



UNIVERSITY OF  
**WATERLOO**

# **CS 456/656**

# **Computer Networks**

## **Lecture 7: Transport Layer – Part 3**

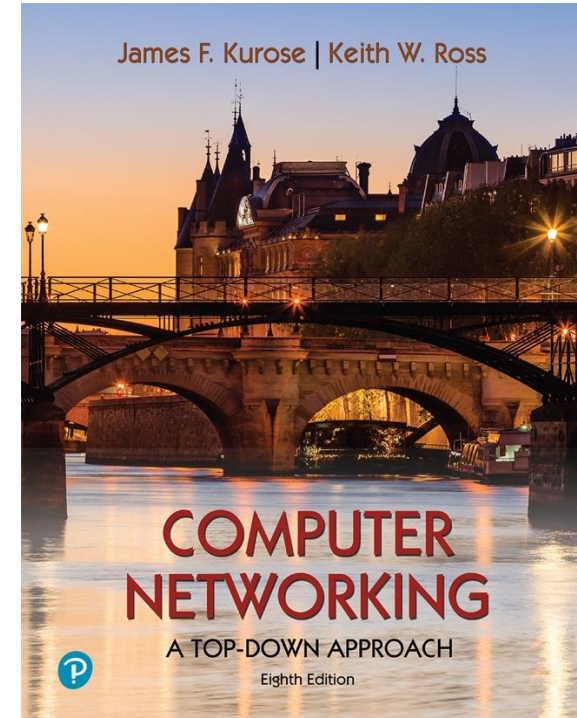
Mina Tahmasbi Arashloo and Bo Sun

Fall 2024

# A note on the slides

Adapted from the slides that  
accompany this book.

All material copyright 1996-2023  
J.F Kurose and K.W. Ross, All Rights Reserved



## *Computer Networking: A Top-Down Approach*

8<sup>th</sup> edition  
Jim Kurose, Keith Ross  
Pearson, 2020

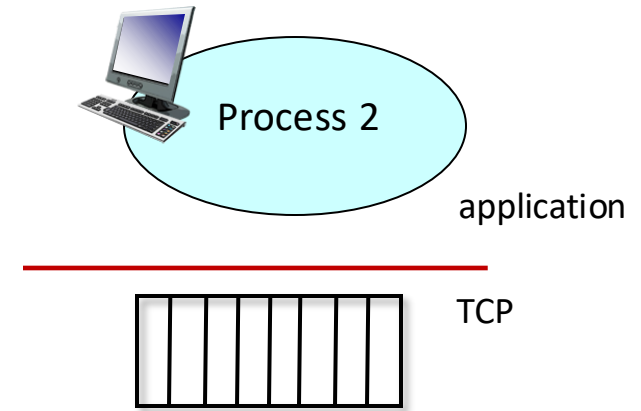
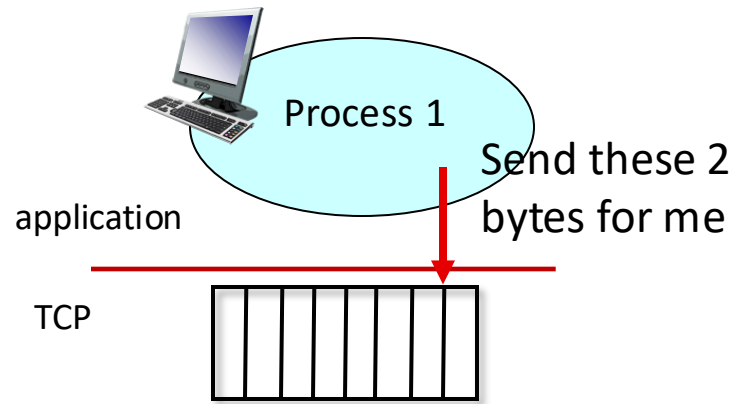
# Transport layer: roadmap

- 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



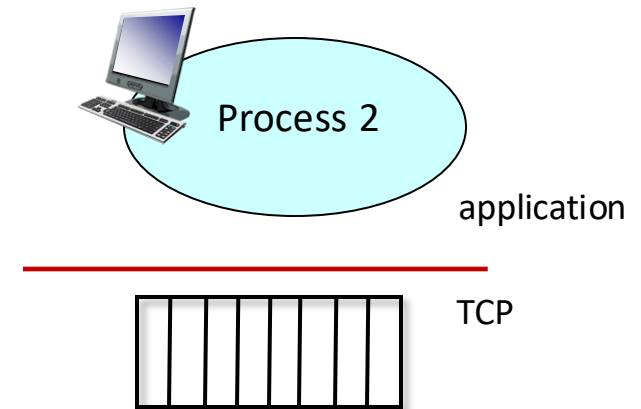
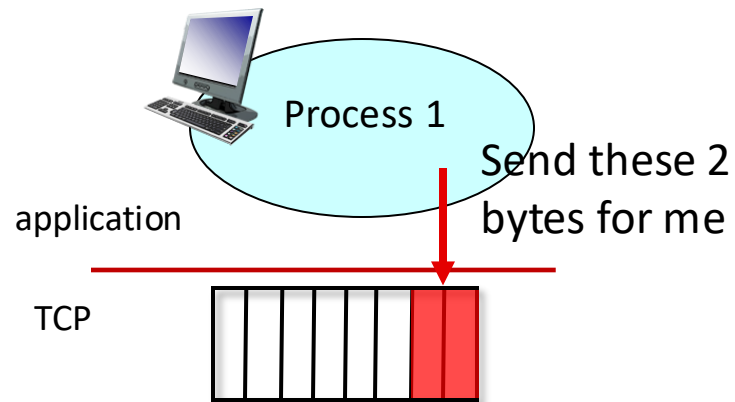
# TCP: a widely-used reliable transport protocol

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



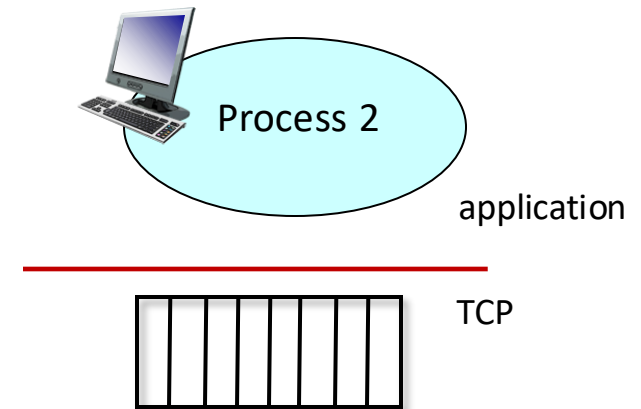
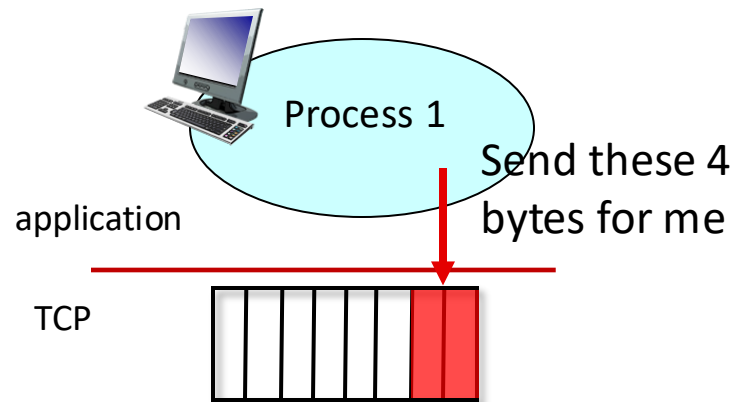
# TCP: a widely-used reliable transport protocol

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



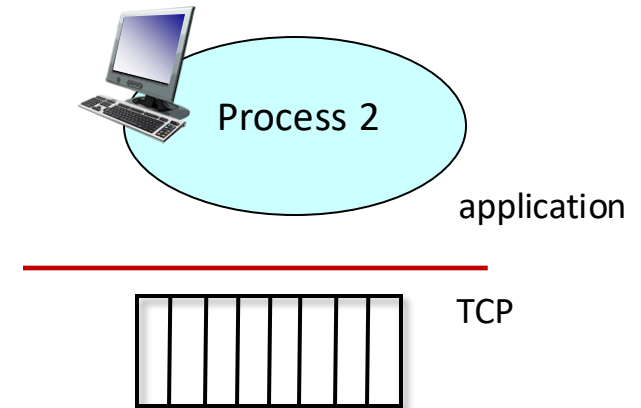
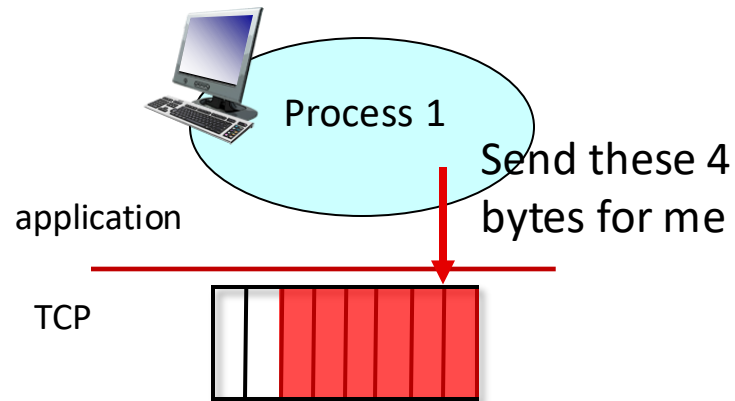
# TCP: a widely-used reliable transport protocol

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



# TCP: a widely-used reliable transport protocol

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



# TCP: a widely-used reliable transport protocol

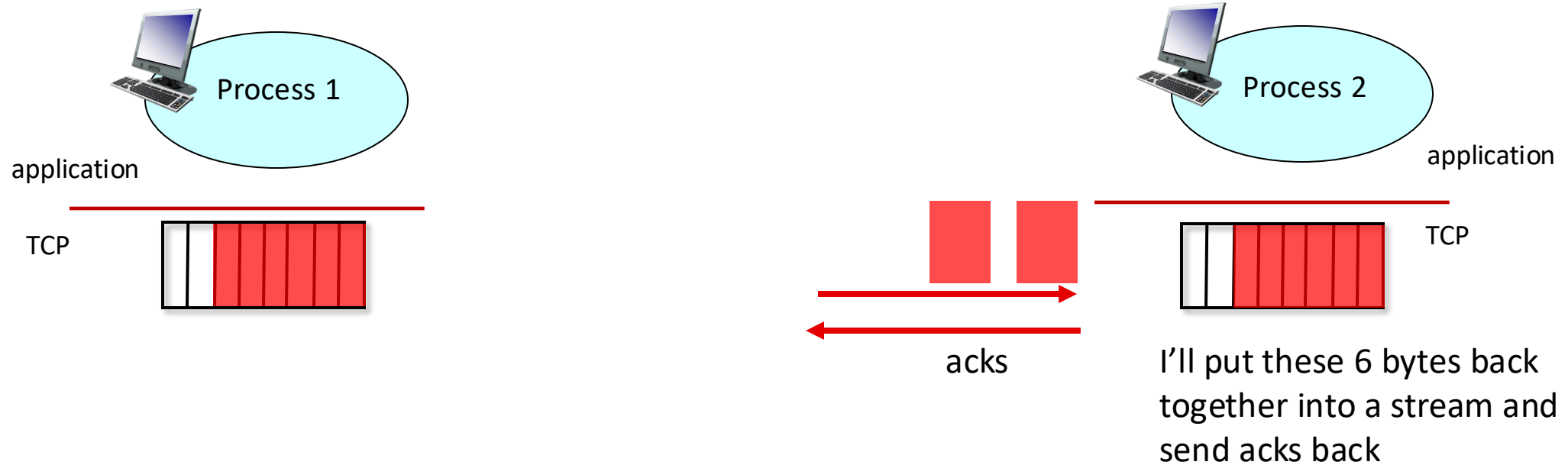
- Guarantees **reliable, in-order *byte stream***:
  - no “message boundaries”





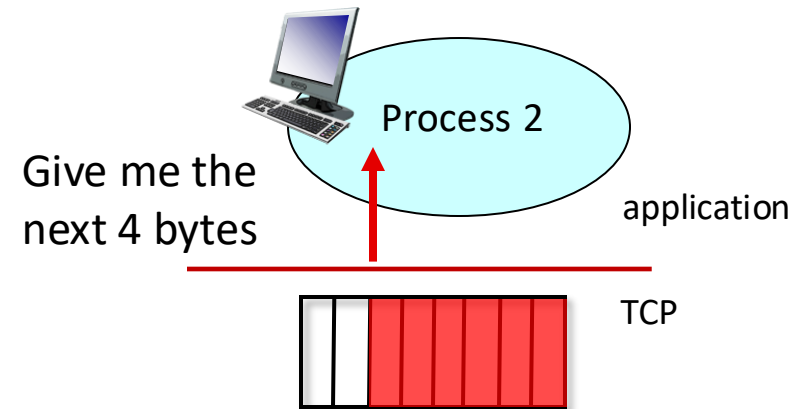
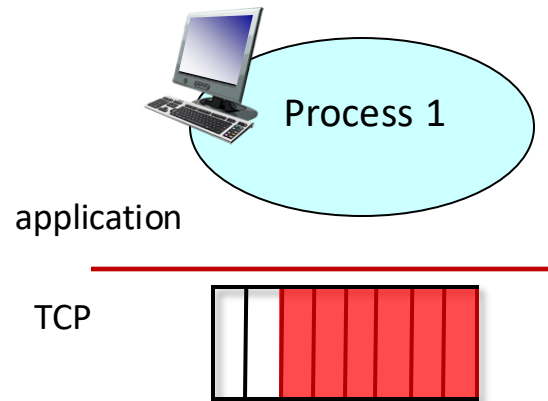
# TCP: a widely-used reliable transport protocol

