UNIVERSITY OF WATERLOO

CS 456/656
Computer Networks

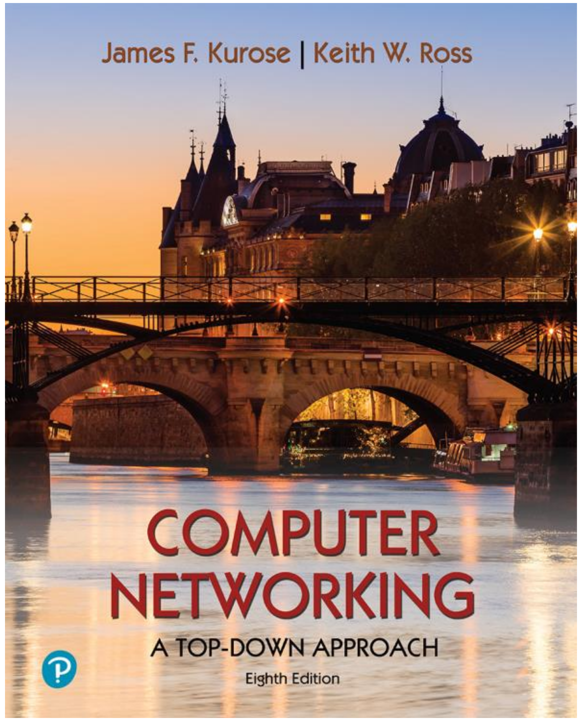
Lecture 7: Transport Layer – Part 3

Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

A note on the slides

Adapted from the slides that accompany this book.



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - connection management
  - reliable data transfer
  - flow control
- Principles of congestion control
- TCP congestion control

TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”

TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”

TCP: a widely-used reliable transport protocol

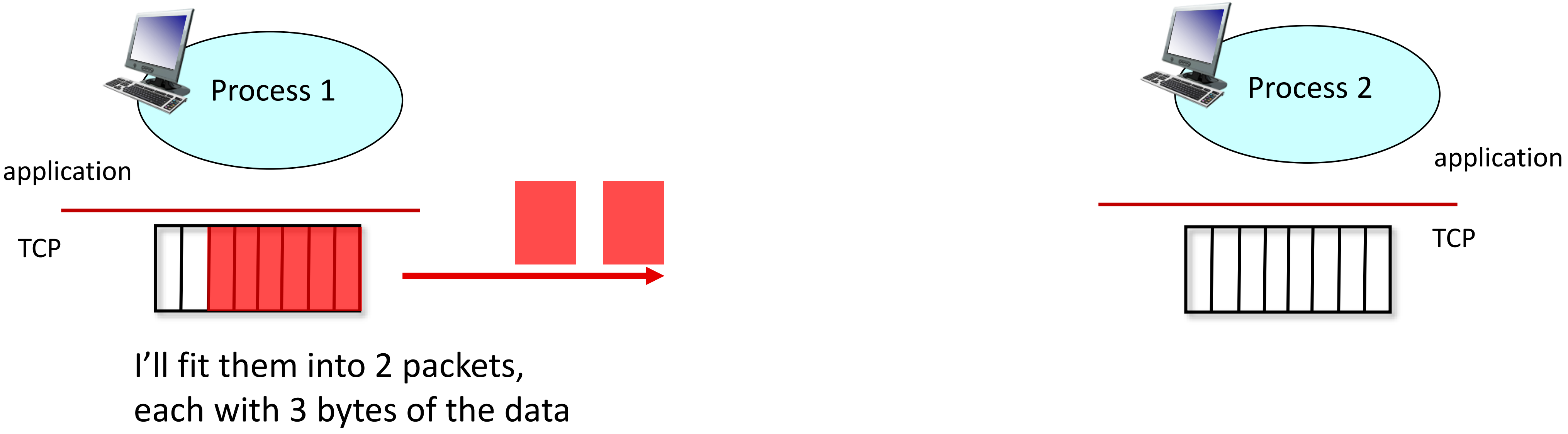
- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”

TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”

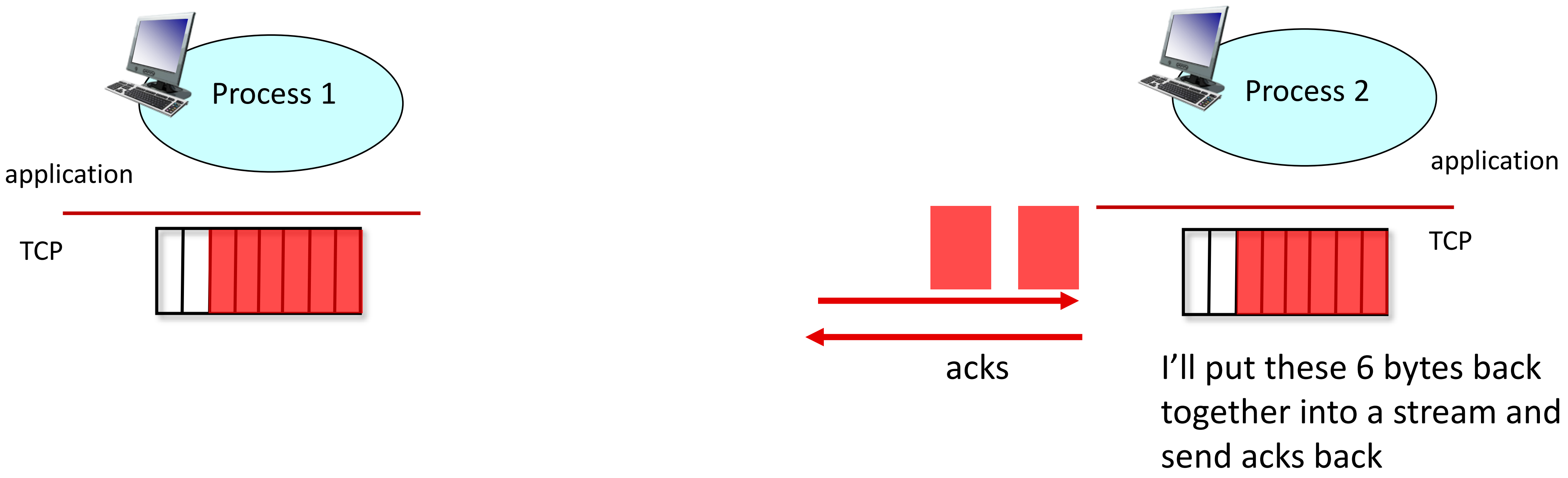
TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no "message boundaries"



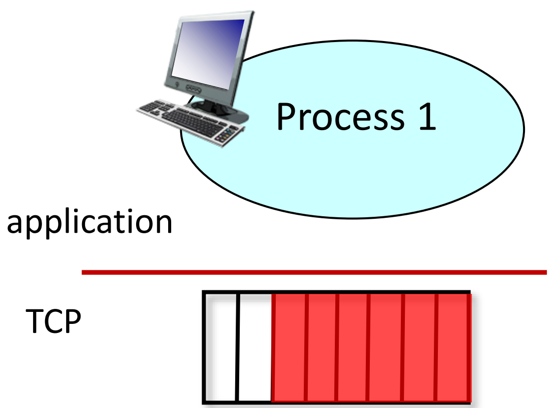
TCP: a widely-used reliable transport protocol

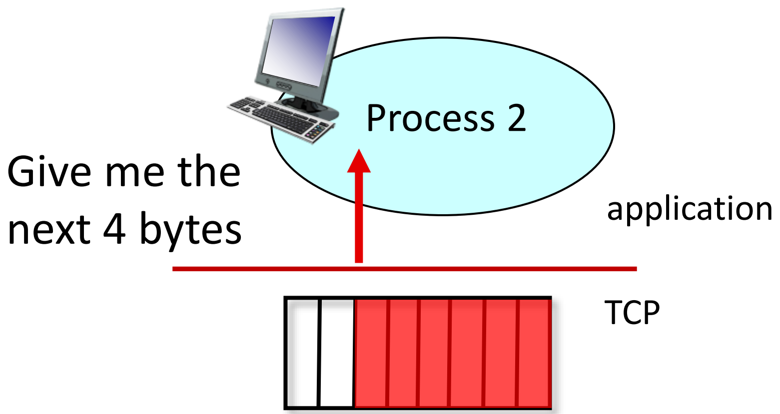
- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”



TCP: a widely-used reliable transport protocol

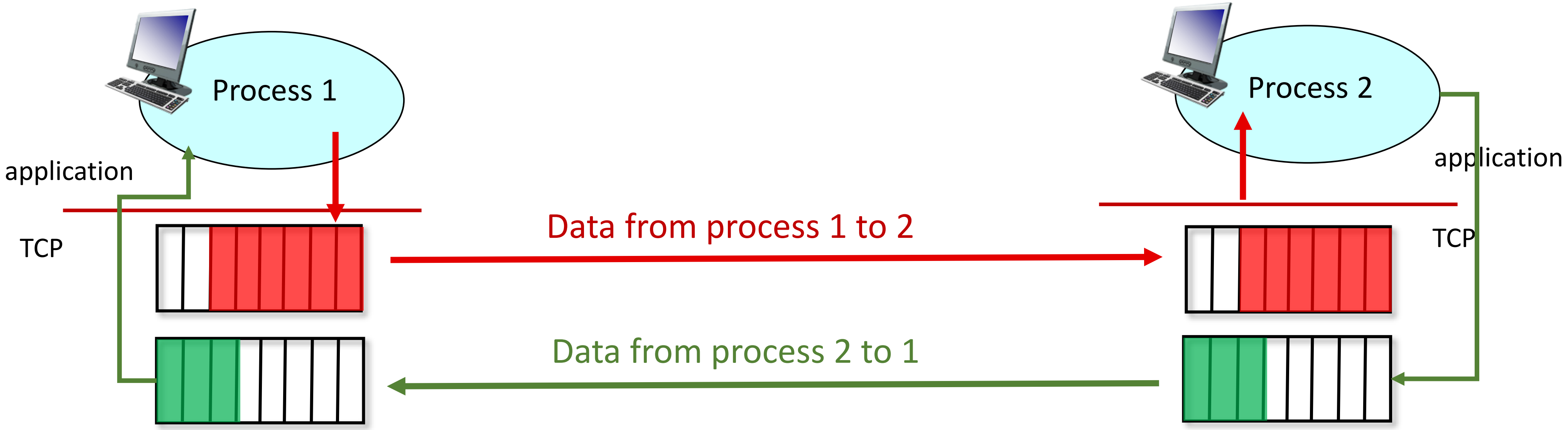
- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”





TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”
- full duplex data: Possible to send data both ways once the two processes establish a connection



TCP: a widely-used reliable transport protocol

- Guarantees reliable, in-order *byte steam:*
  - no “message boundaries”
- full duplex data:
  - Possible to send data both ways once the two processes establish a connection
- Uses the pipelining approach to reliable data transfer
  - A combination of techniques from Go-Back-N (cumulative acks) and Selective Repeat (only retransmitting presumably lost segment)
  - Performance optimizations like fast retransmit and delayed acks.

