Lecture 4: Transport Layer

Acknowledgement: Material presented in this lecture is predominantly based on the Request for Comments:

- RFC 793 Transmission Control Protocol (Local copy)
- See also RFC 1122, and many updates
- *Computer Networking: A Top Down Approach,* J. Kurose, K. Ross, 7th ed., 2017, Addison-Wesley, Chapter 3

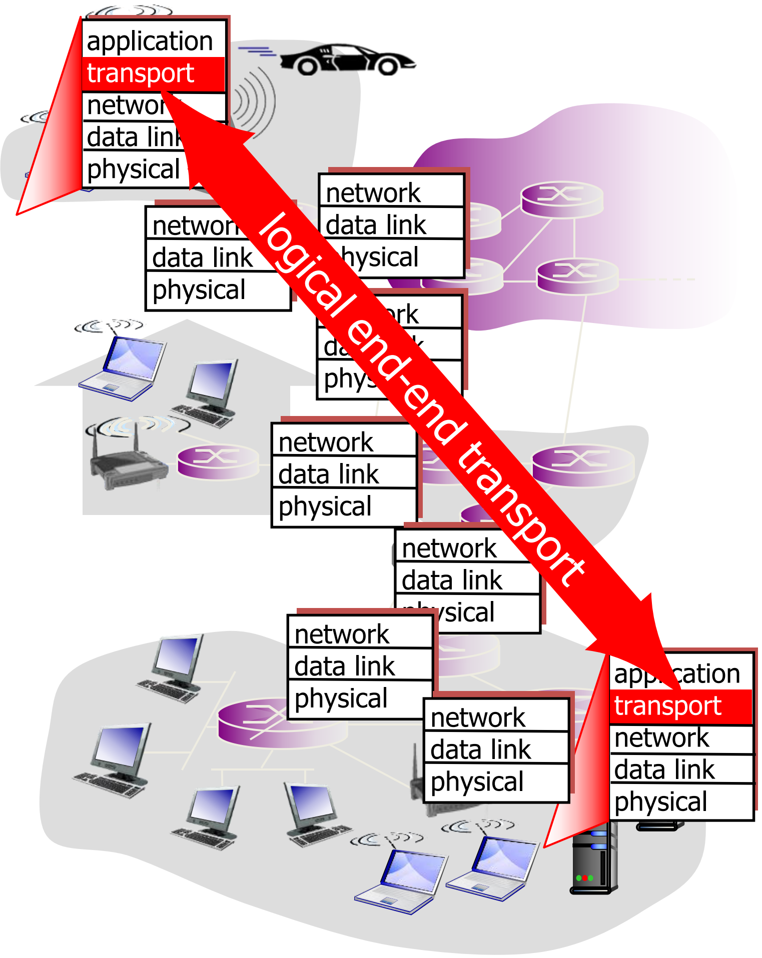
(2019 version)

Lecture 4: Transport Layer Outline

- Transport-layer services
- Multiplexing and de-multiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer USE TUTE TRAIN Example!
  - flow control
  - connection management
- TCP congestion control

Transport services and protocols

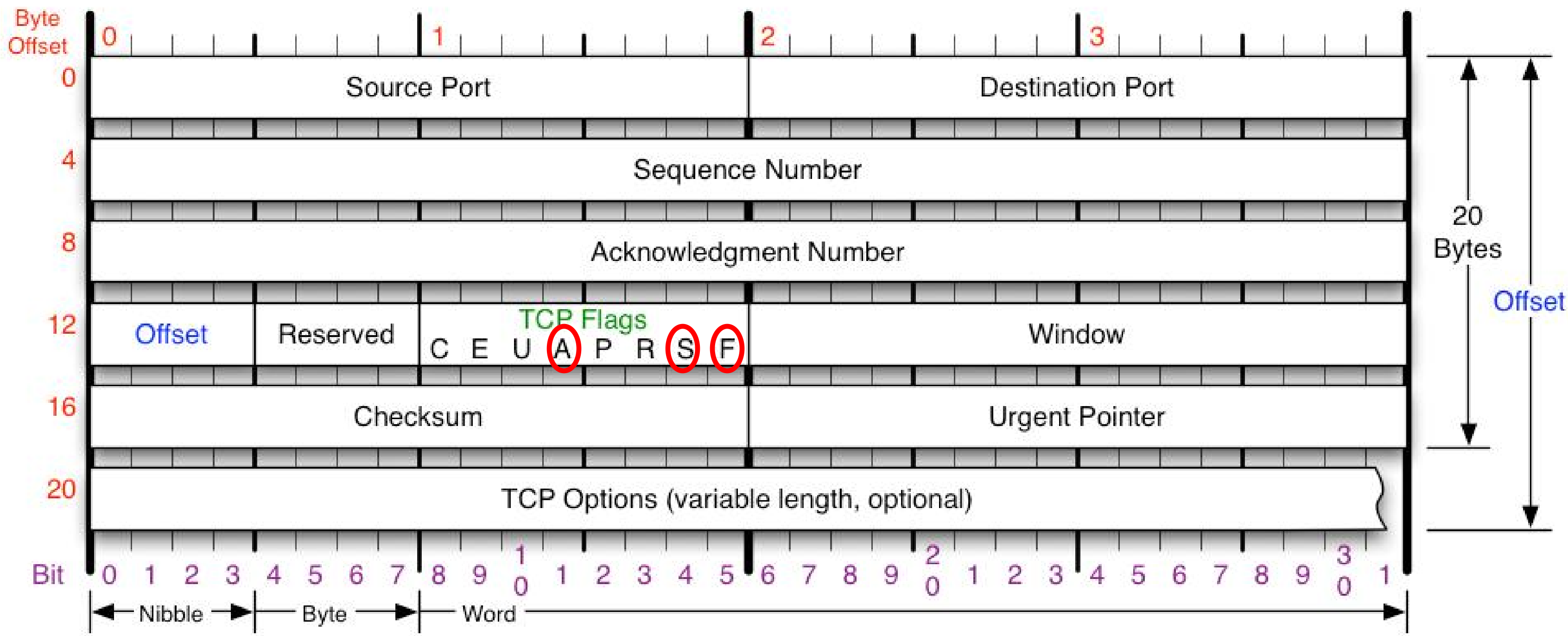
- Interfaces the application layer **processes** to the network layer
- transport protocols run in the end systems (not in the routers)
- Two basic Internet transport protocols:
- **TCP** (RFC 793 Local copy) – reliable, connection-oriented
- **UDP** (RFC 768 Local copy) – unreliable, connectionless



Principles of Operations (from RFC 793)

- The primary purpose of the TCP is to provide reliable, securable logical connection service **between pairs of processes.**
- To provide this service on top of a less reliable internet communication system requires facilities in the following areas:
  - Reliable Data Transfer
  - Flow Control
  - Multiplexing
  - Connections

TCP Header



- **Source** (sending) and **Destination** (receiving) ports
- **Sequence number**: the first data byte = Seq# + SYN flag
- **Acknowledgement number**: if ACK then Ack# = byte# that the receiver is expecting.
- **Data offset**: Header size in 32-bit words (5..20)
- **Checksum**: 16-bit checksum of the pseudo-**header and data**
- 16-bit **receive Window**: # bytes that the receiver is currently willing to receive

Flags (aka Control bits):

CWR – Congestion Window Reduced
ECE – ECN-Echo (RFC 3168).
URG – the **URGent Pointer** field is significant
ACK – indicates that the ACKnowledgment field is significant
PSH – Push function: send data from the buffer up to application
RST – Reset the connection
SYN – Synchronize sequence numbers
FIN – No more data from sender

Checksum

- The checksum field is the 16 bit one's complement of the **one's complement sum of all 16 bit words** in the header and payload.
- If a segment contains an **odd number of** header and text **bytes** to be check-summed, the **last byte is padded** on the right with zeros to form a 16-bit word for checksum purposes.
- The pad is not transmitted as part of the segment.
- While computing the checksum, the checksum field itself is replaced with zeros.
- The checksum **also covers a 96 bit (IPv4) pseudo header** conceptually **prefixed** to the TCP header.
- This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length.
- This gives the TCP protection against misrouted segments.
- This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

Pseudoheaders (IPv4, v6, UDP)

TCP pseudo-header for checksum computation (IPv4)

Bit offset	0–3	4–7	8–15	16–31
0	Source address			
32	Destination address			
64	Zeros		Protocol	TCP length

The source and destination addresses are those of the IPv4 header.
The protocol value is 6 for TCP.
The TCP length field is the length of the TCP header and data (measured in bytes).