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



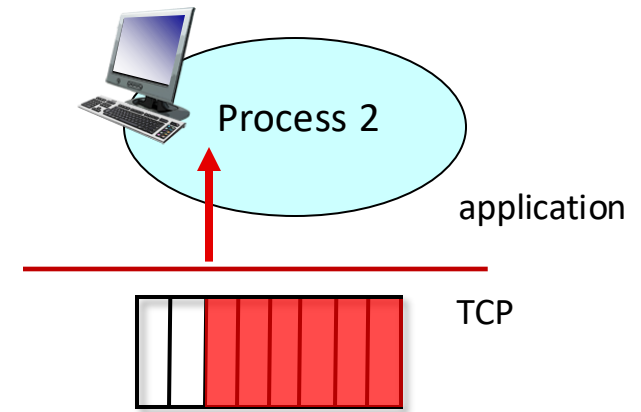
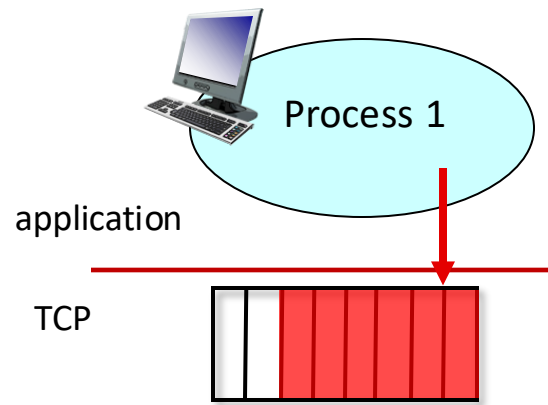
# TCP: a widely-used reliable transport protocol

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



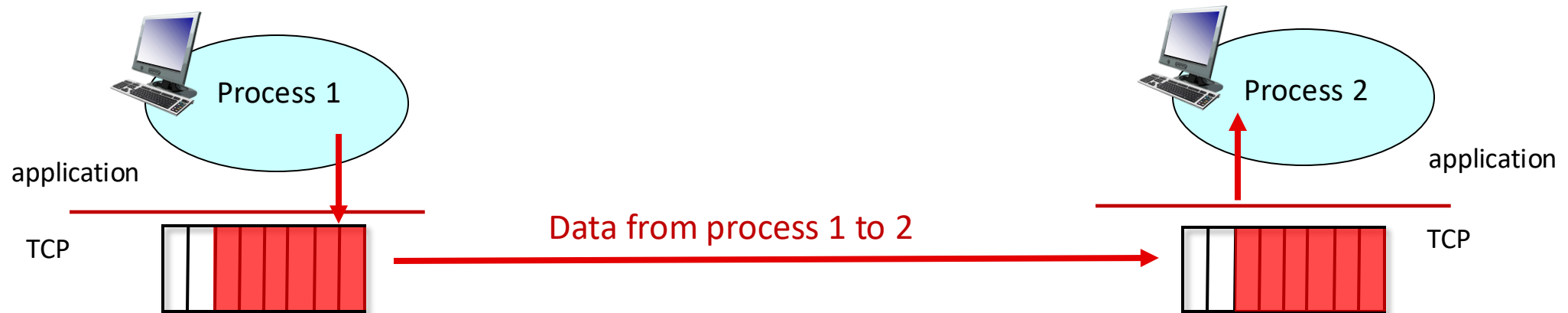
# TCP: a widely-used reliable transport protocol

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



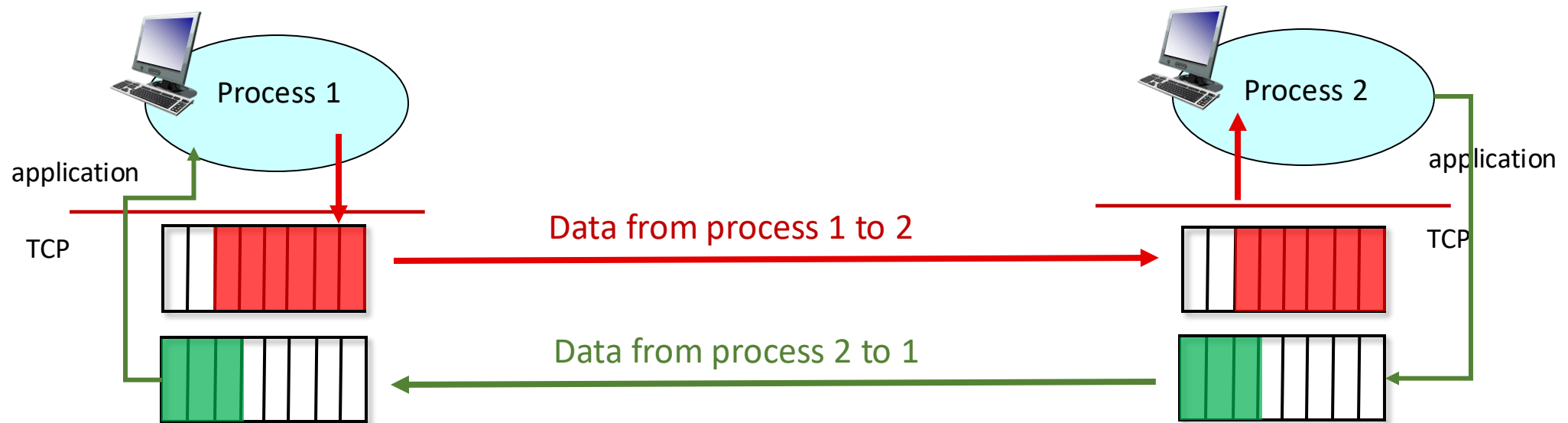
# TCP: a widely-used reliable transport protocol

- Guarantees **reliable, in-order *byte stream***:
  - 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 stream***:
  - 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 stream***:
  - 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

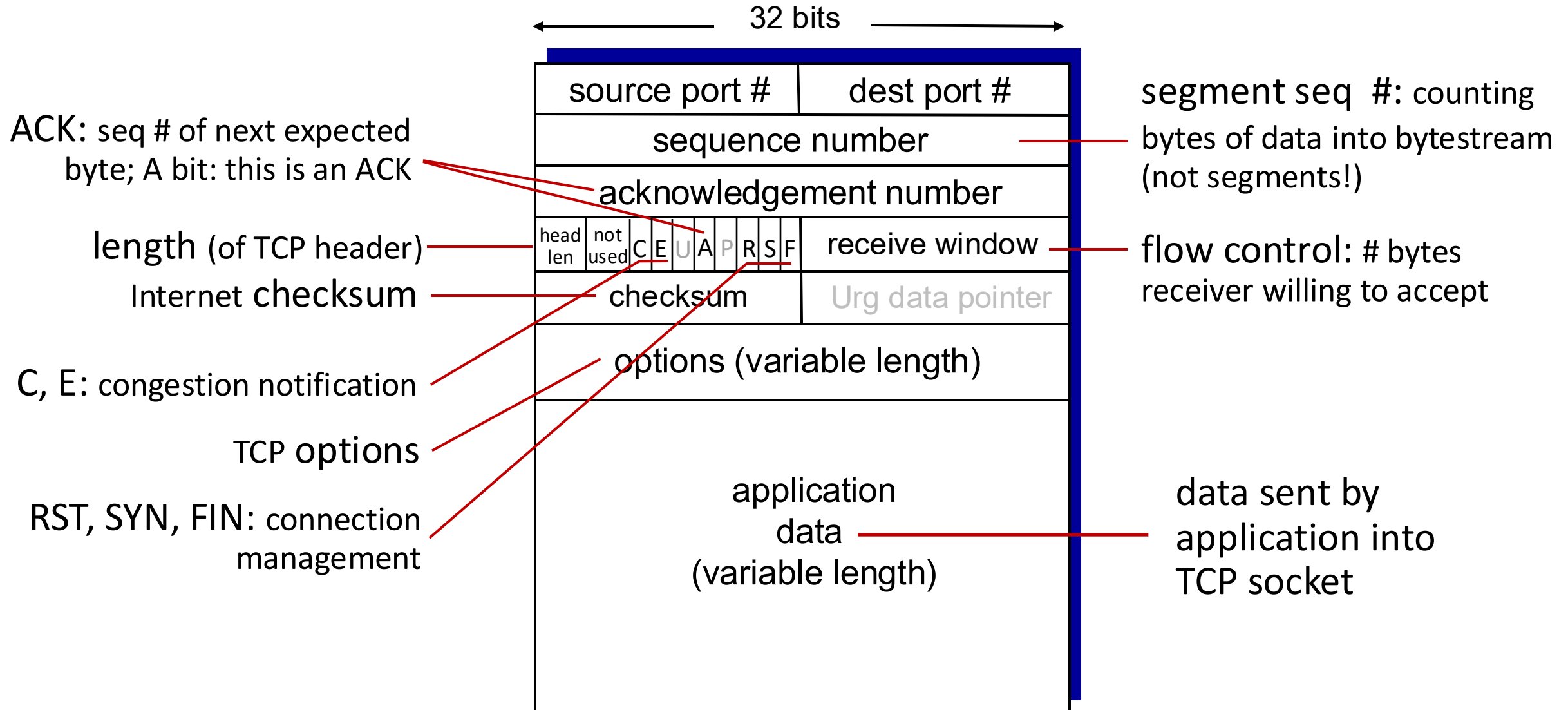
# Transport layer: roadmap

- 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**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control



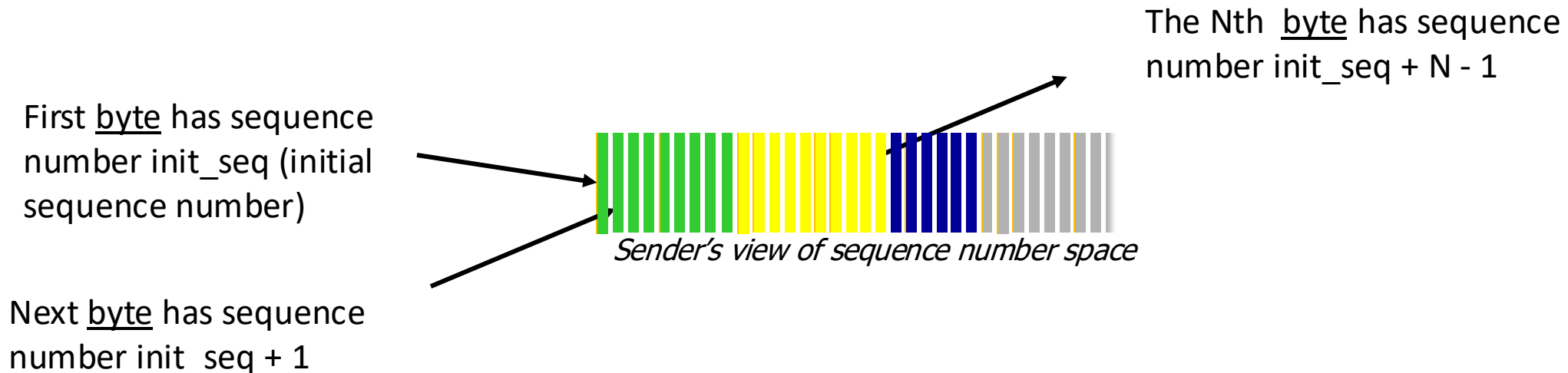
# TCP reliable data transfer

*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

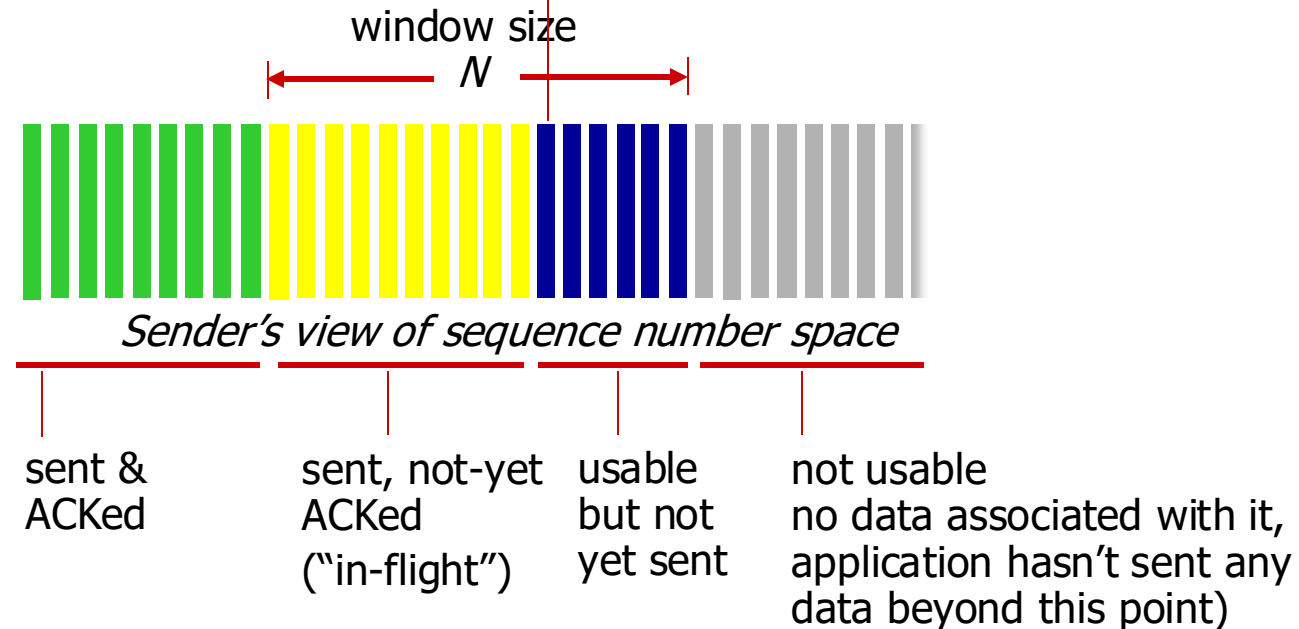
- The interface between a sending process and TCP is a byte stream.
- TCP assigns a sequence number to every byte
  - As opposed to every segment, as we discussed in the last lecture
- It keeps track of the “status” of every byte
  - Is it sent yet? Is it acknowledged yet?



