# CMPE 150/L : Introduction to Computer Networks

Chen Qian
Computer Engineering
UCSC Baskin Engineering

Lecture 9

# Logistics

Reminder: Midterm 2/16 (next Thursday)

TA Extra Office Hours:

Yalda: Friday 2/10 @ 2pm – 4pm
Room: E2-215

Cole: Monday 2/13 @ 5:30pm – 7pm
Room: BE-312C/D

Come prepared -- Bring Questions!

# 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

# Pipelined protocols: overview

## Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

# 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

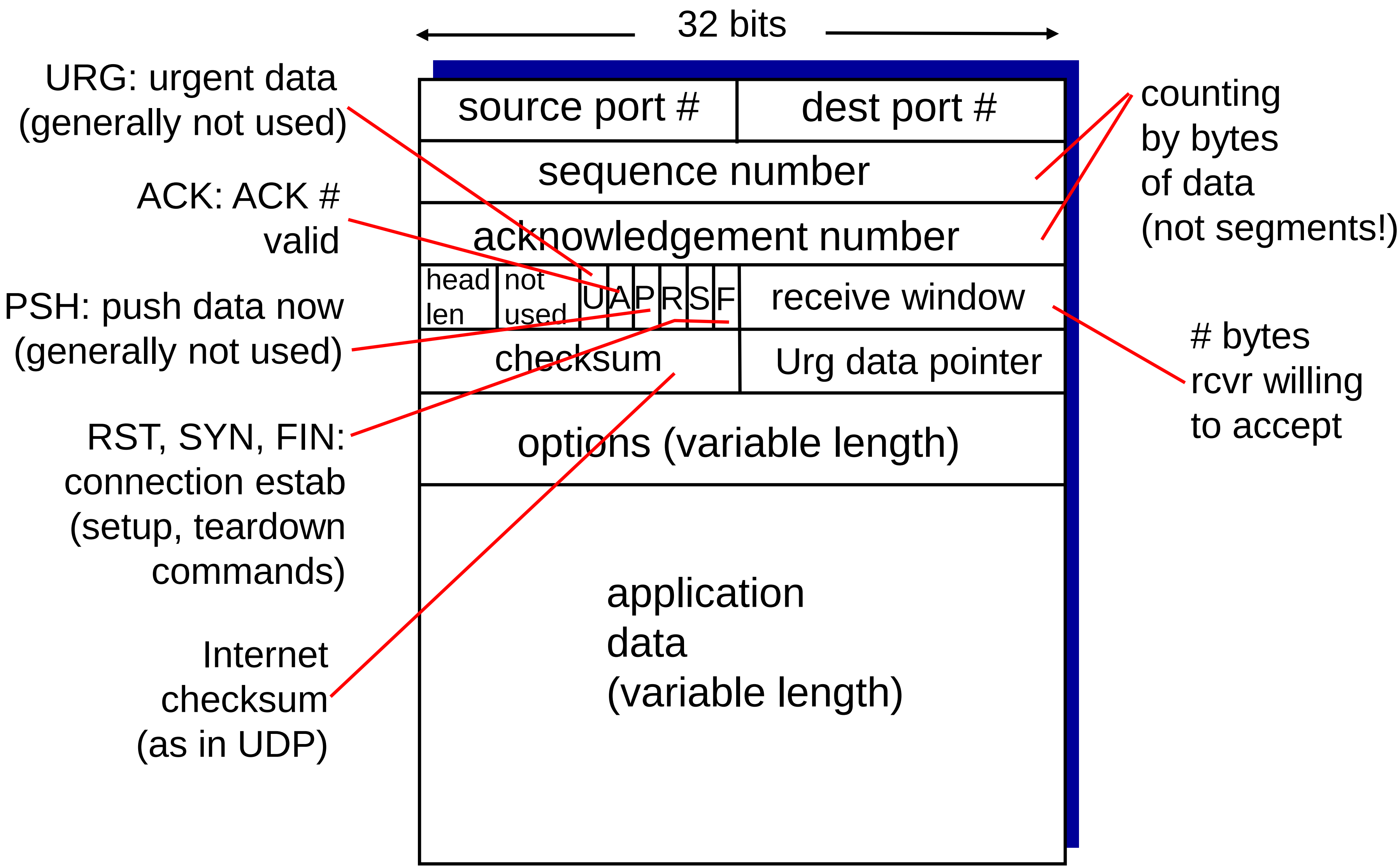
