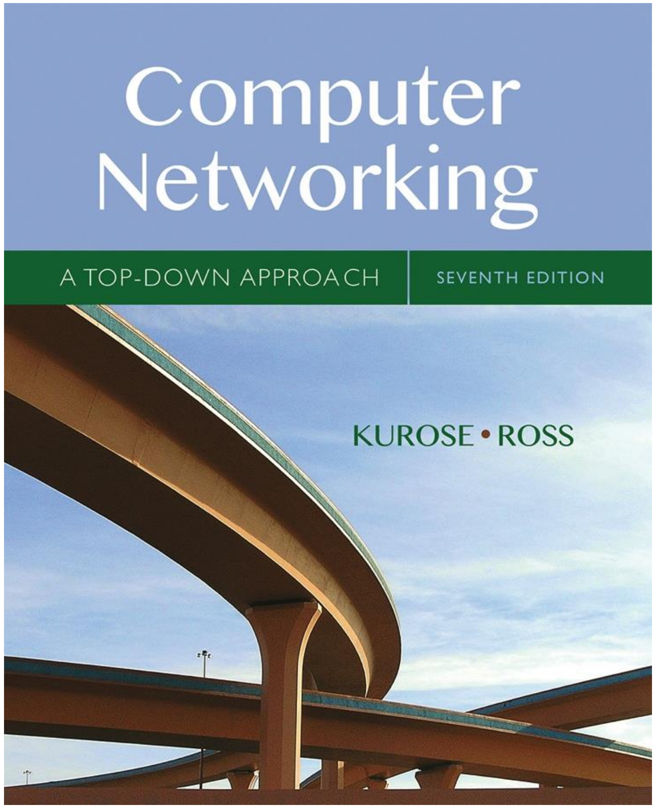
Chapter 3
Transport Layer
Part 4/5

A note on the use of these Powerpoint slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top Down Approach

7th edition

Jim Kurose, Keith Ross

Pearson/Addison Wesley
April 2016

Chapter 3 outline

3.1 transport-layer services

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

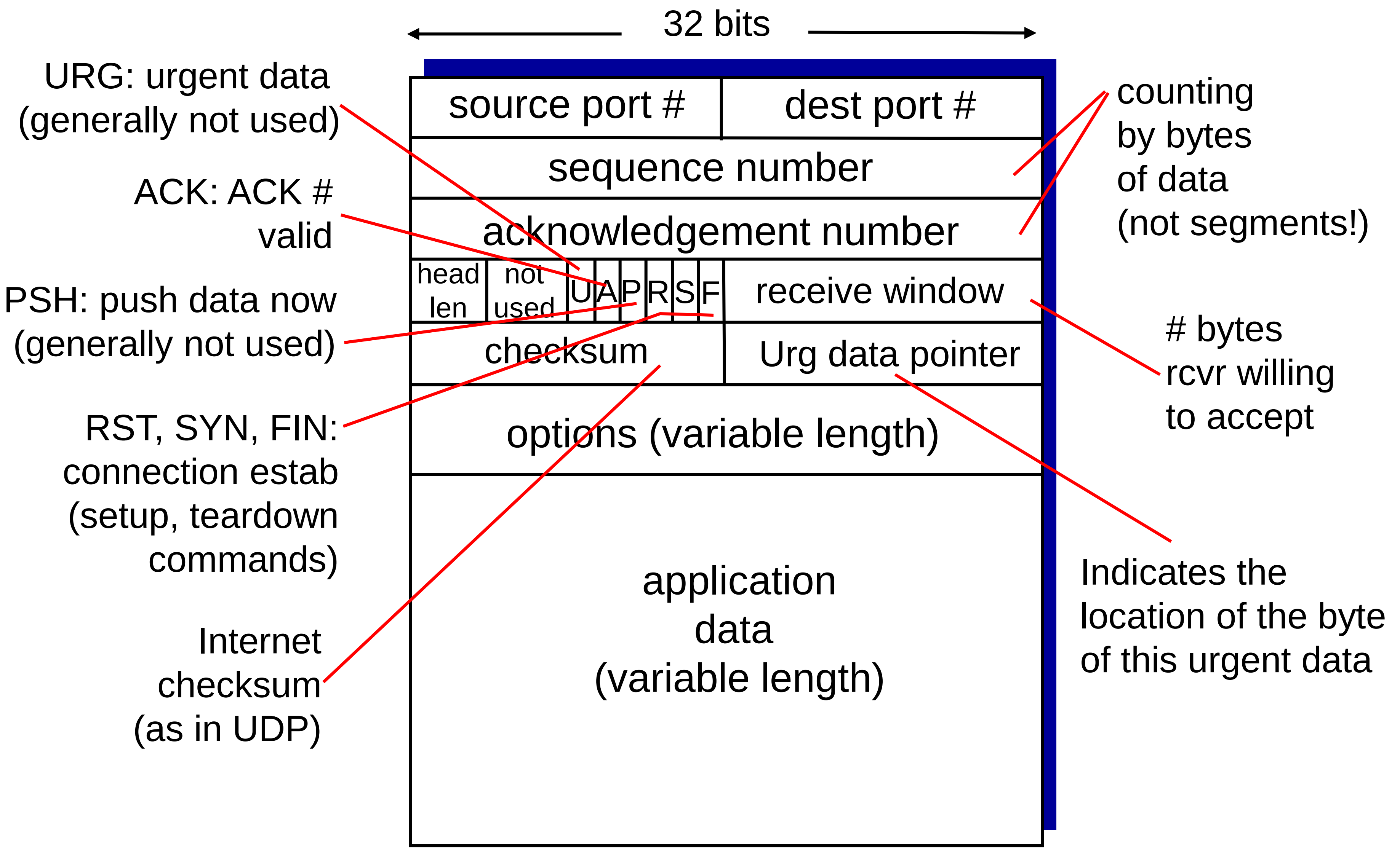
3.6 principles of congestion control

3.7 TCP congestion control

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - Segments delivered to app in order
- 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) initiates sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. numbers, ACKs

sequence numbers:

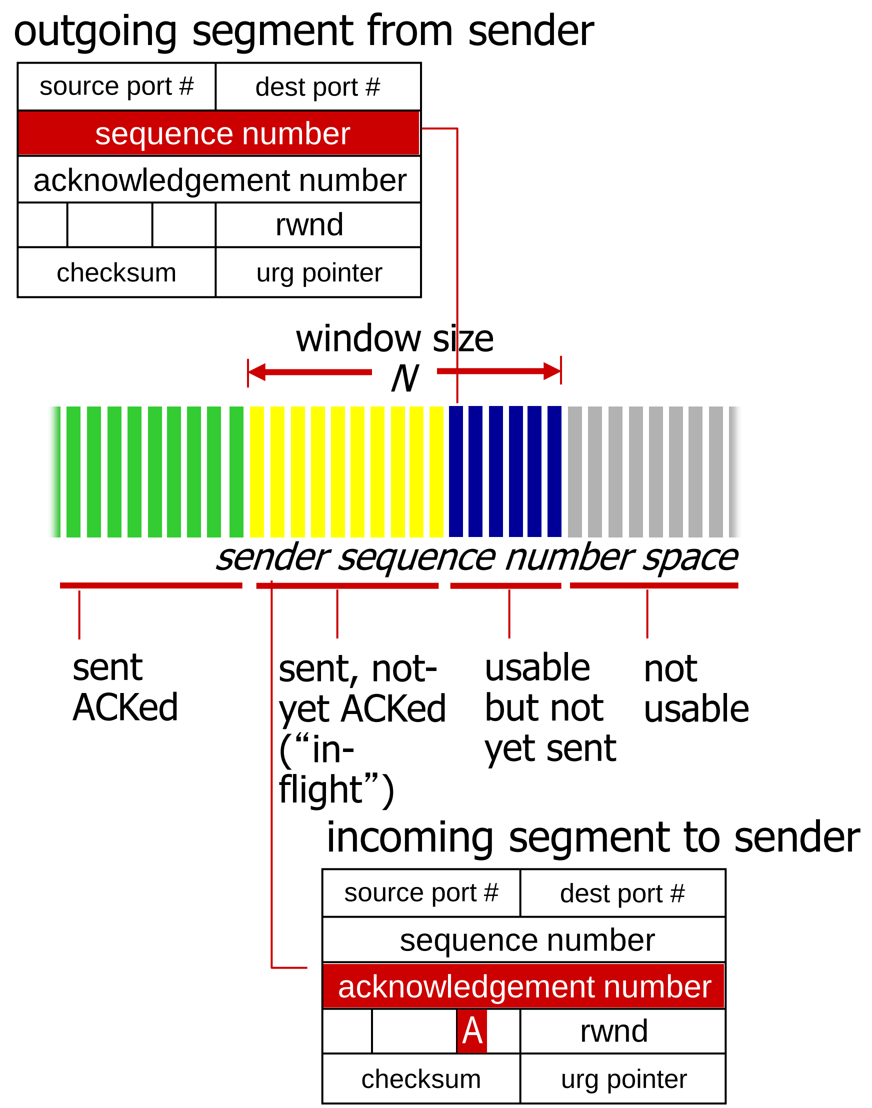
- byte stream "number" of first byte in segment's data