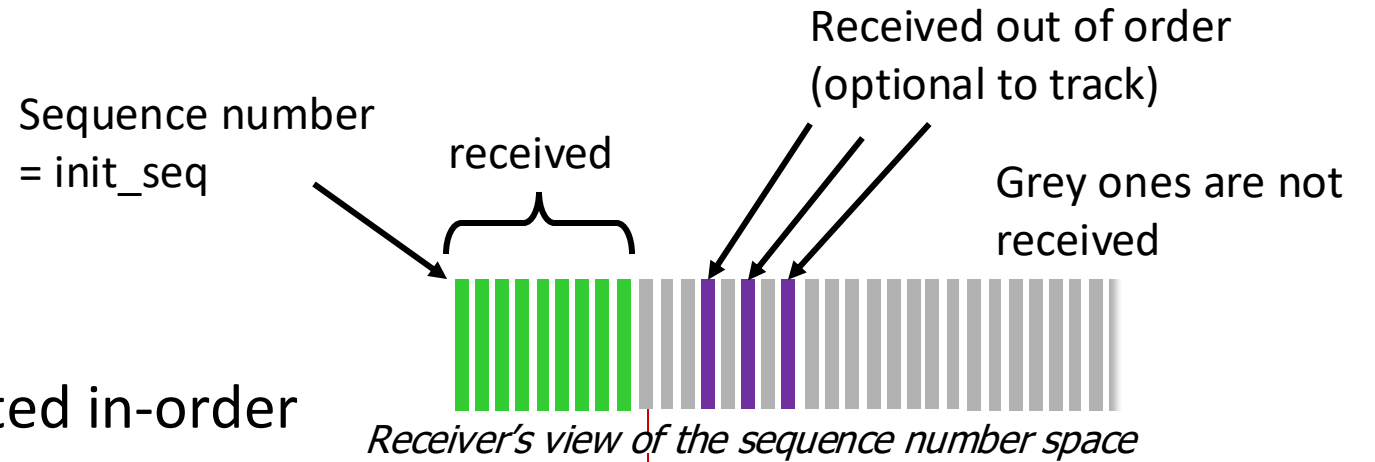
# TCP sequence numbers

outgoing segment from sender

source port #		dest port #	
sequence number			
acknowledgement number			
		rwnd	
checksum		urg pointer	



# TCP ACKs



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

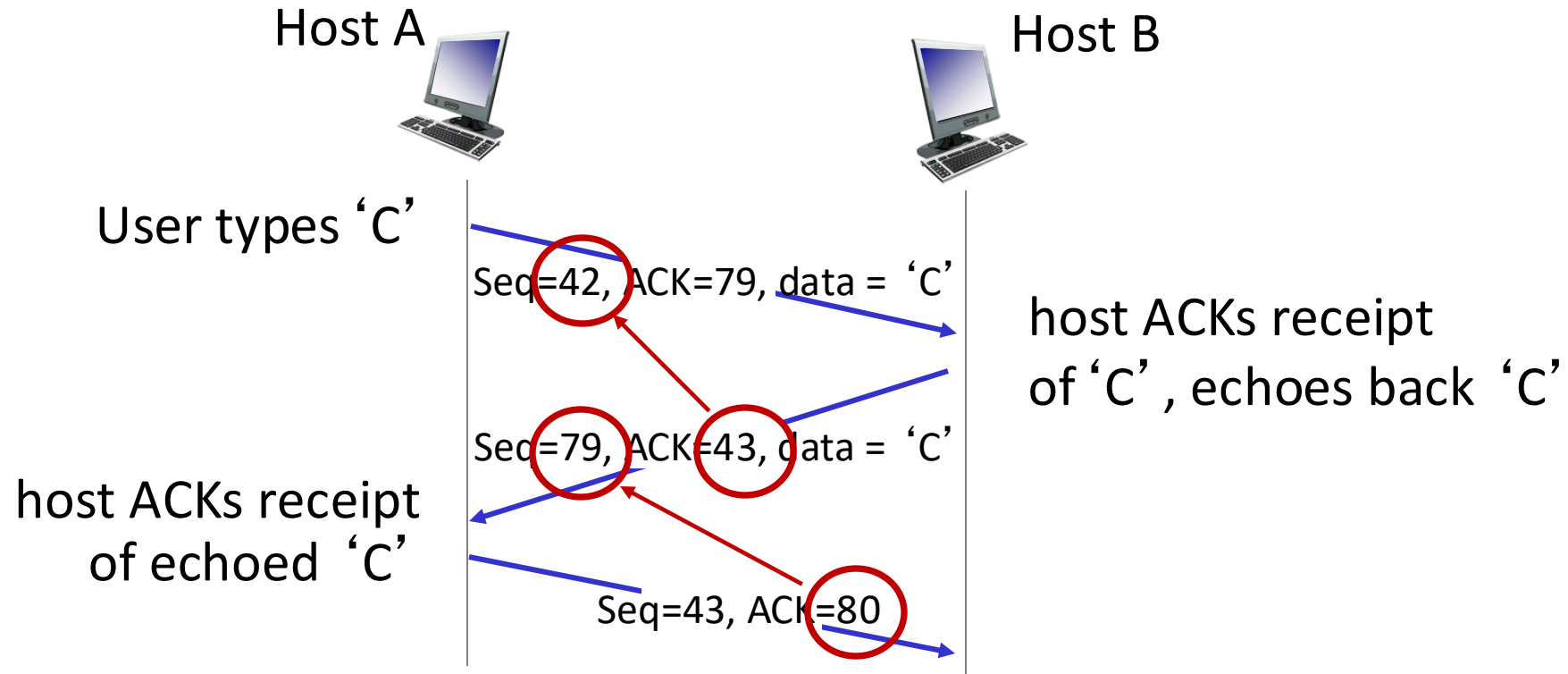
Q: What about out-of-order segments?

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

outgoing segment from receiver

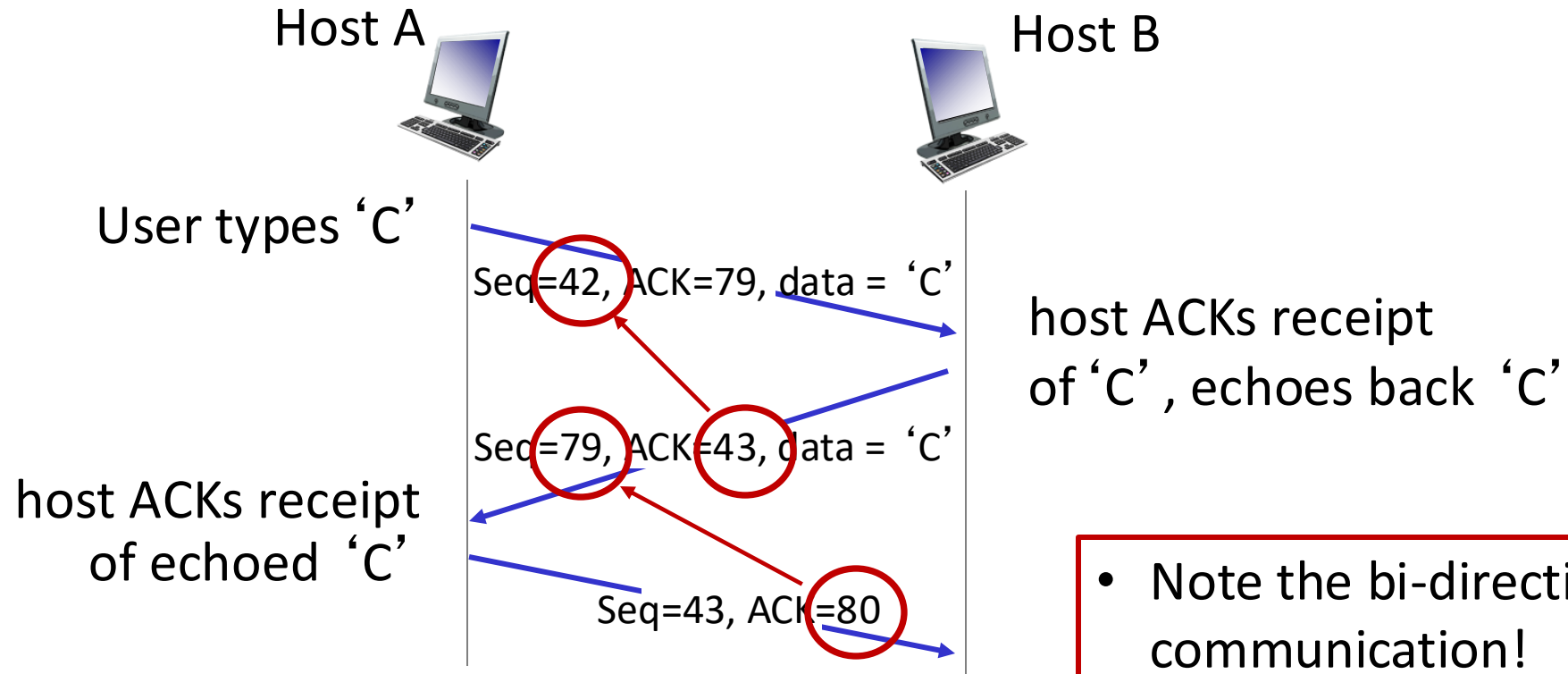
source port #	dest port #
sequence number	
acknowledgement number	
	A rwnd
checksum	urg pointer

# 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*

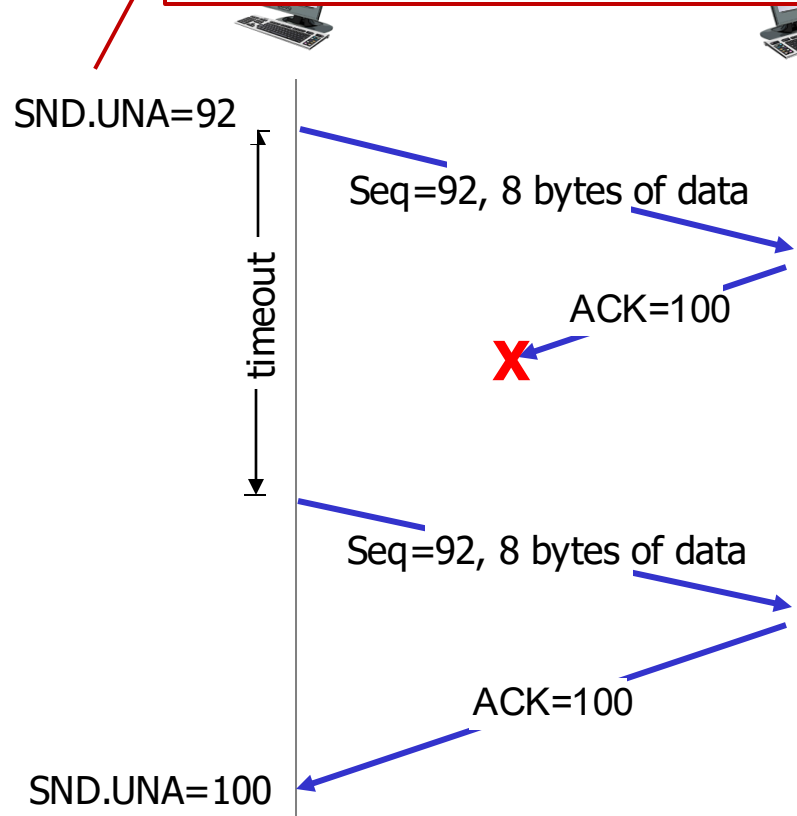
- retransmit segment that caused timeout
- restart timer

event: *ACK received*

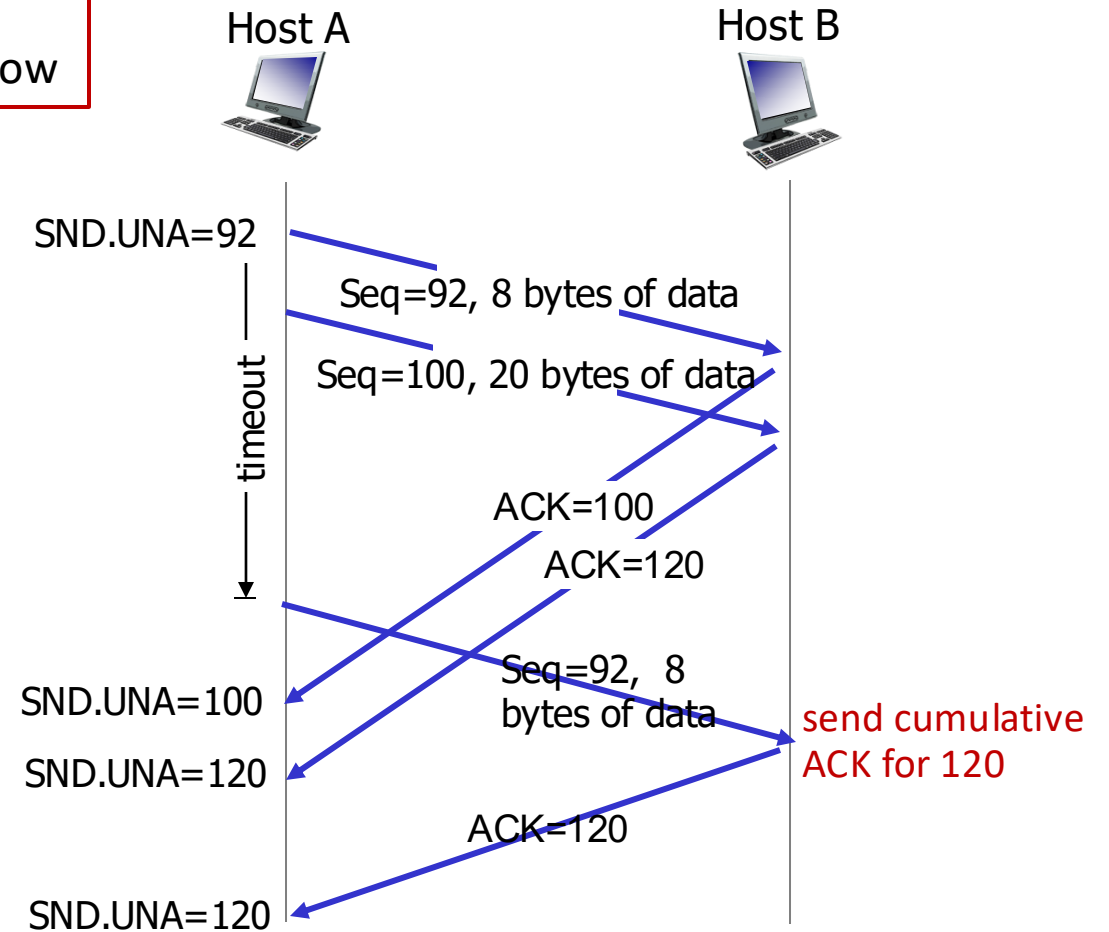
- if ACK acknowledges previously unACKed segments
  - update what is known to be ACKed
  - restart timer if there are still unACKed segments