3.6 principles of congestion control

3.7 TCP congestion control

# TCP: Overview RFCs: 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



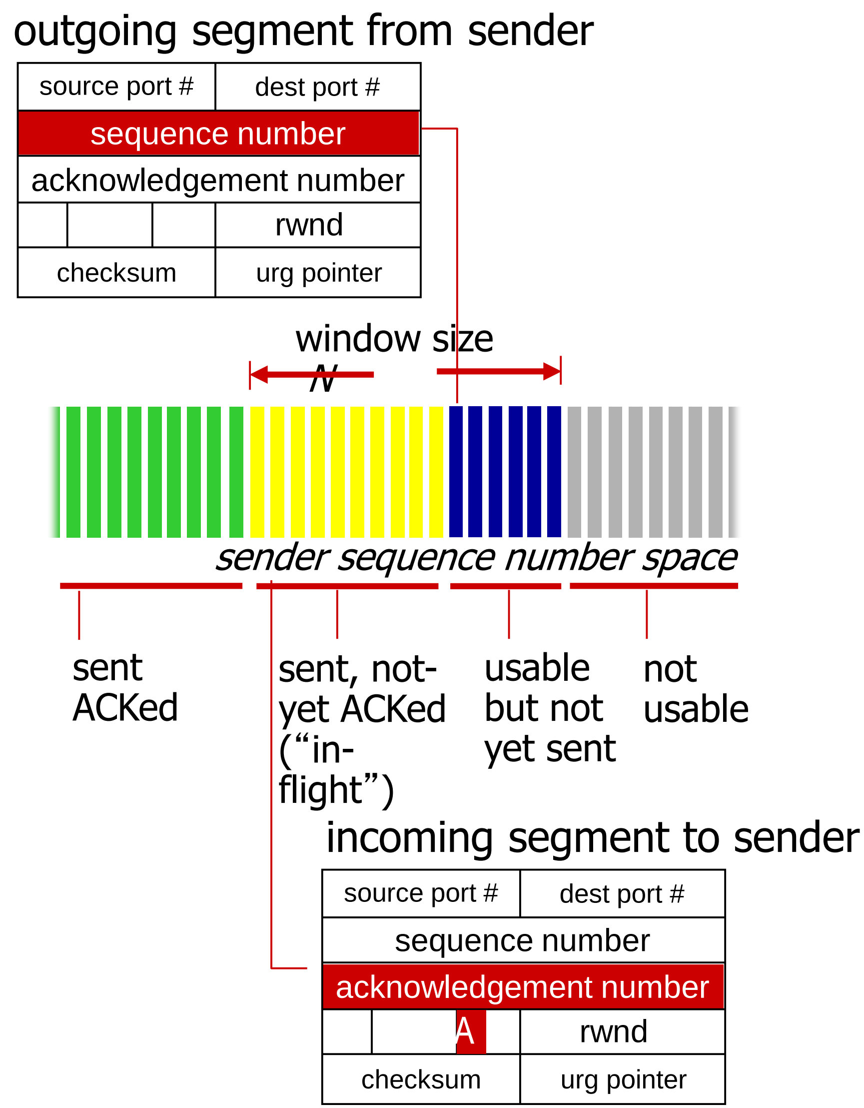
# TCP seq. numbers, ACKs

sequence numbers:

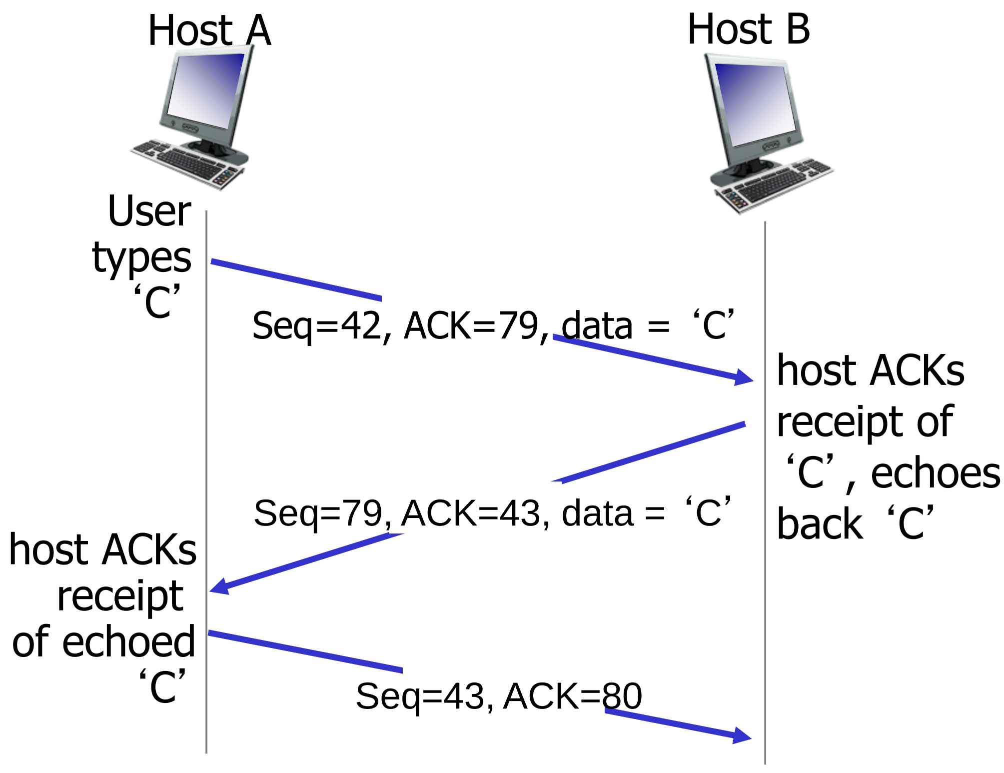
- byte stream "number" of first byte in segment's data

acknowledgements:

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



# 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

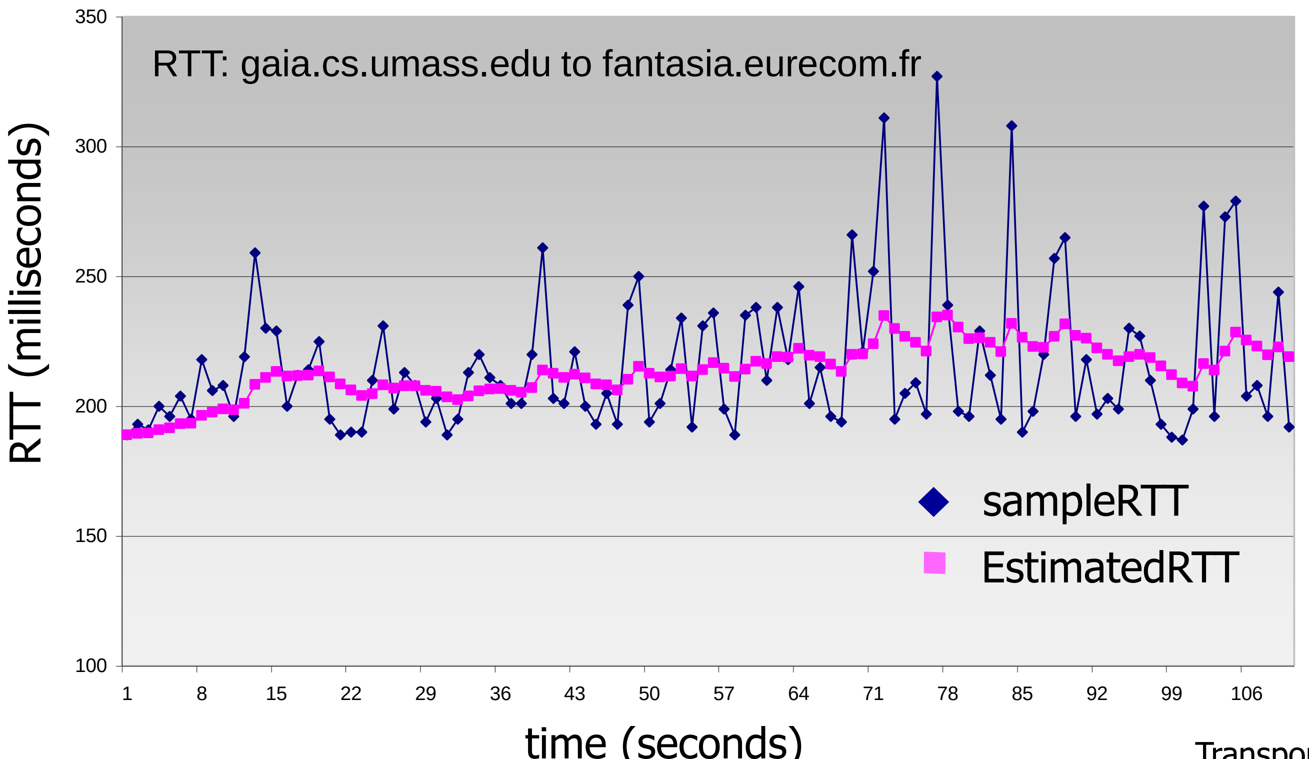
Q: how to estimate RTT?

