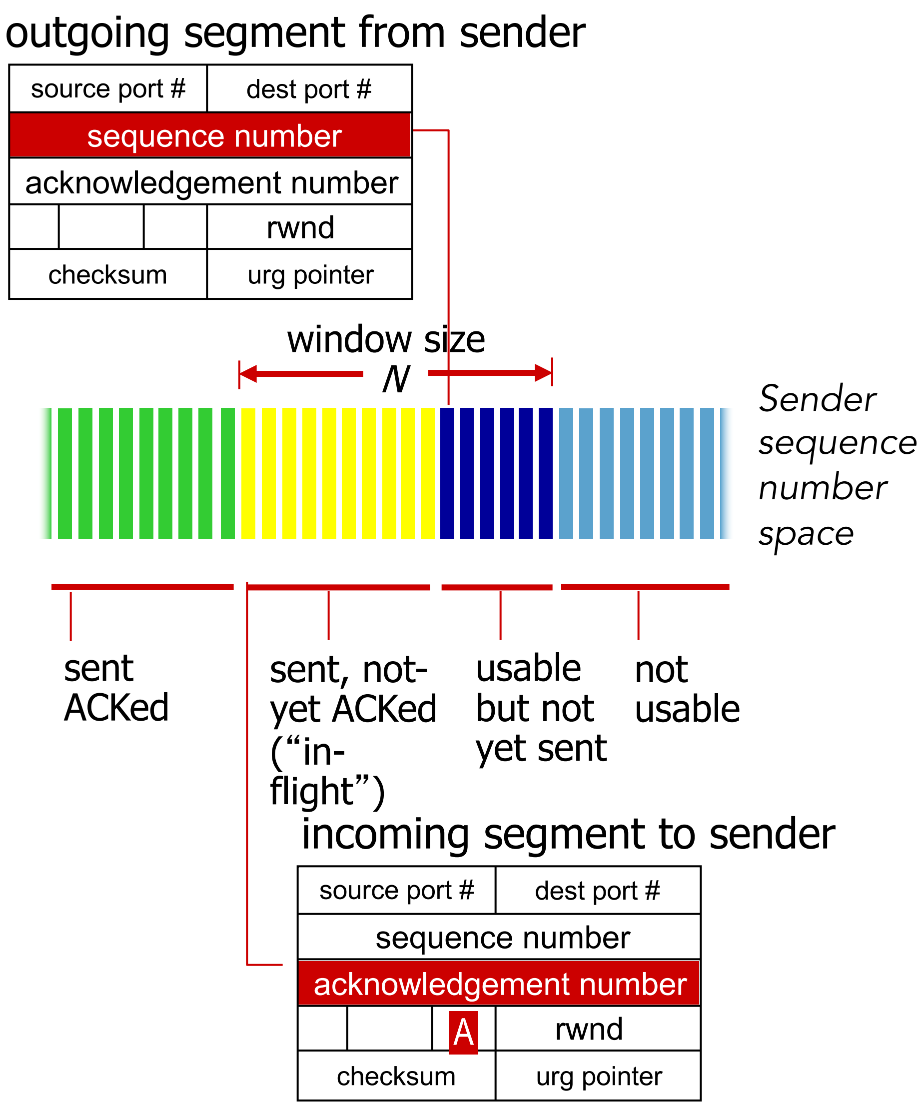
PSH: push data now (generally not used)

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

For flow control # bytes rcvr willing to accept

TCP sequence and ACK numbers

- Before looking at how they are used …
- Sequence numbers
  - Byte stream "number" of first byte in segment's data
- Acknowledgements
  - Seq # of next byte expected from other side, cumulative ACK
- *How receiver handles out-of-order segments?*
  - TCP spec doesn't say, up to implementor



TCP sequence numbers, ACKs

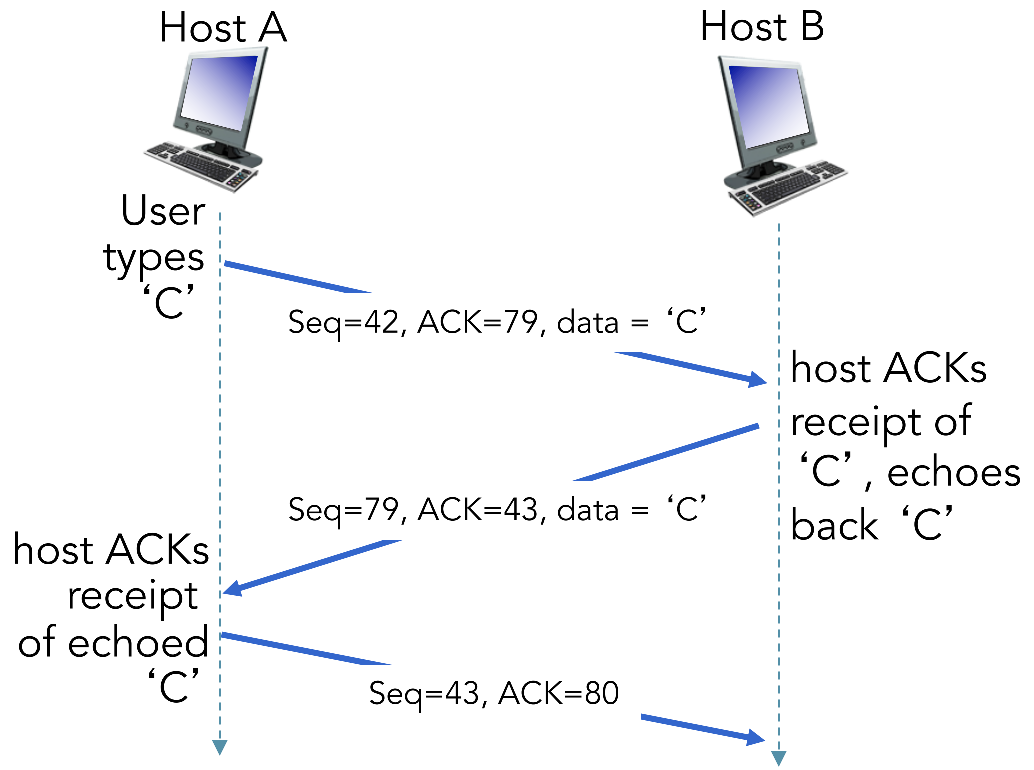
Both ends start with randomly chosen seq # – 42 and 79 (to avoid mistaken a segment from an older connection for a valid segment for this one)