# TCP: retransmission scenarios

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

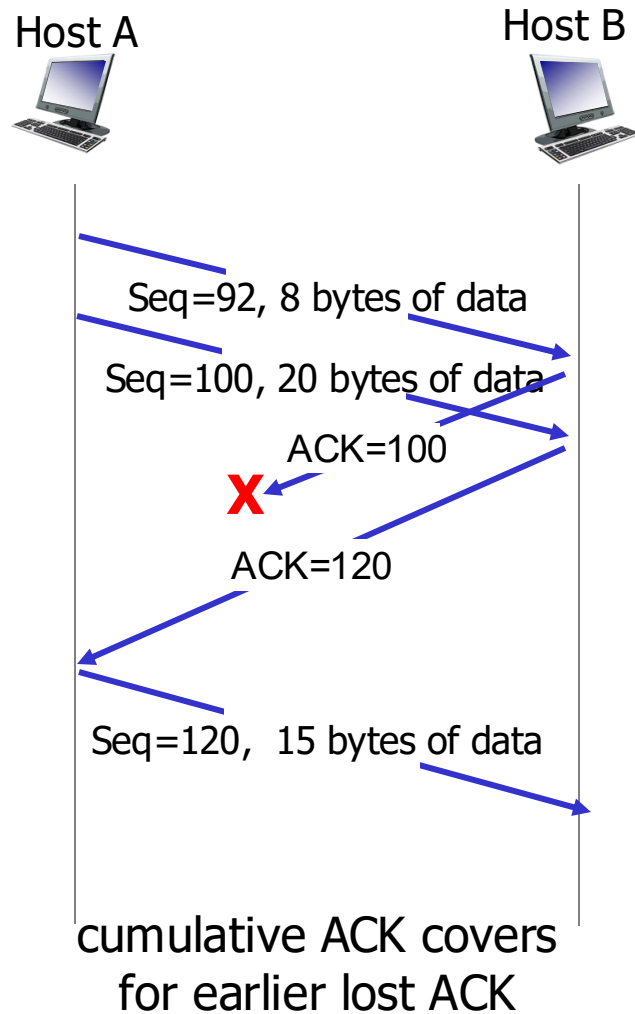


lost ACK scenario

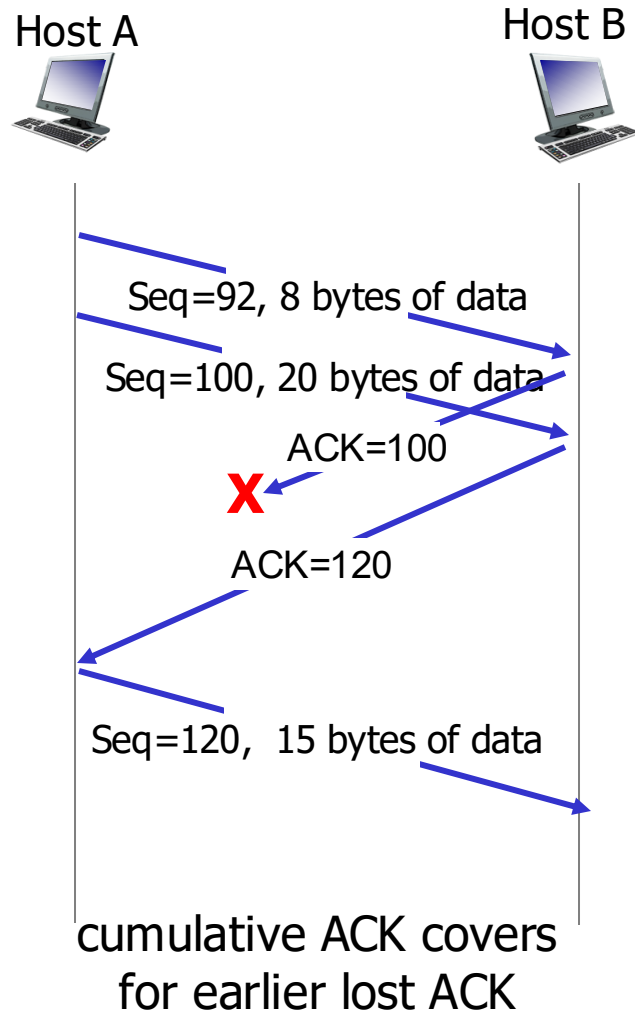


premature timeout

# TCP: retransmission scenarios

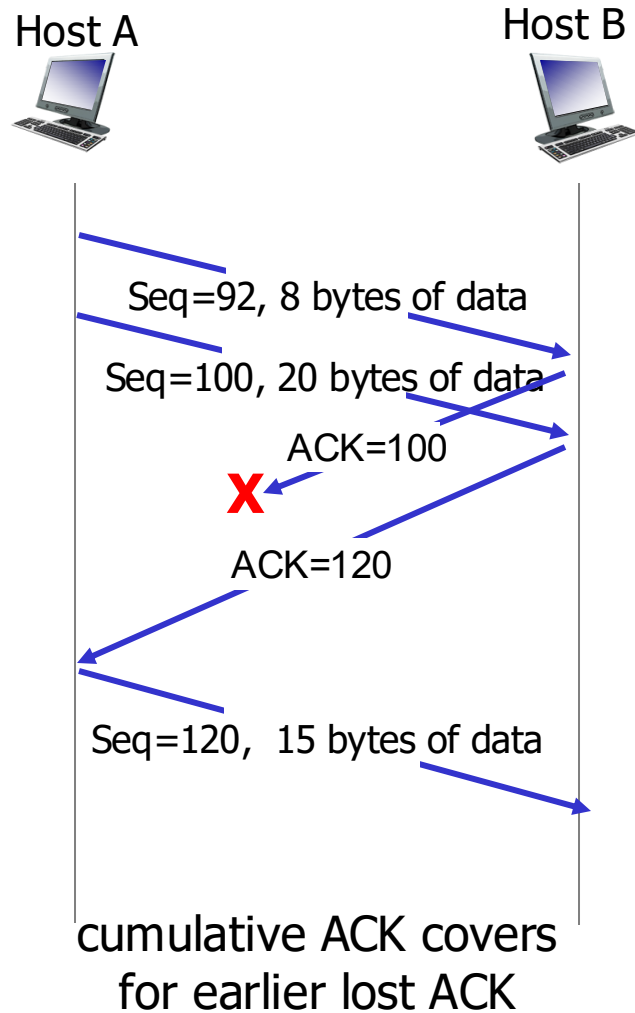


# TCP: retransmission scenarios



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

# TCP: retransmission scenarios



- 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

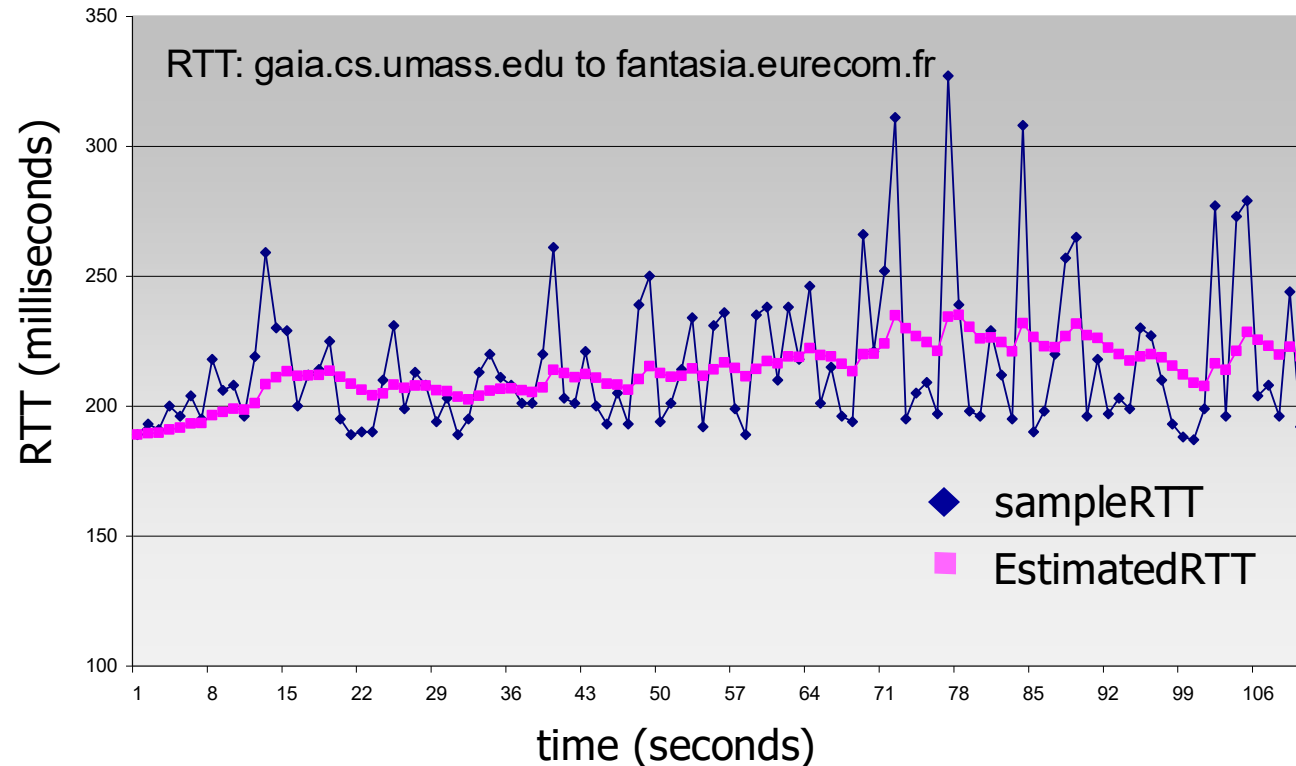
Q: how to estimate RTT?

- `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

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- exponential weighted moving average (EWMA)
- 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**: want a larger safety margin

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



↑  
estimated RTT

↑  
“safety margin”

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

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

(typically,  $\beta = 0.25$ )

# 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

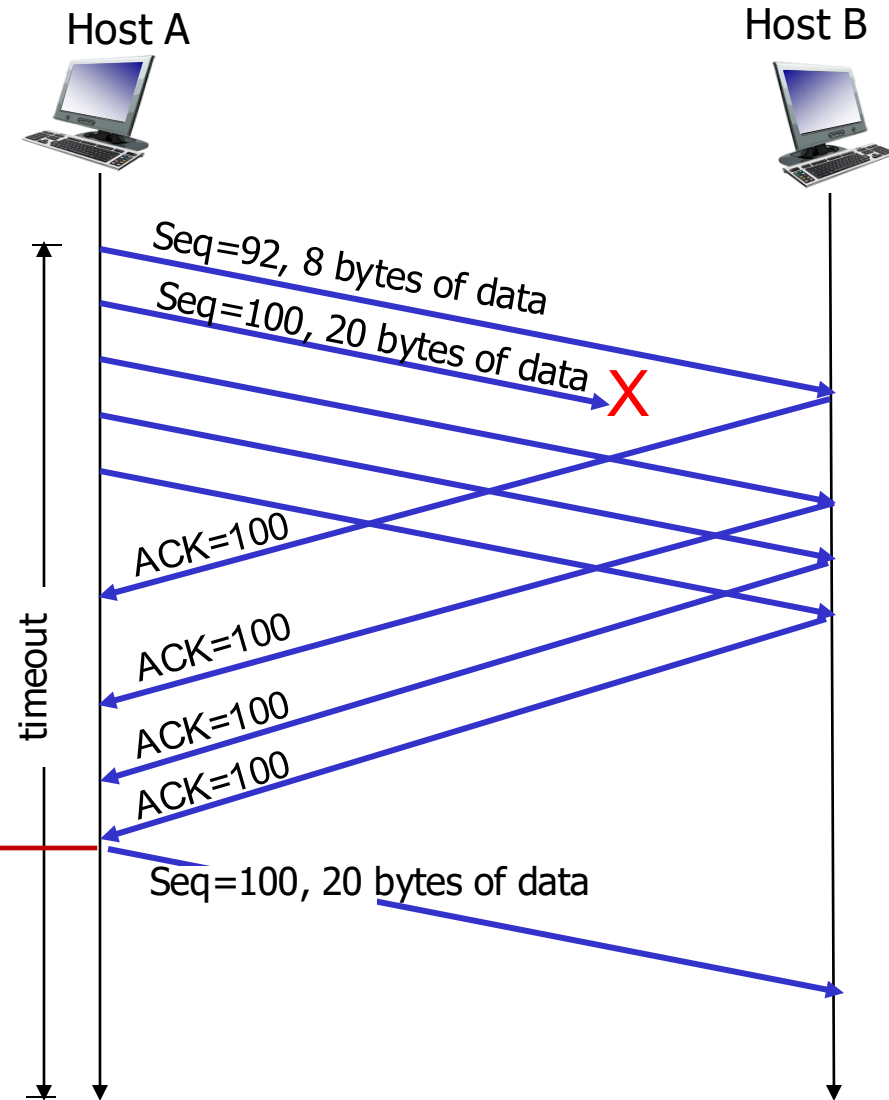
## *TCP fast retransmit*

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



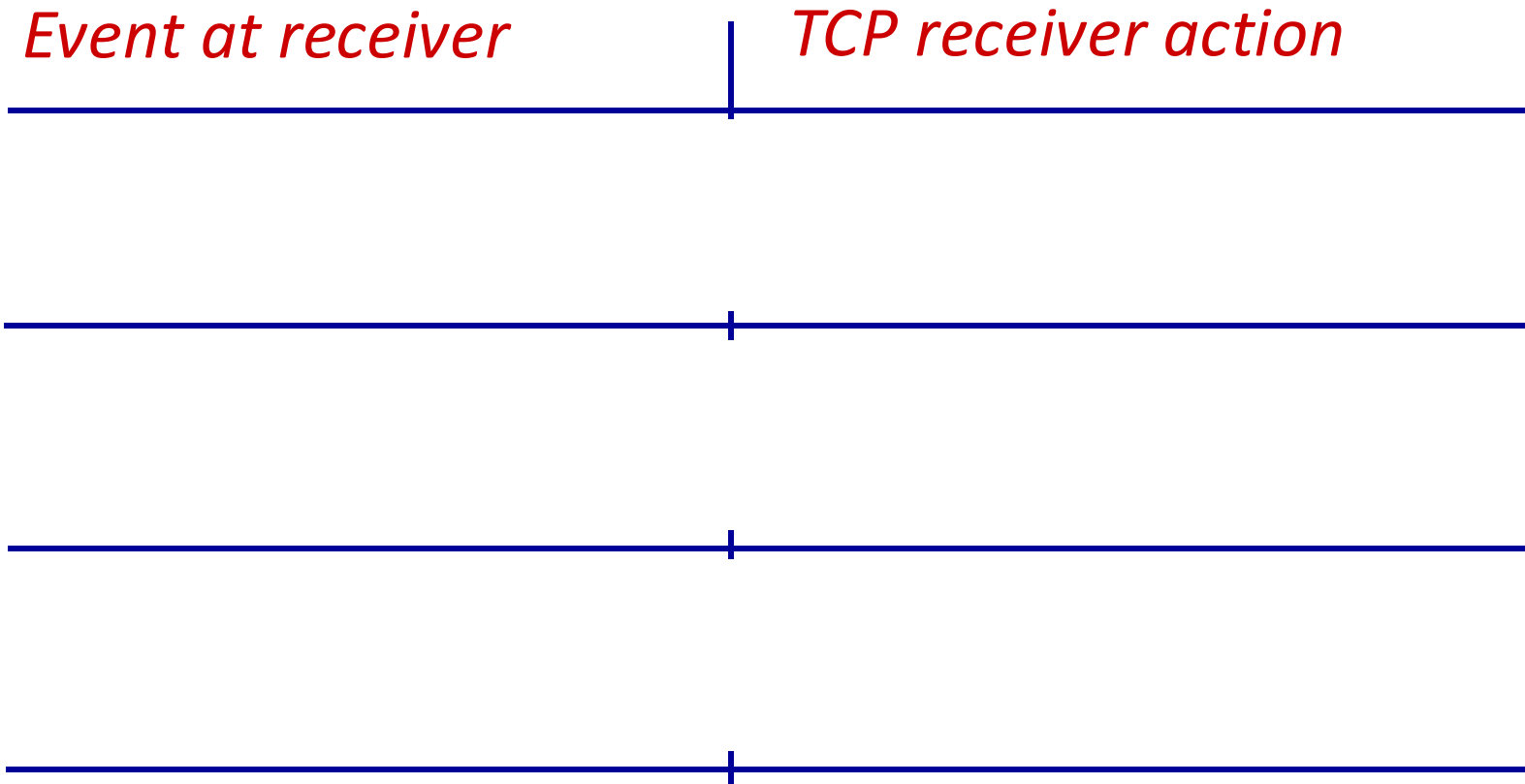
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!