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, - up to implementor



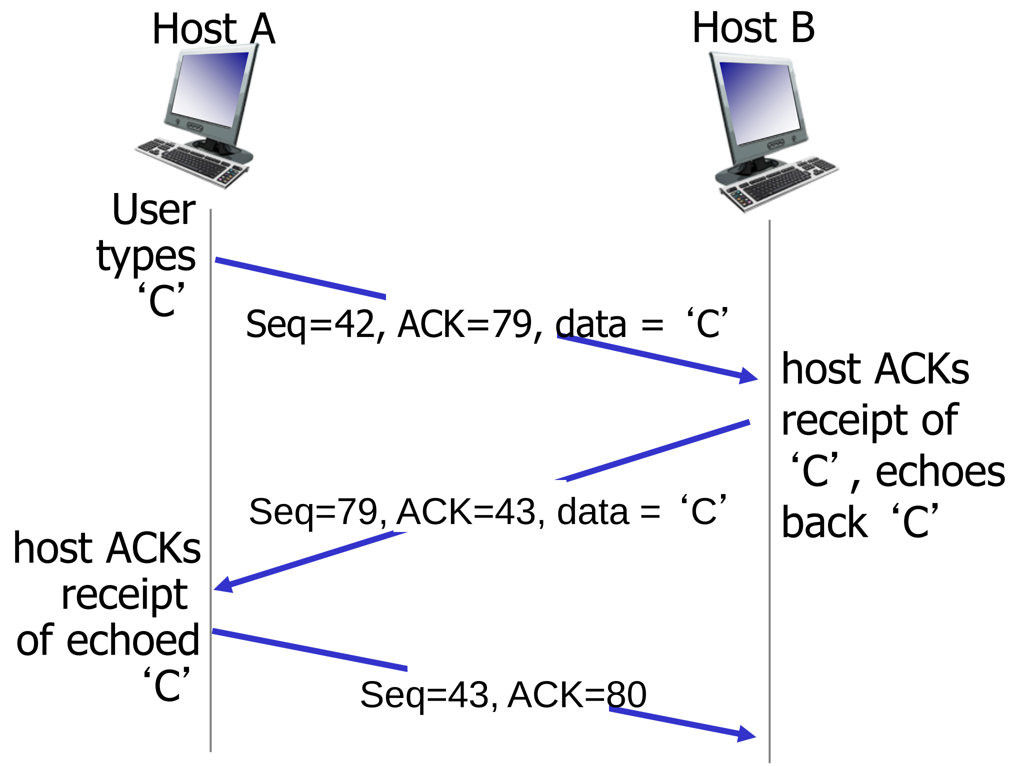
TCP seq. numbers, ACKs

Piggybacked ACK:

- ACK for client-to-server data is carried in a segment carrying server-to-client data.

acknowledgements:

- ACK=43 in server-to-client segment shows server has received everything upto byte 42
- Server waiting for byte 43



TCP round trip time and timeout

Why need to estimate RTT?

- "timeout" and "retransmit" needed to address packet loss
- need to know when to timeout and retransmit

Ideal world:

- exact RTT is needed

Real world:

- RTTs change over time:
  - packets may take different paths
  - network load changes over time
- RTTs can only be estimated

Some intuition

- What happens if too short?
  - Premature timeout
  - Unnecessary retransmissions
- What happens if too long?
  - Slow reaction to segment loss

TCP round trip time and timeout

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`**

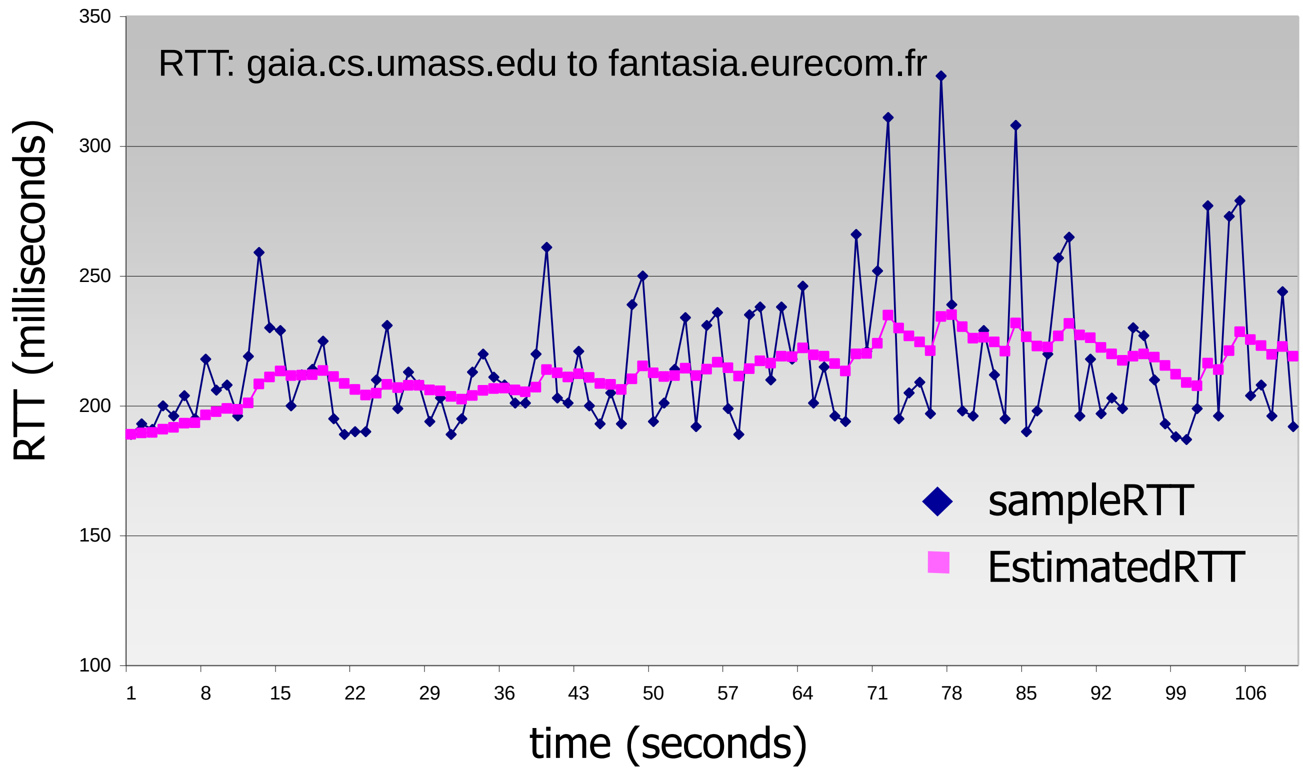
TCP round trip time and timeout

- We also want to give more recent measurements ***higher weight*** in case things do change

$$\texttt{EstimatedRTT}^{(\texttt{current})} = (1-\alpha) * \texttt{EstimatedRTT}^{(\texttt{Previous})} + \alpha * \texttt{SampleRTT}^{(\texttt{recent})}$$

- New EstimatedRTT is weighted combination of previous EstimatedRTT and new SampleRTT
- All the samples are averaged using ***exponential weighted moving average*** (EWMA)
- influence of past sample decreases exponentially fast
- Coefficient α is the ***degree of weighting decrease***
- $0<\alpha<1$, but typical value: $\alpha = 0.125$