TCP pseudo-header for checksum computation (IPv6)

Bit offset	0–7	8–15	16–23	24–31
0	Source address			
96				
128	Destination address			
224				
256	TCP length			
288	Zeros			Next header

Next Header – the protocol value for TCP

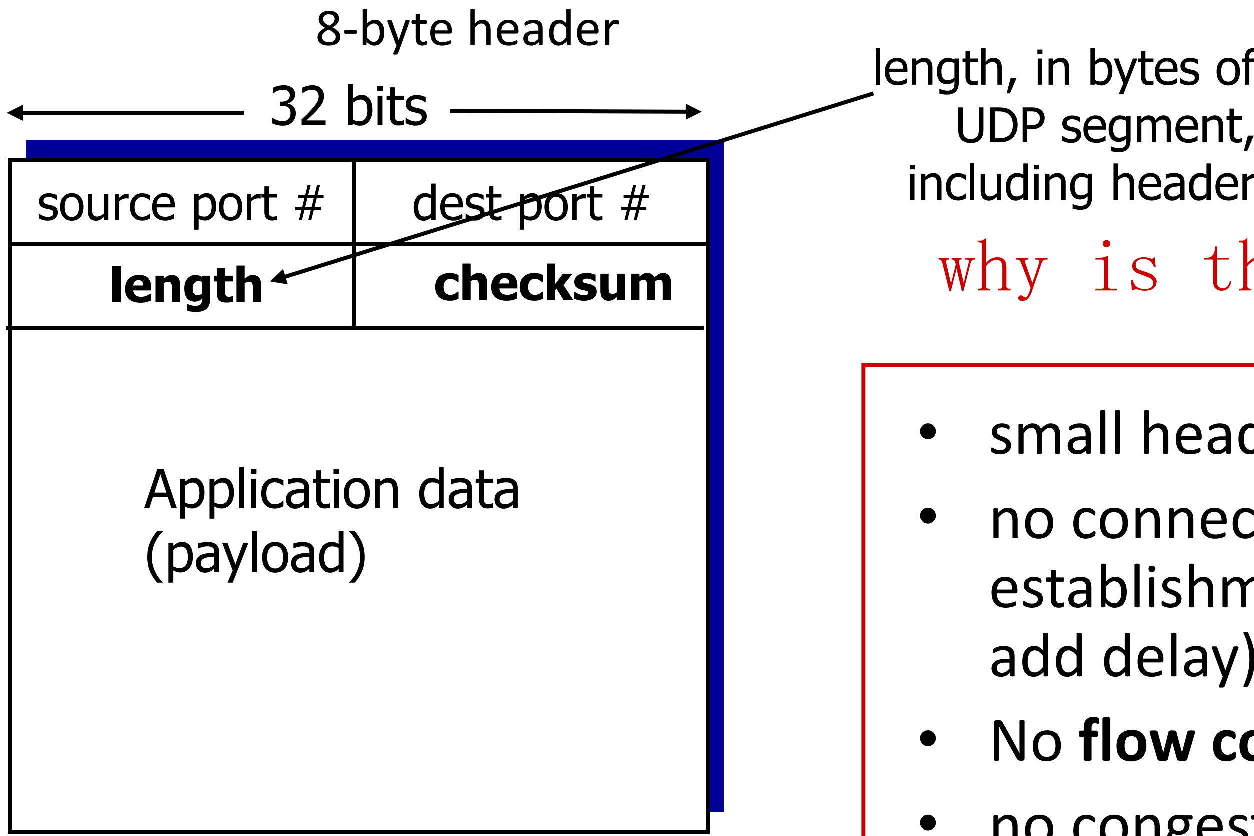
From Wikipedia

UDP: User Datagram Protocol [RFC 768]

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

- UDP used in:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP Simple Net. Mngmt Prot.
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment format



UDP segment format
8 bytes plus payload

why is there a UDP?

- small header size (8 bytes)
- no connection establishment (which can add delay)
- No **flow control,** no ack.
- no congestion control: UDP can blast away as fast as desired

Principles of the Multiplexing (from RFC 793)

- To allow for many processes within a single host to use TCP communication facilities simultaneously, the TCP provides a set of **ports** within each host.
- Concatenated with the network and host IP addresses this forms a **socket.**
- A pair of sockets uniquely identifies each connection.
- A socket may be simultaneously used in multiple connections.
- The binding of ports to processes is handled independently by each host.
- It proves useful to attach frequently used processes to fixed ports which are made known to the public.
- These services can then be accessed through the known ports.

Multiplexing/de-multiplexing

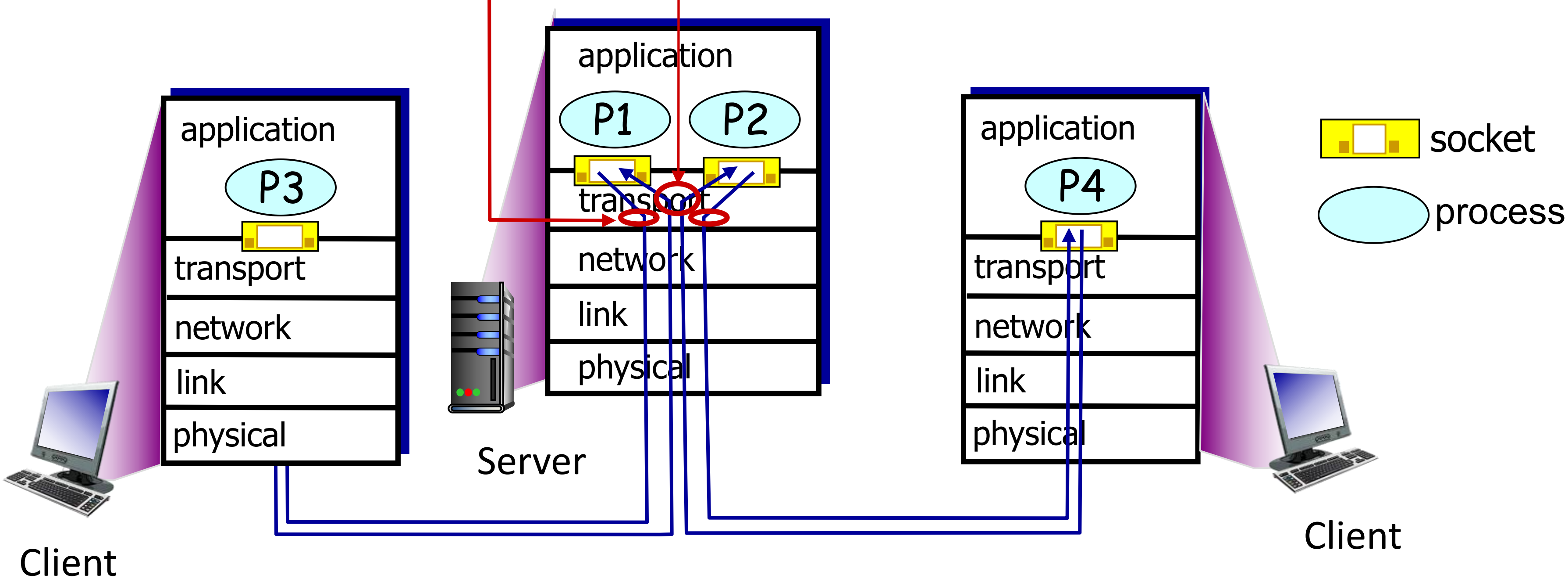
To allow for many processes within a single Host to use TCP communication facilities simultaneously

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for de-multiplexing)

De-multiplexing at receive

use header info to deliver received segments to correc socket



Connectionless (UDP) de-multiplexing

- created socket has host-local port number:

```
DatagramSocket
mySocket1 = new
  DatagramSocket(12534);
```

- when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port number

- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

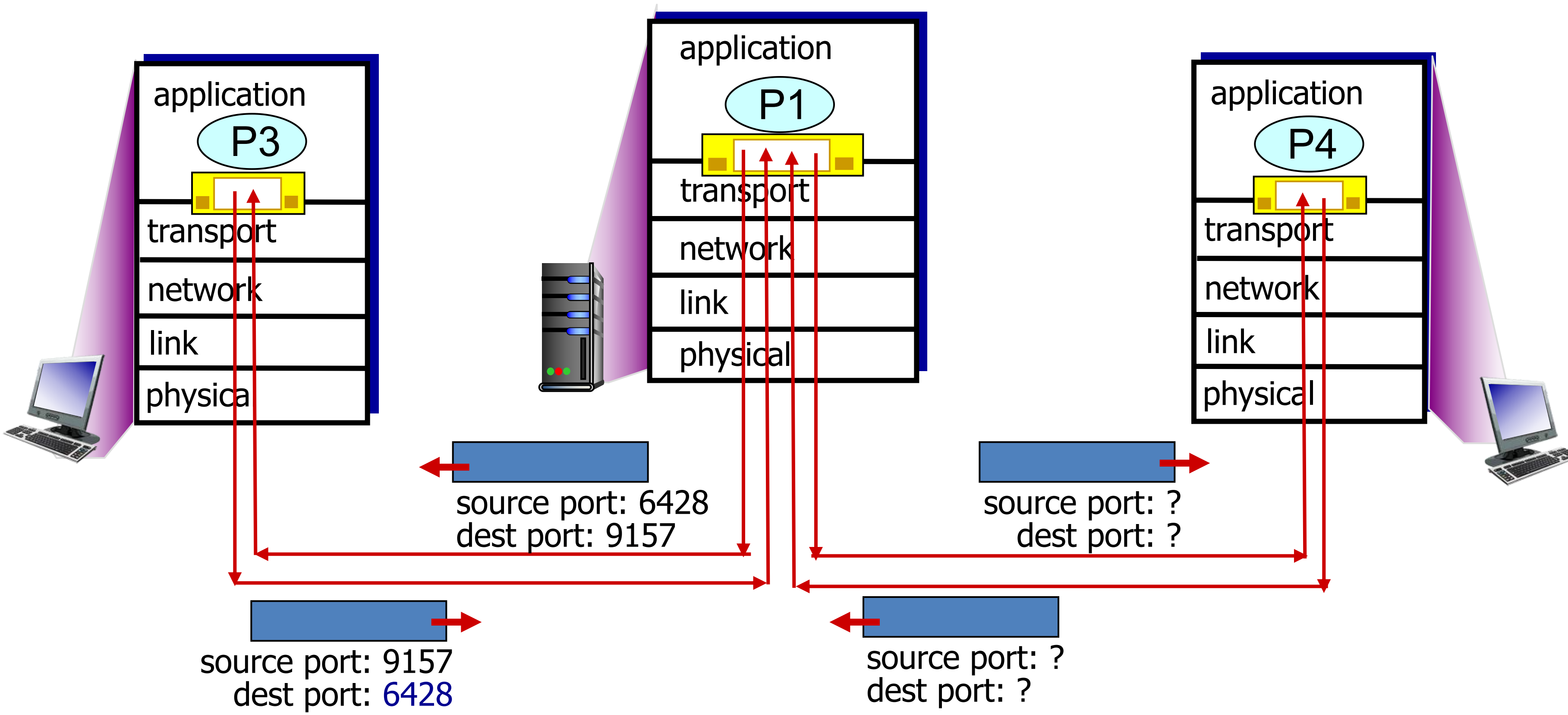
IP datagrams with *same dest. port number,* but different source IP addresses and/or source port numbers will be directed to the *same socket* at destination