# 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.

# TCP Receiver: ACK generation [RFC 5681]



# TCP Receiver: ACK generation [RFC 5681]

<i>Event at receiver</i>	<i>TCP receiver action</i>
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 <i>duplicate ACK</i> , 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

# Optimizations 2: Delays ACKs (cont.)

<i>Event at receiver</i>	<i>TCP receiver action</i>
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 <i>duplicate ACK</i> , indicating seq. # of next expected byte

# Optimizations 2: Delays ACKs (cont.)

<i>Event at receiver</i>	<i>TCP receiver action</i>
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 <i>duplicate ACK</i> , 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

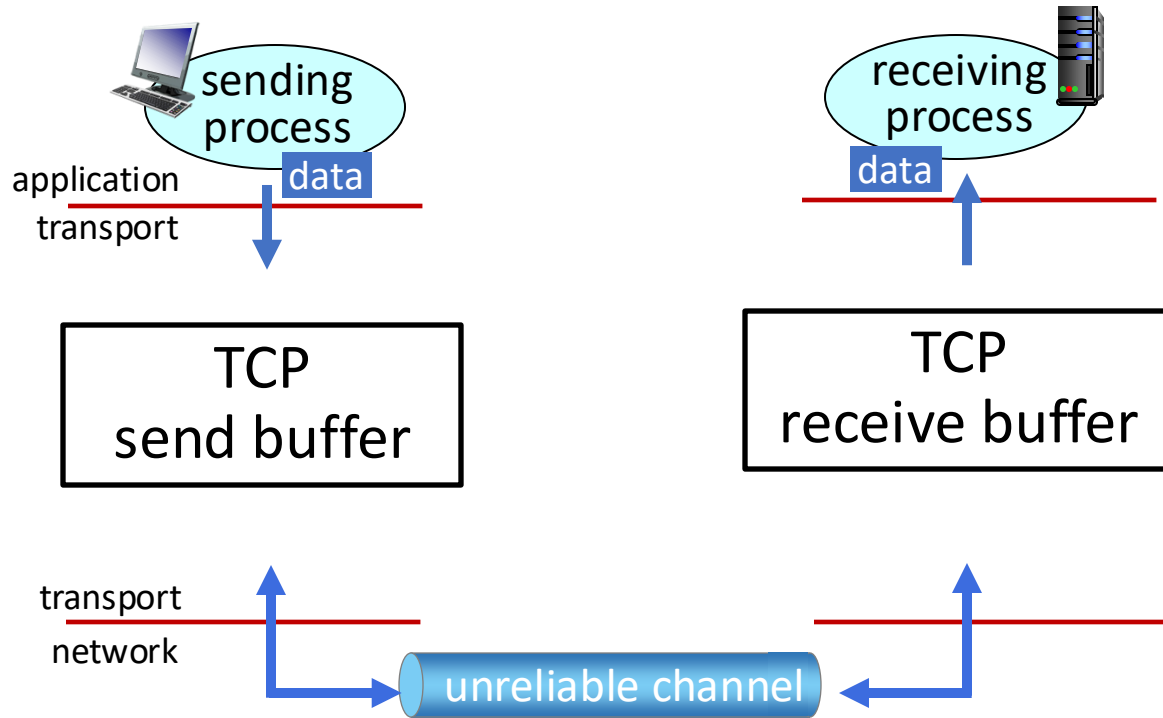


# Transport layer: roadmap

- 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

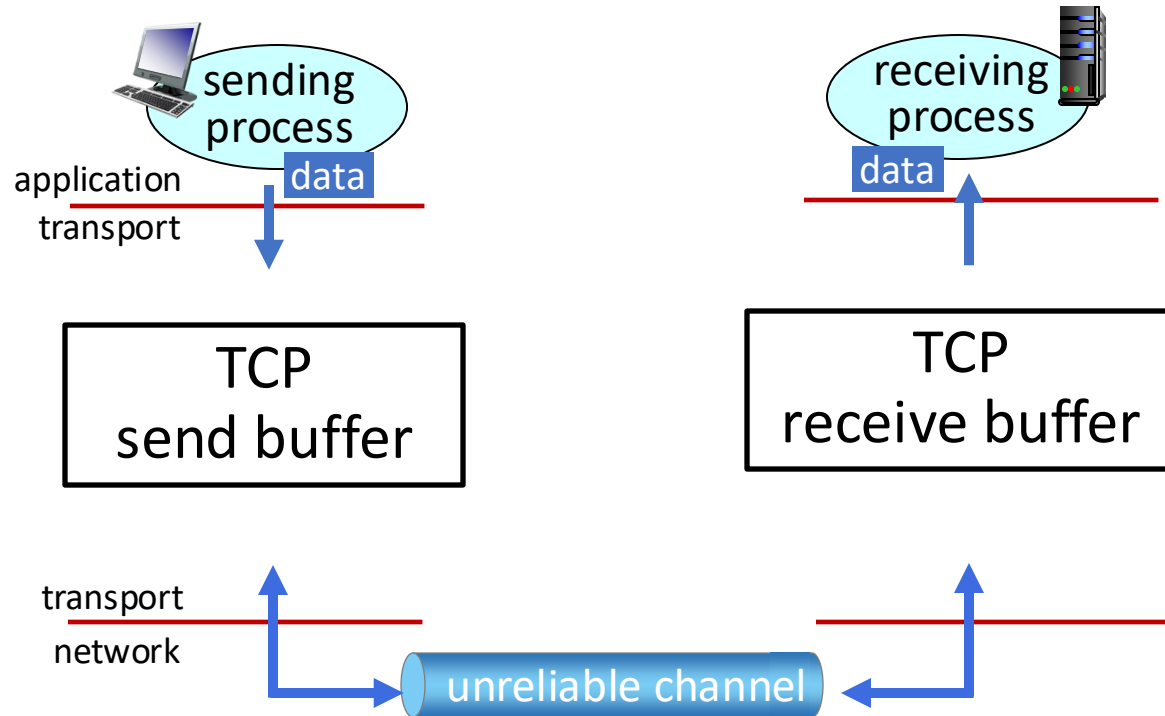


# TCP flow control

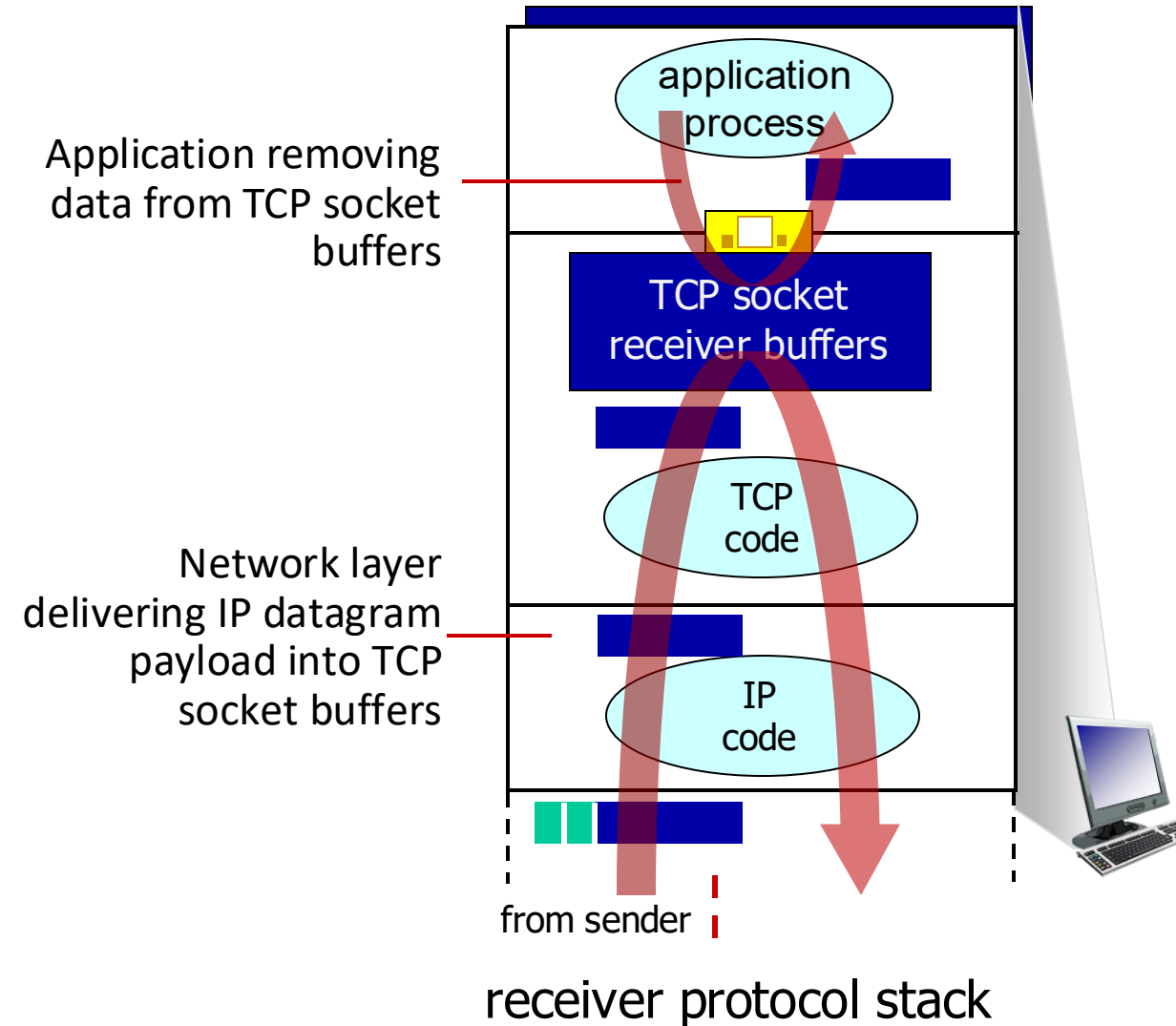


- The send buffer holds the data the application sends to TCP until it is delivered
- The receive buffer holds the data TCP receives from the network until it is delivered to the application

