CS 348

Computer Networks

Lec 17
TCP

Spring 2020 IIT Goa

Course Instructor: Dr. Neha Karanjkar

Note: These slides are adapted from "Computer Networking: A Top-down Approach" by Kurose & Ross, 7th ed

Guiding Questions

- Now that we understand how reliable data transfer can be achieved over an unreliable channel (using checksums, ACKs, timeouts etc.), how does TCP work?
- Which of these protocols does TCP employ? Go-Back-N? Selective-Repeat? How does TCP overcome some of the performance issues in these protocols?
- What does the TCP segment header contain?
- Why is TCP said to be "connection-oriented"?
- Why is a 3-way handshake necessary for setting up a TCP connection? How is a connection breakup performed?
- What is Maximum Segment Size (MSS) and Maximum Transmission Unit (MTU)? How do they differ?
- How is the timeout interval chosen in TCP for retransmission of lost packets?

Features of TCP (RFCs: 793,1122, 2018, 5681, 7323)

- **Point-to-point** (A TCP connection is always between a pair of nodes/processes. No multicasting).
- **Full-duplex** (each end-point of the connection can send as well as receive data)
- **ACKs can be piggy-backed** within a data packet. Example:
  - Say there is a connection between A and B, then the packet flowing from B to A can have data sent by B to A as well as ACK sent by B for some other data packet that arrived from A.

Features of TCP (RFCs: 793,1122, 2018, 5681, 7323)

- **ACKs can be piggy-backed** within a data packet...because of these

  fields in the TCP header:
  - 1-bit Flags (Such as ACK, RST, FIN)
  - sequence number (for DATA being sent)
  - acknowledgement number (for ACK being sent for received data). Only valid if ACK flag==1. Ignored if ACK flag==0.

TCP Header and Segment structure

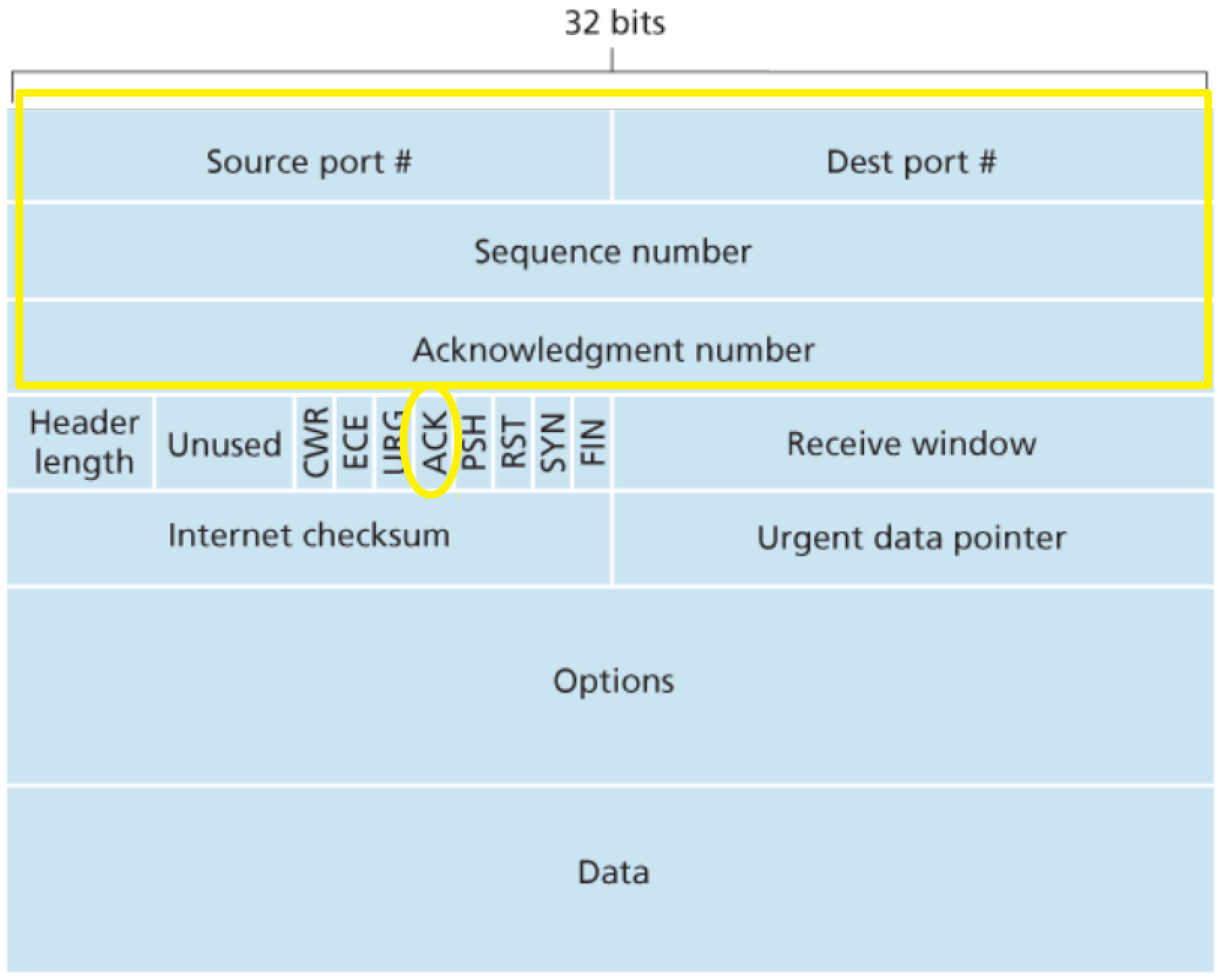


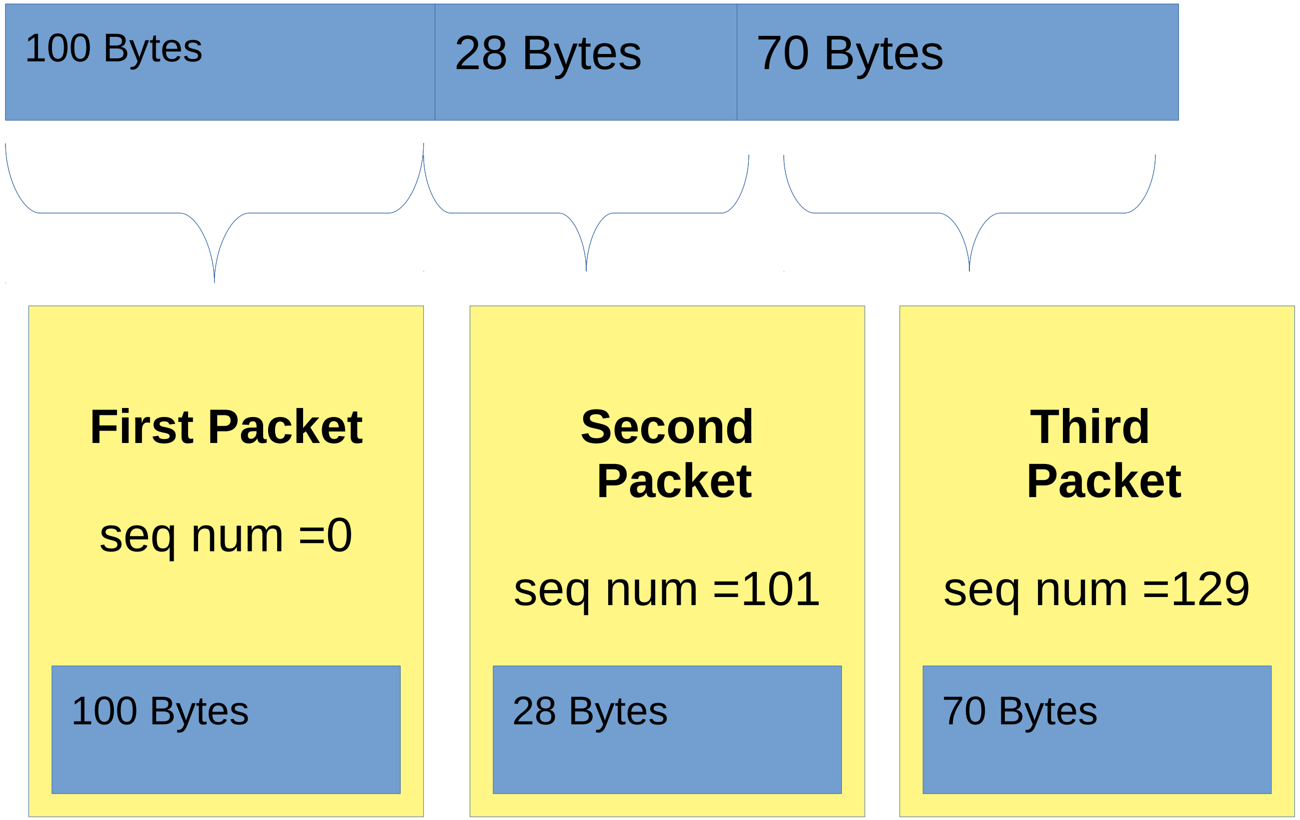
Figure 3.29 TCP segment structure

Sequence and ACK numbers

- **Do not** correspond to the number of packets sent/received!
- **Sequence number**: correponds to the number of the first Byte in the packet in a stream of bytes being sent
- **Ack number:** number of the next Byte expected in the next packet.