TCP: a widely-used reliable transport protocol

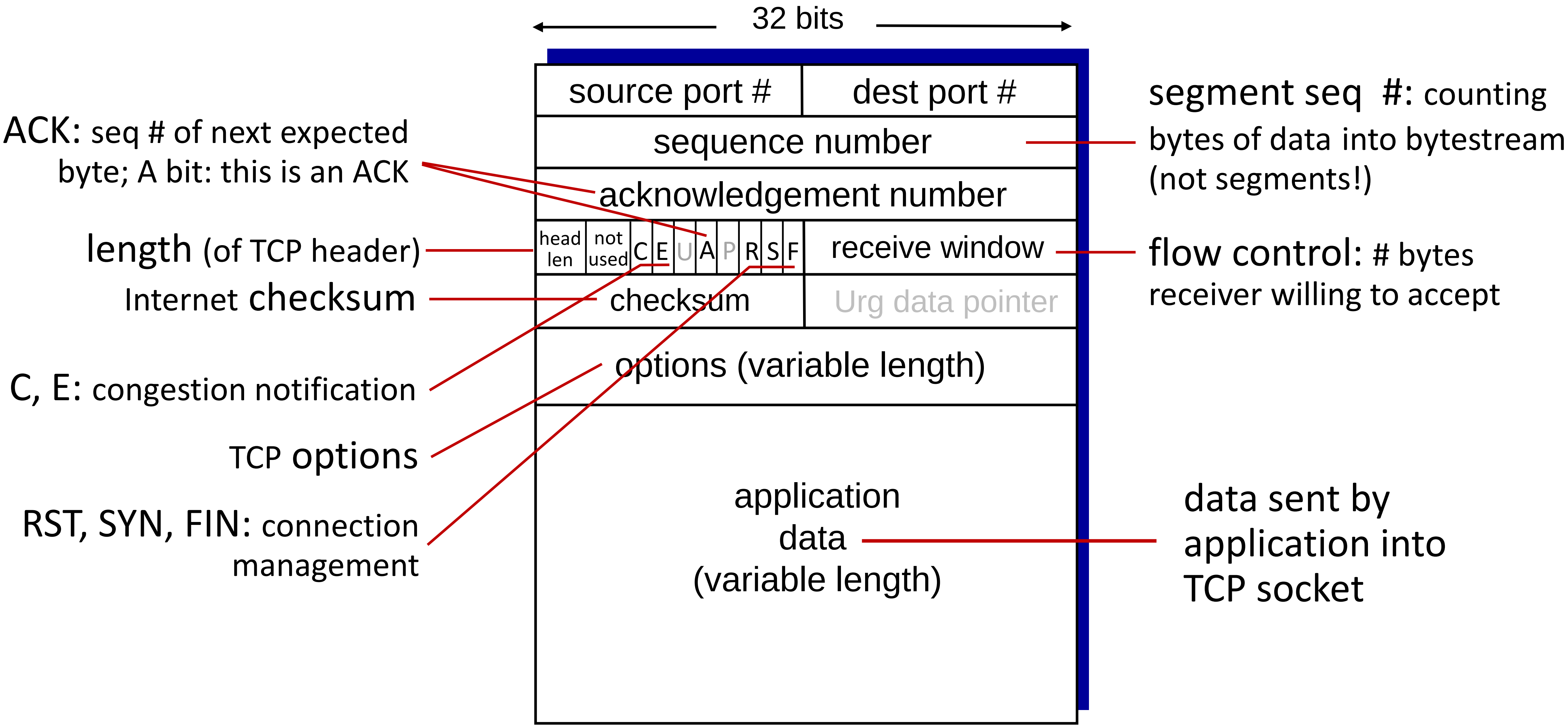
- Connection-oriented
  - Connection establishment: Control messages prior to data exchange to initialize the proper state in the communication endpoints
  - Connection tear-down: Control messages after data exchange to end connection

- Flow controlled
  - sender will not overwhelm receiver

Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP segment structure



Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - <u>segment structure</u>
  - <u>reliable data transfer</u>
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP reliable data transfer

TCP uses all the reliable data transfer tools we have discussed!

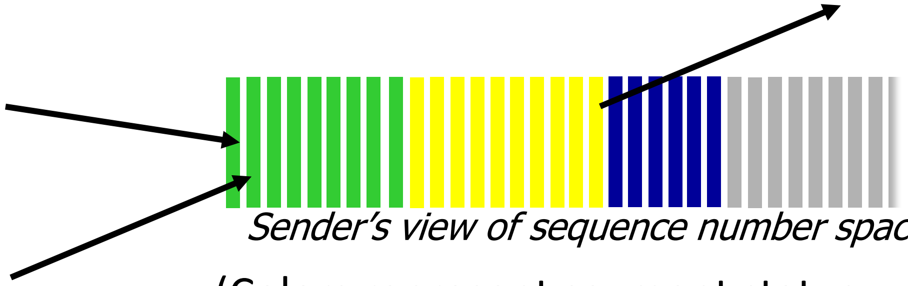
- Checksum
- Sequence number
- Receiver feedback (ACK)
- Timer
- Sliding window/pipelining

TCP sequence numbers – one for every byte

- The interface between a sending process and TCP is a byte stream.
- TCP assigns a sequence number to *<u>every byte</u>*
  - <u>As opposed to every segment</u>, as we discussed in the last lecture
- It keeps track of the "status" of every byte
  - Is it sent yet? Is it acknowledged yet?

The Nth <u>byte</u> has sequence number init_seq + N - 1

First <u>byte</u> has sequence number init_seq (initial sequence number)



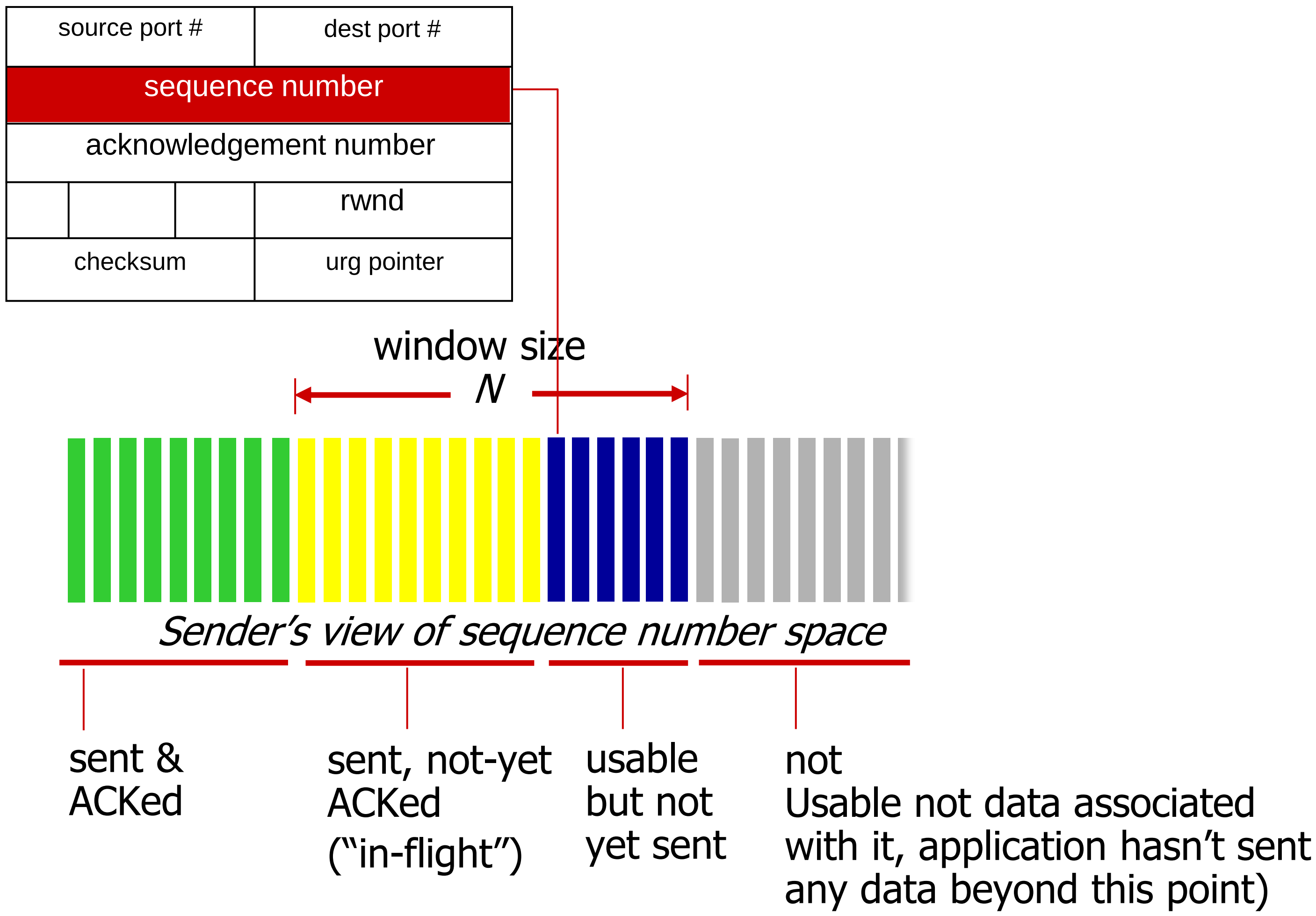
Sender's view of sequence number space

(Colors represent segment status -- see next slide)

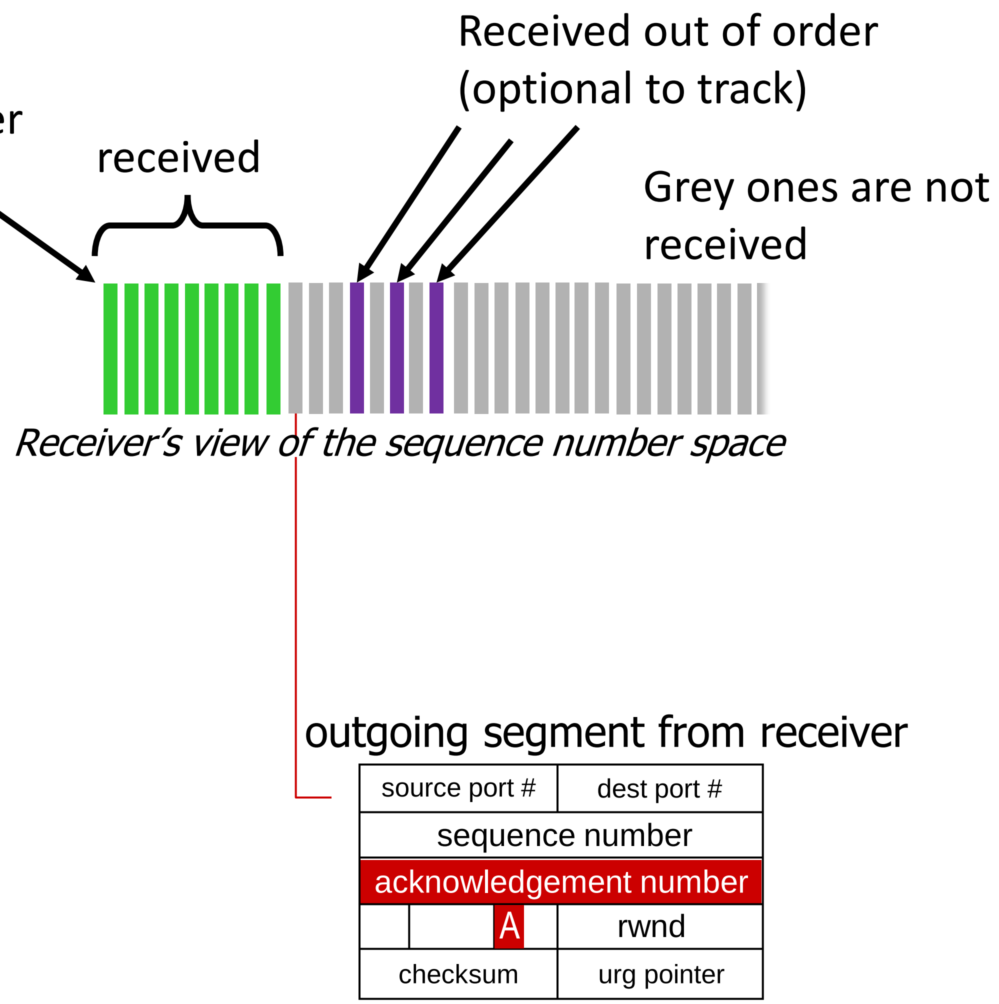
Next <u>byte</u> has sequence number init_seq + 1

