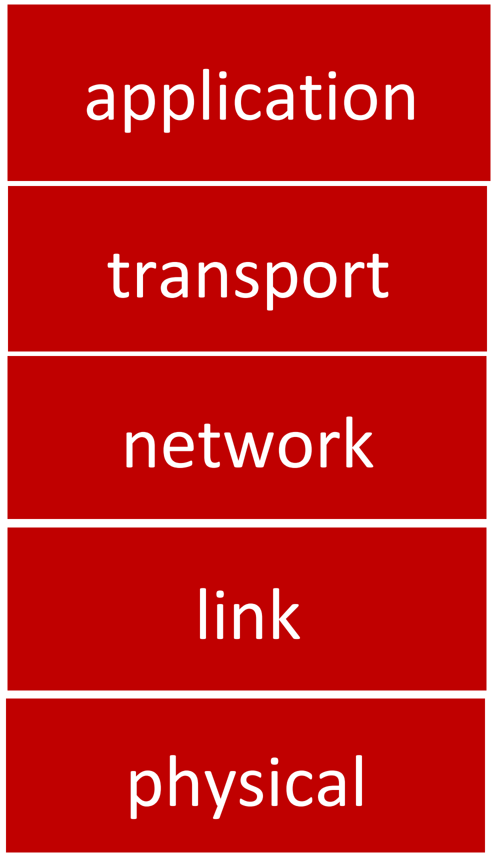
# Transport layer: overview

- between application and network layers

- provides communication services to the application processes

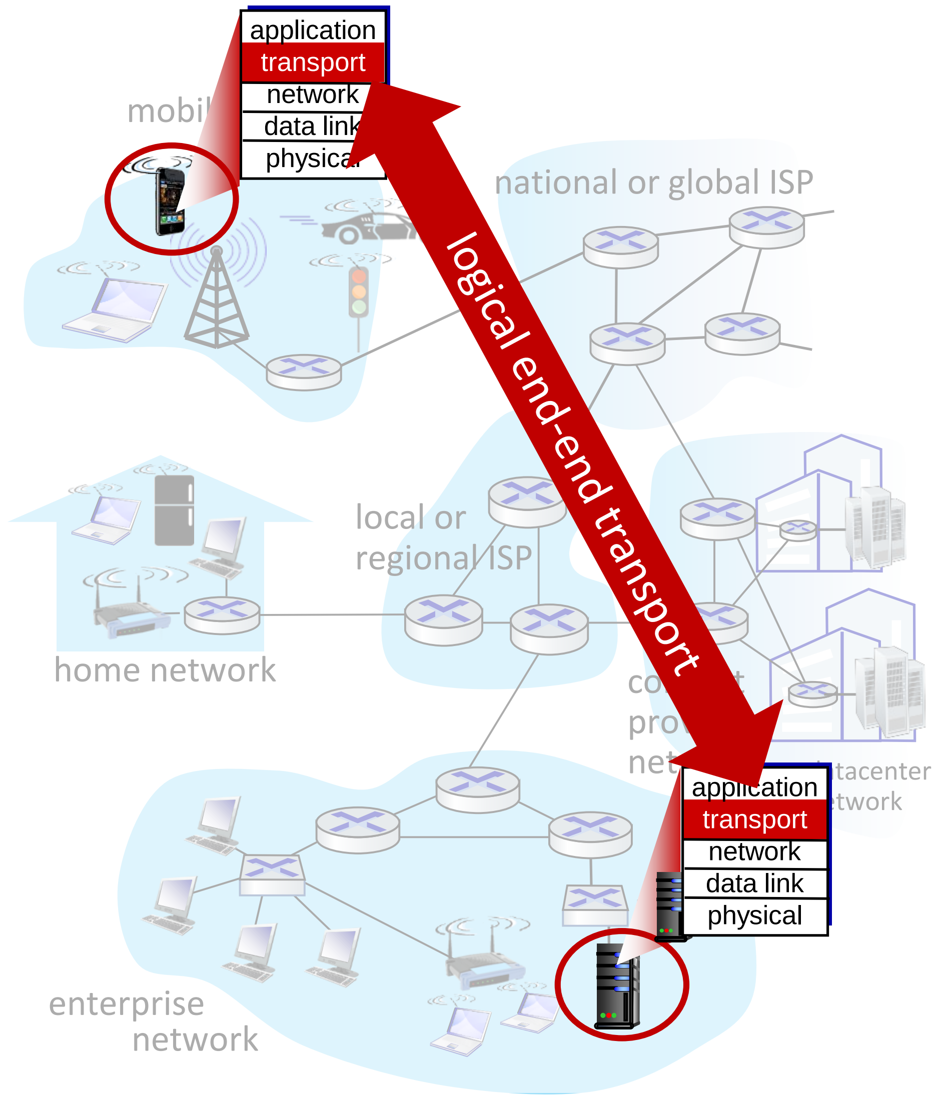
- flow:

- Transport services and protocols
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality



# Transport services and protocols

- a TL protocol provides *logical communication* between application processes running on different hosts *(as if hosts running the processes were directly connected: in reality not directly, via routers and links)*
- physical infrastructure between hosts is not a concern for application processes
- TL protocols are implemented in the end systems or hosts (not in routers)
- transport protocols actions in end systems:
  - sender: breaks application messages into *segments (data chunk+header)*, passes to network layer
  - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
  - TCP, UDP



# Chapter 3: roadmap

- Transport services and protocols
- Multiplexing and demultiplexing *(extending the host-to-host delivery service to a process-to-process delivery service, host-to-host delivery is related with network layer)*
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality

## Multiplexing/demultiplexing

Gather data, encapsulate with header (segments), and pass to NL

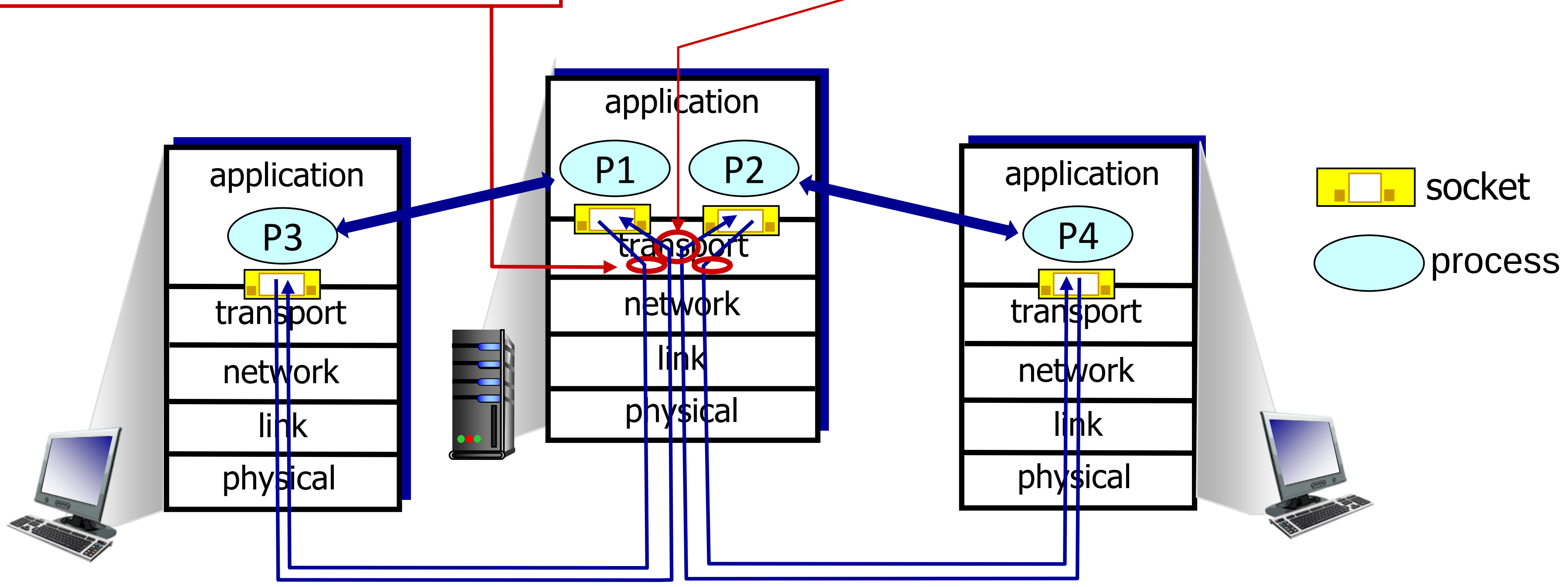
*multiplexing as sender:*

handle data from multiple sockets, add transport header (later used for demultiplexing)