First segment includes the one-byte 'C'

"I got everything up to byte 42, waiting from 43 onwards"; the message also echoes the 'C'

The ACK is piggybacked on the echo message

No data, just an ACK from the client



simple telnet scenario

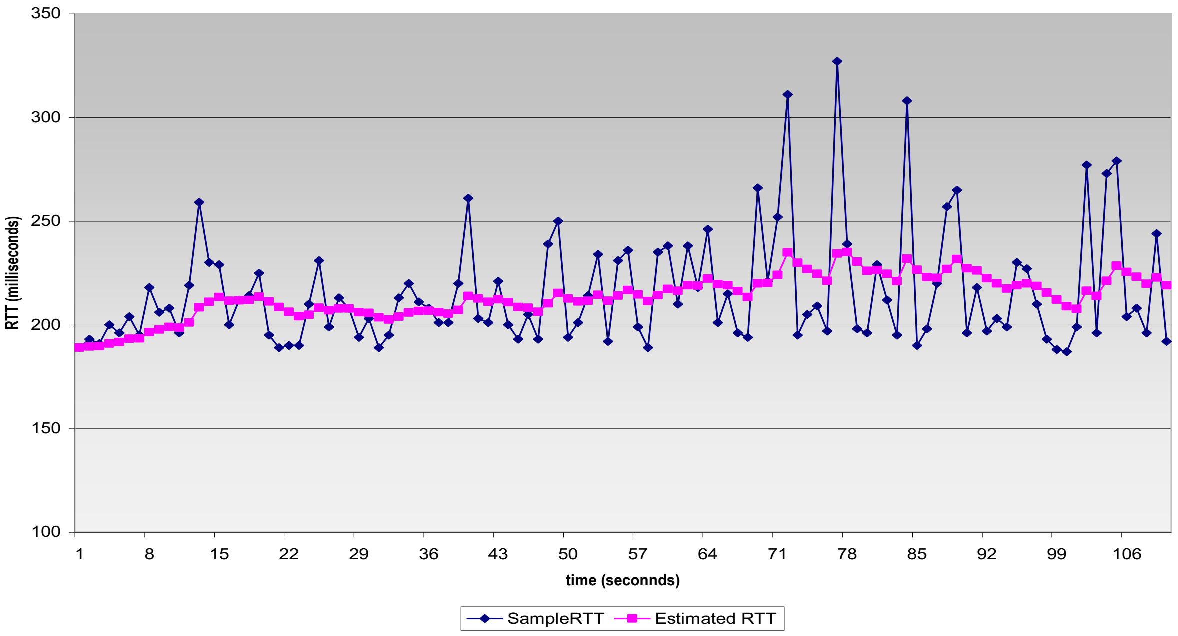
TCP round trip time, timeout

- As RDT, TCP uses timeout/retransmit to recover from loss
- *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
- *How to estimate RTT?*
  - Keep a SampleRTT, time from a segment transmission until ACK receipt
    - So one SampleRTT approximately every RTT
    - Ignore retransmissions, i.e., don’t compute it for a retransmitted segment

TCP round trip time, timeout

- *How to estimate RTT? …*
  - Quite a bit of fluctuation due to congestions and load at end systems
  - To smooth it out, average several recent measurements using an exponential weighted moving average (EWMA)
    - Influence of past sample decreases exponentially fast typical value: α= 0.125

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



TCP round trip time, timeout

- To capture variability

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

(typically, β = 0.25)

  - Note that DevRTT is a EWMA of the difference between EstimatedRTT and SampledRTT
- Timeout interval: EstimatedRTT plus "safety margin"
  - Large variation in EstimatedRTT → larger safety margin
  - Small variation in EstimatedRTT → smaller safety margin

$$\text{TimeoutInterval} = \text{EstimatedRTT} + \underbrace{4*\text{DevRTT}}_{\text{Safety margin}}$$

Initially, TimeoutInterval = 1 sec

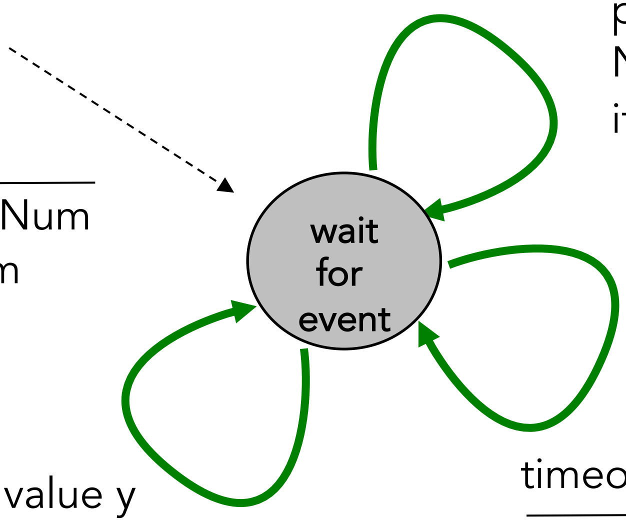
TCP reliable data transfer

- Remember, IP gives you no guarantees on
  - Data delivery | Order | Integrity of the data
- TCP creates rdt service on top of IP's unreliable service
  - Pipelined segments
  - Cumulative acks
  - Single retransmission timer