- **`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: $\alpha = 0.125$



# TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + \underbrace{\texttt{4*DevRTT}}_{\text{“safety margin”}}$$

# 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

# 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

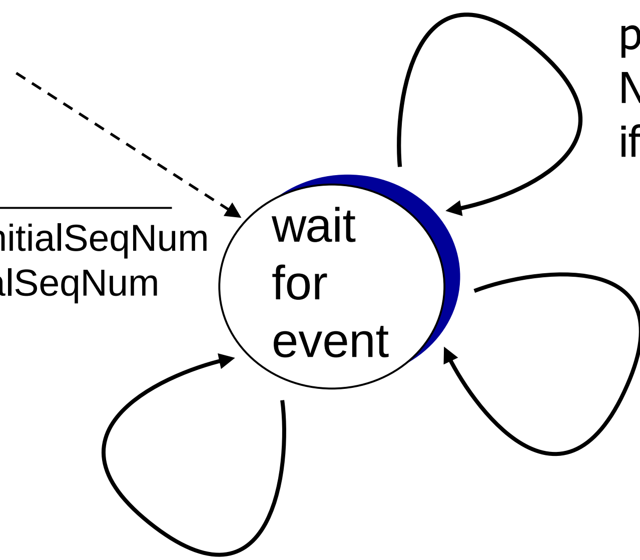
# TCP sender (simplified)

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

$\Lambda$

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



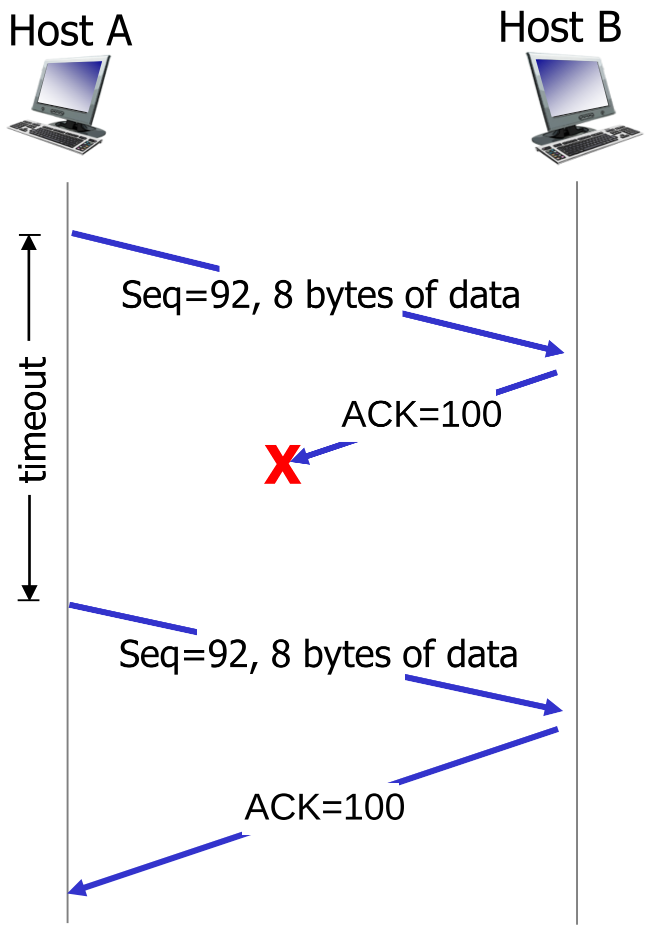

timeout

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

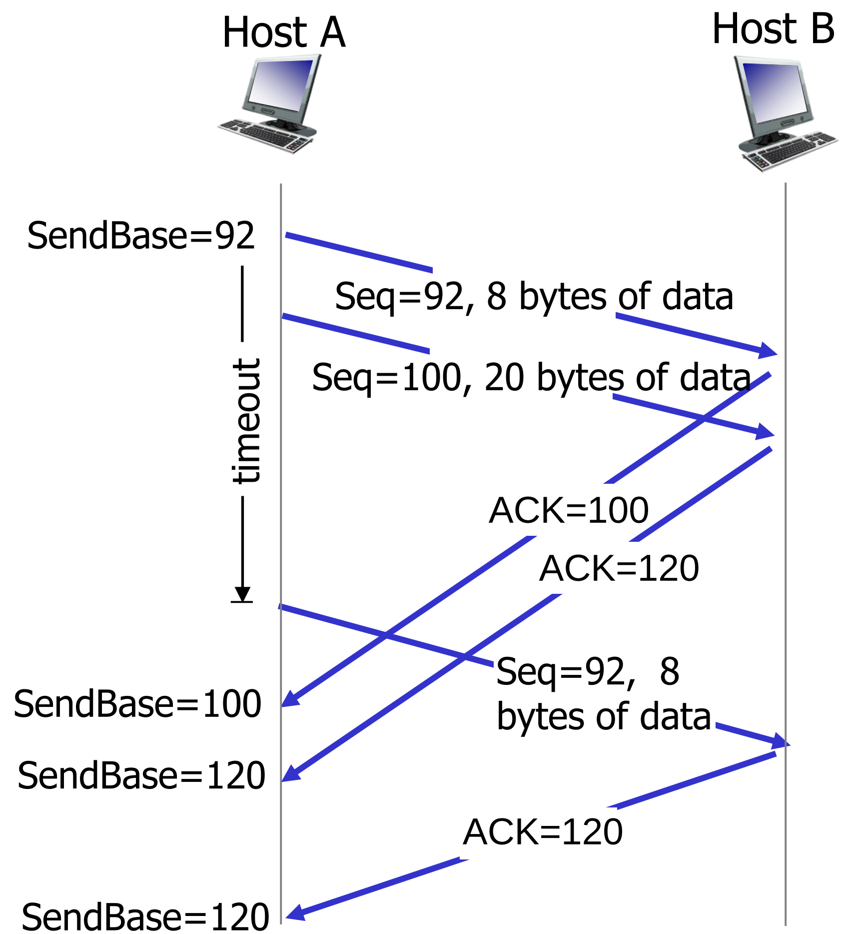
ACK received, with ACK field value y

```
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

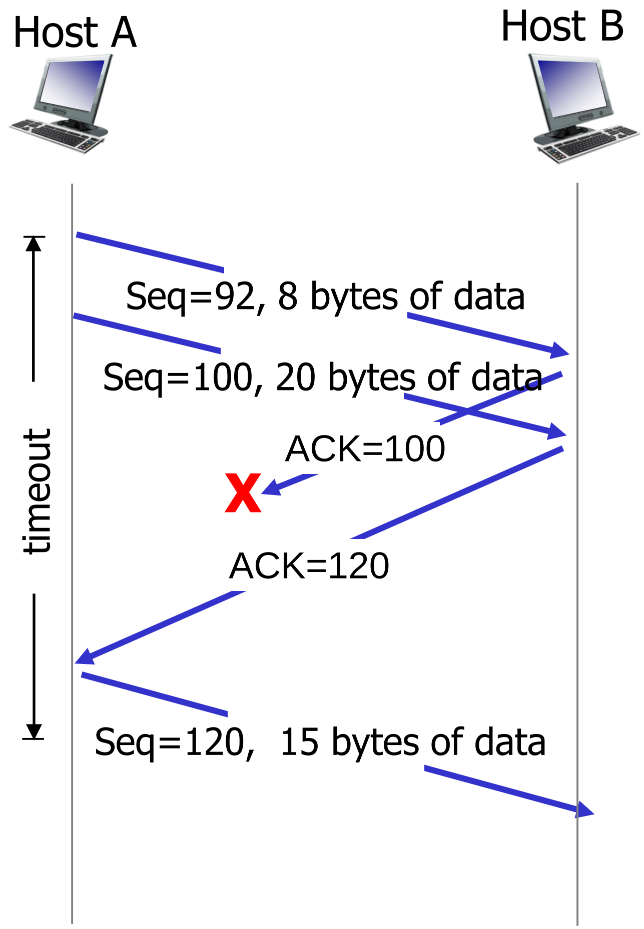



lost ACK scenario




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

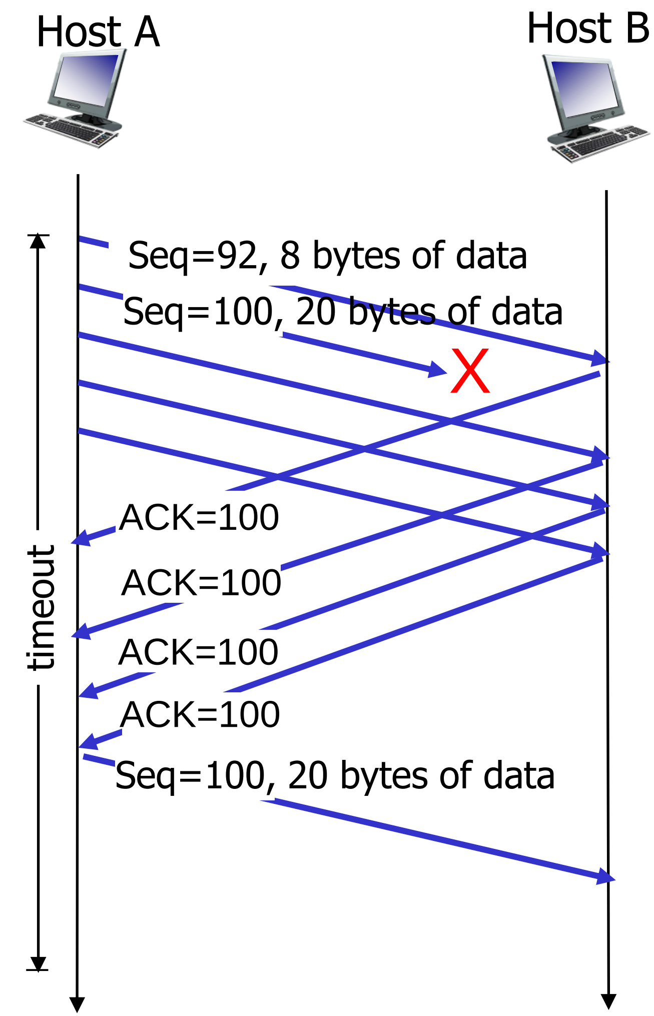
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

*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




fast retransmit after sender receipt of triple duplicate ACK

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

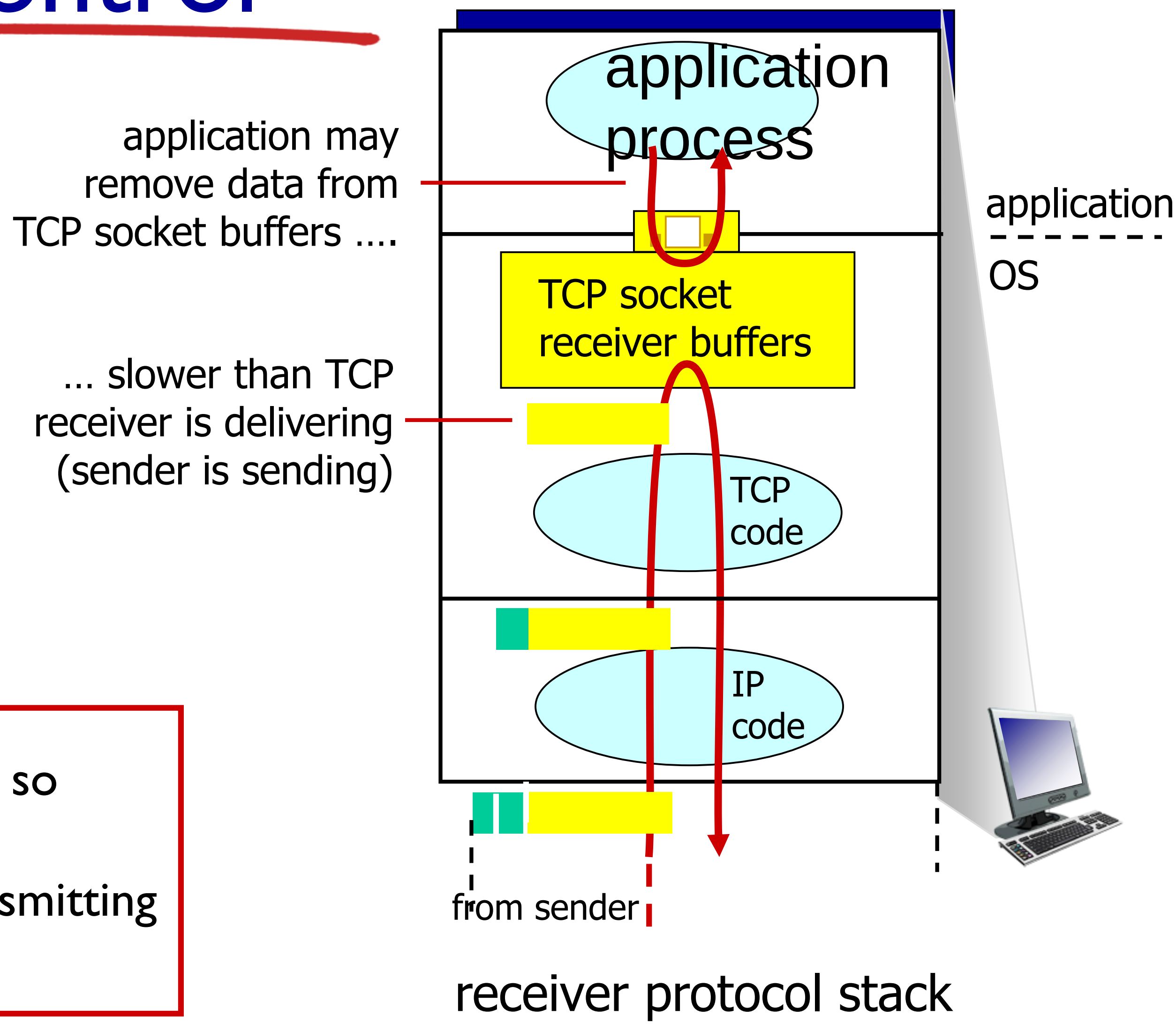
3.4 principles of reliable data transfer

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

# TCP flow control

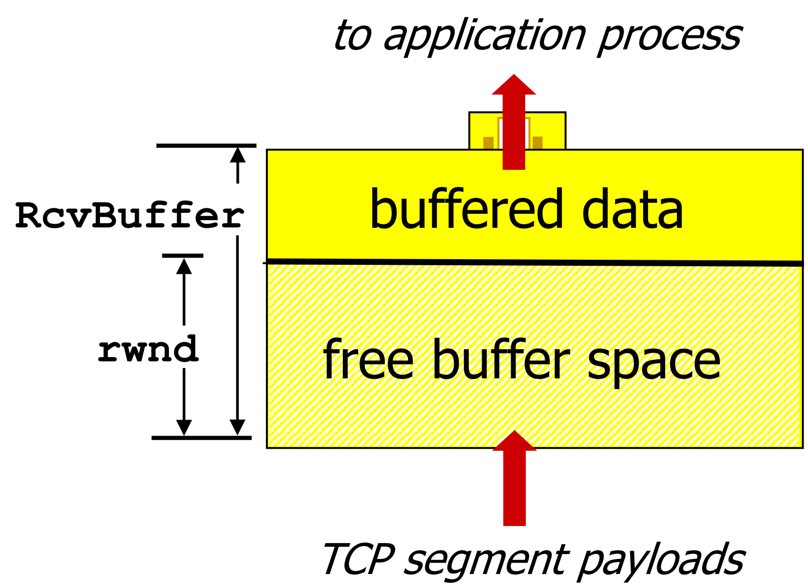



***flow control***

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*

# 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

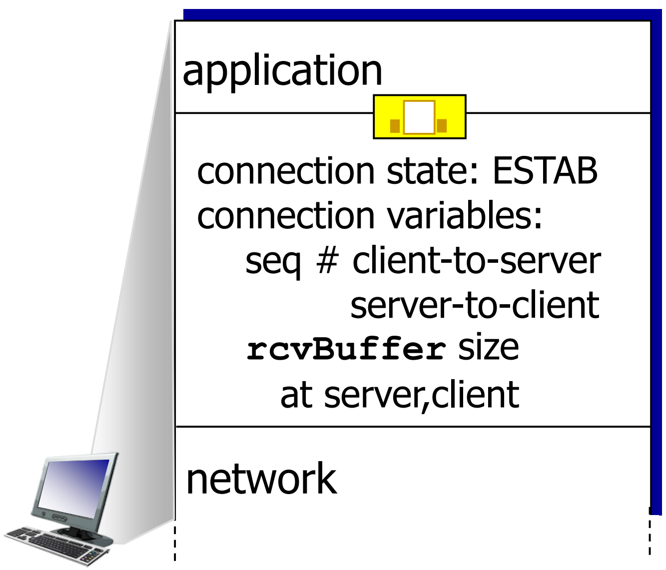
3.6 principles of congestion control

3.7 TCP congestion control

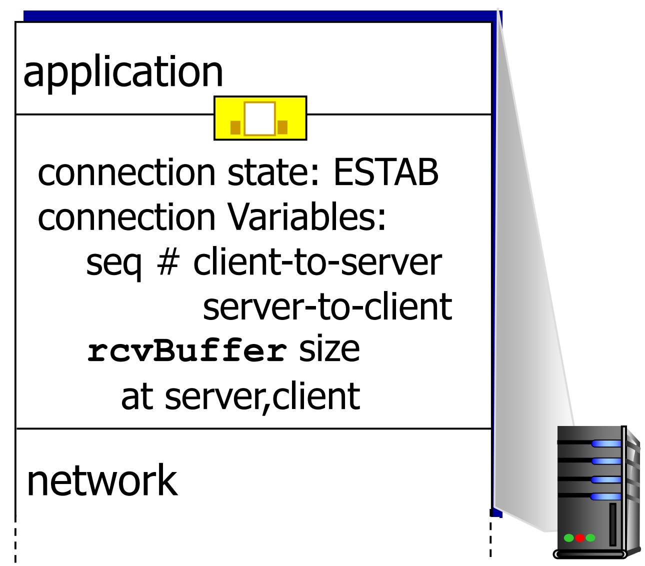
# Connection Management

before exchanging data, sender/receiver "handshake":

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



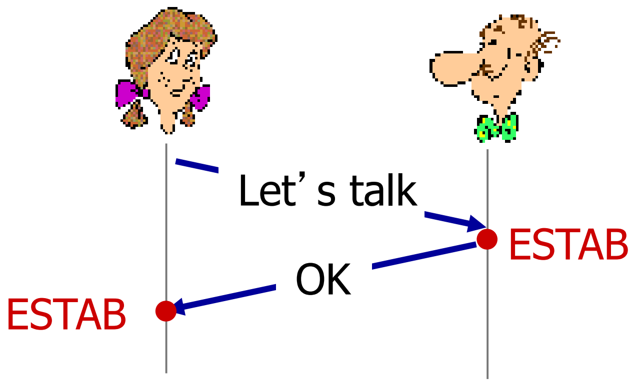

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

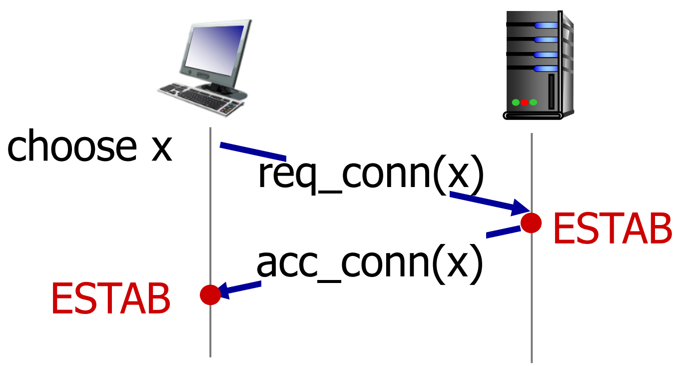



```
Socket connectionSocket =
   welcomeSocket.accept();
```

# Agreeing to establish a connection

2-way handshake:



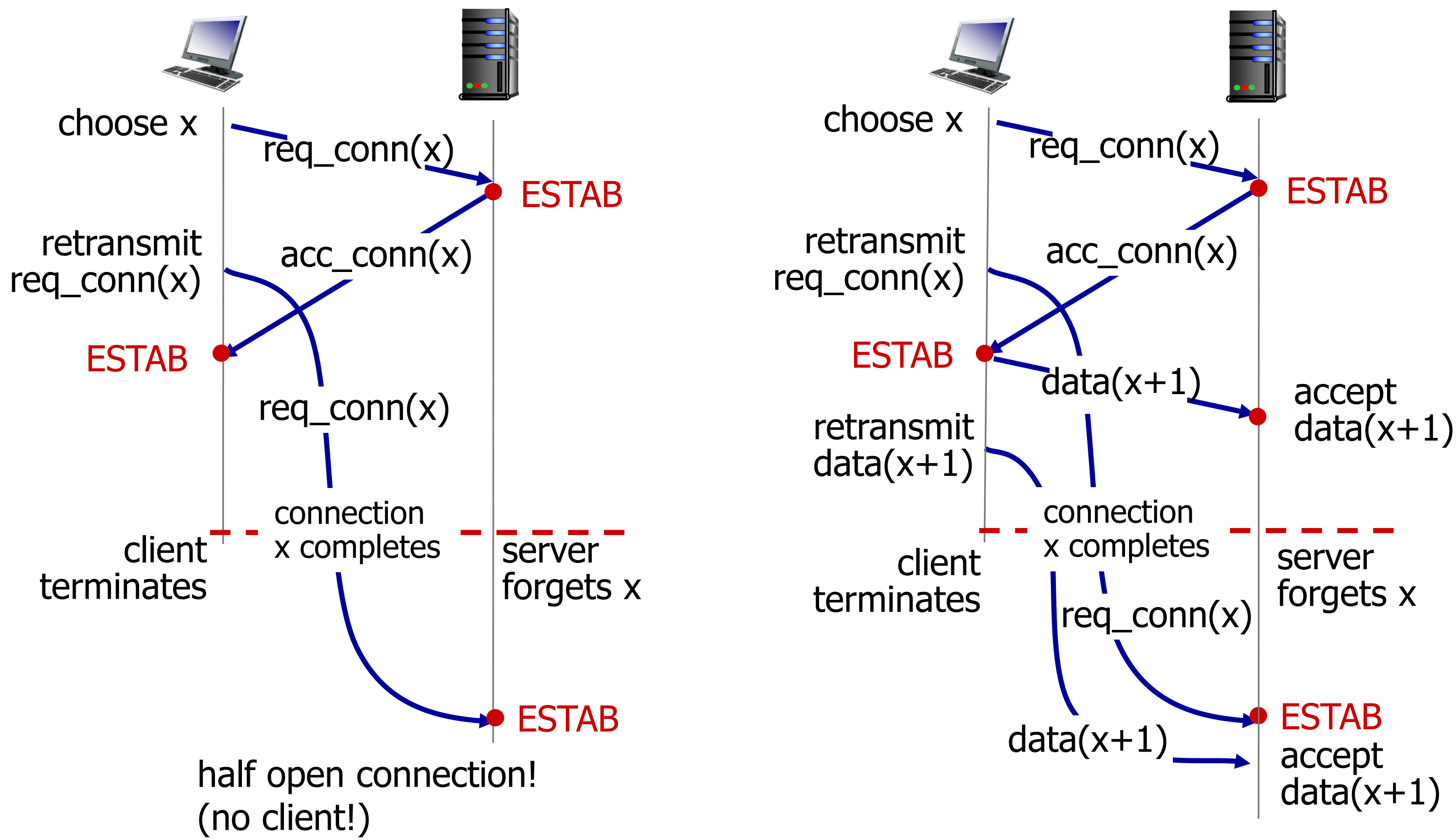




*Q:* will 2-way handshake always work in network?

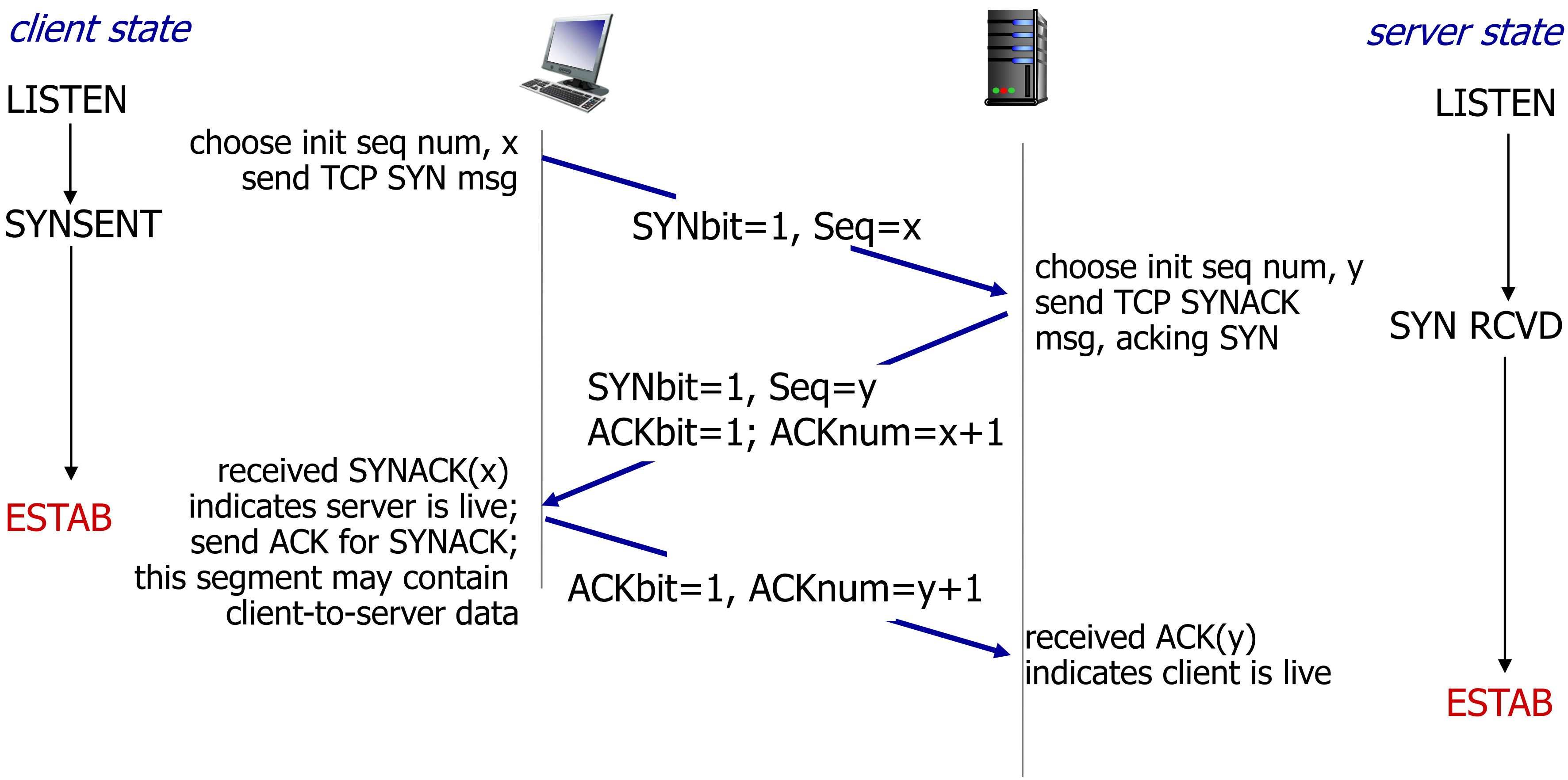
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

# Agreeing to establish a connection

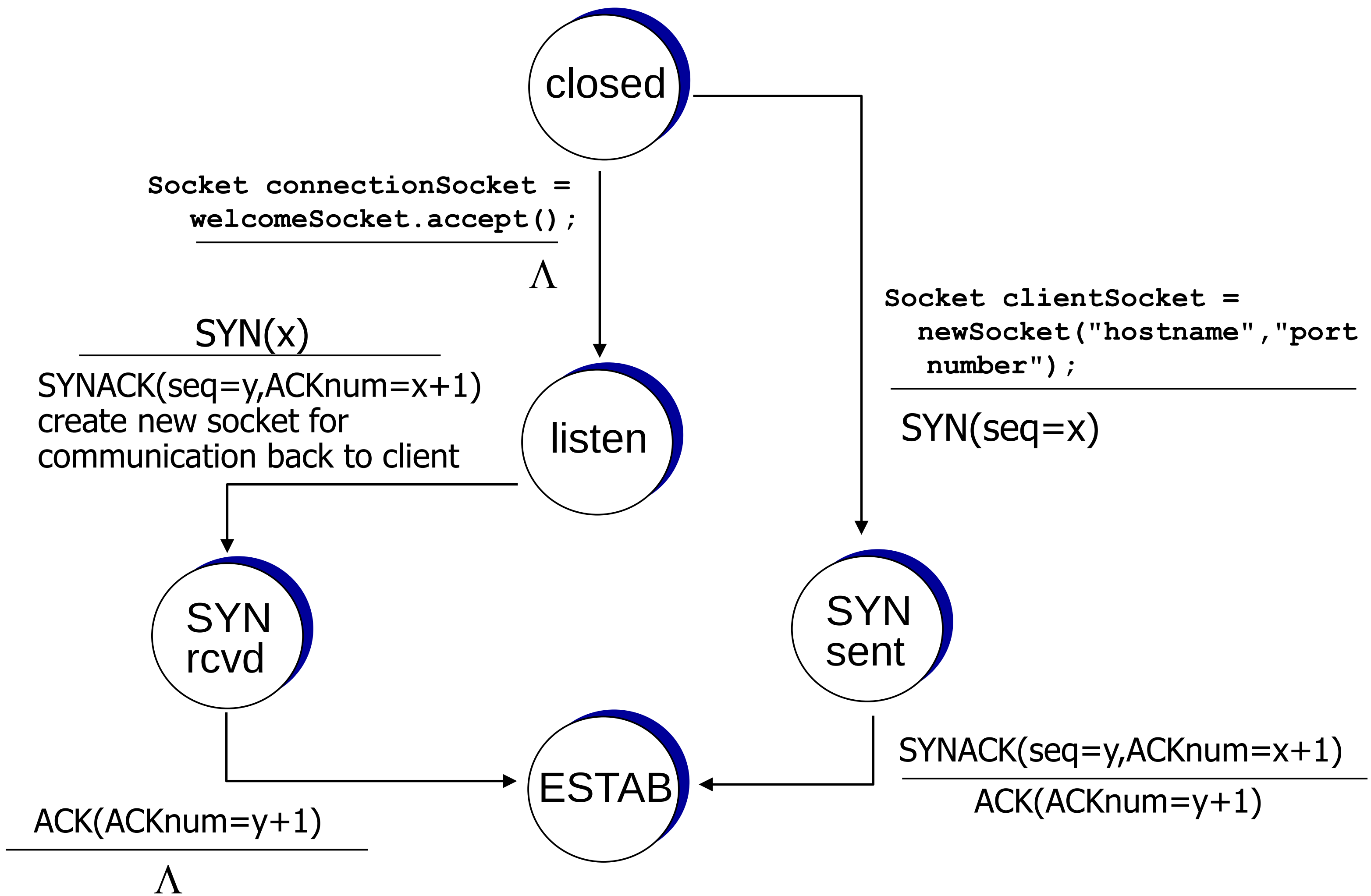
2-way handshake failure scenarios:



# TCP 3-way handshake



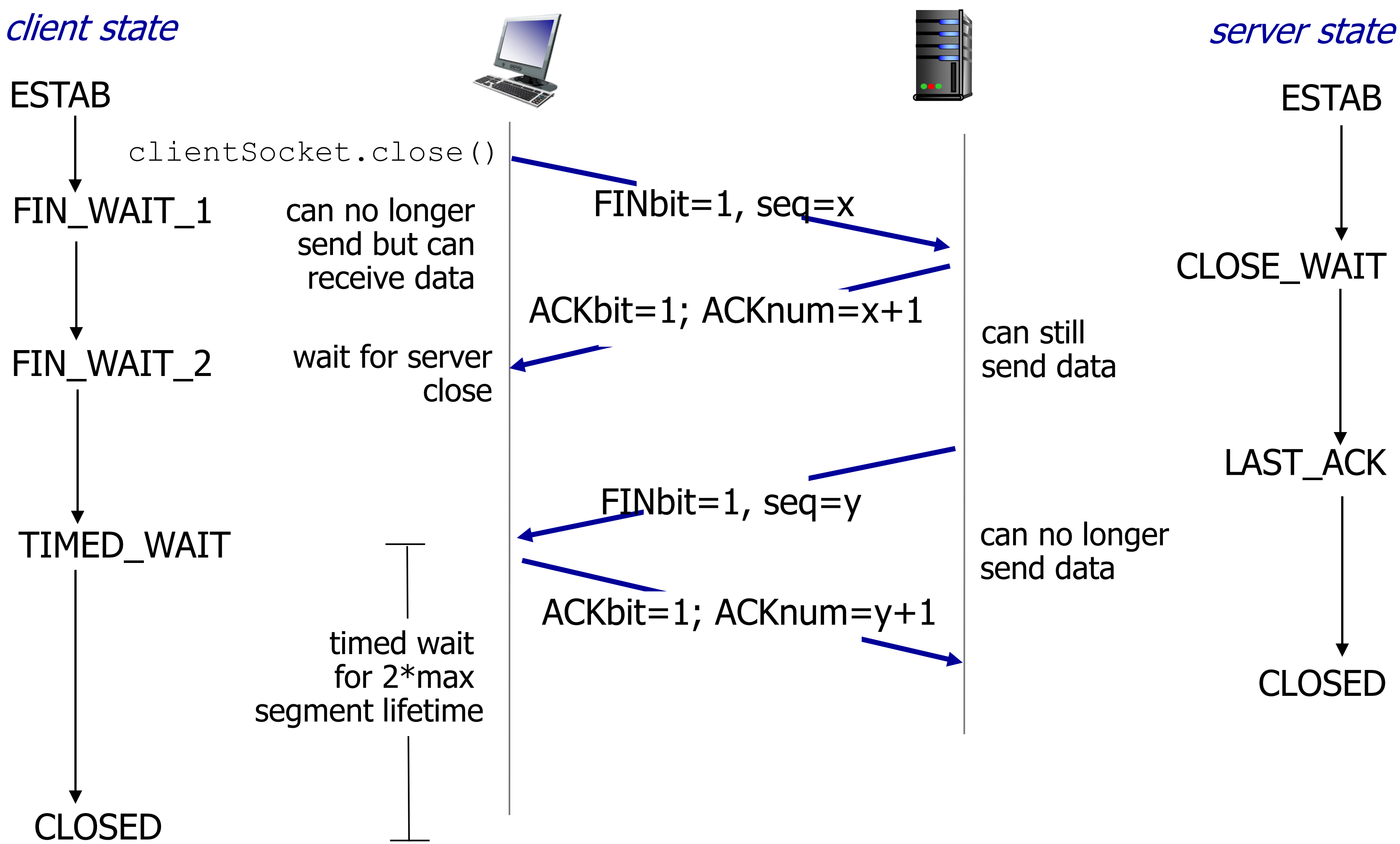
# TCP 3-way handshake: FSM



# 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



# Next class

- Please read Chapter 3.7-3.8 of your textbook BEFORE Class