TCP sequence numbers

outgoing segment from sender



TCP ACKs

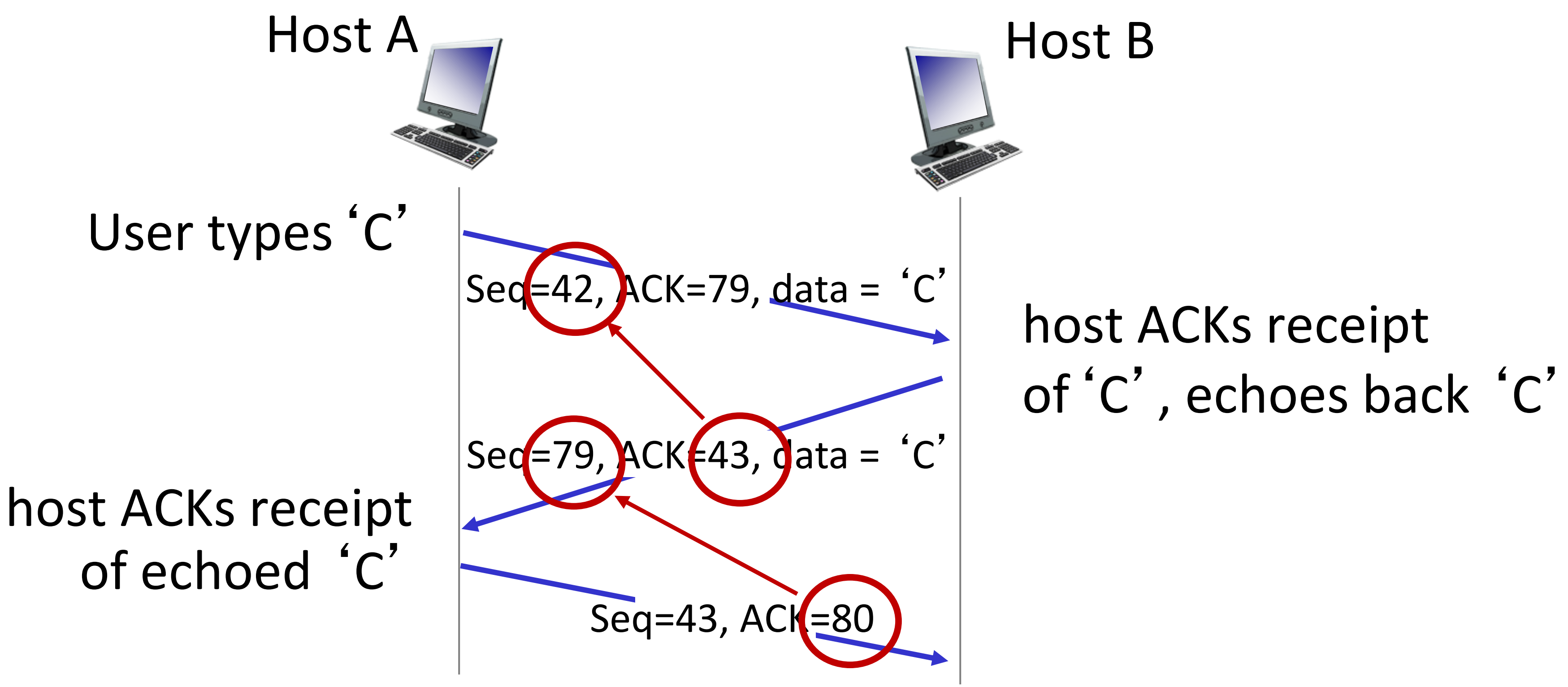


- Cumulative ACK
  - Has seq number of next expected in-order byte
- ACK(n) means:
  - All bytes in [init_seq, n – 1] are received.
  - The receiver is expecting byte n next
- Note the difference from Go-Back-N ack

Q: What about out-of-order segments?

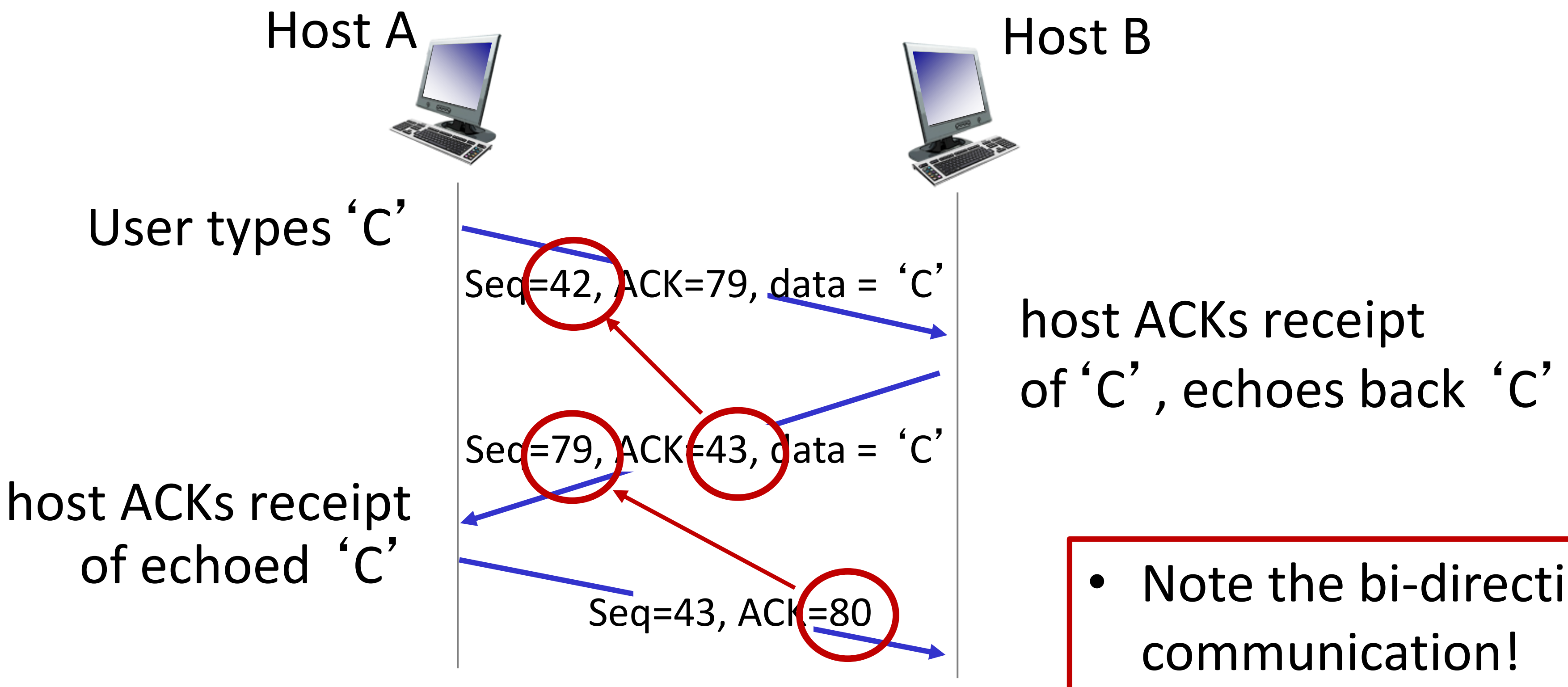
- *A:* TCP spec doesn't specify, - up to implementor

TCP sequence numbers, ACKs



simple telnet scenario

TCP sequence numbers, ACKs



simple telnet scenario

- Note the bi-directional communication!
- There are two data streams:
  - one in each direction
  - each with its own sequence number space

TCP Sender (simplified)

event: data received from application

- create segment with seq #
- seq # is byte-stream offset of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unACKed segment
  - expiration interval: **`TimeOutInterval`**

event: timeout

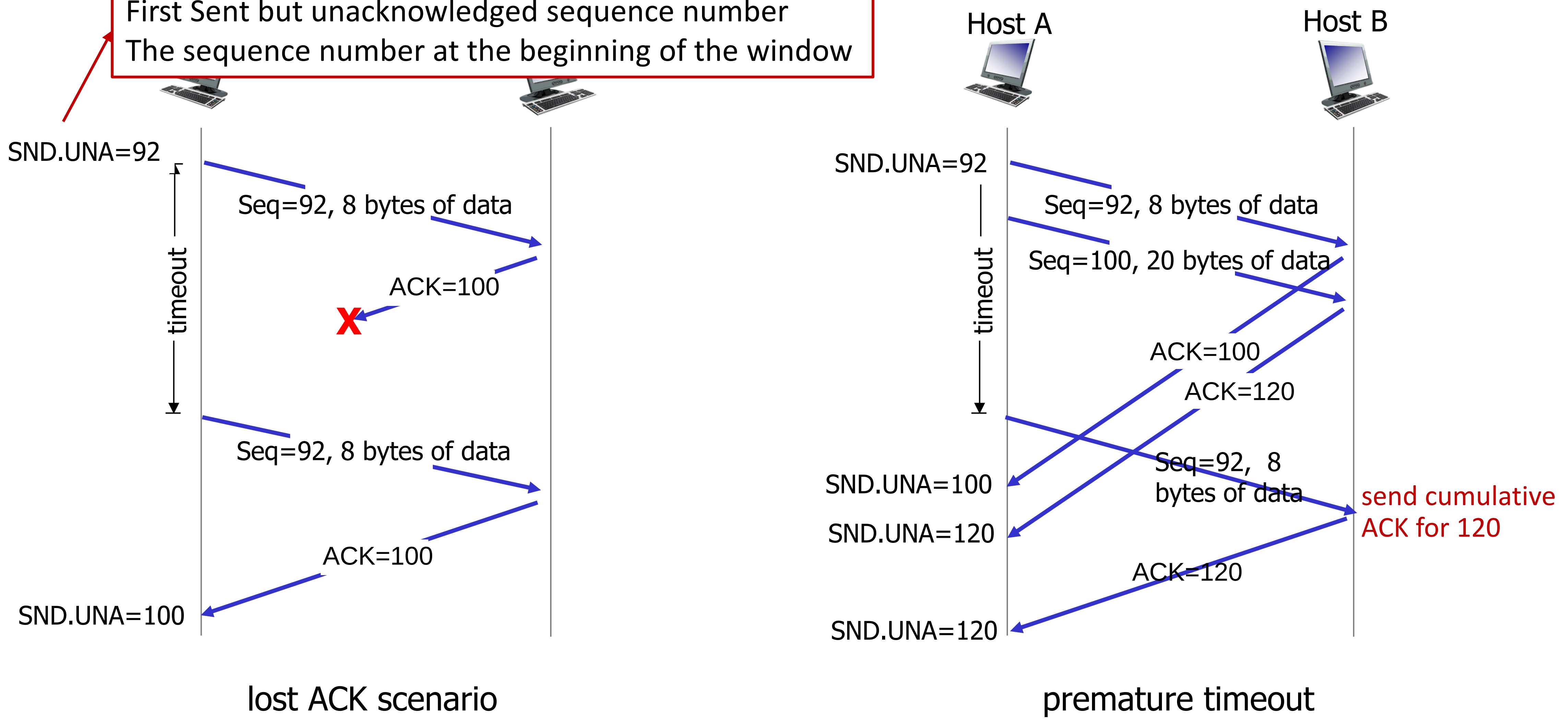
- retransmit segment that caused timeout
- restart timer