Connectionless demux: example

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
  (9157);
```

```
DatagramSocket
  serverSocket = new
  DatagramSocket
  (6428);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
  (5775);
```



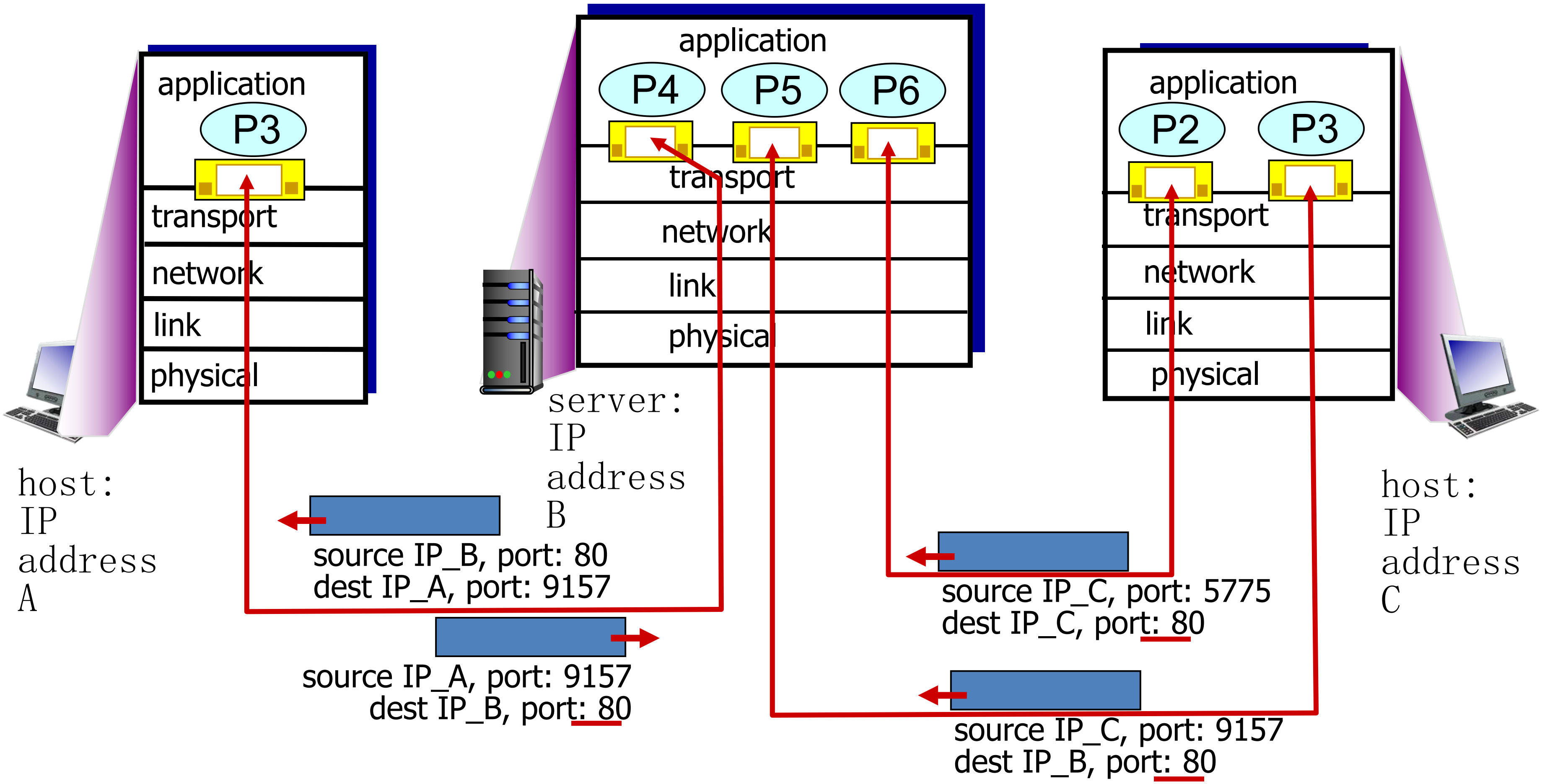
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

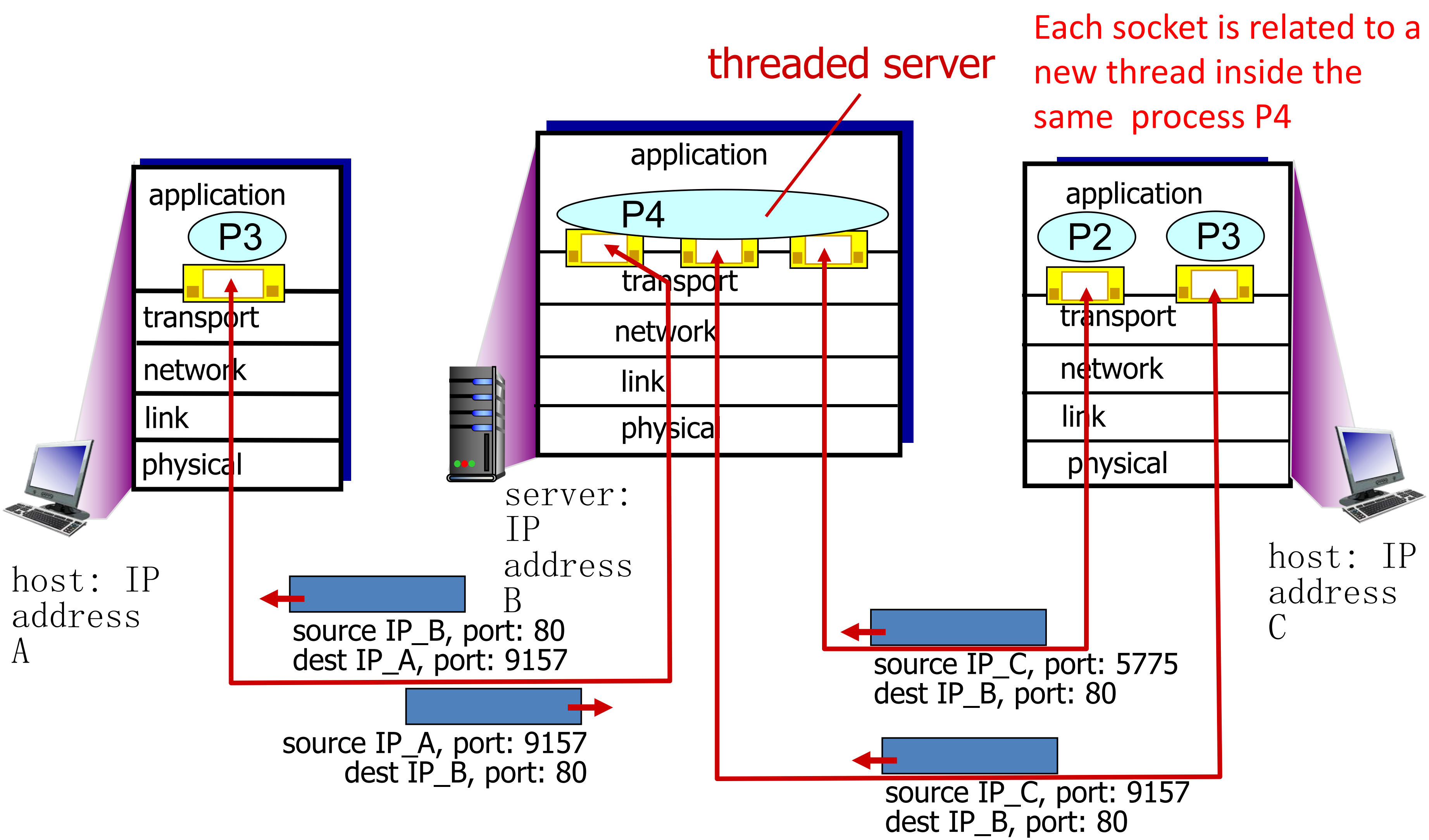
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example

three segments, all destined to IP address: B, dest port: 80
are demultiplexed to *different* sockets



Connection-oriented demux: example



Connections (from RFC 793 p.5)

- The reliability and flow control mechanisms require that TCPs initialize and maintain certain **status information** for each data stream.
- The combination of this information, including sockets, sequence numbers, and window sizes, is called a **connection**.
- When two processes wish to communicate, their TCP's must first establish a connection (initialize the status information on each side).
- When their communication is complete, the connection is terminated or closed.
- Since connections must be established between unreliable hosts and over the unreliable Internet, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

Connection Establishment Passive Open

- A connection is fully specified by the pair of sockets at the ends.
- A connection is specified in the **OPEN call** by the local port and foreign socket arguments.
- In return, the TCP supplies a (short) local connection name by which the user refers to the connection in subsequent calls.