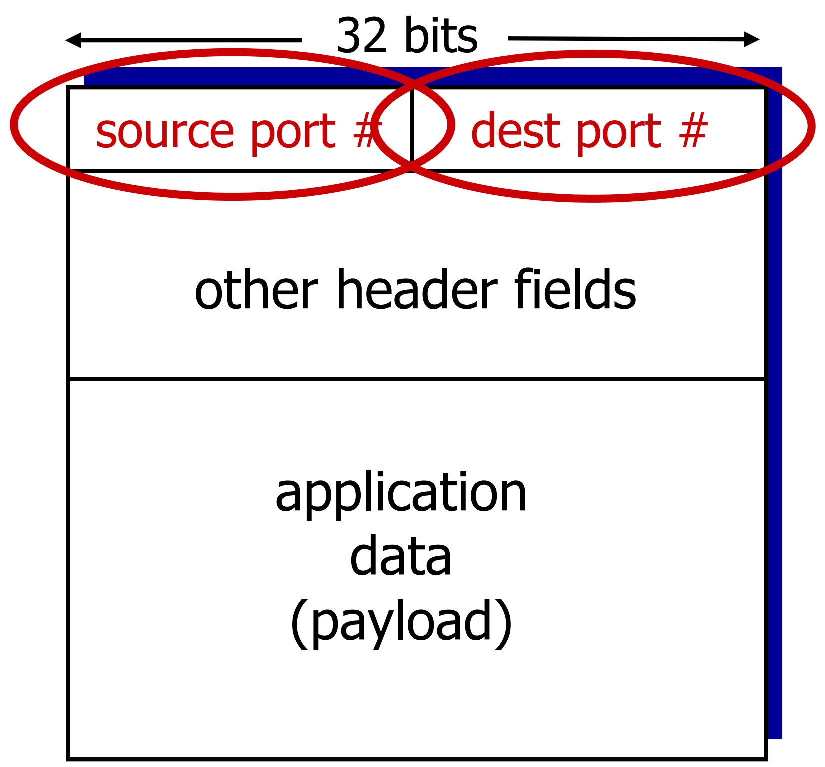
Delivering the data to the correct socket

*demultiplexing as receiver:*

use header info to deliver received segments to correct socket



# How demultiplexing works



TCP/UDP segment format

- Host receives IP datagrams
- In each datagram: src IP addr, dest IP addr (datagram is related to NL)
- In each datagram: one TL segment, in each segment: src and dest port num
- Receiving host uses IP addr & port num to direct segments to appropriate socket

# Connectionless demultiplexing: an example

* UDP socket: TL assigns a port num to the socket on the client side (1024:65535)
* or we can associate a specific num via bind() method

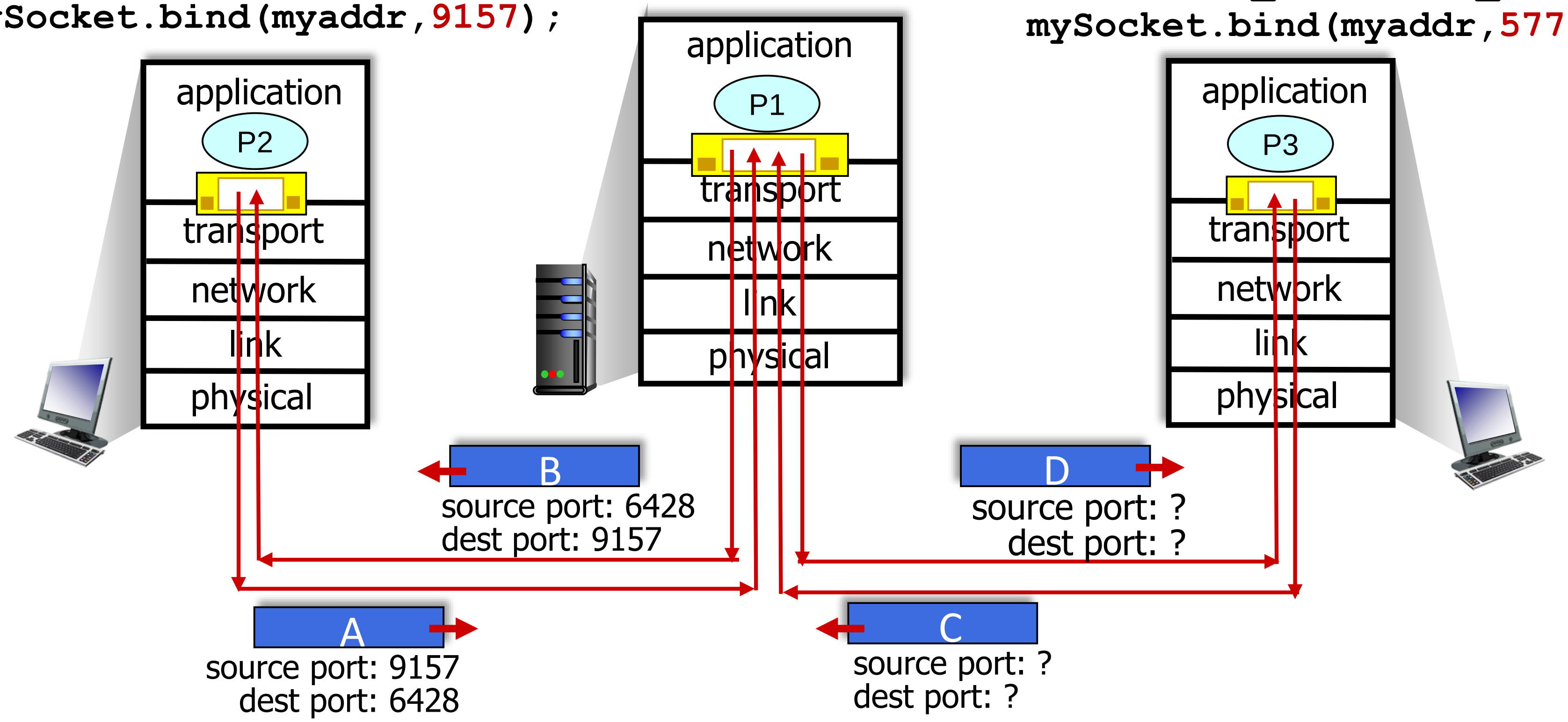
* for the server side we have to assign port num

* a process in A sends application data to a process B
* TL of A creates a segment (app data+src port+dest port)
* TL passes the segment to NL
* NL encapsulates the segment in an IP datagram and delivers it to the B.
* TL at B examines the dest port number and delivers it to its socket
* src port is part of a "return address"

```
mySocket =
 socket(AF_INET,SOCK_DGRAM)
mySocket.bind(myaddr,6428);
```

```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,9157);
```

```
mySocket =
 socket(AF_INET,SOCK_STREAM)
mySocket.bind(myaddr,5775);
```



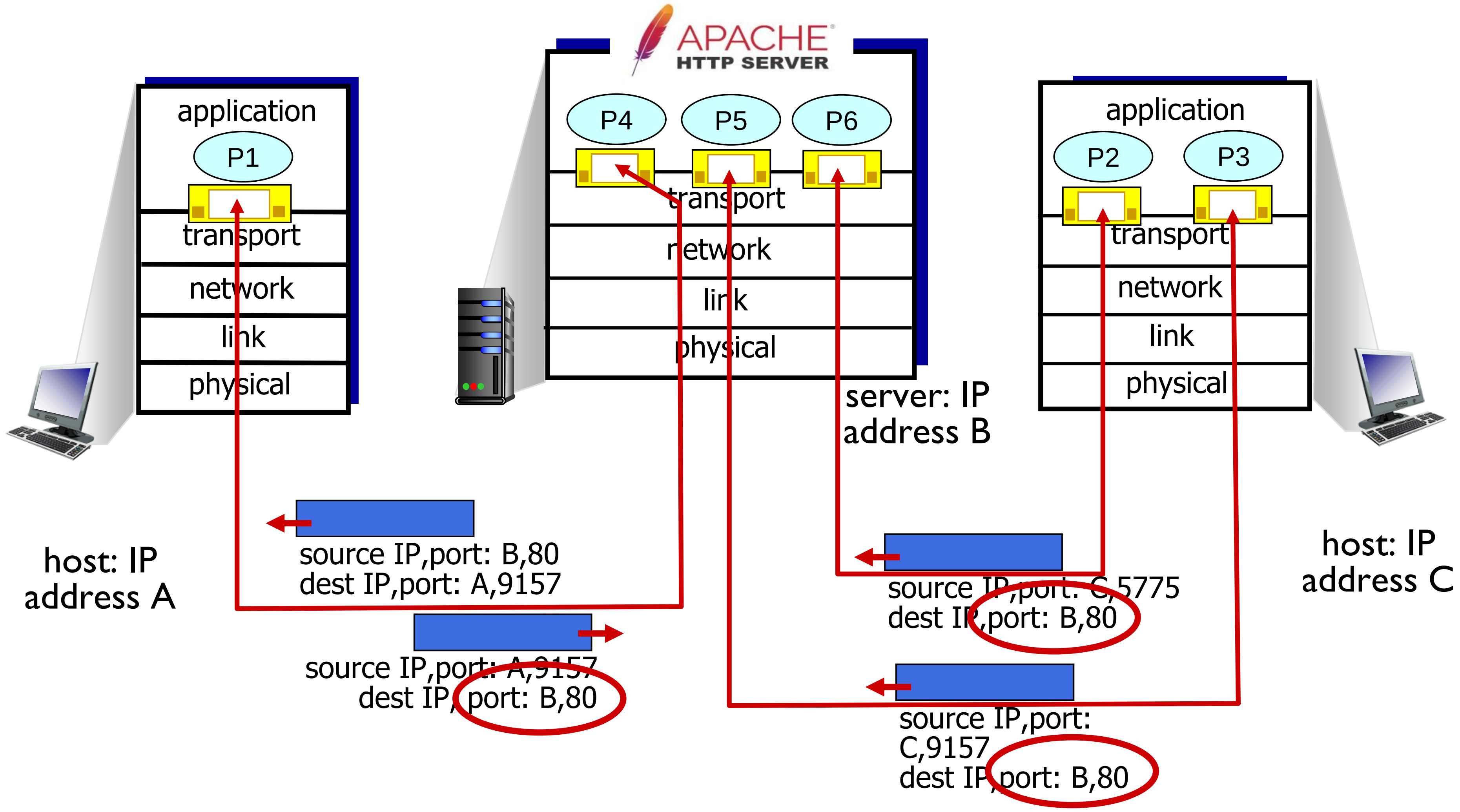
# Connection-oriented demultiplexing: example