# TCP flow control

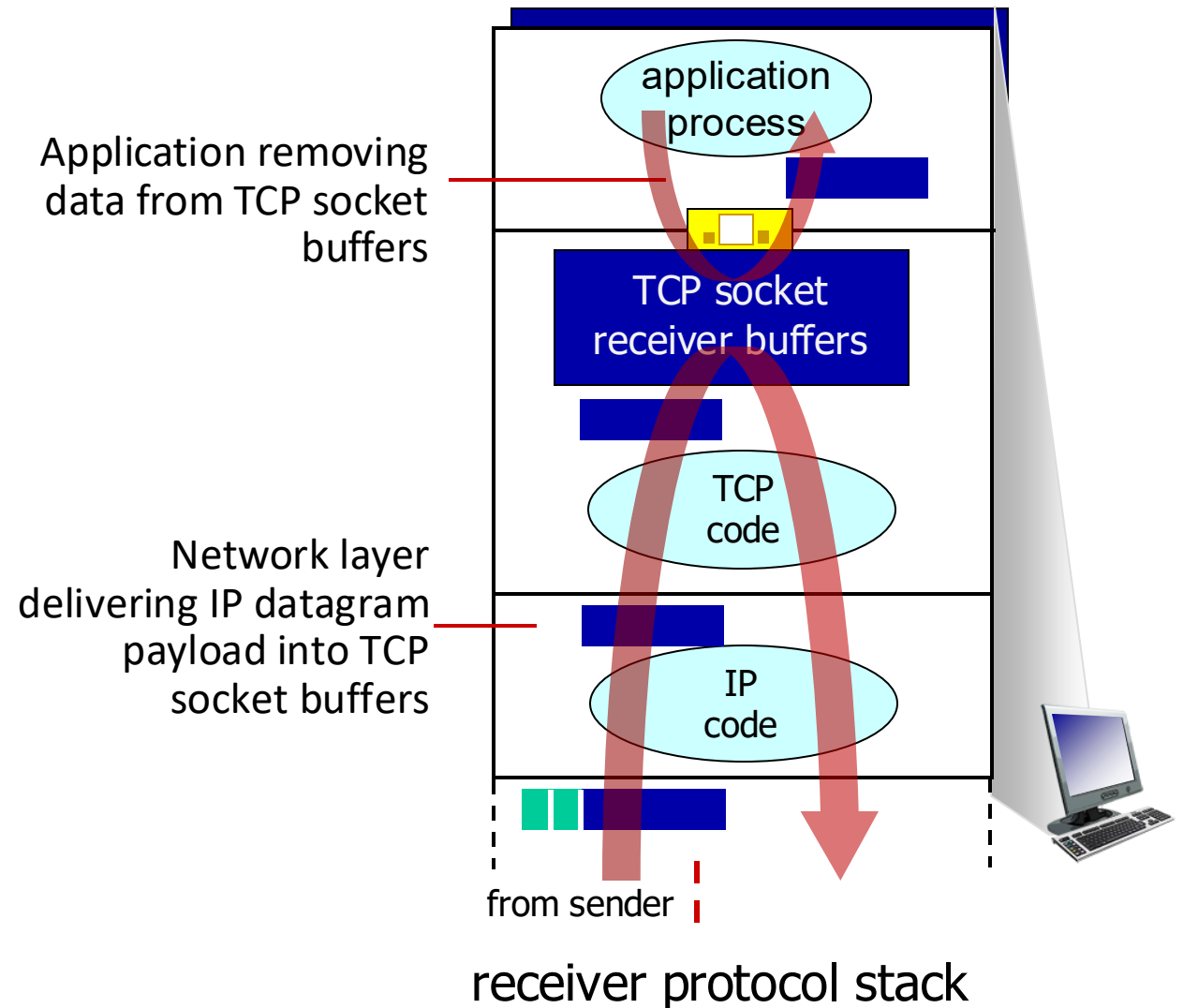


Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



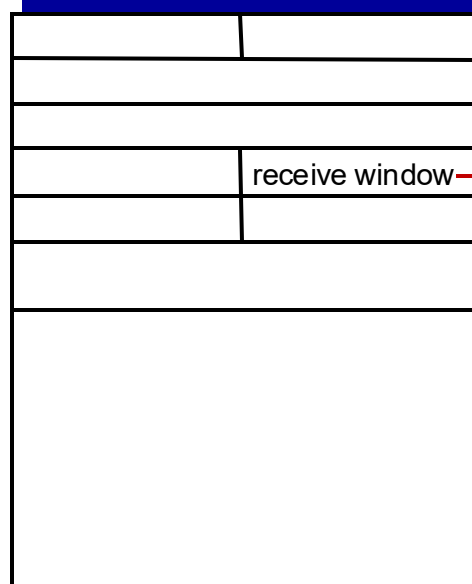
# TCP flow control

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



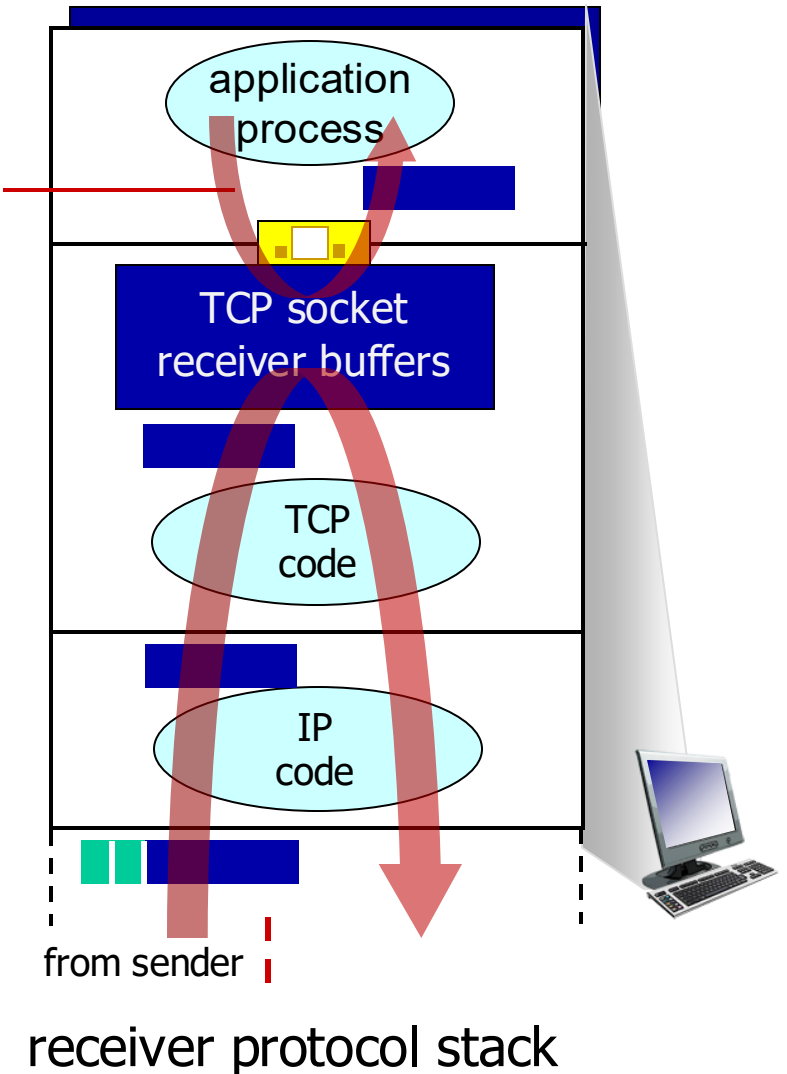
# TCP flow control

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



flow control: # bytes  
receiver willing to accept

Application removing  
data from TCP socket  
buffers

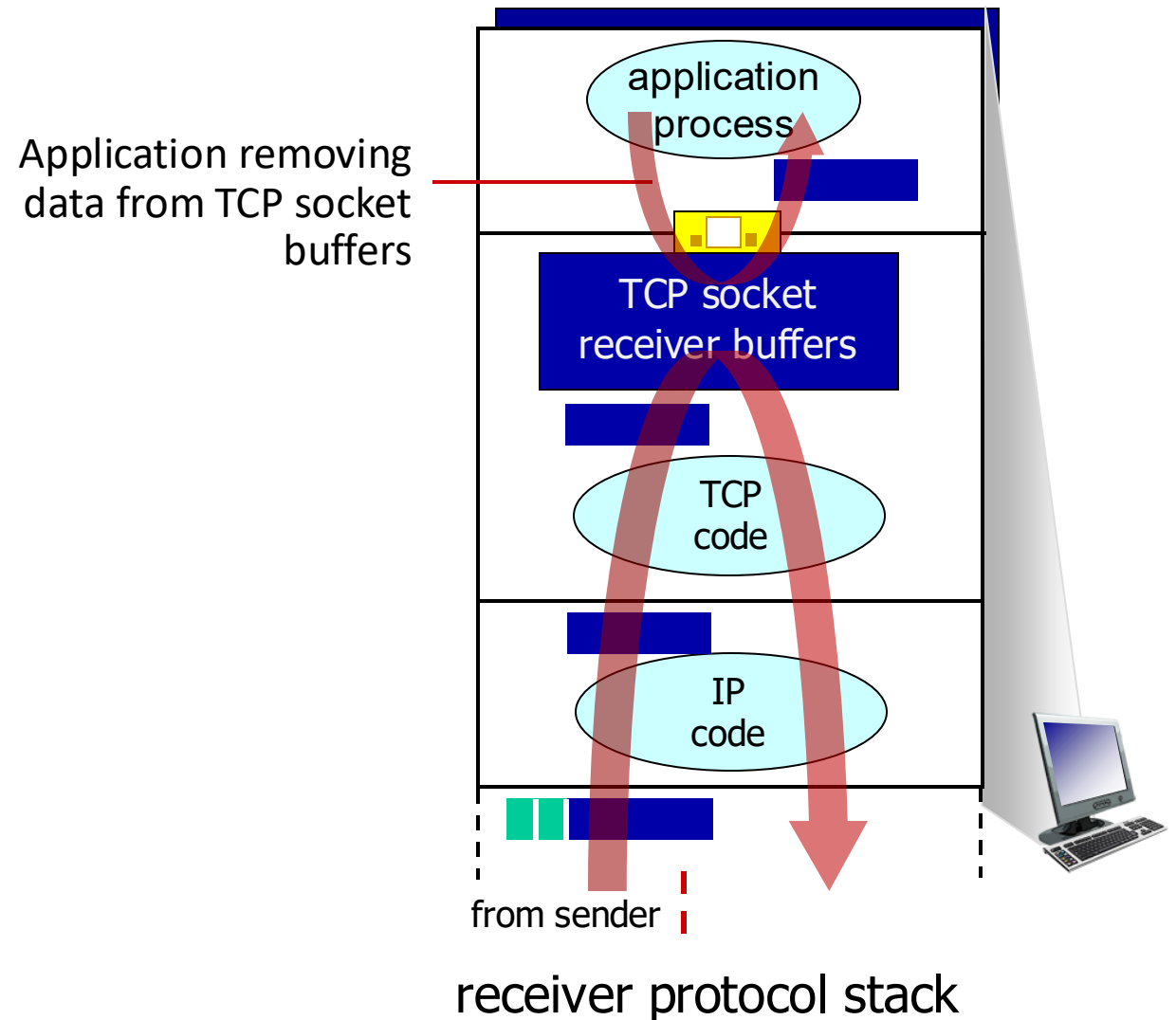


# TCP flow control

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

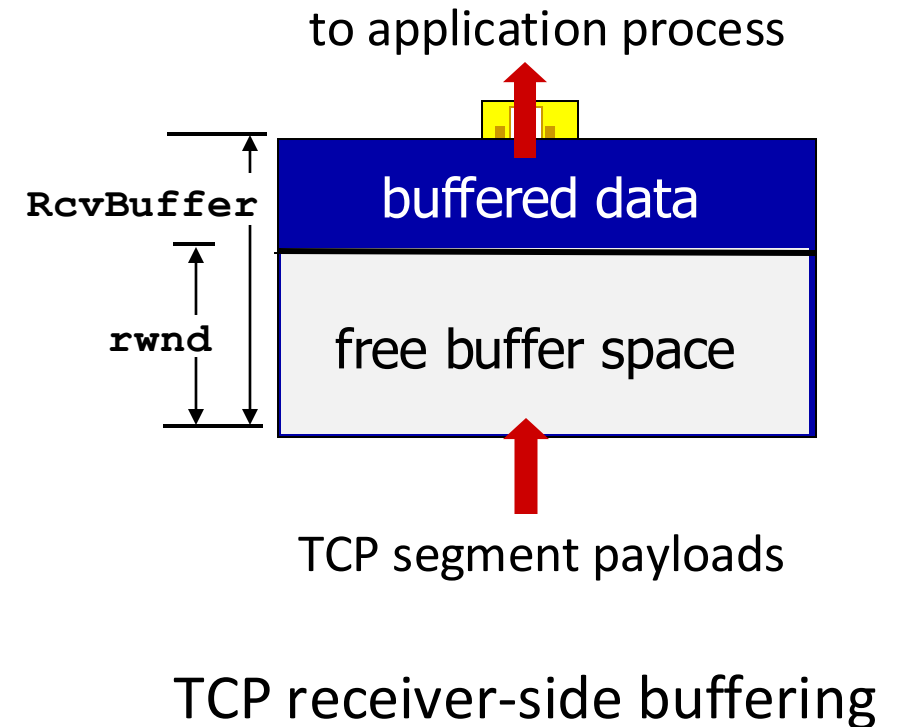
## —flow control—

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



# TCP flow control

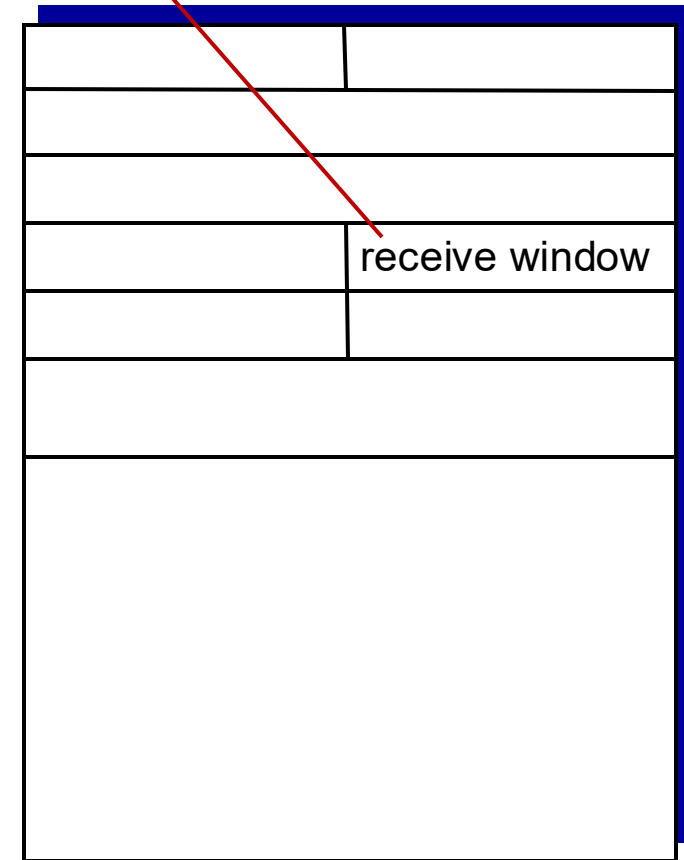
- **RcvBuffer** size set via socket options
  - many operating systems auto-adjust **RcvBuffer**
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive buffer will not overflow



# 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

flow control: # bytes receiver willing to accept



TCP segment format



# Transport layer: roadmap

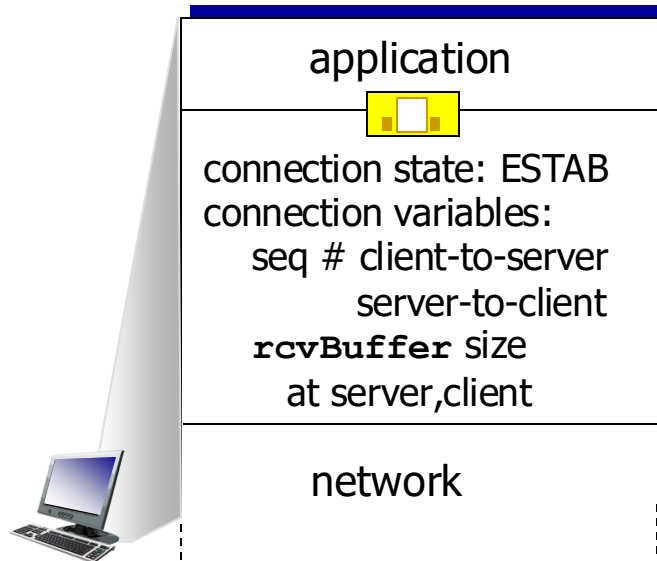
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- **Connection-oriented transport: TCP**
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
-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