Sequence and ACK numbers

Stream of Bytes being sent from **A to B**



Sequence and ACK numbers

Stream of Bytes being sent from **A to B**

100 Bytes	28 Bytes	70 Bytes

Similarly, stream of Bytes being sent from **B to A**

10 Bytes	8 Bytes	71 Bytes

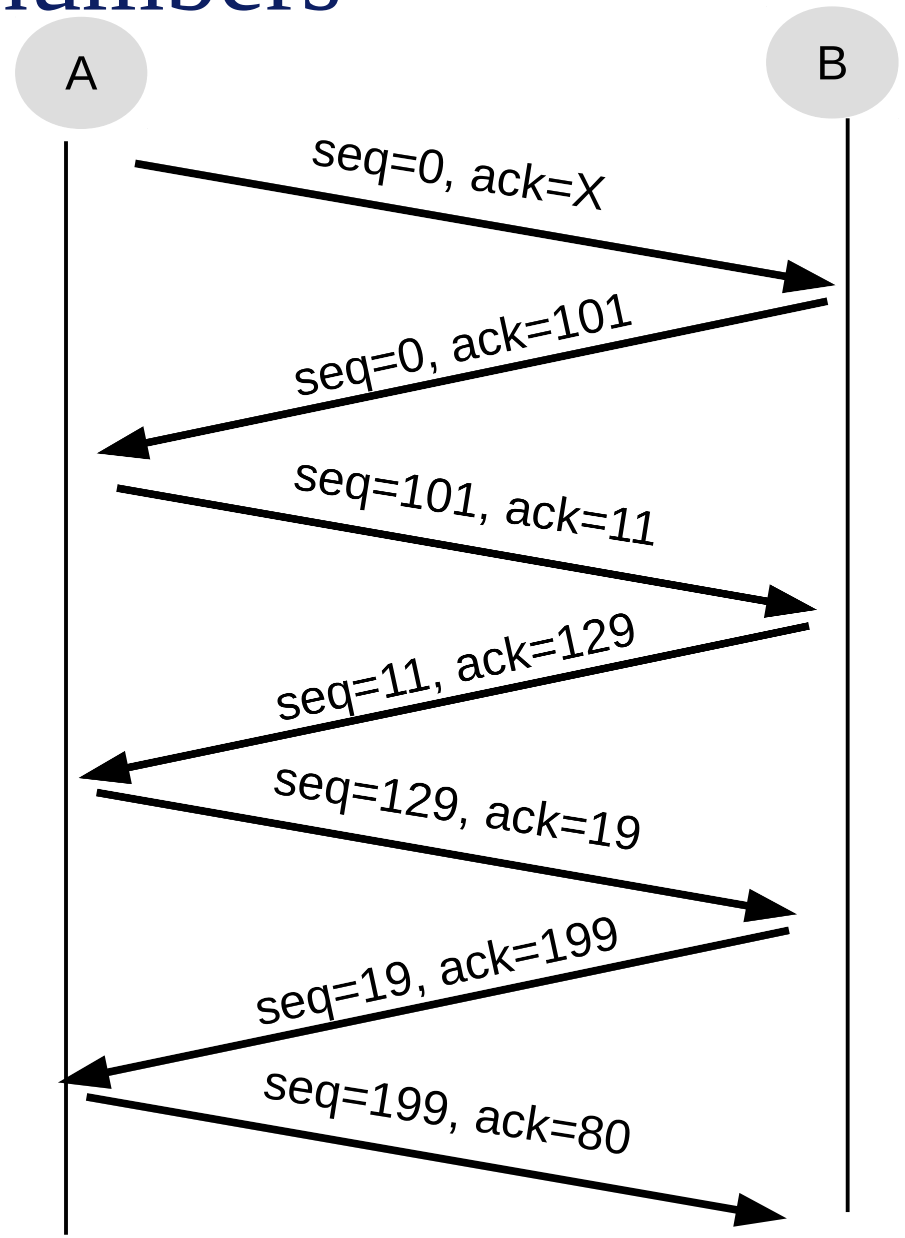
Sequence and ACK numbers

Stream of Bytes being sent from **A to B**

100 Bytes	28 Bytes	70 Bytes

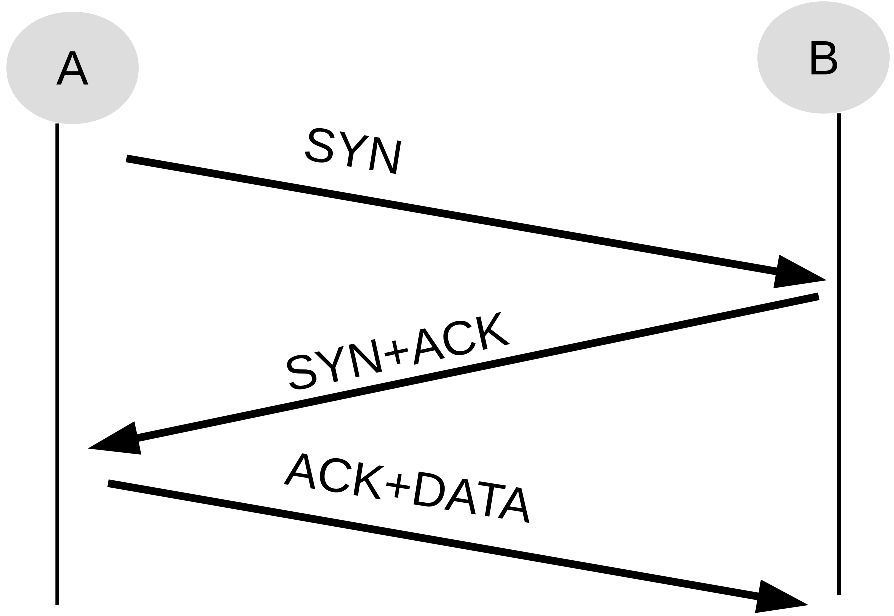
Similarly, stream of Bytes being sent from **B to A**

10 Bytes	8 Bytes	71 Bytes



Sequence and ACK numbers

- **However,** the Initial Sequence Number is not 0, but a randomly chosen 32-bit number.
  - **Why is it not 0?**
- At connection establishment stage, each side conveys their chosen ISN to the other side.---->This is one reason why a 3-way handshake is required)



Features of TCP

- **Point-to-point** (A TCP connection is always between a pair of nodes/processes. No multicasting).
- **Full-duplex** (each end-point of the connection can send as well as receive data)
- **ACKs can be piggy-backed** within a data packet. Seq and ack numbers correspond to the Byte number in the data stream.
- Seq and Ack numbers **do not start from 0**, but a randomly chosen Initial Sequence Number.
- A connection is established using a 3 -way handshake consisting of SYN, SYN+ACK and ACK messages
  - Why is a connection establishment phase necessary at all?
  - Why is a 3-way handshake required? Why can't it be a 2-way handshake?

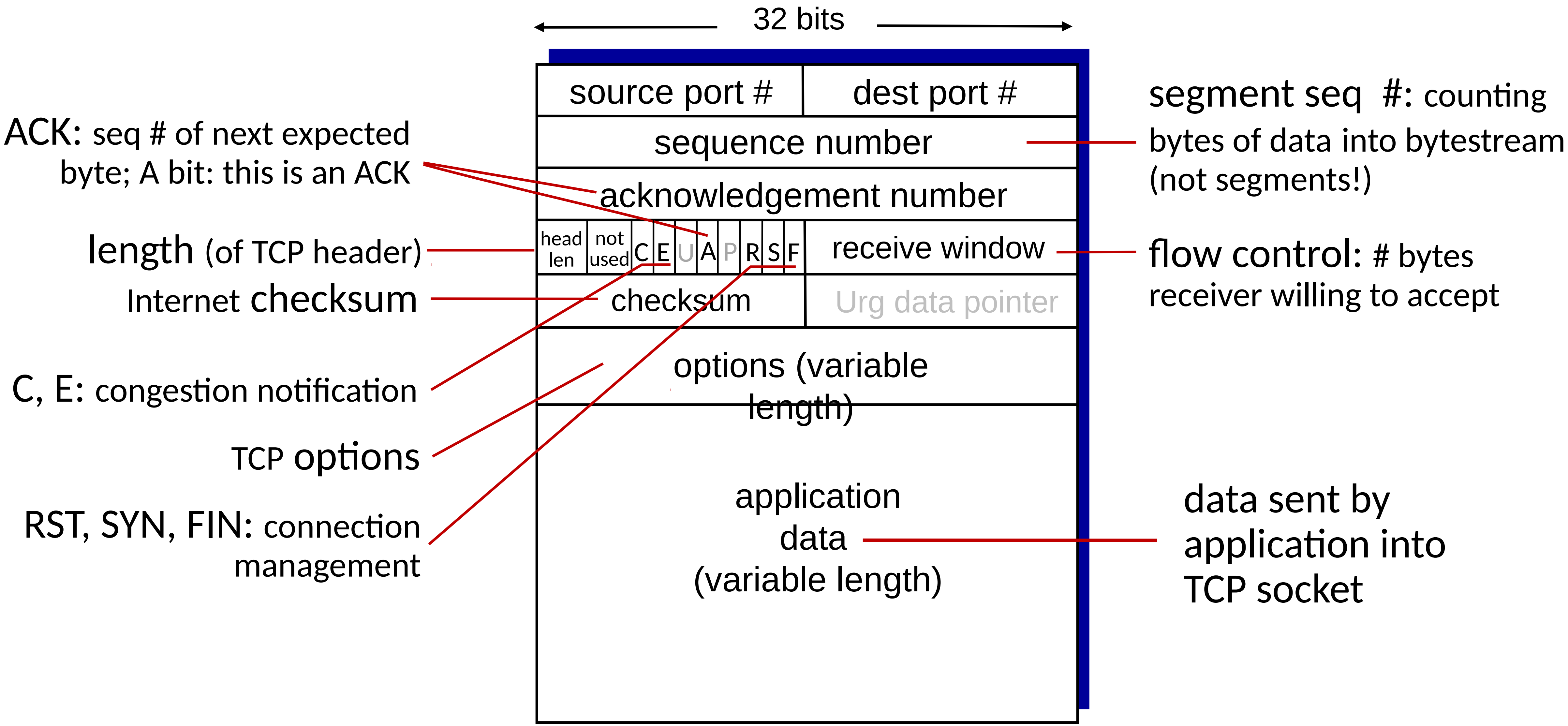
Features of TCP

- **Point-to-point** (A TCP connection is always between a pair of nodes/processes. No multicasting).
- **Full-duplex** (each end-point of the connection can send as well as receive data)
- **ACKs can be piggy-backed** within a data packet. Seq and ack numbers correspond to the Byte number in the data stream.
- Seq and Ack numbers **do not start from 0**, but a randomly chosen Initial Sequence Number.
- A connection is established using a 3 -way handshake consisting of SYN, SYN+ACK and ACK messages
  - Why is a connection establishment phase necessary at all?
  - Why is a 3-way handshake required? Why can’t it be a 2-way handshake?