TCP round trip time and timeout



TCP round trip time and timeout

Setting the timeout

- `timeout = EstimtedRTT`, **any problem with this???**
- add a "safety margin" to `EstimtedRTT`
  - large variation in `EstimatedRTT` -> larger safety margin
- see how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
               (typically, β = 0.25)
```

Then set timeout interval:

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

TCP round trip time and timeout

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

```
SampleDevRTT(recent) = |SampleRTT-EstimatedRTT|

DevRTT(current) = (1-β)*DevRTT(Previous) + β* SampleDevRTT(recent)
```

(0<β<1; **`typically β = 0.25)`**

```
TimeoutInterval = EstimatedRTT(current) + 4*DevRTT(Current)
```

estimated RTT

"safety margin"

TCP round trip time and timeout (class exercises)

- Suppose that TCP's current estimatedRTT and DevRTT are 340 msec and 18 msec, respectively. Suppose that the next three SampleRTTs are 400, 270, and 390 respectively. Compute TCP's new value of estimatedRTT, DevRTT, and the TCP timeout value after each of these three measured RTT values is obtained.Use the values of α = 0.125 and β = 0.25.

$$\text{EstimatedRTT}^{(\text{current})} = (1-\alpha)*\text{EstimatedRTT}^{(\text{Previous})} + \alpha*\text{SampleRTT}^{(\text{recent})}$$

$$\text{DevRTT}^{(\text{current})} = (1-\beta)*\text{DevRTT}^{(\text{Previous})} + \beta*(|\text{SampleRTT}-\text{EstimatedRTT}|)^{(\text{recent})}$$

- Solution:

1)SampleRTT$_{1(\text{recent})}$ = 400ms, EstimatedRTT$_{(\text{previous})}$=340ms, DevRTT$_{(\text{previous})}$=18ms

estimatedRTT$_{(\text{current})}$ = ?

DevRTT $_{(\text{current})}$ = ?

TimeoutInterval = ?

TCP round trip time and timeout (class exercises)

- Suppose that TCP's current estimatedRTT and DevRTT are 340 msec and 18 msec, respectively. Suppose that the next three SampleRTTs are 400, 270, and 390 respectively. Compute TCP's new value of estimatedRTT, DevRTT, and the TCP timeout value after each of these three measured RTT values is obtained.Use the values of α = 0.125 and β = 0.25.

$$\text{EstimatedRTT}^{(\text{current})} = (1-\alpha) * \text{EstimatedRTT}^{(\text{Previous})} + \alpha * \text{SampleRTT}^{(\text{recent})}$$

$$\text{DevRTT}^{(\text{current})} = (1-\beta) * \text{DevRTT}^{(\text{Previous})} + \beta * (|\text{SampleRTT}-\text{EstimatedRTT}|)^{(\text{recent})}$$

- Solution:

1) **SampleRTT_1 = 400ms, EstimatedRTT=340ms, DevRTT=18ms**

estimatedRTT = 0.875*340 + 0.125*400 = 347.5 msecs

DevRTT = 0.75*18 + 0.25*(abs(400 - 340)) = 28.5 msecs

TimeoutInterval = 347.5 + 4*28.5 = 461.5 msecs

TCP round trip time and timeout (class exercises)

2) SampleRTT_2 = 270ms , EstimatedRTT=347.5ms, DevRTT=28.5ms

estimatedRTT = 0.875*347.5 + 0.125*270 = 337.8125 msecs

DevRTT = 0.75*28.5 + 0.25*(abs(270 – 347.5)) = 40.75 msecs

TimeoutInterval = 337.8125 + 4*40.625= ? msecs

3) SampleRTT_3 = 390ms , EstimatedRTT=337.8125ms, DevRTT=40.625ms

estimatedRTT = 0.875*337.8125 + 0.125*390 = 344.335 msecs

DevRTT = ??

TimeoutInterval = ??

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
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control
- ignore 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

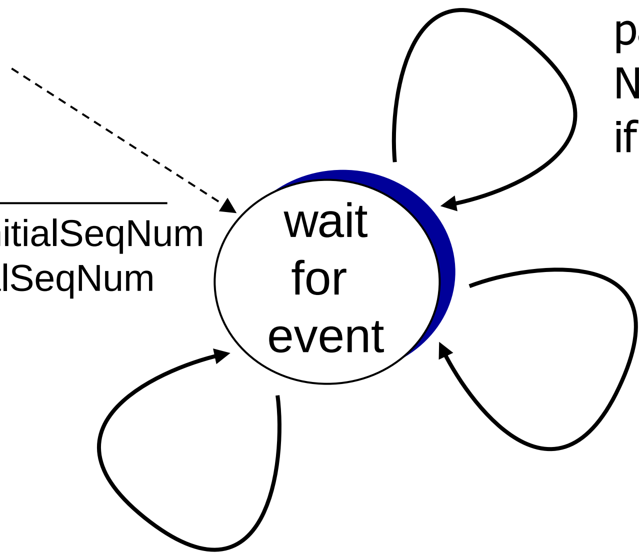
TCP sender (simplified)

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



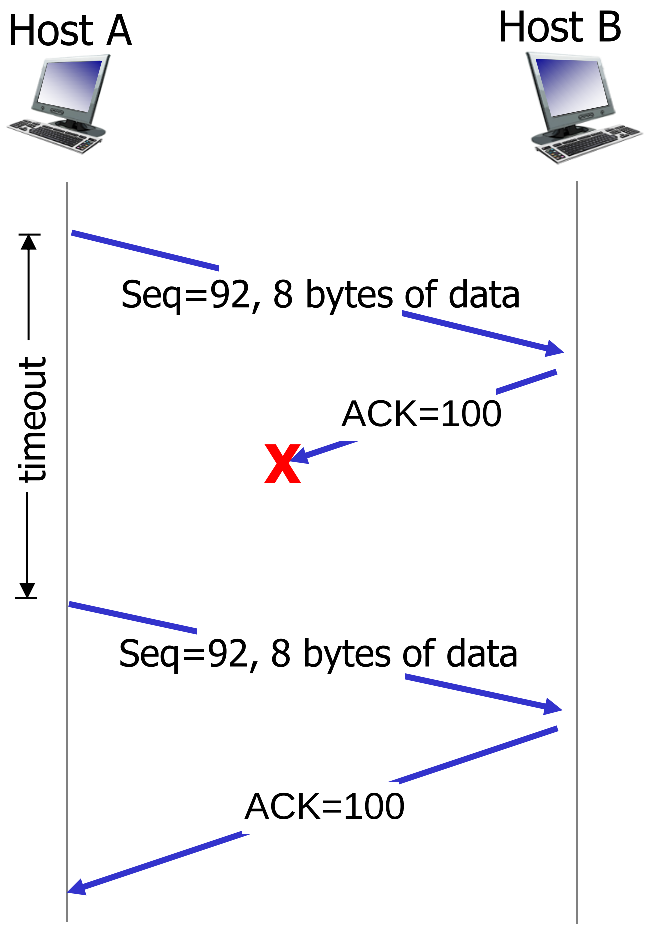
timeout

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

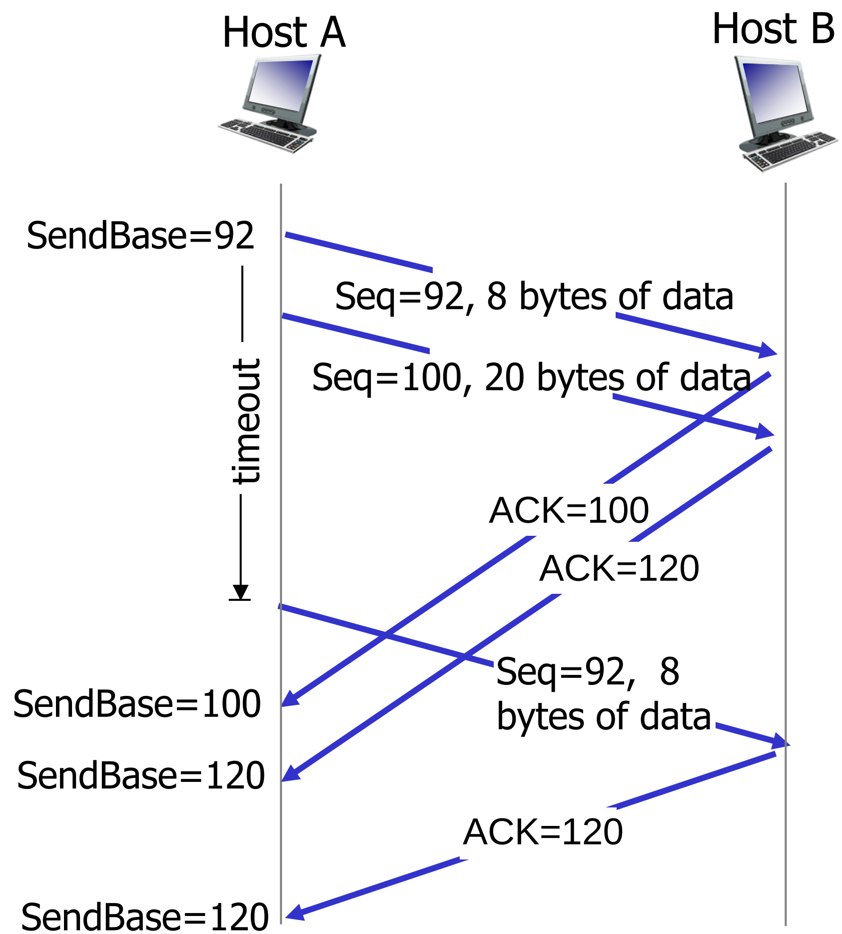
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y    /* update SendBase */
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
        start timer
     else stop timer //no in-flight packet, so no need to start timer
 }
```

TCP: retransmission scenarios

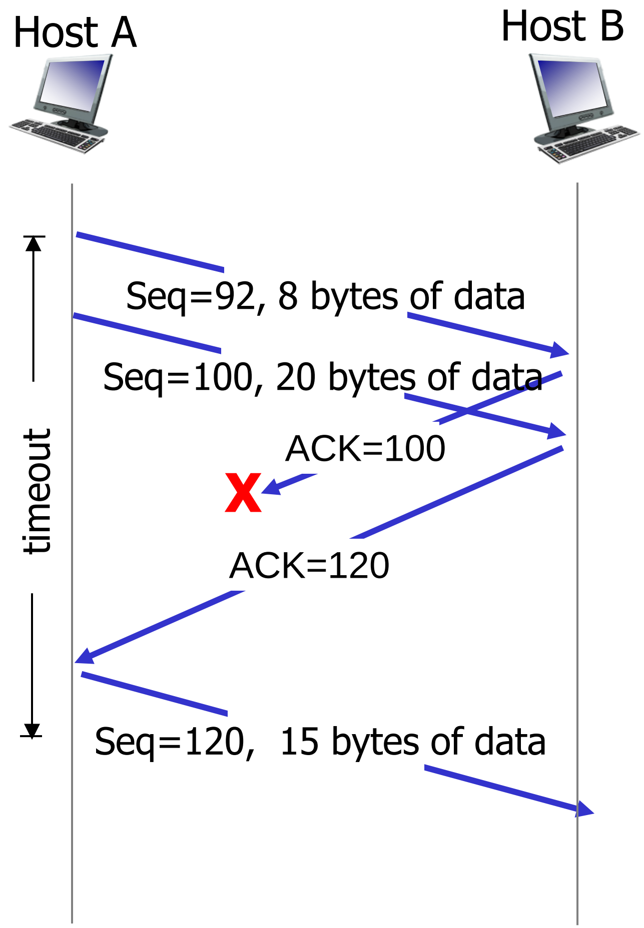


lost ACK scenario



premature timeout
(In the last, receiver sends Cumulative ACK)

TCP: retransmission scenarios



cumulative ACK

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.

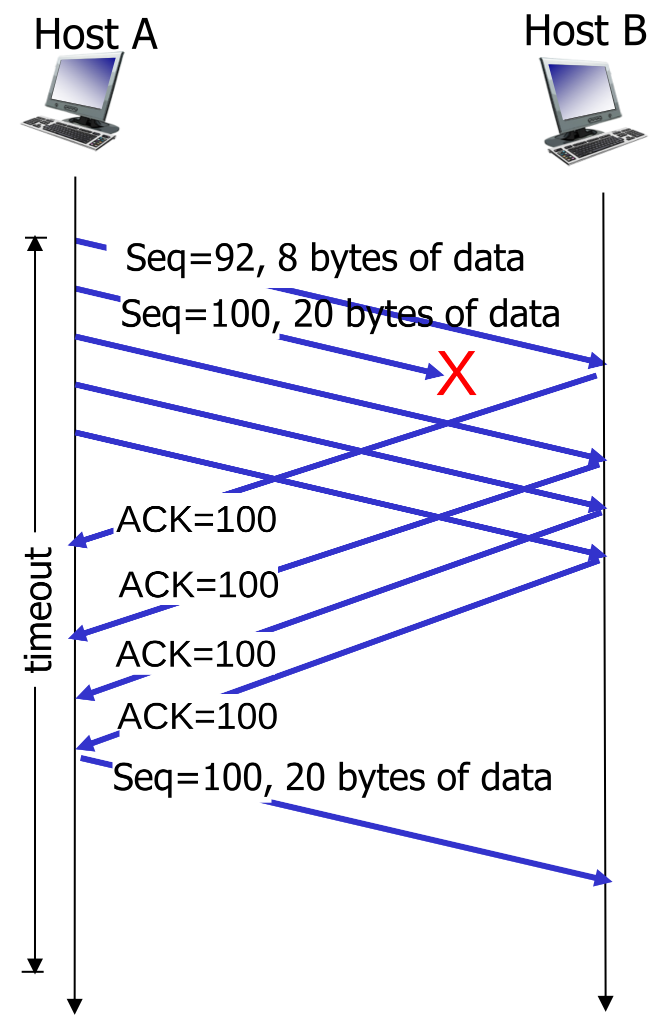
TCP fast retransmit

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

- Normally TCP sender will wait for the time-out before retransmission,
  - But in TCP fast retransmit, sender will retransmit before the timeout happens if it receives 3 dup ack.



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

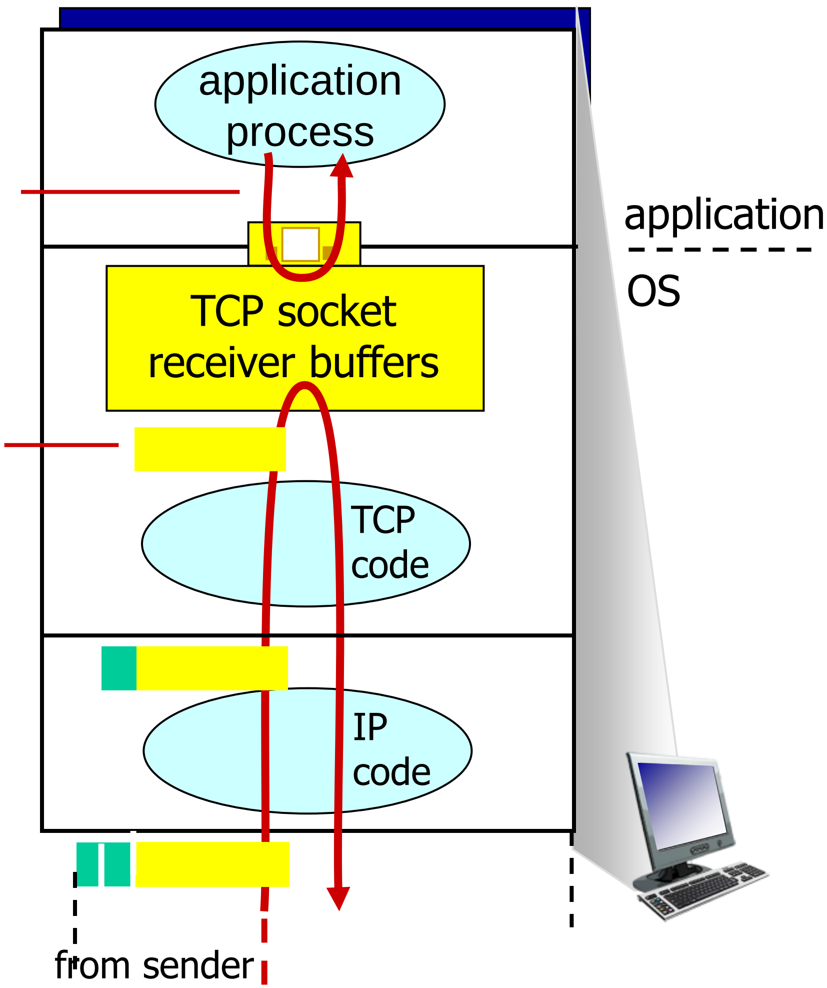
3.6 principles of congestion control

3.7 TCP congestion control

TCP flow control

application may remove data from TCP socket buffers ….

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



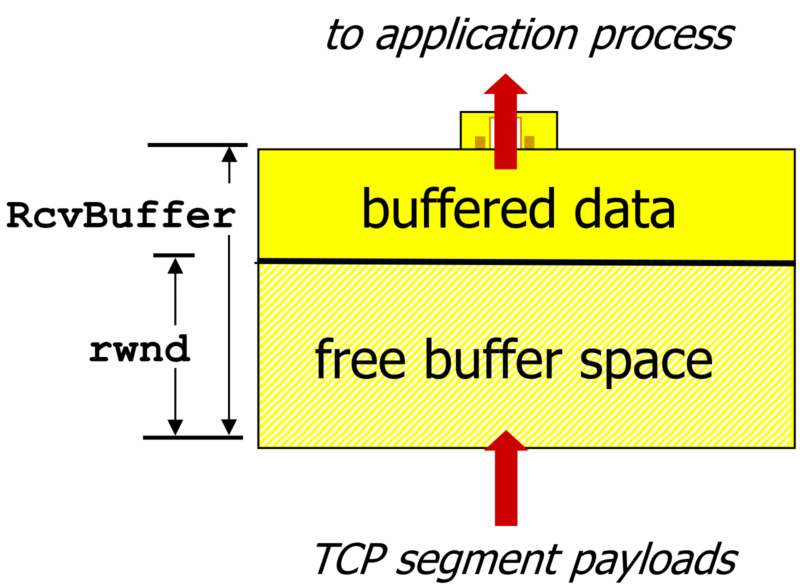
receiver protocol stack

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



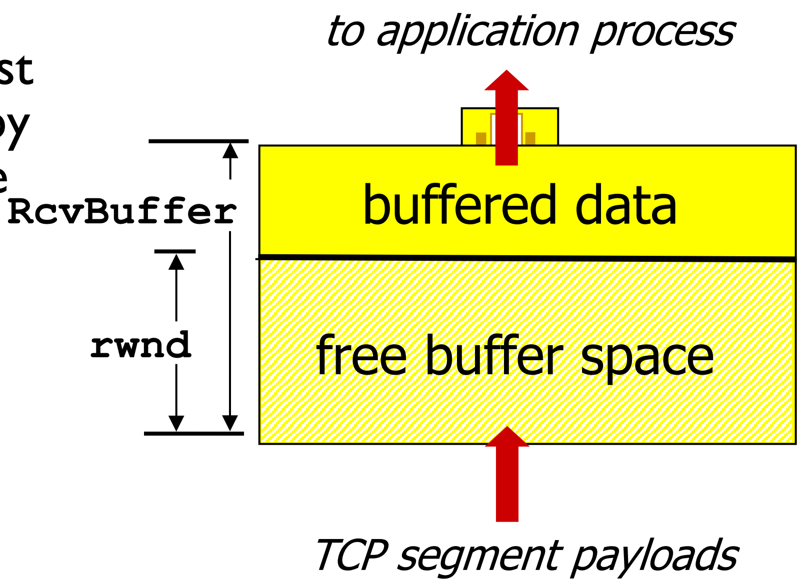
receiver-side buffering

TCP flow control: how to calculate rwnd?

- Calculating rwnd
  - **`rwnd = RcvBuffer-[LastByteRcvd-LastByteRead]`**

- **LastByteRead:** the sequence number of the last byte in the data stream read from the buffer by the application process of the receiving host
- **LastByteRcvd:** the sequence number of the last byte in the data stream that has been received by the receive buffer of the receiving host from the network
- Relationship between RcvBuffer, LastByteRcvd, LastByteRead

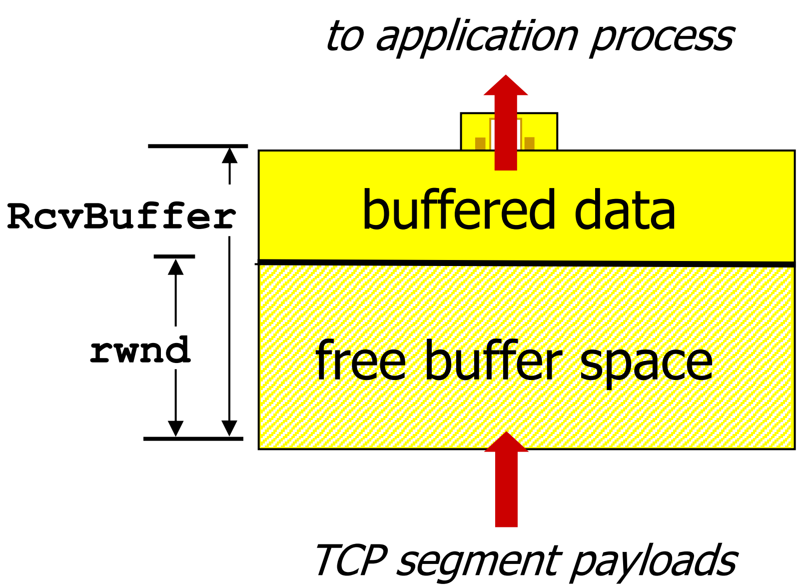
`LastByteRcvd-LastByteRead ≤ RcvBuffer`



receiver-side buffering

TCP flow control

- Deadlock Situation
  - **`Receiver consumed some data and free up some receive buffer space`**
  - **`Sender thinks rwnd=0 and doesn't send segment`**
  - Sender **has some data in the *send buffer***, it ***periodically* sends a *one byte* TCP segment** to the receiver to **trigger a response** from the **receiver**
  - **Ack** for the **byte size probe TCP segment** will contain the **non-zero value rwnd** that the **sender** uses for transmission



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

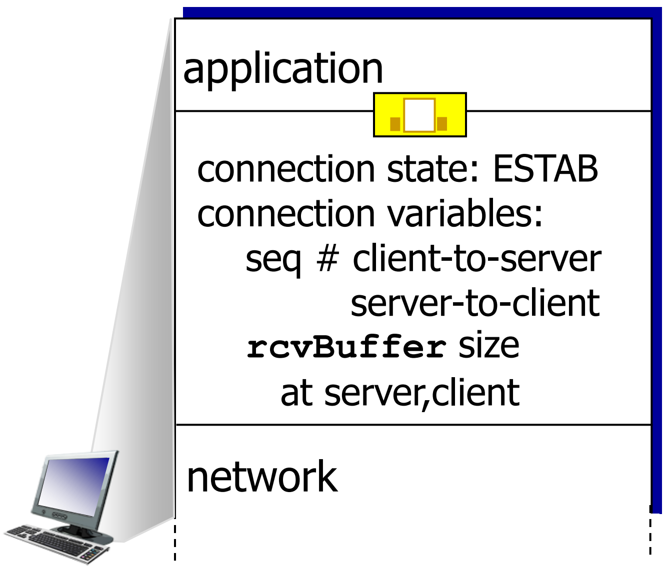
3.6 principles of congestion control

3.7 TCP congestion control

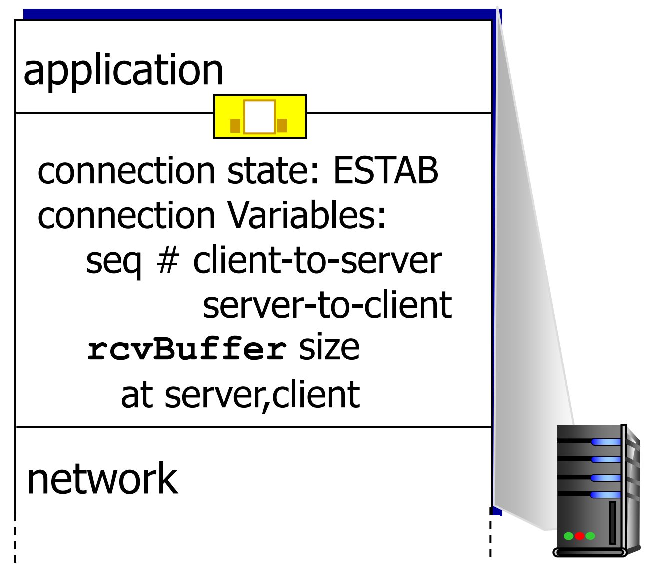
Connection Management

before exchanging data, sender/receiver "handshake":