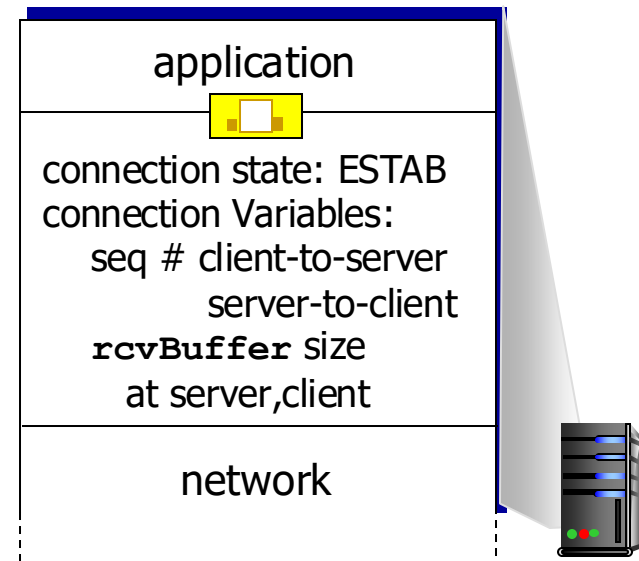
# TCP connection management

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)



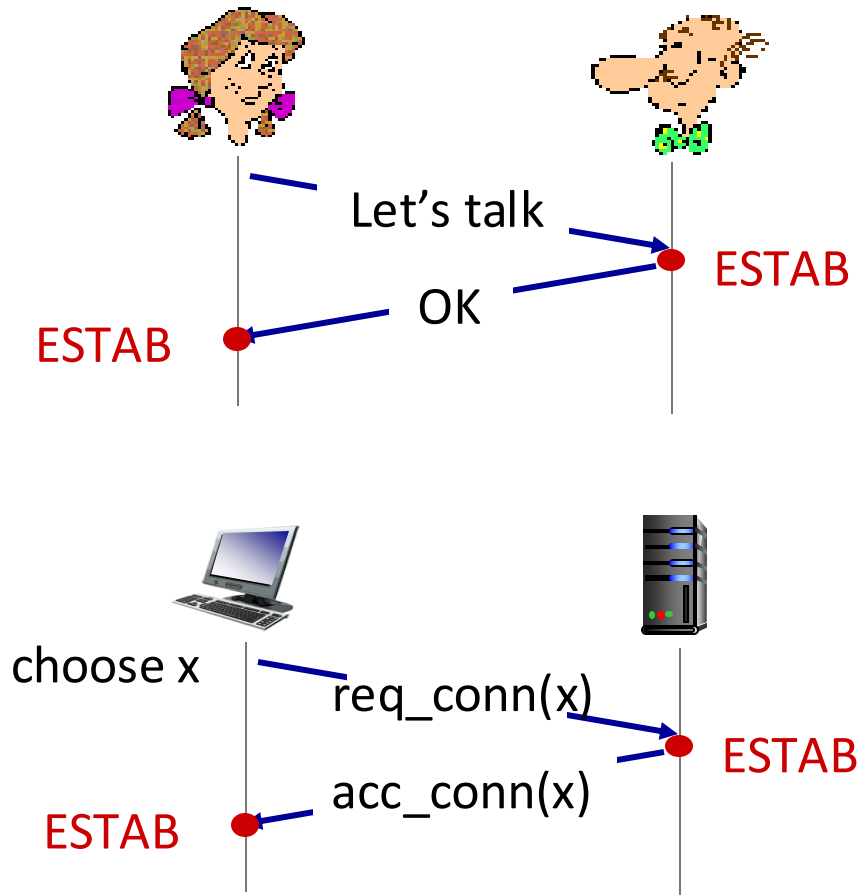
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



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

# Agreeing to establish a connection

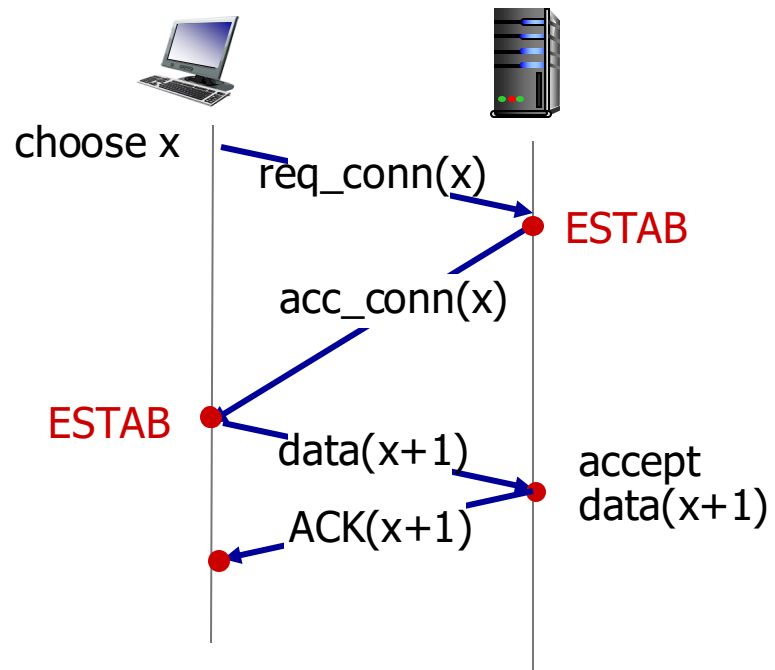
2-way handshake:



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

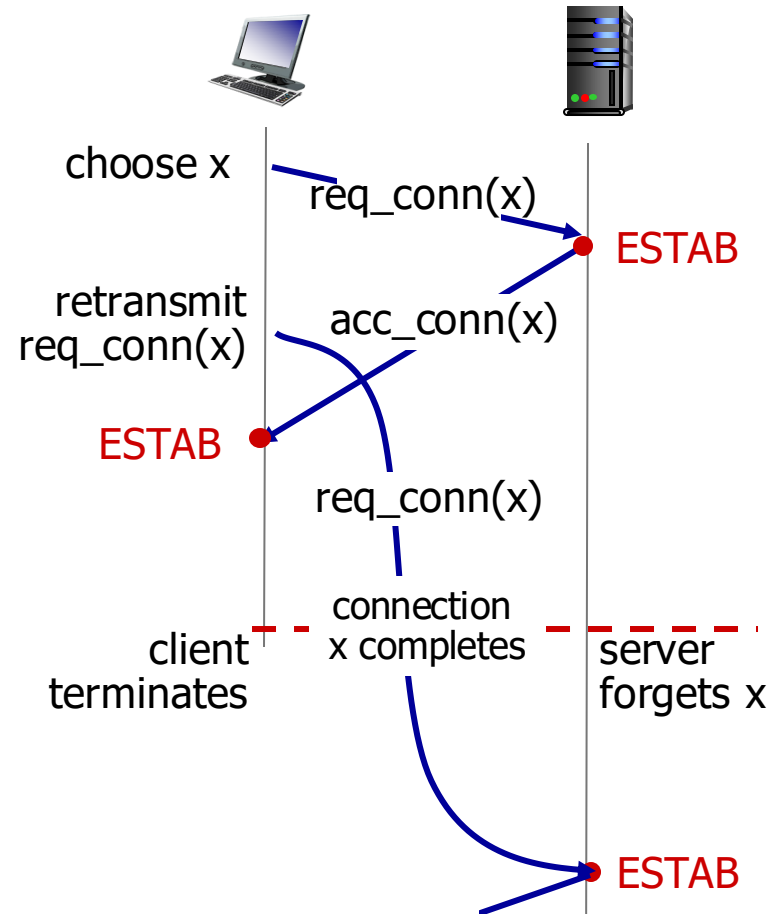
# 2-way handshake scenarios



No problem!

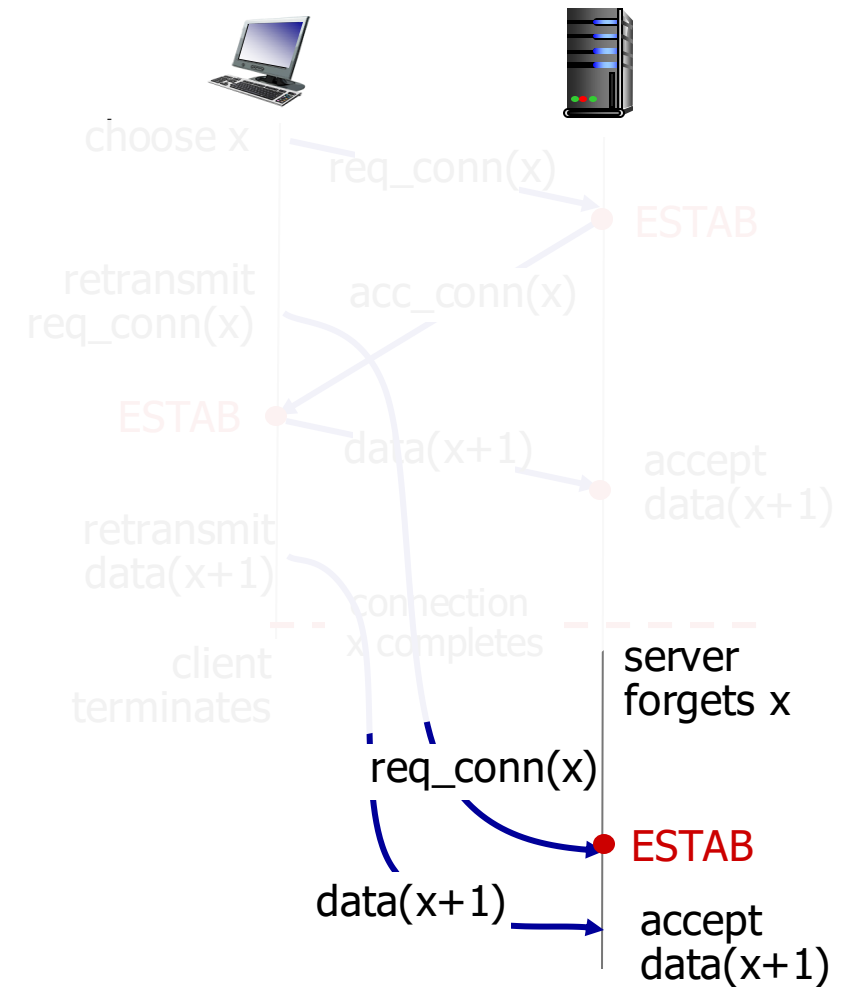



# 2-way handshake scenarios



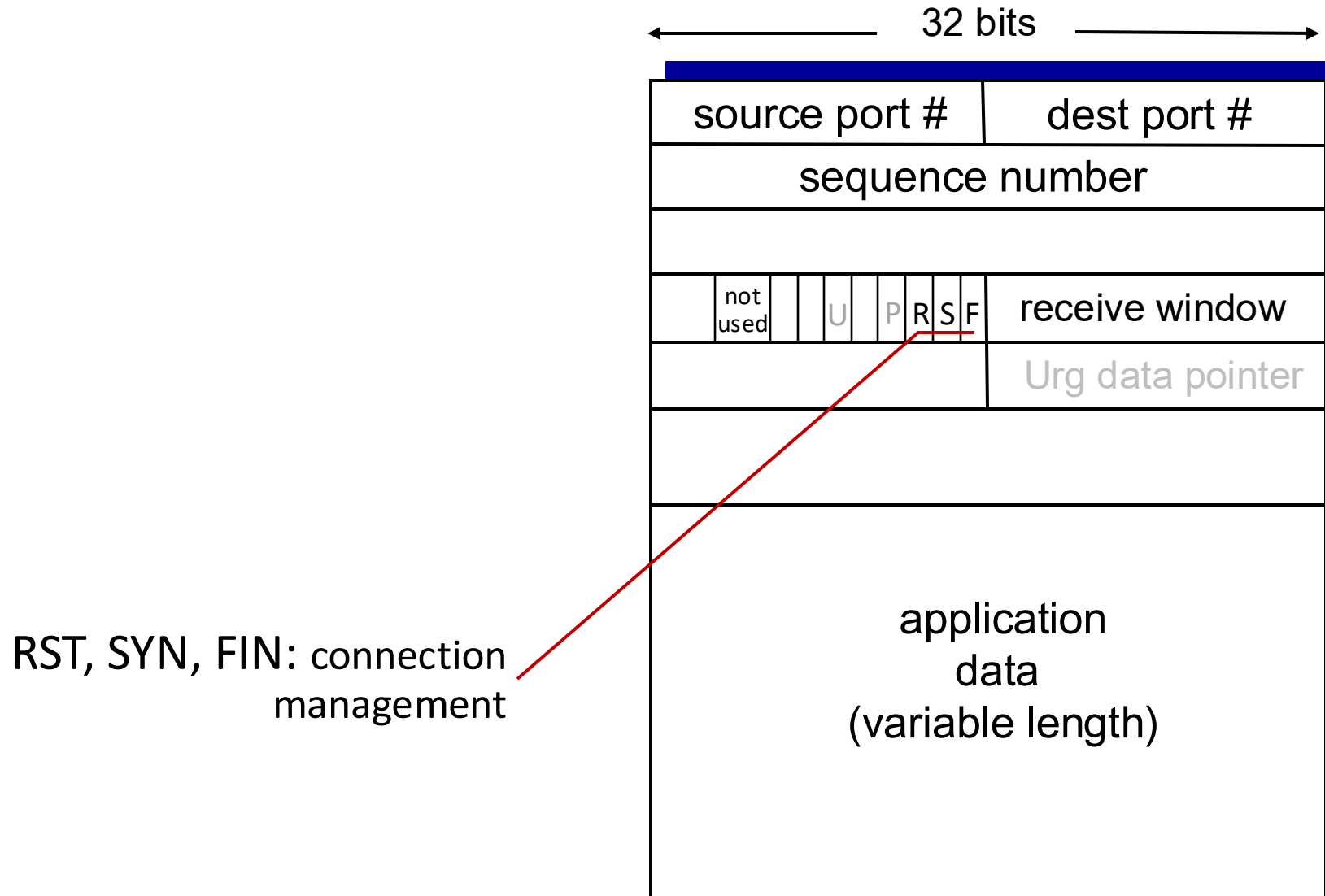
Problem: half open connection! (no client)

# 2-way handshake scenarios



 Problem: dup data accepted!