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

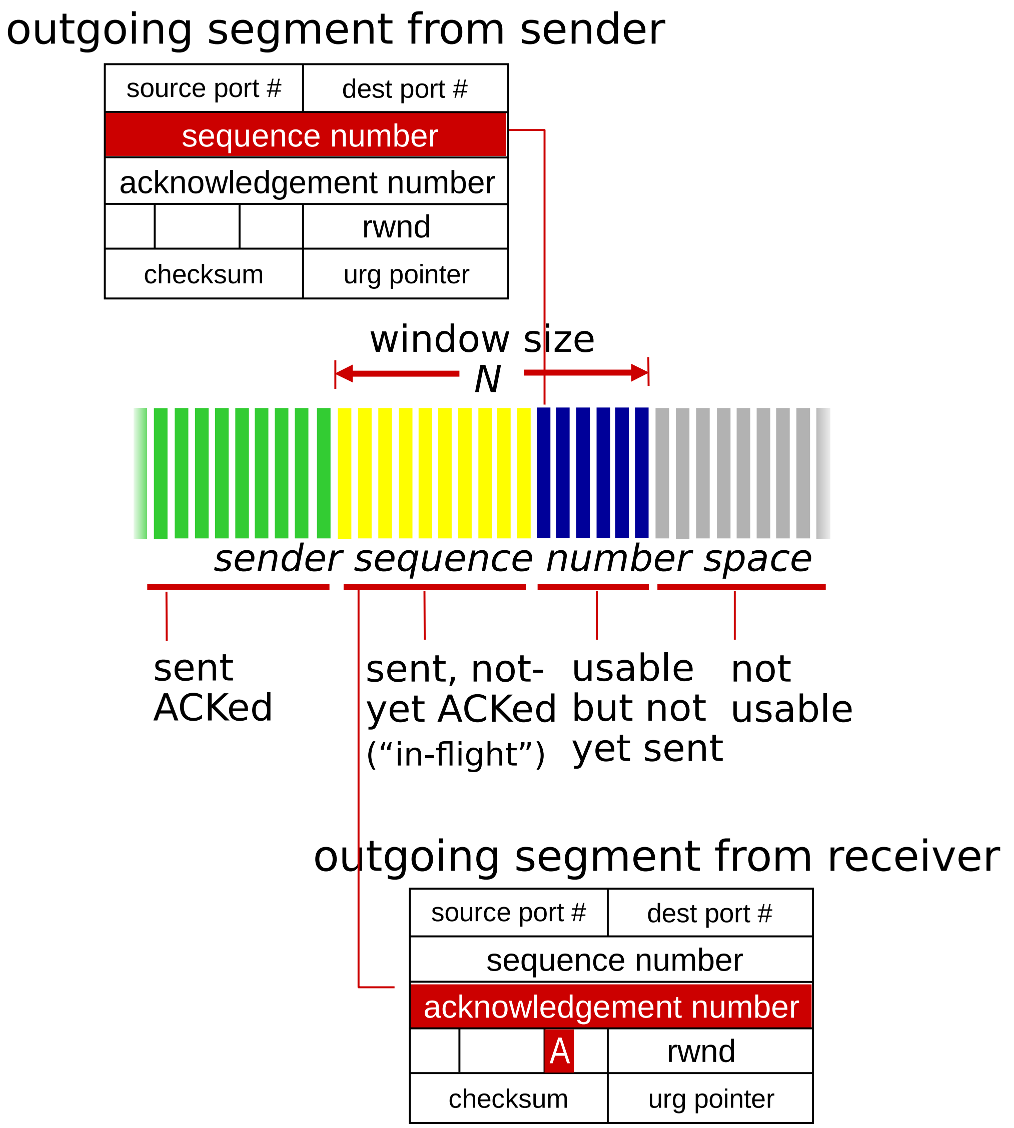


TCP sequence numbers, ACKs

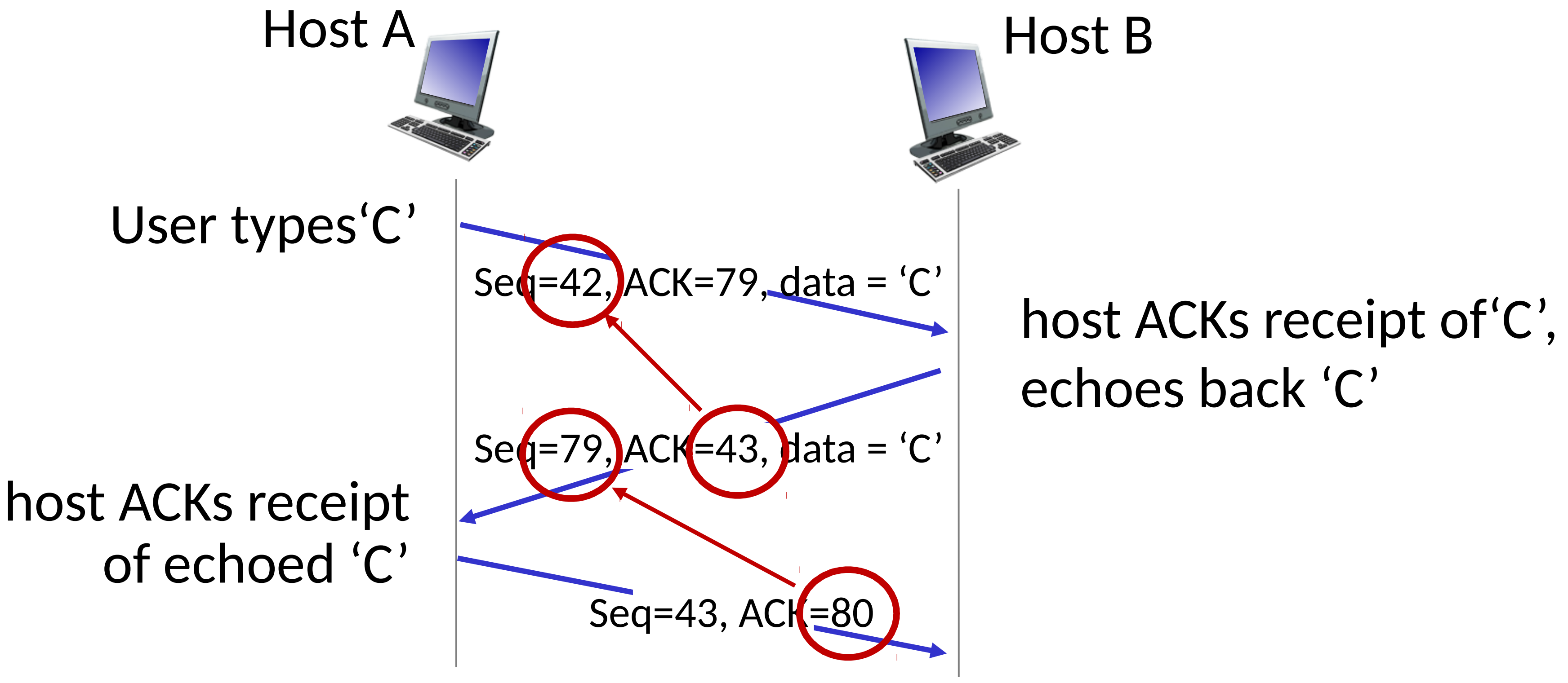
Sequence numbers:
- byte stream "number" of first byte in segment's data

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 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

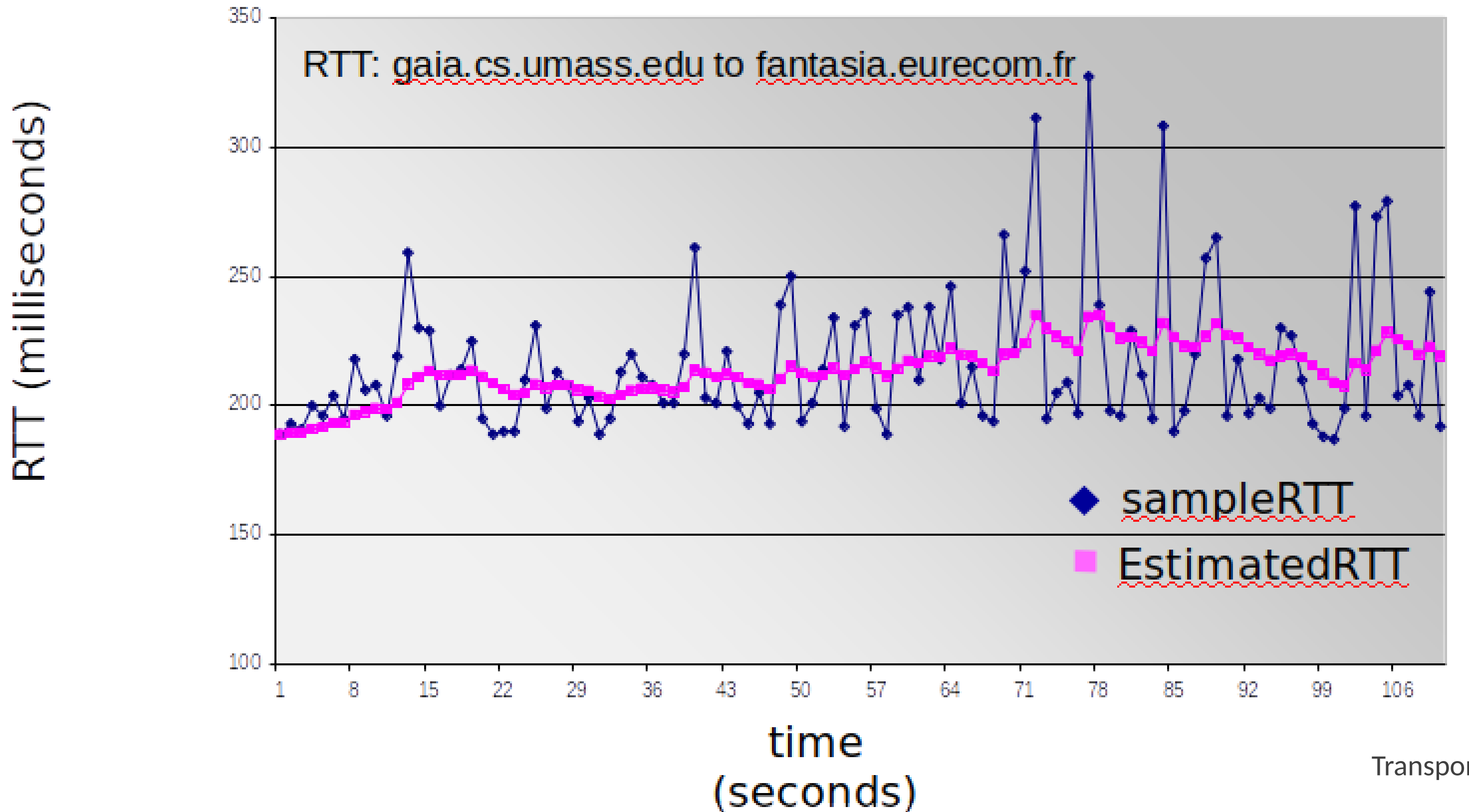
Q: how to estimate RTT?

