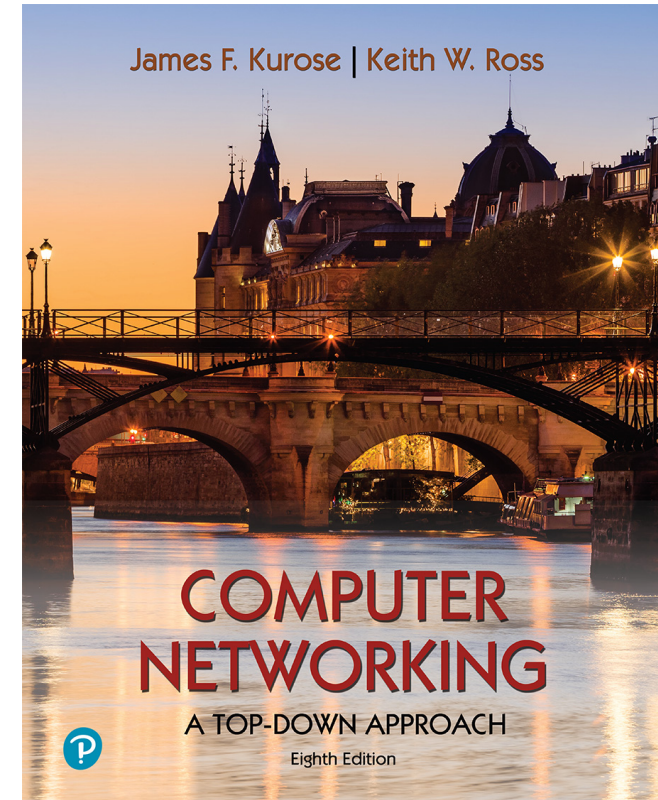


# Chapter 3

## Transport Layer



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

8<sup>th</sup> edition

Jim Kurose, Keith Ross  
Pearson, 2020

Acknowledgement: Based on the textbook's website:  
[https://gaia.cs.umass.edu/kurose\\_ross/index.php](https://gaia.cs.umass.edu/kurose_ross/index.php)

# Chapter 3: roadmap

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

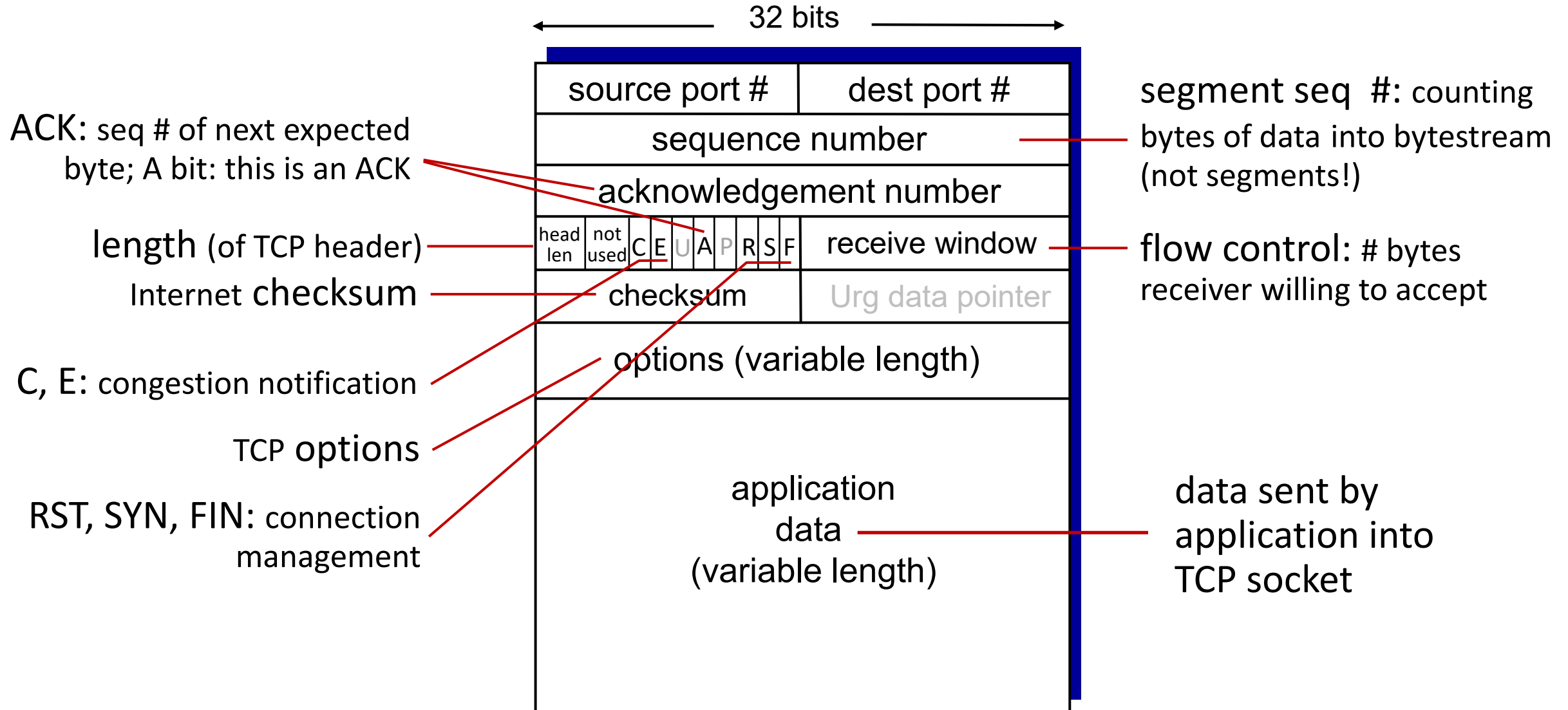


# TCP: overview

RFCs: 793, 1122, 2018, 5681, 7323

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order *byte stream*:**
  - no “message boundaries”
- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- **cumulative ACKs**
- **pipelining:**
  - TCP congestion and flow control set window size
- **connection-oriented:**
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- **flow controlled:**
  - sender will not overwhelm receiver