event: ACK received

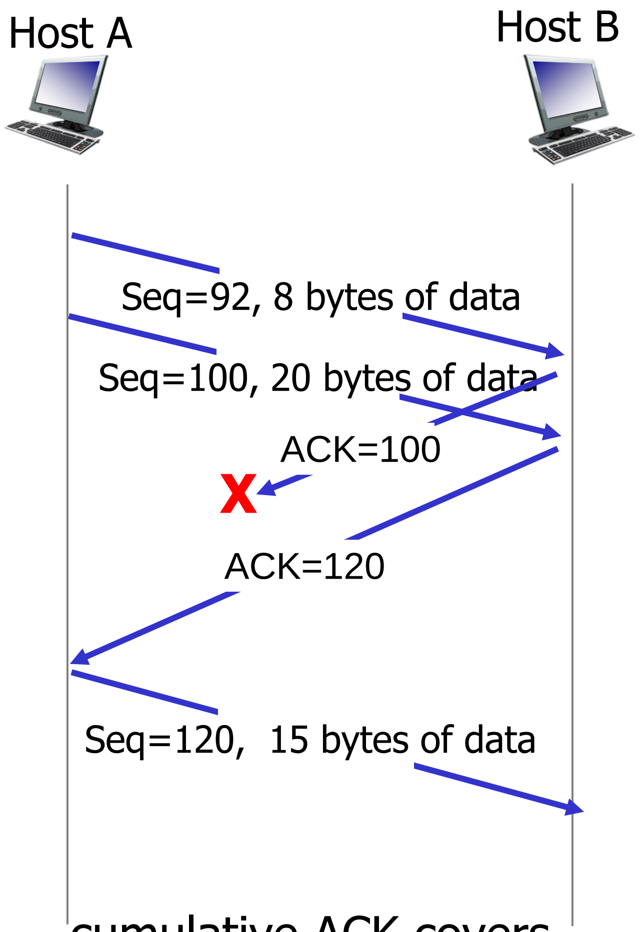
- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - restart timer if there are still unACKed segments

TCP: retransmission scenarios

First Sent but unacknowledged sequence number
The sequence number at the beginning of the window

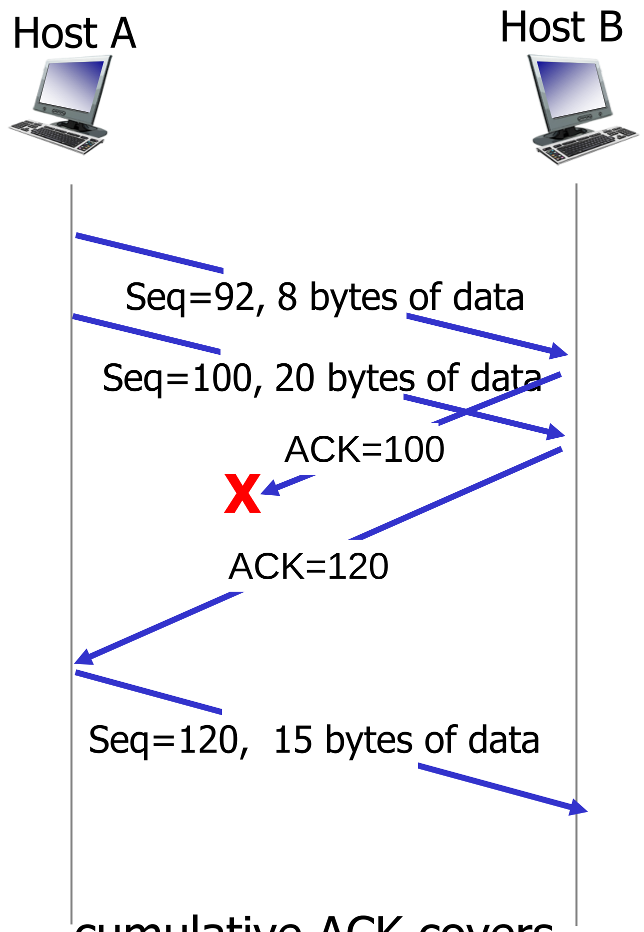


TCP: retransmission scenarios



cumulative ACK covers
for earlier lost ACK

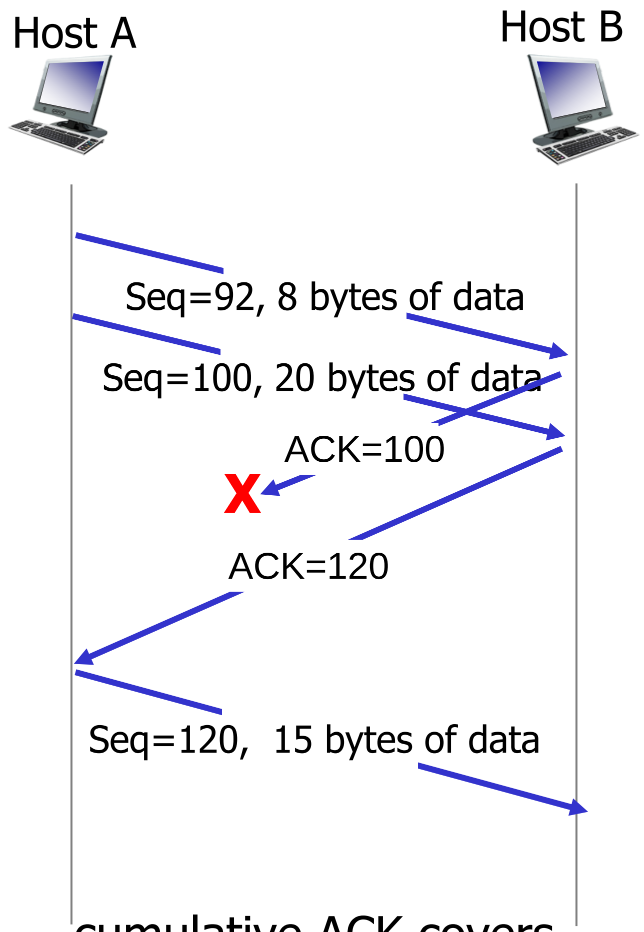
TCP: retransmission scenarios



cumulative ACK covers
for earlier lost ACK

- (short) in class exercise:
  - What is the value of SND.UNA after sending and receiving each packet?

TCP: retransmission scenarios



cumulative ACK covers for earlier lost ACK

- Q: How is TCP similar to Go-Back-N? How is it different? How about Selective Repeat?

Knowledge Check

- Make sure you understand and can complete a TCP send and receive timeline.
- This includes, but is not limited to
  - sequence and acknowledgement numbers on packets going back and forth
  - how the sender and receiver view of the sequence number space changes as a result of packets being sent and received (e.g., status of the bytes, position of the sliding window, etc.)

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

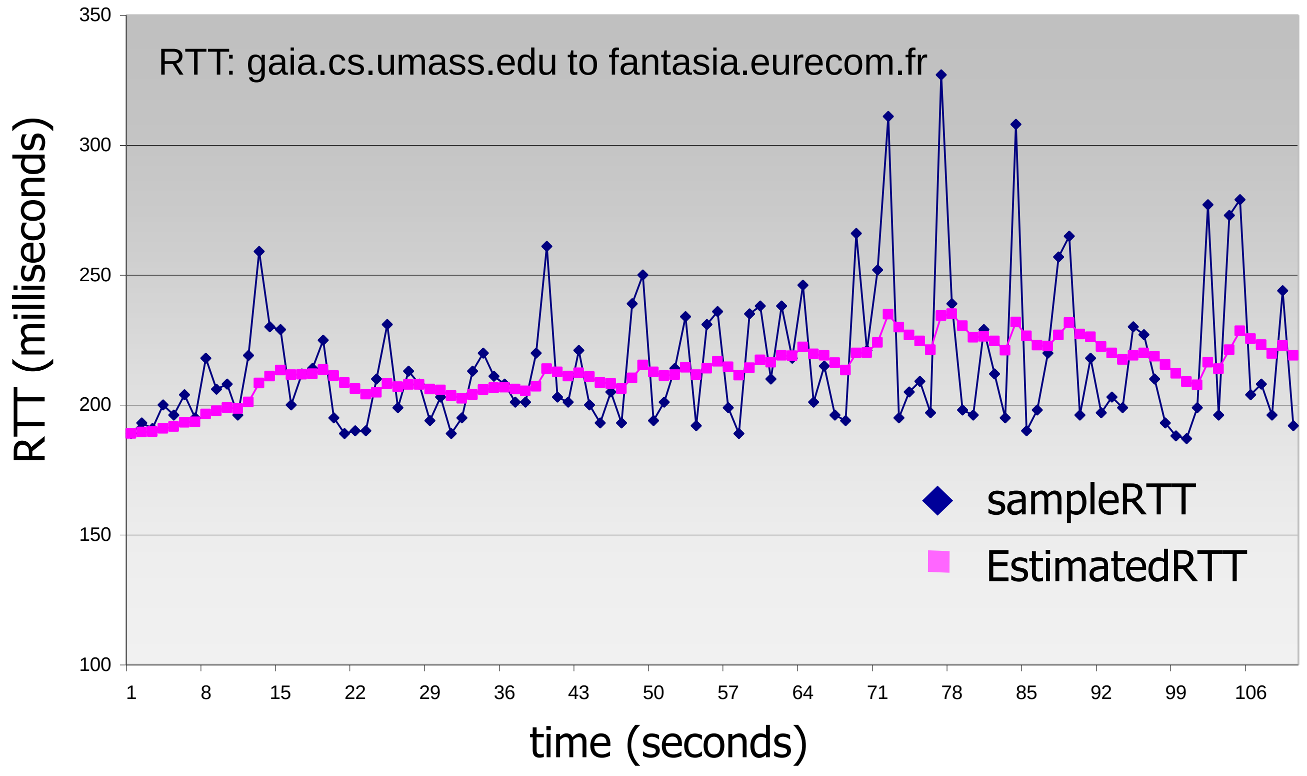
Q: how to estimate RTT?