MAIN DIFF
* If two UDP segments have different src IP addresses and/or different src port num with the same dest IP addr and dest port num, this two segments will be directed to the same dest process via the same dest socket

* For TCP, this two segments will be directed (multiplexed) to two different sockets (conn establishment between processes)

* UDP demultiplexing uses dest port num + IP addr

* TCP demultiplexing uses src and dest IP addr and port nums



# Chapter 3: roadmap

- Transport services and protocols
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP
- TCP congestion control
- Evolution of transport-layer functionality

## UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol, does as little as a transport protocol can do
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

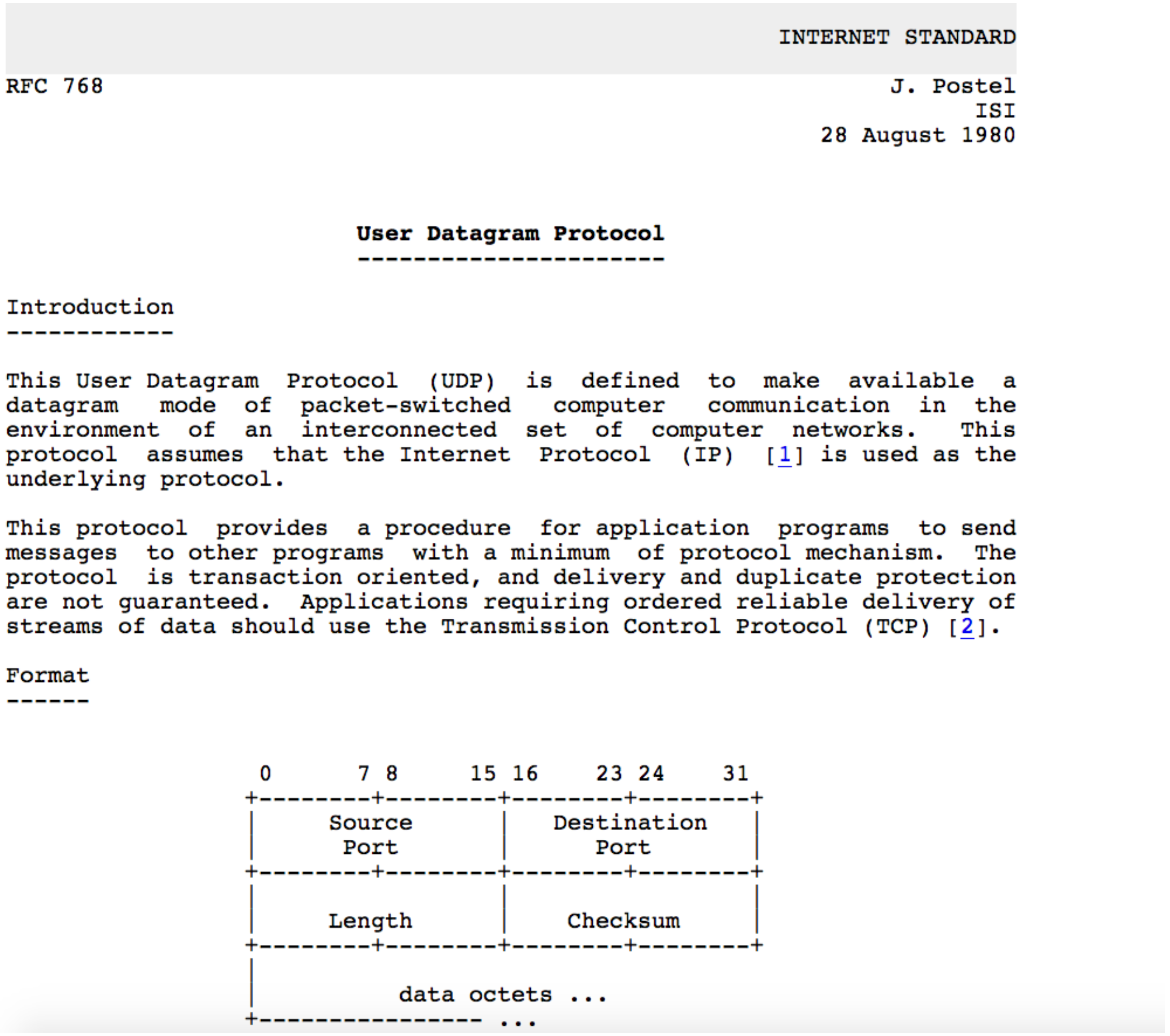
### Why is there a UDP?

- needs and application's requirements are important (making a choice)
- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver; UDP can support many more active clients
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

## UDP: User Datagram Protocol

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - HTTP/3
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

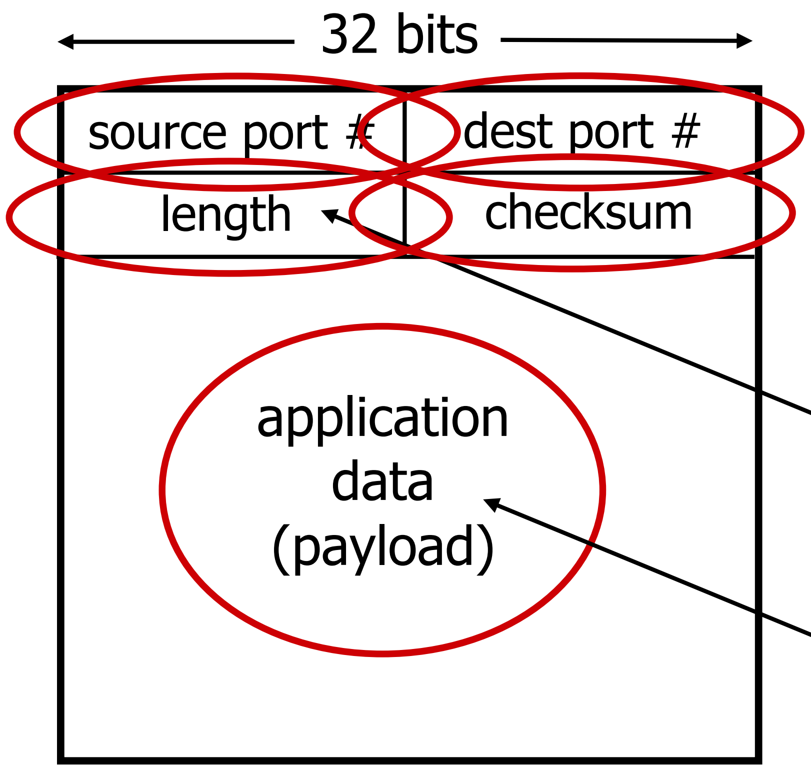
# UDP: User Datagram Protocol [RFC 768]



```
                                                INTERNET STANDARD
RFC 768                                                  J. Postel
                                                               ISI
                                                    28 August 1980



                      User Datagram Protocol
                      ----------------------

Introduction
------------

This User Datagram  Protocol  (UDP)  is  defined  to  make  available  a
datagram   mode  of  packet-switched   computer   communication  in  the
environment  of  an  interconnected  set  of  computer  networks.   This
protocol  assumes  that the Internet  Protocol  (IP)  [1] is used as the
underlying protocol.

This protocol  provides  a procedure  for application  programs  to send
messages  to other programs  with a minimum  of protocol mechanism.  The
protocol  is transaction oriented, and delivery and duplicate protection
are not guaranteed.  Applications requiring ordered reliable delivery of
streams of data should use the Transmission Control Protocol (TCP) [2].

Format
------


                  0      7 8     15 16    23 24    31
                 +--------+--------+--------+--------+
                 |     Source      |   Destination   |
                 |      Port       |      Port       |
                 +--------+--------+--------+--------+
                 |                 |                 |
                 |     Length      |    Checksum     |
                 +--------+--------+--------+--------+
                 |
                 |          data octets ...
                 +---------------- ...
```

# UDP segment header

length, in bytes of UDP segment, including header



UDP segment format

length, in bytes of UDP segment, including header

data to/from application layer (e.g. DNS query/response, audio/video)

# UDP checksum *(error detection: changed/altered or not)*

*Goal:* detect errors (*i.e.,* flipped bits) in transmitted segment

| | 1st number | 2nd number | sum |
|---|---|---|---|
| Transmitted: | 5 | 6 | 11 |
| Received: | 4 | 6 | 11 |

receiver-computed checksum ≠ sender-computed checksum (as received)

# Internet checksum: an example

* On the sender side: UDP segment (UDP header fields + IP addresses) is converted to sequences of 16-bit integers

All of the sequences are added and the 1's complement is obtained (0's are coverted to 1's and vice versa)

* On the receiver side: the same jobs are done and the two results are compared

example: add two 16-bit integers

```
                            1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
                            1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
                          ---------------------------------
wraparound                1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
(overflow occured)        |------------------------------->
                          ---------------------------------
                     sum    1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
                checksum    0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

*Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result

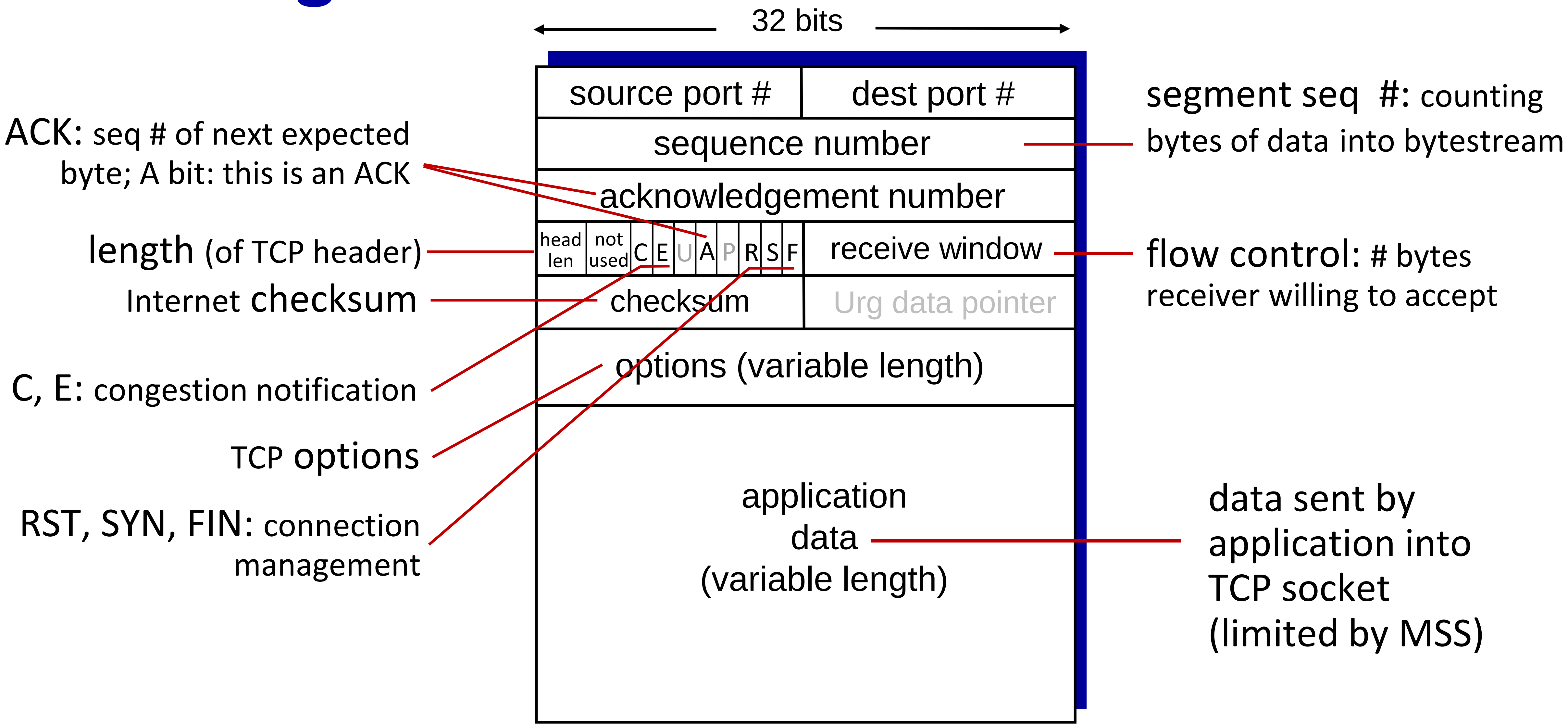
# Chapter 3: roadmap

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

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

- point-to-point:
  - one sender, one receiver: no multicast (1:n), first handshaking
- reliable
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size (1460 bytes in general)
- cumulative ACKs
- pipelining:
  - TCP congestion and flow control
- connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange (a logical conn.)
- flow controlled:
  - sender will not overwhelm receiver

# TCP segment structure



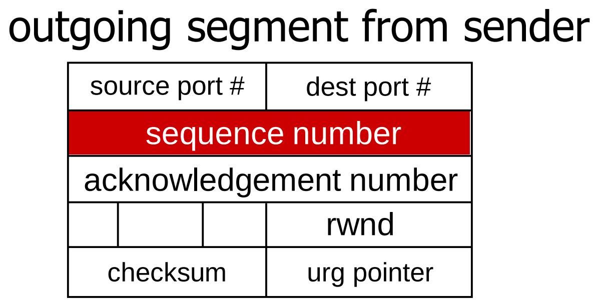
32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
C
E
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
segment seq #: counting bytes of data into bytestream
ACK: seq # of next expected byte; A bit: this is an ACK
length (of TCP header)
Internet checksum
flow control: # bytes receiver willing to accept
C, E: congestion notification
TCP options
RST, SYN, FIN: connection management
data sent by application into TCP socket (limited by MSS)

PSH (P): receiver should pass data to AL immediately
URG (U): data is marked as urgent by the sender (last byte of this urgent data is indicated by the 16-bit urgent data pointer field)
PSH, URG, and the urgent data pointer are not used in practice

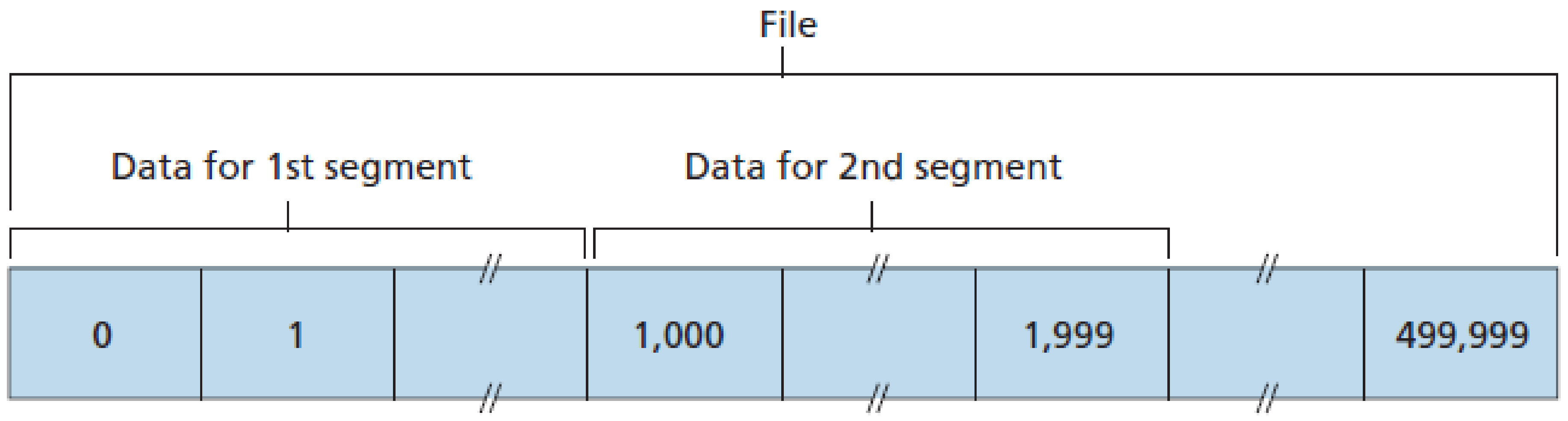
# TCP sequence numbers

*Sequence numbers:*

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



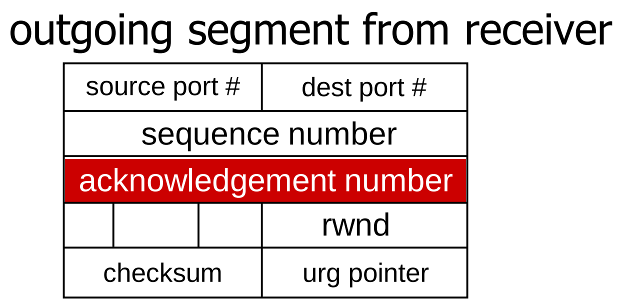
- data stream has a size of 500,000 bytes
- MSS is 1,000 bytes
- TCP constructs 500 segments of the data stream
- The first segment's sequence number: 0, the second segment's sequence number: 1,000, the third segment's sequence number 3,000 …
- each sequence number is inserted in the sequence number field of the corresponding segment



# TCP ACKs

*Acknowledgements:*

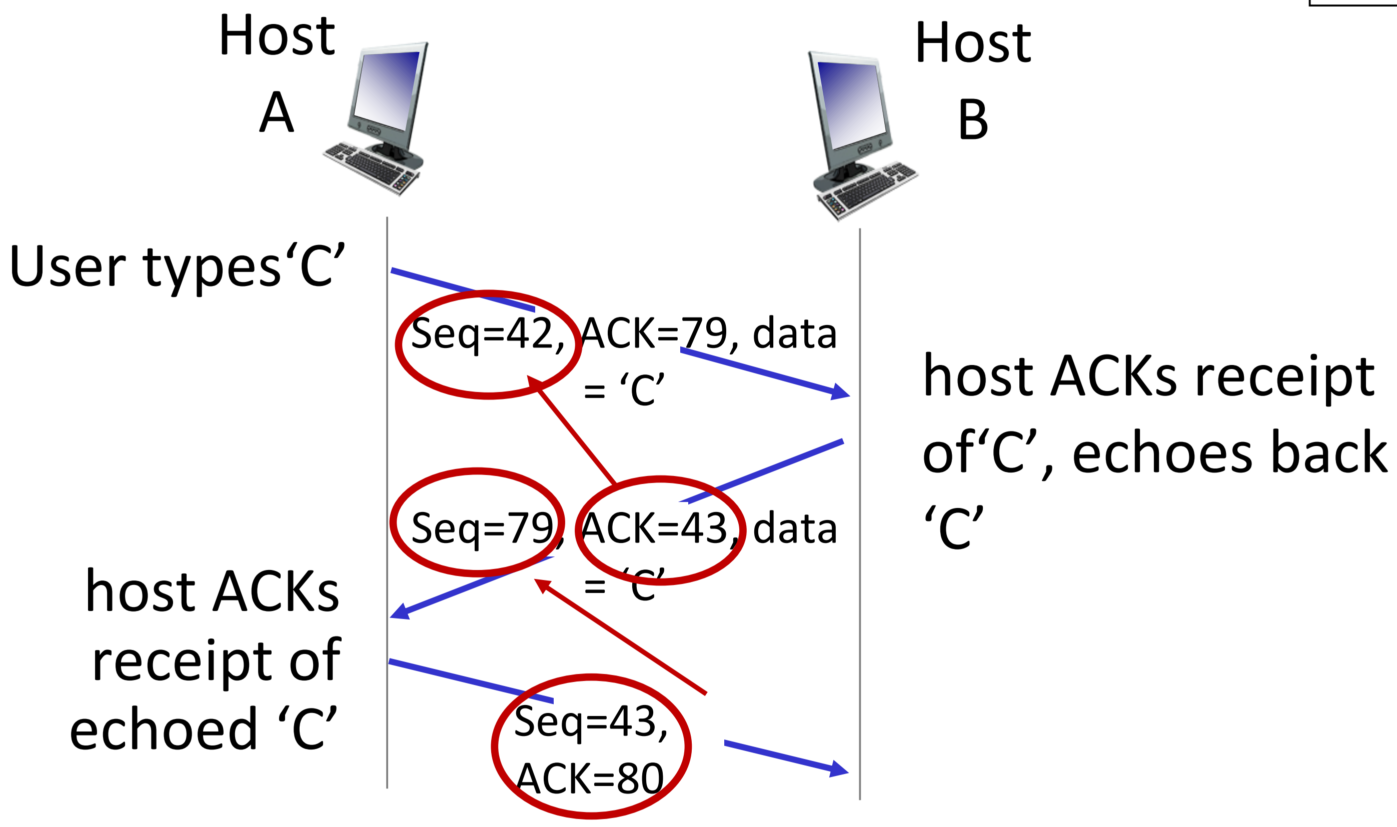
- seq # of next byte expected from other side



- A has received all bytes numbered 0 through 78 from B

- A is waiting for byte 79 and all the subsequent bytes from B

- When A is sending a segment to B, A puts 79 in the acknowledgment number field of the segment it sends to B


Host A
Host B
User types'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80

# TCP ACKs

*Acknowledgements:*

- cumulative ACK

- A has received one segment from B **containing bytes 0 through 35** and another segment **containing bytes 90 through 100**

- A has not yet received bytes 36 through 89 (A is still waiting for byte 36 and beyond)

- A's next segment to B will contain **36** in the acknowledgment number field

- TCP only acknowledges bytes up to the first missing byte in the stream (TCP provides cumulative acknowledgments for the first 35 bytes)

- A received the third segment (bytes 90 through 100) before receiving the second segment (bytes 36 through 89), the third segment arrived out of order. What will A do?

There can be two choices:

the receiver immediately discards out-of-order segments

the receiver keeps the out-of-order bytes and waits for the missing bytes to fill in the gaps (more efficient in terms of network bandwidth, and it is the approach prefered in practice)

## TCP round trip time, timeout

- Some segments maybe lost, when a segment should be sent again or retransmitted?
- First define the RTT for a segment by ‘exponential weighted moving average’ formula

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

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

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

- SampleRTT: time between when the segment is sent and when an acknowledgment is received
- SampleRTT is calculated for segments that have been transmitted once
- SampleRTT values will fluctuate from segment to segment (cause of congestion and the network load)
- In order to estimate a typical RTT, average of the SampleRTT values are taken (EstimatedRTT)
- When a new SampleRTT value is calculated, TCP updates EstimatedRTT
- another measurement, DevRTT: variability of the RTT; estimate of how much SampleRTT deviates from EstimatedRTT
- If SampleRTT values have little fluctuation, then DevRTT will be small; if there is a lot of fluctuation, DevRTT will be large, recommended value of beta is 1/4

# 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"

- Timeout interval should be greater than or equal to EstimatedRTT
- If it is less than RTT, TCP would make unnecessary retransmissions
- If it is much larger than EstimatedRTT; when a segment is lost, TCP would not quickly retransmit the segment
- timeout interval is set to the EstimatedRTT plus some margin
- This margin should be large when there is a lot of fluctuation in the SampleRTT values and it should be small when there is little fluctuation
- **DevRTT is used since it is a measurement of fluctuation**

## 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*

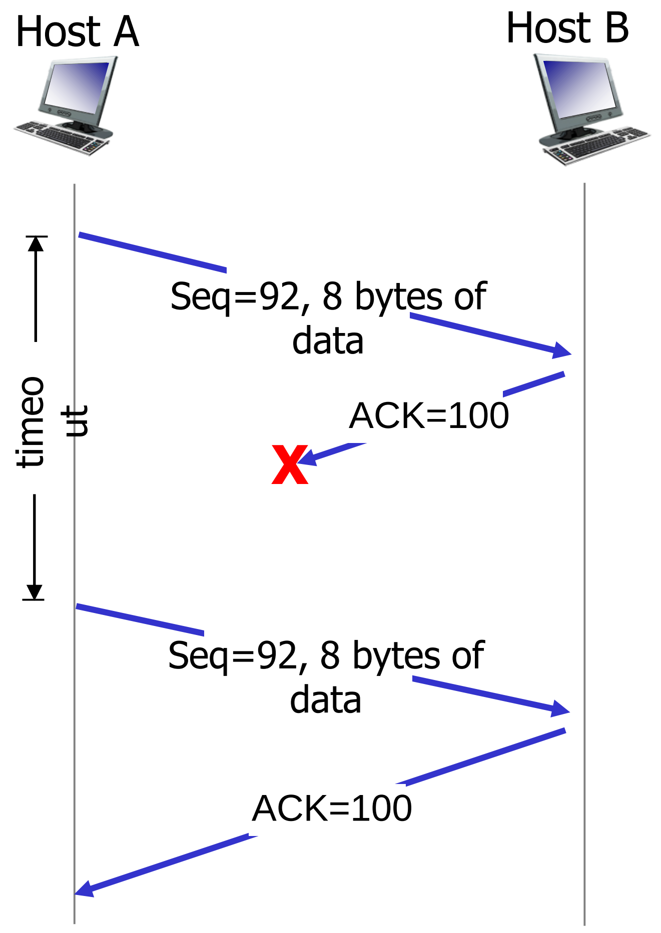
- retransmit segment that caused timeout
- restart timer

*event: ACK received*

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

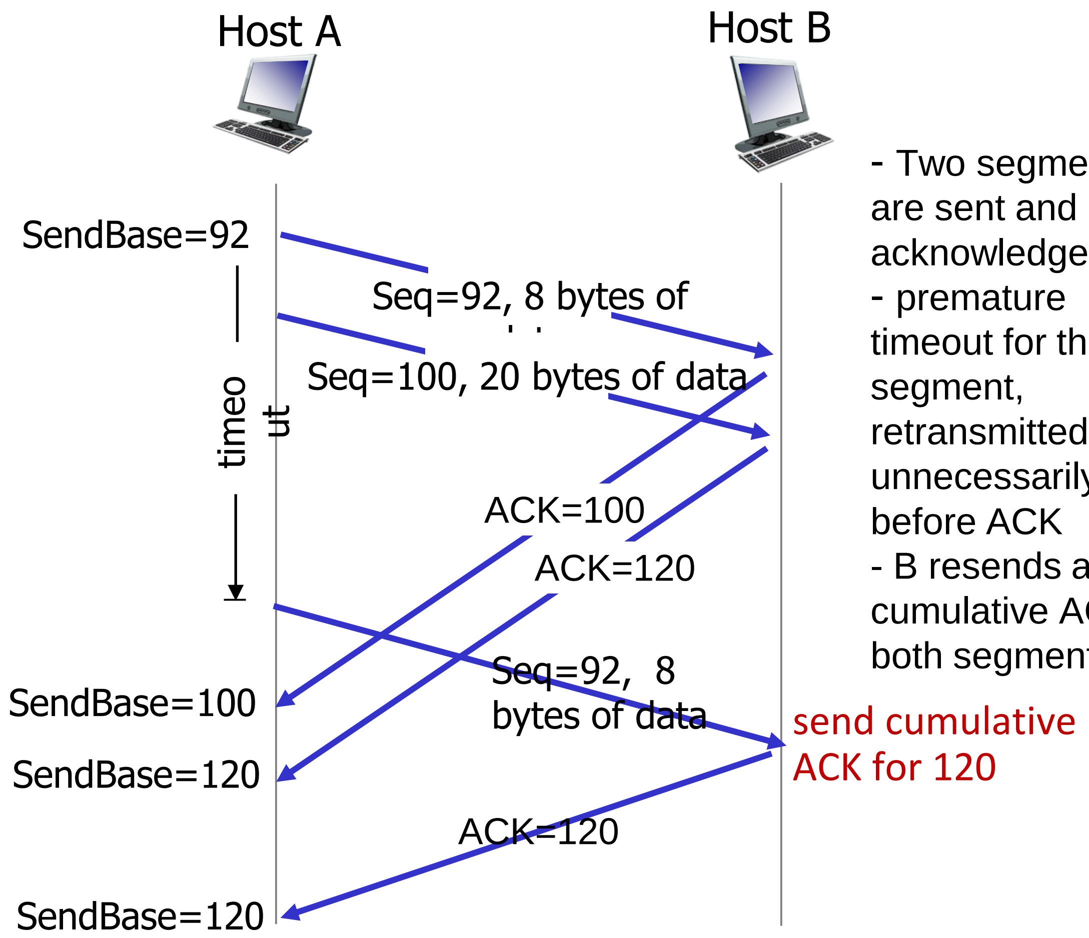
# TCP: retransmission scenarios

- ACK is lost
- TCP timeout mechanism steps in
- Another copy the segment is transmitted



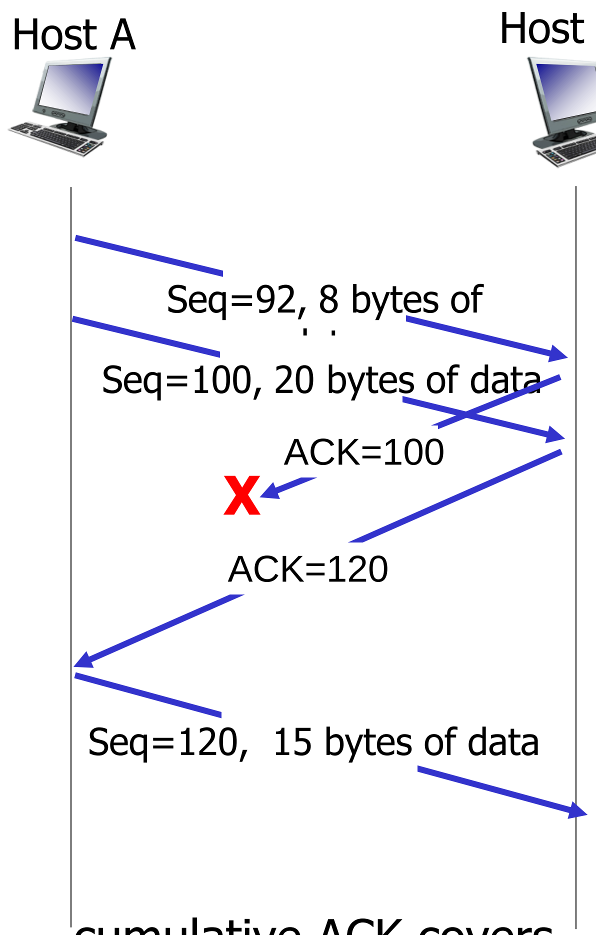
lost ACK scenario

- Two segments are sent and acknowledged
- premature timeout for the first segment, retransmitted unnecessarily before ACK
- B resends a cumulative ACK for both segments



premature timeout

## TCP: retransmission scenarios



cumulative ACK covers for earlier lost ACK

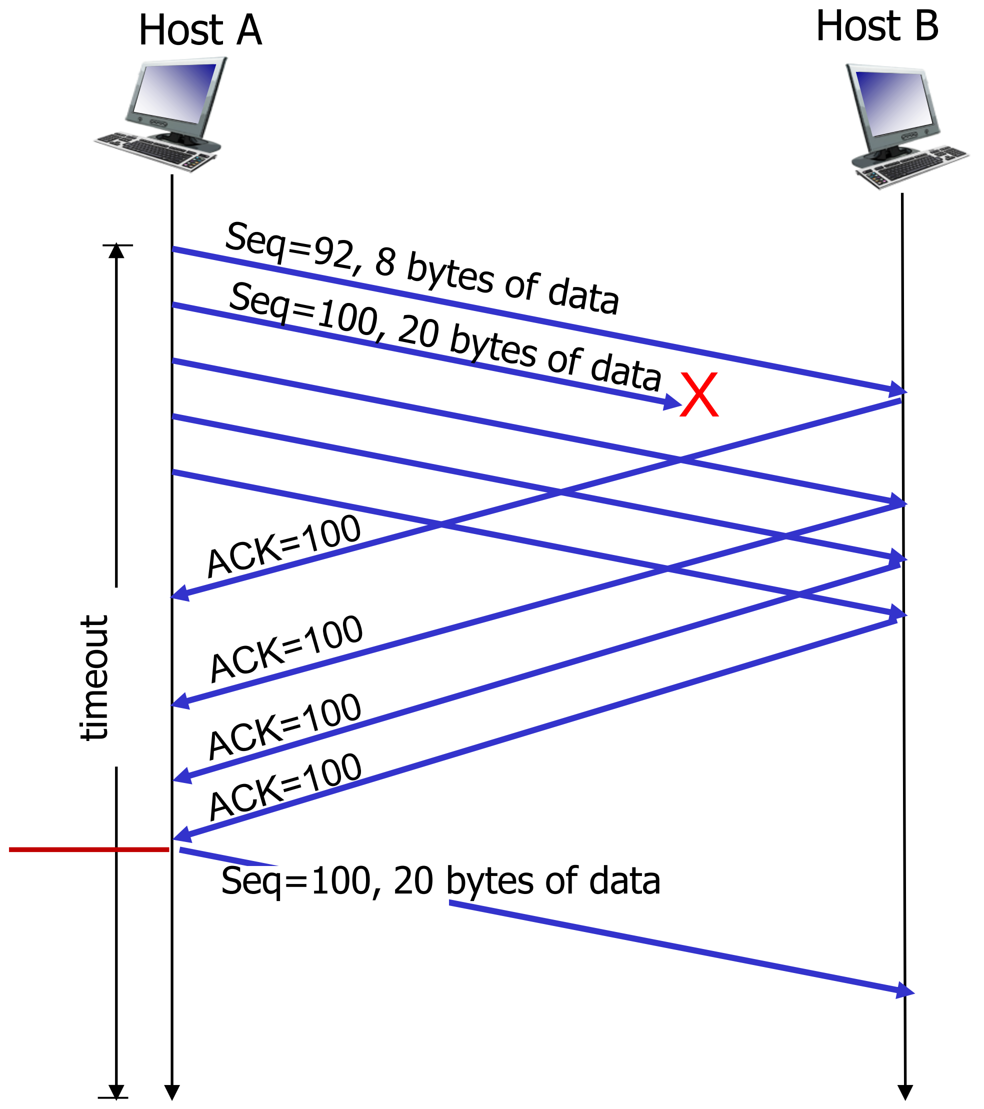
- Two segments are sent
- The first ACK is lost but the second ACK, a cumulative ACK, arrives at the sender
- So a third segment can be transmitted

# TCP fast retransmit

*TCP fast retransmit*

- till now reliability has been covered

- 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 (retransmit ASAP)**



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

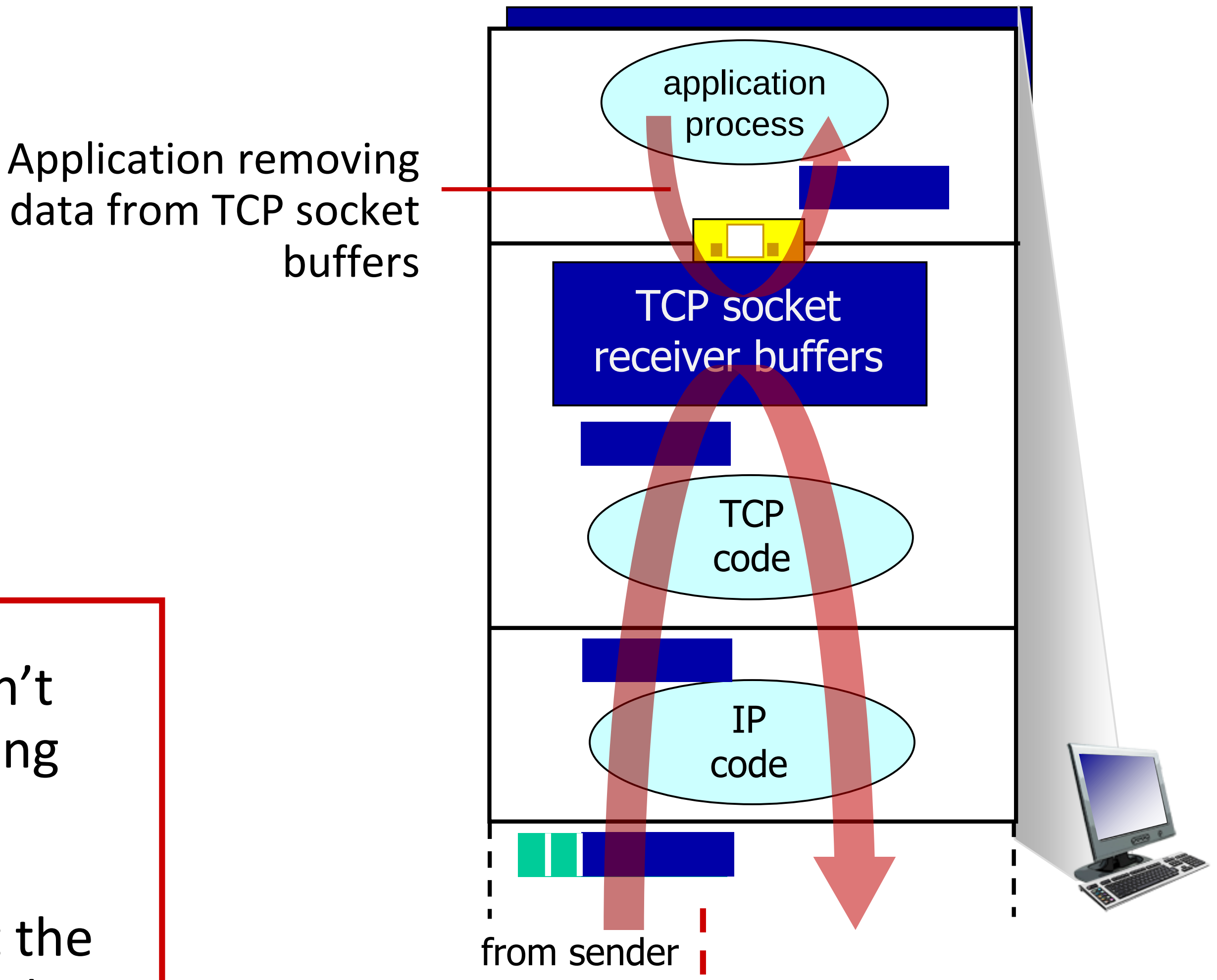
# TCP flow control

*Q:* What happens if network layer delivers data faster than application layer removes data from socket buffers? (i.e. application is slow at reading the data, the sender can overflow the receiver side)

## flow control

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

Speedmatching: matching the rate that the sender is sending, against the rate that the receiving application is reading


Application removing data from TCP socket buffers
application process
TCP socket receiver buffers
TCP code
IP code
from sender

receiver protocol stack

# TCP flow control

- Sender maintain a variable called the receive window, it gives the sender an idea of 'how much free buffer space is available at the receiver'
- Since TCP is full-duplex, the sender and the receiver are both send and receive data, they are both sender and receiver, each host has a receive buffer and also a distinct receive window
- Suppose that A is sending a large file to B. B allocates a receive buffer to this connection and the size of receive buffer is RcvBuffer. Application process in B reads from this buffer.

prevents overflowing the allocated buffer

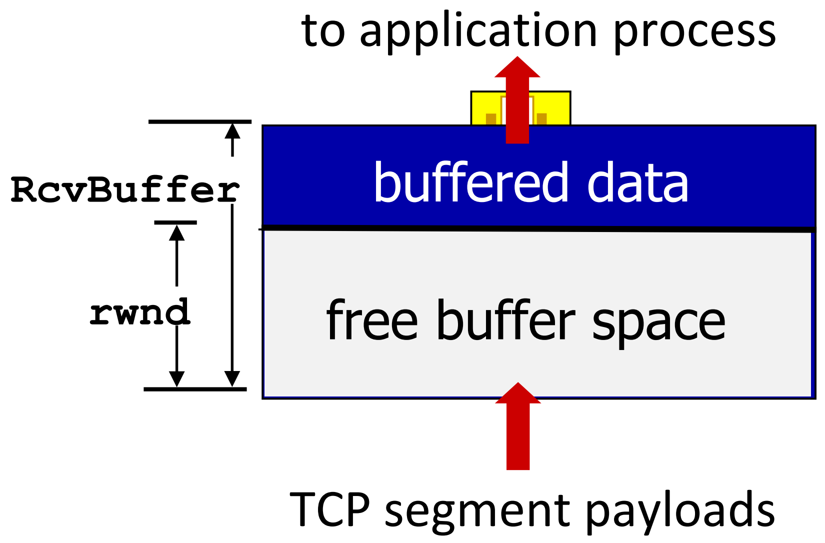
$$\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$$

receive window: amount of free space in buffer, dynamic

$$\text{rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$$

- value of rwnd is placed in receive window field of each segment sent by B
- A keeps track of LastByteSent and LastByteAcke
- LastByteSent - LastByteAcked: amount of unacknowledged data that A sent
- If it is less than or equal to rwnd then A is assured that it is not overflowing the receive buffer at B



TCP receiver-side buffering

# TCP connection management: TCP 3-way handshake

**- client and server create a TCP socket, enter the LISTEN state**
**- client then connects to the server sending a SYN message with a sequence number x**
**- server receives the SYN message and then enters the SYN received state**
**- server sends a SYN ACK message back to the client: ACK bit is set to 1 and acknowledgement number is x plus 1 and sequence number is y**
**- client sends an ACK message to the server and when the server receives this segment, then it enters the ESTABLished state**

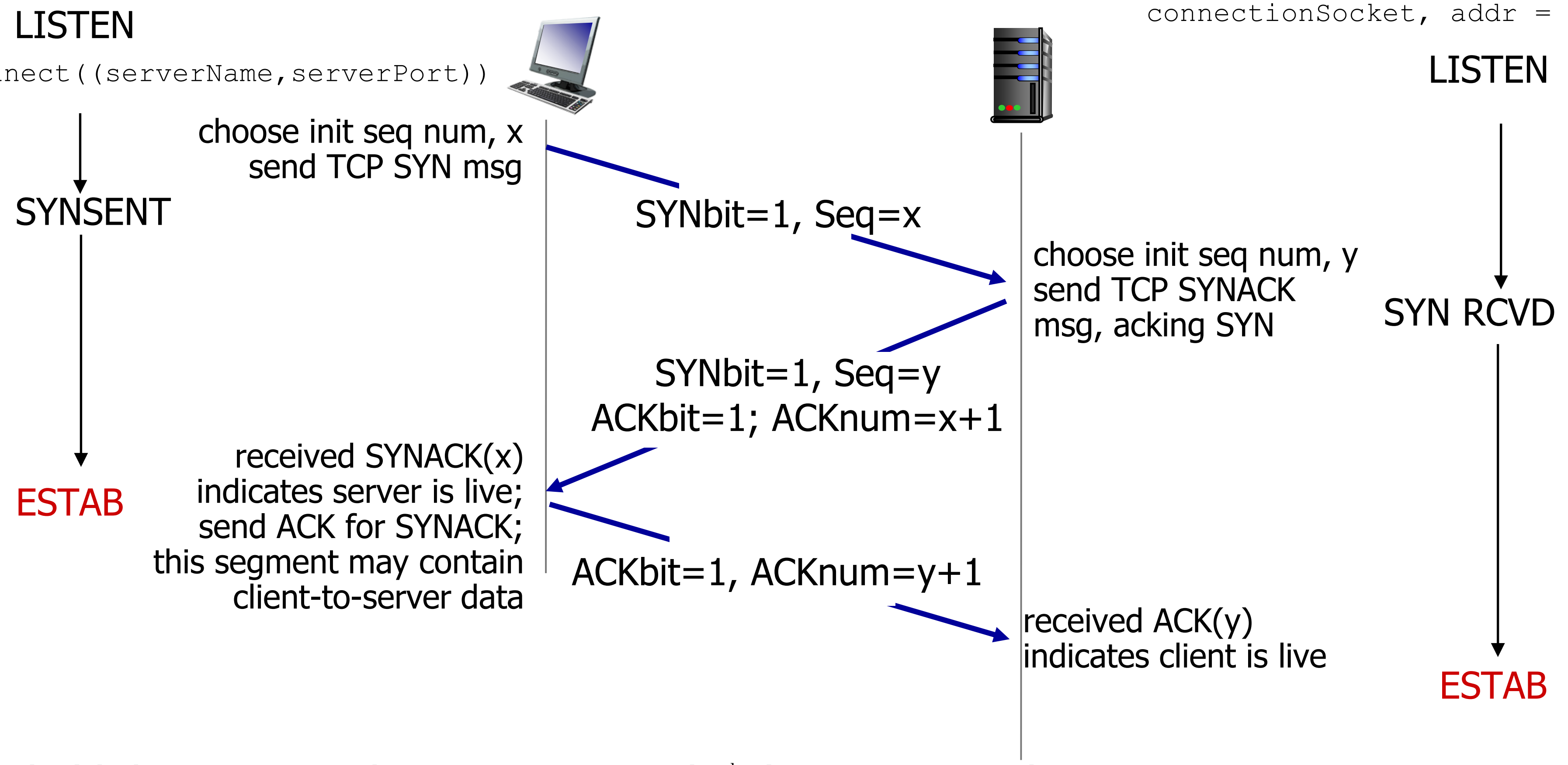
## Server state

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

## Client state

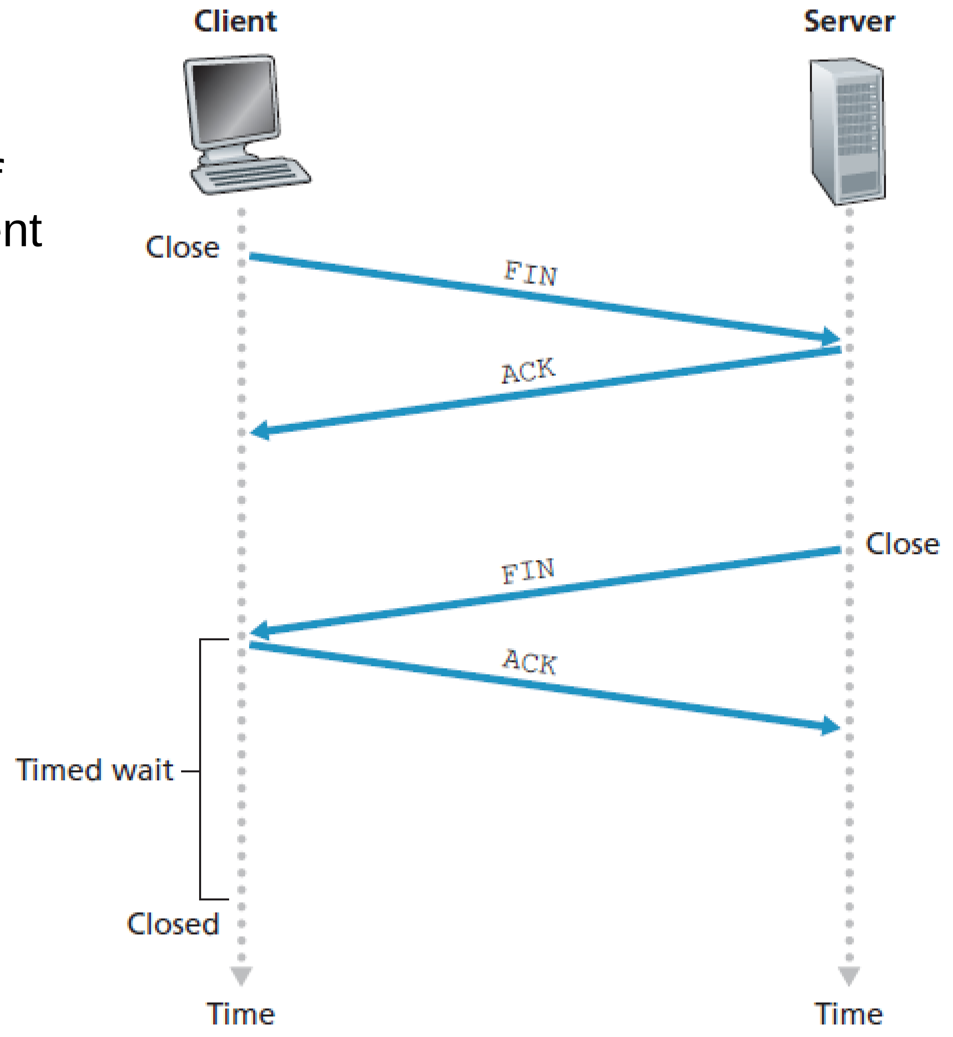
```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

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



# Closing a TCP connection

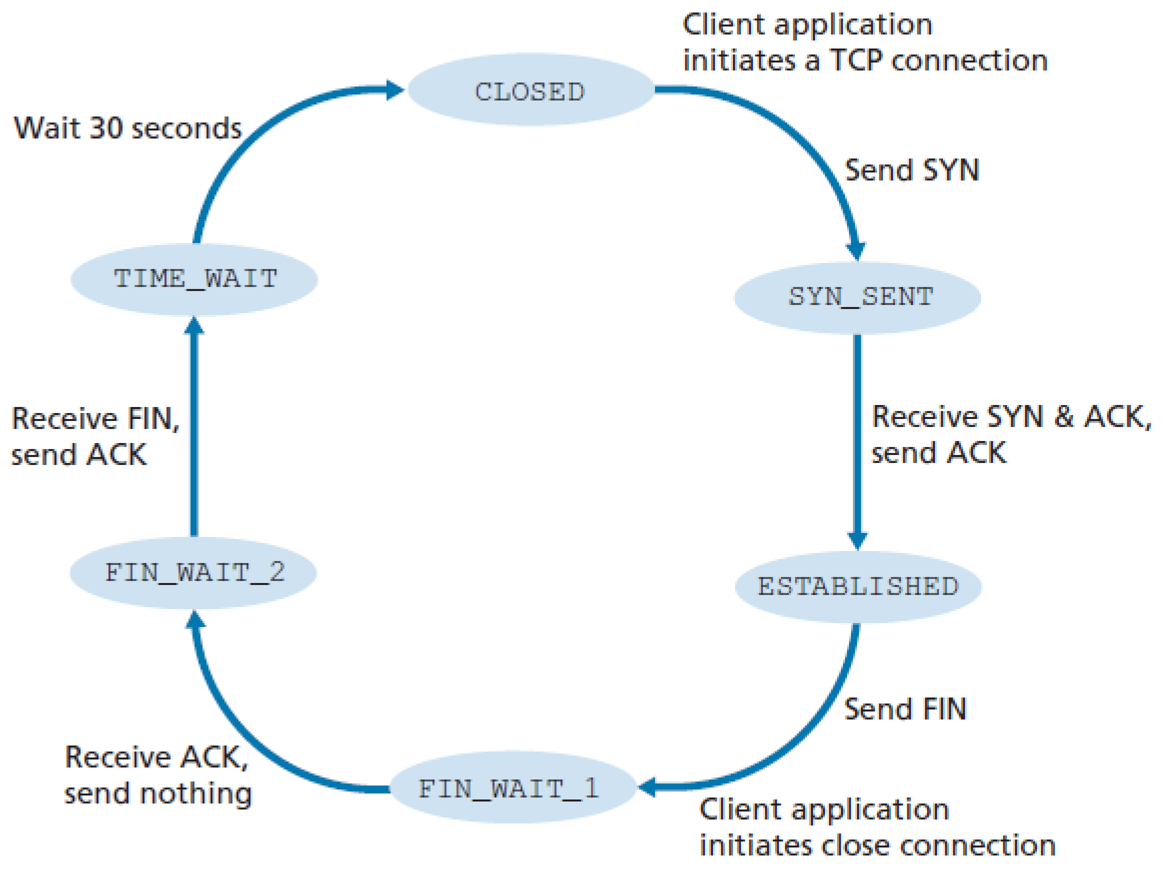
both parties close their side of connection with a TCP segment (FIN bit set) and send ACK



# TCP states – client side

A number of TCP states during the life of a TCP conn.
- client TCP begins in CLOSED, application on the client initiates a new TCP conn (by sending a SYN segment to TCP in the server)
- After sending client TCP enters the SYN_SENT state
- While in the SYN_SENT state, the client TCP waits for a segment from the server TCP: syn & ack
- then the client send the acknowledgement and enters the ESTABLISHED state
- While in the ESTABLISHED state, the TCP client can send and receive TCP segments containing payload

Suppose that client wants to terminate the conn
- Client TCP sends a TCP segment with the FIN bit set and enters the FIN_WAIT_1 state
- While in the FIN_WAIT_1 state, the client TCP waits for a TCP segment from the server with an acknowledgment
- When it receives this segment, the client TCP enters the FIN_WAIT_2 state
- While in the FIN_WAIT_2 state, the client waits for another segment from the server with the FIN bit set
- After receiving this segment, the client TCP acknowledges the server's segment and enters the TIME_WAIT state
- In TIME_WAIT state, TCP client resends the final acknowledgment if the previous ACK is lost
- time spent in the TIME_WAIT state is implementation-dependent, but typically it takes 30 seconds, 1 minute, and 2 minutes
- After the wait, the connection is closed and all resources on the client side (including port numbers) are released


Client application initiates a TCP connection
CLOSED
Send SYN
SYN_SENT
Receive SYN & ACK, send ACK
ESTABLISHED
Send FIN
Client application initiates close connection
FIN_WAIT_1
Receive ACK, send nothing
FIN_WAIT_2
Receive FIN, send ACK
TIME_WAIT
Wait 30 seconds

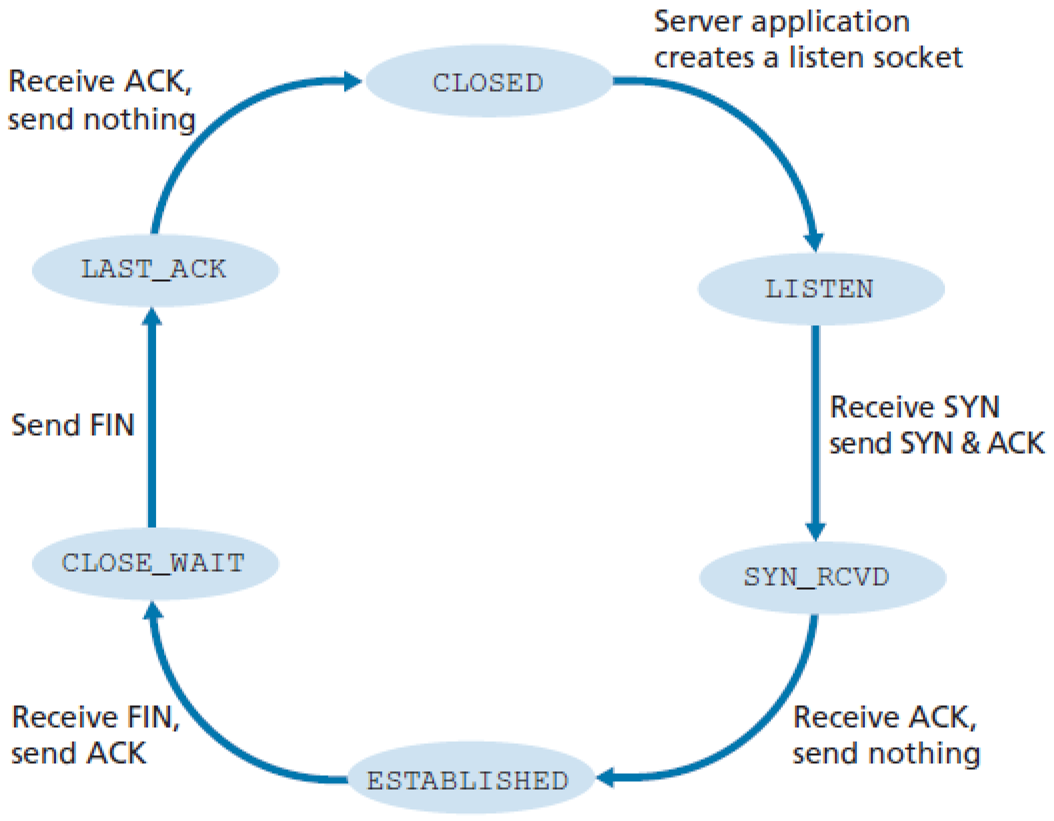
# TCP states – server side

states of server side TCP

- server TCP begins in the CLOSED state
- server application creates a listen socket then enters the LISTEN state
- While in LISTEN state, the application on the client side initiates a new TCP connection by sending a SYN segment to TCP in the server
- Server TCP sends SYN & ACK segment and then enters SYN received state
- In this state, server waits for the final ACK from the client to establish the conn
- While in the SYN_SENT state, the client TCP waits for a segment from the server TCP

- After getting the ACK, server enters ESTABLISHED state.
- While in the ESTABLISHED state, the TCP server can send and receive TCP segments containing payload

- server gets FIN from the client and then sends ACK to this FIN segment and enters CLOSE_WAIT state
- server send it’s FIN to client and enters LAST_ACK state
- waits for the last ACK from the client and sends nothing
- After getting the last ack from the client, connection is closed


CLOSED
Server application creates a listen socket
LISTEN
Receive SYN send SYN & ACK
SYN_RCVD
Receive ACK, send nothing
ESTABLISHED
Receive FIN, send ACK
CLOSE_WAIT
Send FIN
LAST_ACK
Receive ACK, send nothing