Transmission Control Protocol (TCP)

CS 352, Lecture 8

http://www.cs.rutgers.edu/~sn624/352-S19

Srinivas Narayana

(slides heavily adapted from text authors' material)

RUTGERS
UNIVERSITY | NEW BRUNSWICK

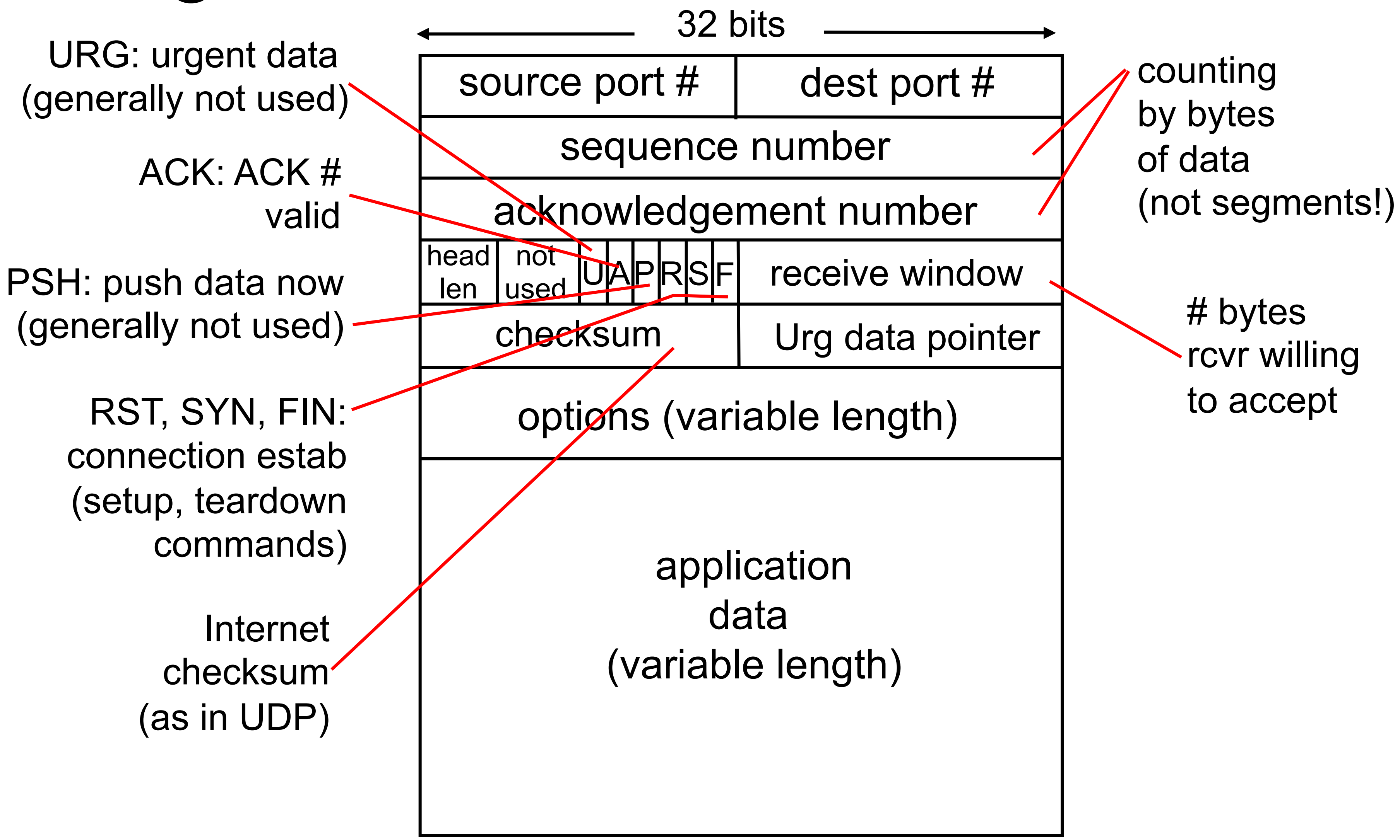
Transmission Control Protocol: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - 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

RFCs: 793,1122,1323, 2018, 2581

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

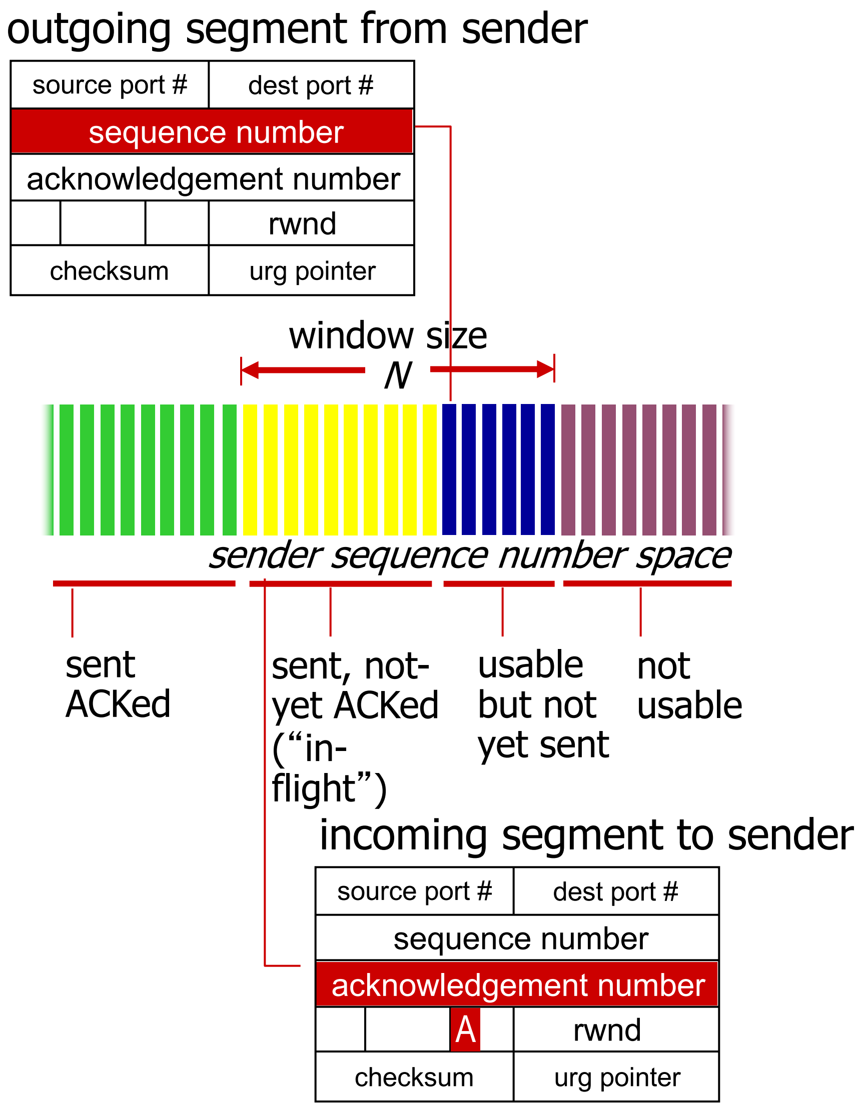
- byte stream "number" of first byte in segment' s data