- `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 (EWMA)
- 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`**: want a larger safety margin

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

estimated RTT   "safety margin"

- **`DevRTT`**: EWMA of **`SampleRTT`** deviation from **`EstimatedRTT`**:

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

(typically, $\beta = 0.25$)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Performance optimizations for TCP

- So far, we have covered “the basics” of TCP’s rdt
  - Sequence number
  - Cumulative ACKs
  - Pipelined segments
  - Retransmission timer
- Next, we will discuss some optimizations

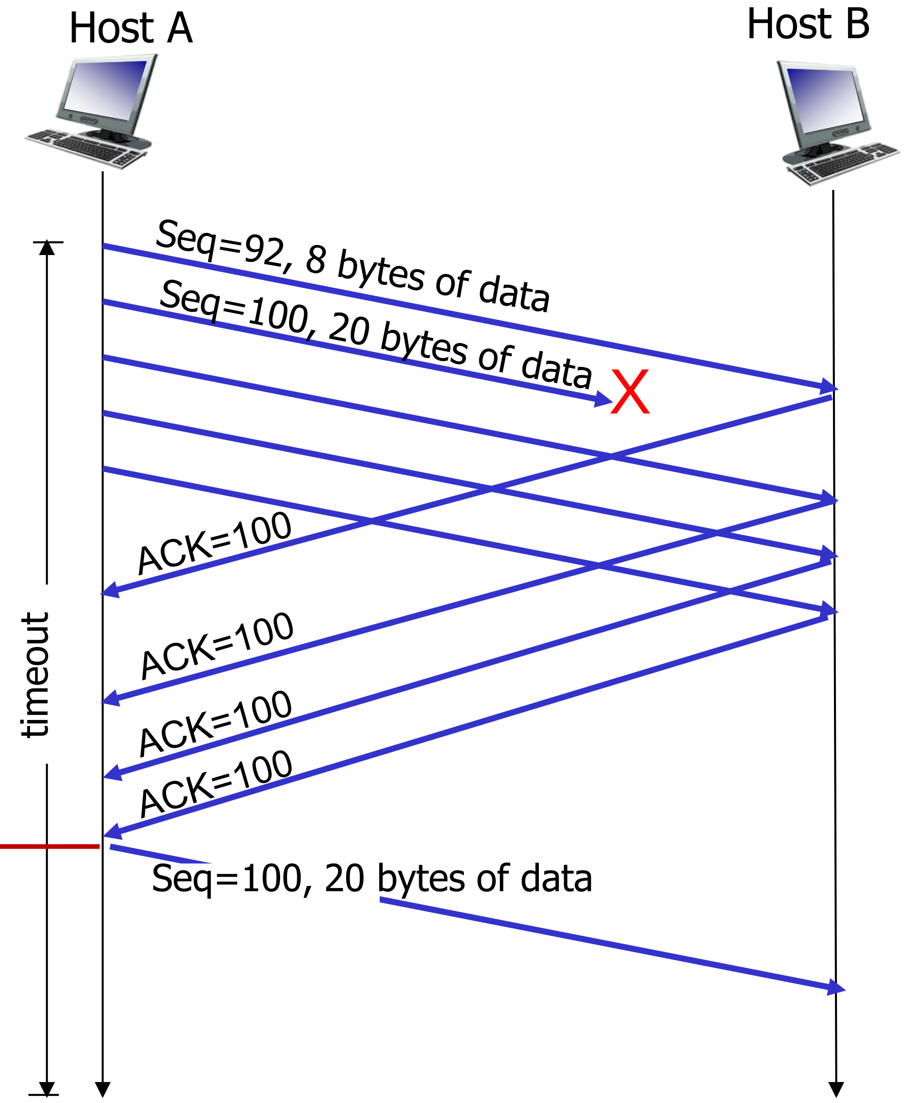
Optimization 1: Fast Retransmit

TCP fast retransmit

if sender receives 3 additional ACKs for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

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

Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



Optimization 2: Delayed ACKs

- Instead of generating an ACK in response to every segment the moment it arrives
  - Wait for some time to see if there is another segment right afterwards
  - Create one ACK for both.
- Benefits?
  - Saves bandwidth
- Disadvantages?
  - Increases delay in responding to the sender.

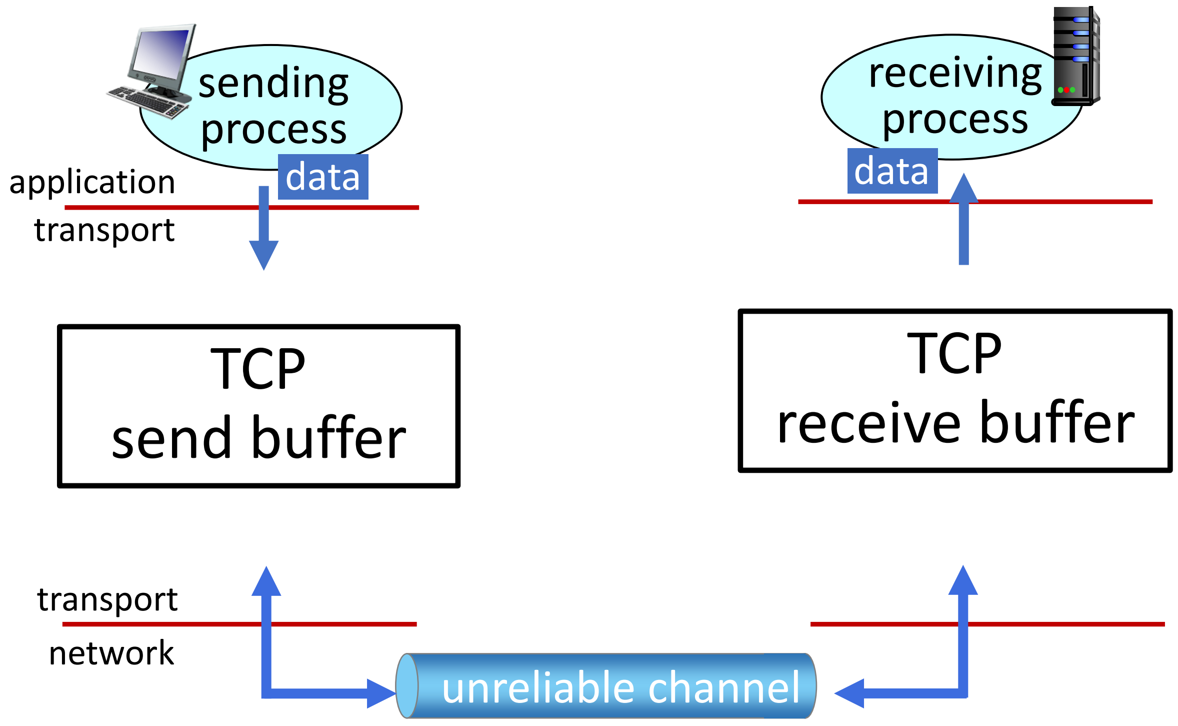
Optimizations 2: Delays ACKs (cont.)

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

Transport layer: roadmap

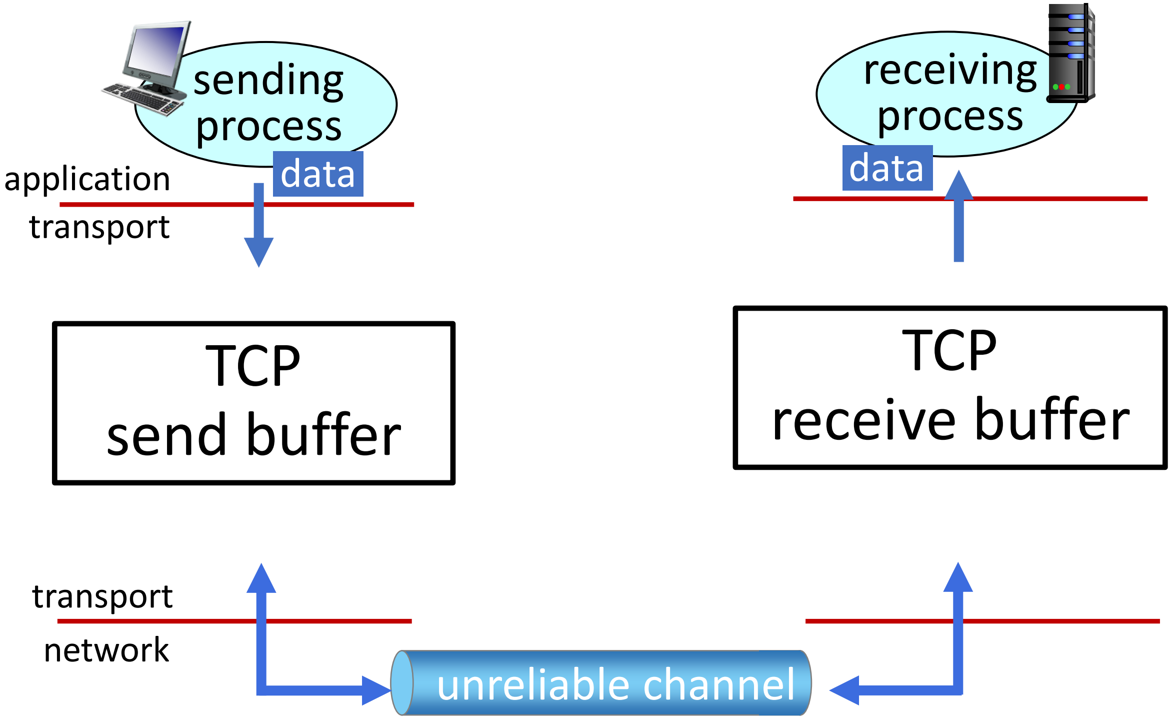
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

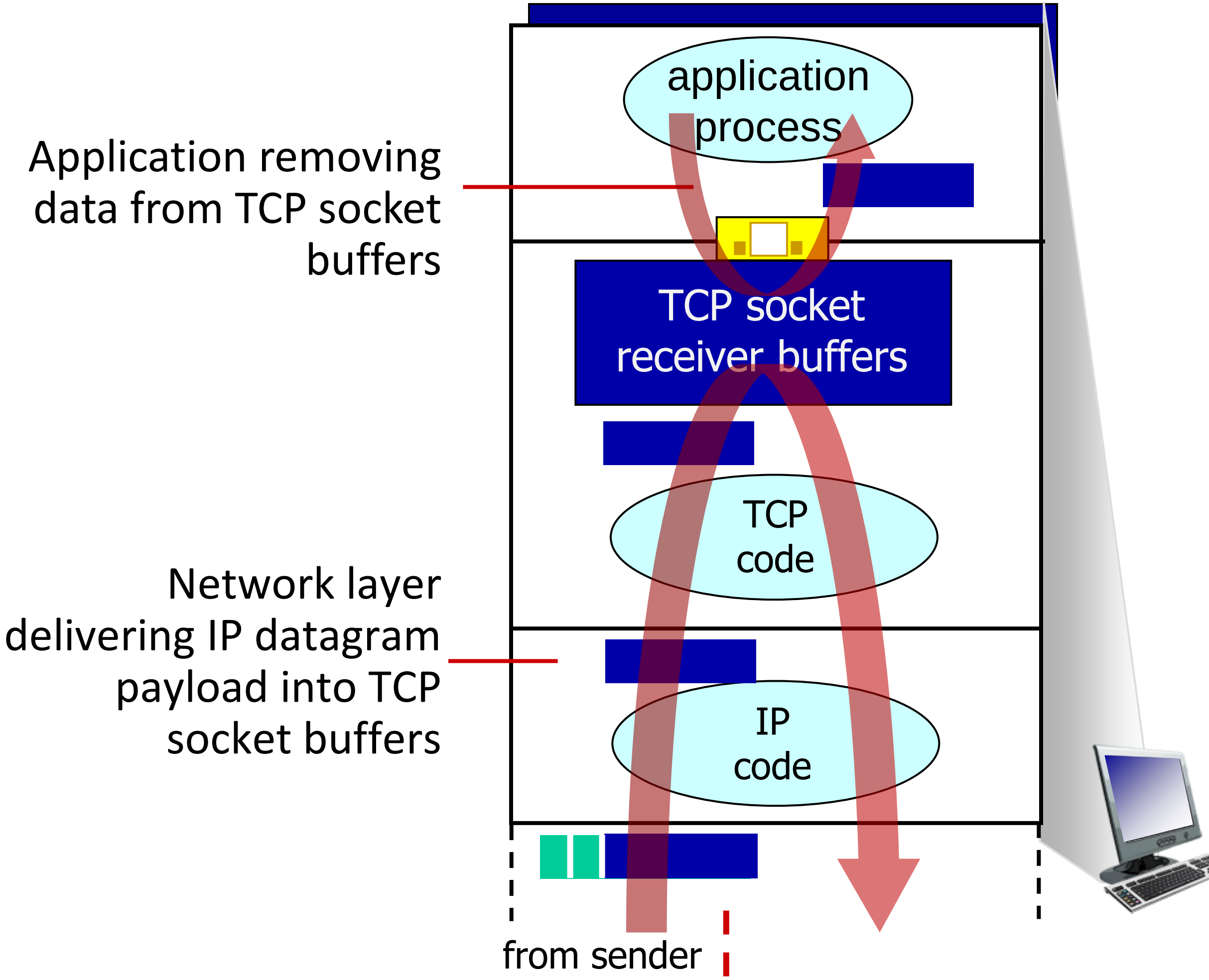
TCP flow control



- The send buffer holds the data the application sends to TCP until it is delivered
- The receive buffer holds the data TCP receives from the network until it is delivered to the application