- Information related to the connection is stored in a data structure called a **Transmission Control Block** (TCB).
- One implementation strategy would have the local connection name to be a pointer to the **TCB** for this connection.
- The OPEN call also specifies whether the connection establishment is to be actively pursued, or to be passively waited for.
- A passive OPEN request means that the process wants to accept incoming connection requests rather than attempting to initiate a connection.

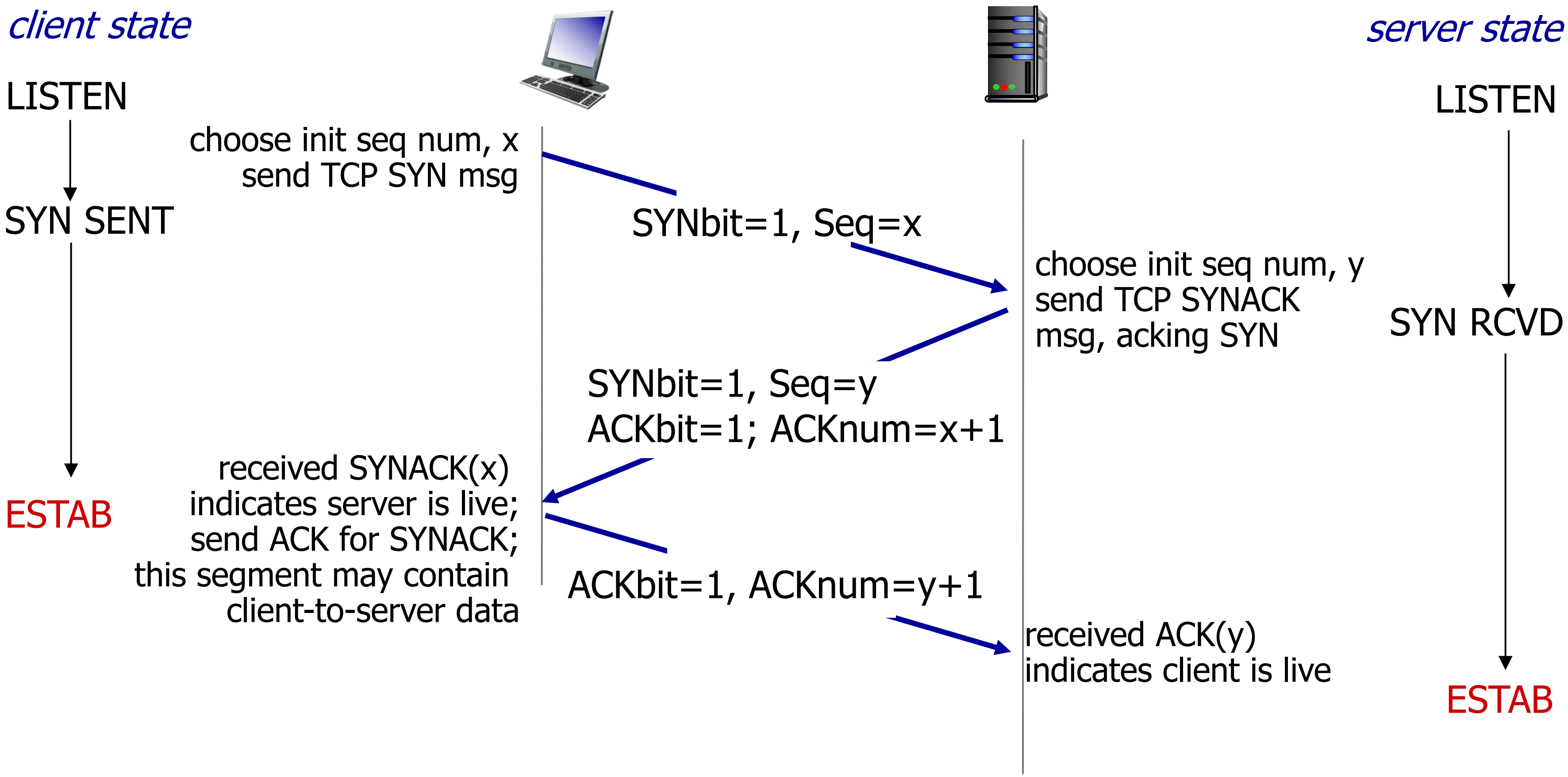
Connection Establishment

To establish a connection, the **three-way handshake** occurs:

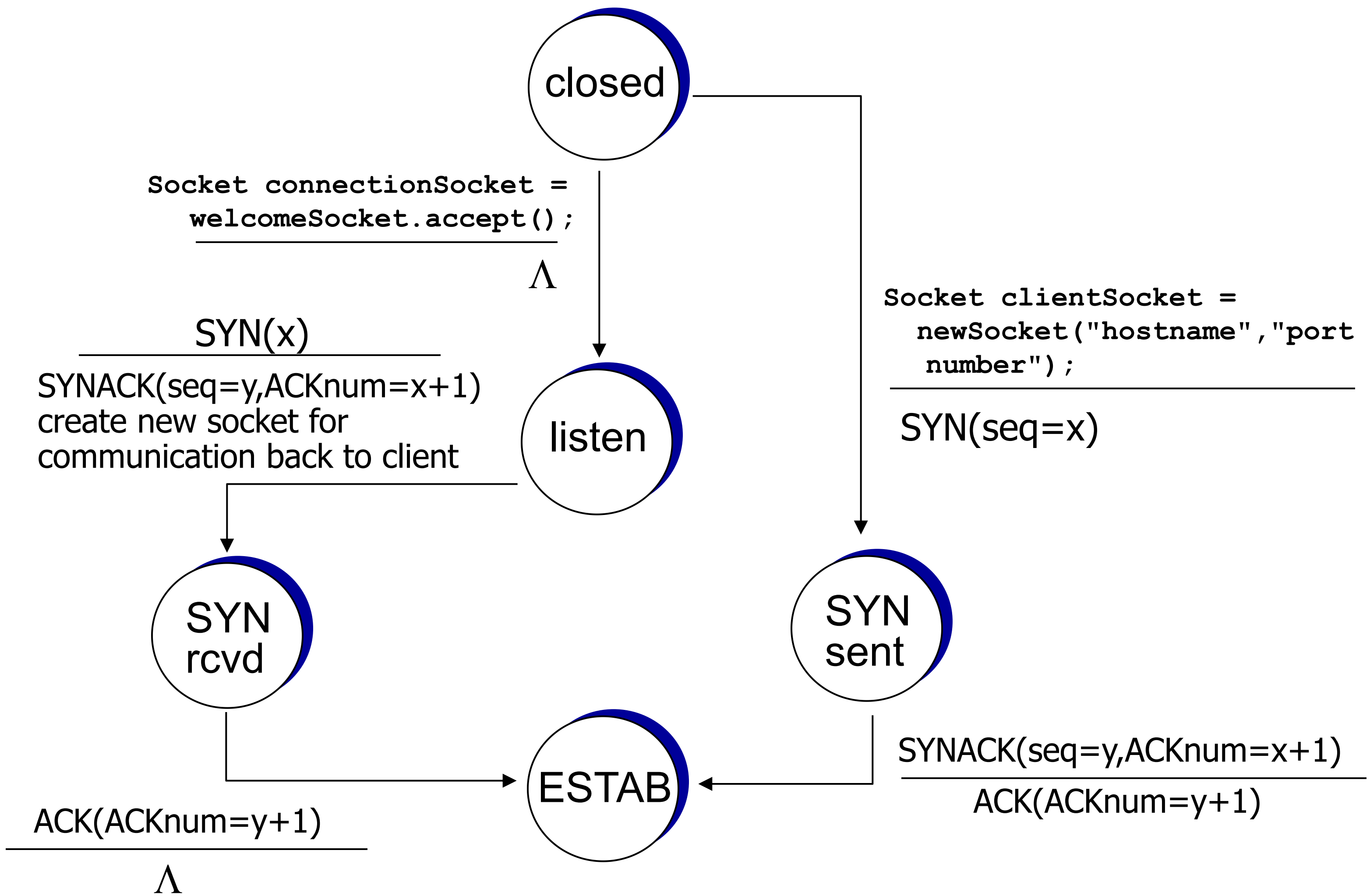
1. **SYN**: The active open is performed by the client sending a frame with SYN to the server. It sets the segment's sequence number to a random value A.
2. **SYN-ACK**: In response, the server replies with a SYN-ACK. The ack. number is set to one more than the received sequence number (A + 1), and the sequence number that the server chooses for the packet is another random number, B.
3. **ACK**: Finally, the client sends an ACK back to the server. The sequence number is set to the received acknowledgement value i.e. A + 1, and the ack. number is set to one more than the received sequence number i.e. B + 1.

At this point, both the client and server have received an acknowledgment that the connection has been established.

TCP 3-way handshake



TCP 3-way handshake: FSM



Active open example

- Consider the following frames from the **trainSuzhou.pcap** Wireshark capture:

```
11 4.090733  172.16.8.1      130.194.64.145 TCP    66 61981 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1260 WS=4 SACK_PERM=1
12 4.353041  172.16.8.1      130.194.64.145 TCP    66 61982 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1260 WS=4 SACK_PERM=1
13 4.757967  130.194.64.145 172.16.8.1      TCP    66 80 → 61981 [SYN, ACK] Seq=0 Ack=1 Win=50400 Len=0 MSS=1460 WS=1 SACK_PERM=1
14 4.758114  172.16.8.1      130.194.64.145 TCP    54 61981 → 80 [ACK] Seq=1 Ack=1 Win=66780 Len=0
```

- Frames 11, 13, 14 belong to the same TCP stream 61981 ↔ 80 and form the 3-way handshake

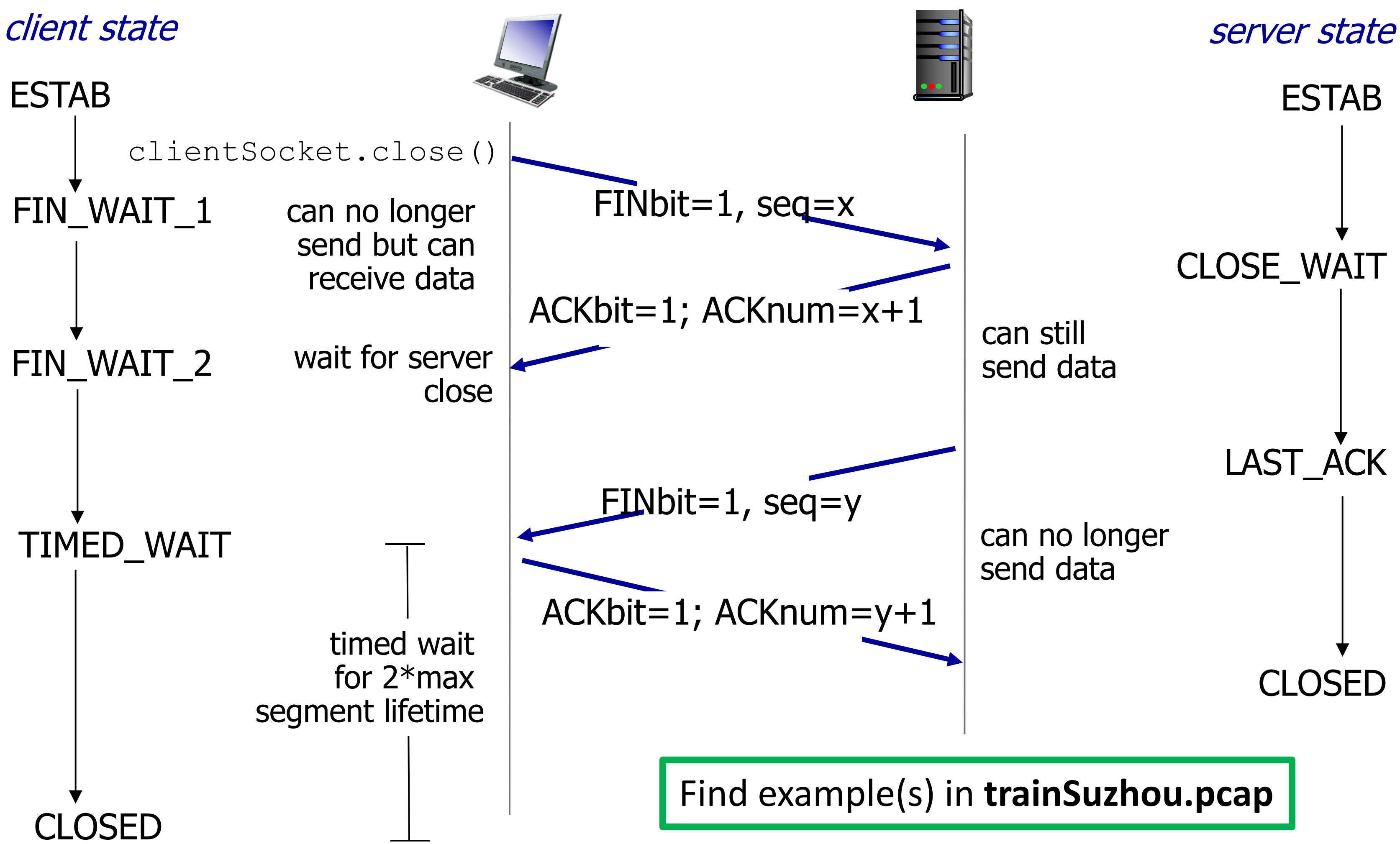
During the active open the sender and receiver negotiate:

- **MSS** (Maximum Segment Size). The default value from RFC 879 is MSS=536. The agreed value are MSS = 1260 and 1460 which is OK for what the link layer can handle.
- **Win**, the receive windows for the client and server for the purpose of the flow control. Note that both the client and server can be a sender and a receiver.

TCP: closing a 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

TCP: closing a connection



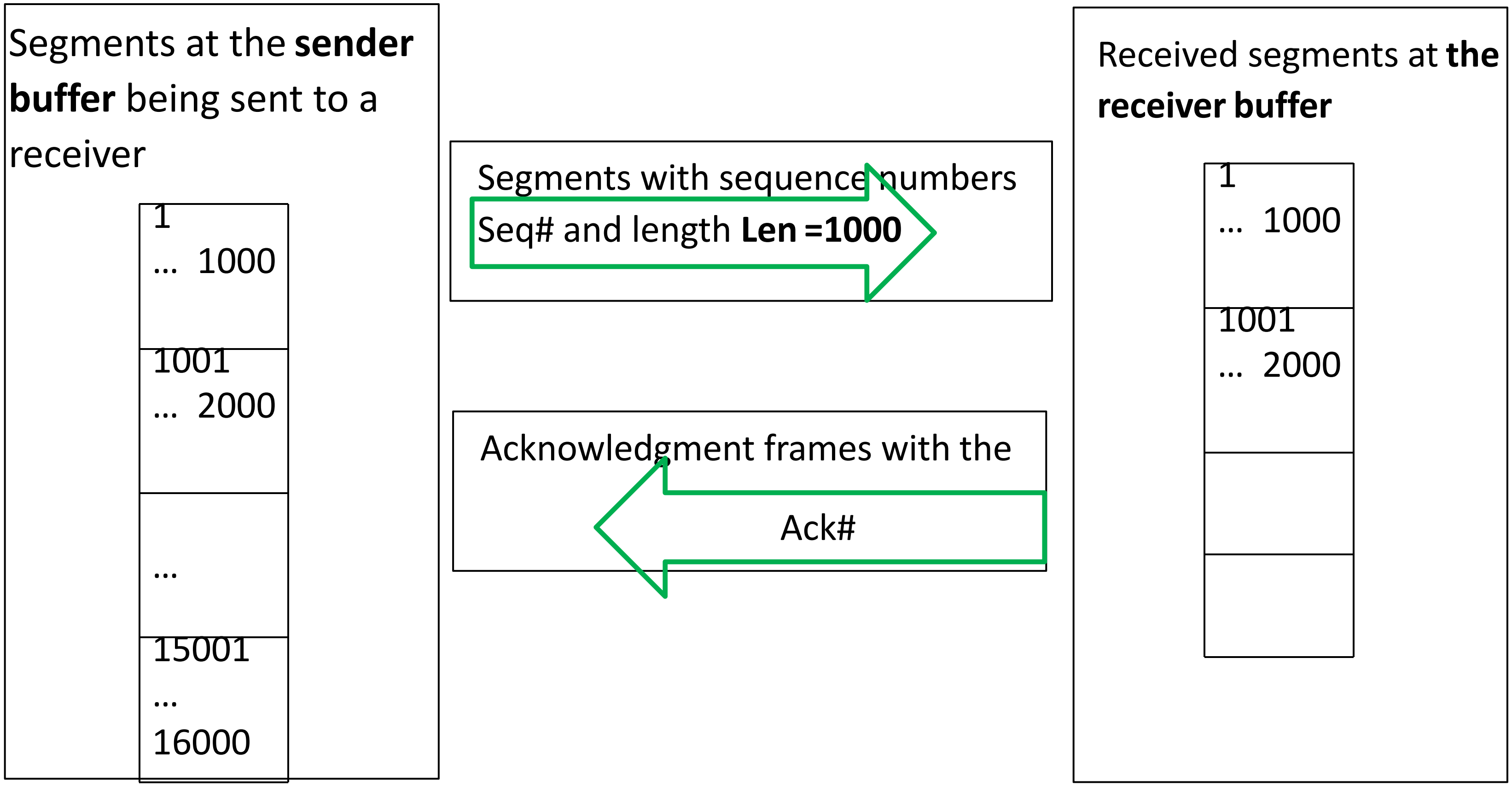
Find example(s) in **trainSuzhou.pcap**

Principles of the Reliable Data Transfer (1)

- The TCP is able to transfer through the internet system a **continuous stream of bytes** packed into **segments** in each direction between its users.
- Transmission is made reliable via the use of sequence numbers and acknowledgments.
- The TCP is able to recover from data that is **damaged**, **lost**, **duplicated**, or delivered **out of order.**