# TCP segment structure



# TCP 3-way handshake

Pay close attention to sequence and  
ack numbers during handshake

## Client state

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

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

SYNSENT

ESTAB

choose init seq num, x  
send TCP SYN msg

SYNbit=1, Seq=x

SYNbit=1, Seq=y  
ACKbit=1; ACKnum=x+1

received SYNACK(x)  
indicates server is live;  
send ACK for SYNACK;  
this segment may contain  
client-to-server data

ACKbit=1, ACKnum=y+1

received ACK(y)  
indicates client is live

## Server state

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

LISTEN

SYN RCVD

ESTAB





# 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

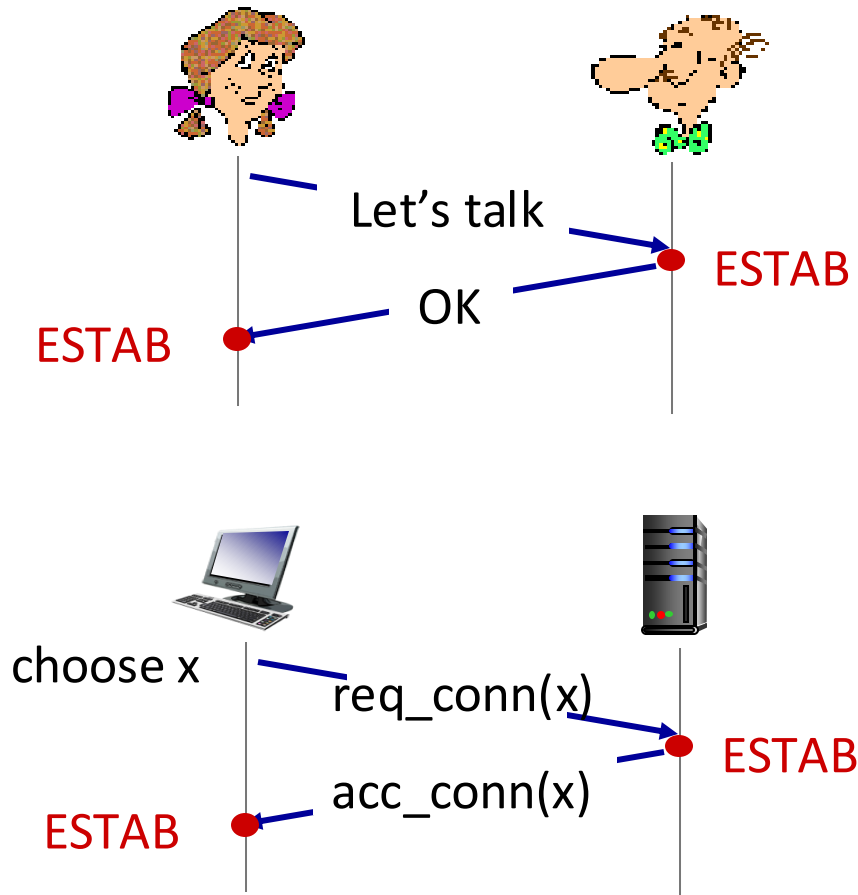
# Knowledge Check

- 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

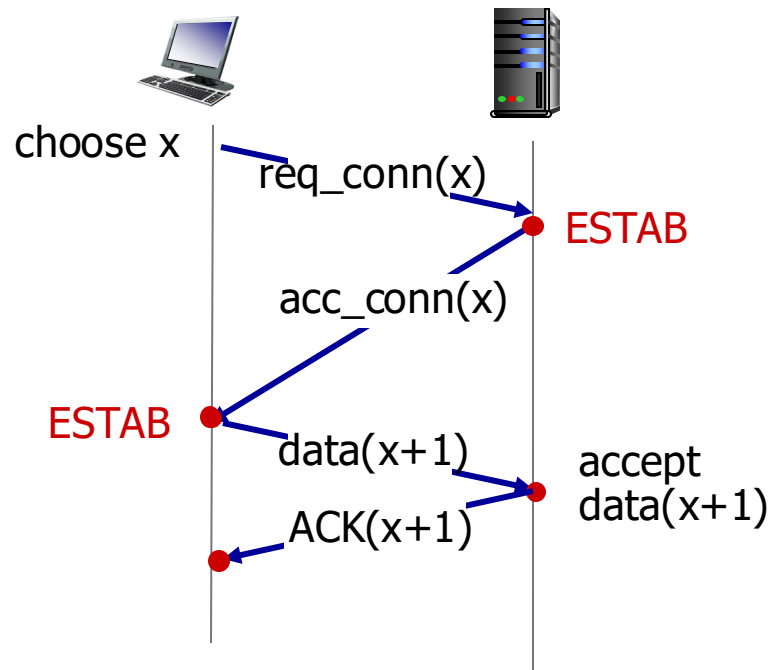
2-way handshake:



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

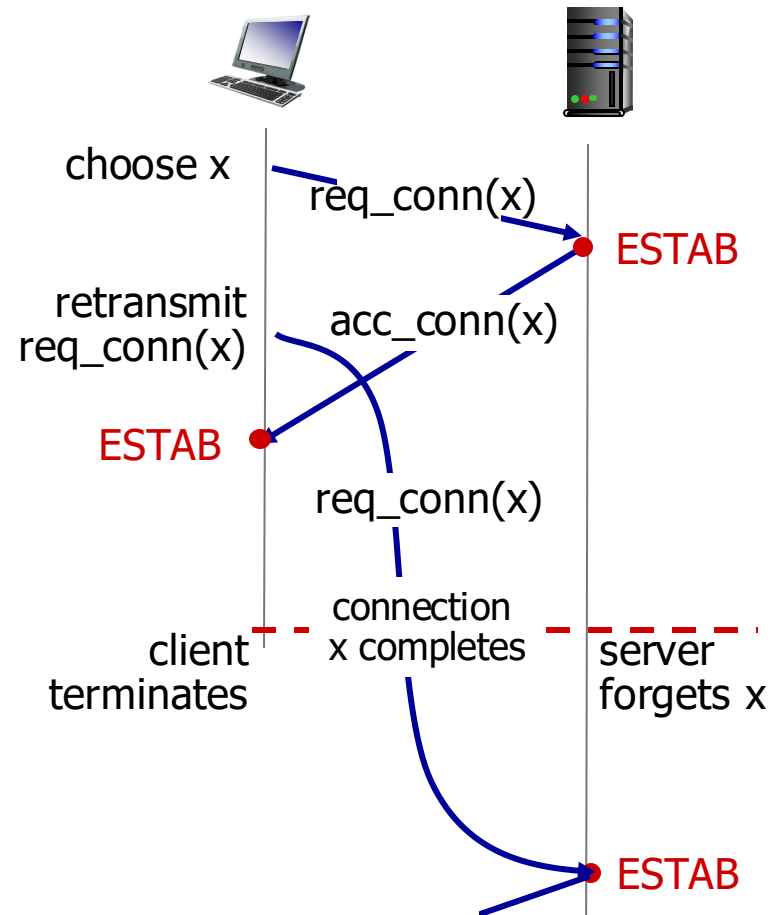
# 2-way handshake scenarios



No problem!

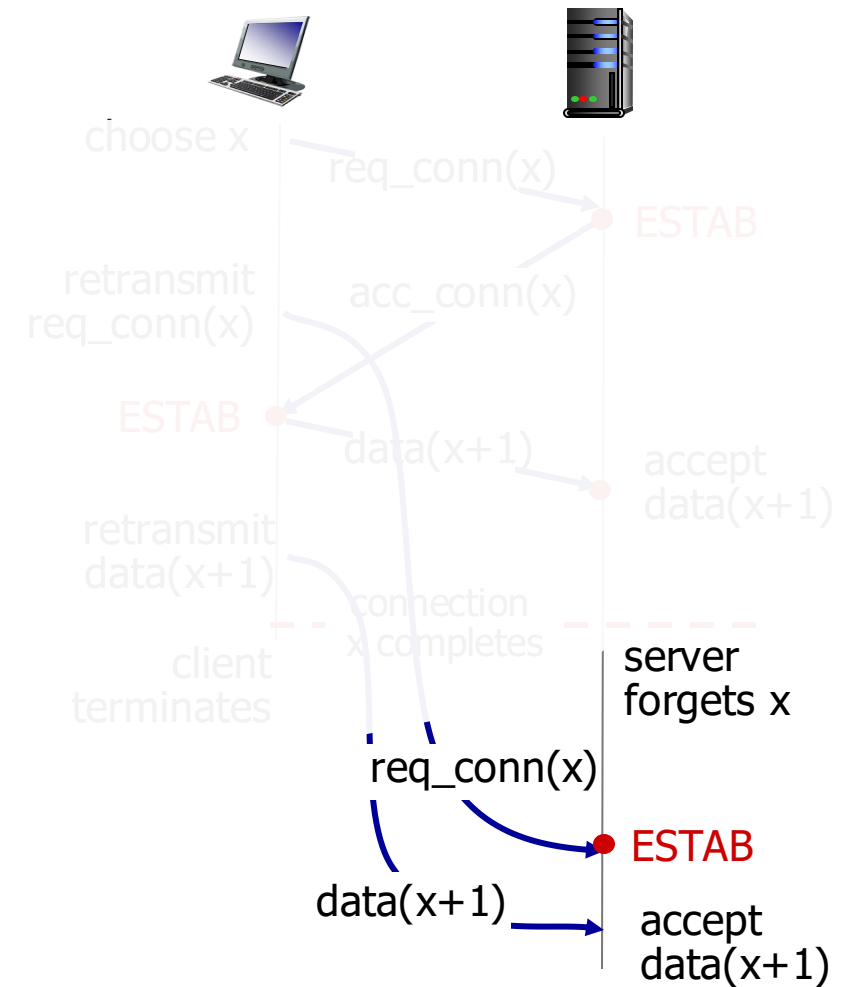


# 2-way handshake scenarios



Problem: half open connection! (no client)

# 2-way handshake scenarios



**Problem: dup data accepted!**



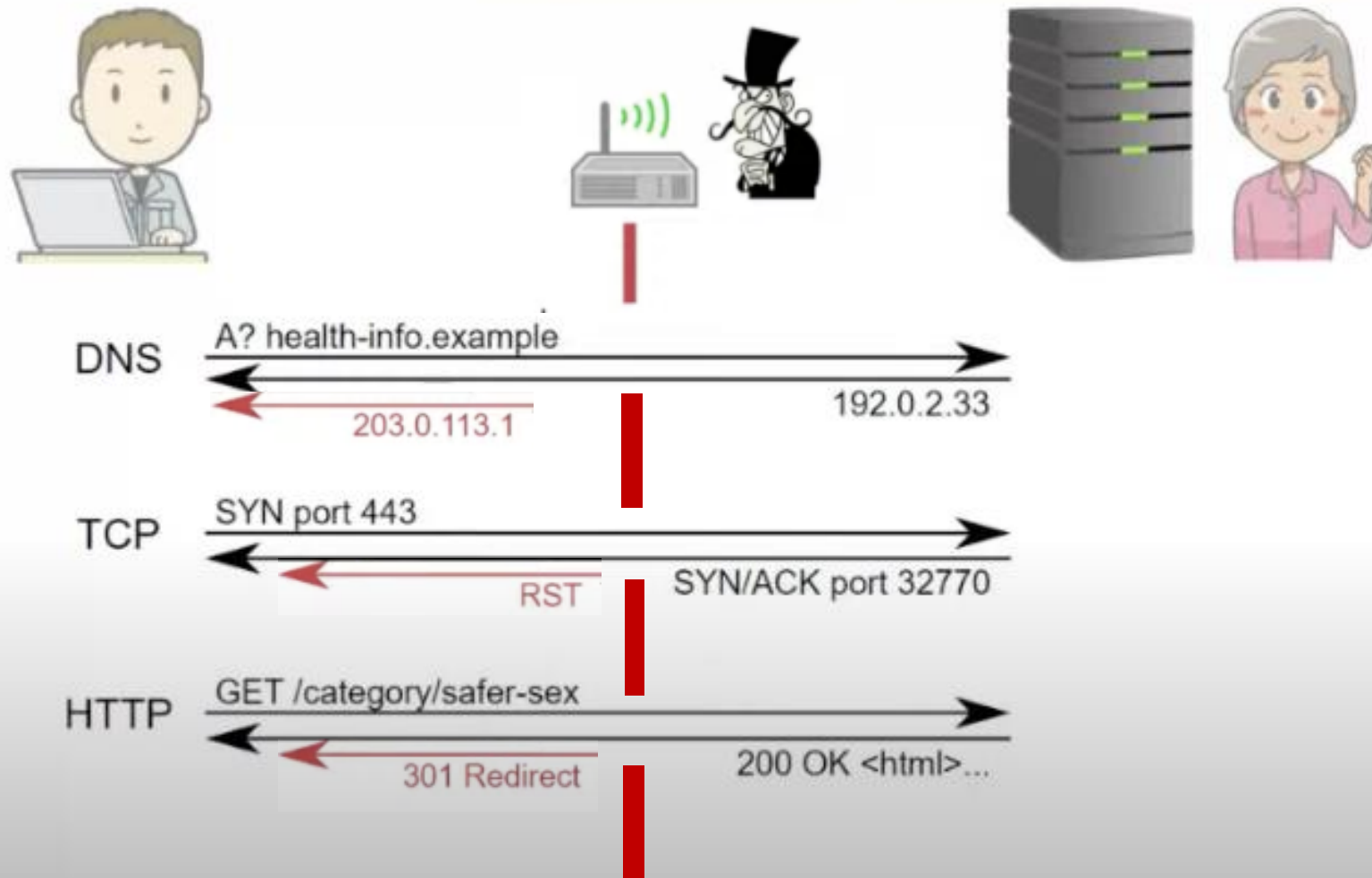
# SYN flood attack

- A common DoS attack

# Nmap-port scanning tool

- say port 6789, on a target host, nmap will send a TCP SYN segment
- Three possible outcomes:
  1. Receives a TCP synack segment
  2. Receives a TCP RST segment
    - Syn segment reached the target host, but the target host is not running an application with tcp port 6789.
    - But the sender at least knows that the segments destined to the host at port 6789 are not blocked
  3. Receives nothing

# Internet Censorship



Source: <https://www.youtube.com/watch?v=6iJ7KczFARw>

# ICLab: Detailed Probes for Network Censorship

Internet Measurement Village 2020

Presenter: Zachary Weinberg

<https://iclab.org/> • [info@iclab.org](mailto:info@iclab.org)



Arian Akhavan  
Niaki



Shinyoung Cho



Zachary Weinberg



Nguyen Phong  
Hoang



Abbas  
Razaghpanah



Diogo Barradas



Nicolas Christin



Phillipa Gill