TCP flow control



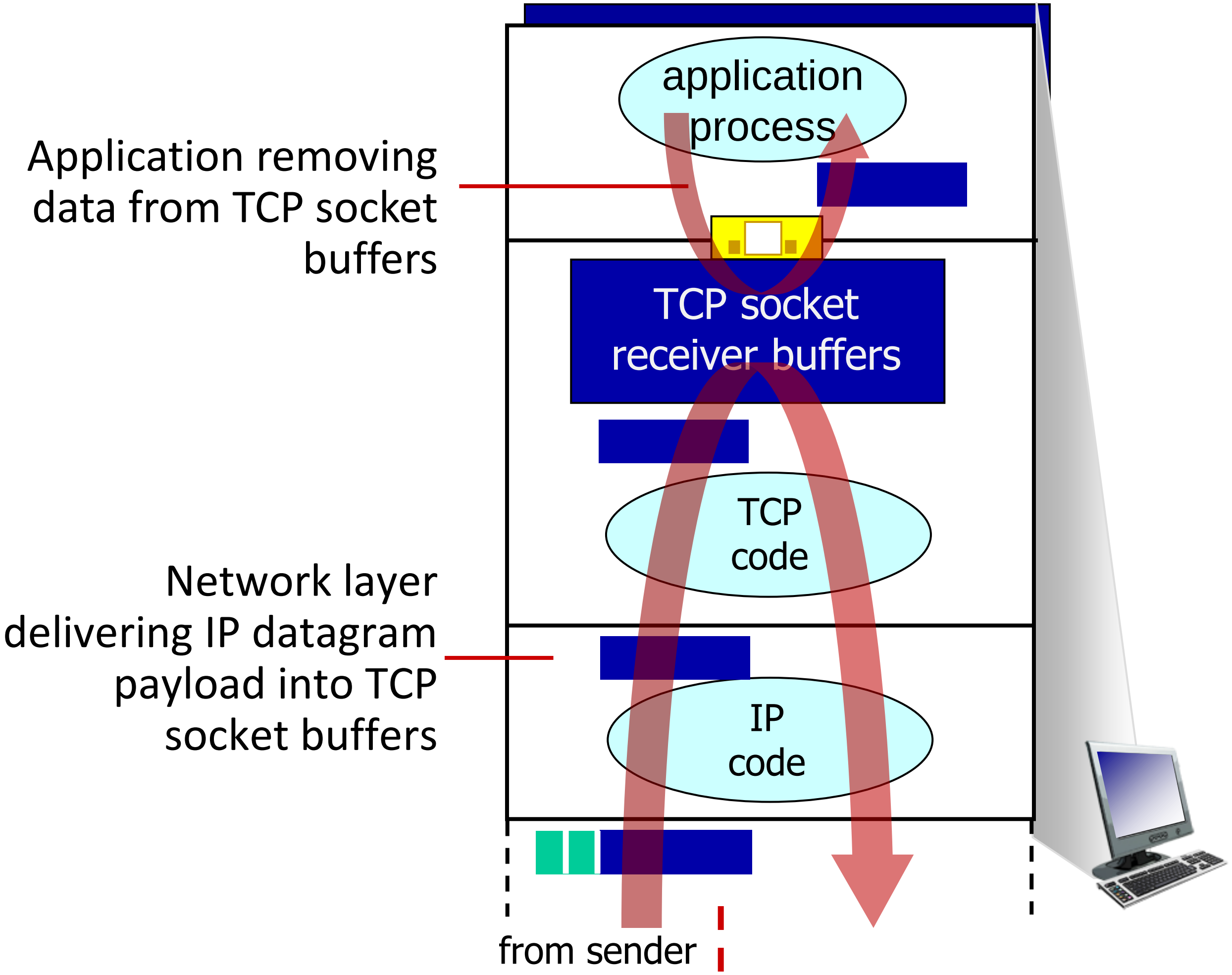


receiver protocol stack

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

TCP flow control

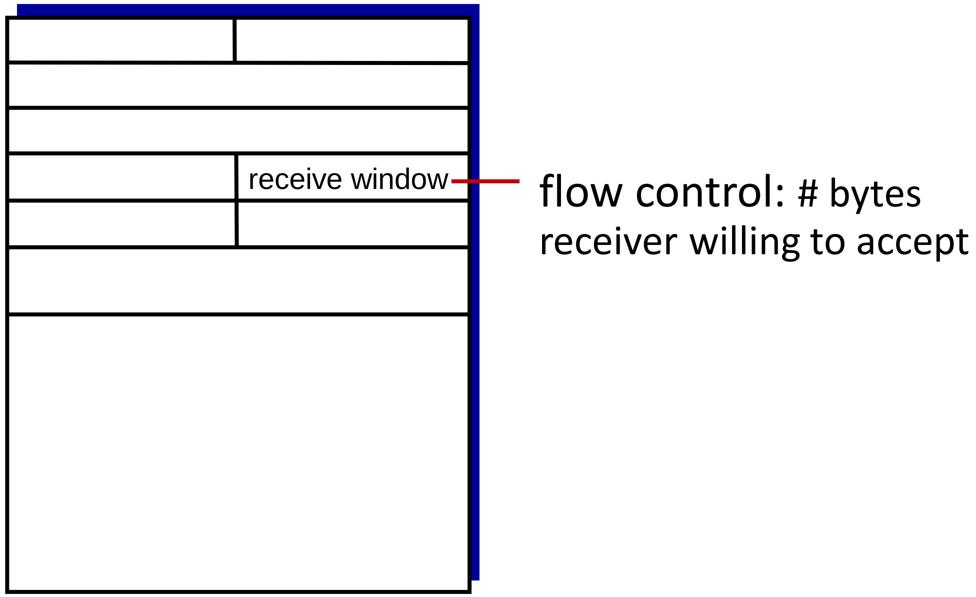
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