- Conceptually, each byte of data is assigned a **sequence number**.
- The sequence number of the first byte of data in a segment is transmitted with that segment and is called the **segment sequence number**.
- Segments also carry an **acknowledgment number** which is the sequence number of the **next expected data byte** of transmissions in the reverse direction.

Top view of the TCP in working

- Assume that the sender has a data to be sent organized into segments, each segment of the **Len = 1000** size, Len ≤ MSS



- The sender sends segments **continuously** without waiting for acknowledgment frames from the receiver.
- The receiver acknowledges the received frames **cumulatively,** at its convenience.

TCP Data transfer

sequence numbers:

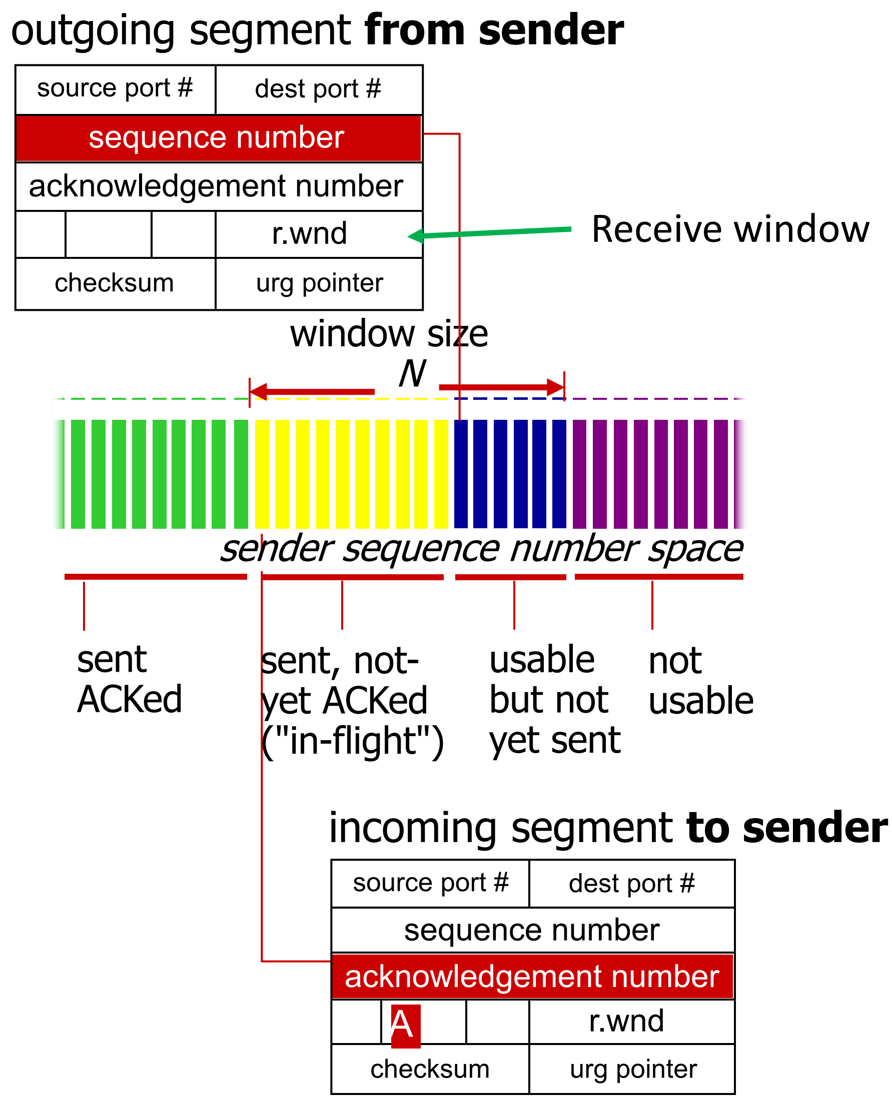
- byte stream "number" of the first byte in segment's data

acknowledgements:

- seq# of next byte expected from the other side
- cumulative ACK (of received bytes)

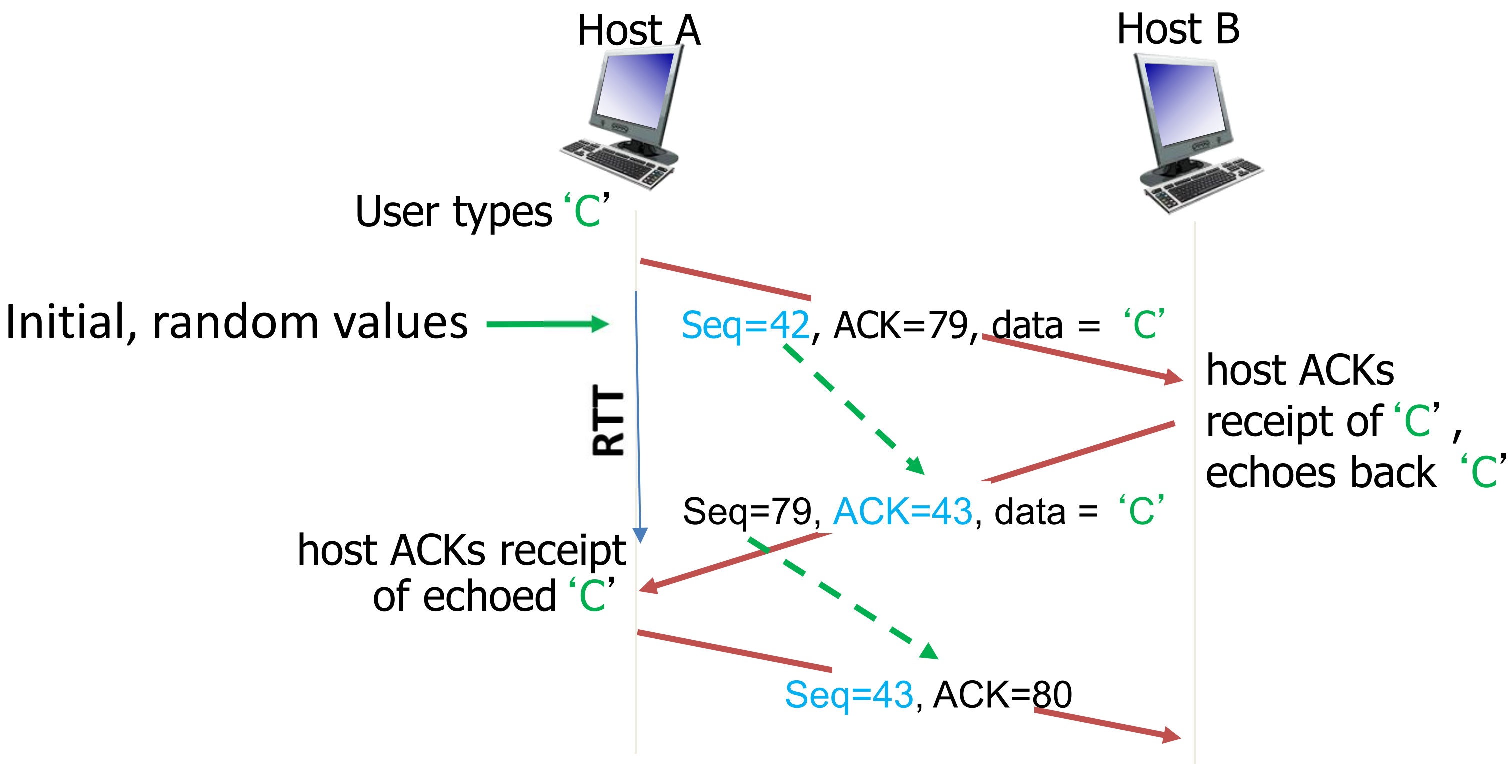
Q: how a receiver handles out-of-order segments

A: TCP spec does not say, up to implementer



simple telnet scenario (sending one character and echoing it back):

Introducing RTT – Round Trip Time



TCP Round Trip Time (RTT), 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

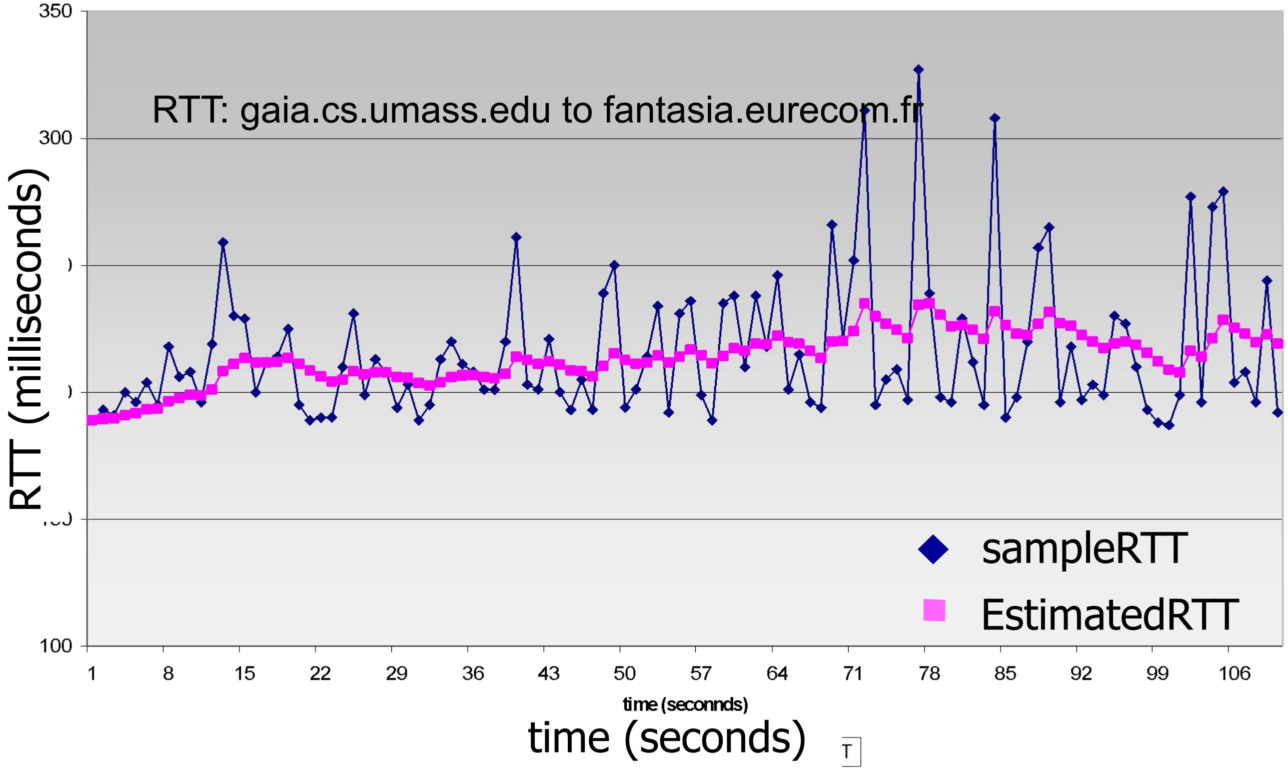
Q: how to estimate RTT?

- **`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 (RTT)

EstimatedRTT = (1-α)*EstimatedRTT + α*SampleRTT

- **exponential weighted moving average**
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** → larger safety margin
- estimate **SampleRTT** deviation from **EstimatedRTT** :