- `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

```
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

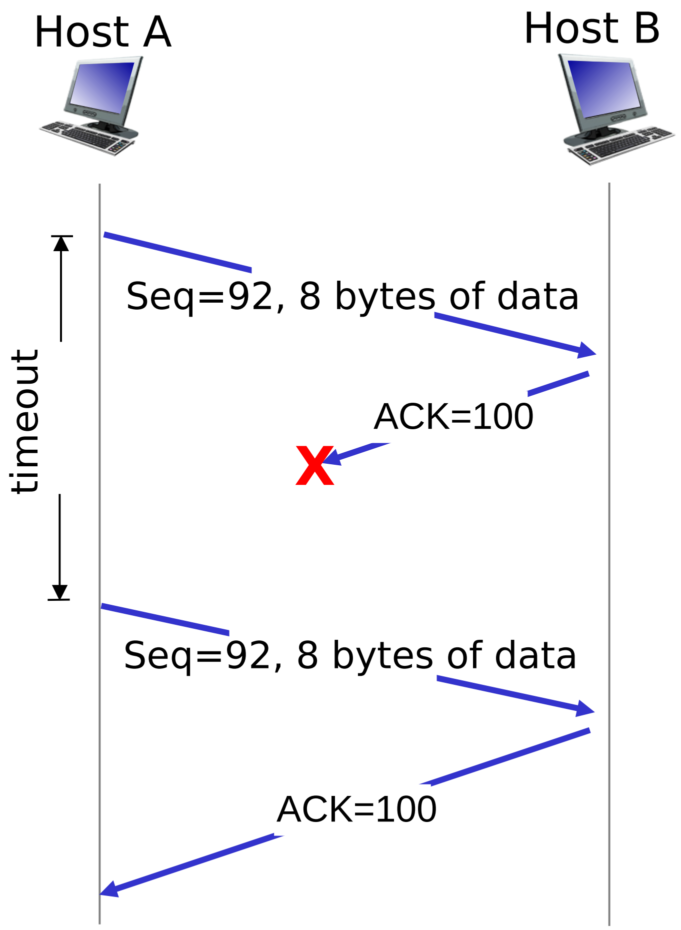
event: ACK received

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

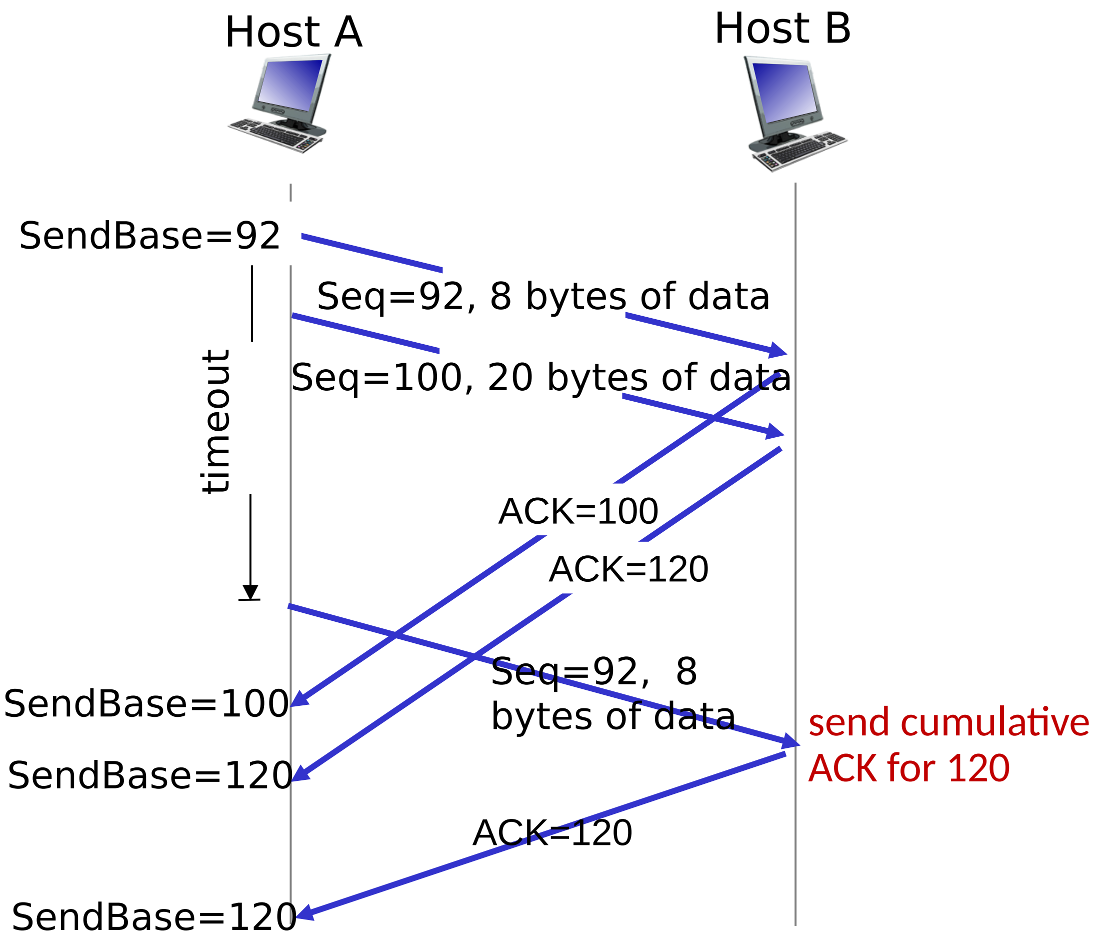
TCP Receiver: ACK generation [RFC 5681]

Event at receiver	*TCP receiver action*

TCP: retransmission scenarios

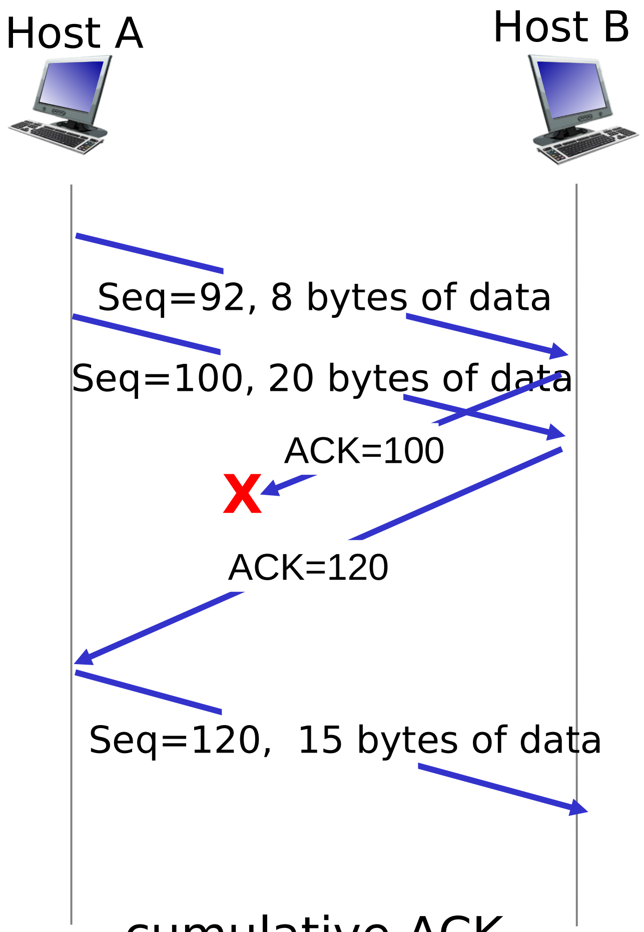


lost ACK scenario



premature timeout

TCP: retransmission scenarios



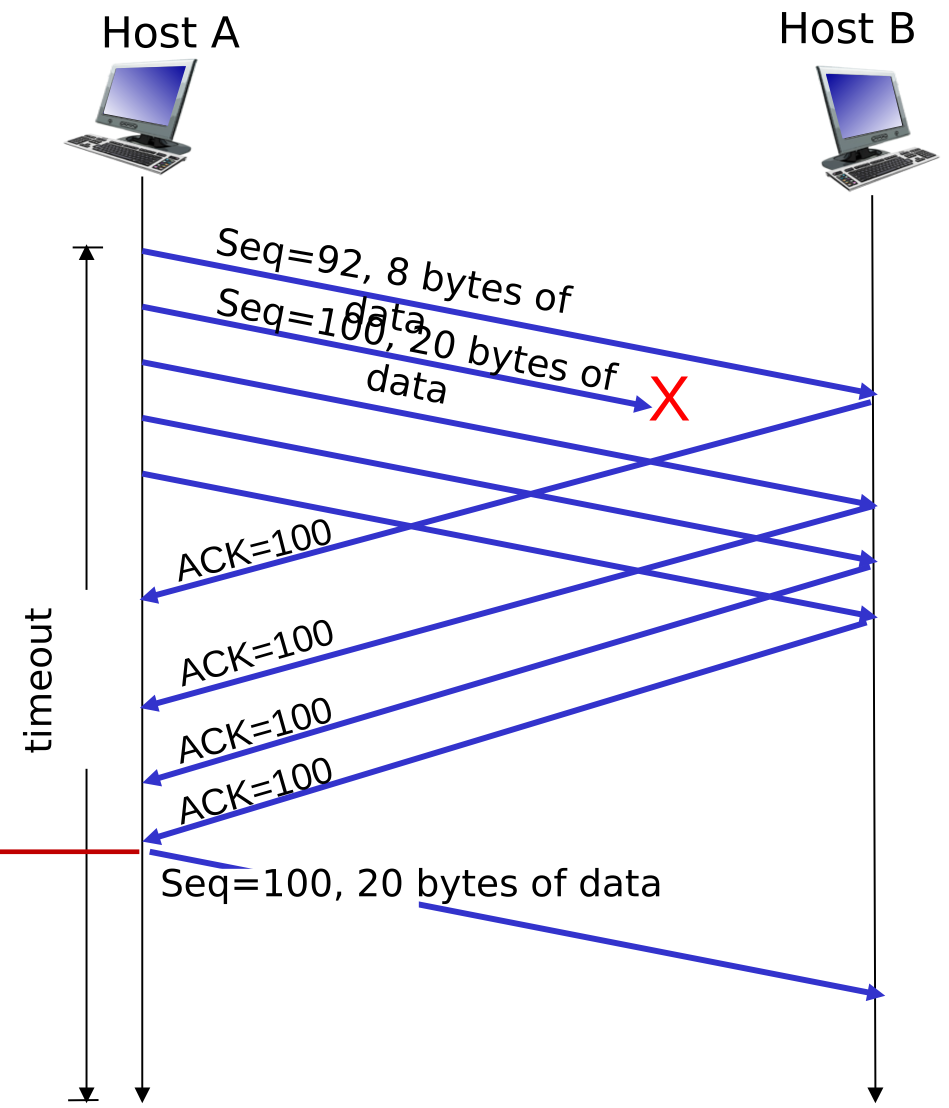
cumulative ACK covers for earlier lost ACK

TCP 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