# TCP segment structure



# TCP sequence numbers, ACKs

## Sequence numbers:

- byte stream “number” of first byte in segment’s data

## Acknowledgements:

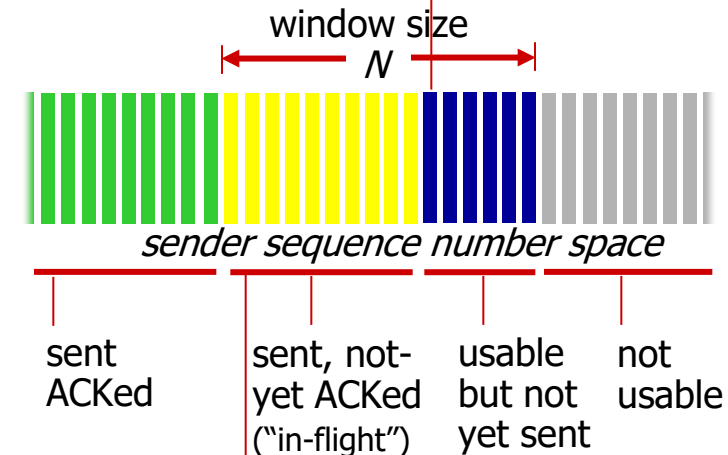
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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

outgoing segment from sender

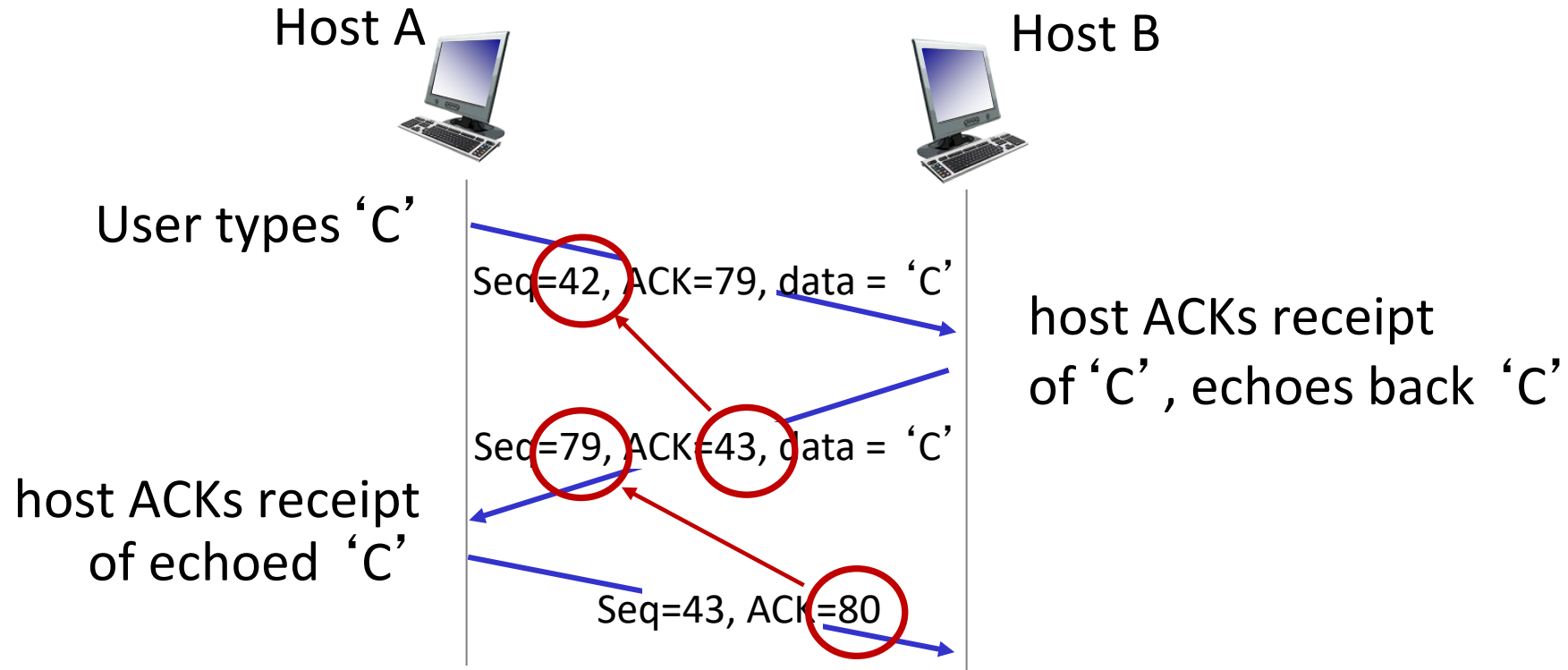
source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer



outgoing segment from receiver

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

# TCP sequence 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

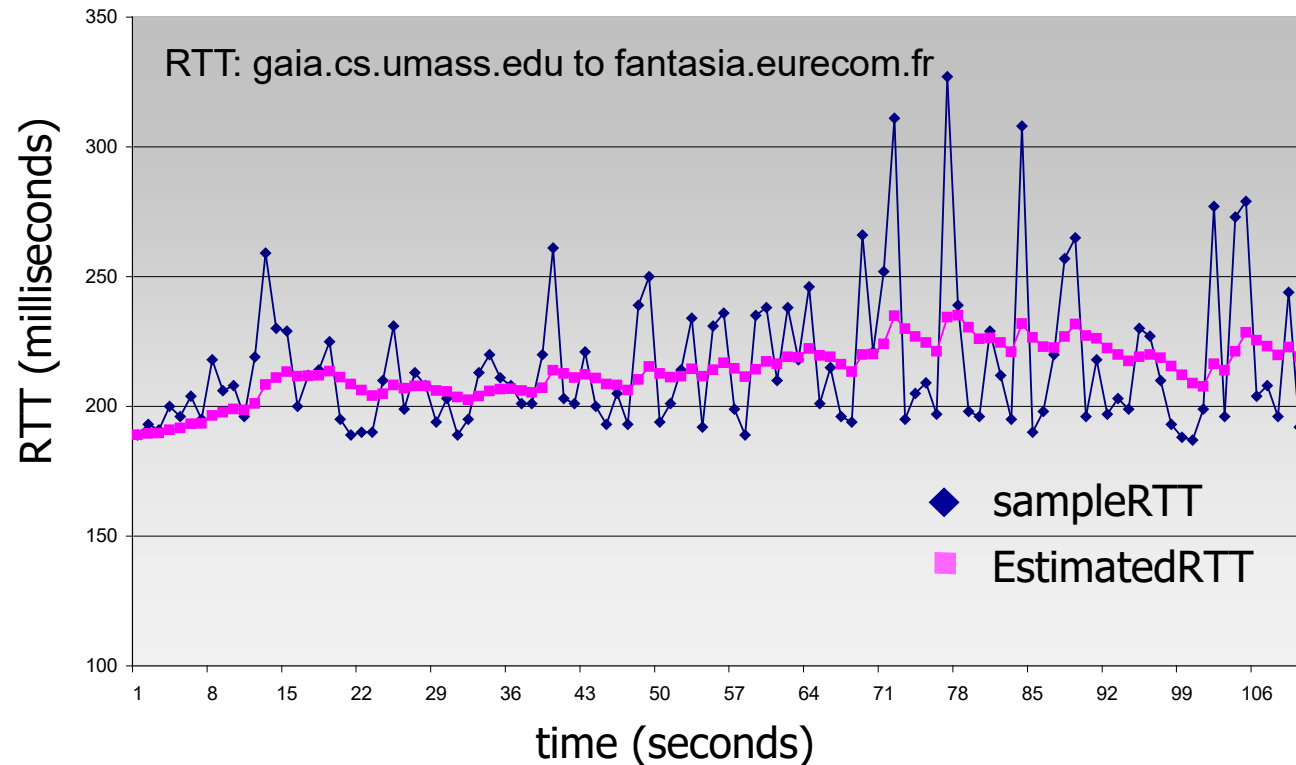
Q: how to estimate RTT?

- *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

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

- 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,  $\beta = 0.25$ )

\* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose\\_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)

# TCP Sender (simplified)

event: data received from application

- 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**

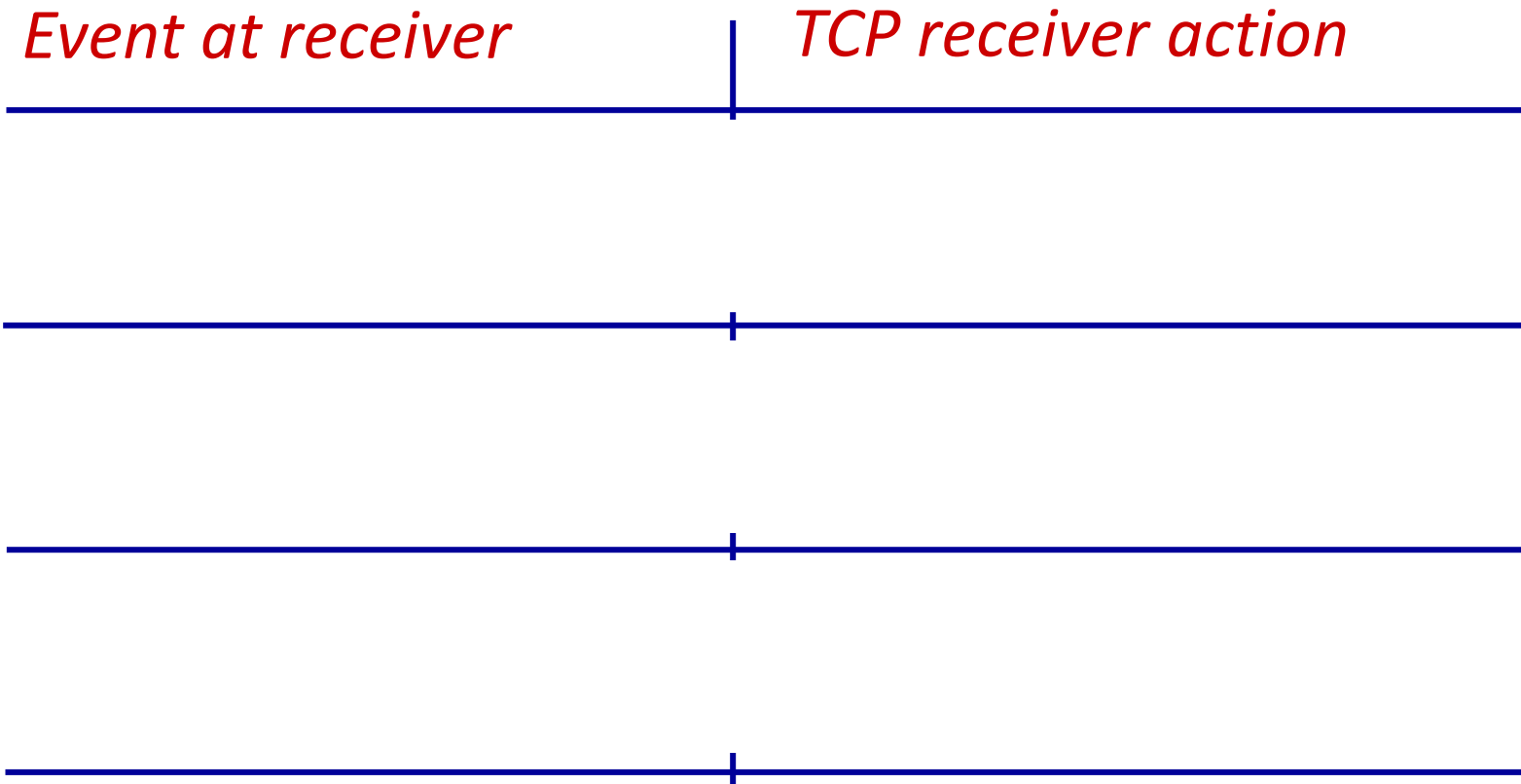
event: *timeout*

- retransmit segment that caused timeout
- restart timer

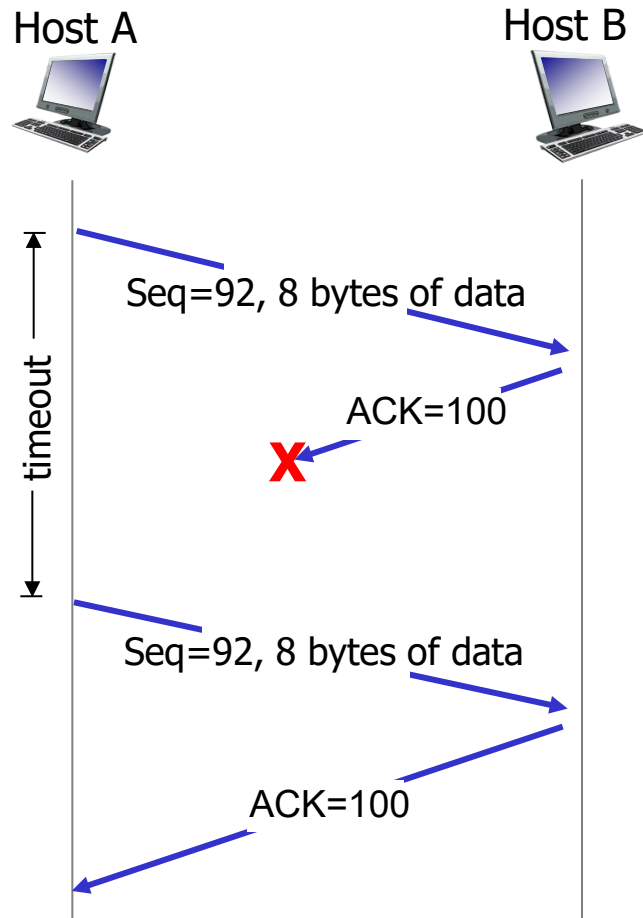
event: *ACK received*

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

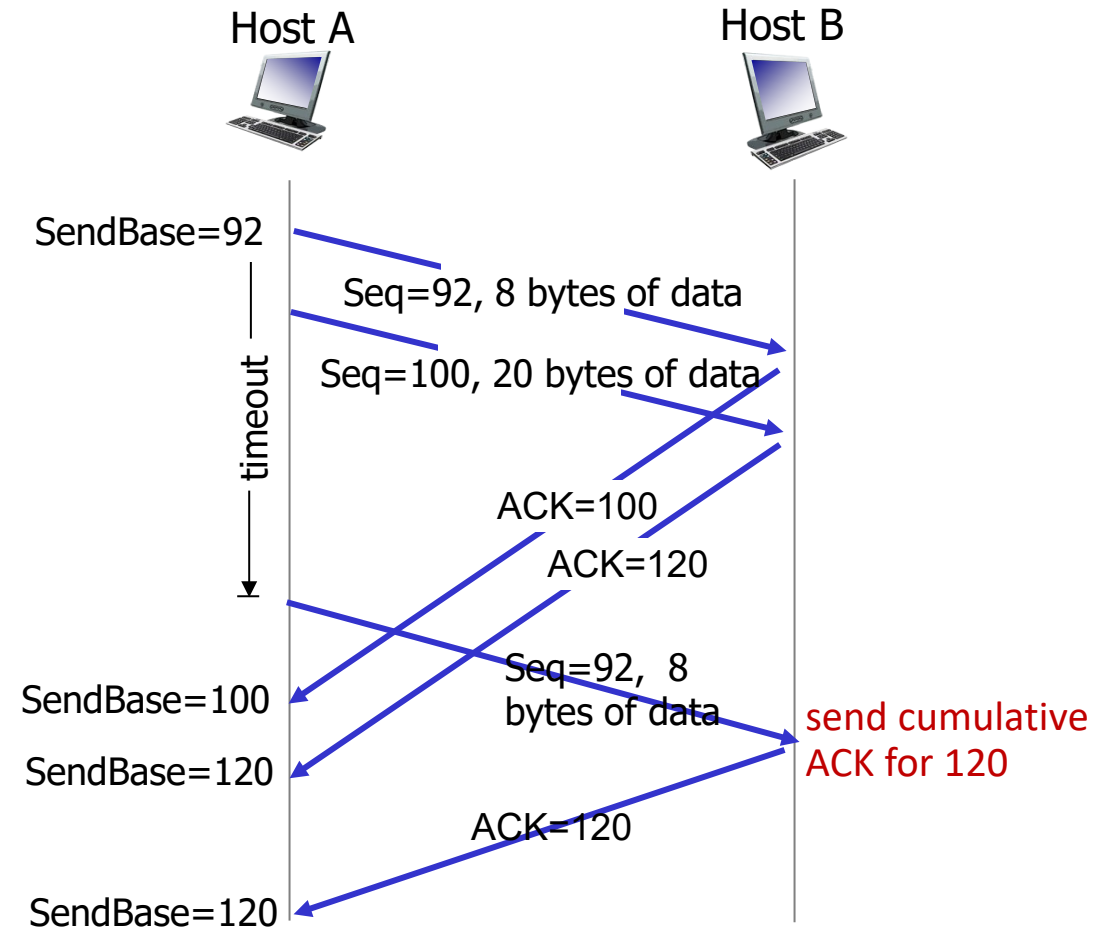
# TCP Receiver: ACK generation [RFC 5681]