- Let's start with a simplified version of TCP reliable data transfer
  - Ignore duplicate acks
  - Ignore flow control, congestion control

TCP sender (simplified)

Three major events …



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

One timer, instead of one per transmitted, not-yet-ACK segment

ACK received, with ACK field value y

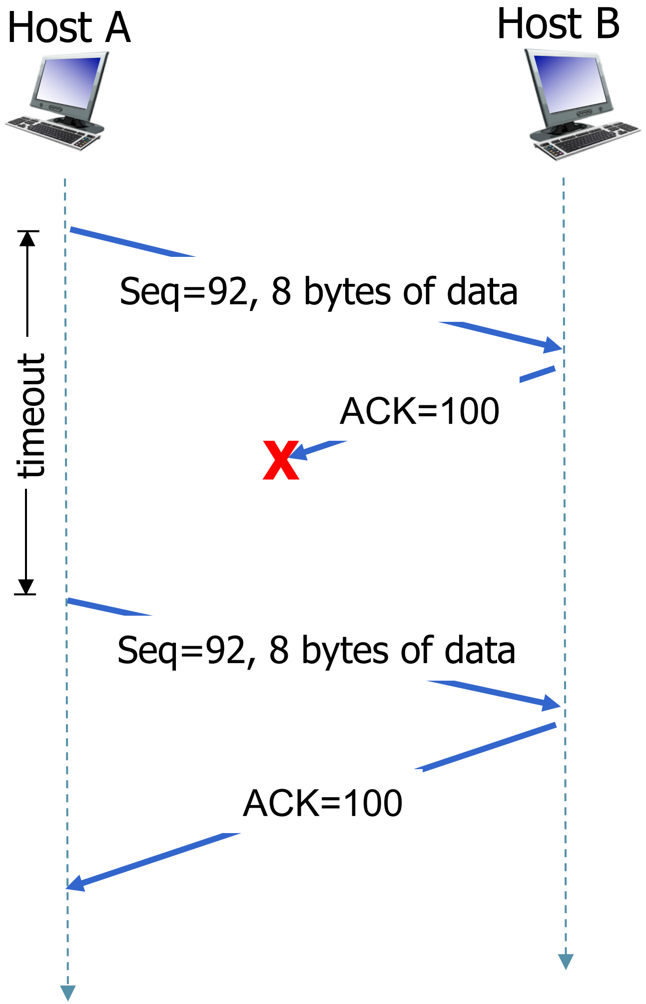
```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
        start timer
   else stop timer
}
```