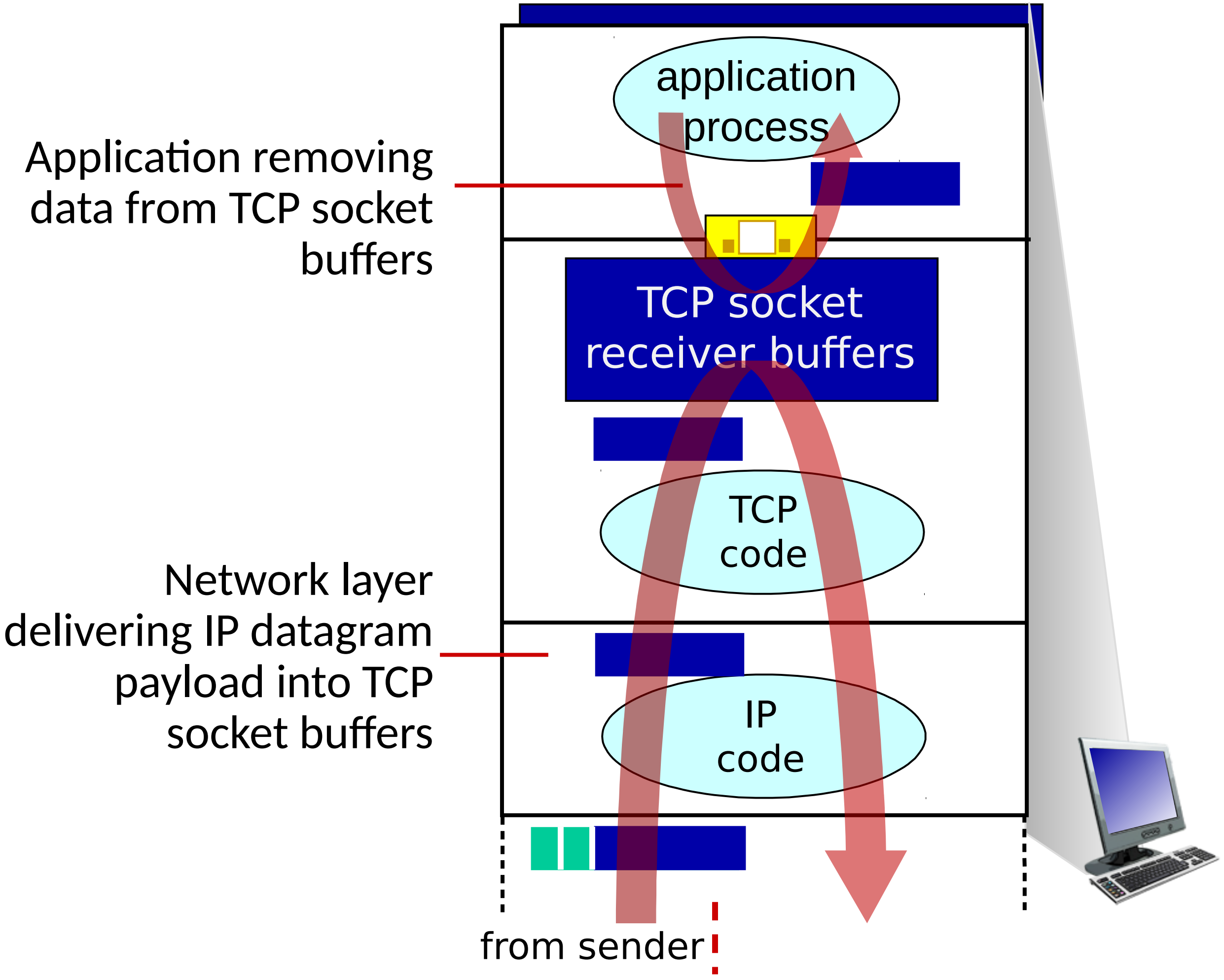
Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

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

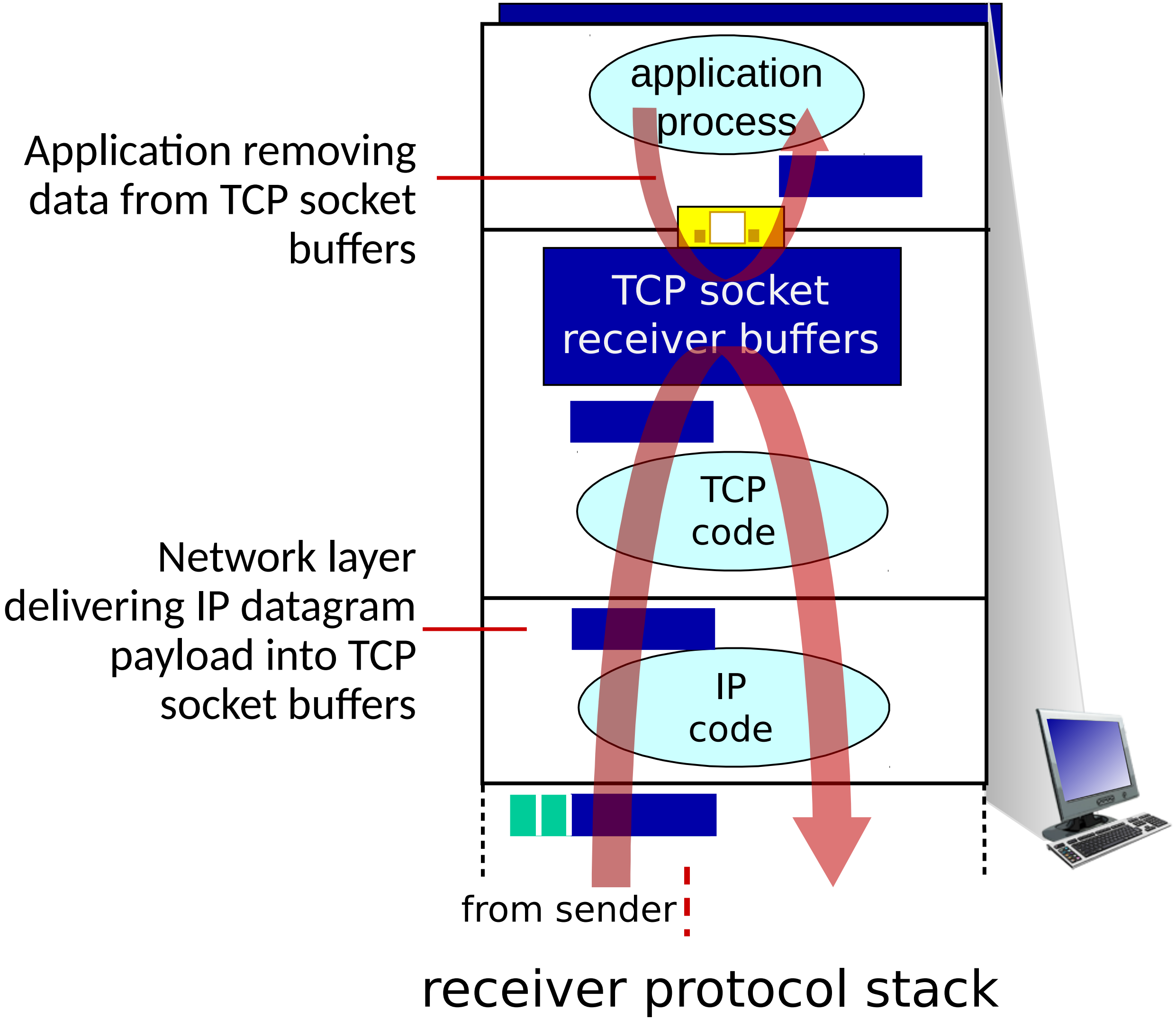
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

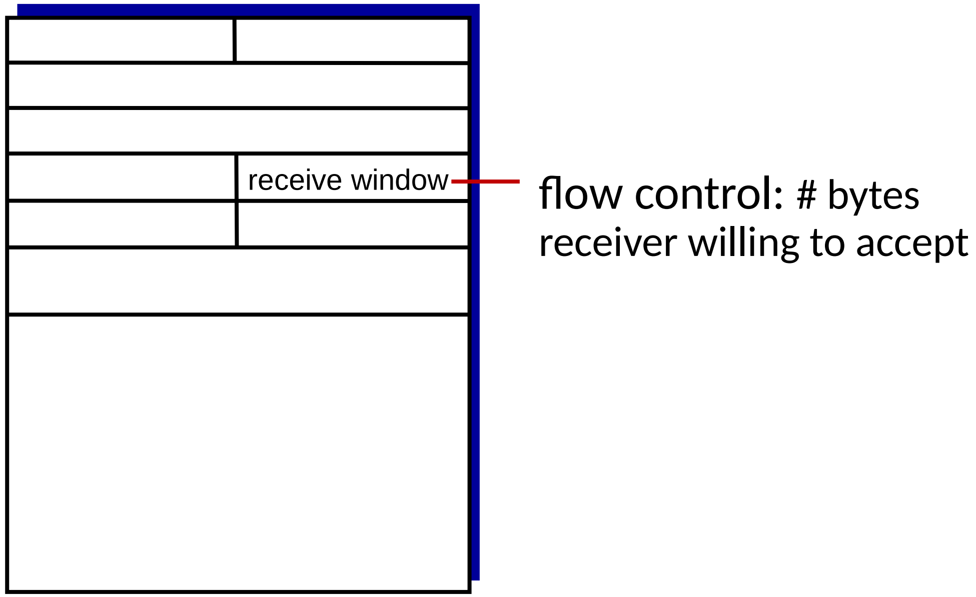
TCP flow control

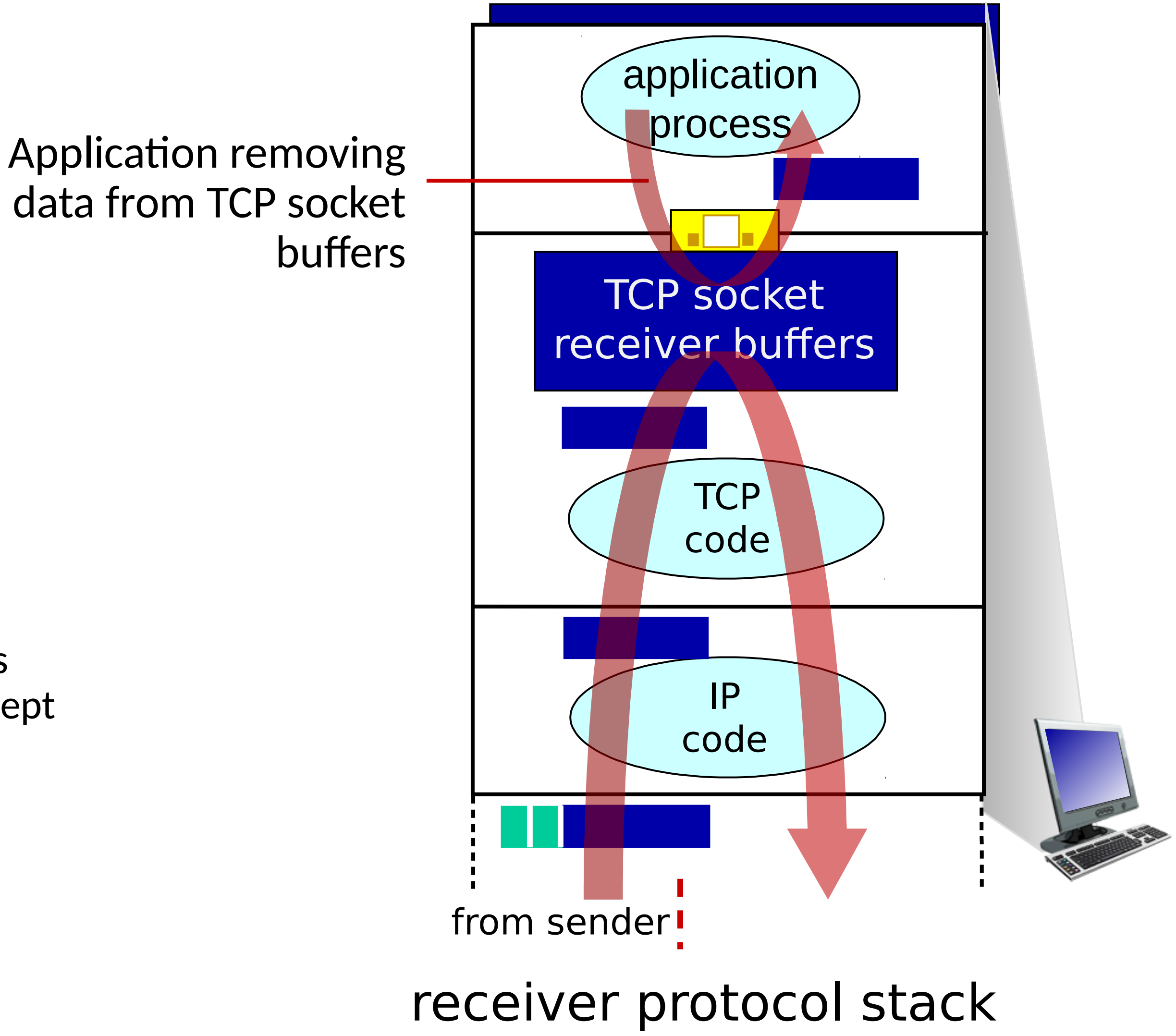
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



TCP flow control

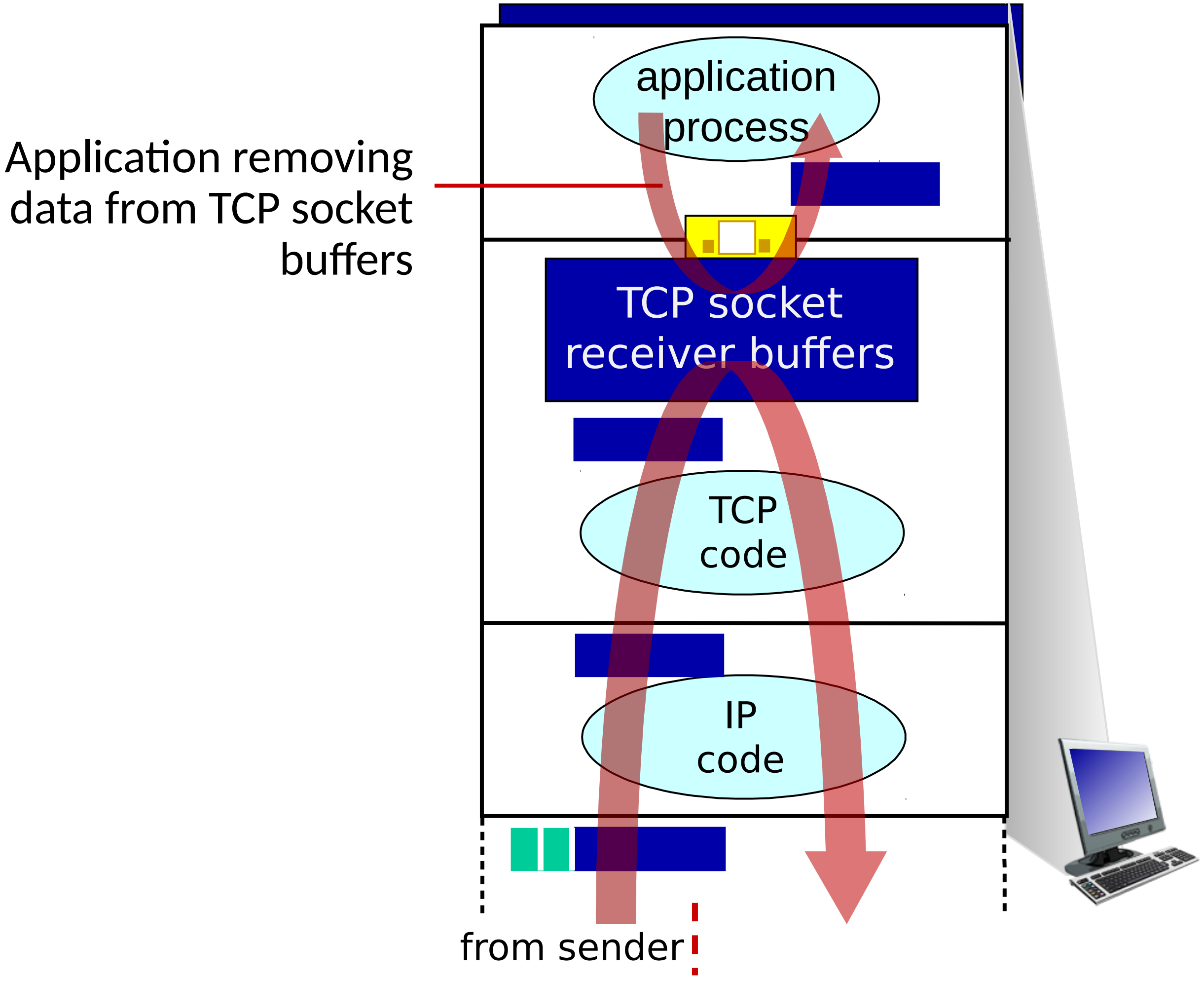
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?





TCP flow control

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



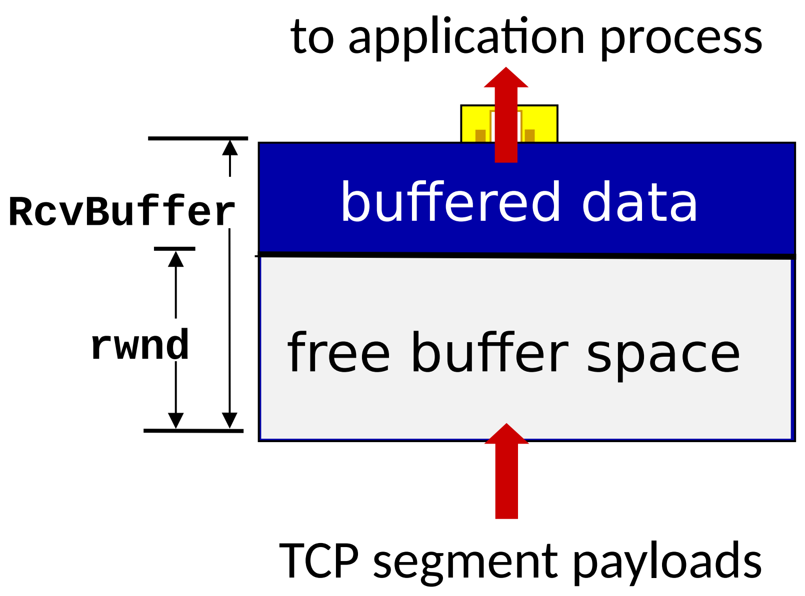
receiver protocol stack

flow control

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

TCP flow control

- 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 autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

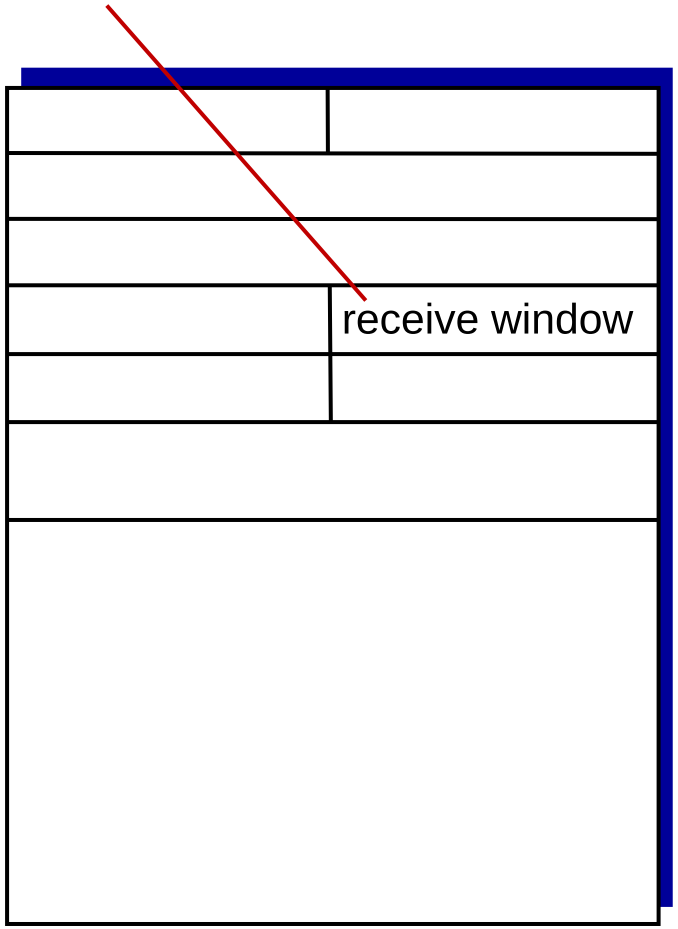


TCP receiver-side buffering

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 autoadjust **`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

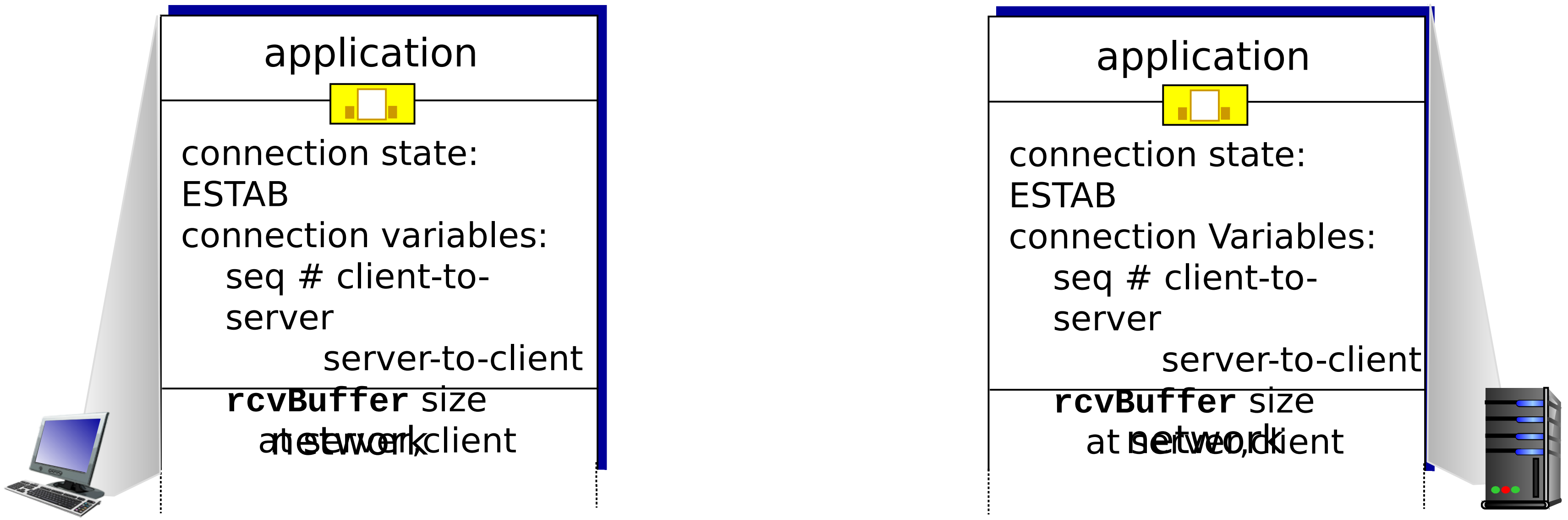


TCP segment format

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)

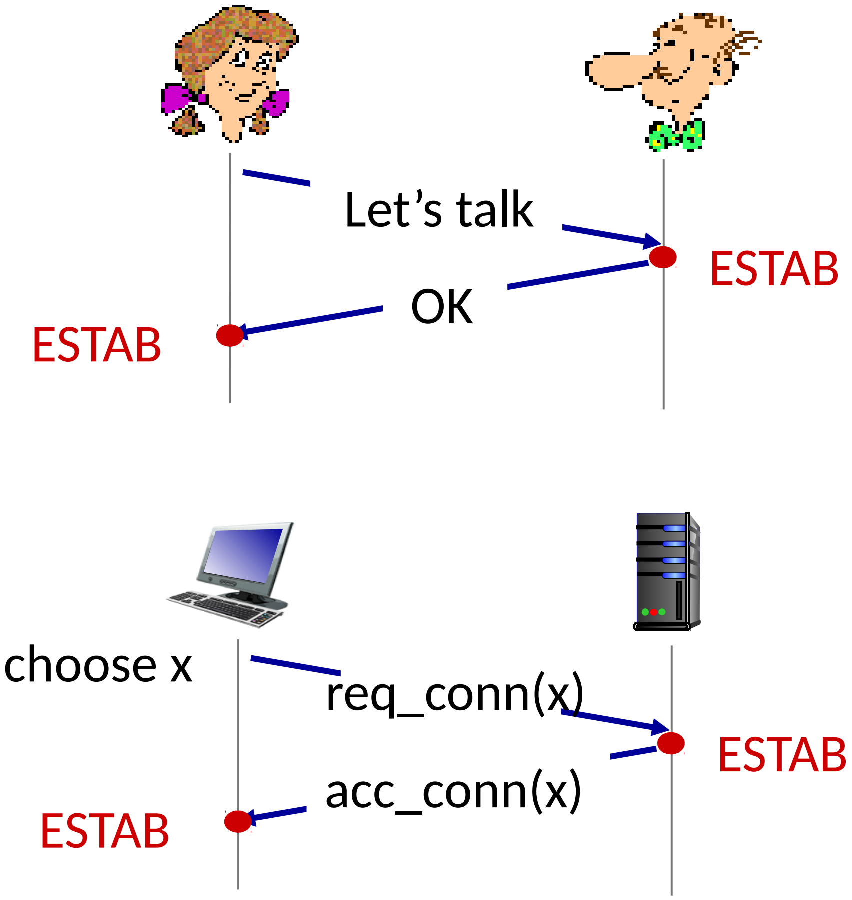