acknowledgements:

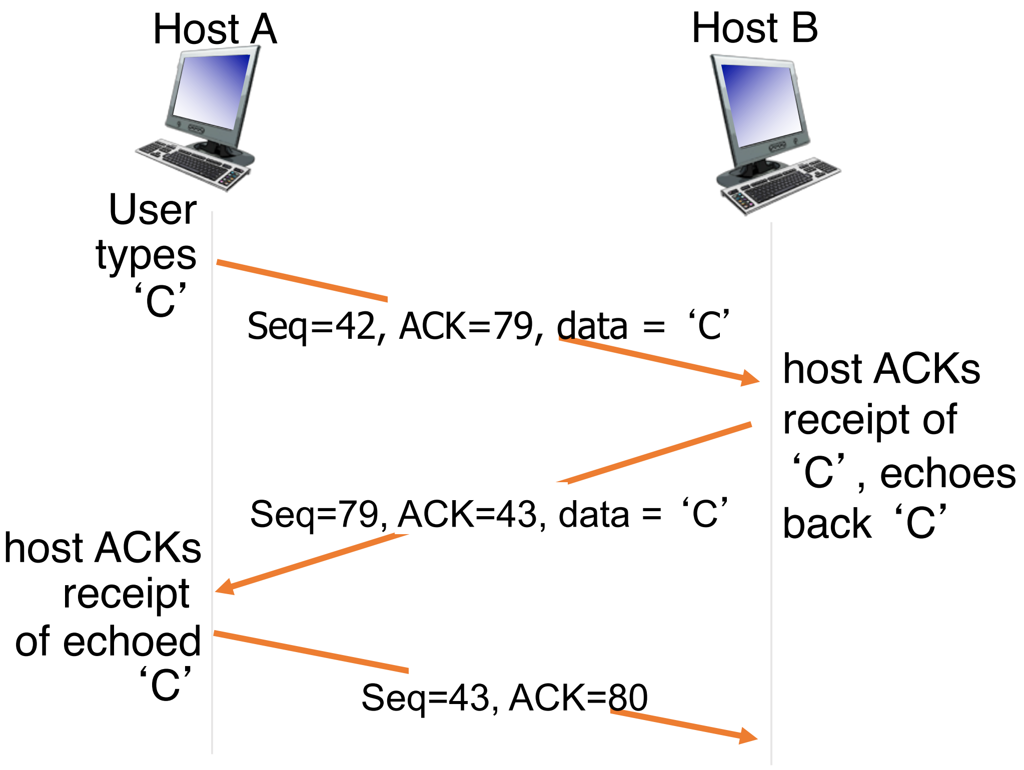
- seq # of next byte expected from other side
- cumulative ACK

How does receiver handle out-of-order segments?

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



TCP seq. numbers, ACKs



simple telnet scenario

Reliable transfer in TCP

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, congestion control

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

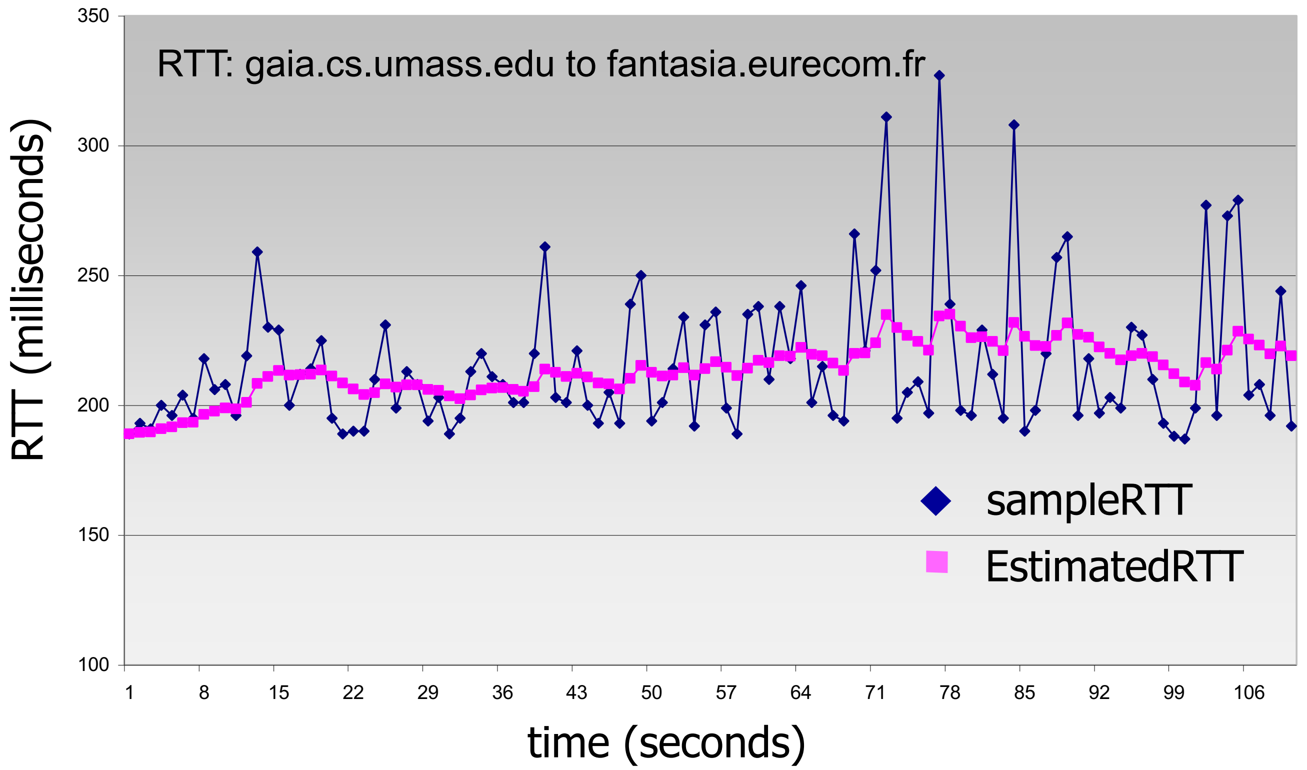
Q: how to estimate RTT?