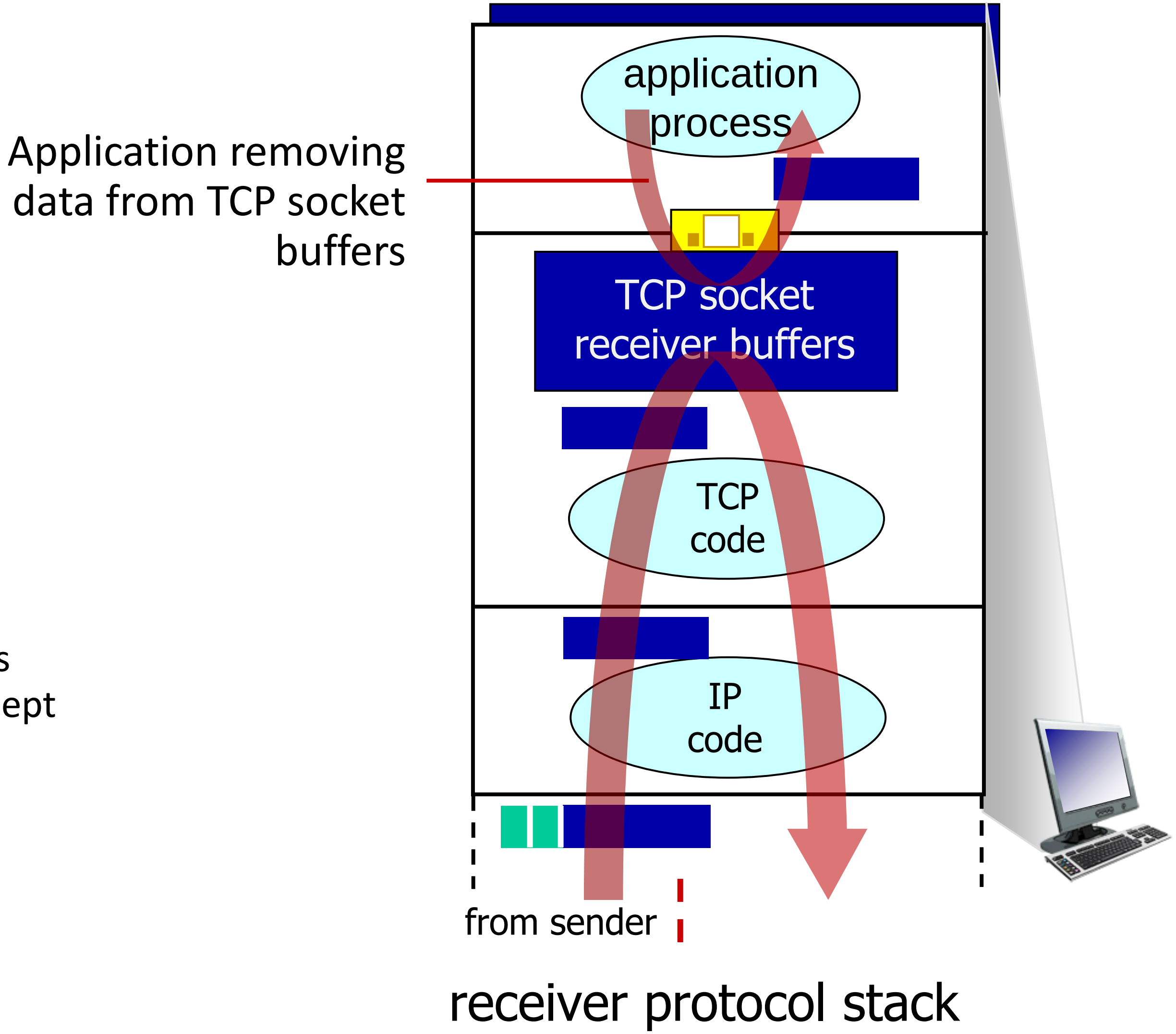


receiver protocol stack

TCP flow control

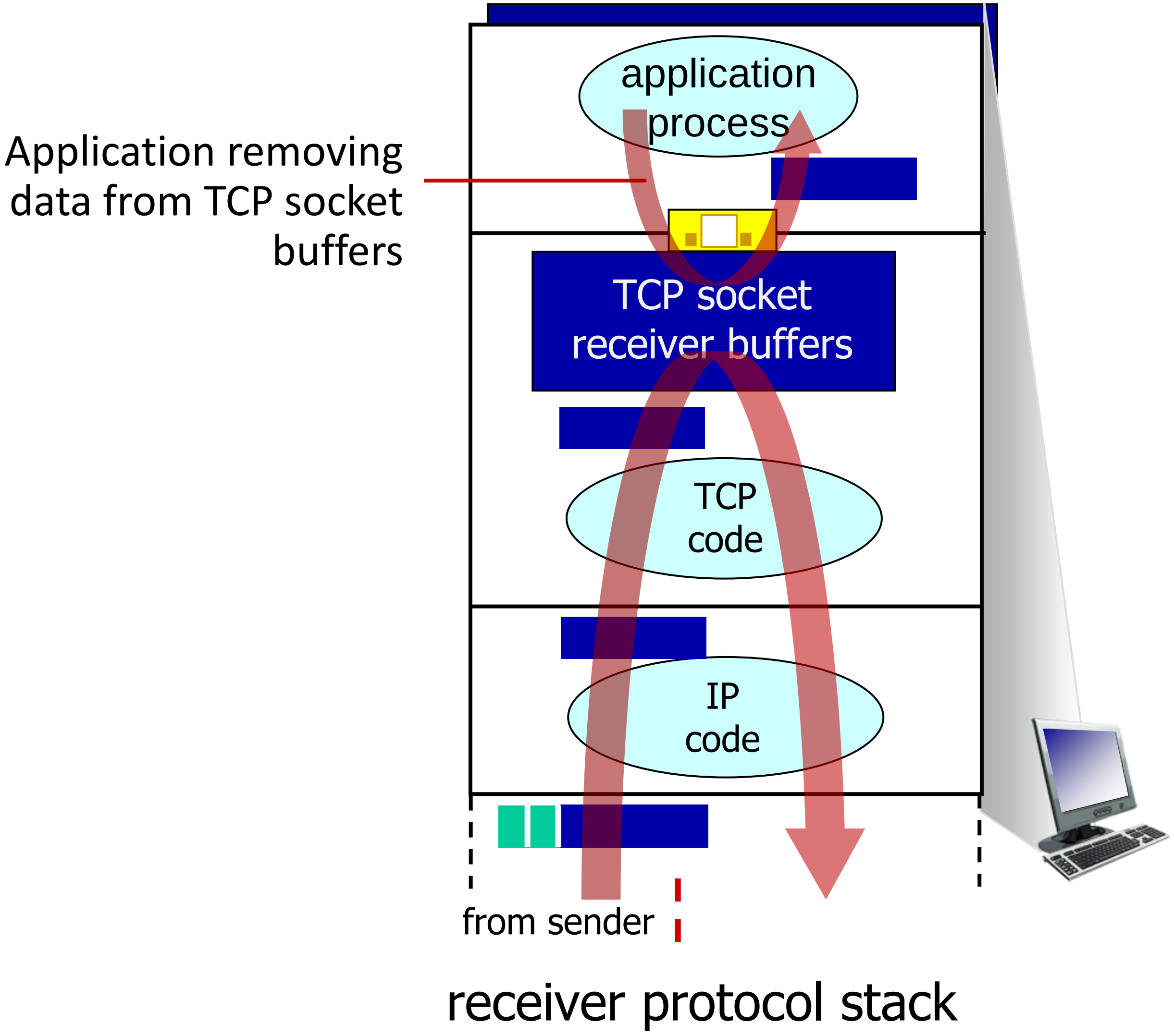
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





TCP flow control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

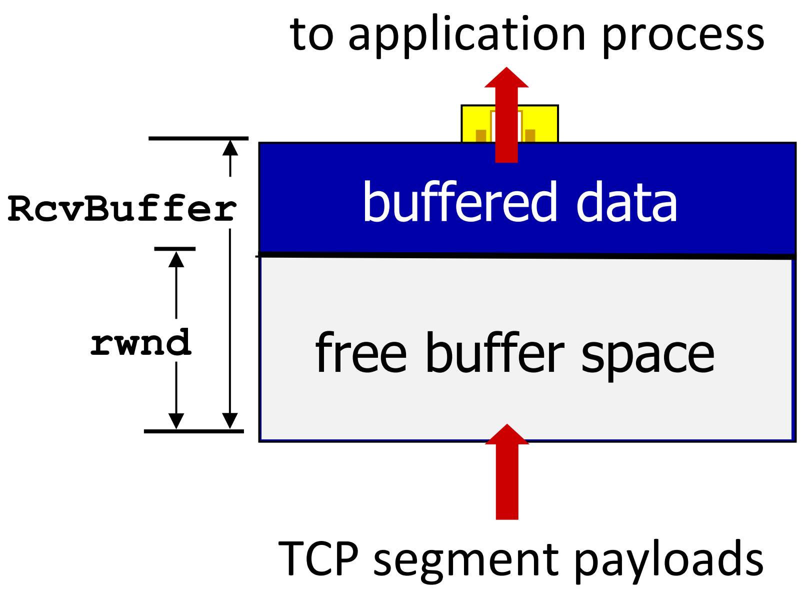


flow control

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

TCP flow control

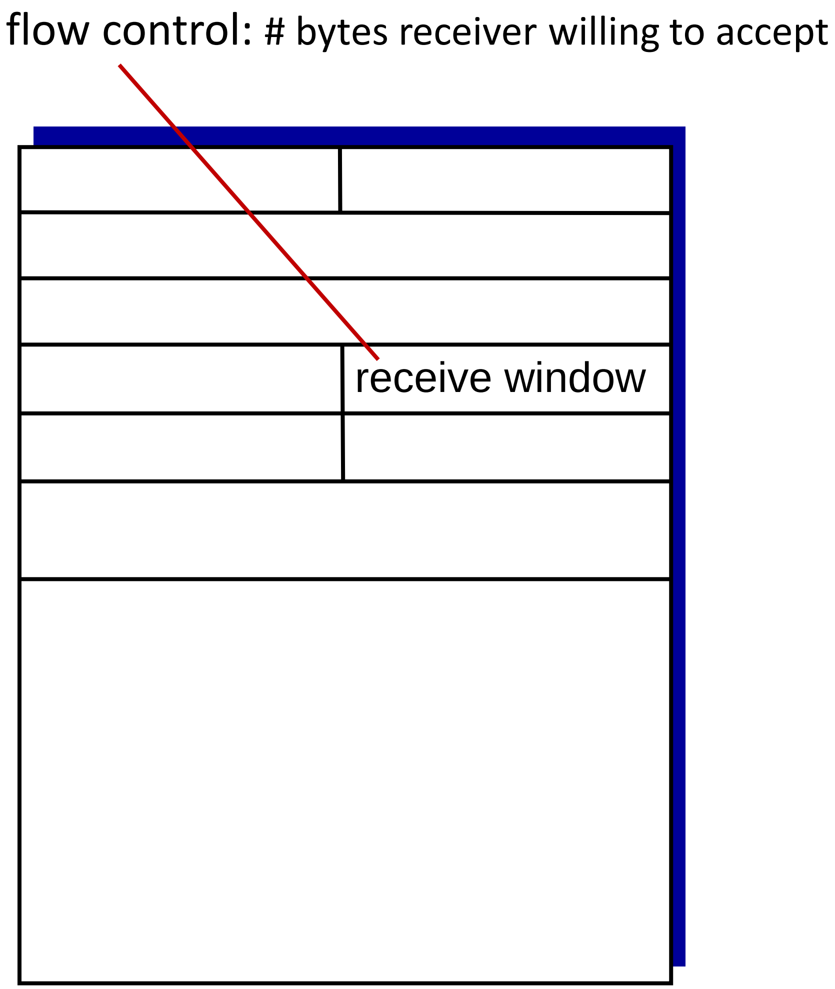
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

TCP flow control

- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



TCP segment format

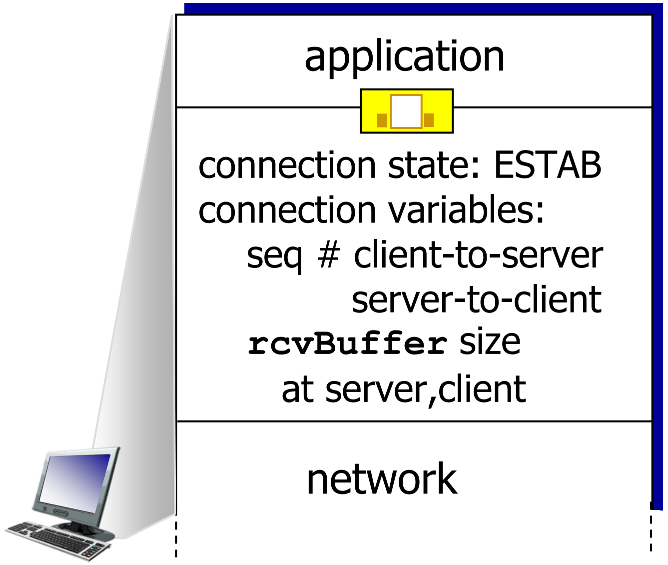
Transport layer: roadmap

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

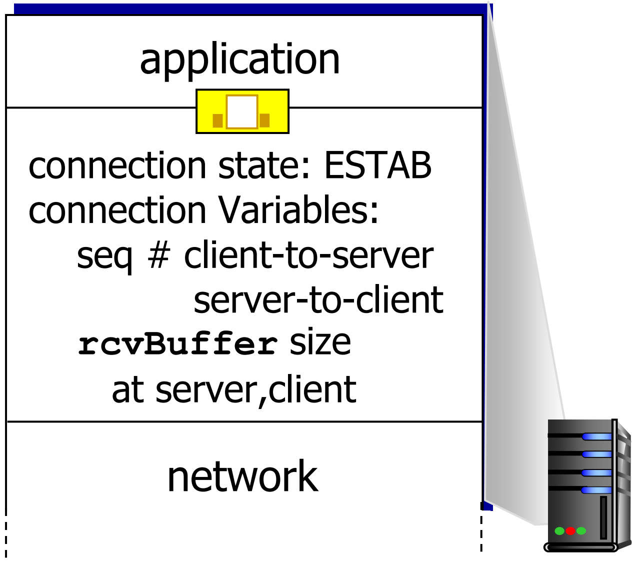
TCP connection management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters (e.g., starting seq #s)