timeout

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

Expiration time is TimeoutInterval we just calculated

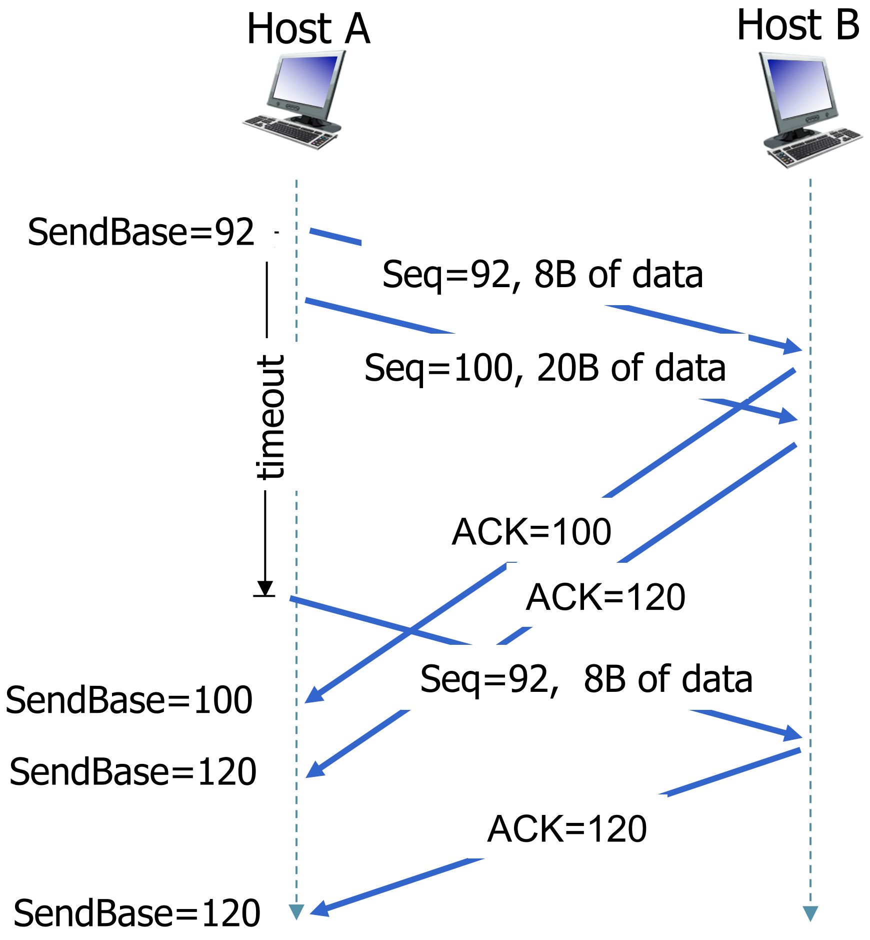
TCP retransmission scenarios – Lost ACK



A sends a segment with seq # 92 and 8B of data to *B*

ACK never makes it back, timeout and *A* retransmits

TCP retransmission scenarios – Premature timeout

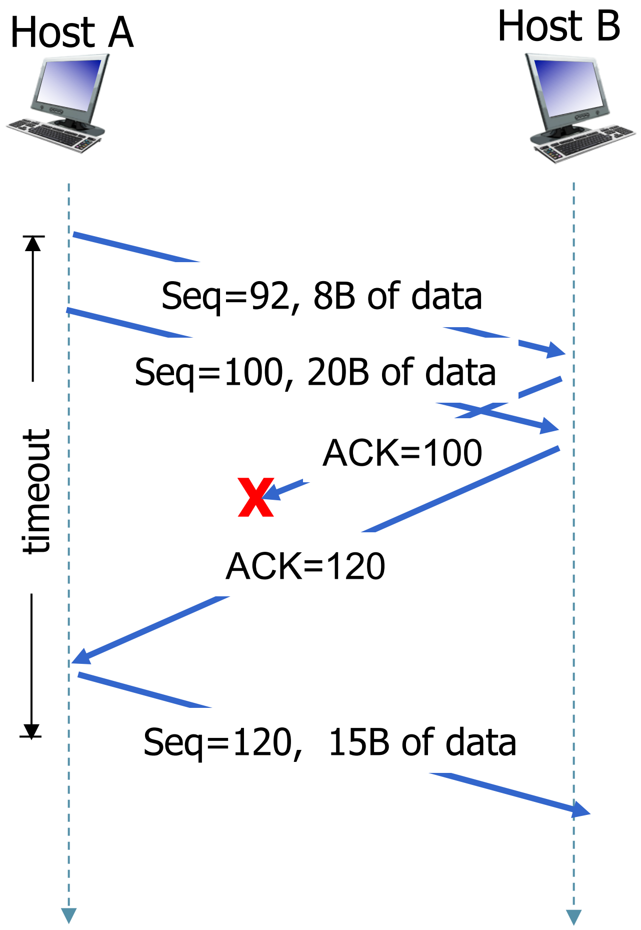


A sends two segments back to back with seq # 92 and 100

It retransmit the first segment because of the timeout

If the second segment's ACK arrives before the new timeout, the segment will not be retransmitted

TCP retransmission scenarios – Cumulative ACK



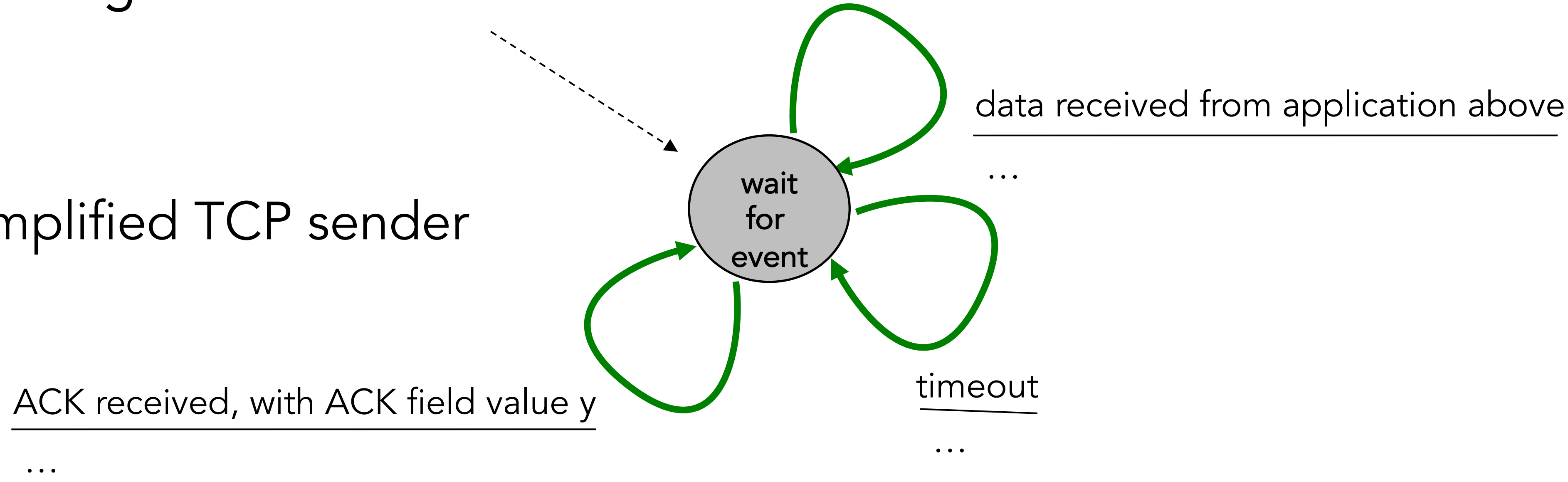
A sends two segments back to back with seq # 92 and 100

First ACK is lost but second ACK arrives before the timeout; *A* knows that *B* received everything up through byte 119, no need to resend

Timeout re-setting

- Upon a timeout, double the timeout interval instead of using `EstimatedRTT` and `DevRTT`
  - So timeout grows exponentially with retransmission
  - When timer is restarted by the other events (data from above, ACK received), `TimeoutInterval` is set based on recent estimation of RTT
  - A form of congestion control, as delays and so timeouts are likely caused by congestion

Our simplified TCP sender



TCP fast retransmit

- Timeout period can be relatively long
  - Longer to resend, increased end-to-end delay
- Sender can 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

To see that, look at what the receiver does with ACKs

TCP ACK generation [RFC 1122, RFC 2581]

- There are no NACKs, the duplicate ACK indicates the gap

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-expected 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