```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Agreeing to establish a connection

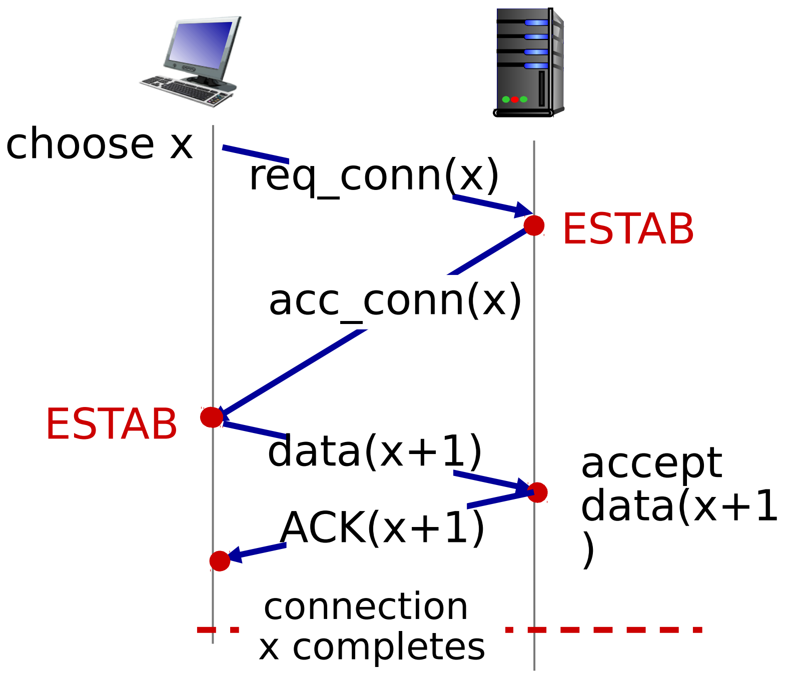
2-way handshake:



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

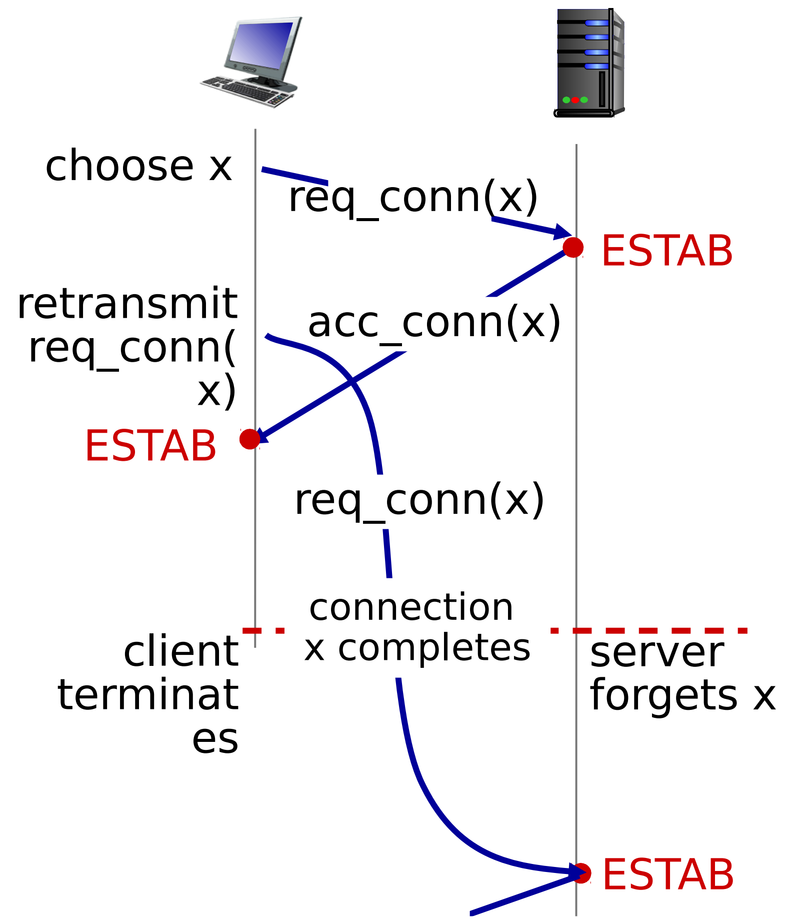
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can't "see" other side

2-way handshake scenarios



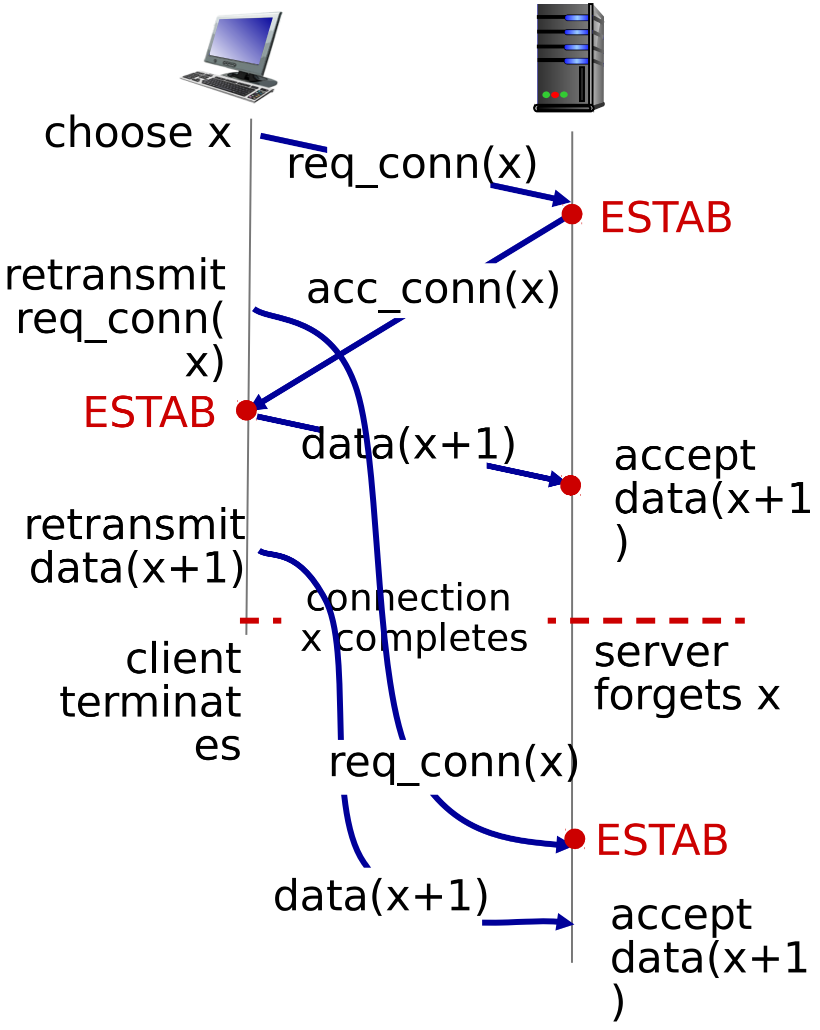
No problem!

2-way handshake scenarios



Problem: half open connection! (no client)

2-way handshake scenarios



Problem: dup data accepted!

TCP 3-way handshake

Client state

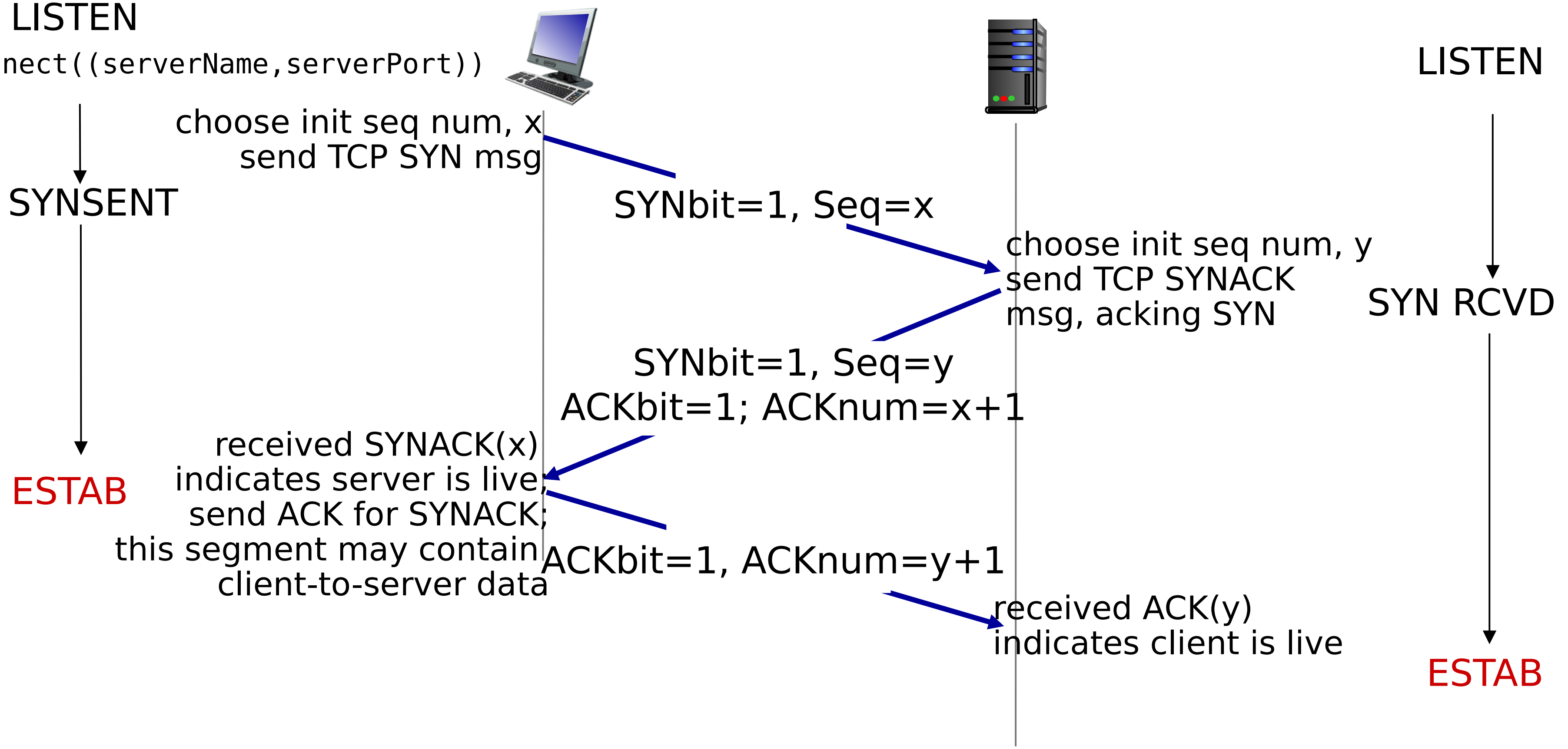
```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

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

Server state

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



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

References and Reading Assignment

- **Kurose and Ross 6th ed or 7th ed:** Section 3.5

So far...

- Structure and Physical components of the Internet
- Design of the Internet: Layering and Encapsulation
- The Applications Layer:
  - Sockets Interface
  - The Web and HTTP
  - DNS
- **The Transport Layer: how it works**
  - **Basic services, UDP**
  - **Principles of Reliable Data Transfer (rdt 3.0 etc)**
  - **Pipelined data transfer (Sliding window protocols)**
  - **TCP details**
  - **Congestion and Flow control**