- `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:

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

              (typically, β = 0.25)
```

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

estimated RTT

“safety margin”

TCP sender events: Managing a single timer

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`

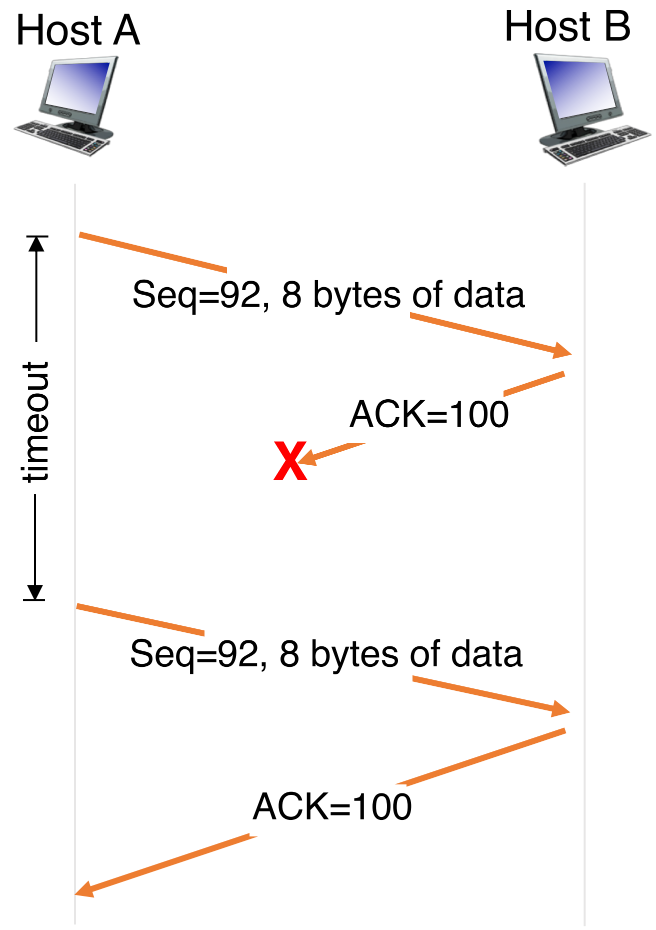
timeout:

- retransmit segment that caused timeout
- restart timer

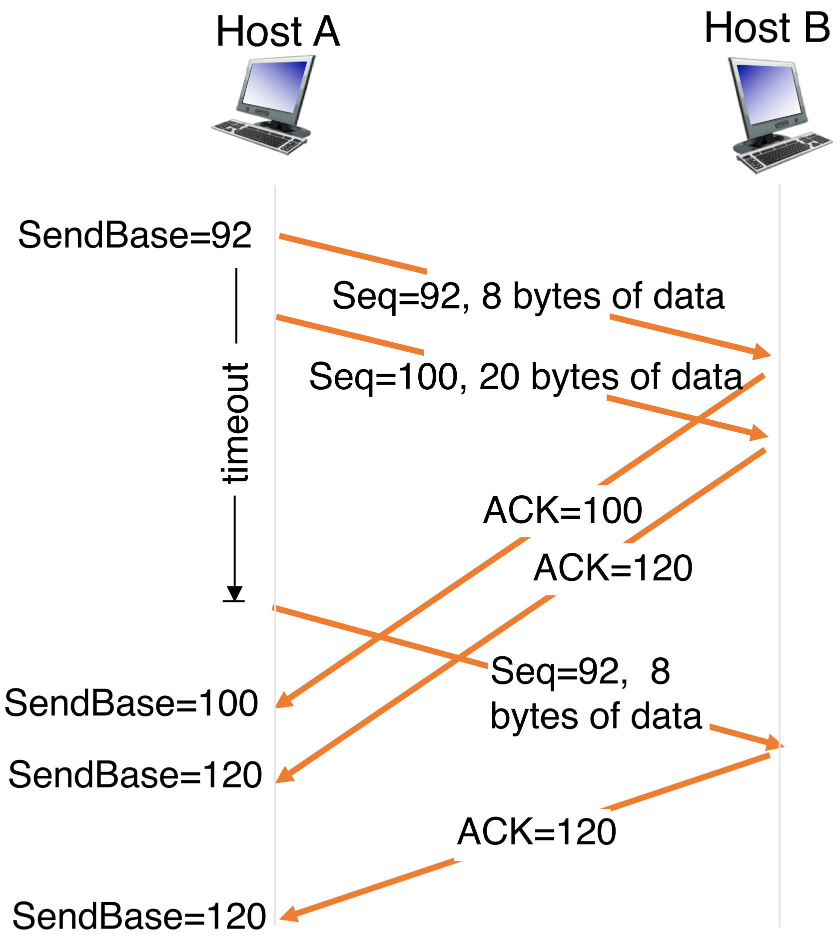
ack rcvd:

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

TCP: retransmission scenarios

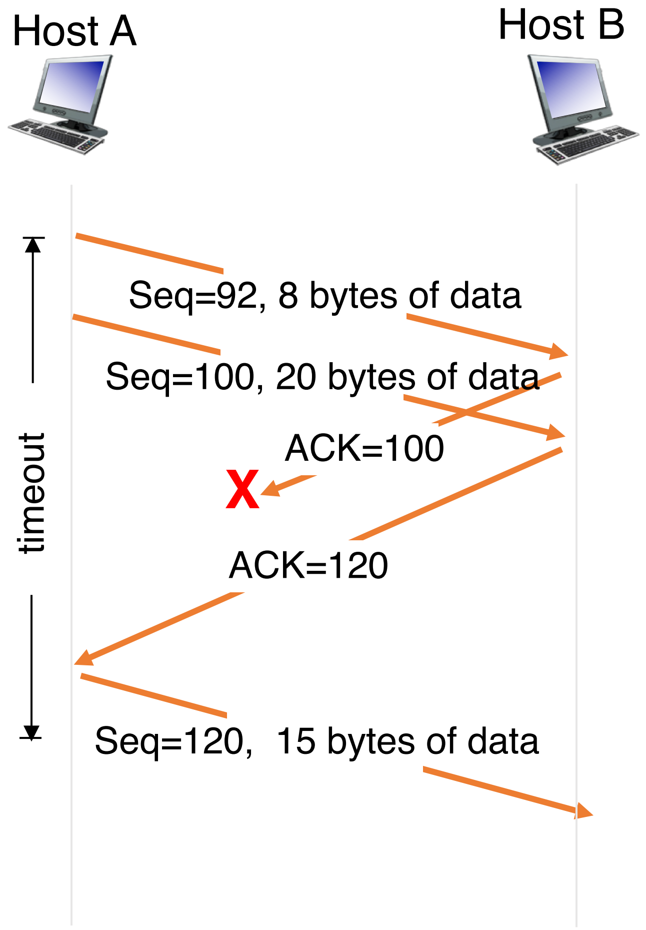


lost ACK scenario