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



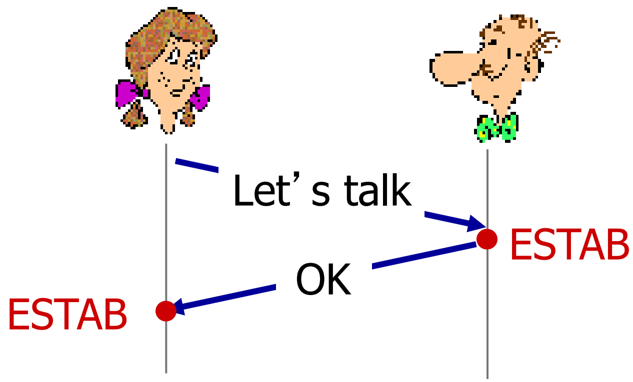

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

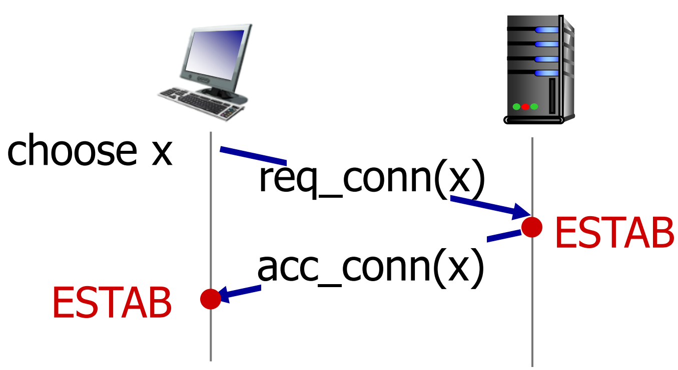



```
Socket connectionSocket =
   welcomeSocket.accept();
```

Connection Management: Agreeing to establish a connection

2-way handshake:



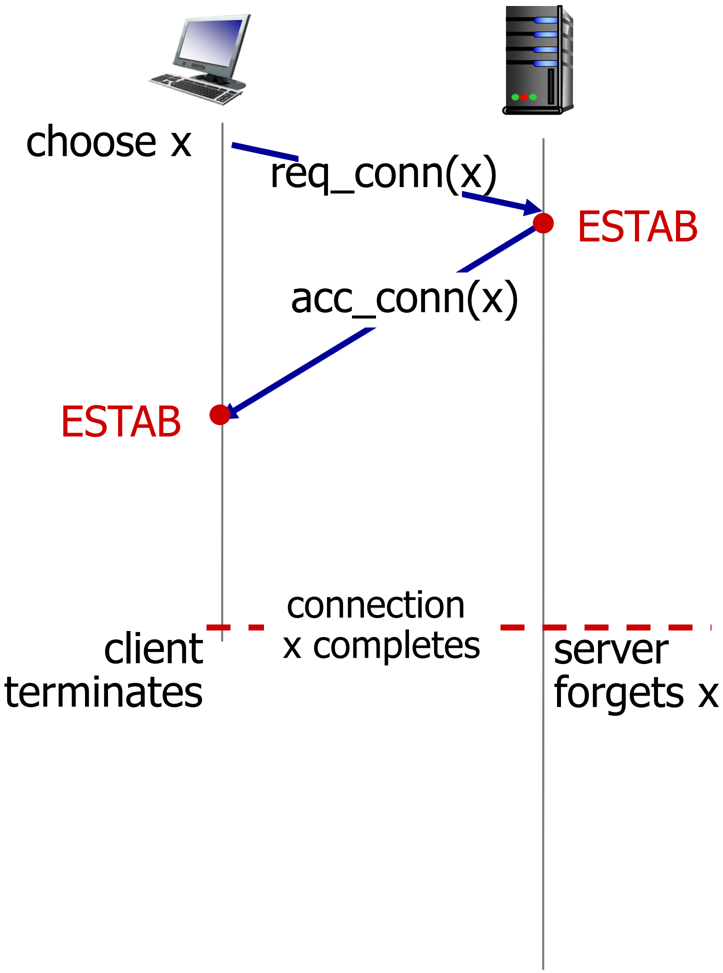


Q: will 2-way handshake always work in network?

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

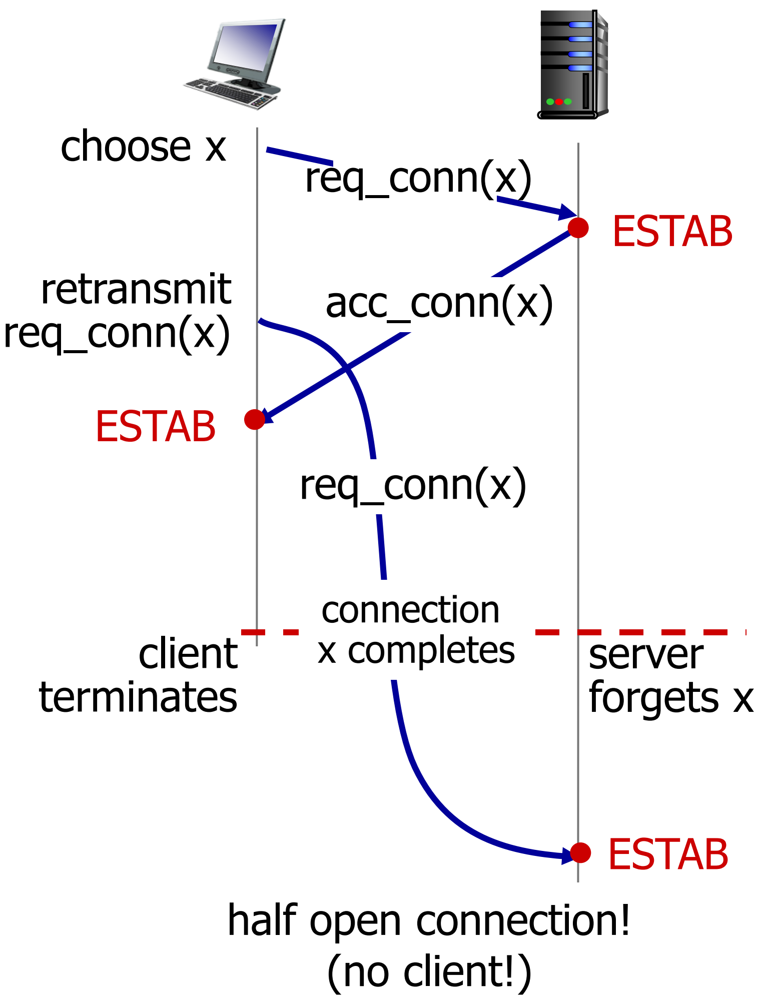
Agreeing to establish a connection: Ist scenario

2-way handshake failure scenarios:



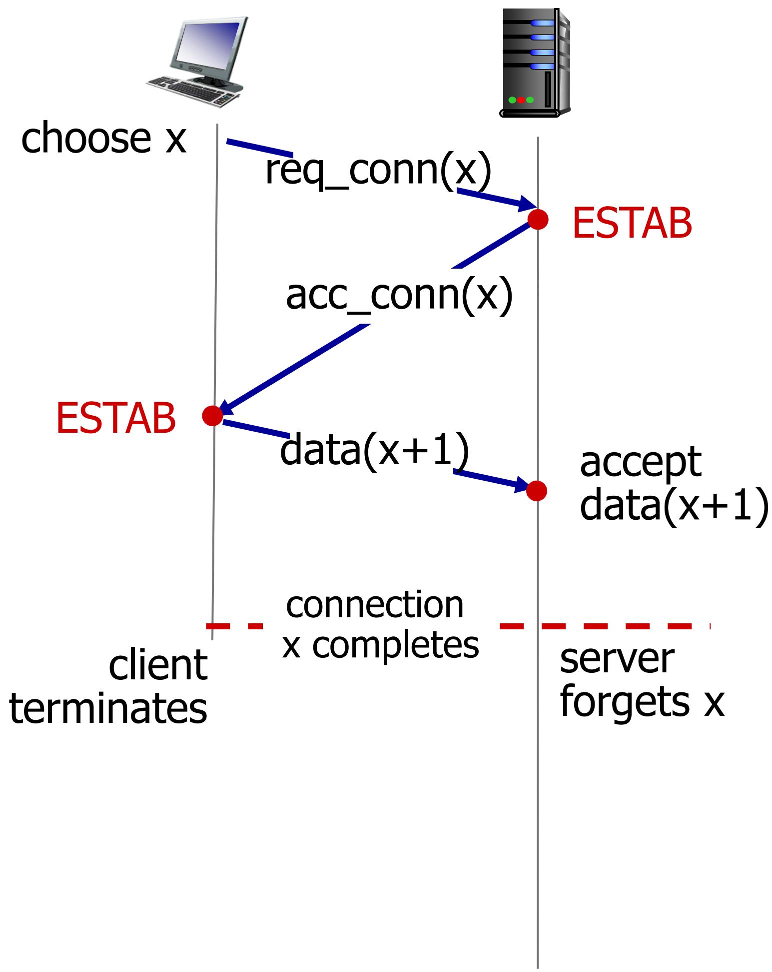
Agreeing to establish a connection: 1st Scenario

2-way handshake failure scenarios:



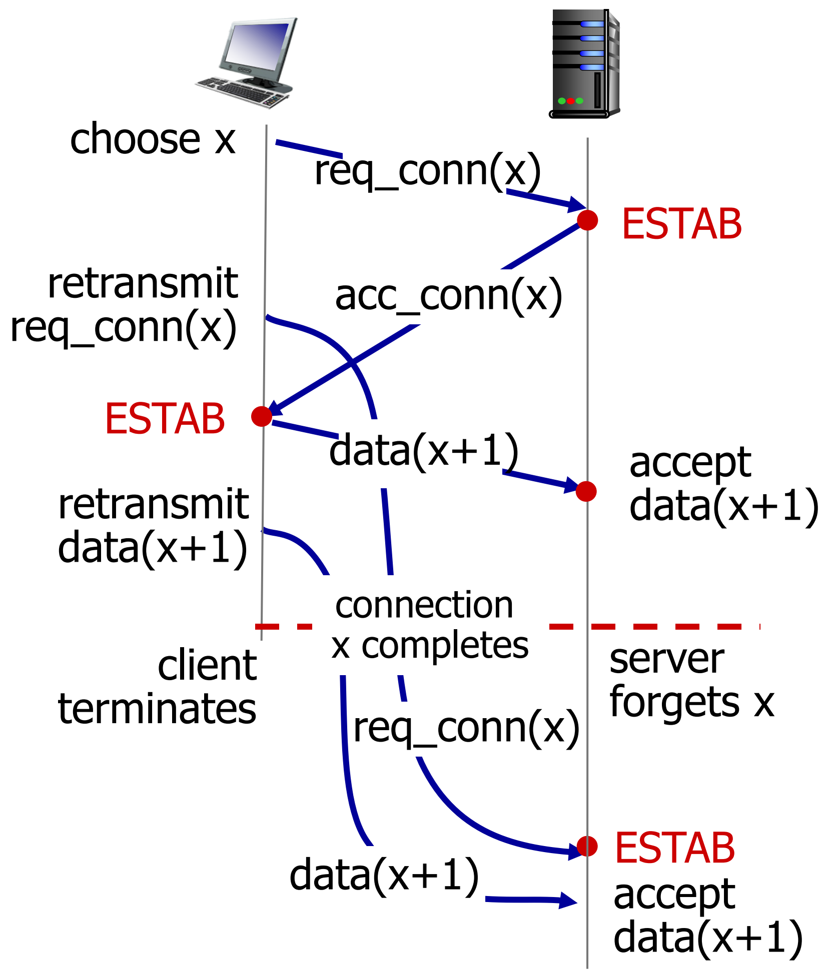
Agreeing to establish a connection: 2nd Scenario

2-way handshake failure scenarios:

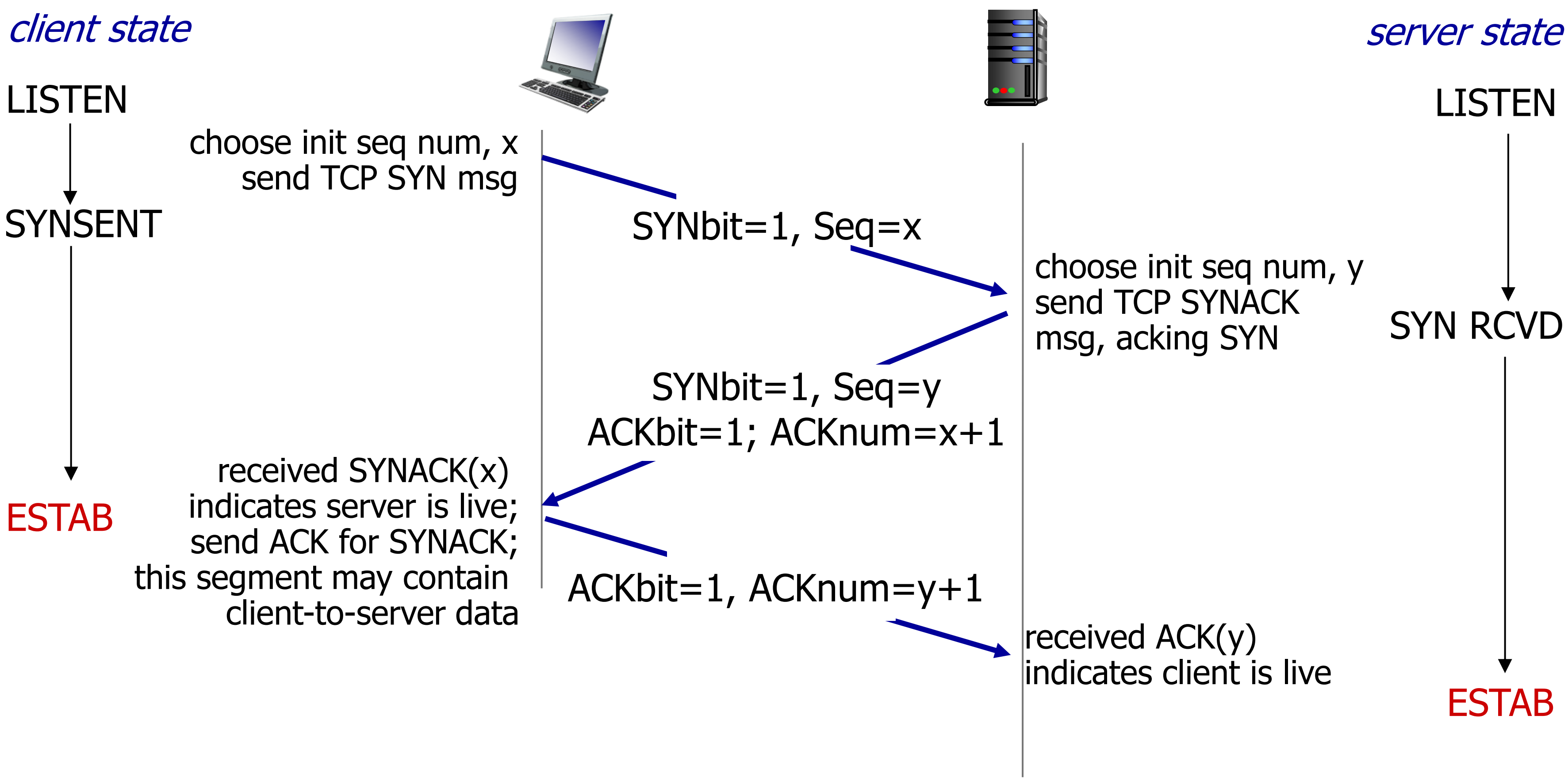


Agreeing to establish a connection: 2nd Scenario

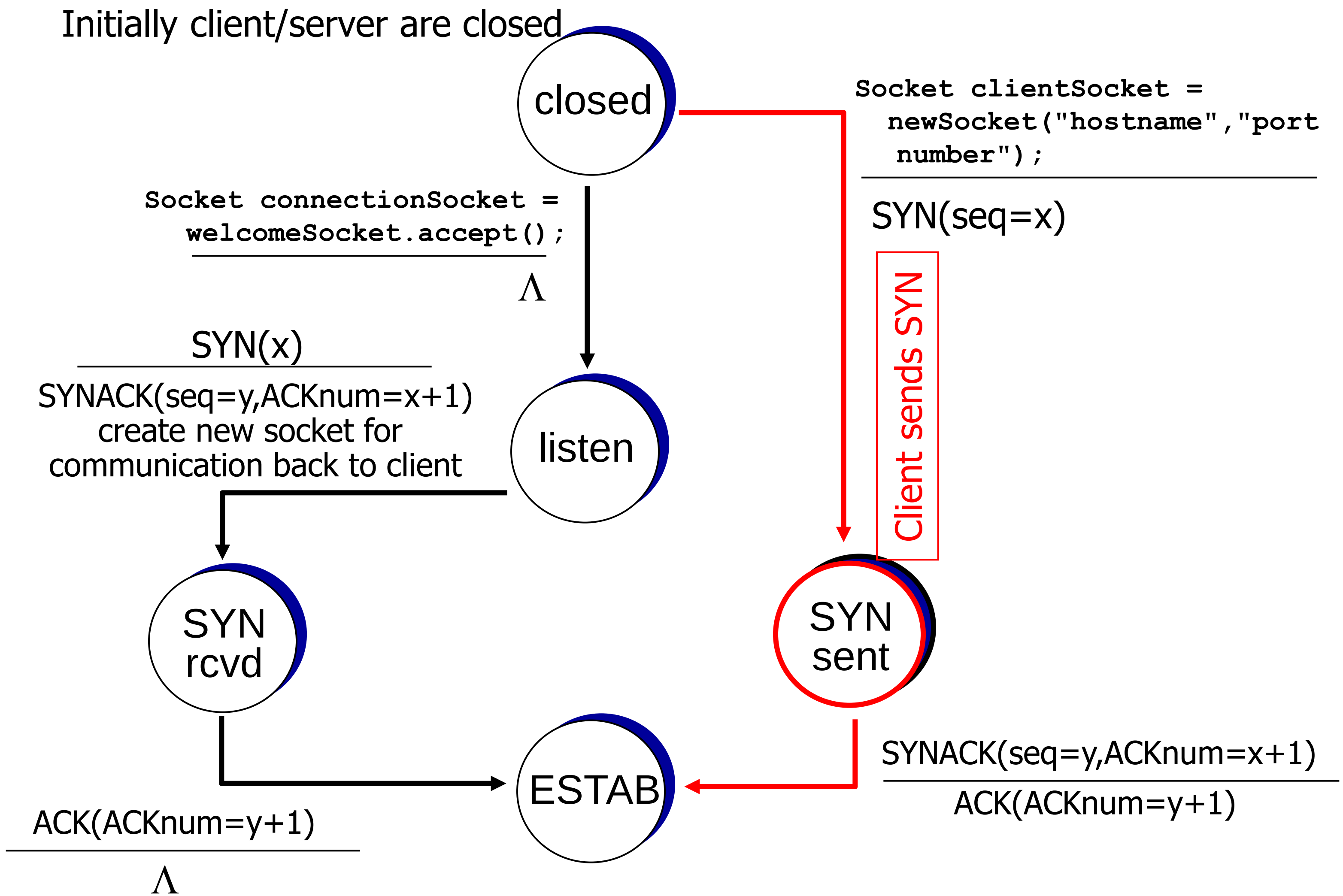
2-way handshake failure scenarios:



TCP 3-way handshake



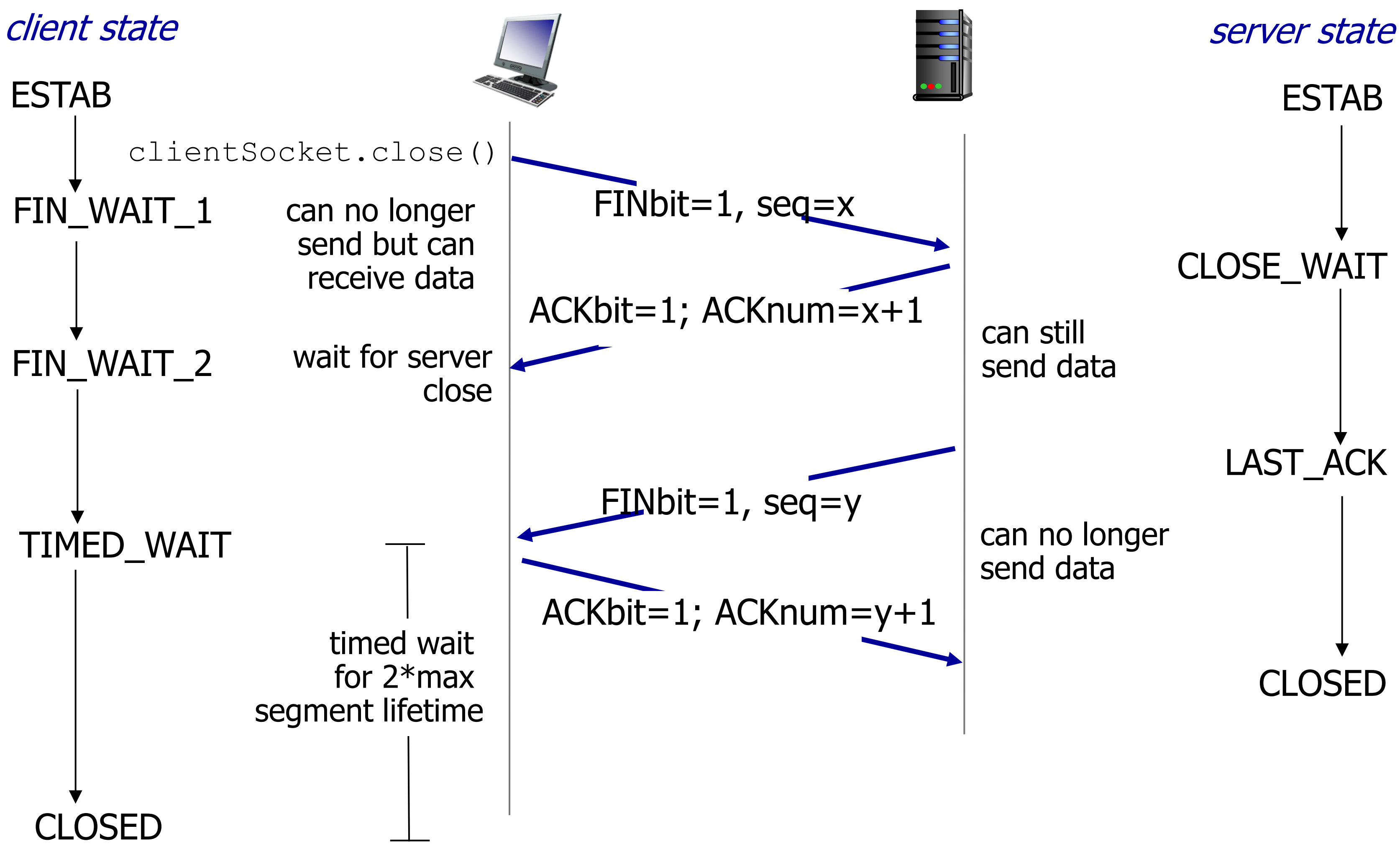
TCP 3-way handshake: FSM



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



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