```
DevRTT = (1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT

"safety margin"

More on TCP reliable data transfer

- TCP creates Reliable Data Transfer service on top of IP's unreliable service. Note:
  - pipelined segments
  - cumulative ACKs
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate ACKs

Let's initially consider simplified TCP sender:

- ignore duplicate ACKs
- ignore flow control and congestion control

TCP sender events:

data rcvd from app:

- 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`**

timeout:

- retransmit segment that caused timeout
- restart timer

ACK rcvd:

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

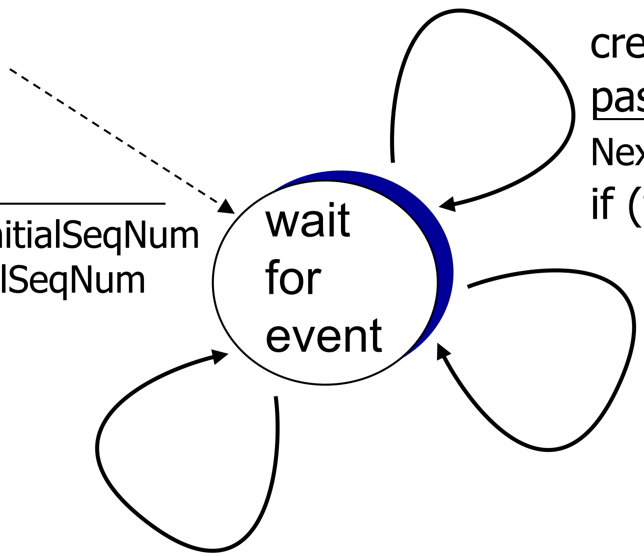
TCP sender (simplified, optional).

