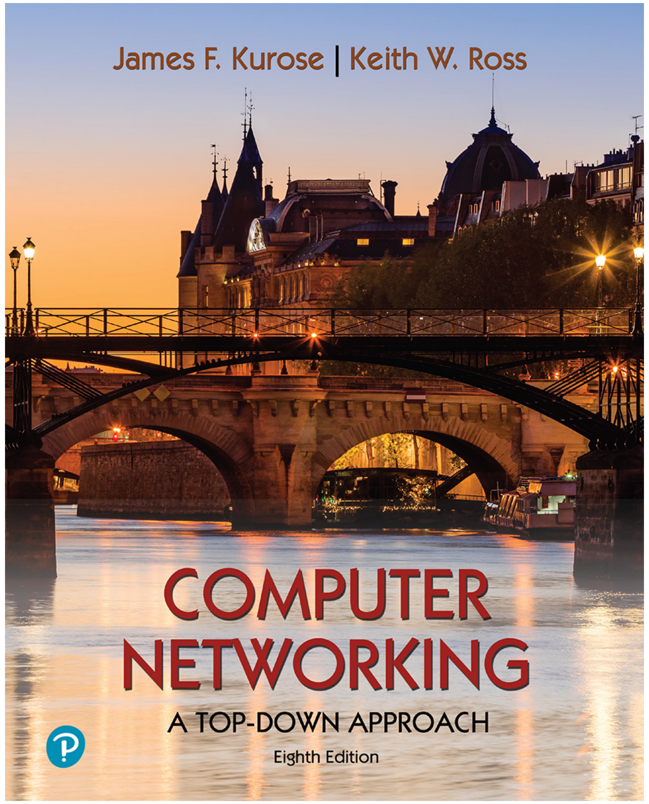
Chapter 3
Transport Layer



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

Acknowledgement: Based on the textbook's website:
https://gaia.cs.umass.edu/kurose_ross/index.php

Chapter 3: roadmap

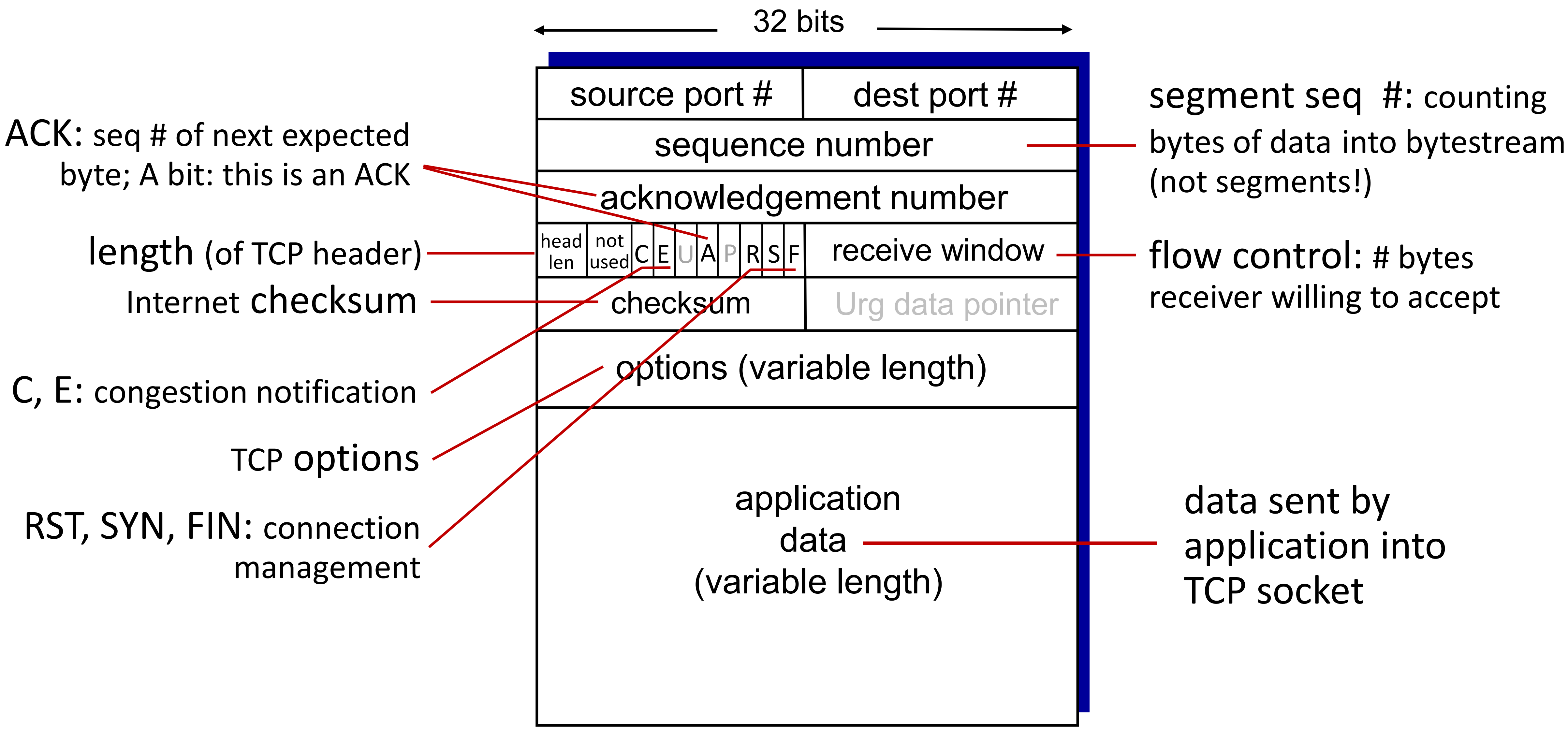
- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

IMPORTANT

TCP: overview RFCs: 793,1122, 2018, 5681, 7323

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - 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:

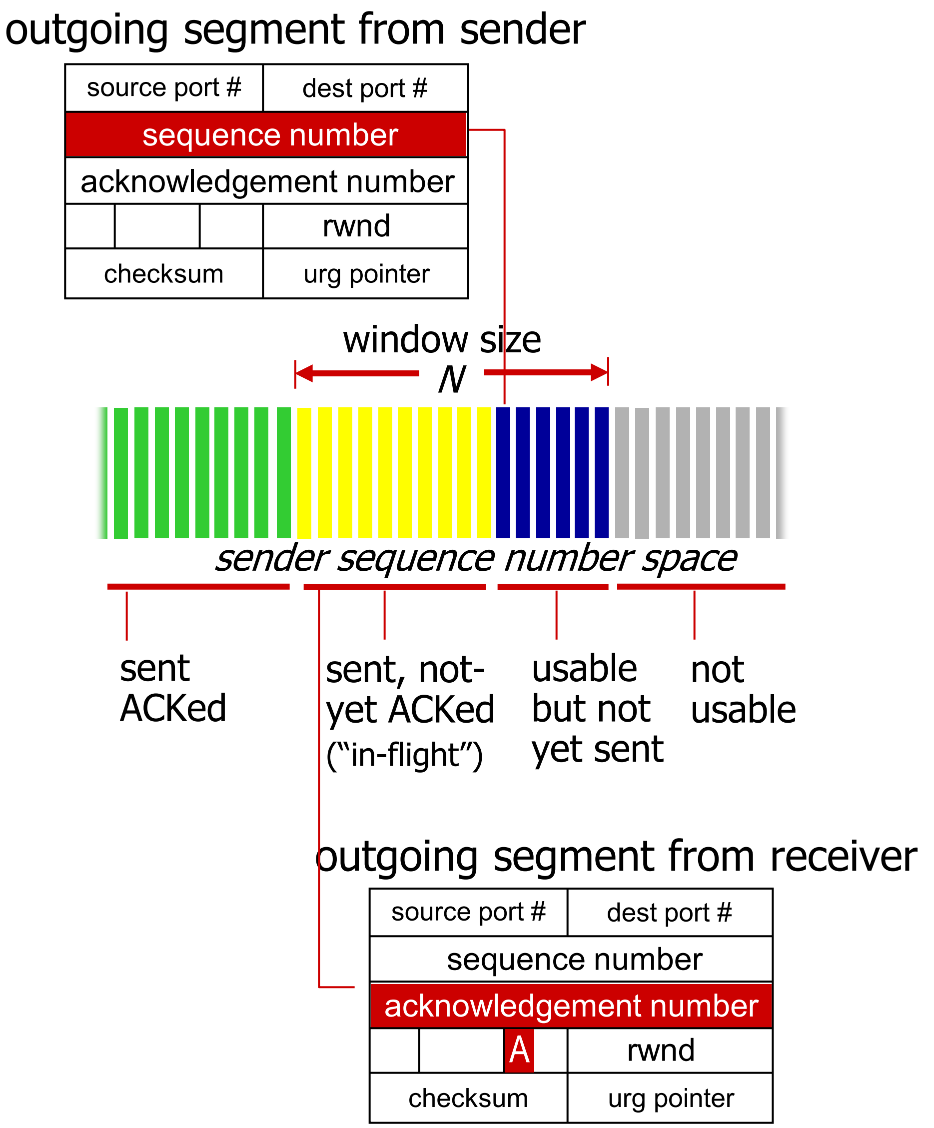
- byte stream "number" of first byte in segment's data

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



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

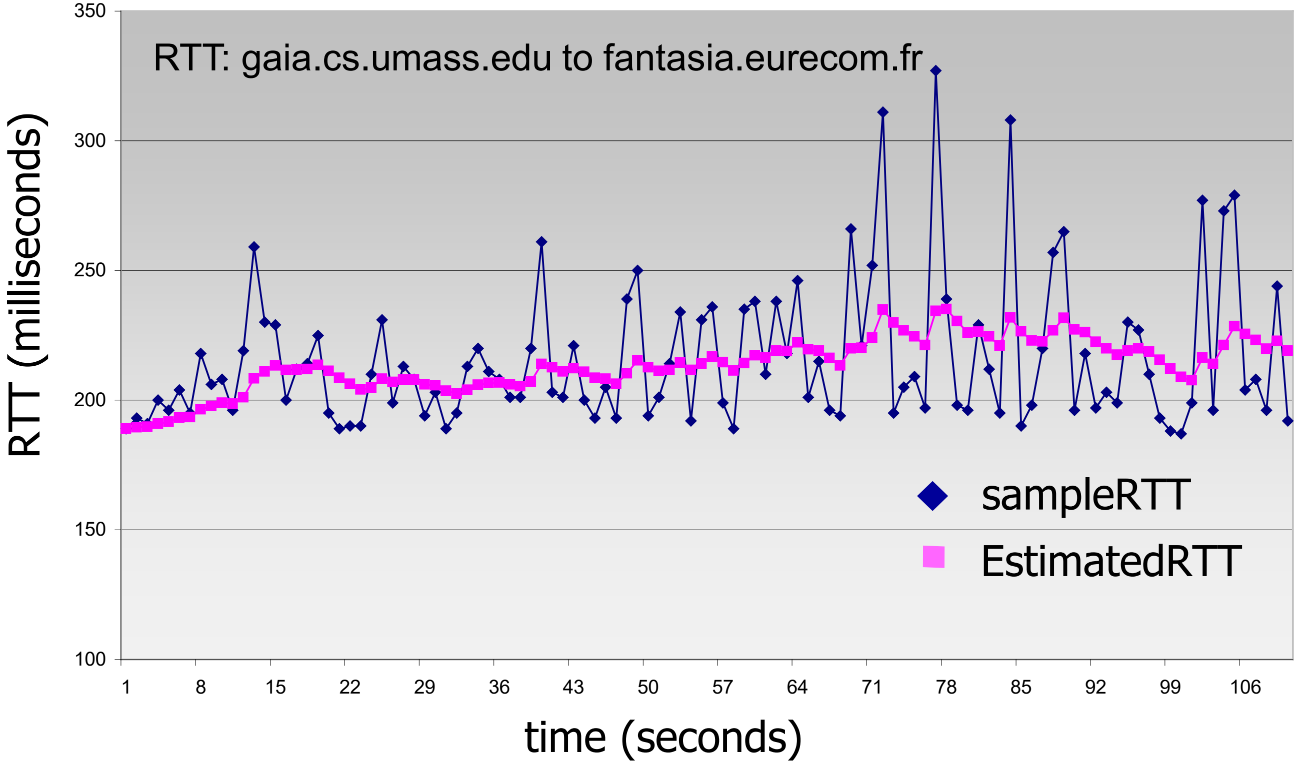
Q: how to estimate RTT?

- `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, β = 0.25)

* Check out the online interactive exercises for more examples: 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

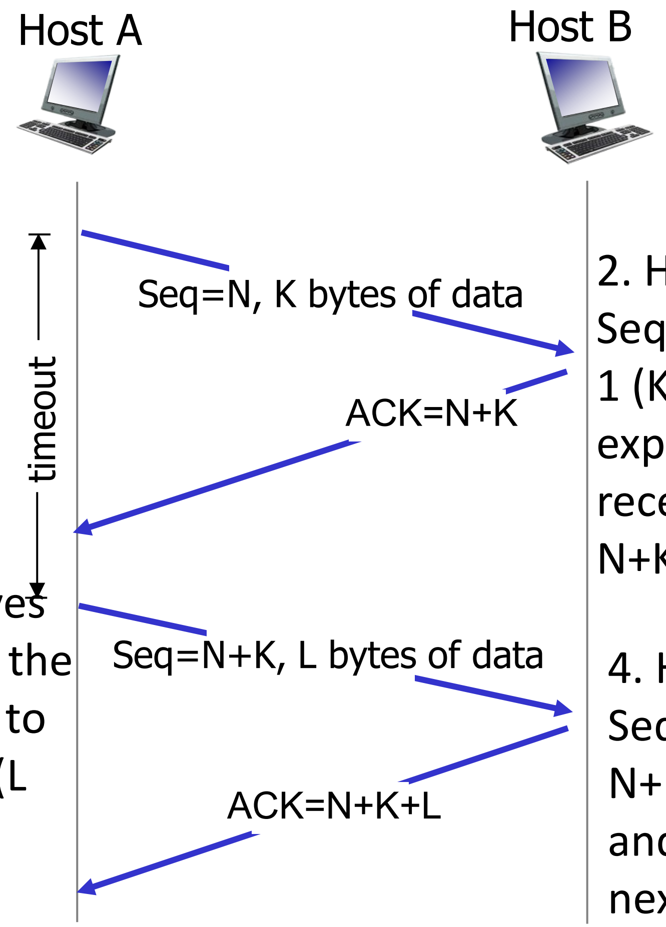
TCP Receiver: ACK generation [RFC 5681]

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

IMPORTANT

TCP sequence numbers, ACKs

1. HostA sends Seq# N to Seq# N+K-1 (K Bytes)



2. HostB receives Seq# N to Seq# N+K-1 (K Bytes), and expects the next received Seq# to be N+K

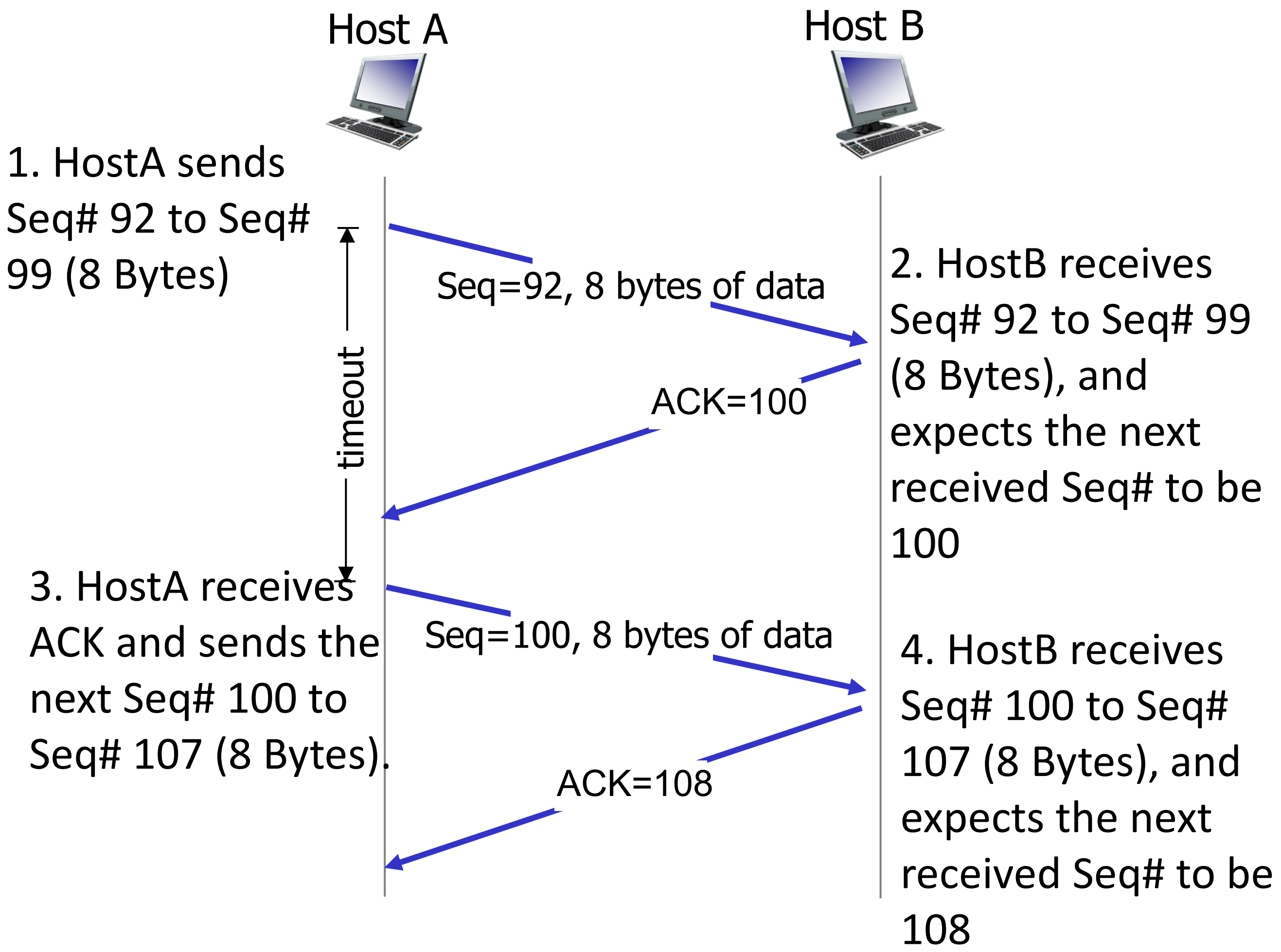
3. HostA receives ACK and sends the next Seq# N+K to Seq# N+K+L-1 (L Bytes).

4. HostB receives Seq# N+K to Seq# N+K+L-1 (L Bytes), and expects the next received Seq# to be N+K+L

Receiver ACK=N means that “I have received all Bytes up to sequence#N-1, and I am expecting the next Byte I receive to have Seq # N”.

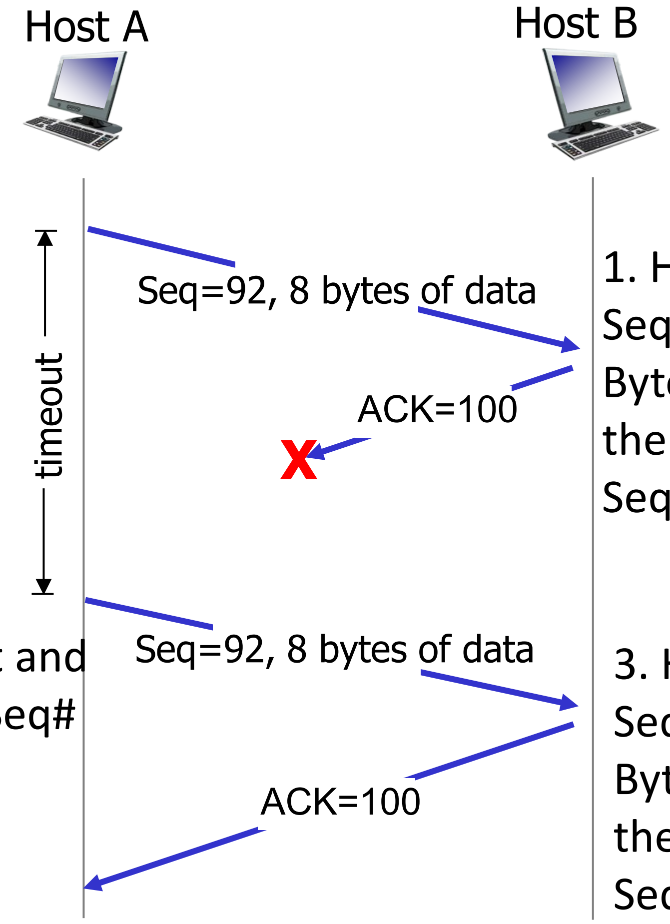
IMPORTANT

TCP sequence numbers, ACKs



IMPORTANT

TCP: retransmission scenarios

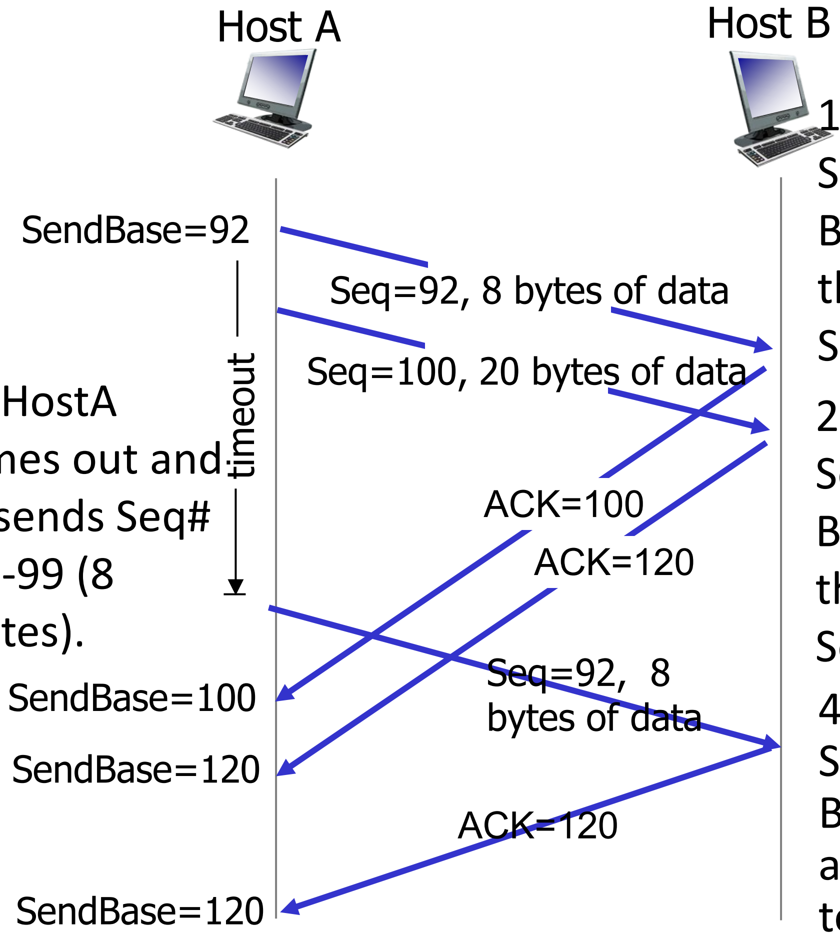


1. HostB receives Seq# 92-99 (8 Bytes), and expects the next received Seq# to be 100

2. HostA times out and resends Seq# 92-99 (8 Bytes).

3. HostB receives Seq# 92-99 (8 Bytes), and expects the next received Seq# to be 100

lost ACK scenario



1. HostB receives Seq# 92-99 (8 Bytes), and expects the next received Seq# to be 100

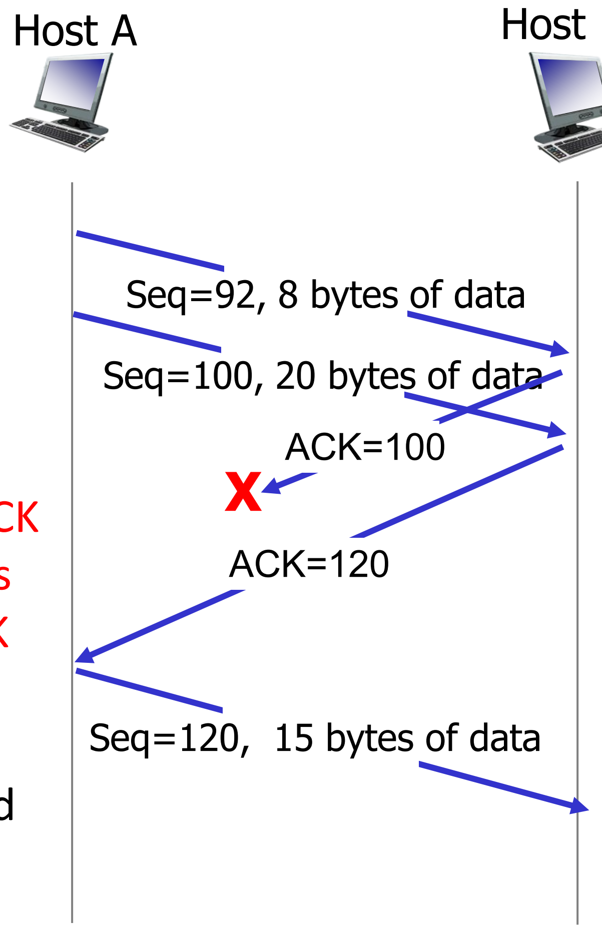
2. HostB receives Seq# 100-119 (20 Bytes), and expects the next received Seq# to be 120

3. HostA times out and resends Seq# 92-99 (8 Bytes).

4. HostB receives Seq# 92-99 (8 Bytes). But it has already received up to Seq# 119, so it sends cumulative ACK for Seq# 120

premature timeout

TCP: retransmission scenarios



1. HostB receives Seq# 92-99 (8 Bytes), and expects the next received Seq# to be 100, but the ACK is lost.

2. HostB receives Seq# 100-119 (20 Bytes), and expects the next received Seq# to be 120

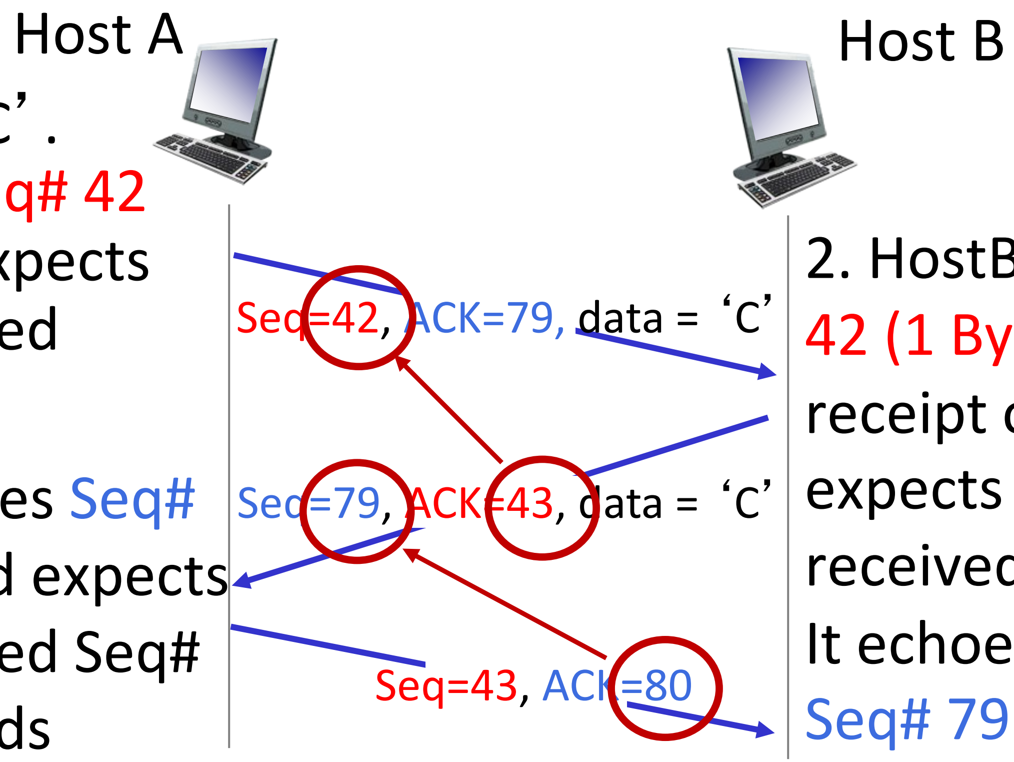
3. HostA receives ACK for Seq# 120. This cumulative ACK of Seq# 120 covers for earlier lost ACK of Seq#100, so HostA knows that HostB has received all bytes up to Seq#119, so it can send the next 15 Bytes (Seq#120-134).

- 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

TCP sequence numbers, ACKs

IMPORTANT

1. User types 'C'. HostA sends Seq# 42 (1 Byte), and expects the next received Seq# to be 79



2. HostB receives Seq# 42 (1 Byte), ACKs receipt of 'C', and expects the next received Seq# to be 43. It echoes back 'C', with Seq# 79 (1 Byte)

3. HostA receives Seq# 79 (1 Byte), and expects the next received Seq# to be 80. It sends another Byte, with Seq# 43 (1 Byte)

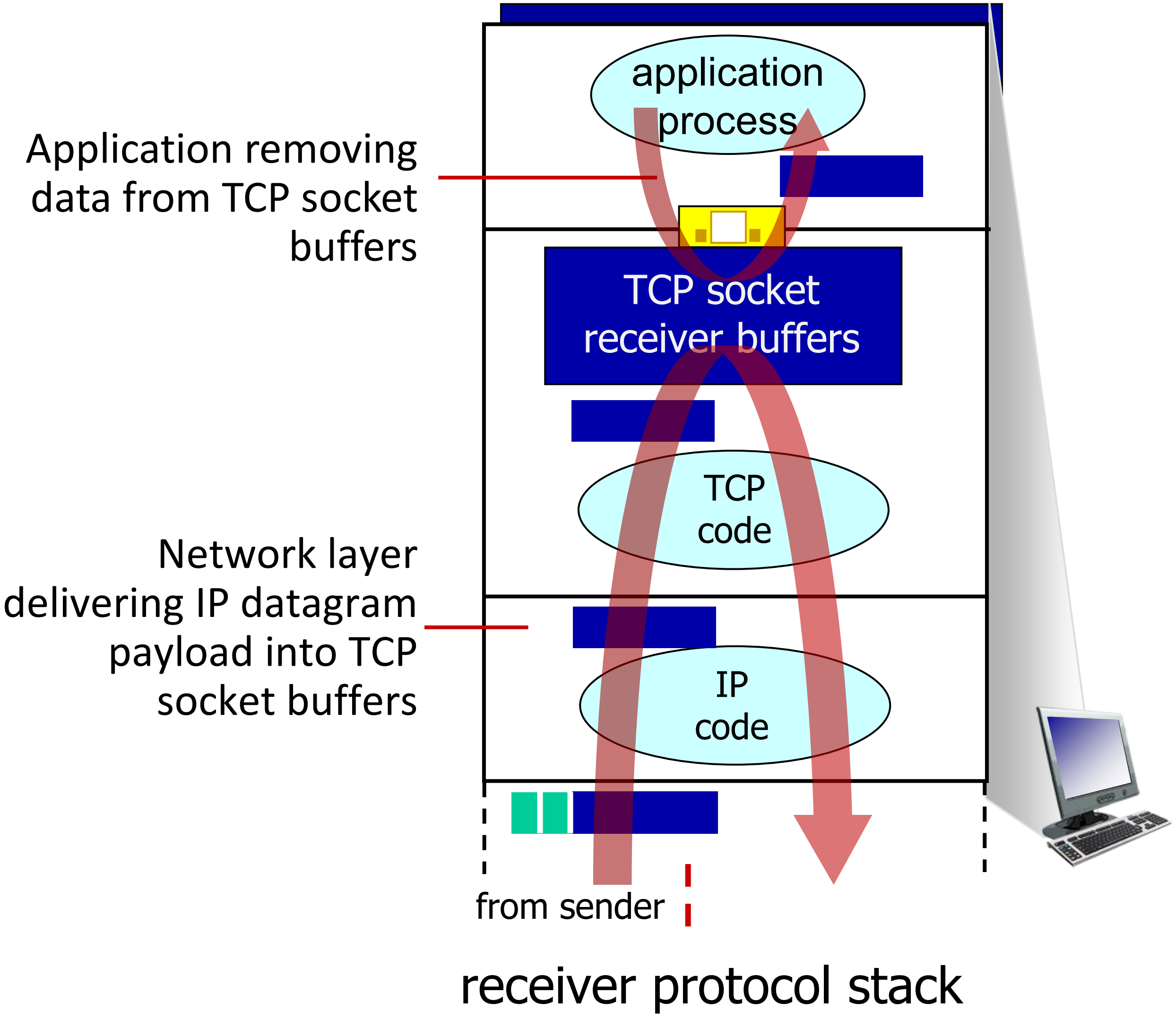
simple telnet scenario

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

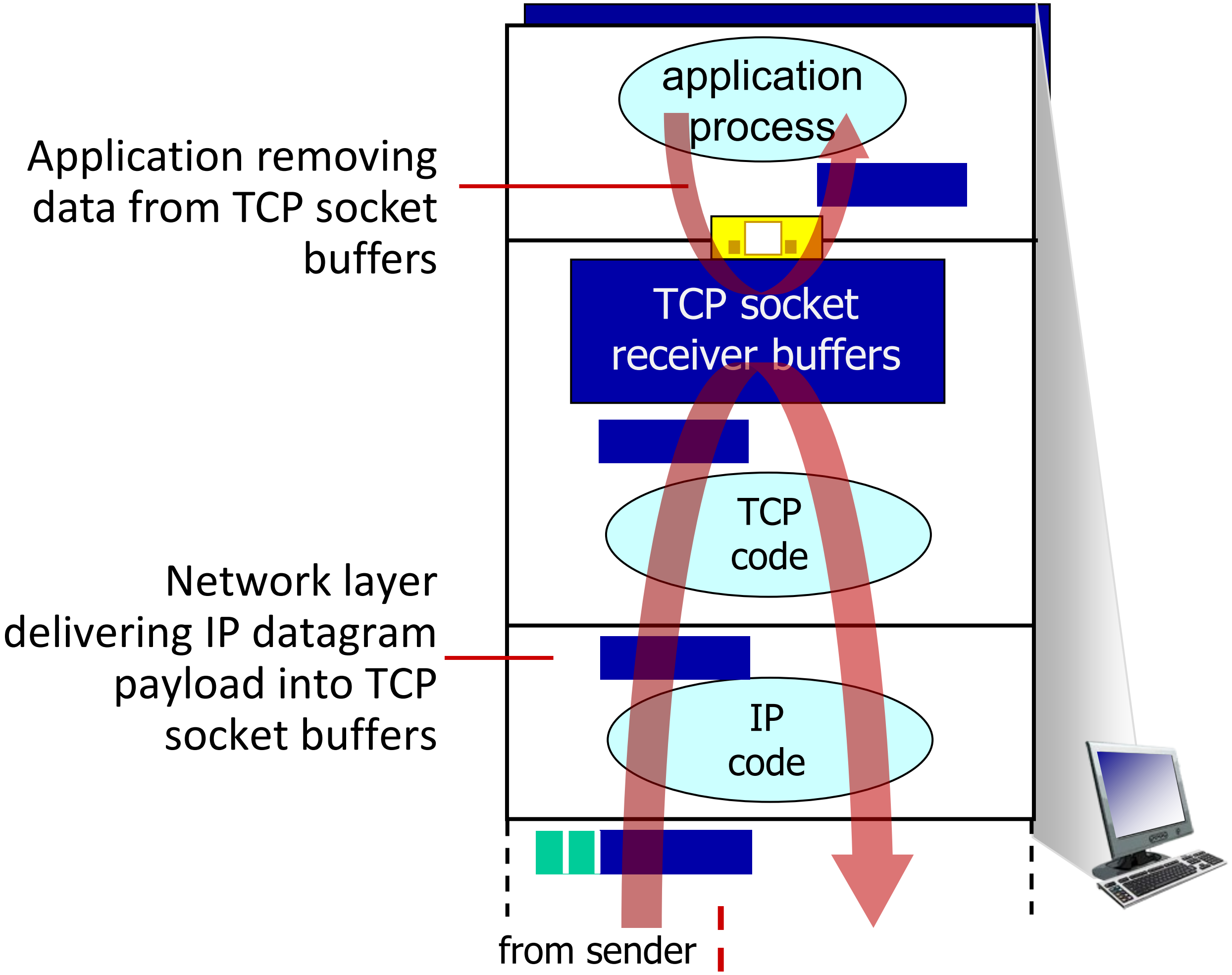
TCP flow control

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



TCP flow control

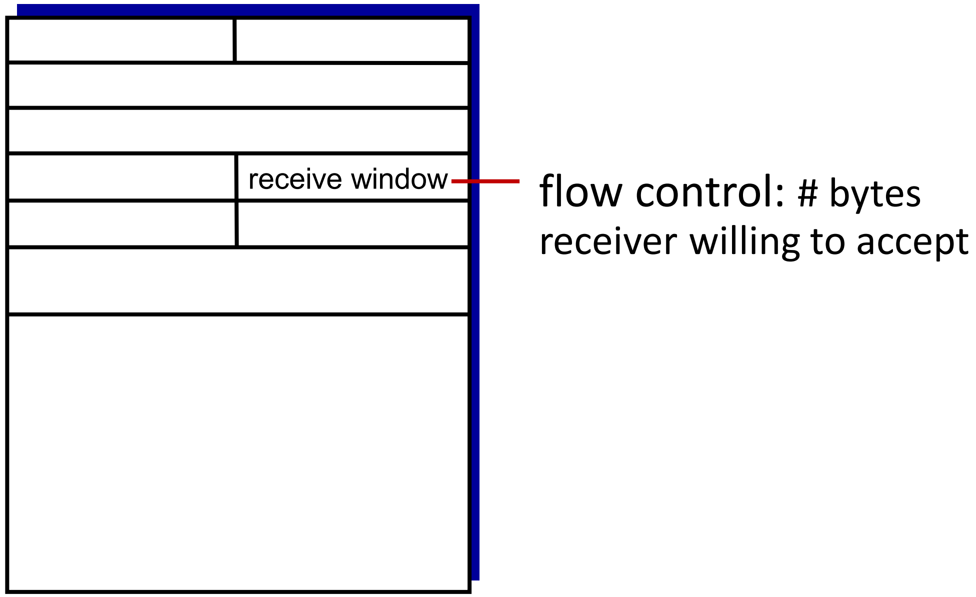
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?

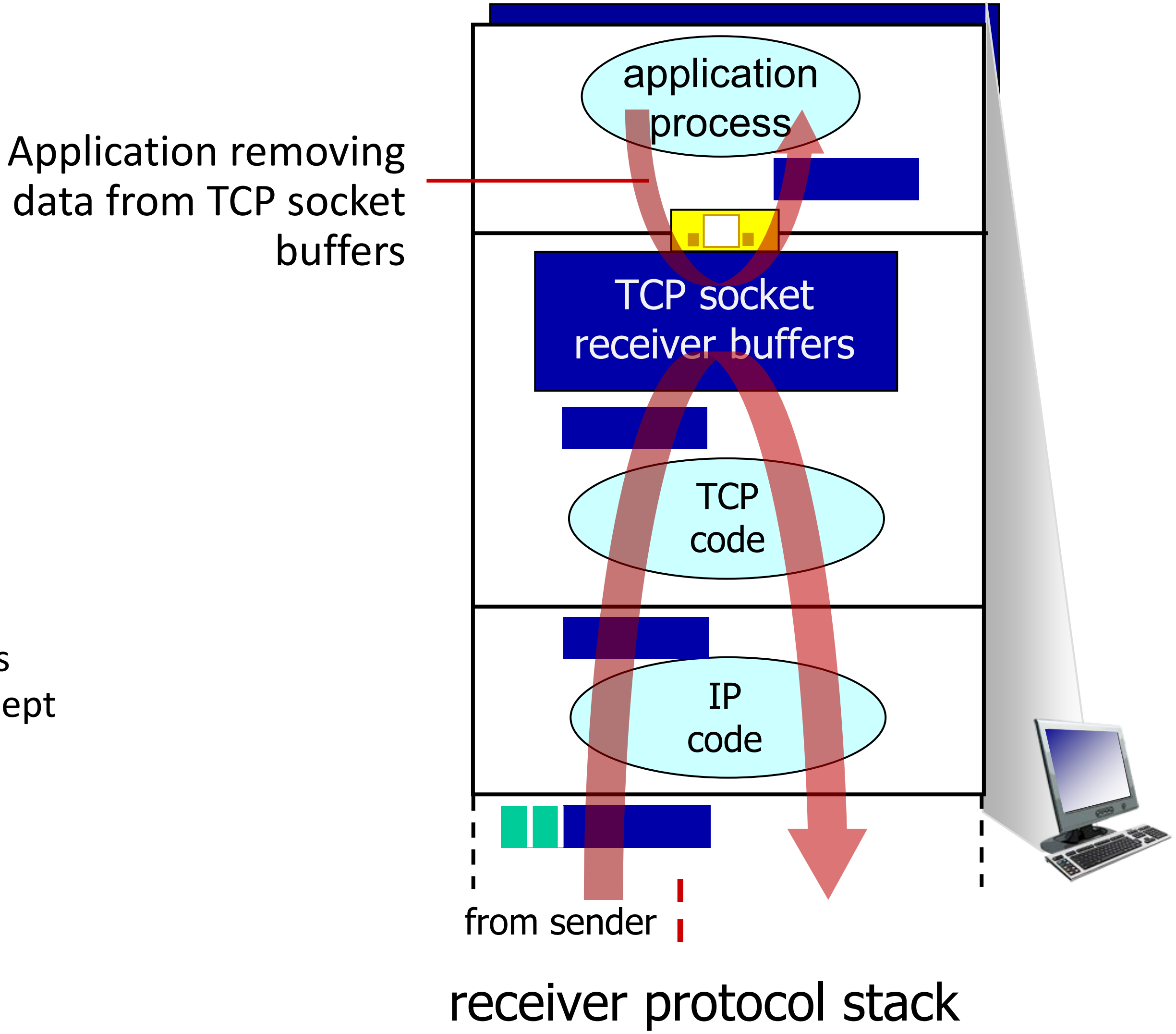


receiver protocol stack

TCP flow control

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



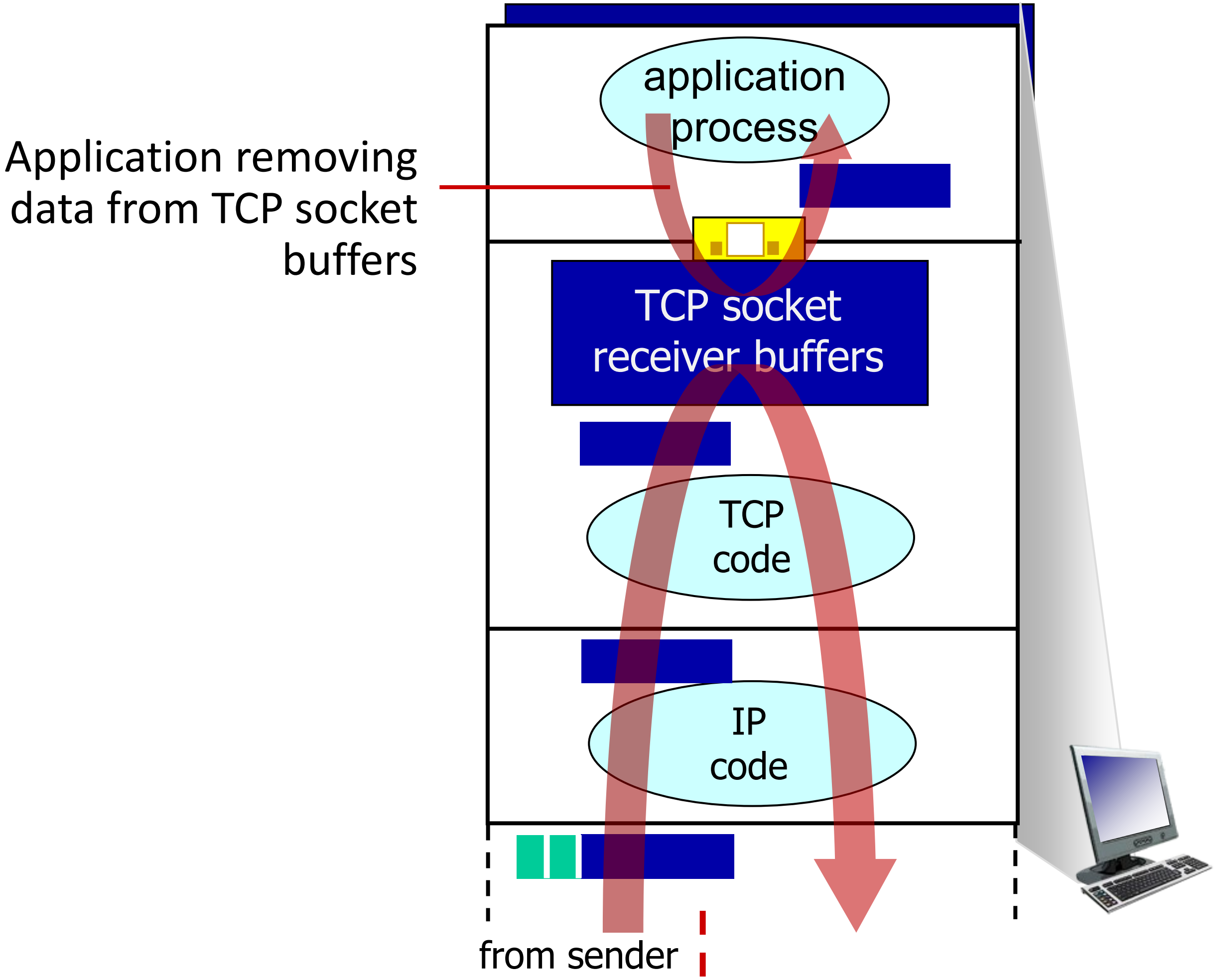


TCP flow control

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

flow control

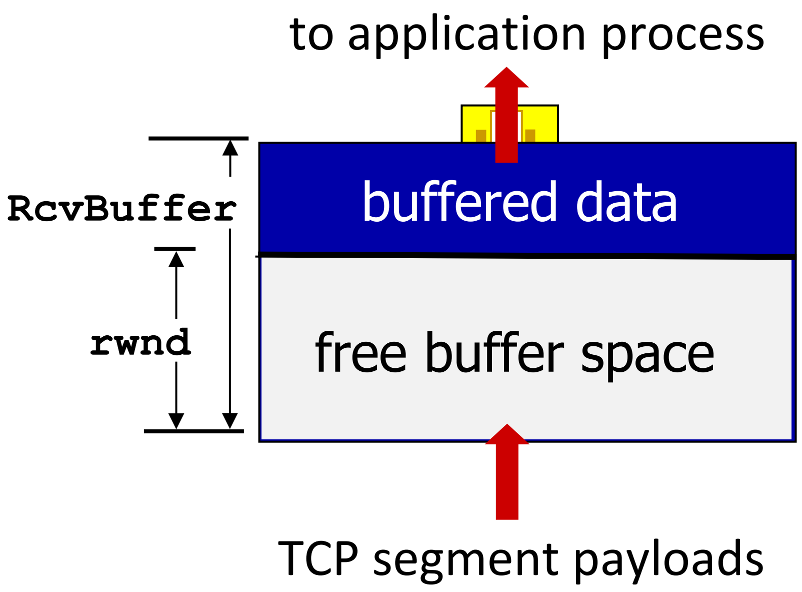
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast



receiver protocol stack

TCP flow control

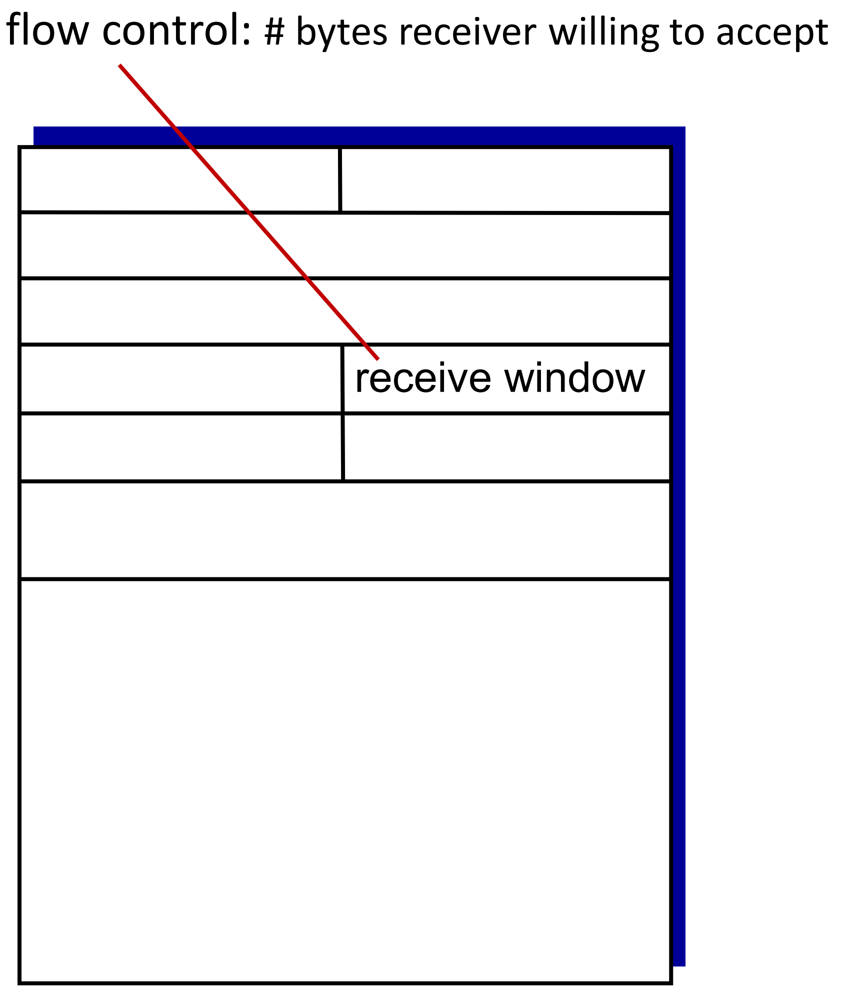
- TCP receiver "advertises" free buffer space in **rwnd** field in TCP header
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **RcvBuffer**
- sender limits amount of unACKed ("in-flight") data to received **rwnd**
- guarantees receive buffer will not overflow



TCP receiver-side buffering

TCP flow control

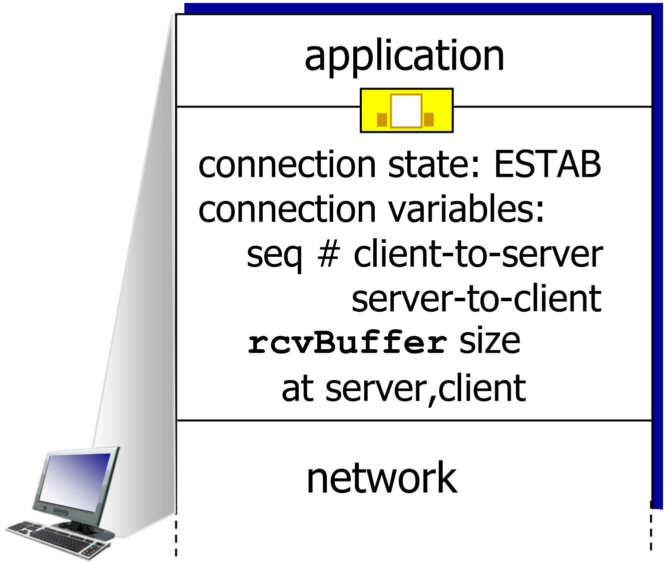
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems auto-adjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow



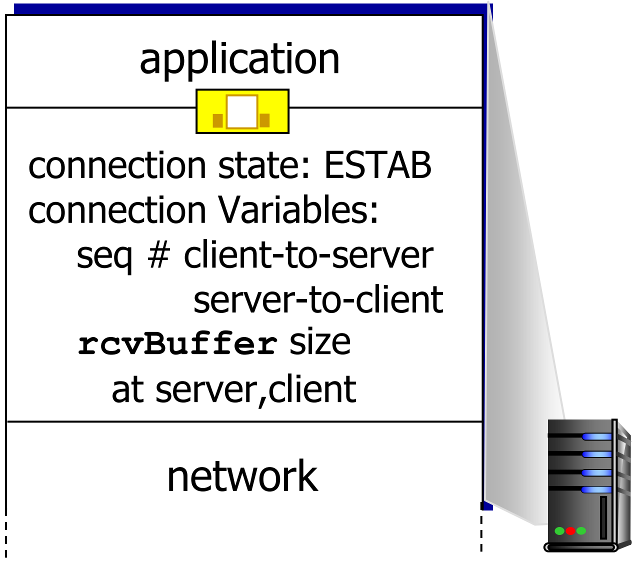
TCP segment format

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)



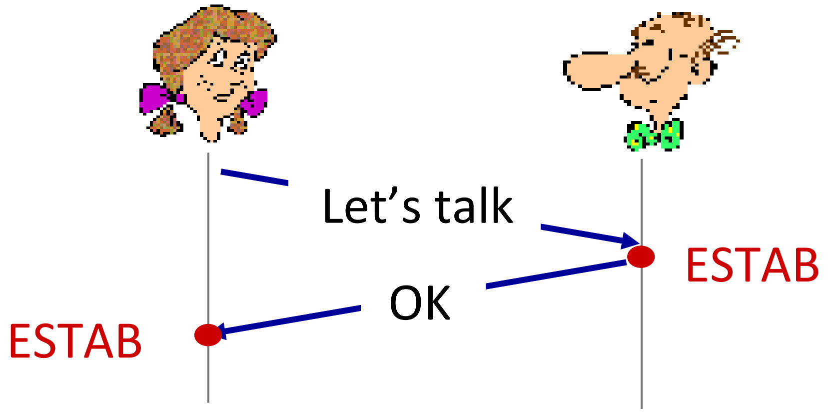

```
Socket clientSocket =
  newSocket("hostname","port number");
```

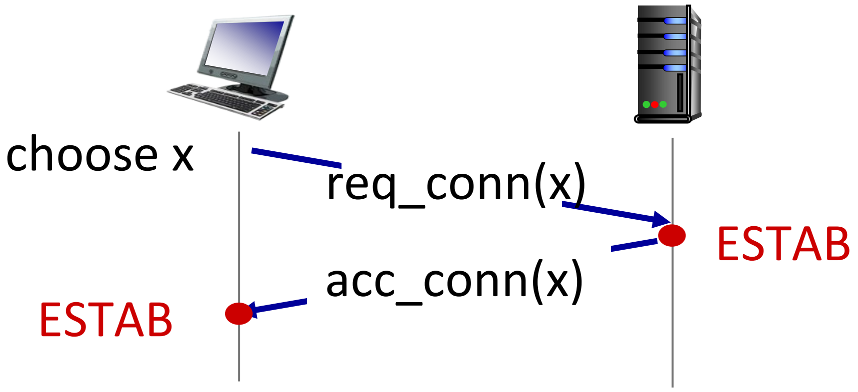



```
Socket connectionSocket =
  welcomeSocket.accept();
```

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

IMPORTANT

TCP 3-way handshake

Client state

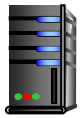
```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

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

Server state

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



LISTEN

choose init seq num, x
send TCP SYN msg

SYNSENT

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYN RCVD

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

ESTAB

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

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