# TCP: retransmission scenarios

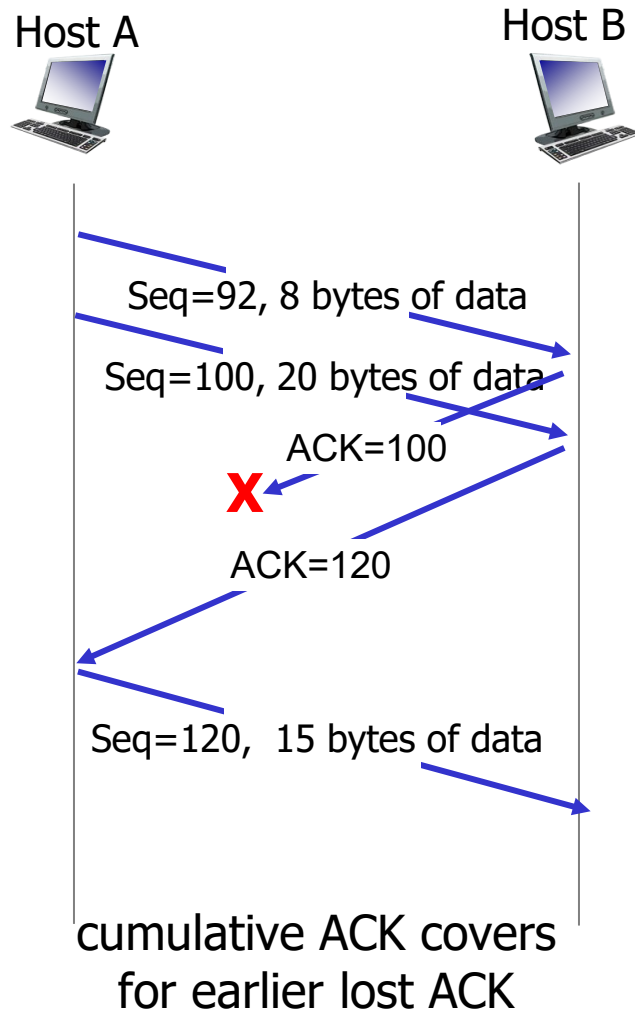


lost ACK scenario



premature timeout

# TCP: retransmission scenarios



- Q: what happens if the segment with `Seq=92`, 8 bytes of data from Host A to Host B gets lost?
- A: Host B will not send `ACK=120`, since a cumulative `ACK=120` implies that all previous segments with `Seq ≤ 120` have been received

# Chapter 3: 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



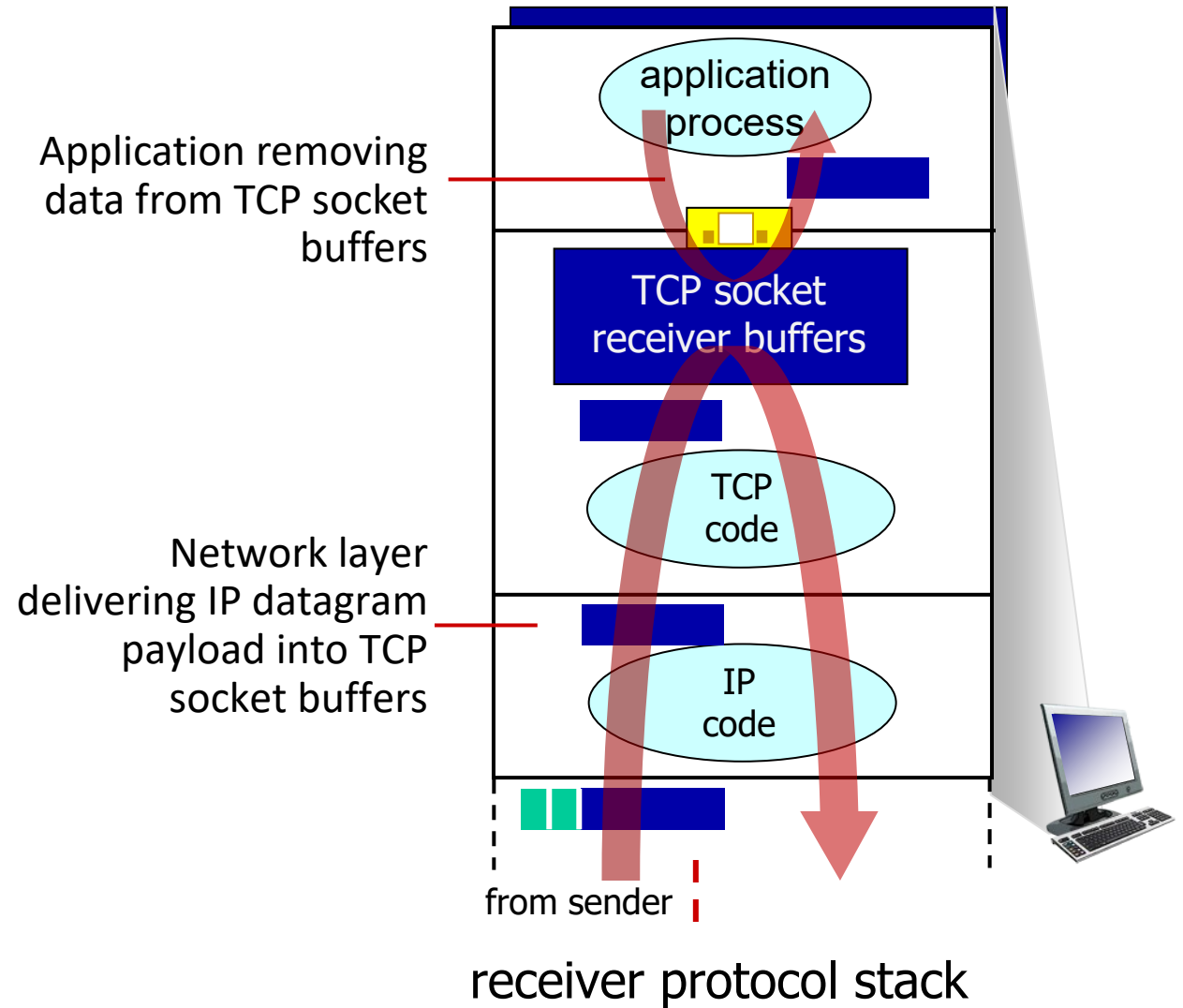
The diagram illustrates the receiver protocol stack, showing the flow of data from the network layer to the application layer. The stack is divided into four layers:

- Application layer:** Contains the **application process** (light blue oval). A red arrow points from the application process to the TCP socket receiver buffers, labeled "Application removing data from TCP socket buffers".
- Transport layer:** Contains the **TCP socket receiver buffers** (dark blue rectangle). A red arrow points from the TCP socket receiver buffers to the TCP code, labeled "Network layer delivering IP datagram payload into TCP socket buffers".
- Network layer:** Contains the **TCP code** (light blue oval).
- Link layer:** Contains the **IP code** (light blue oval).

At the bottom, a red arrow points from the IP code to the TCP socket receiver buffers, labeled "from sender". The entire stack is labeled "receiver protocol stack" at the bottom. A small illustration of a computer is shown on the right side of the diagram.

# TCP flow control

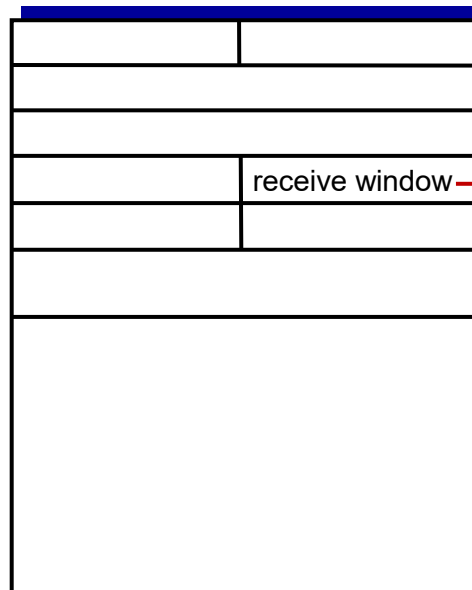
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





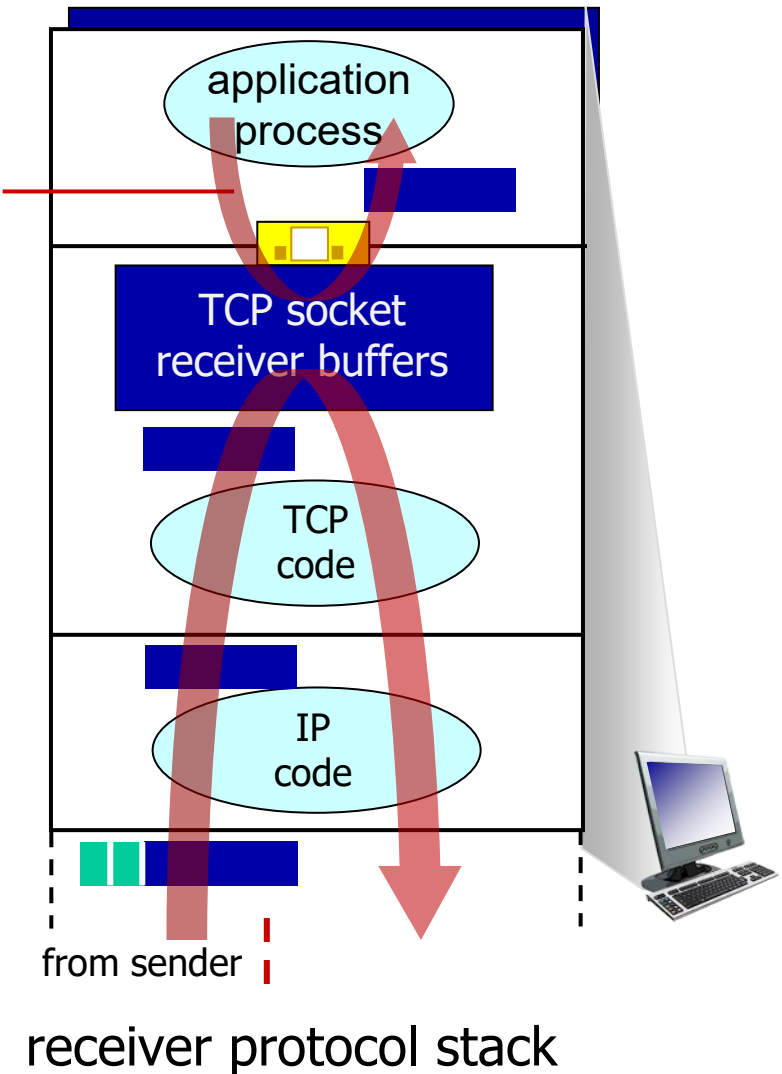
# TCP flow control

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

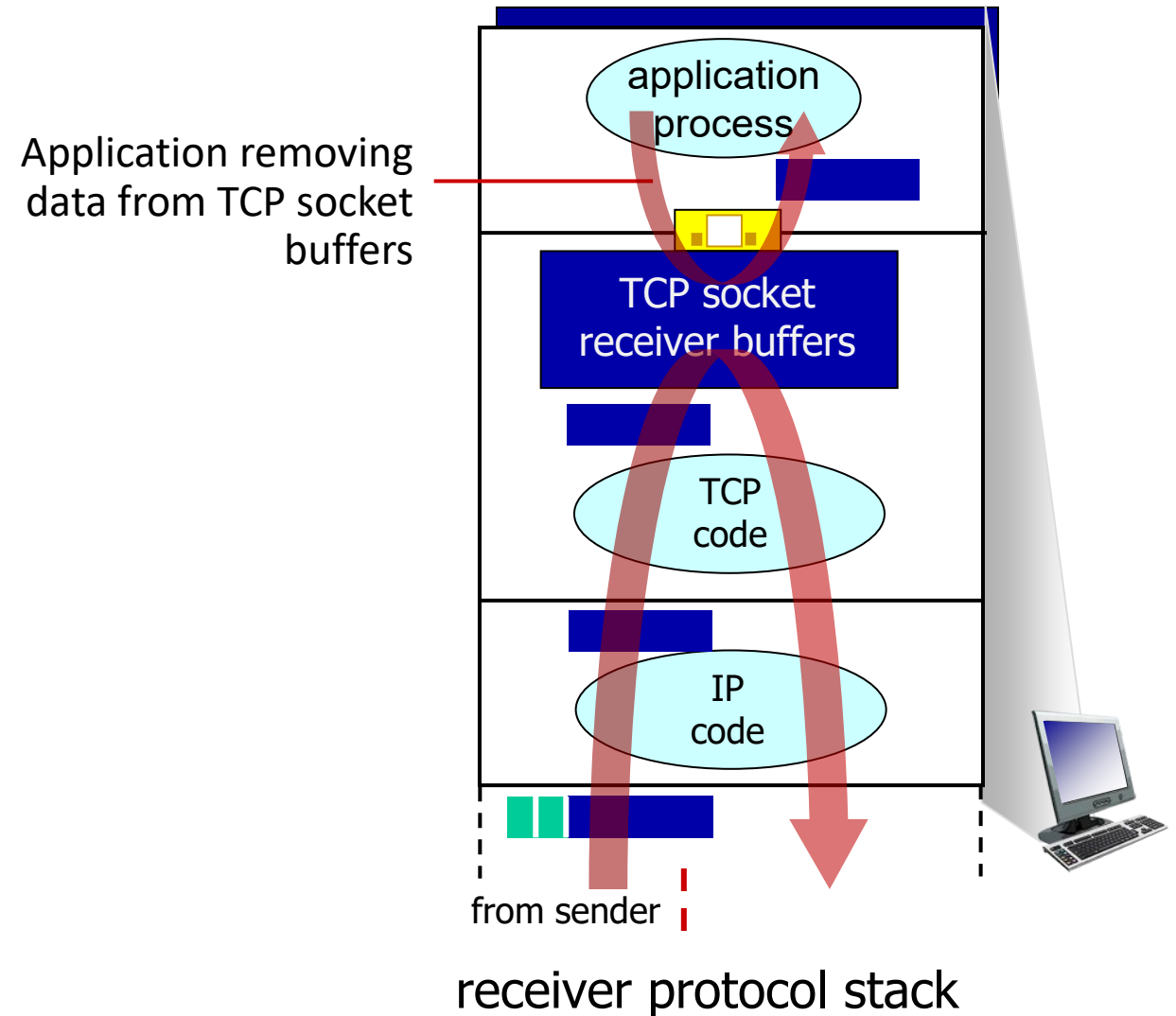


# TCP flow control

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

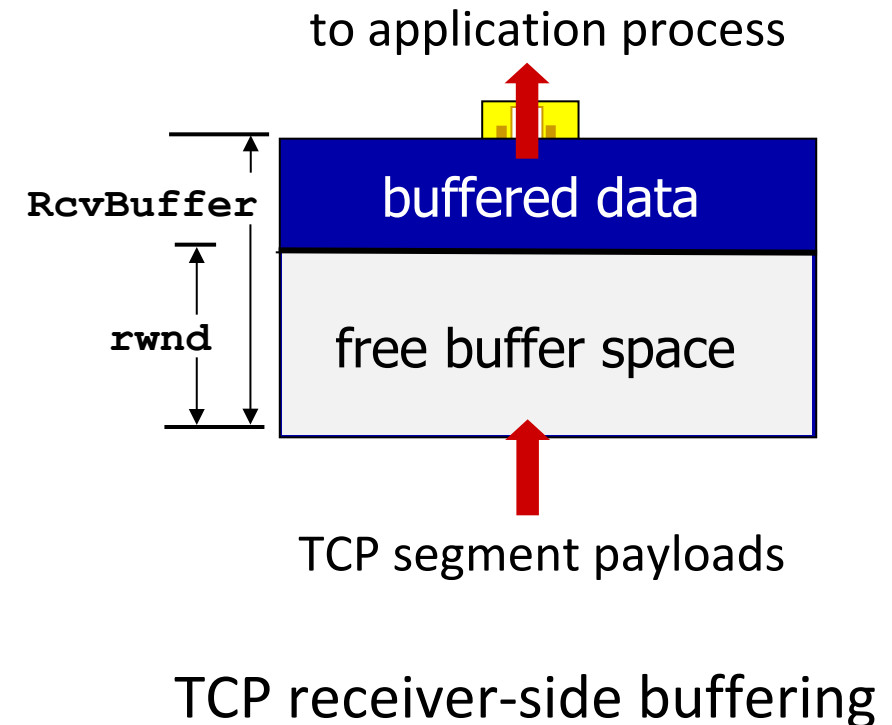
## —flow control—

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



# TCP flow control

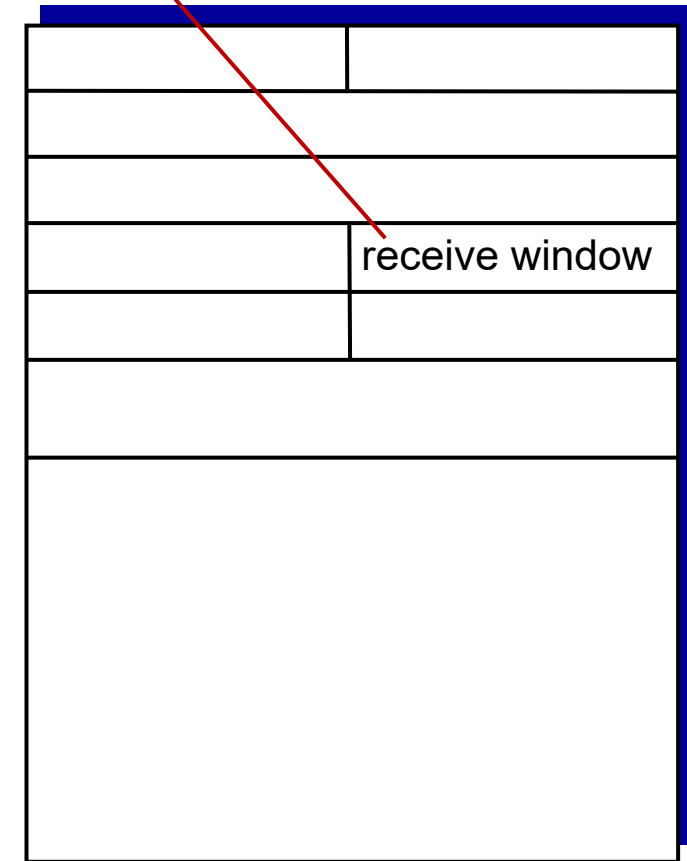
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive 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 (typical default is 4096 bytes)
  - 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

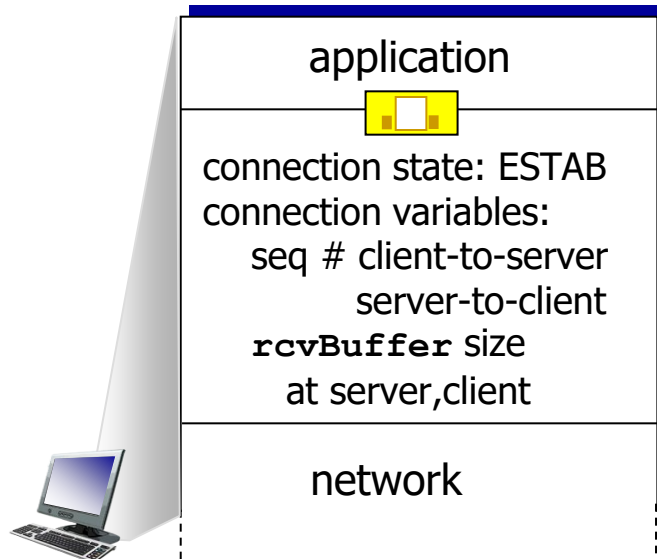


TCP segment format

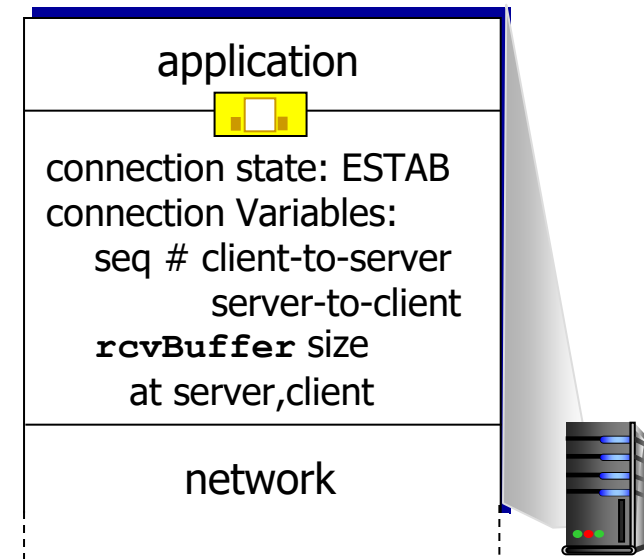
# TCP connection management

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)



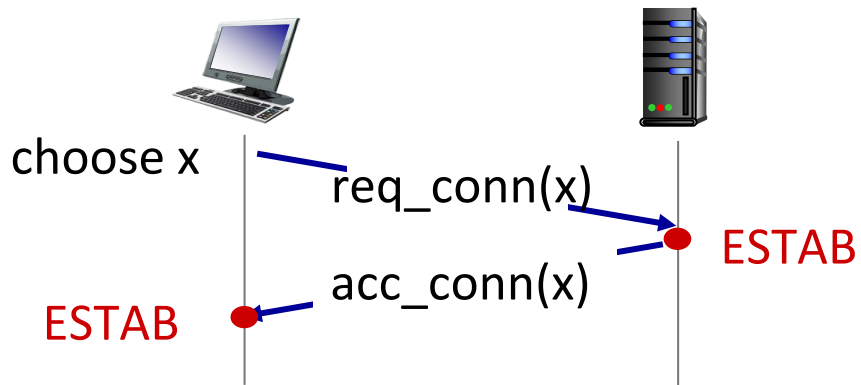
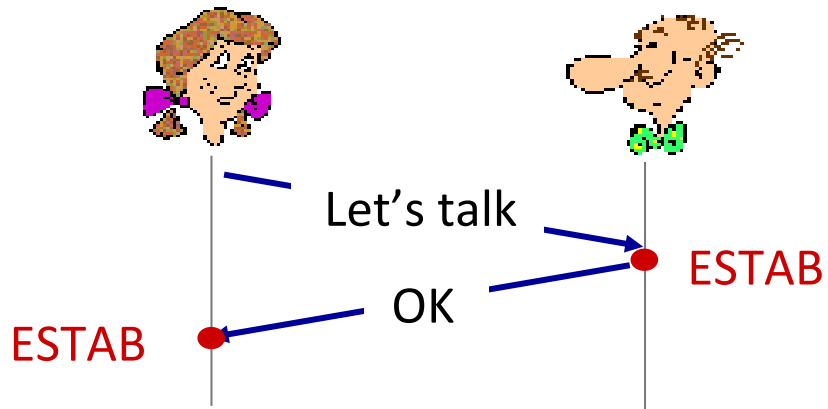
```
Socket clientSocket =  
    newSocket("hostname", "port number");
```



```
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

# TCP 3-way handshake

## Server state

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

## Client state

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
```

LISTEN

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

LISTEN

SYN RCVD

ESTAB

1. It allows both parties to synchronize their sequence numbers
2. Confirm that both sides are ready for data transfer
3. Agree on initial parameters for the connection



# 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