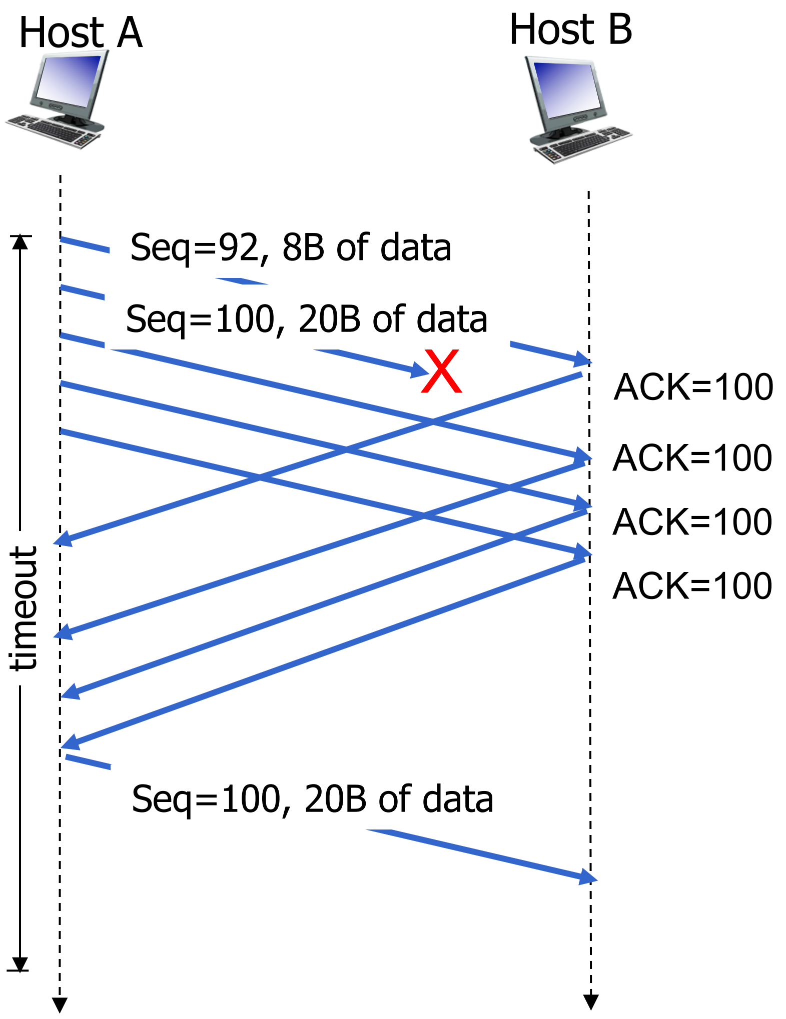
TCP fast retransmit

- Timeout period can be relatively long
  - Longer to resend, increased end-to-end delay
- Sender can 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
  - Wait for three duplicate ACKs …

Fast retransmit: retransmit the missing segment *before* the segment's timer expires

```
Event: ACK received, with ACK field value of y
       if (y > SendBase) {
          SendBase = y
          if (there are currently any not yet ACK'ed segments) start time
       } else { /* a duplicate ACK for already ACKed segment */
         Increment number of duplicate ACKs received
         if (number of duplicate ACKs for y == 3)
           resend segment with seq number y /* fast retransmit */
       }
```

TCP fast retransmit



Fast retransmit after sender receipt of triple duplicate ACK

TCP … Go-Back-N or Selective Repeat

Go-Back-N

- Only one timer is kept, but →
- Send cumulative ACKs, but →
- Duplicate ACK for early segment

Selective Repeat

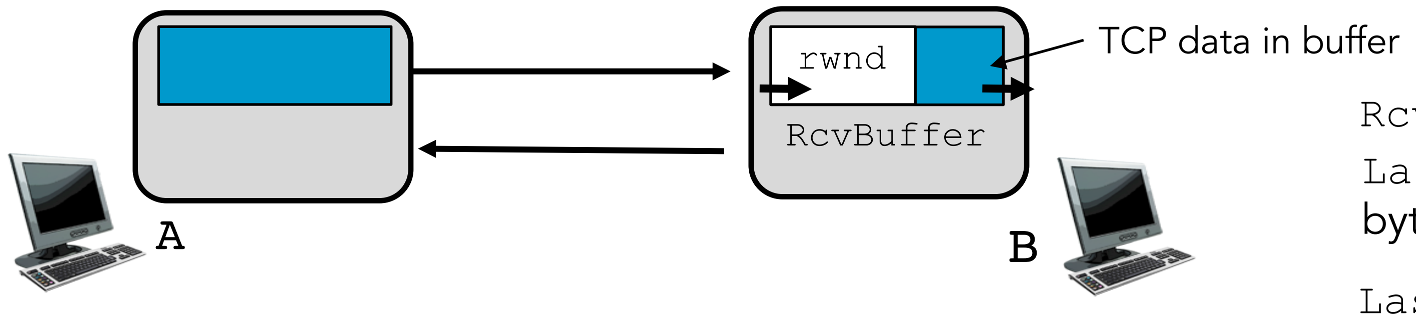
- Re-send just one segment on timeout
- Receiver may save out-of-order segments for later reassembly

- Plus some new features
  - Guidelines for setting timeout interval, based on observations
  - Delayed ACKs
  - Triple duplicate ACK triggers a retransmit.
  - Connection setup with 3-way handshake, and teardown
  - Window size changes to implement flow & congestion control

TCP flow control

- Bot sides of a TCP connection set aside a receive buffer
  - Arrived data goes there until it is read
  - If the app doesn’t read fast enough, sender can overflow the buffer
- *Flow control* – a speed-matching service
  - Receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - Like congestion control but different motivation
  - Of course, nothing like it in UDP
- *How?* The idea
  - Sender keeps a *receive window* (`rwnd`) variable, an approximation of free buffer space in the receiver (full-duplex, so both ends have one)
  - Sender makes sure to have less than `rwnd` sent and un-acknowledge

TCP flow control



`RcvBuffer`: Size of buffer

`LastByteRead`: Number of the last byte read by the app from the buffer

`LastByteRcvd`: Number of the last byte arrived and placed in the buffer

`LastByteSent`
`LastByteAcked`

`LastByteSent – LastByteAcked`:
the amount of unack'ed data sent by A

`LastByteRcvd – LastByteRead ≦ RcvBuffer`
`Rwnd = RcvBuffer – [LastByteRcvd – LastByteRead]`
Host B tells A of `rwnd` in every segment it sends back

Host A keeps `LastByteAcked – LastByteAcked ≦ rwnd`

One more detail, what happens after B runs out of space, advertises it and has nothing to send to A? A will never know when B empties the buffer …

TCP specs require A to keep sending one data byte segments when B's rwnd is zero

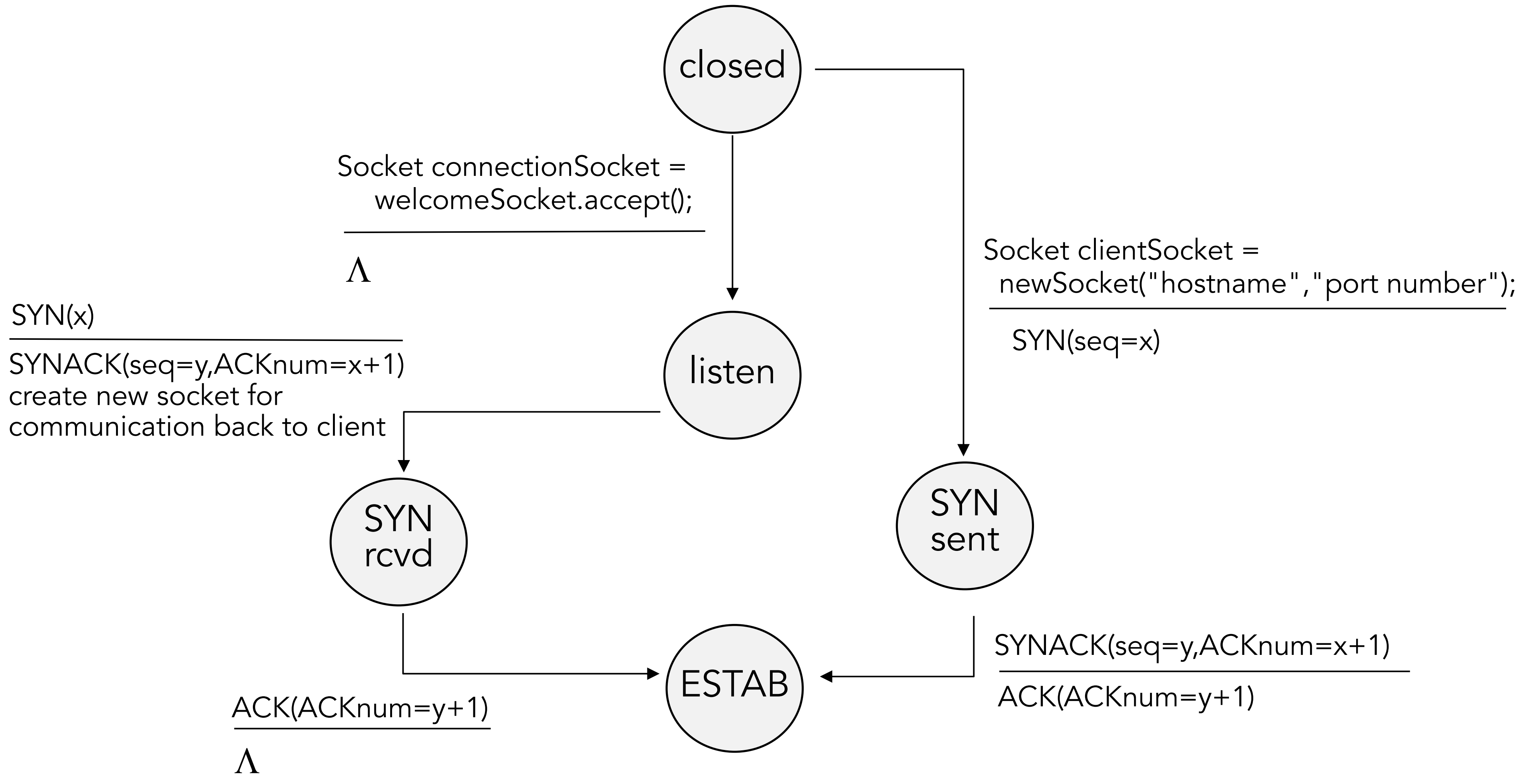
TCP connection management

- How a TCP connection is established and torn down
  - Can add to perceived latency
  - Can be exploited for attacks
- Steps

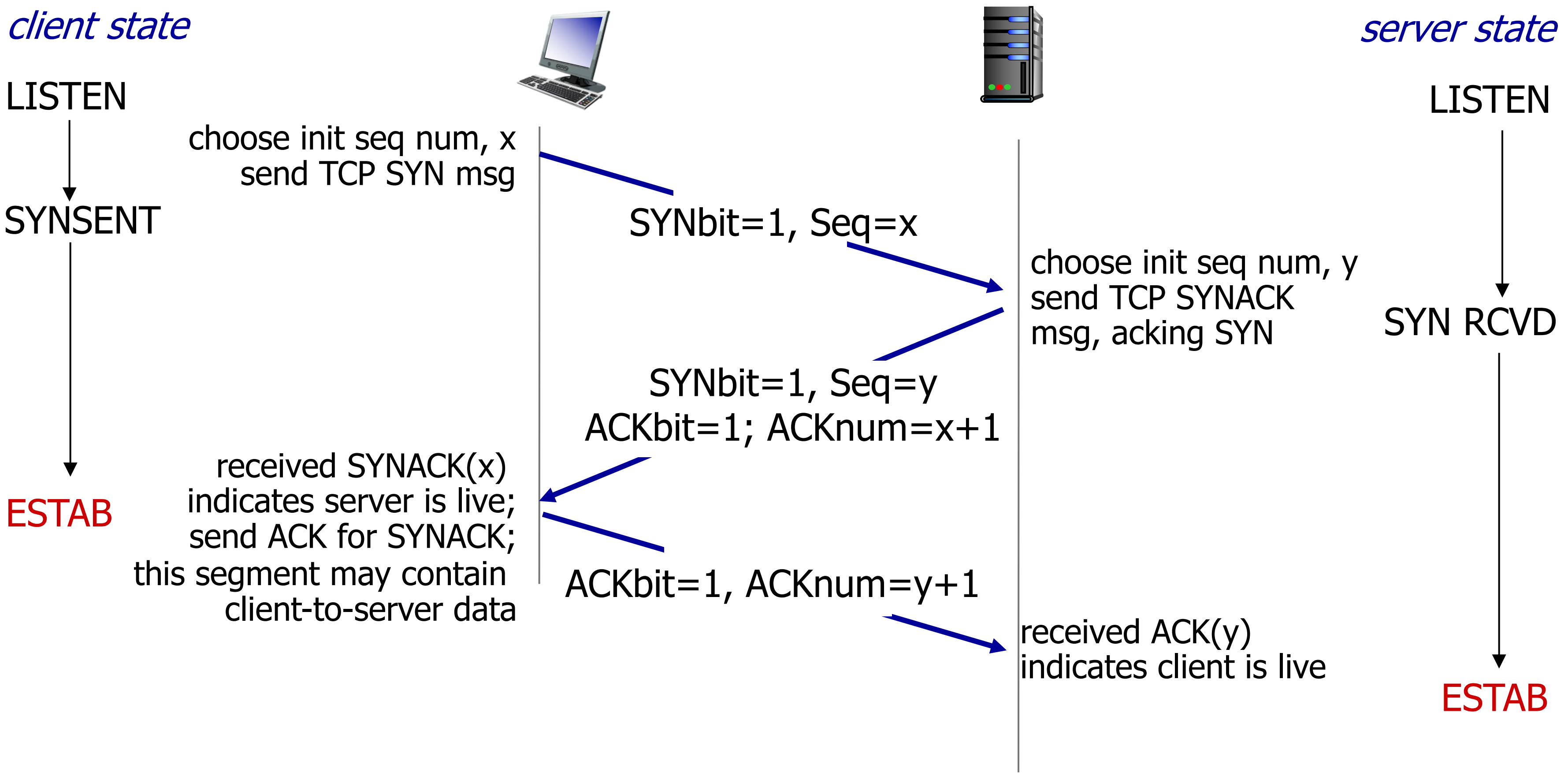
A three-way handshake

1. Client-side sends a special SYN segment, no app data, SYN-bit set, includes initial sequence number (`client_isn`) in seq #
2. Server receives SYN, allocates TCP buffers and variables to the connection, and sends a SYNACK connection-granted segment. Here SYN bit is set, ACK field has `client_isn+1`, server chooses its own initial seq number (`server_isn`) and puts it in the seq # field
3. On receiving SYNACK, client allocate buffers and variables, and sends an ACK segment (`putting server_isn+1` in ACK) that could include app data (SYN bit is zero)

TCP 3-way handshake: FSM

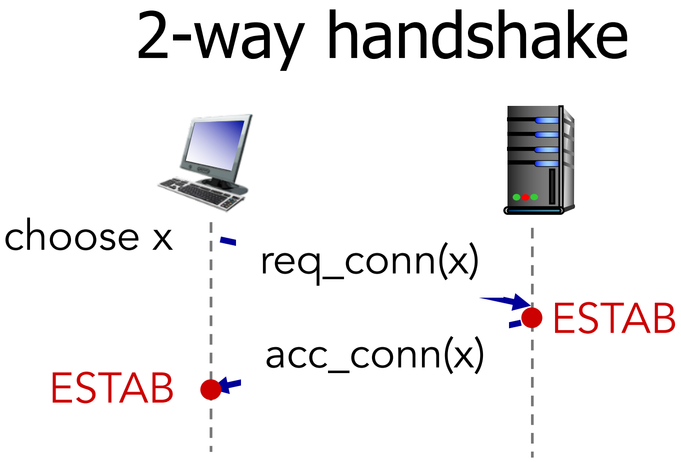


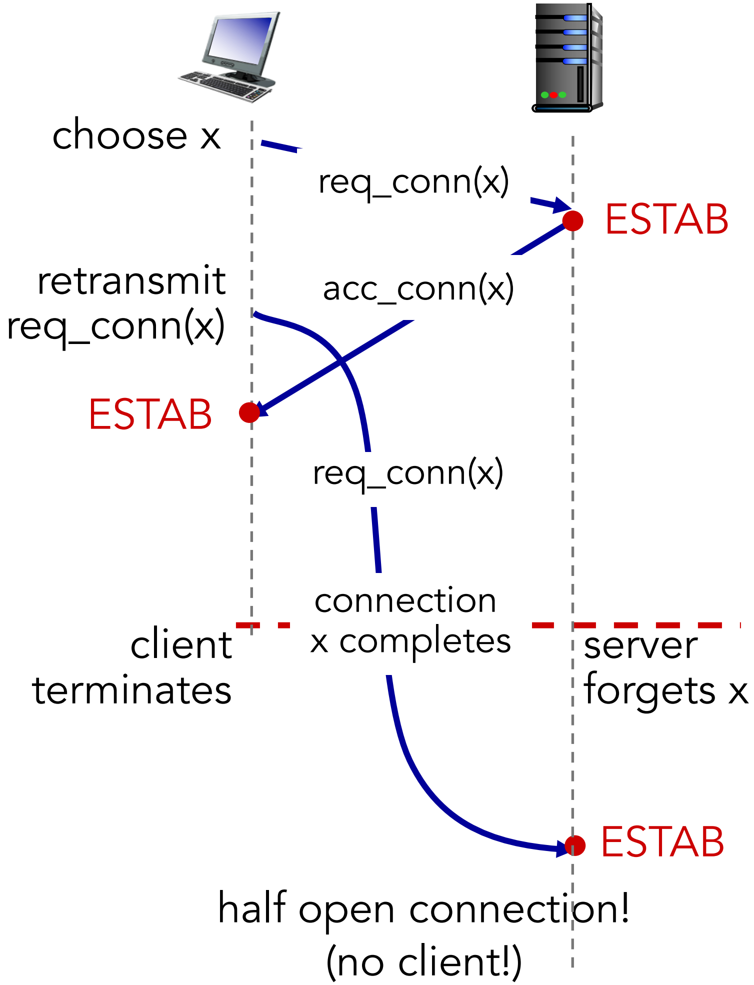
TCP 3-way handshake



Would 2-way handshake have worked?

- 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

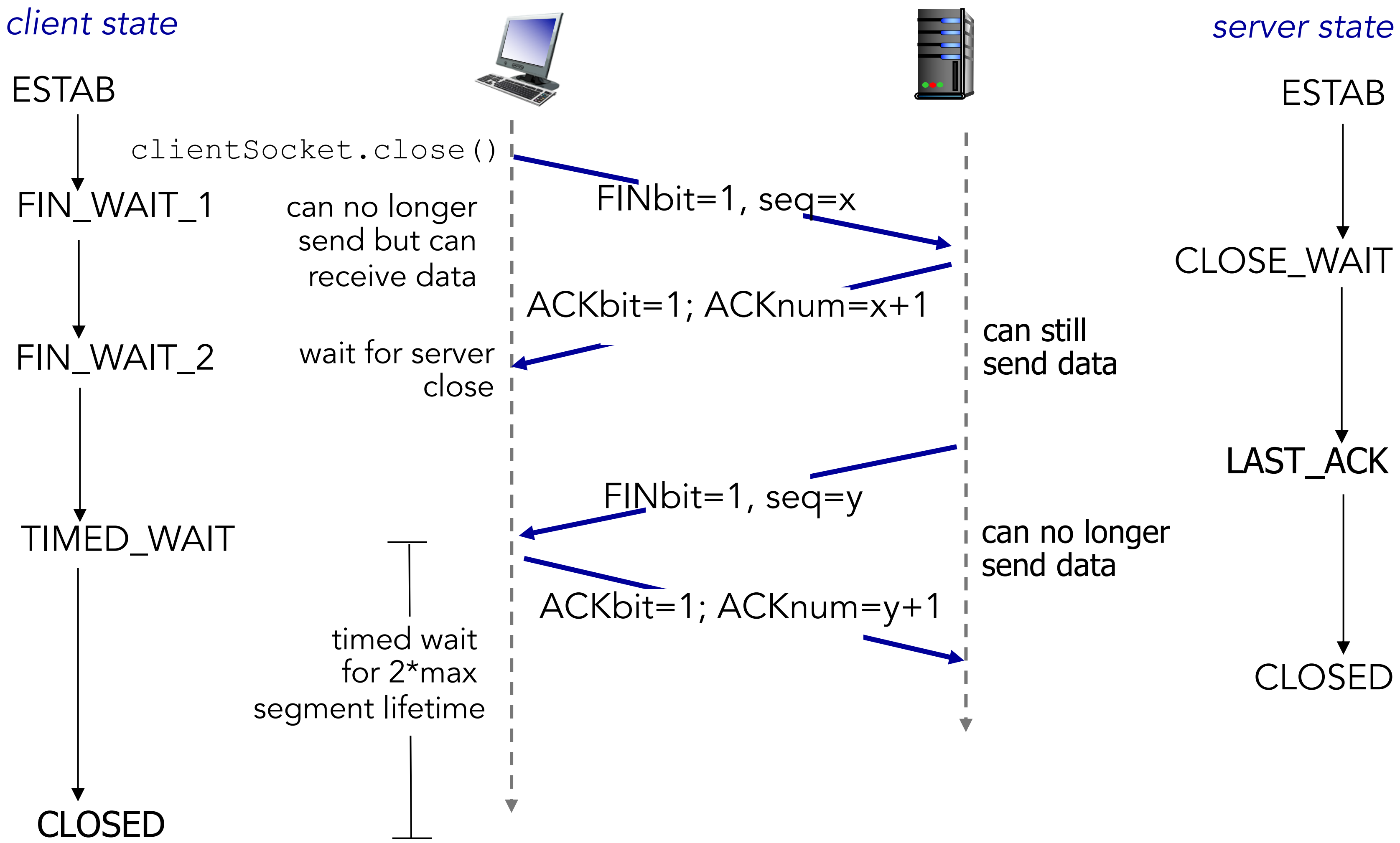




TCP: closing a connection

- 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

Closing a TCP connection



Recap

- TCP implements a combination of GBN and Selective Repeat
- ACK timeout can be appropriately set with EWMA of recent RTT
- Connection setup requires a 3-way handshake
- Flow control is implemented with explicit Receive Window