Premature timeout
But segment 120 not transmitted

TCP: retransmission scenarios



Cumulative ACK avoids retransmission altogether

TCP receiver events: ACKing [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

- timeout period often relatively long:
  - long delay before resending lost packet
- Instead: 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

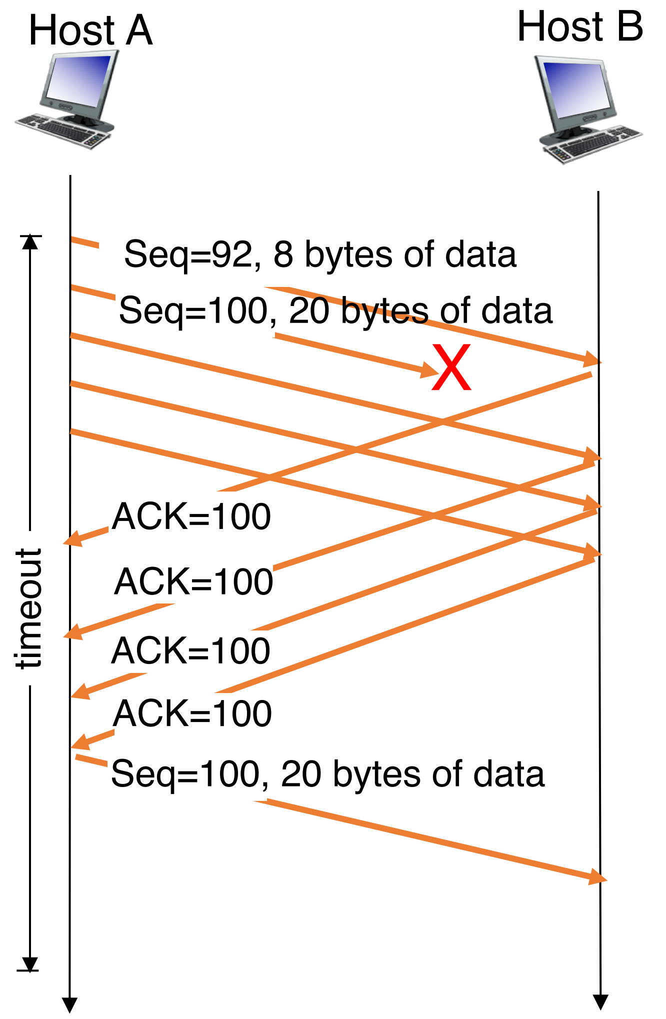
if sender receives 3 ACKs for same data

("triple duplicate ACKs"),

resend unacked segment with smallest seq #

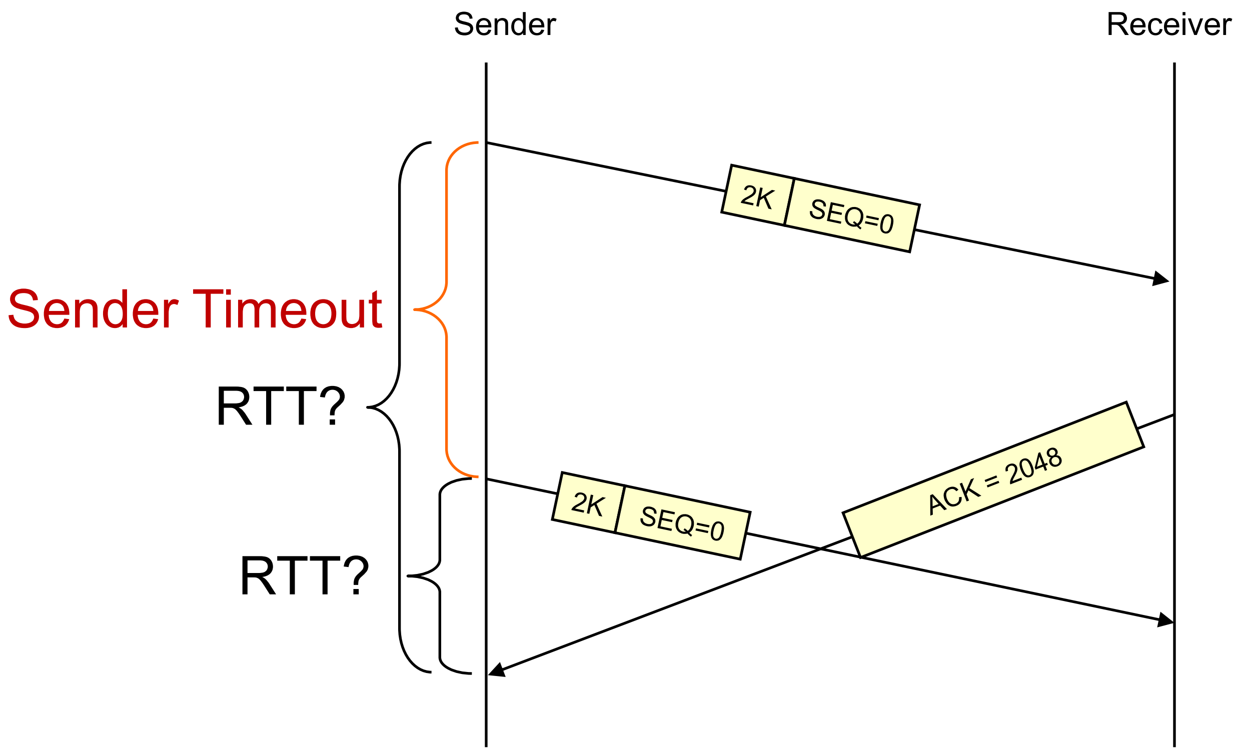
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender
receipt of triple duplicate ACK

Problem with RTT Calculation



Karn’s algorithm

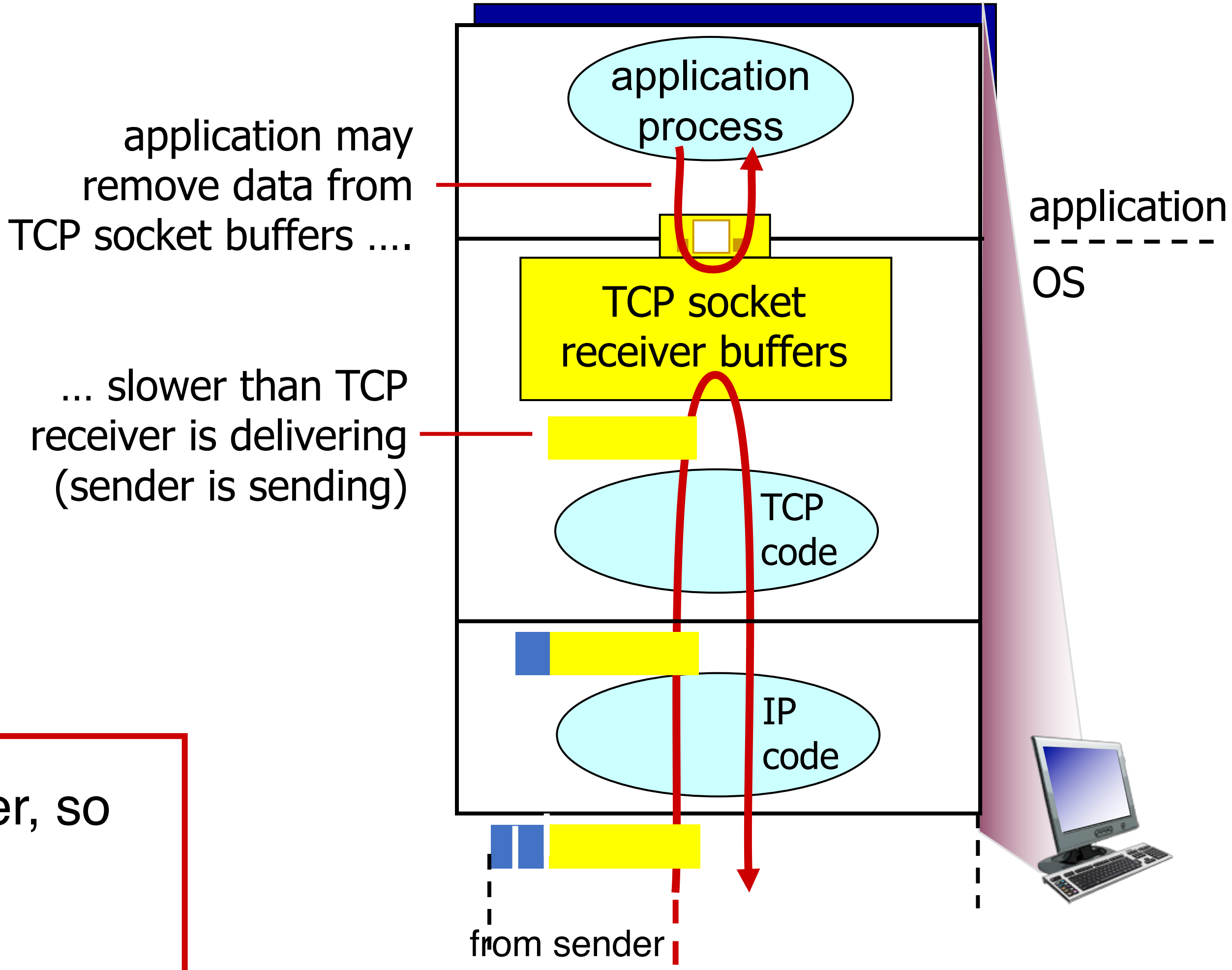
- Retransmission ambiguity
  - Measure RTT from original data segment
  - Measure RTT from most recent segment
- Either way there is a problem in RTT estimate
- One solution
  - Never update RTT measurements based on acknowledgements from retransmitted packets
- Problem: Sudden change in RTT can cause system never to update RTT
  - Primary path failure leads to a slower secondary path

Karn’s algorithm

- Use back-off as part of RTT computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from SRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from SRTT

Flow Control

TCP flow control



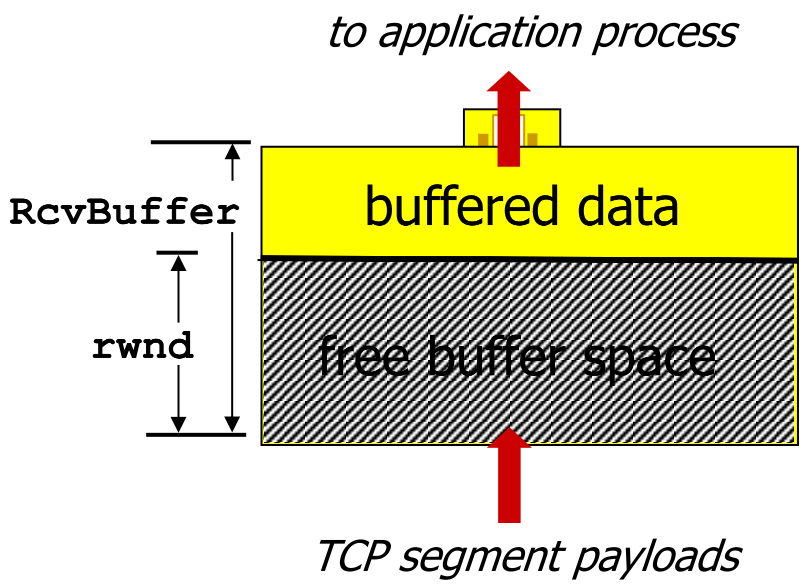
receiver protocol stack

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 autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



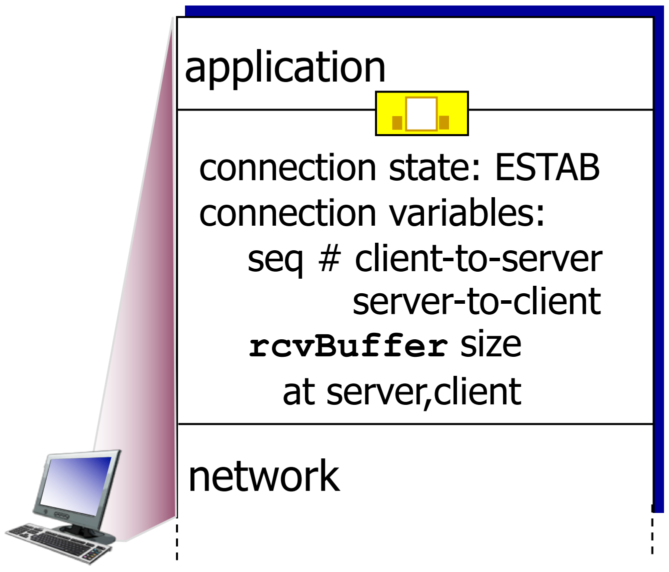
receiver-side buffering

Connection Management

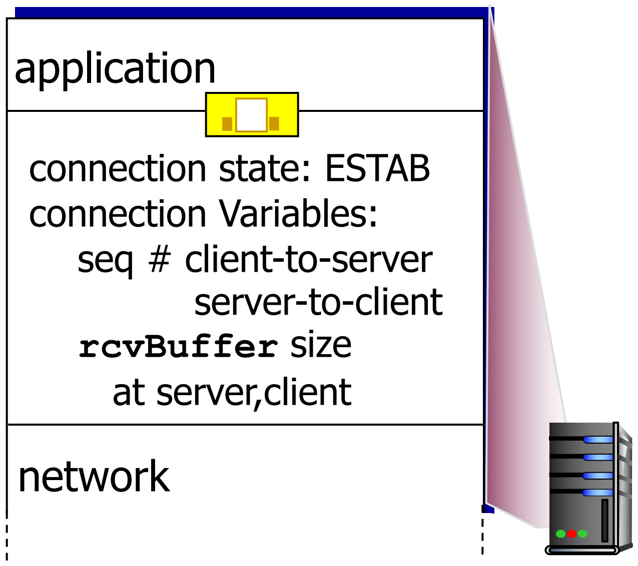
Connection Management

before exchanging data, sender/receiver "handshake":

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



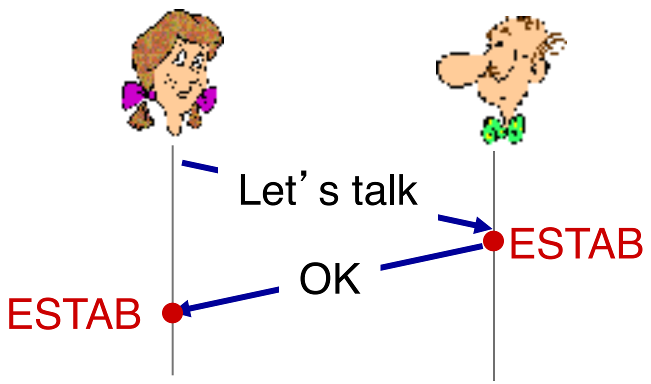

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

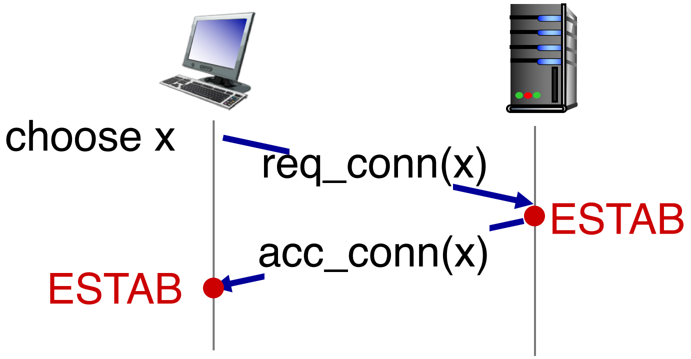



```
Socket connectionSocket =
   welcomeSocket.accept();
```

Agreeing to establish a connection

2-way handshake:



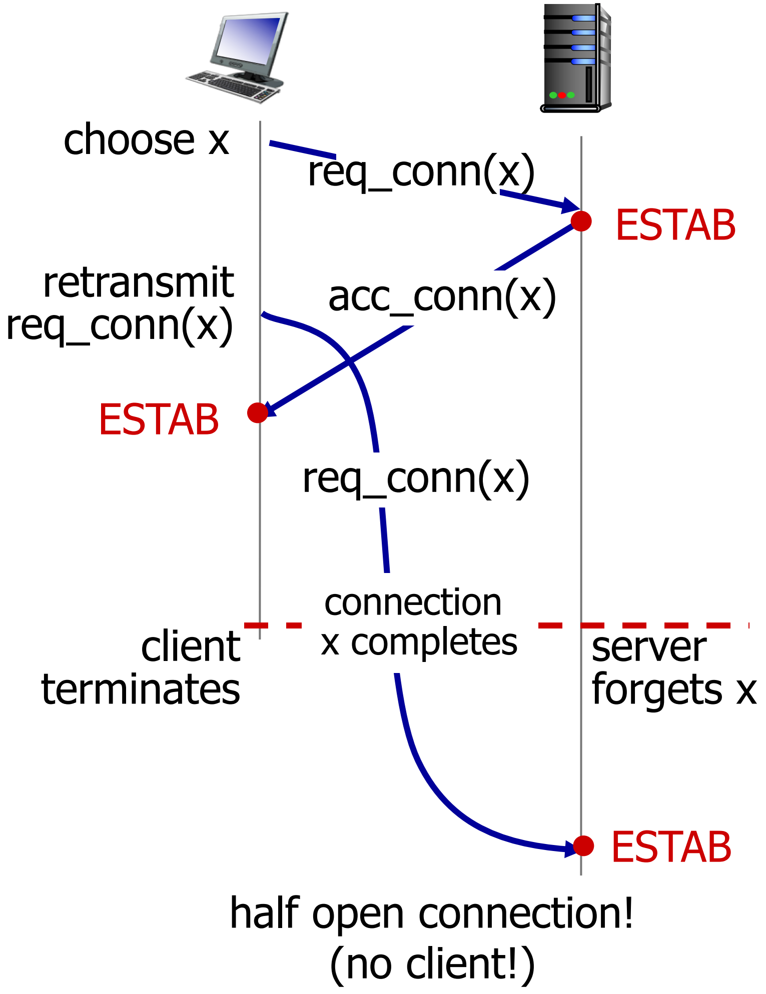


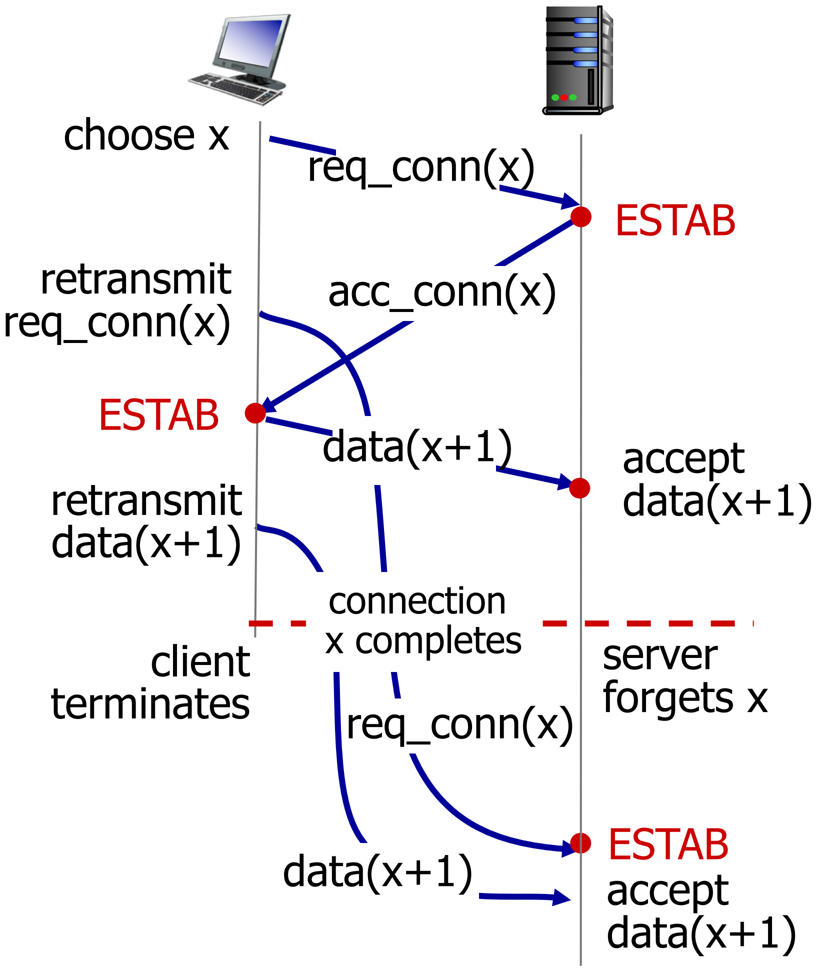
Q: will 2-way handshake always work in network?

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

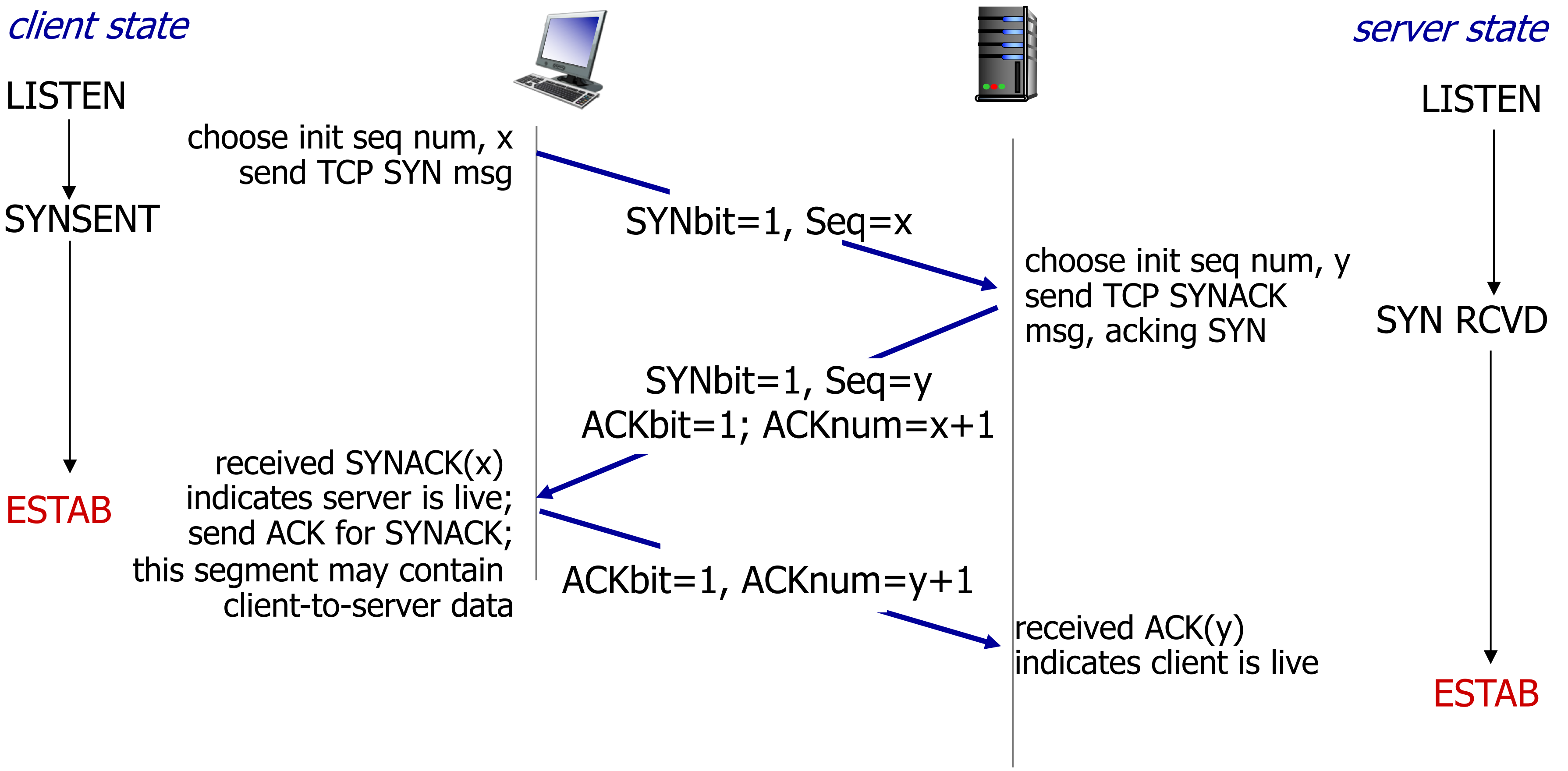
Agreeing to establish a connection

2-way handshake failure scenarios:

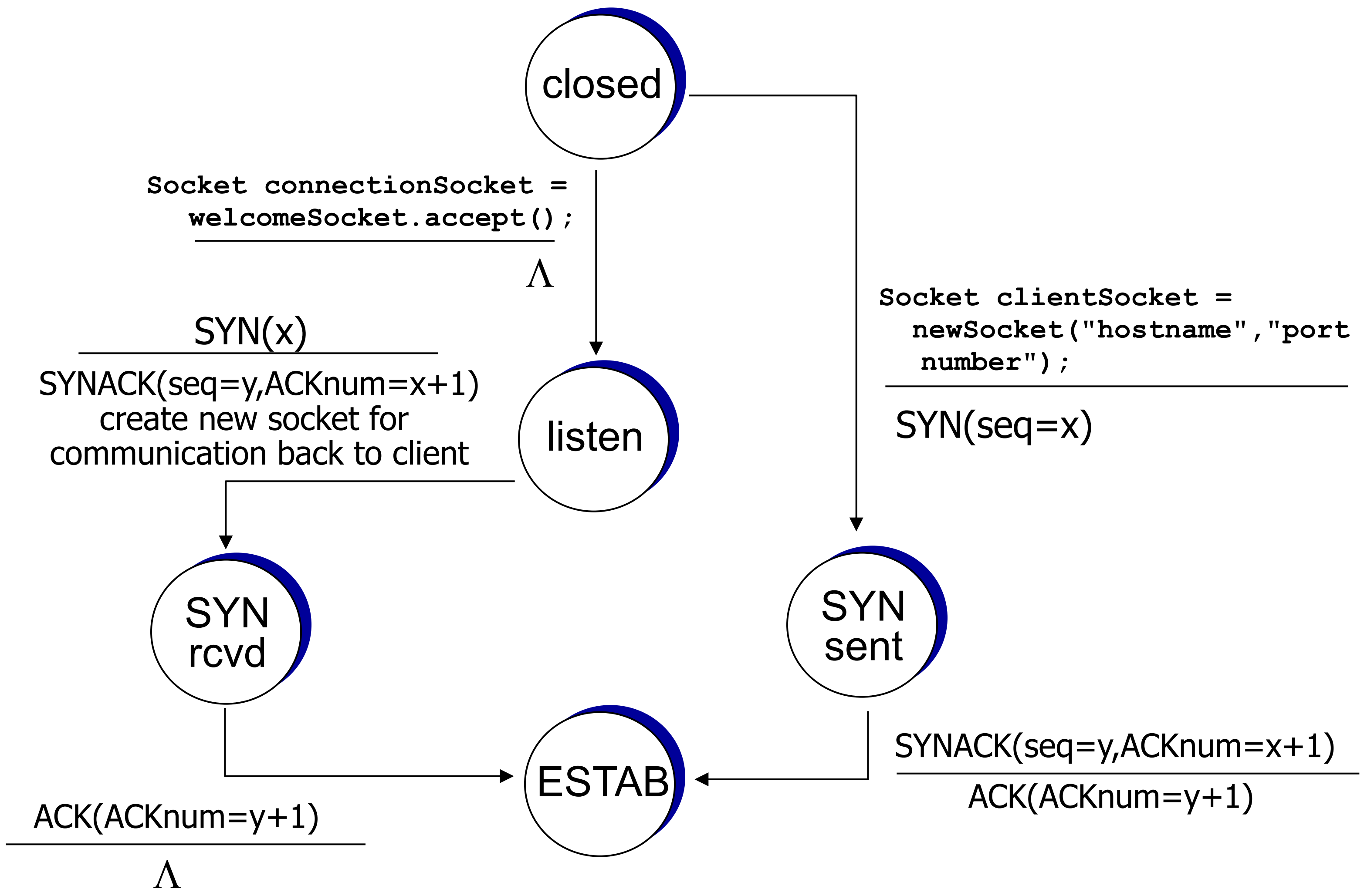




TCP 3-way handshake



TCP 3-way handshake: 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