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




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

TCP 3-way handshake

Pay close attention to sequence and ack numbers during handshake

Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

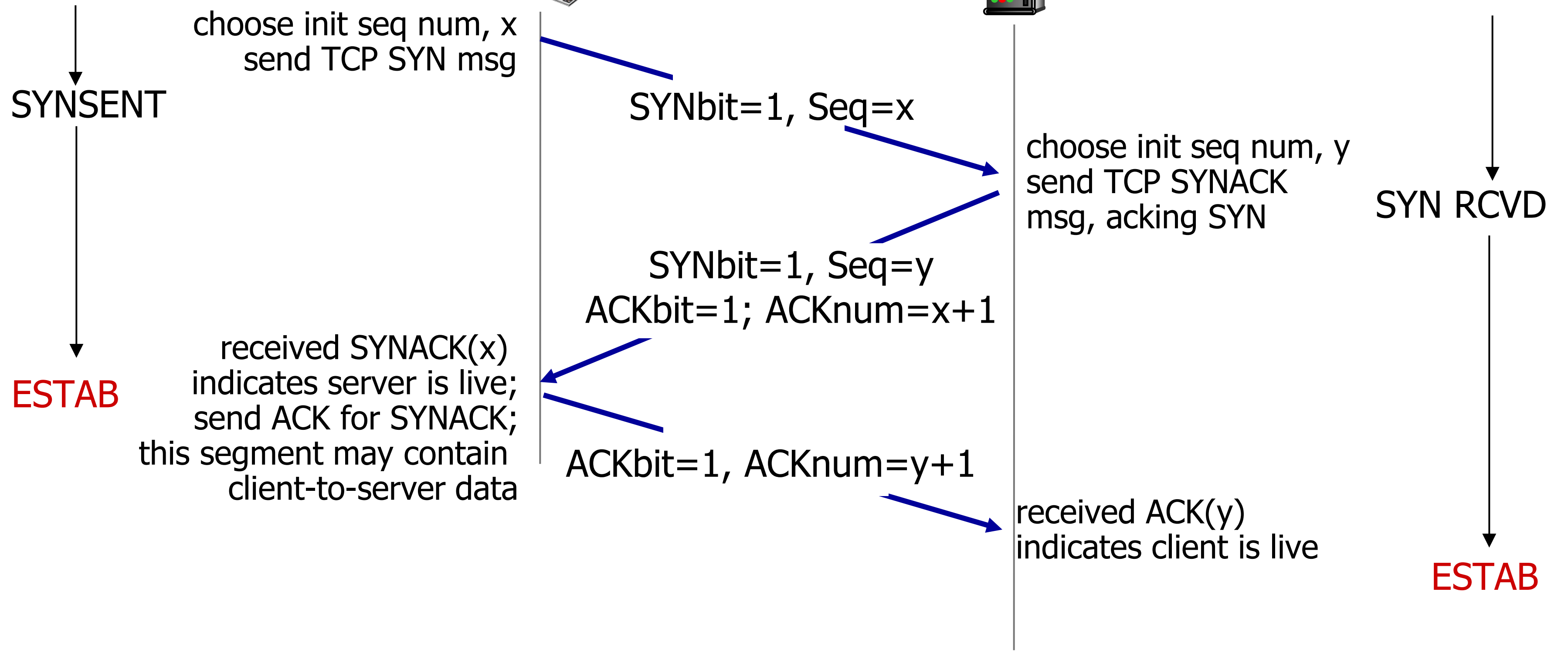
LISTEN

```
clientSocket.connect((serverName,serverPort))
```

Server state

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

LISTEN



A human 3-way handshake protocol

Closing a TCP 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

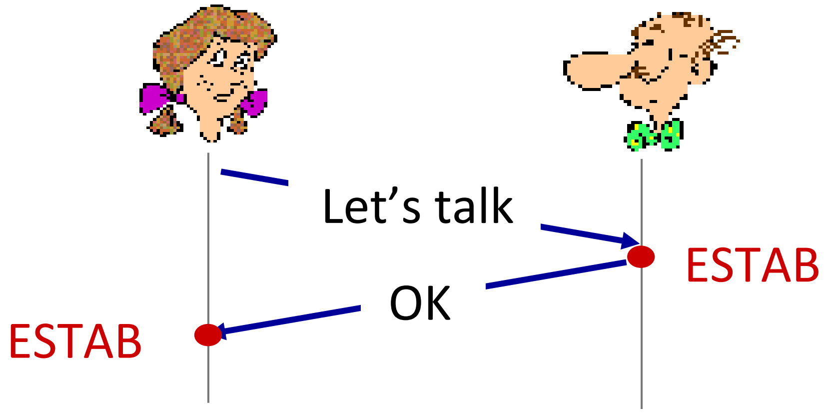
Knowledge Check

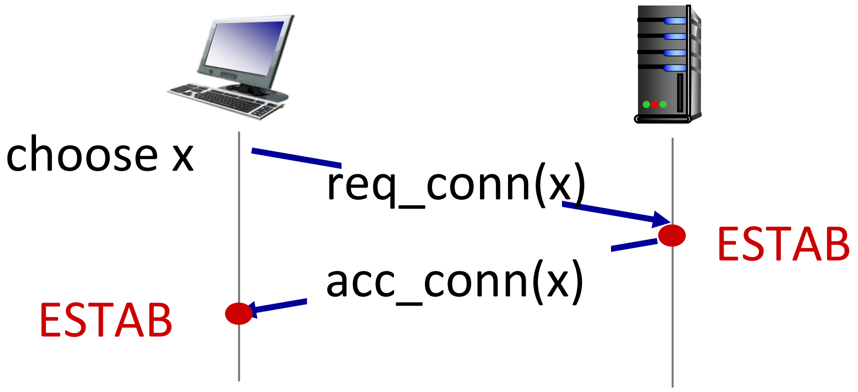
- Make sure you understand and can complete a TCP connection timeline
  - From connection establishment, through reliable data transfer (with optimizations and flow control), to connection tear-down
- This includes, but is not limited to
  - sequence and acknowledgement numbers on packets going back and forth
  - how the sender and receiver view of the sequence number space changes as a result of packets being sent and received (e.g., status of the bytes, position of the sliding window, etc.)

Additional Slides

Agreeing to establish a connection

2-way handshake:



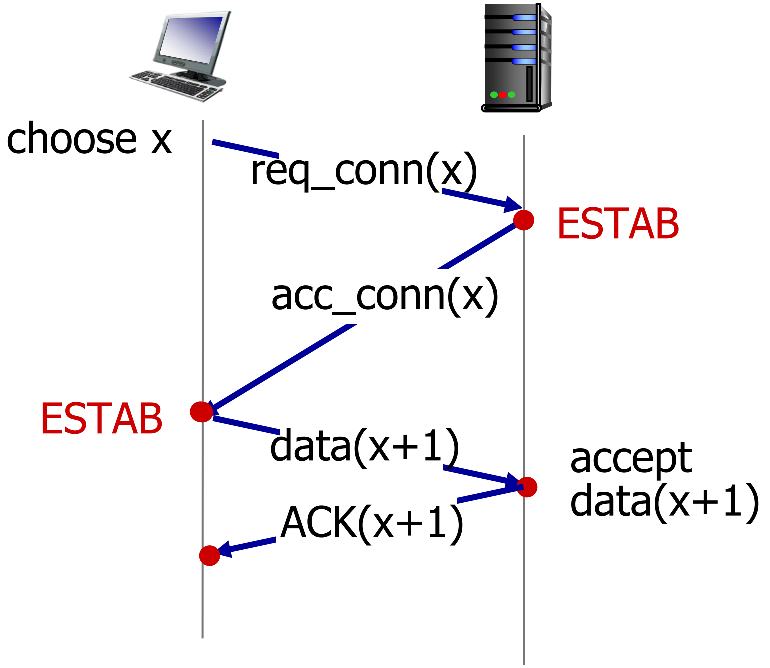


Q: will 2-way handshake always work in network?

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

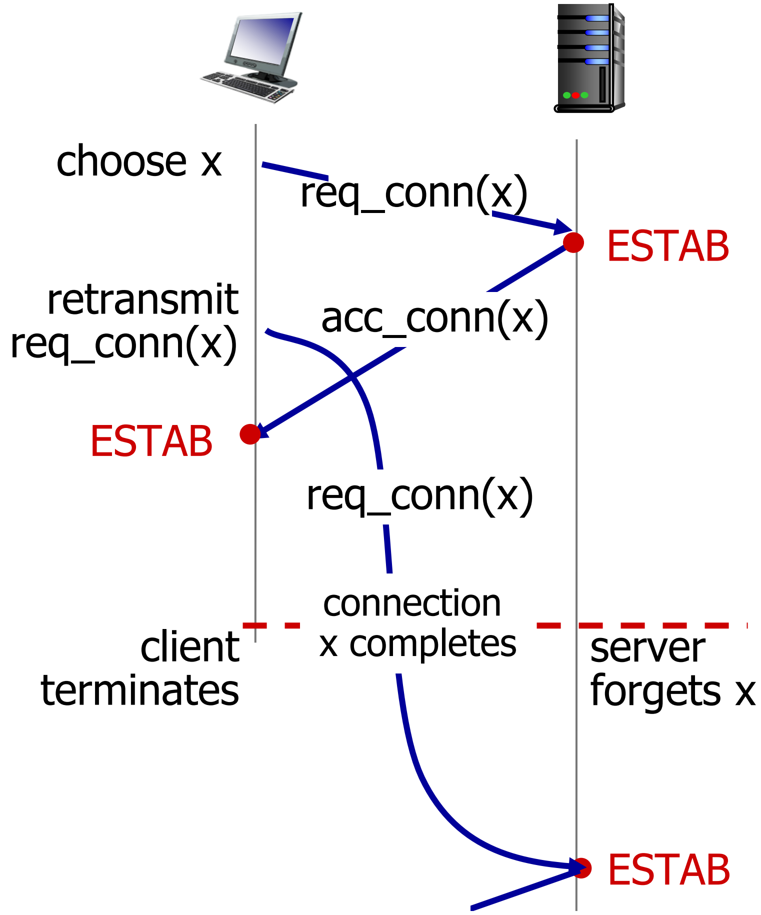
x completes

2-way handshake scenarios



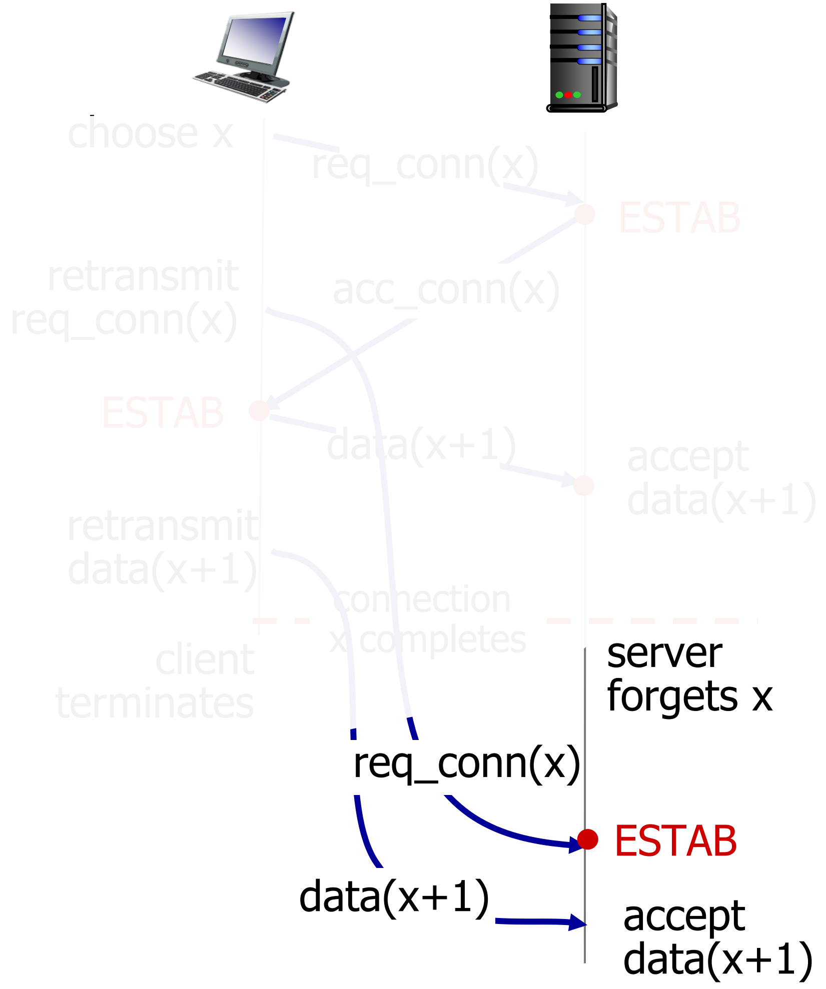
No problem!

2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!