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



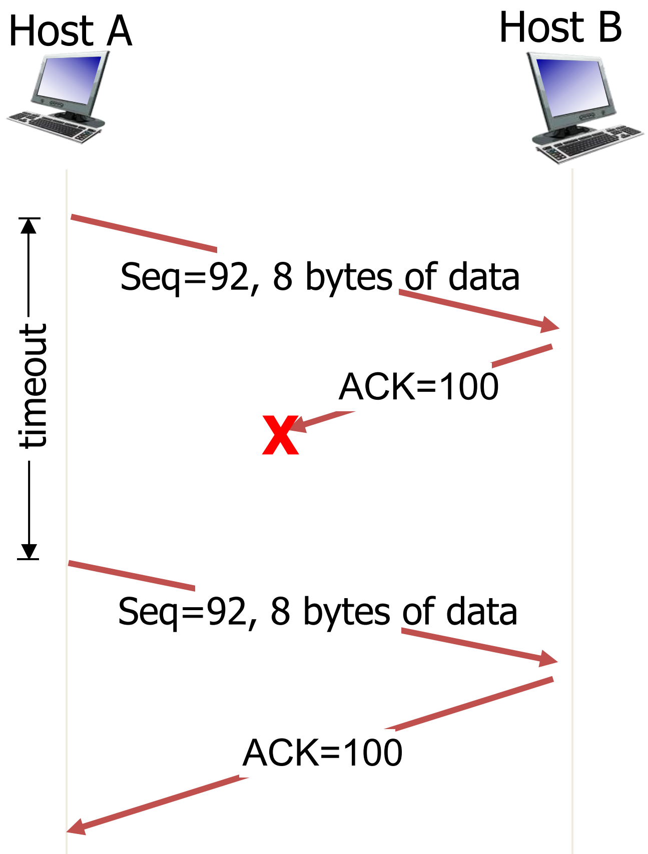
timeout

retransmit not-yet-ACKed segment
    with smallest seq. #
start timer

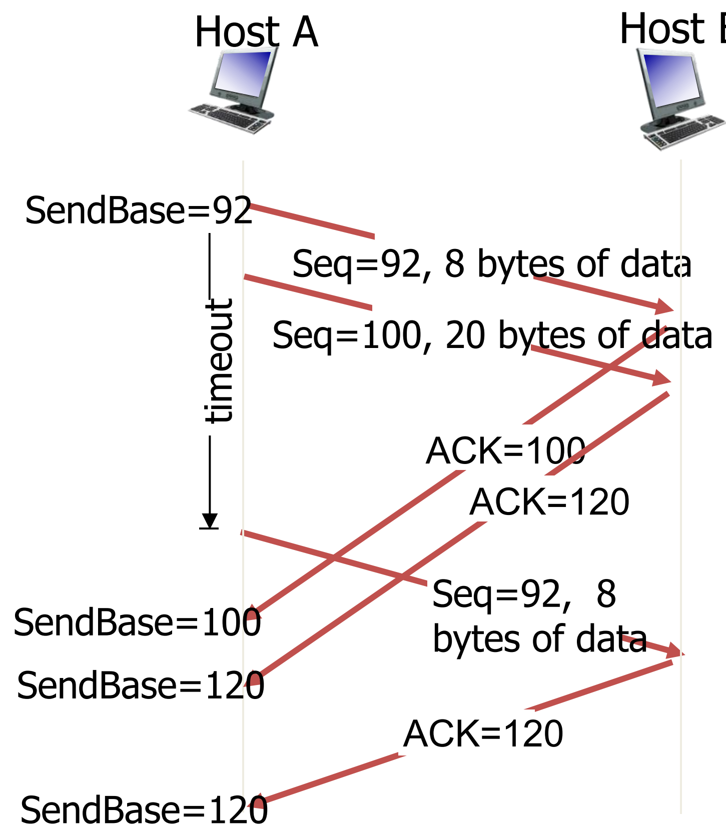
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-ACKed segments)
        start timer
    else stop timer
 }
```

TCP: retransmission scenarios

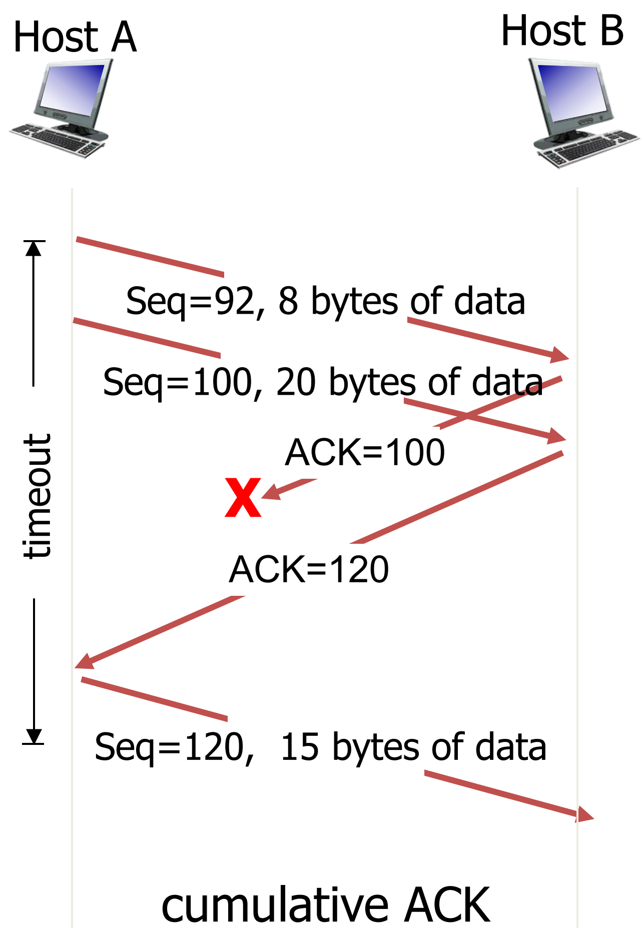


lost ACK scenario:
no ACK within timeout.
Retransmission



premature timeout
Retransmission on timeout
After timeout two “old” ACKs arrived
Retransmitted bytes are ACKed

TCP: retransmission scenarios



ACK = 100 has been lost. However the sender is not aware of this fact and also received ACK = 120 (expect byte # 120) within the timeout. Therefore, the sender sends n bytes starting from 120.
Lost ACK has been harmlessly ignored.

TCP ACK generation [RFC 1122, RFC 2581]

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
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

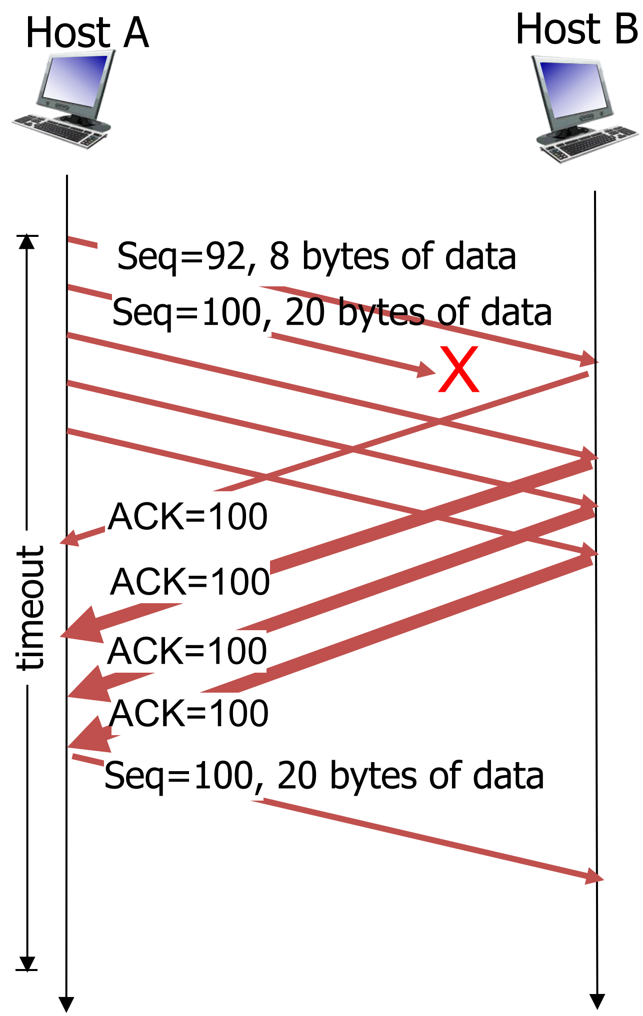
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives **3x2 ACKs** for same data ("triple duplicate ACKs"), resend unACKed segment with smallest seq #

- likely that unACKed segment lost, so don't wait for timeout

TCP fast retransmit

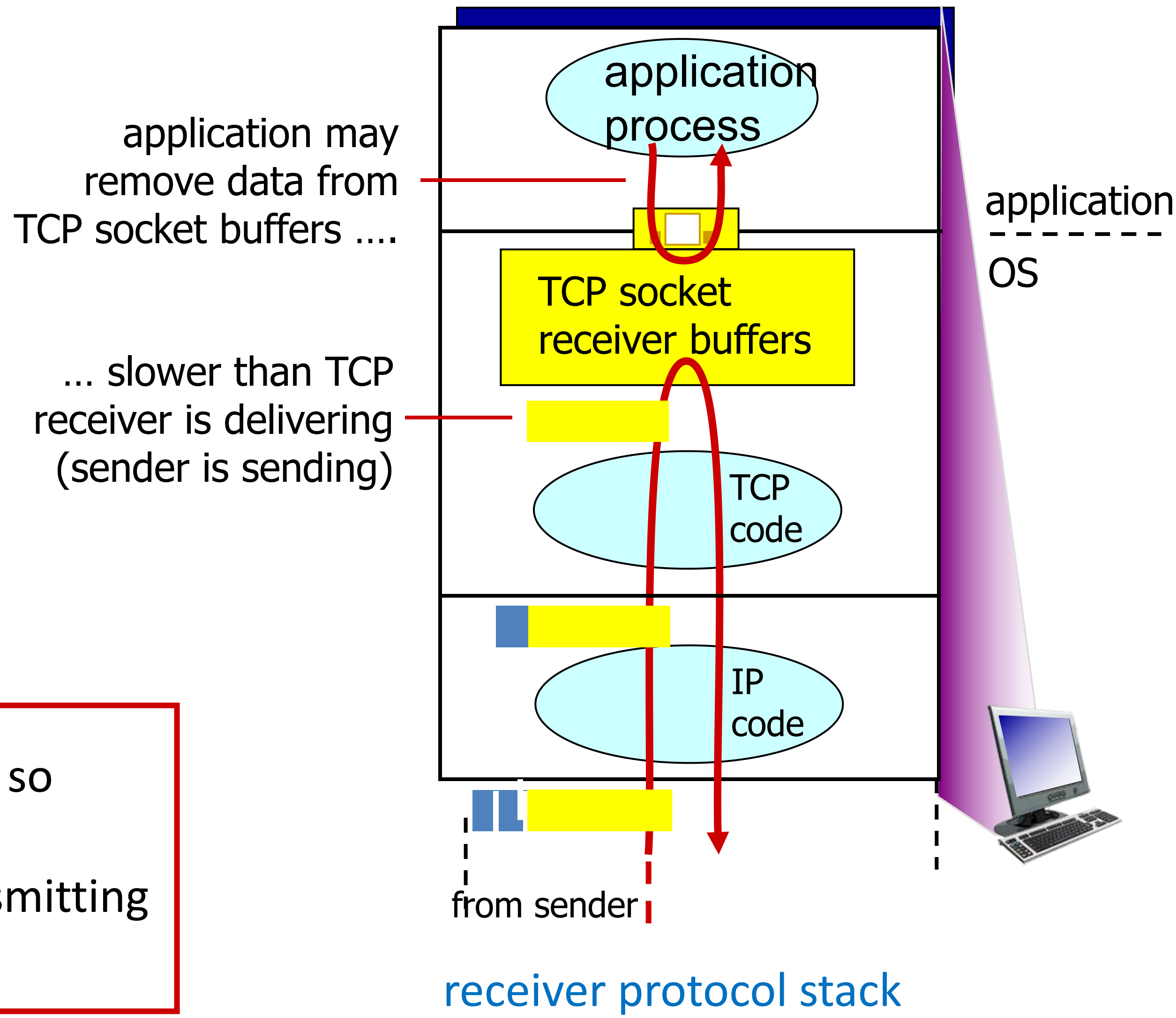


8 bytes from 92 have been received and ACKed by ACK =100
Then 20 bytes from 100 are lost.
Sender keeps sending the next chunks of data.
Receiver noticed the gap and initiates the fast retransmission (inside a timeout) by repeating ACK =100 **six times** in response to incoming data.
Sender, after receiving triple duplicate ACK, retransmit 20 bytes starting from 100.

Principles of the Flow Control (from RFC 793)

- The flow control mechanism in TCP provides a means for the receiver to govern the amount of data sent by the sender
- This is achieved by the **receiving** TCP reporting a **window with every ACK** to the **sending** TCP.
- This window specifies the number of bytes, starting with the acknowledgment number, that the receiving TCP is currently prepared to receive.

TCP flow control

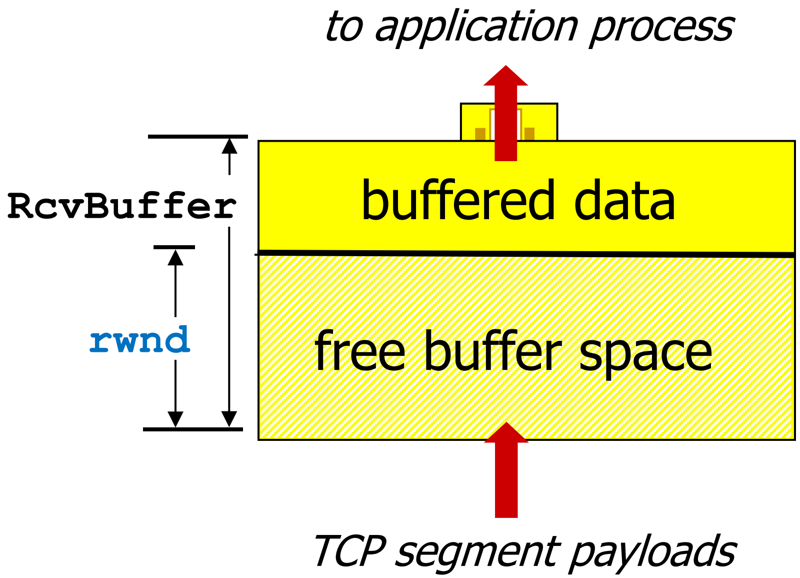


flow control

receiver controls sender, so sender will not overflow receiver's buffer by transmitting too much, too fast

TCP flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - An operating system can autoadjust **`RcvBuffer`**
- sender limits amount of unACKed ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow



receiver-side buffering

Enough for one lecture?

- Congestion control moved to the next lecture.

TCP congestion control summary:
Additive Increase, Multiplicative Decrease (AIMD)

- *approach:* sender increases transmission rate (congestion window size **cwnd**), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **cwnd** (congestion window size) by 1 MSS (maximum segment size) every RTT until loss detected
  - *multiplicative decrease*: cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth