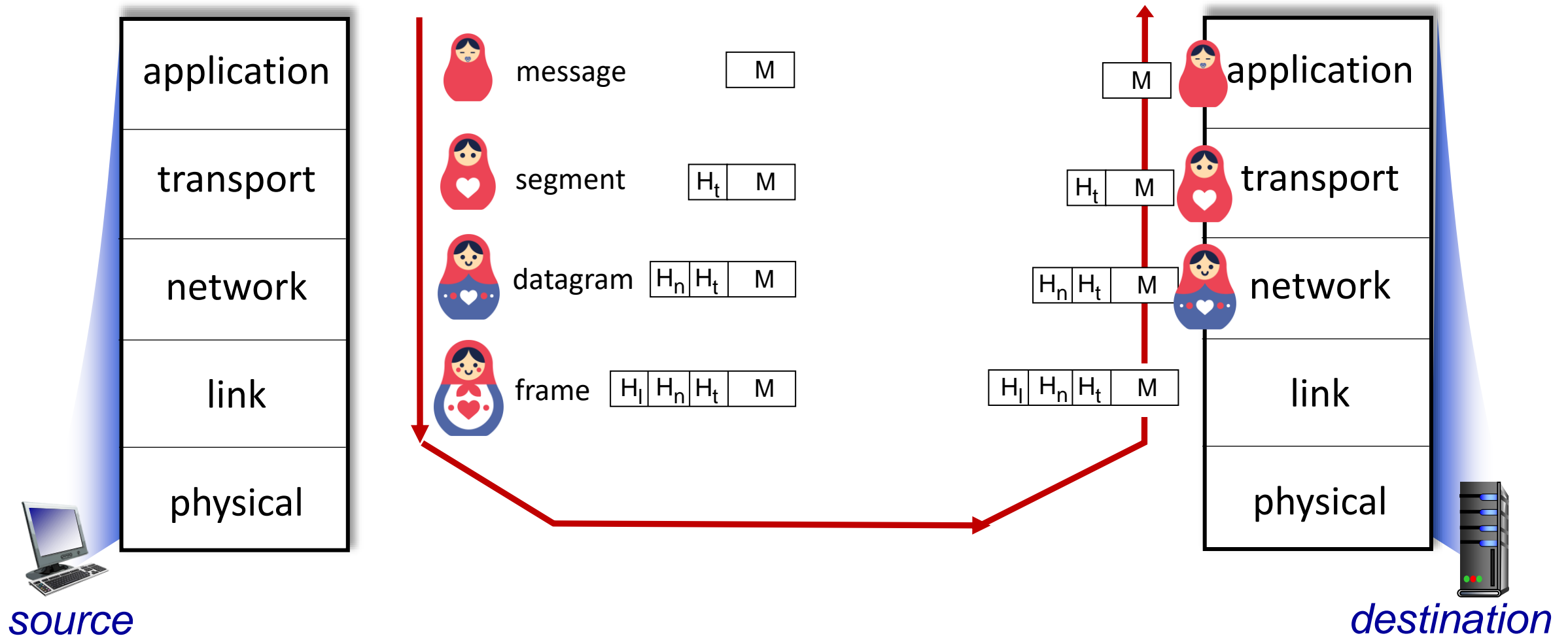


UNIT – III

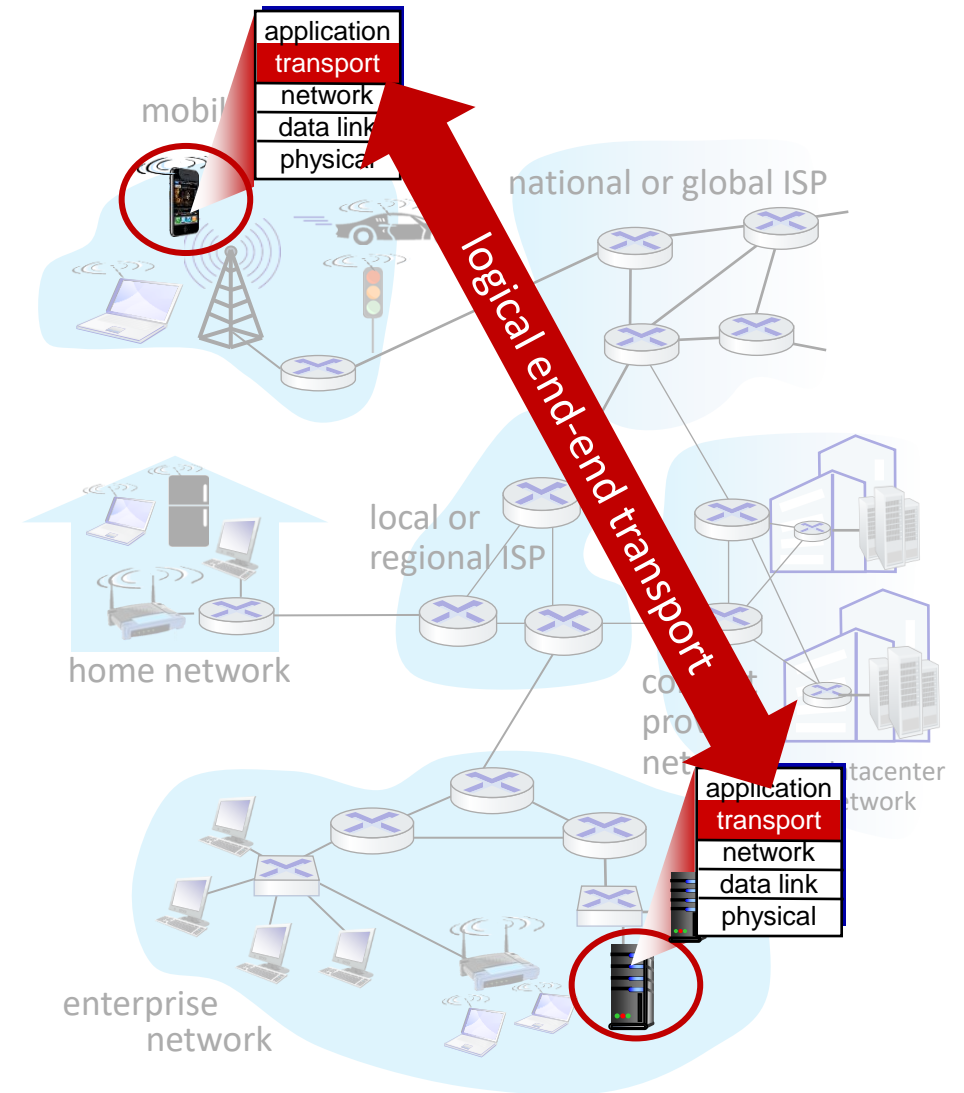
- **Network Layer:** Switching, Logical addressing - IPV4, IPV6; Address mapping - ARP, RARP, BOOTP and DHCP-Delivery, Forwarding and Unicast Routing protocols
- **Transport Layer:** Process to Process Communication, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), SCTP Congestion Control; Quality of Service (QoS), QoS improving techniques - Leaky Bucket and Token Bucket algorithms

Services, Layering and Encapsulation



Transport services and protocols

- provide *logical communication* between application processes running on different hosts
- transport protocols actions in end systems:
 - sender: breaks application messages into *segments*, passes to network layer
 - receiver: reassembles segments into messages, passes to application layer
- two transport protocols available to Internet applications
 - TCP, UDP



Transport vs. network layer services and protocols

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- Application messages = letters in envelopes
- processes = kids
- transport protocol = Ann and Bill who mux and demux to in-house kids
- hosts = houses
- network-layer protocol = postal service

Transport vs. network layer services and protocols

- **Transport layer:** communication between *processes*
 - relies on network layer services
- **Network layer:** communication between *hosts*

Process-to-Process Communication

- Process-to-process communication in the transport layer refers to the exchange of data between running applications on different hosts within a network.
- The transport layer ensures reliable, efficient, and ordered data delivery between these processes.

Overview of the Transport Layer

The **transport layer (Layer 4)** of the **OSI model** and **TCP/IP model** provides end-to-end communication services. It enables **processes** (running applications) to communicate by establishing logical connections.

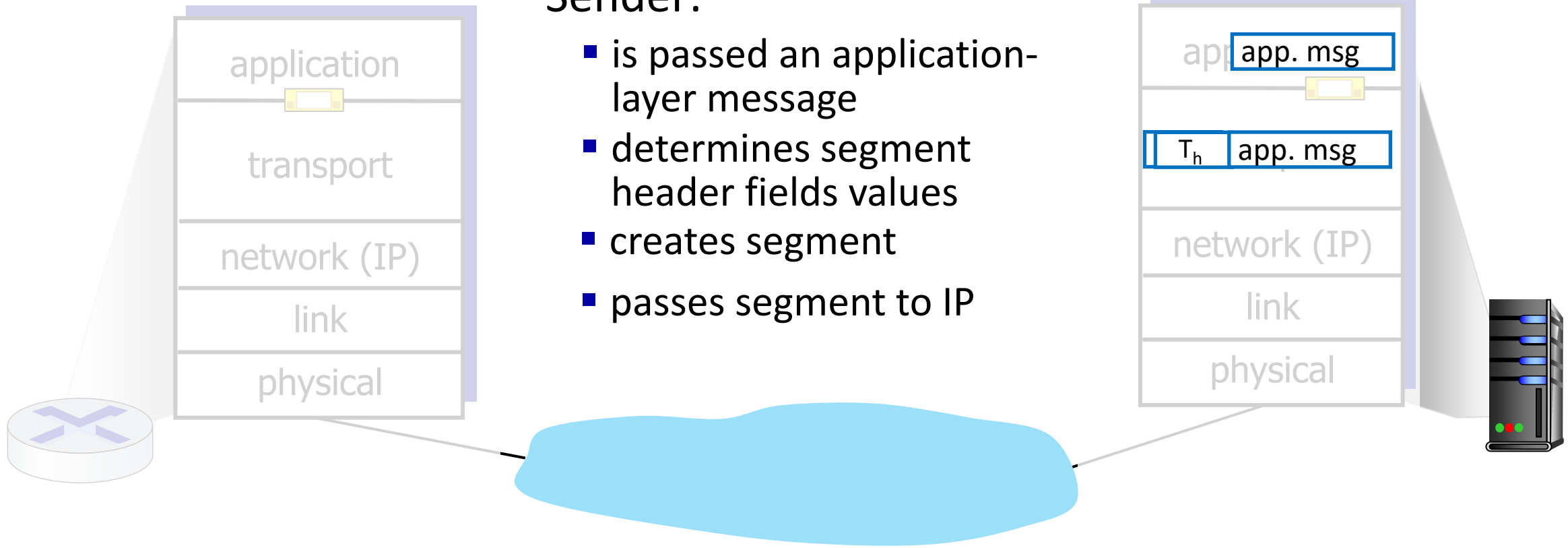
Key Responsibilities of the Transport Layer:

- Process addressing (Multiplexing & Demultiplexing)
- Reliability (Error detection & correction, flow control, congestion control)
- Segmentation and reassembly
- Connection management

Transport Layer Actions

Sender:

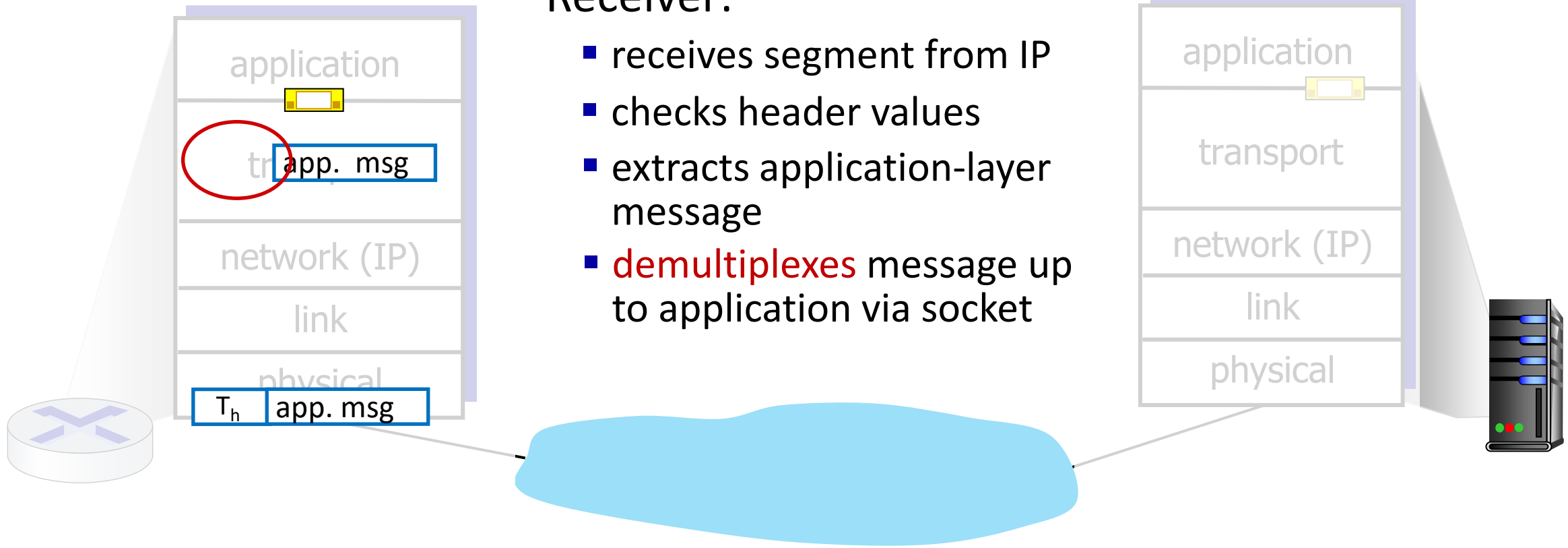
- is passed an application-layer message
- determines segment header fields values
- creates segment
- passes segment to IP



Transport Layer Actions

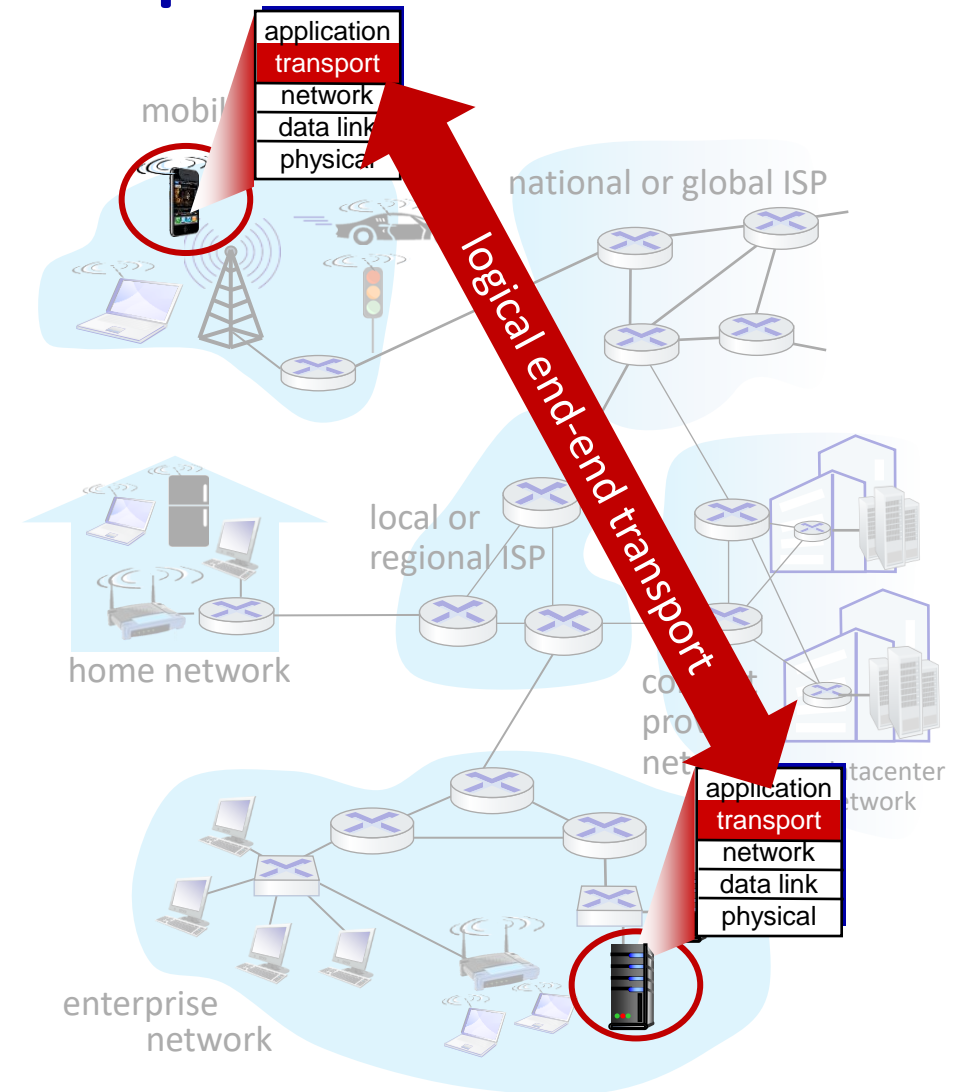
Receiver:

- receives segment from IP
- checks header values
- extracts application-layer message
- **demultiplexes** message up to application via socket

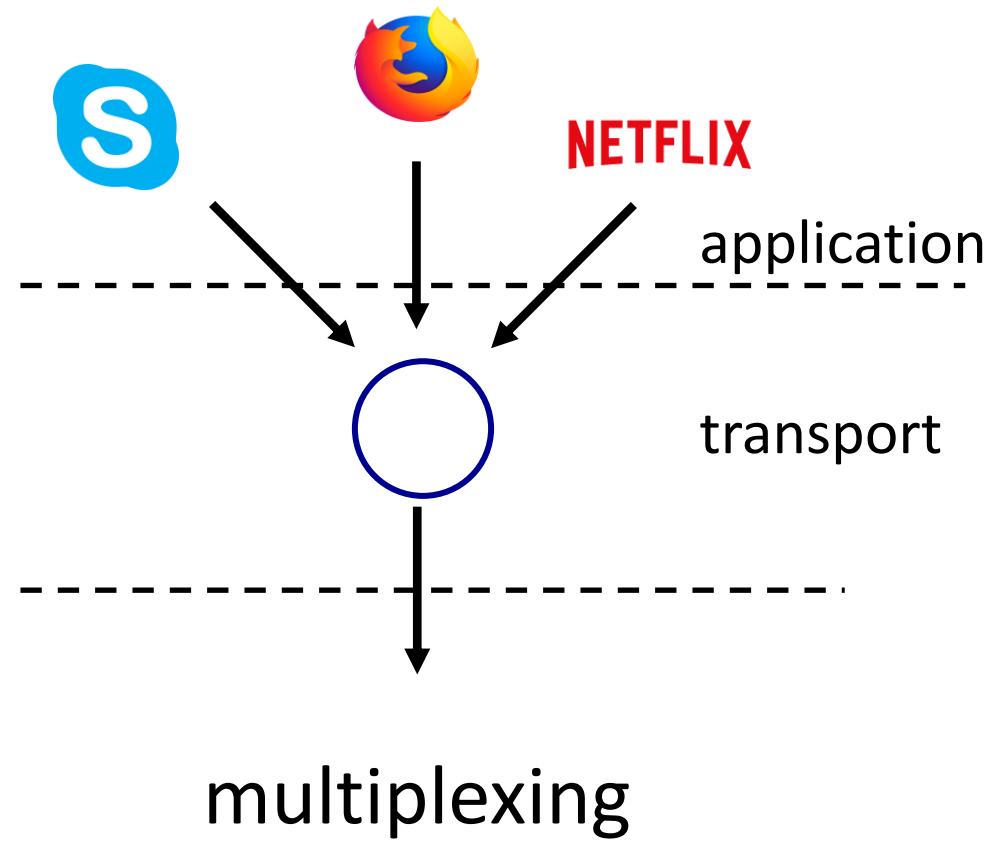


Two principal Internet transport protocols

- **TCP:** Transmission Control Protocol
 - reliable, in-order delivery
 - congestion control
 - flow control
 - connection setup
- **UDP:** User Datagram Protocol
 - unreliable, unordered delivery
 - no-frills extension of “best-effort” IP
- services *not* available:
 - delay guarantees
 - bandwidth guarantees



Why IP service model is a best effort delivery service?





Multiplexing

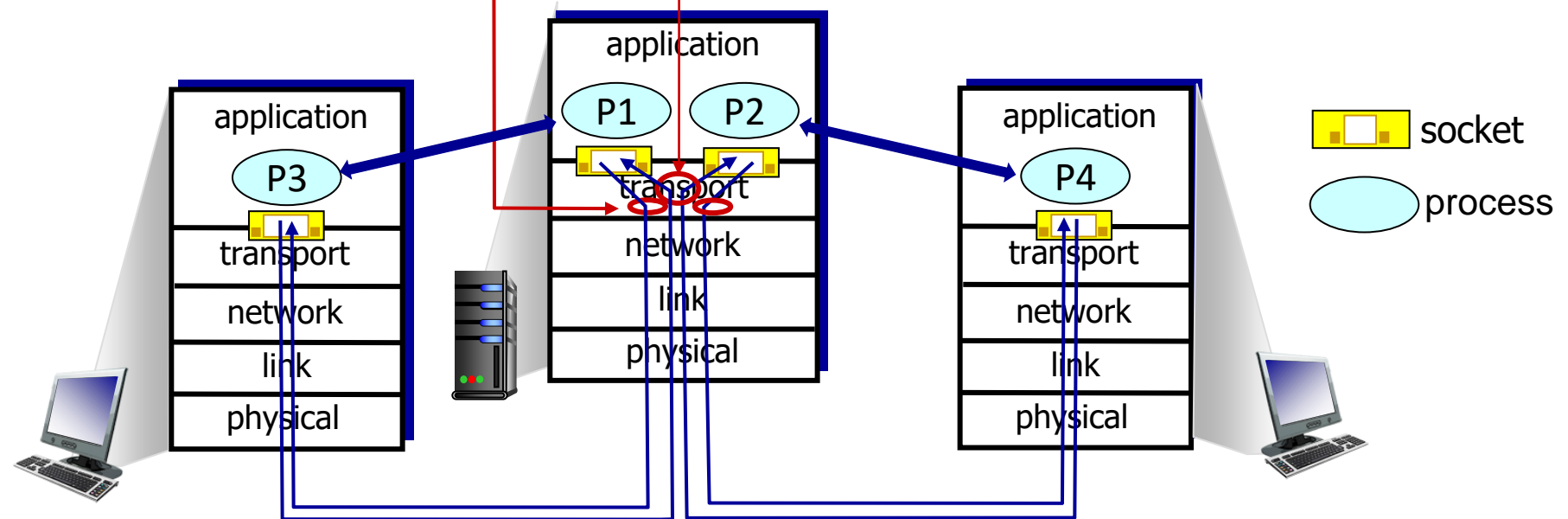
Multiplexing/demultiplexing

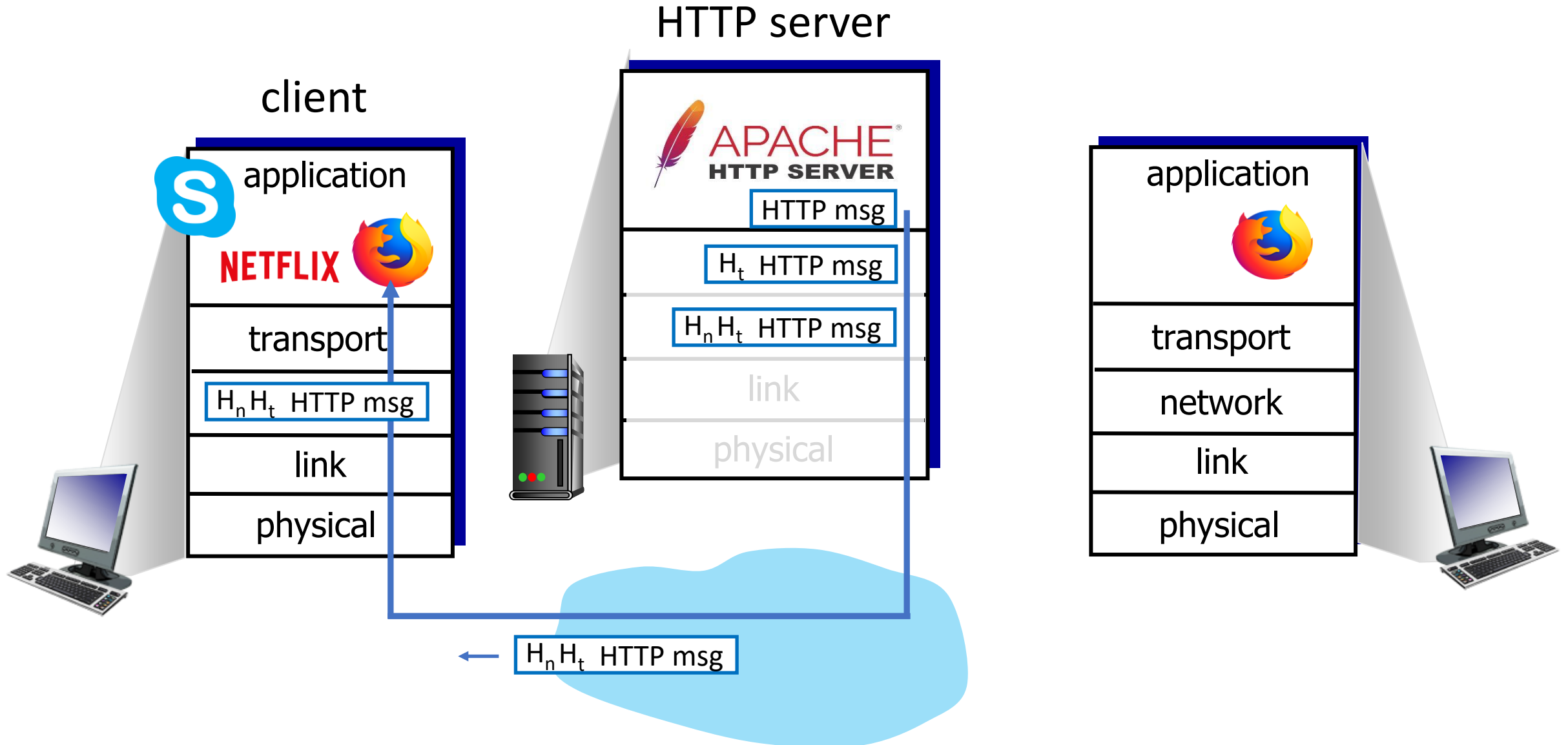
multiplexing as sender:

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

demultiplexing as receiver:

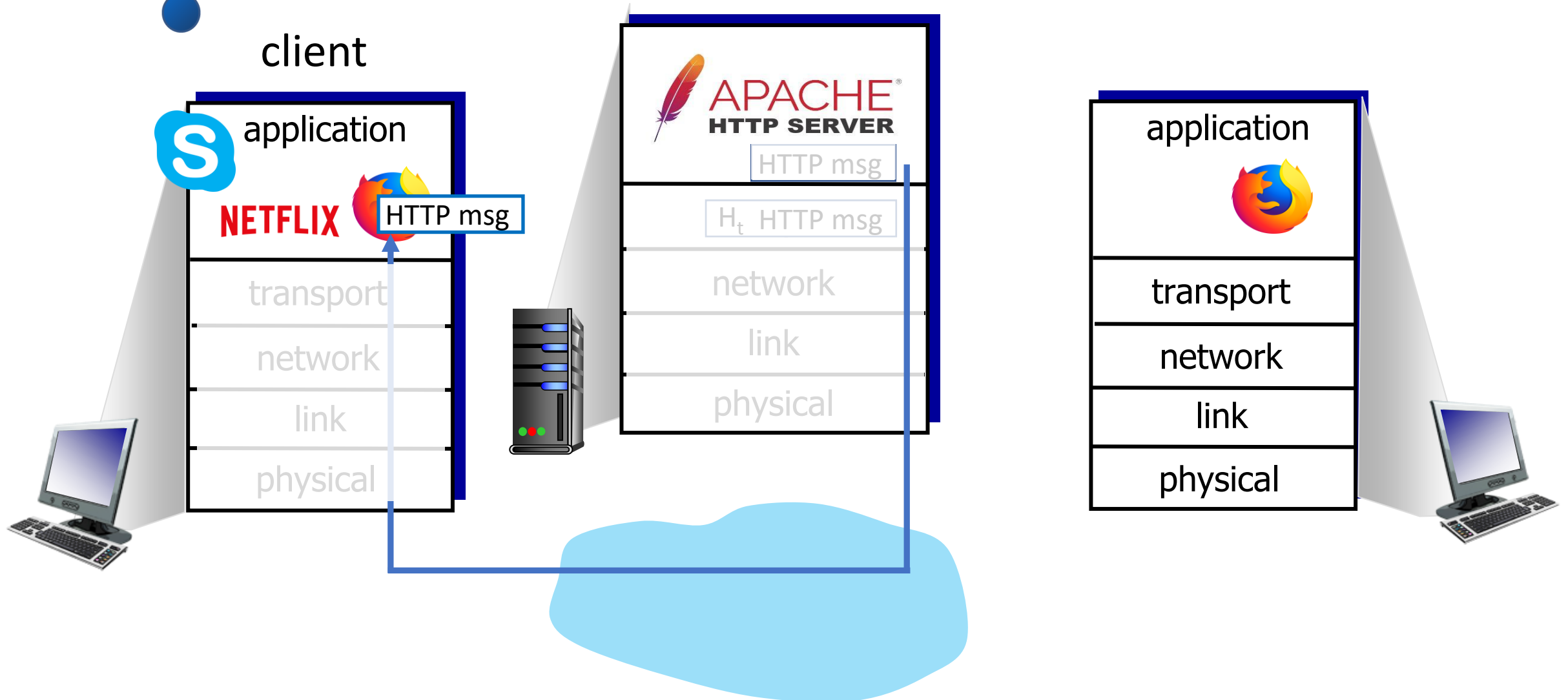
use header info to deliver received segments to correct socket





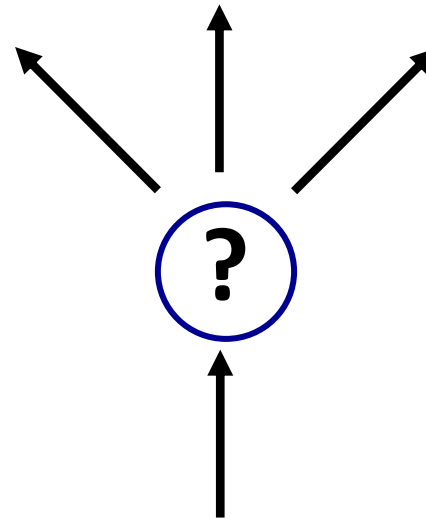


Q: how did transport layer know to deliver message to Firefox browser process rather than Netflix process or Skype process?

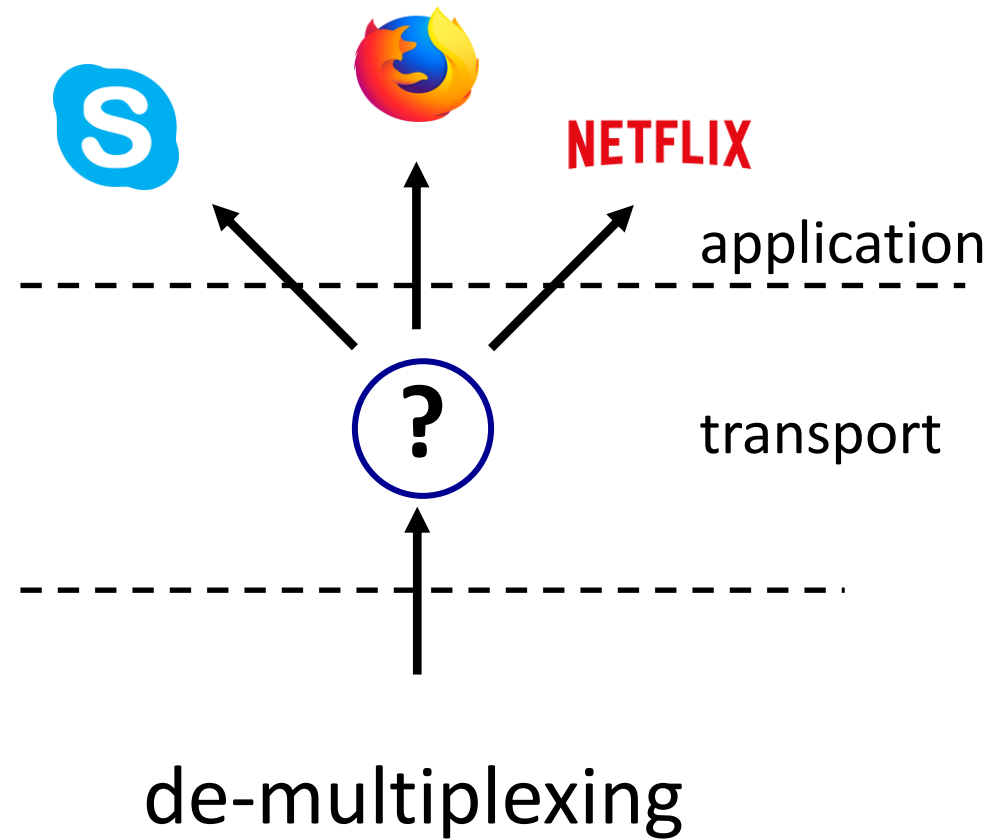


Solution

- The transport layer (usually using protocols like TCP or UDP) works with the network layer to ensure data is transmitted to the correct destination.
- When a packet reaches the destination machine, the transport layer uses a combination of the IP address and a port number to determine where the data should go.
- IP Address: Ensures the packet reaches the correct device on the network.
- Port Number: Ensures the packet reaches the correct application/process on that device.
- Each application or service running on a machine is associated with a specific port number.
- For example, web browsers like Firefox often use port 80 (HTTP) or 443 (HTTPS).
- Skype and other applications will use different ports.
- When the transport layer receives the packet, it checks the destination port number included in the packet's header.
- Based on this port number, the transport layer can determine which application (or process) on the machine is supposed to receive the data.
- So, in this case, the message was destined for a specific port that the Firefox browser was using, and that's how it knew to deliver the message to Firefox rather than to the Netflix or Skype processes.



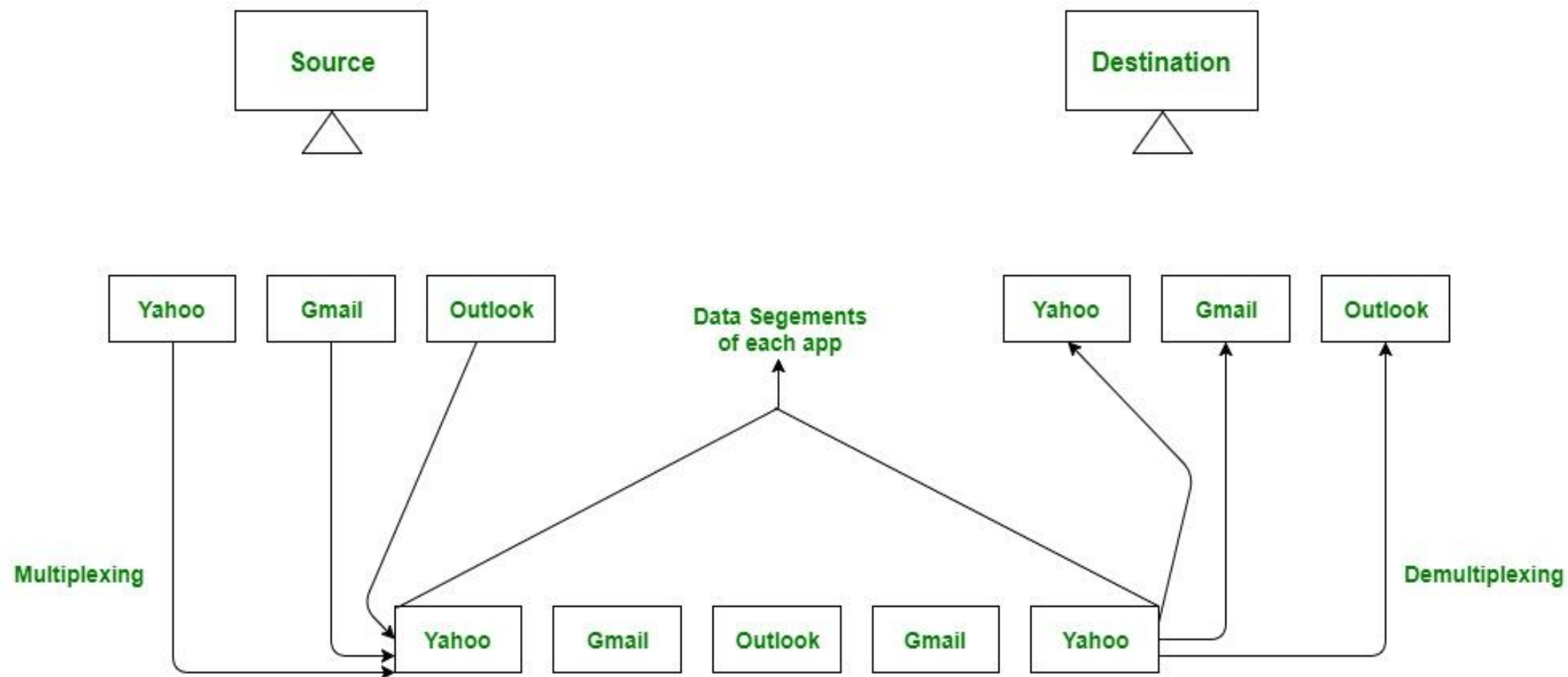
de-multiplexing





Demultiplexing

Multiplexing / Demultiplexing



A's SmartPhone [Sender]

B's SmartPhone [Receiver]

Whatsapp
(Port No. 30)

Hike
(Port No. 40)

Source IP : A

Destination IP : B

Source Port No. : 30

Destination Port No. : 50

Message 1

Source IP : A

Destination IP : B

Source Port No. : 40

Destination Port No. : 60

Message 2

Whatsapp
(Port No. 50)

Hike
(Port No. 60)

Message 1

Message 2

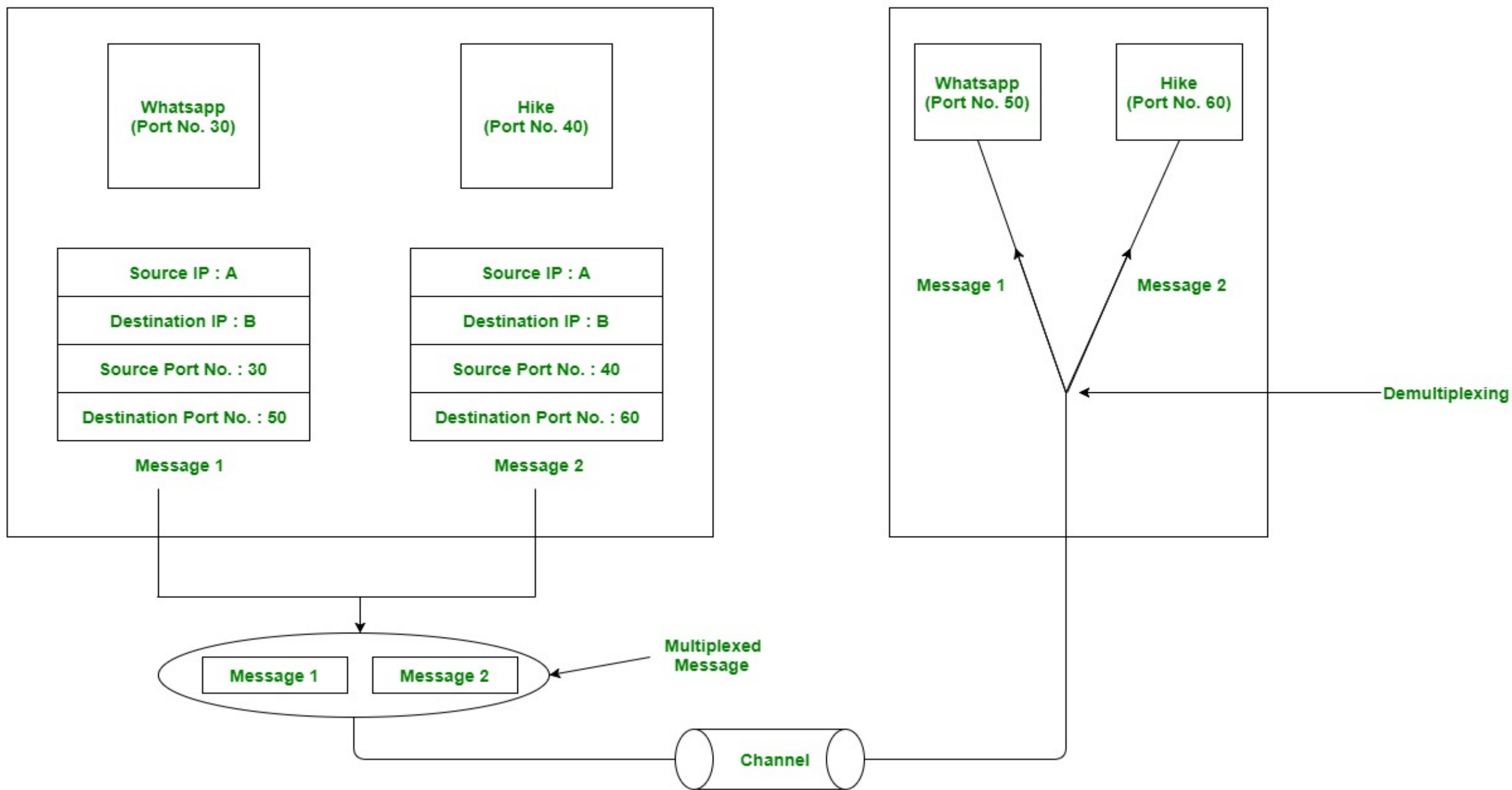
Demultiplexing

Message 1

Message 2

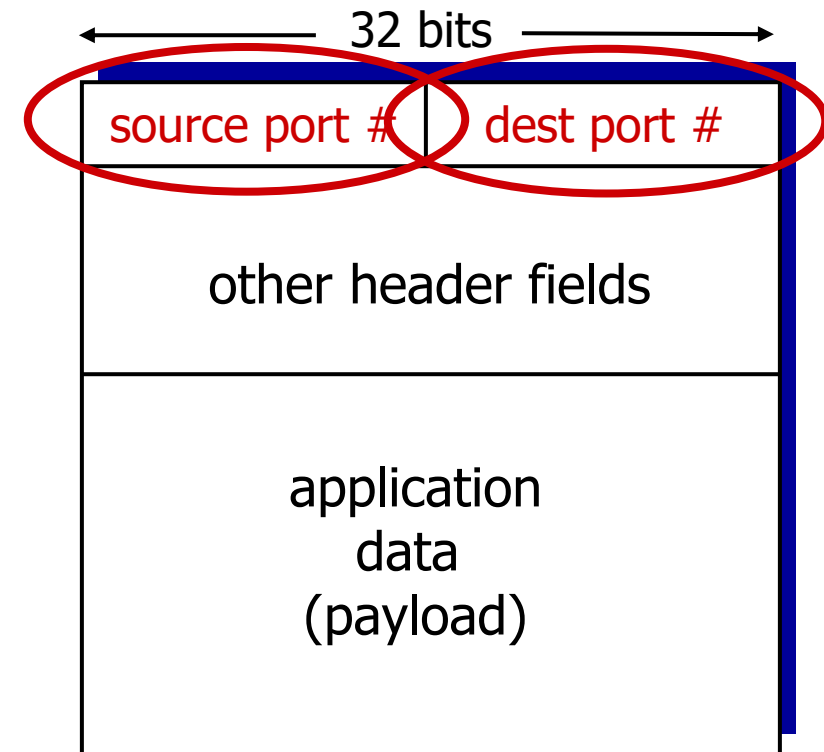
Multiplexed
Message

Channel



How demultiplexing works

- host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one transport-layer segment
 - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing[UDP]

Recall:

- when creating socket, must specify *host-local* port #:

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

- when creating datagram to send into UDP socket, must specify
 - destination port #

when receiving host receives *UDP* segment:

- checks destination port # in segment
- directs UDP segment to socket with that port #



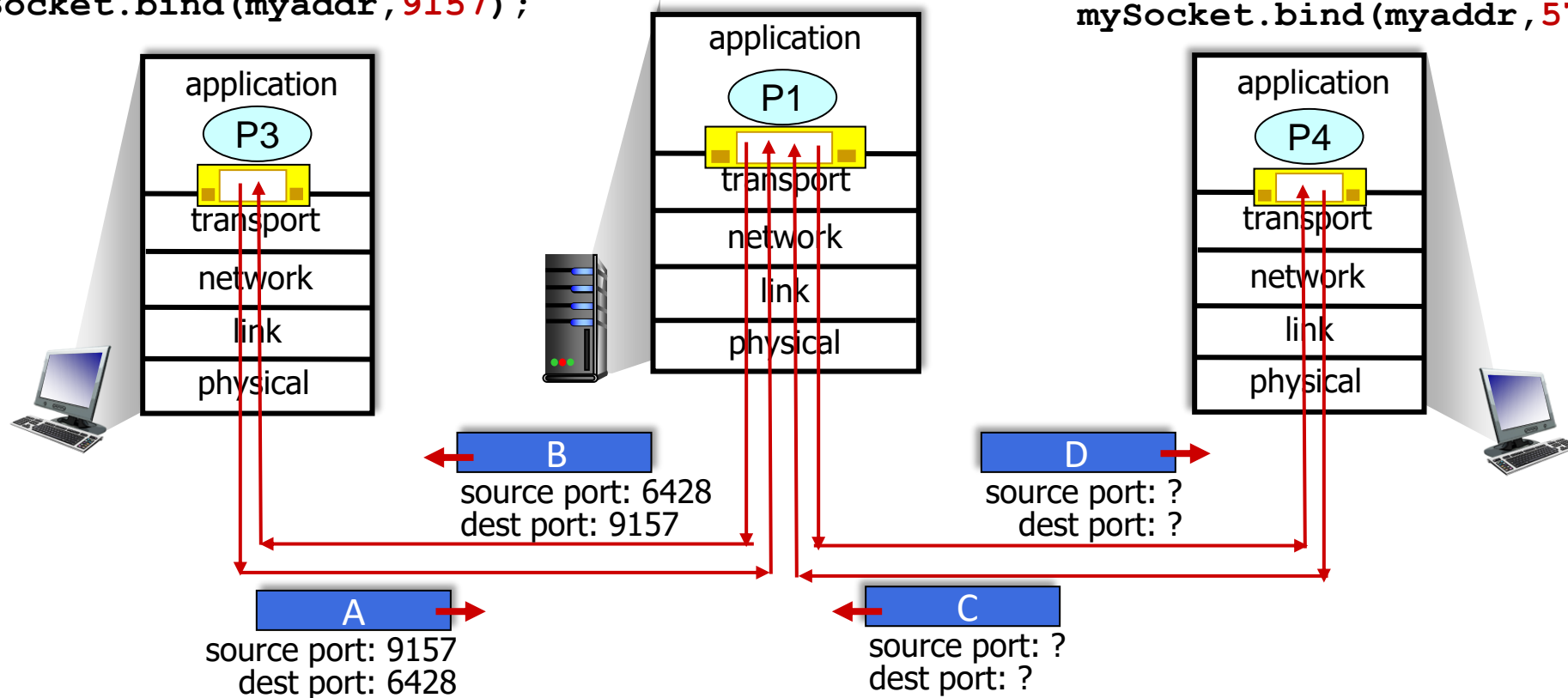
IP/UDP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at receiving host

Connectionless demultiplexing: an example

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

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

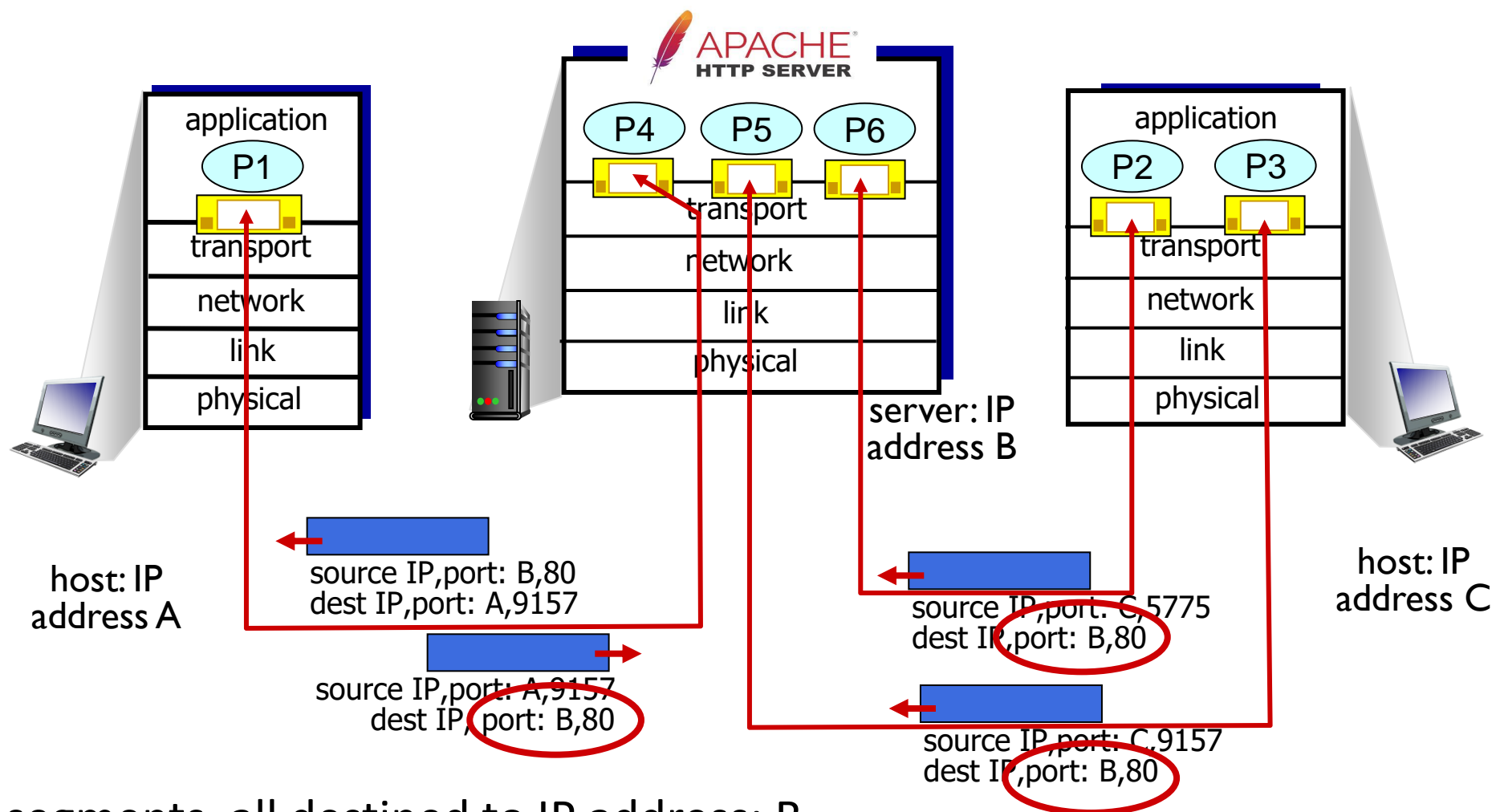
```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



Connection-oriented demultiplexing[TCP]

- TCP socket identified by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket
- server may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
 - each socket associated with a different connecting client

Connection-oriented demultiplexing: example



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

Think about this!!!

- If two applications from the same source IP address are using the same source port number to send data to the same destination port on the same destination IP address, what will happen?

Solution

- The source port number needs to be unique for each outgoing connection from the same source IP address.
- **Most operating systems will not allow two applications to bind to the same source port number at the same time if they are both trying to use that port for outgoing connections.** Typically, the OS will assign a different ephemeral (temporary) source port number to each connection if the applications do not specify one.
- In practice, TCP relies on the uniqueness of the source port for demultiplexing. If two applications attempt to use the same source port number, the operating system will generally prevent this situation by assigning different ephemeral ports or rejecting the binding request, thereby avoiding any potential conflicts.

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

User Datagram Protocol (UDP)

UDP

- UDP is a **fast, lightweight, and efficient transport layer protocol best suited for real-time, low-latency applications.**
- Unlike TCP, it does **not guarantee reliable delivery**, but it **outperforms TCP in speed-sensitive scenarios** like gaming, video streaming, and VoIP.
- UDP provides a mechanism to detect corrupt data in packets, but it does *not* attempt to solve other problems that arise with packets, such as lost or out of order packets.
- That's why UDP is sometimes known as the ***Unreliable Data Protocol***.
- UDP is simple but fast.
- It's often used for time-sensitive applications (such as real-time video streaming) where ***speed is more important than accuracy***.

User Datagram Protocol (UDP) - Best Effort Communication

UDP provides **unreliable, connectionless communication** for speed-sensitive applications.

Features:

- **No connection establishment:** Data is sent without a handshake.
- **No reliability:** No acknowledgment, retransmission, or error correction.
- **Low overhead & fast:** Ideal for real-time applications.

UDP Process Communication Example

UDP is widely used in **real-time and streaming applications** where losing some data is acceptable.

A. Domain Name System (DNS) Query Example

- A client sends a **DNS request** to a server using UDP (port 53).
- The server responds with the requested **IP address**.
- No connection setup or acknowledgment is needed.

B. Video Streaming (YouTube, Netflix, VoIP)

- Packets are sent **without retransmission**, avoiding delays.
- If a few packets are lost, the video or voice **continues playing smoothly**.

UDP: User Datagram Protocol [RFC 768]

	INTERNET STANDARD
RFC 768	J. Postel ISI 28 August 1980

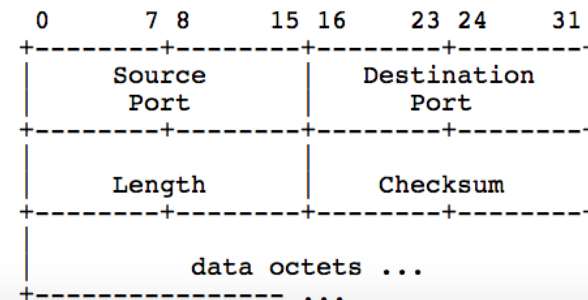
User Datagram Protocol

Introduction

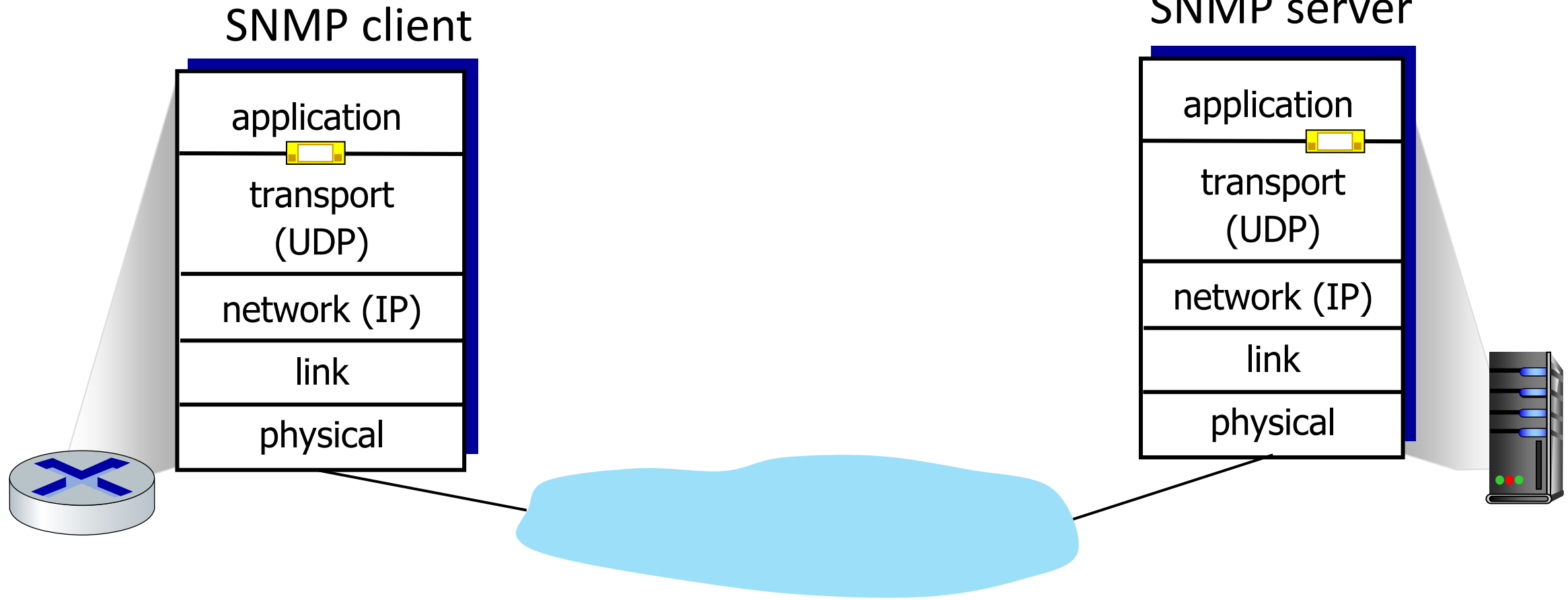
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

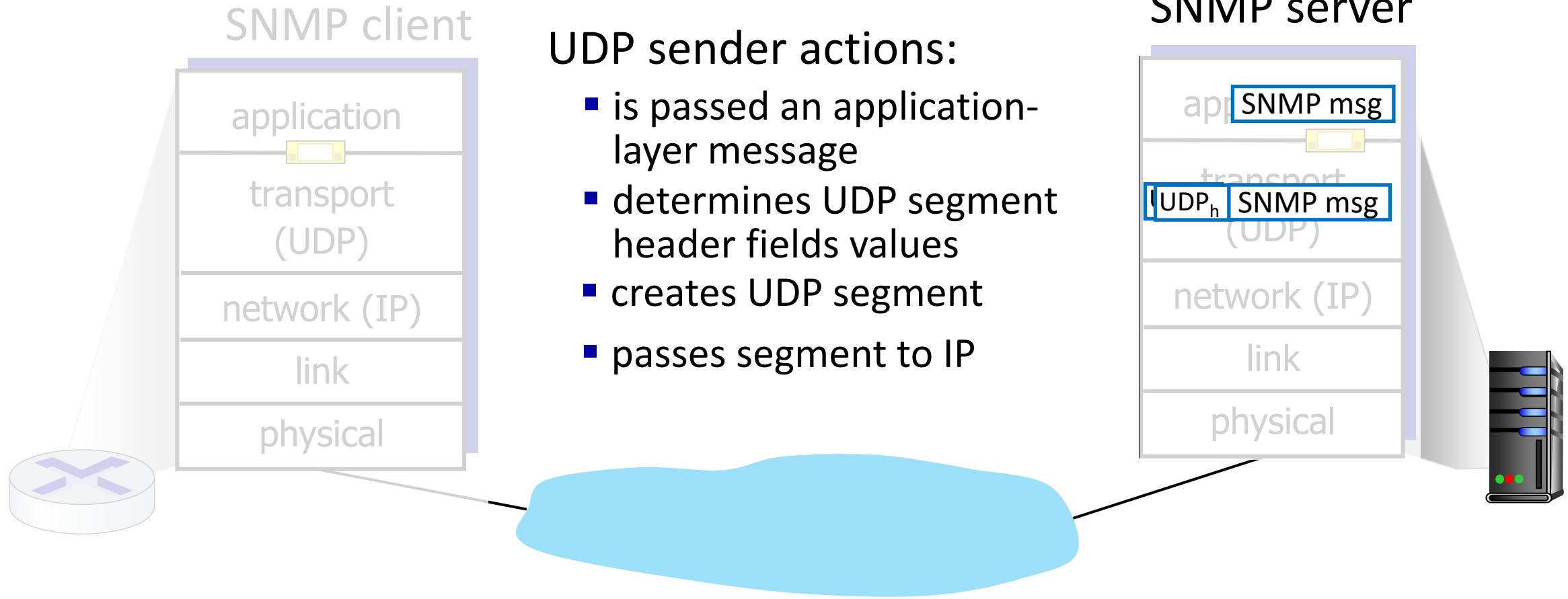
Format



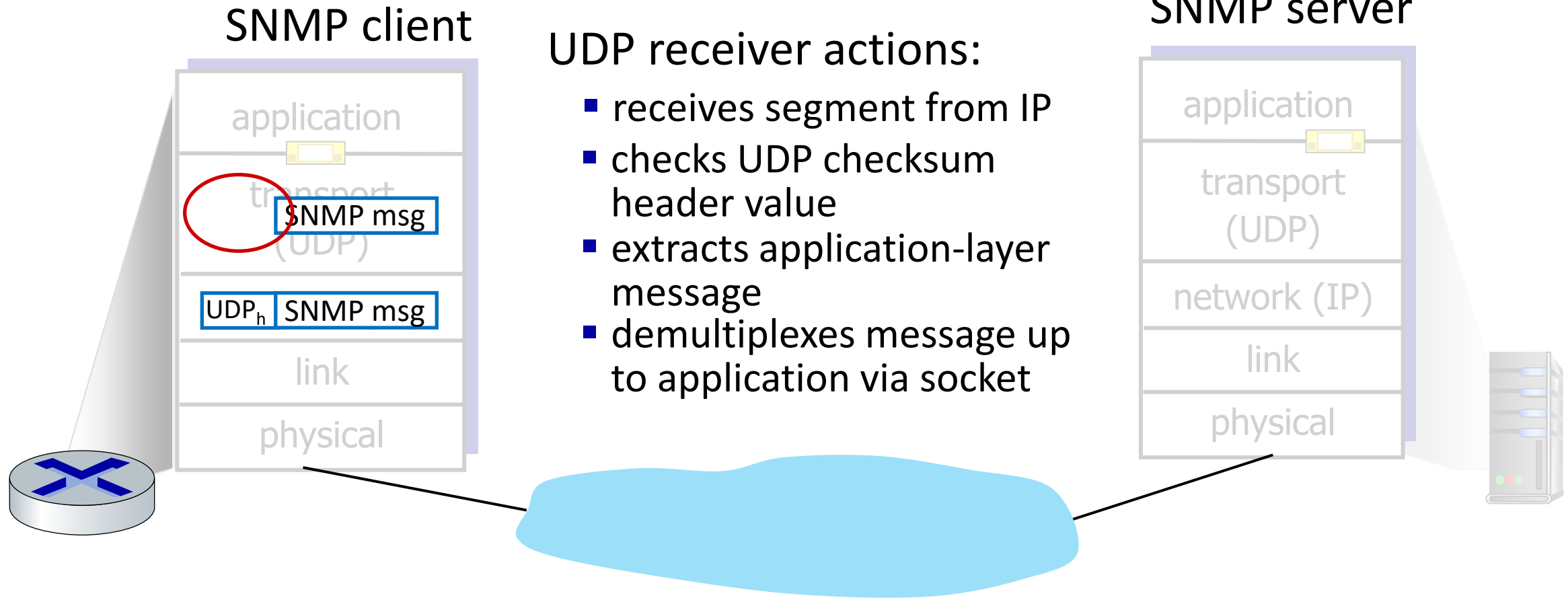
UDP: Transport Layer Actions



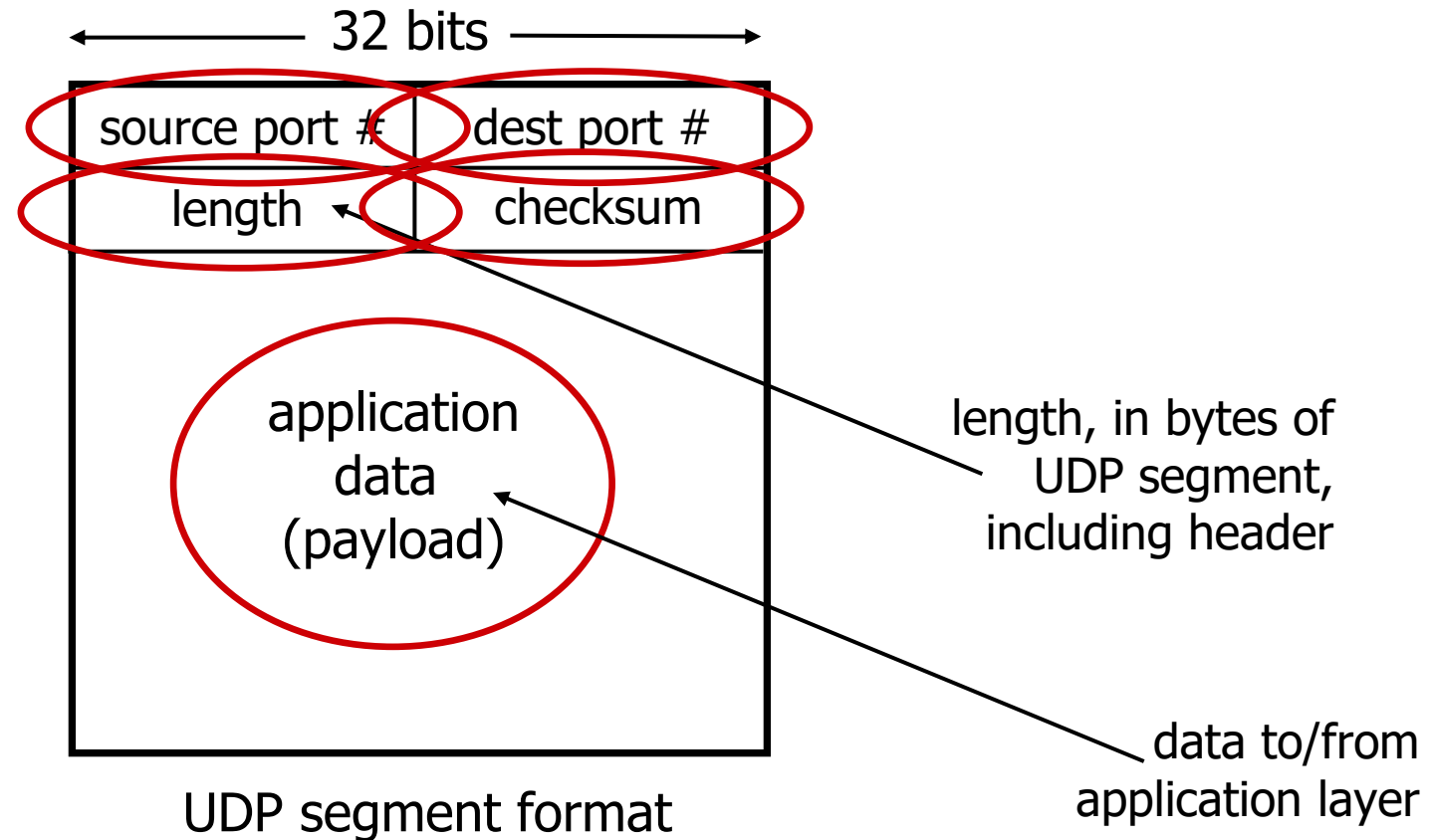
UDP: Transport Layer Actions



UDP: Transport Layer Actions



UDP segment header



maximum length of a UDP segment is 65,535 bytes

Note

- Unlike TCP, UDP **does not** have sequence numbers, acknowledgments, or flow control mechanisms.

Common UDP Port Numbers

Service	Port
DNS (Domain Name System)	53
DHCP (Dynamic Host Configuration Protocol)	67, 68
TFTP (Trivial File Transfer Protocol)	69
SNMP (Simple Network Management Protocol)	161
VoIP (Voice over IP)	5060

When to Use UDP?

Use Case

Video Streaming (YouTube, Netflix)

Online Gaming

VoIP (Skype, WhatsApp calls)

DNS Lookups

IoT Communication

Why Use UDP?

Speed is more important than perfect delivery

Low latency is critical, minor packet loss is acceptable

Smooth audio is more important than retransmissions

Fast request-response, no need for reliability

Lightweight protocol with minimal overhead

Applications Using UDP

- Applications which require one response for one request use UDP. Example- **DNS**.
- Routing Protocols like **RIP and OSPF** use UDP because they have very small amount of data to be transmitted.
- Trivial File Transfer Protocol (**TFTP**) uses UDP to send very small sized files.
- Broadcasting and multicasting applications use UDP.
- **Streaming applications** like multimedia, video conferencing etc use UDP since they require speed over reliability.
- **Real time applications** like chatting and online games use UDP.
- **SNMP** uses UDP.
- **Bootp / DHCP** uses UDP.
- Other protocols that use UDP are- Kerberos, Network Time Protocol (NTP), Network News Protocol (NNP), Quote of the day protocol etc.

Solve

A video streaming application sends **UDP packets** of **1200 bytes** each, with a **transmission rate of 5 Mbps**.

Find:

1. The number of packets sent per second.
2. The transmission time for one packet.

Solution

1. Packets Sent per Second

Packets per second = Transmission Rate / Packet Size

$$= 5 \times 10^6 \text{ bps} / 1200 \times 8 \text{ bits}$$

$$= 5 \times 10^6 / 9600$$

$$= 520.83$$

≈ 520 packets per second

2. Transmission Time for One Packet

Transmission Time = Packet Size × 8 / Transmission Rate

$$= 1200 \times 8 / 5 \times 10^6$$

$$= 9600 / 5 \times 10^6$$

$$= 1.92 \text{ milliseconds}$$

Solve

- A **UDP segment** carries **data payload of 1000 bytes**. Find the percentage of UDP header overhead.

Solution:

- **UDP Header Size = 8 bytes**
- **Total Packet Size = 1000 + 8 = 1008 bytes**
- **Overhead Percentage :**
- **Overhead = $(8/1008) \times 100 = 0.79\%$**
- **Thus, UDP header overhead is 0.79% of the total packet size.**

UNIT – III

- **Network Layer:** Switching, Logical addressing - IPV4, IPV6; Address mapping - ARP, RARP, BOOTP and DHCP-Delivery, Forwarding and Unicast Routing protocols
- **Transport Layer:** Process to Process Communication, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), SCTP Congestion Control; Quality of Service (QoS), QoS improving techniques - Leaky Bucket and Token Bucket algorithms

Transmission Control Protocol (TCP)

TCP OVERVIEW

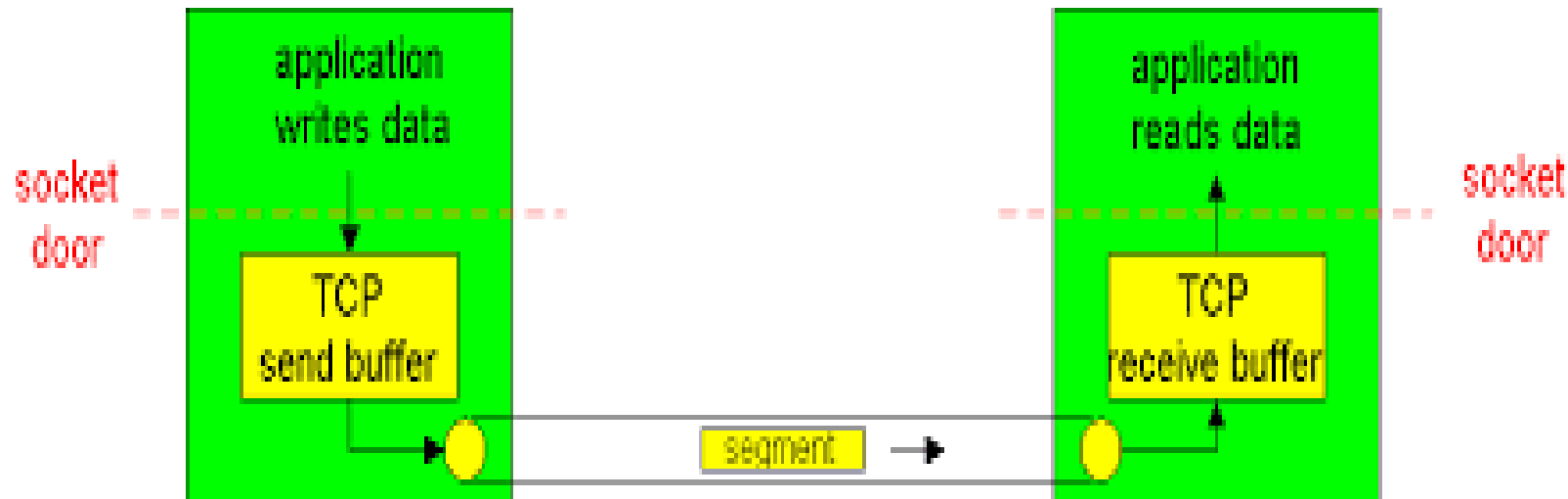
- Services
- TCP connection Management
- TCP segment structure
- TCP RTT
- TCP fast retransmit
- TCP Flow control
- TCP Congestion control

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

- **point-to-point:**
 - one sender, one receiver
- **Reliable:**
 - Logical connection
 - Runs only in end-systems
- **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 Connection

- Client Process, Server Process
- `clientSocket.connect(serverName,serverPort)`
- Three Segments are transferred(3-way Handshaking)



TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

❑ initialize TCP variables:

- seq. #s
- buffers, flow control info (e.g. RcvWindow)

❑ *client*: connection initiator

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

❑ *server*: contacted by client

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

Three way handshake:

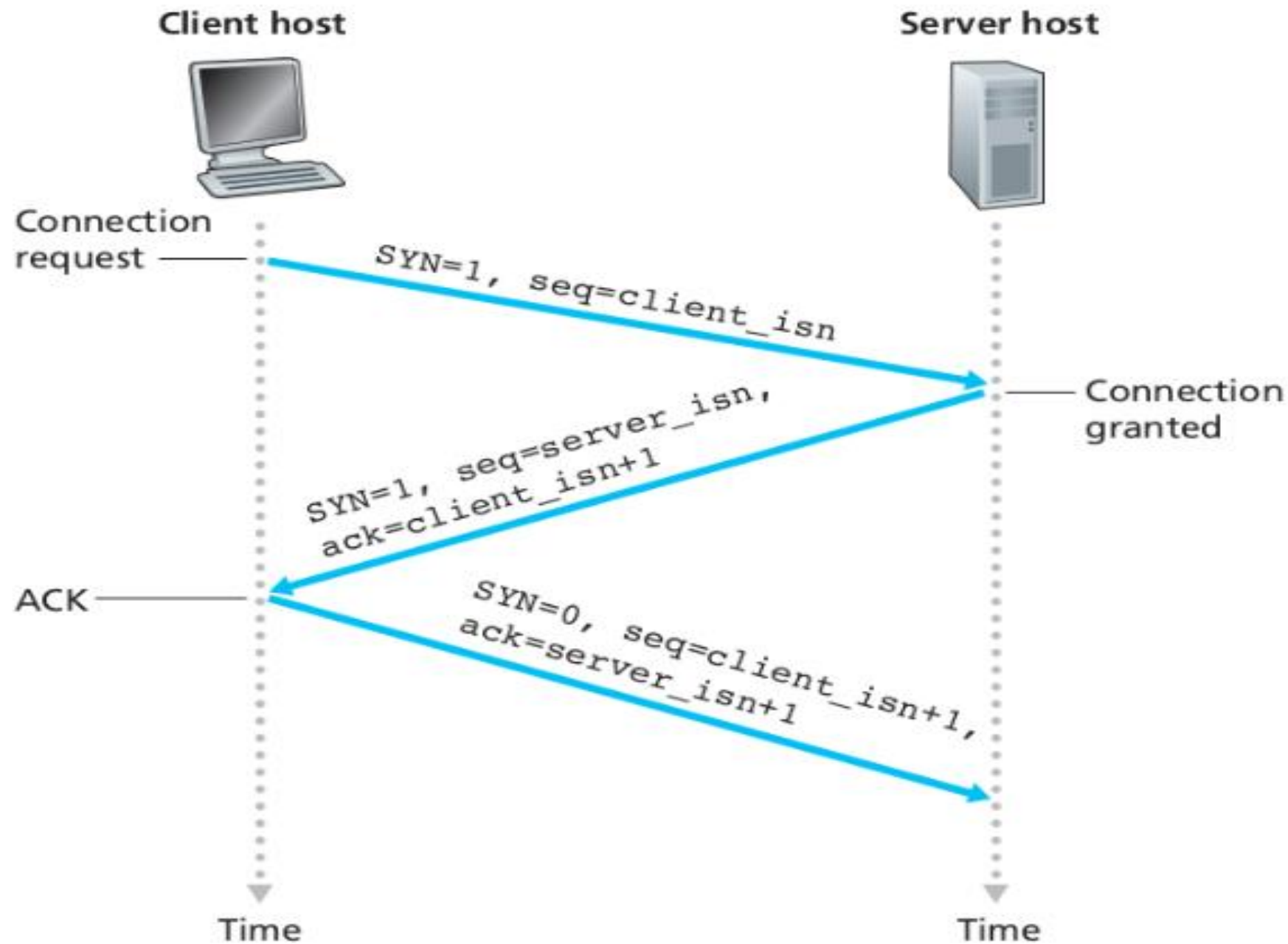
Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



Note

- This connection-granted segment is saying, in effect, “I received your SYN packet to start a connection with your initial sequence number, client_isn. I agree to establish this connection. My own initial sequence number is server_isn.”
- The connection-granted segment is referred to as a **SYNACK segment**.

- Upon receiving the SYNACK segment, the client also allocates buffers and variables to the connection.
- The client host then sends the server yet another segment; this last segment acknowledges the server's connection-granted segment (the client does so by putting the value `server_isn+1` in the acknowledgment field of the TCP segment header).
- The SYN bit is set to zero, since the connection is established.
- This third stage of the three-way handshake may carry client-to-server data in the segment payload.
- Once these three steps above have been completed, the client and server hosts can send segments containing data to each other. In each of these future segments, the SYN bit will be set to zero.

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

TCP Connection Management (cont.)

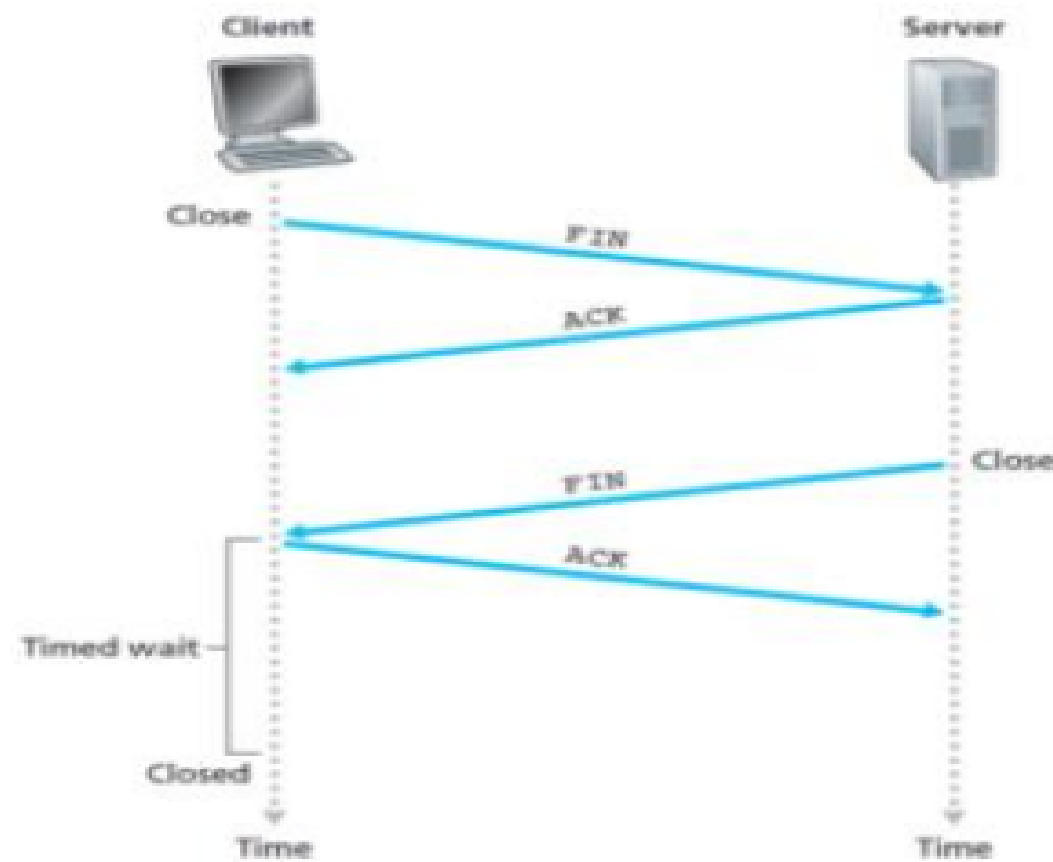
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system
sends TCP FIN control
segment to server

Step 2: server receives
FIN, replies with ACK.
Closes connection, sends
FIN.



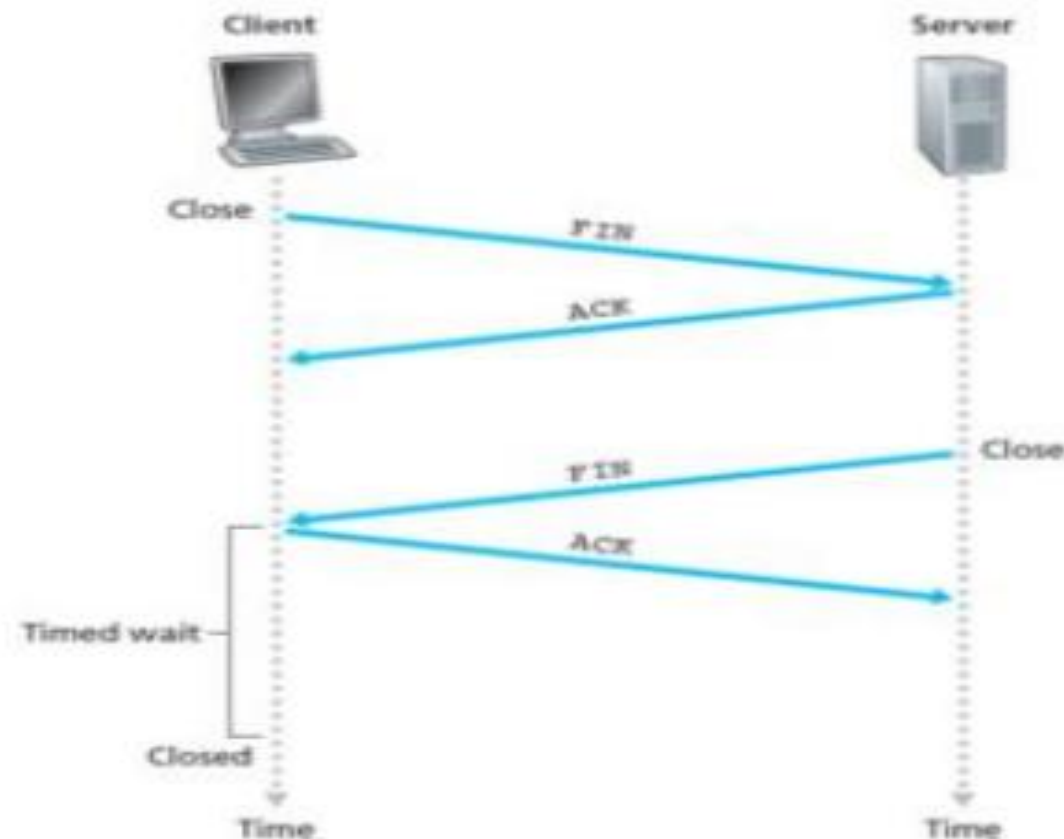
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

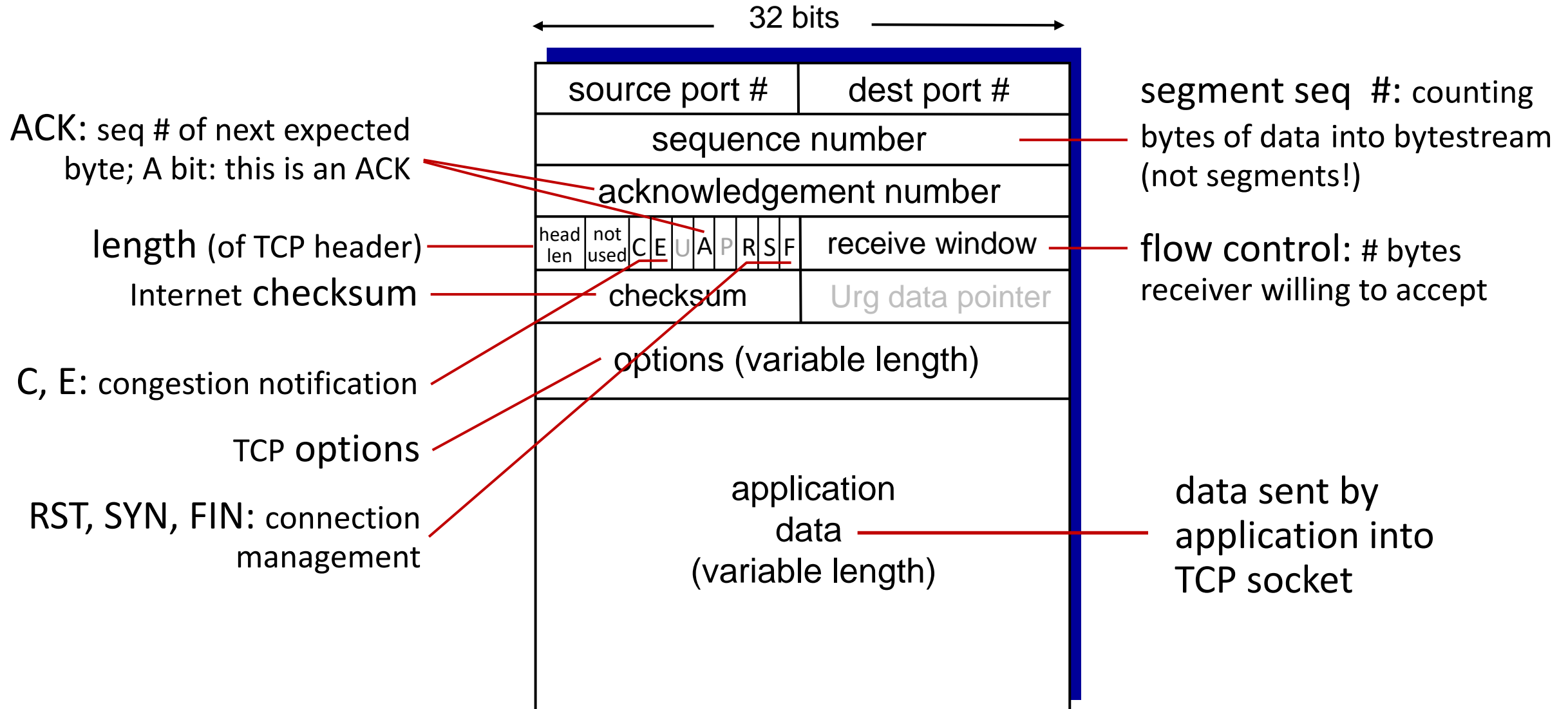
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP segment structure



TCP sequence numbers, ACKs

Sequence numbers:

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

Acknowledgements:

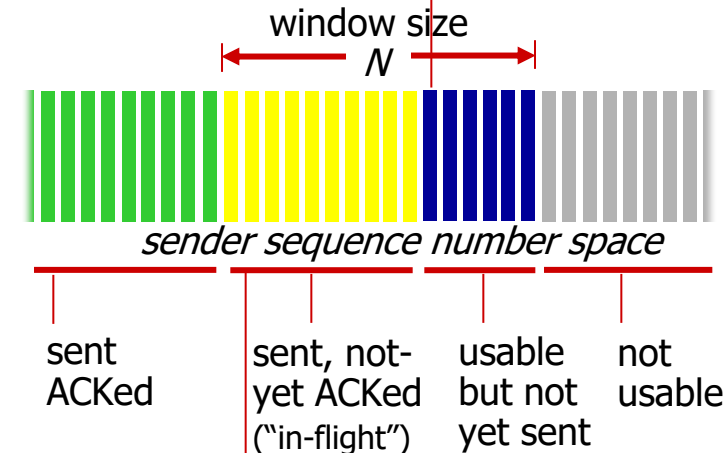
- seq # of next byte expected from other side
- cumulative ACK

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

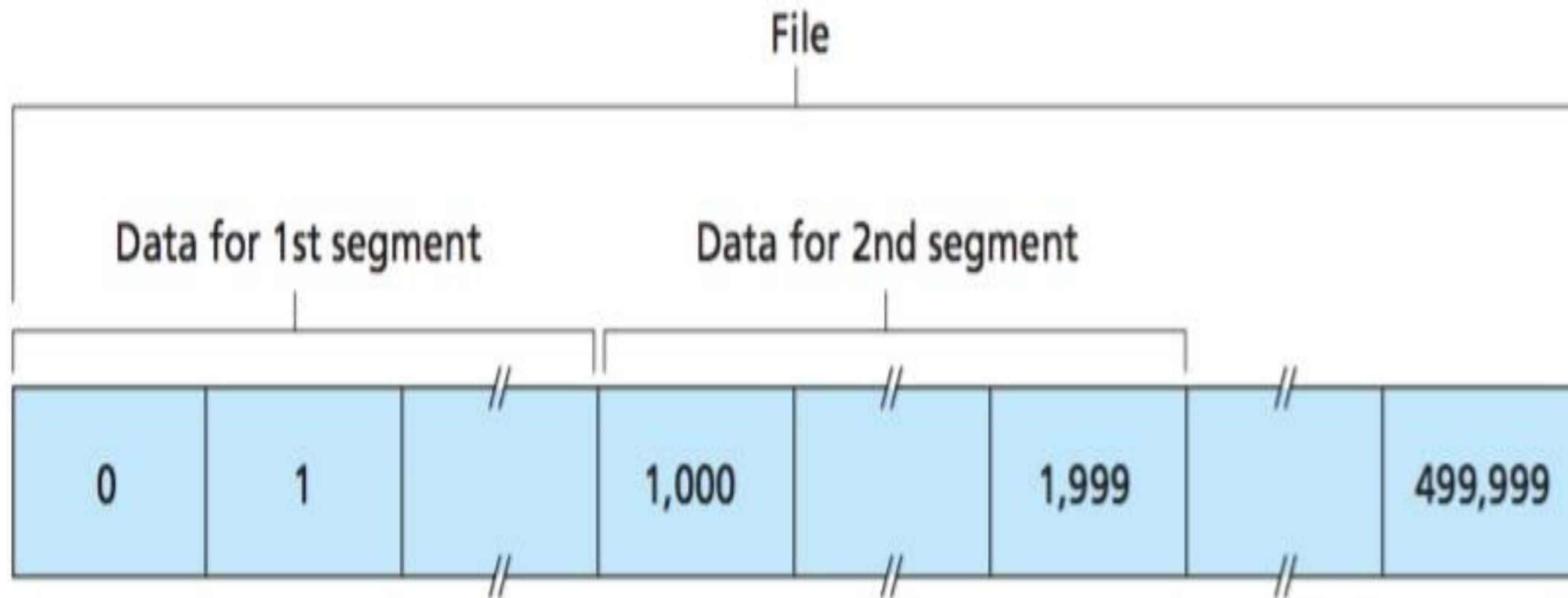
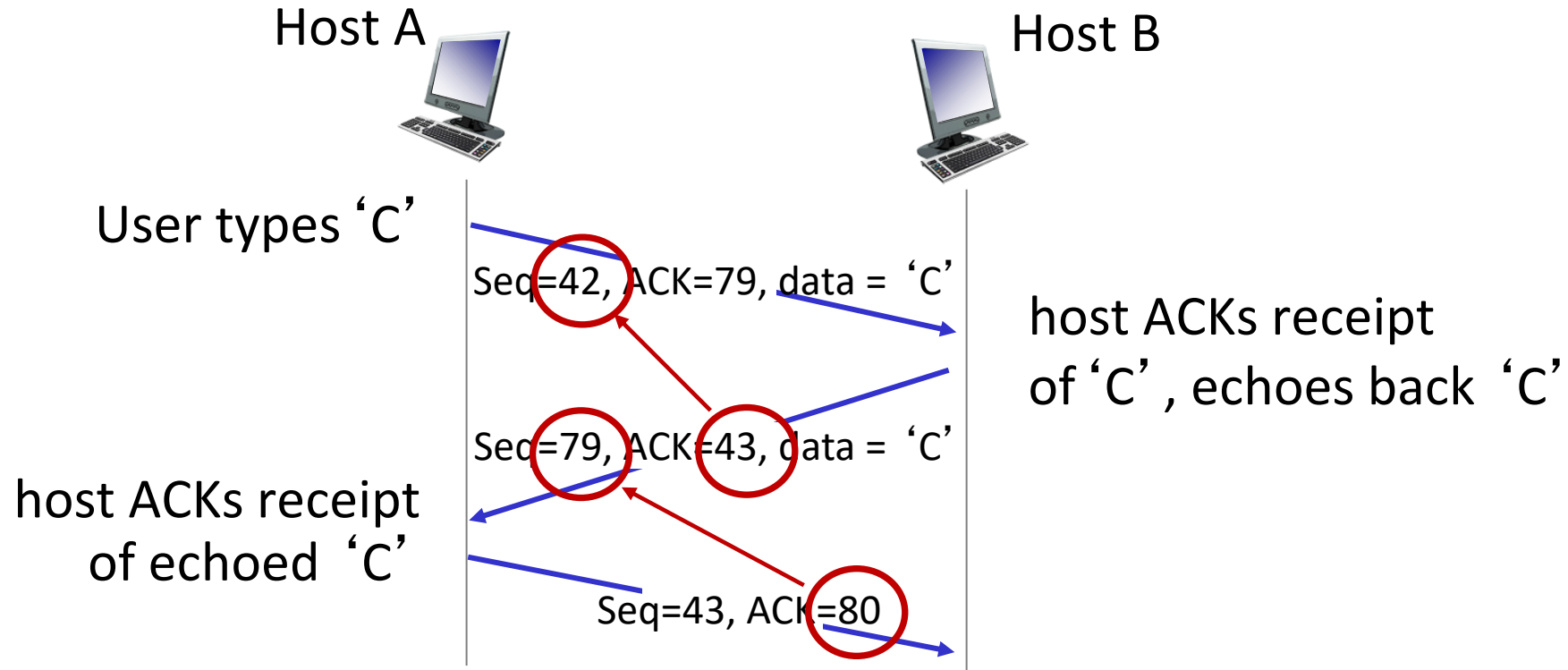


Figure 3.30 ♦ Dividing file data into TCP segments

TCP sequence numbers, ACKs



simple telnet scenario

Think!!!

- Suppose Host A has received all bytes numbered 0 through 535 from Host B and suppose that it is about to send a segment to Host B. So Host A puts ___ in ___ field of the segment it sends to B.
- Suppose Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1000. For some reason Host A has not yet received bytes 536 through 899. What does host A now put in Ack number field?

Note

- Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgment**.

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*

Note

- Most TCP implementations take only one SampleRTT measurement at a time.
- TCP never computes a SampleRTT for a segment that has been retransmitted.

Overview

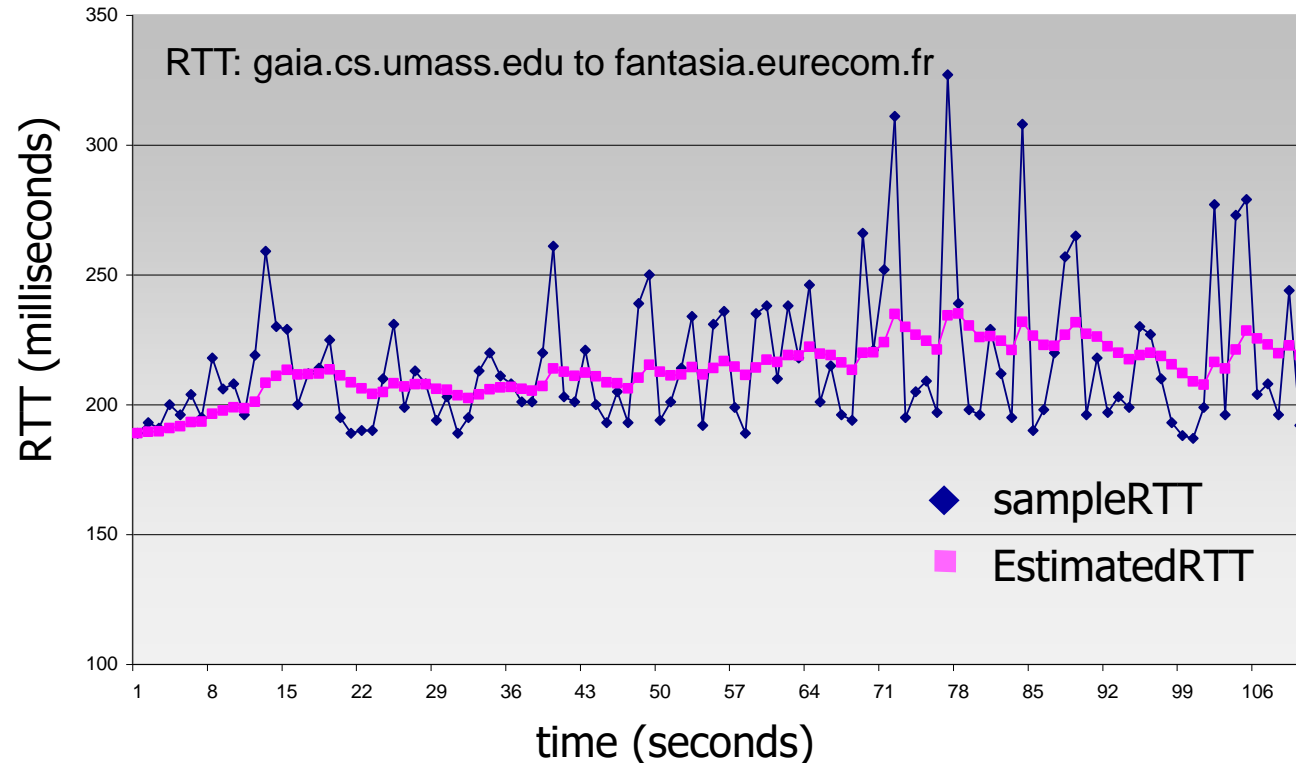
- We use different methods and mediums for sending and receiving data in our daily life, like SMS. Similarly, if we talk about computers, communications occur through networks in computers. Imagine that you send a text to your friend and are waiting for their reply, and then you receive a reply. The time that is taken to complete the process of requesting (your text) and receiving a response (friend's answer) is known as RTT (round trip time).

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: $\alpha = 0.125$

A constant smoothing factor
(typically between 0 and 1) used
to balance the influence of
previous and current RTT values



Estimated RTT

- In computer communication networking, the RTTs of different packets can be different.
- For example, the first packet takes the round trip time of 1.1ms, the second packet takes 1.3ms, and the third packet takes 0.98ms so, each sample RTT varies.
- That is why estimated RTT is used, as it is the average of recent measurements, not just the current sample RTT.

Problem

- For example, the sample RTT is 100ms, we have to compute the estimated RTT using $\alpha = 0.125$, and we assume the value of the estimated RTT just before the sample RTT was 110ms. So, by using the formula, we get:

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

- Estimated RTT = $(1 - 0.125) * 110\text{ms} + (0.125) * 100\text{ms}$
- Estimated RTT = $0.875 * 110\text{ms} + 12.5\text{ms}$
- Estimated RTT = $96.25\text{ms} + 12.5\text{ms} = 108.75\text{ms}$
- **Hence, the required estimated RTT is 108.75ms.**

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, $\beta = 0.125$)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Deviation in RTT(Dev-RTT)

- Measure that indicates how evenly the RTT is distributed during the measurement.
- It depends upon the previous estimated RTT and helps to find the retransmission time-out.
- For example, the sample RTT is 100ms, we have to compute the DevRTT, and we assume the value of the estimated RTT is 108.75ms and the previous DevRTT was 20ms. So, by using the formula, we get:

$$\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{Sample RTT} - \text{Estimated RTT}|$$

Where typically, the value of $\beta = 0.125$.

- $\text{Dev RTT} = (1 - 0.125) * 20\text{ms} + (0.125) * |100\text{ms} - 108.75\text{ms}|$
- $\text{Dev RTT} = (0.875) * 20\text{ms} + (0.125) * 8.75\text{ms}$
- $\text{Dev RTT} = 17.5\text{ms} + 1.09\text{ms} = 18.59\text{ms}$
- **Hence, the required DevRTT is 18.59ms**

Time-out

- Time-out is longer than RTT, but as RTT varies, we need to add some margin i.e., safety margin, to it.
- The selection of a time-out value is essential because the longer the value of estimated RTT, the slower its performance.
- We will be facing long delays in this case. Similarly, in the case of a minimal value, the connection can be lost before the RTT completes or before the arrival of the response or acknowledgment.
- So for the safety, we use:
- **Time-out Interval = Estimated RTT + 4 * DevRTT**
- Where the DevRTT value is used for a safety margin.

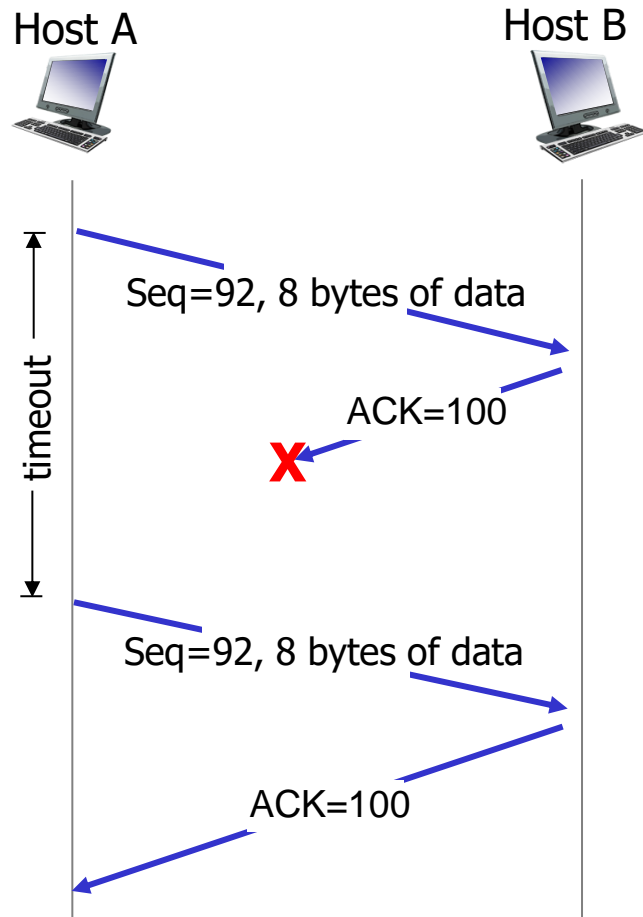
Problem

- So, using the above-computed values of DevRTT = 18.59ms and Estimated RTT = 108.75ms, we compute the time-out interval as:
- **Time-out Interval = 4 * DevRTT + Estimated RTT**
- Time-out Interval = 4 * 18.59ms + 108.75ms
- Time-out Interval = 74.36ms + 108.75ms = 183.11ms
- Hence, the required time-out interval is 183.11ms.

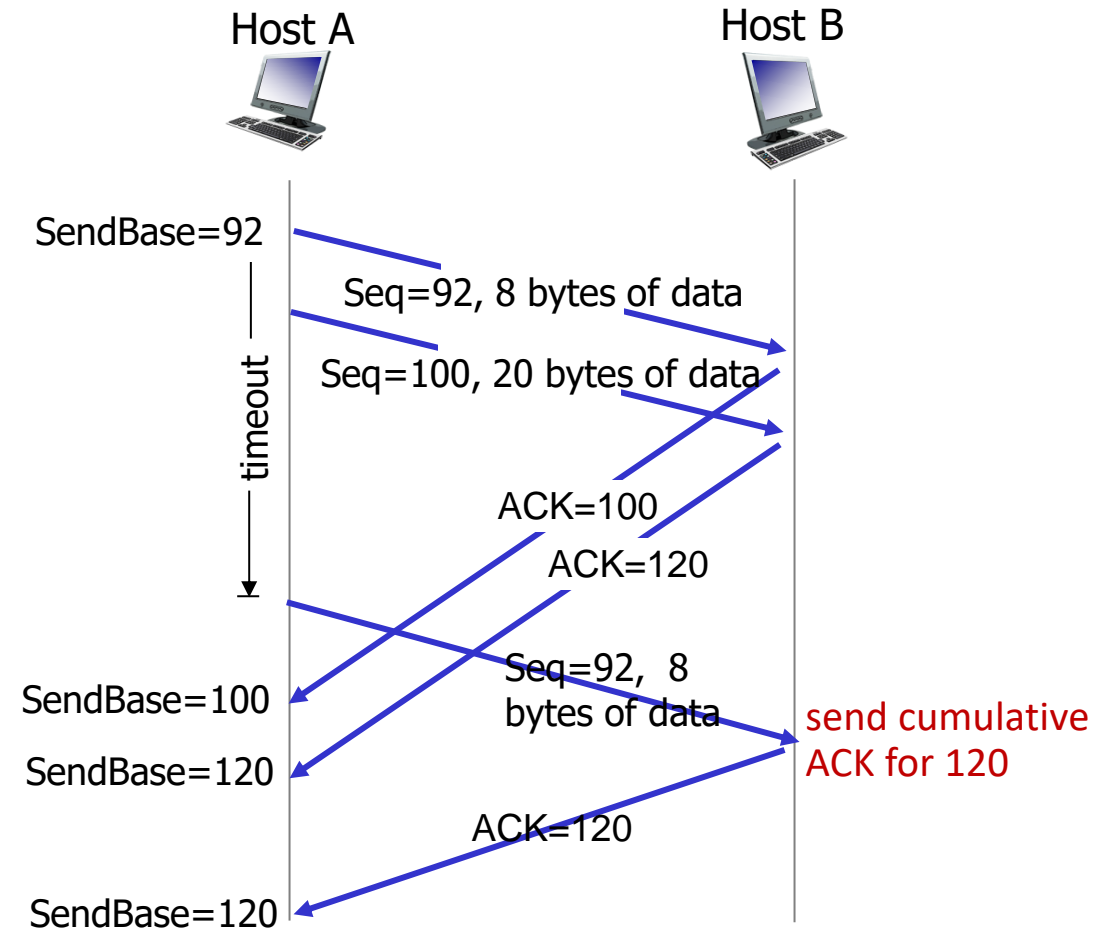
Note

- We can see that the time-out interval is dependent on both DevRTT and estimated RTT.
- Also, DevRTT is dependent on estimated RTT.
- So, if we need to find DevRTT, we have to compute the estimated RTT first.
- To compute the time-out interval, we have to find both DevRTT and estimated RTT first.

TCP: retransmission scenarios



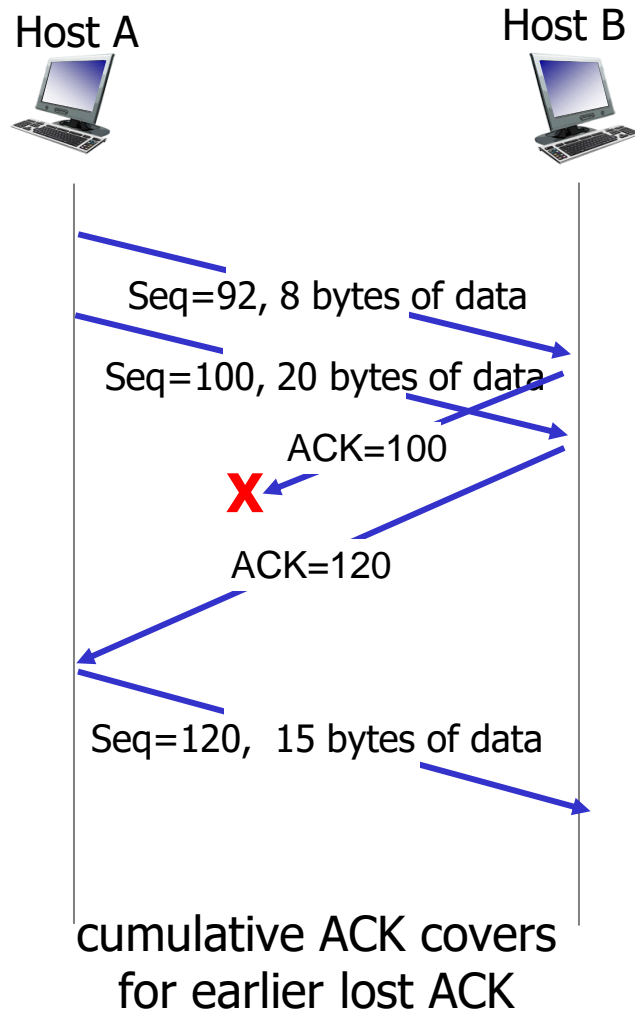
lost ACK scenario



premature timeout

As long as the ACK for the second segment arrives before the new timeout, the second segment will not be retransmitted.

TCP: retransmission scenarios




So when the client didn't successfully receive the expecting ACK segment, it will not always resend the oldest not acknowledged segment.

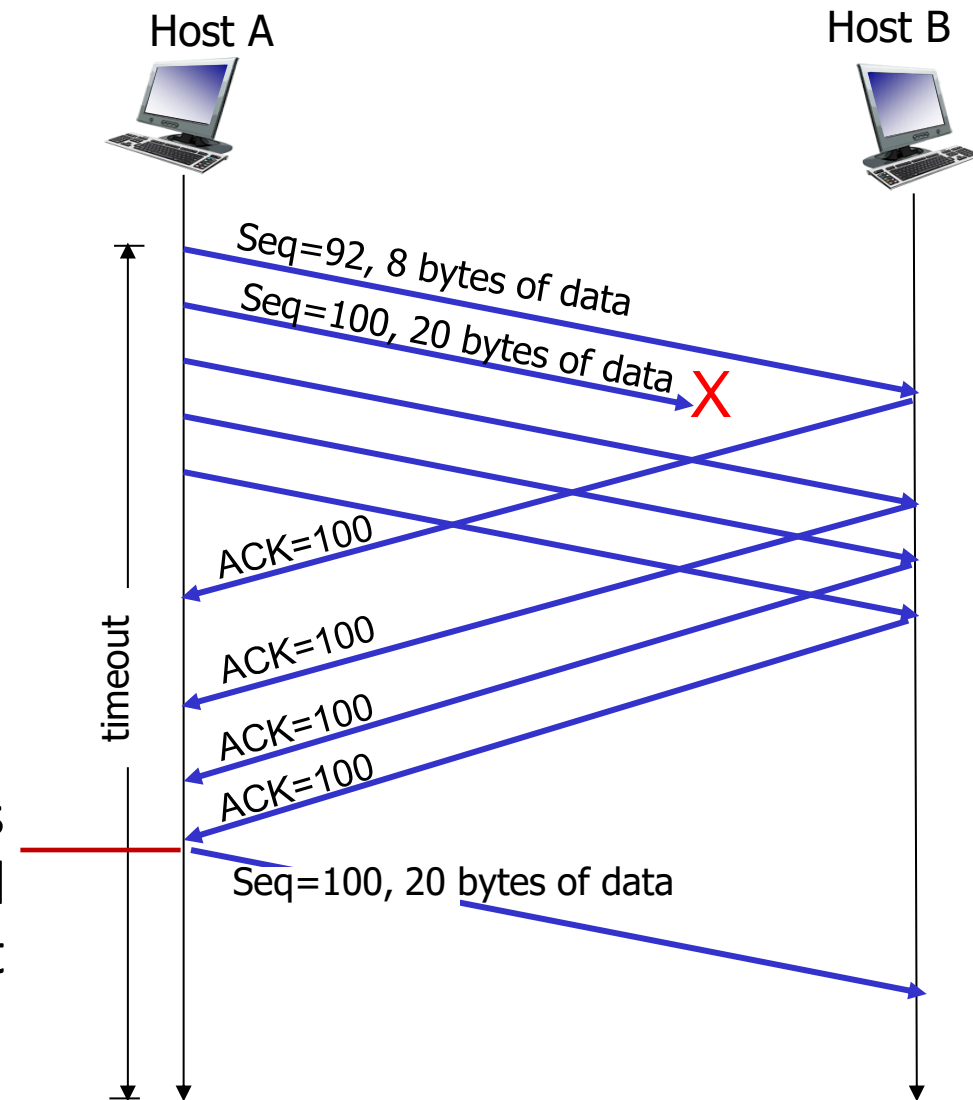
TCP fast retransmit

TCP fast retransmit

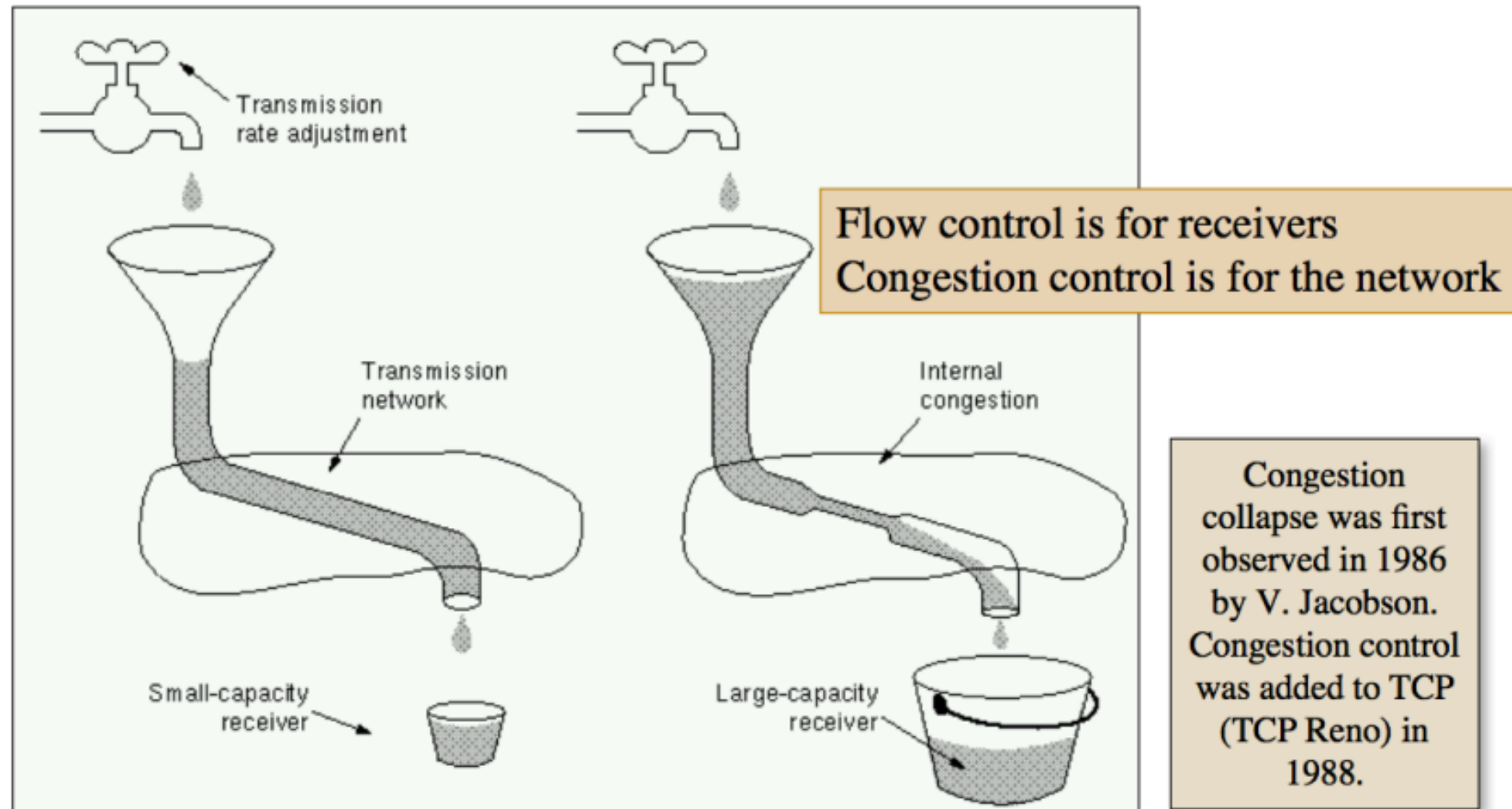
if sender receives 3 additional ACKs for same data (“triple duplicate ACKs”), resend unACKed segment with smallest seq #

- 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!



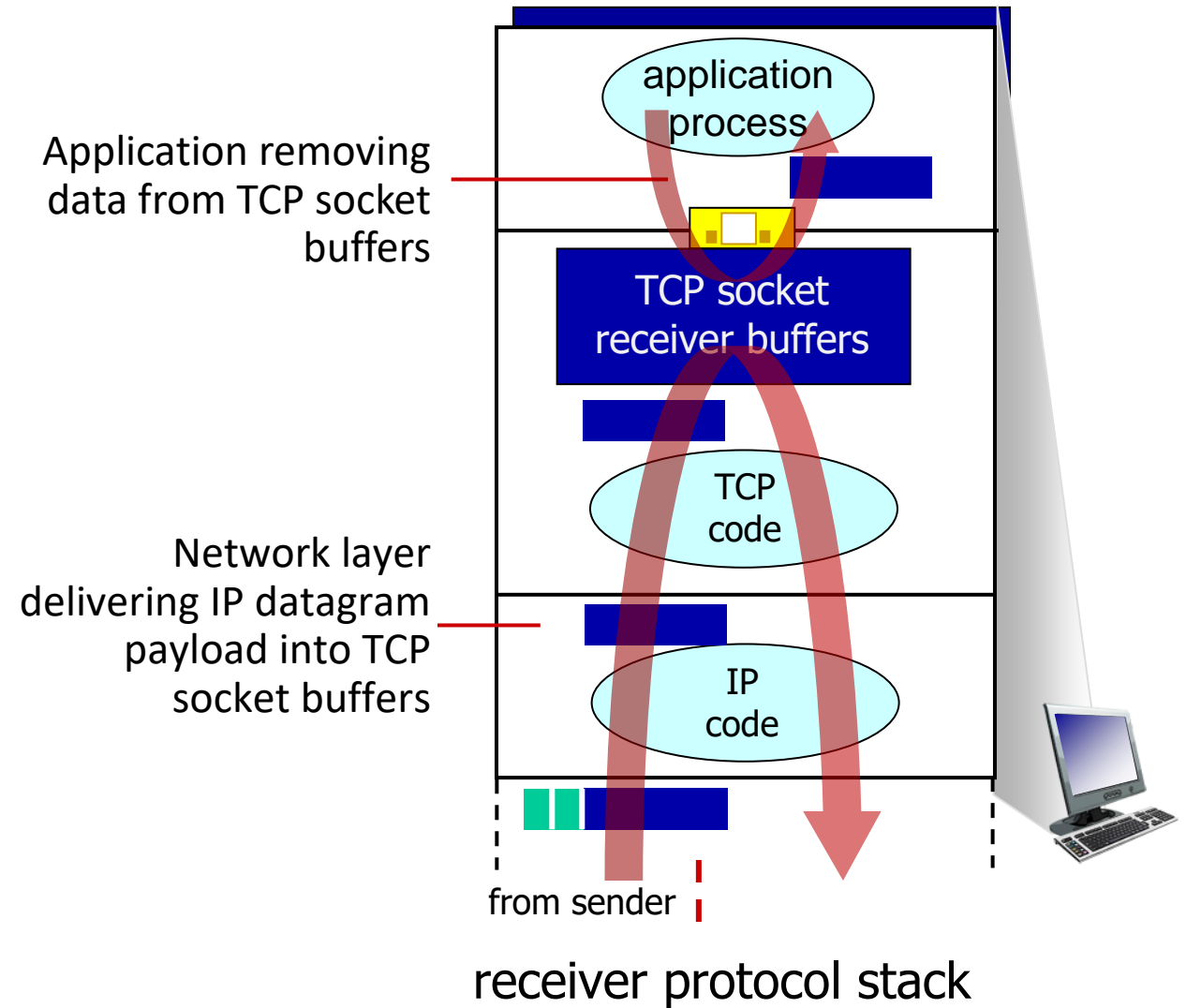
Let's go back to the origin!



From Computer Networks, A. Tanenbaum

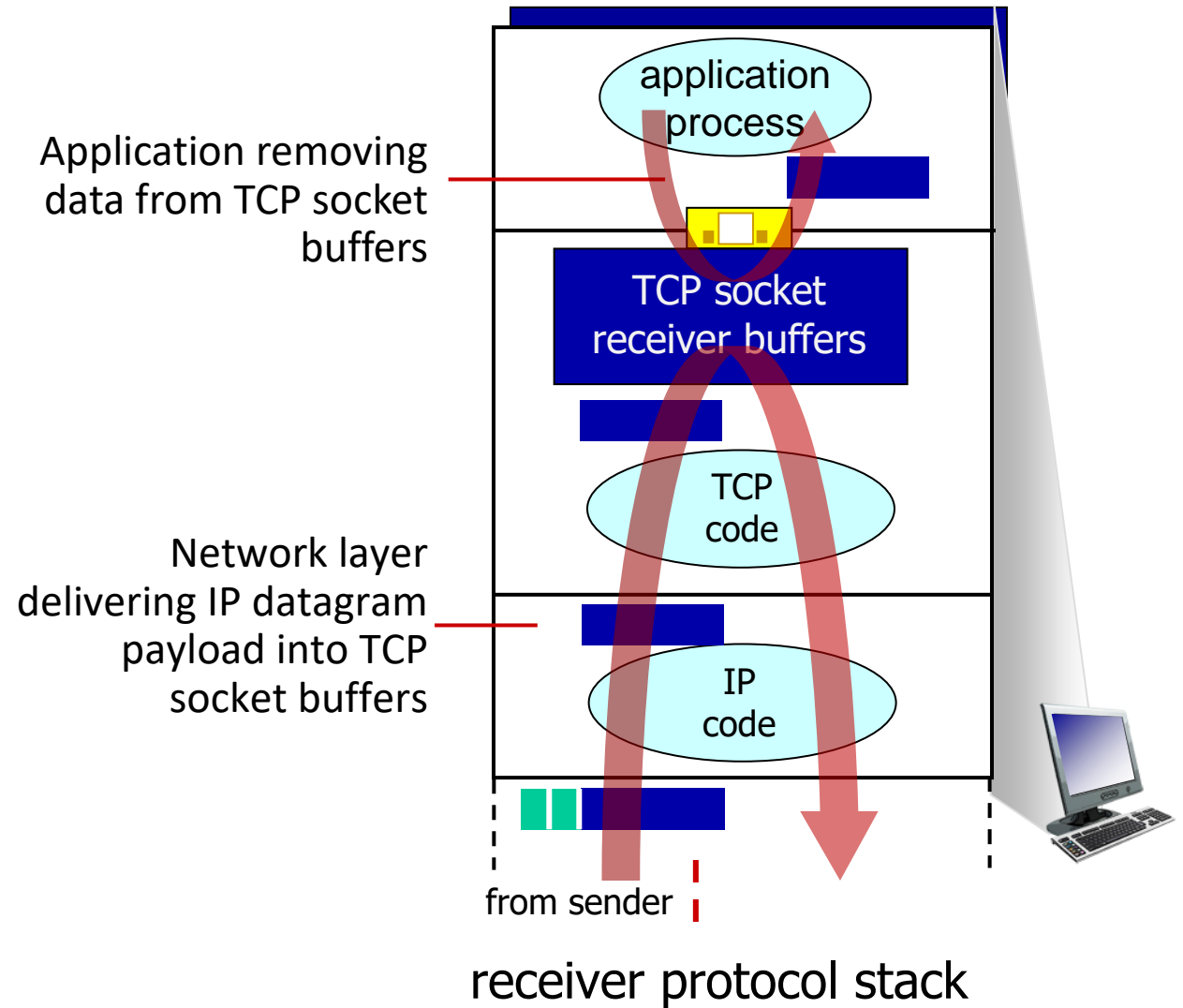
TCP flow control

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



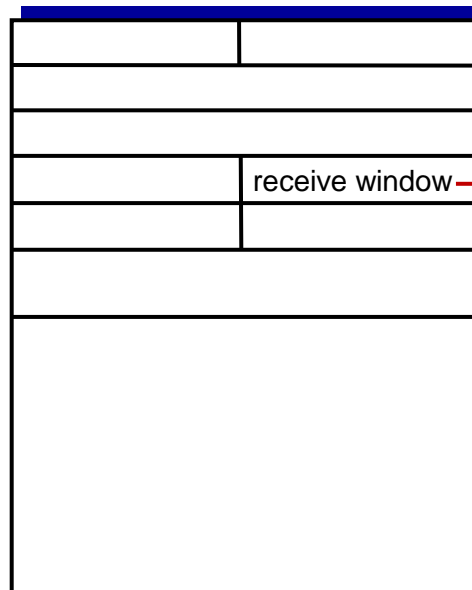
TCP flow control

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



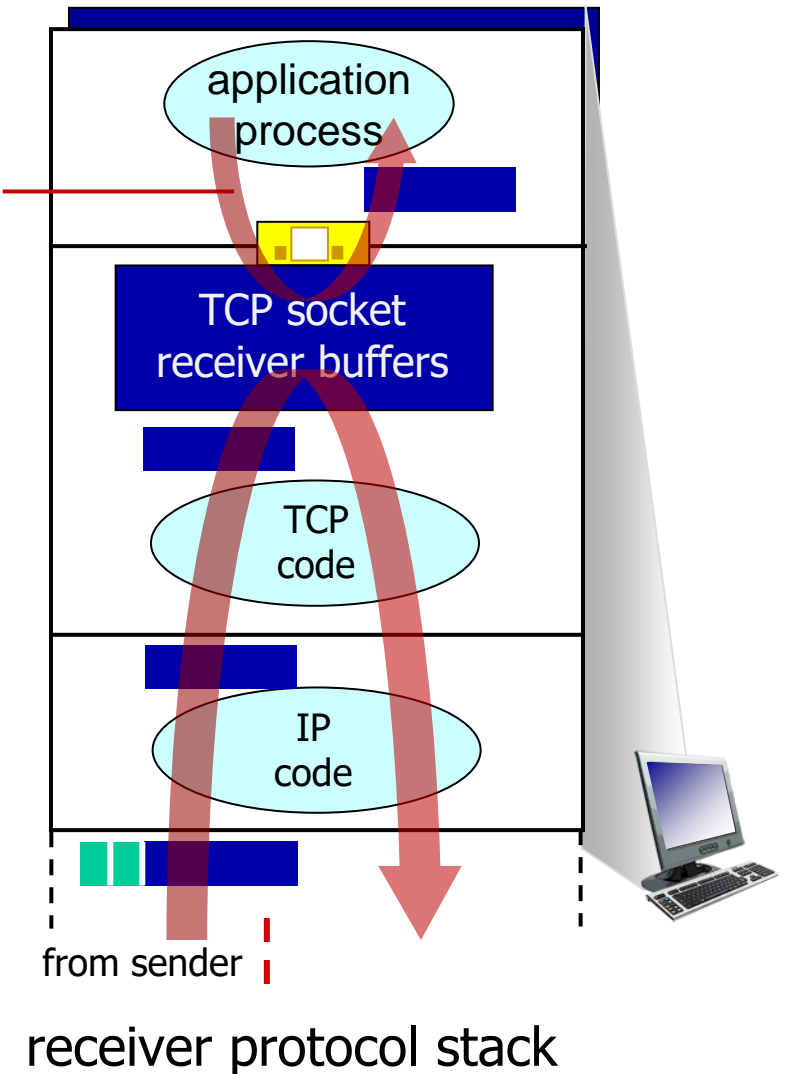
TCP flow control

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



flow control: # bytes receiver willing to accept

Application removing data from TCP socket buffers



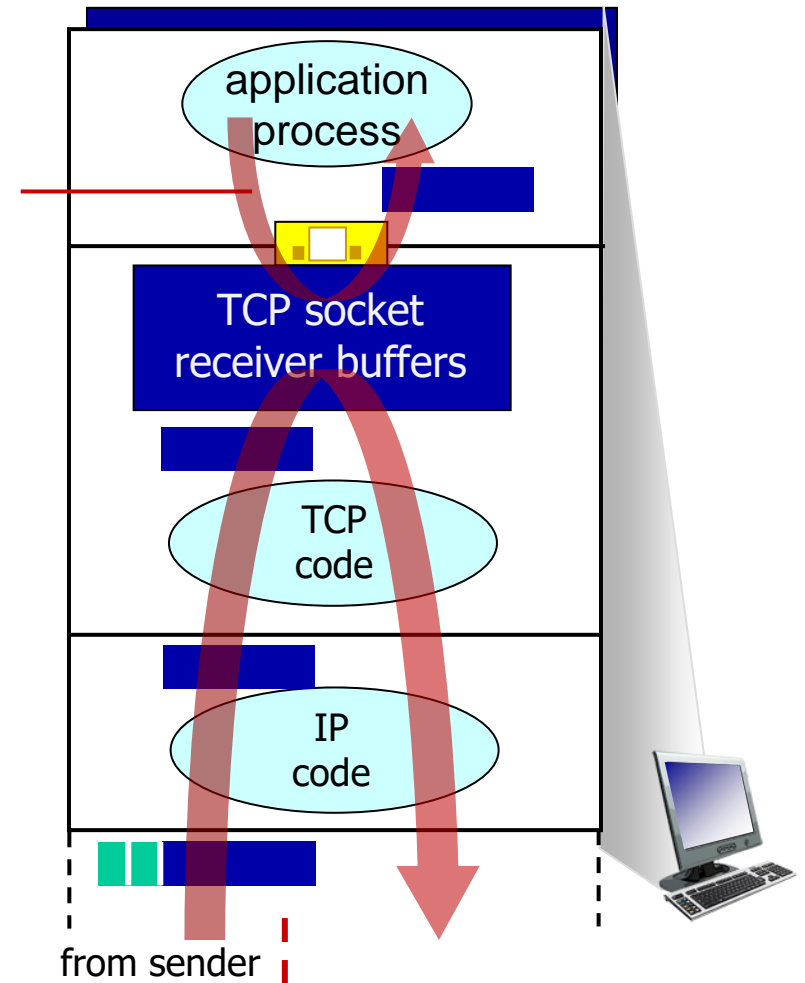
TCP flow control

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

—flow control—

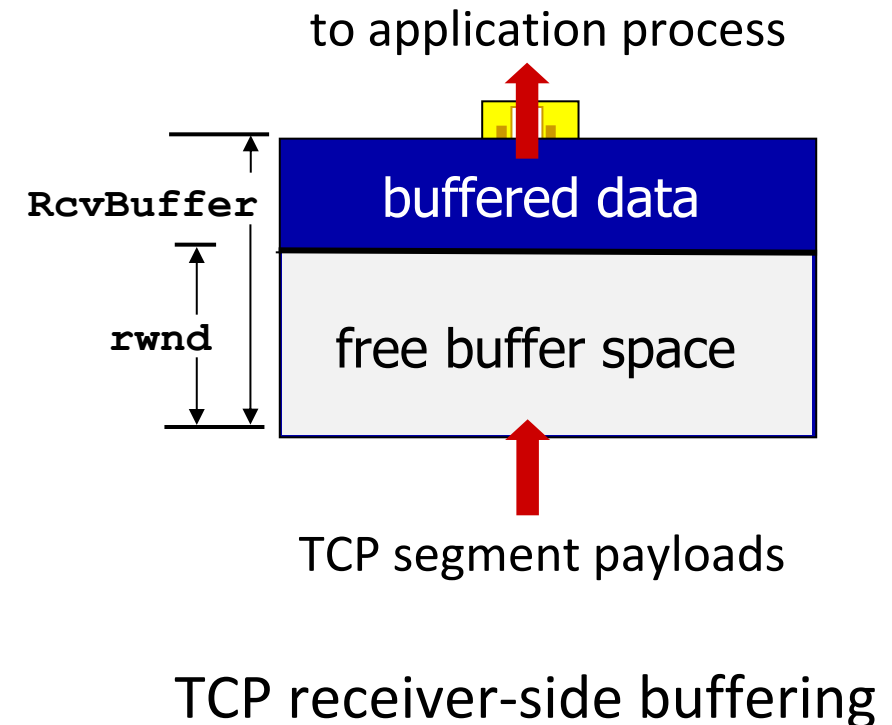
receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

Application removing data from TCP socket buffers



TCP flow control

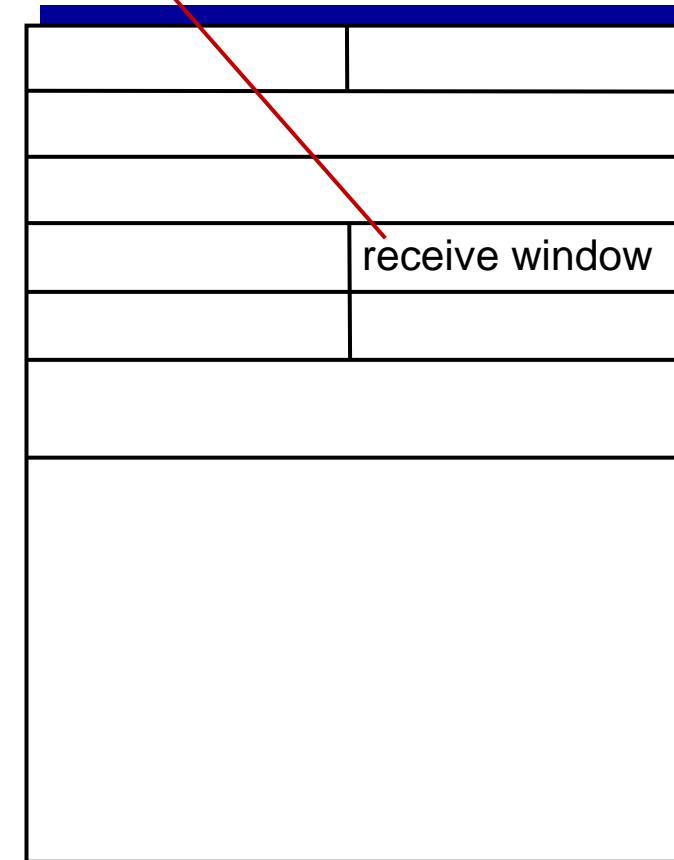
- TCP receiver “advertises” free buffer space in **rwnd** field in TCP header
 - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
 - many operating systems auto-adjust **RcvBuffer**
- sender limits amount of unACKed (“in-flight”) data to received **rwnd**
- guarantees receive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

Principles of congestion control

Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- manifestations:
 - long delays (queueing in router buffers)
 - packet loss (buffer overflow at routers)
- different from flow control!
- a top-10 problem!



congestion control:

too many senders,
sending too fast



flow control: one sender
too fast for one receiver

TCP Connection

- TCP can “send data in segment at its own convenience”.
- The max amount of data that can be grabbed and placed in a segment is limited by “MSS”.
- MSS is set by determining the length of the largest link-layer frame that can be sent by local sending host(MTU).
- MSS to ensure “TCP segment + TCP/IP header length” to fit into a link layer frame.
- Both Ethernet and PPP have an MTU of 1500 bytes.

$$MTU - (TCP\ header + IP\ header) = MSS$$

- MSS – Max amount of application-layer data in the segment not the Max size of TCP segment including headers.

TCP MSS example

- Suppose a network router has an MTU of 1,500, meaning it only accepts packets up to 1,500 bytes long. (Longer packets will be fragmented.) What should the MSS for the router be set to?
- $MTU - (TCP\ header + IP\ header) = MSS$
- $1,500 - (20 + 20) = 1,460$
- The router's MSS should be set to 1,460 bytes. **Packets with a payload size larger than 1,460 bytes will be dropped.**
- *One of the key differences between MTU and MSS is that if a packet exceeds a device's MTU, it is broken up into smaller pieces, or "fragmented." In contrast, if a packet exceeds the MSS, it is dropped and not delivered.*

The size of the TCP header is typically 20 bytes while the size of UDP header is 8 bytes.

Congestion in Network

- Congestion refers to a network state where-*The message traffic becomes so heavy that it slows down the network response time.*
- Congestion leads to the loss of packets in transit.
- Necessary to control the congestion in network.

Congestion Control

- Congestion control refers to techniques and mechanisms that can-
 1. Either prevent congestion before it happens
 2. Or remove congestion after it has happened

TCP Congestion Control

- TCP reacts to congestion by reducing the sender window size.
- **The size of the sender window is determined by the following two factors-**
 1. Receiver window size
 2. Congestion window size

1. Receiver Window Size

Receiver window size is an advertisement of-

“How much data (in bytes) the receiver can receive without giving any acknowledgement?”

- 1.The sender should not send data greater than that of the size of receiver window.
- 2.If the data sent is greater than that of the size of the receiver's window, then it causes retransmission of TCP due to the dropping of TCP segment.
- 3.Hence sender should always send data that is less than or equal to the size of the receiver's window.
- 4.TCP header is used for sending the window size of the receiver to the sender.

2. Congestion Window

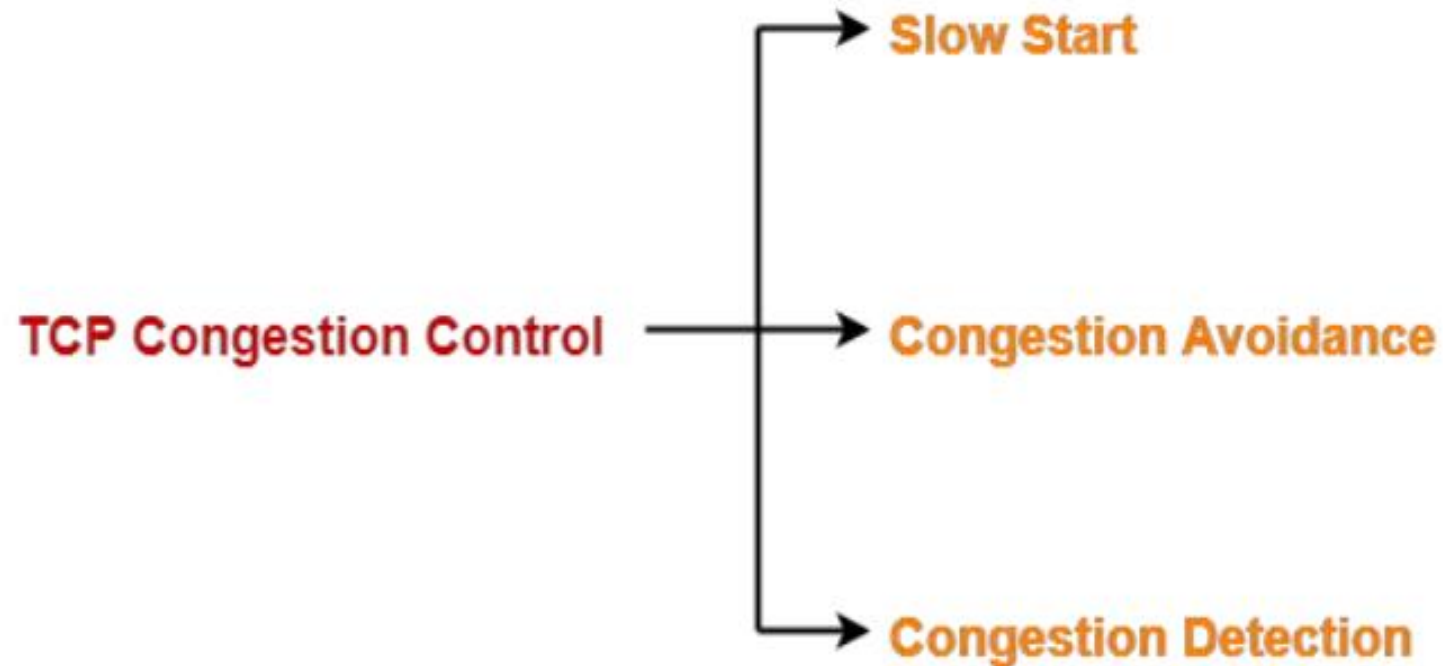
It is the state of TCP that limits the amount of data to be sent by the sender into the network even before receiving the acknowledgment.

- Sender should not send data greater than congestion window size.
- Otherwise, it leads to dropping the TCP segments which causes TCP Retransmission.
- So, sender should always send data less than or equal to congestion window size.
- Different variants of TCP use different approaches to calculate the size of congestion window.
- **Congestion window is known only to the sender and is not sent over the links.**

■ **Sender window size =
Minimum (Receiver window
size, Congestion window size)**

TCP Congestion Policy

- TCP's general policy for handling congestion consists of following three phases-



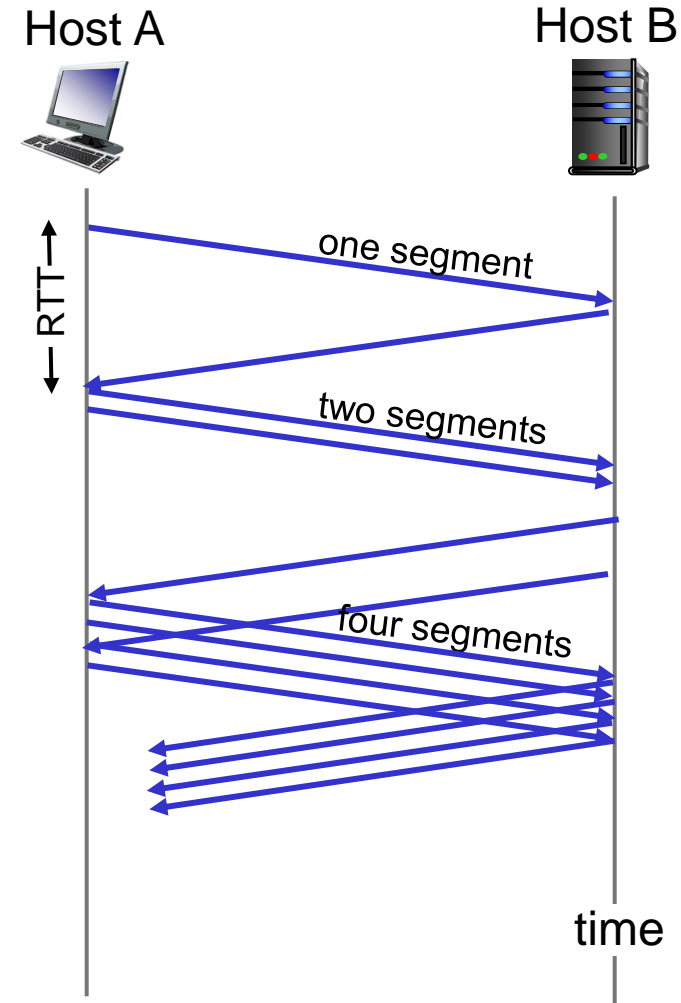
1. Slow Start Phase

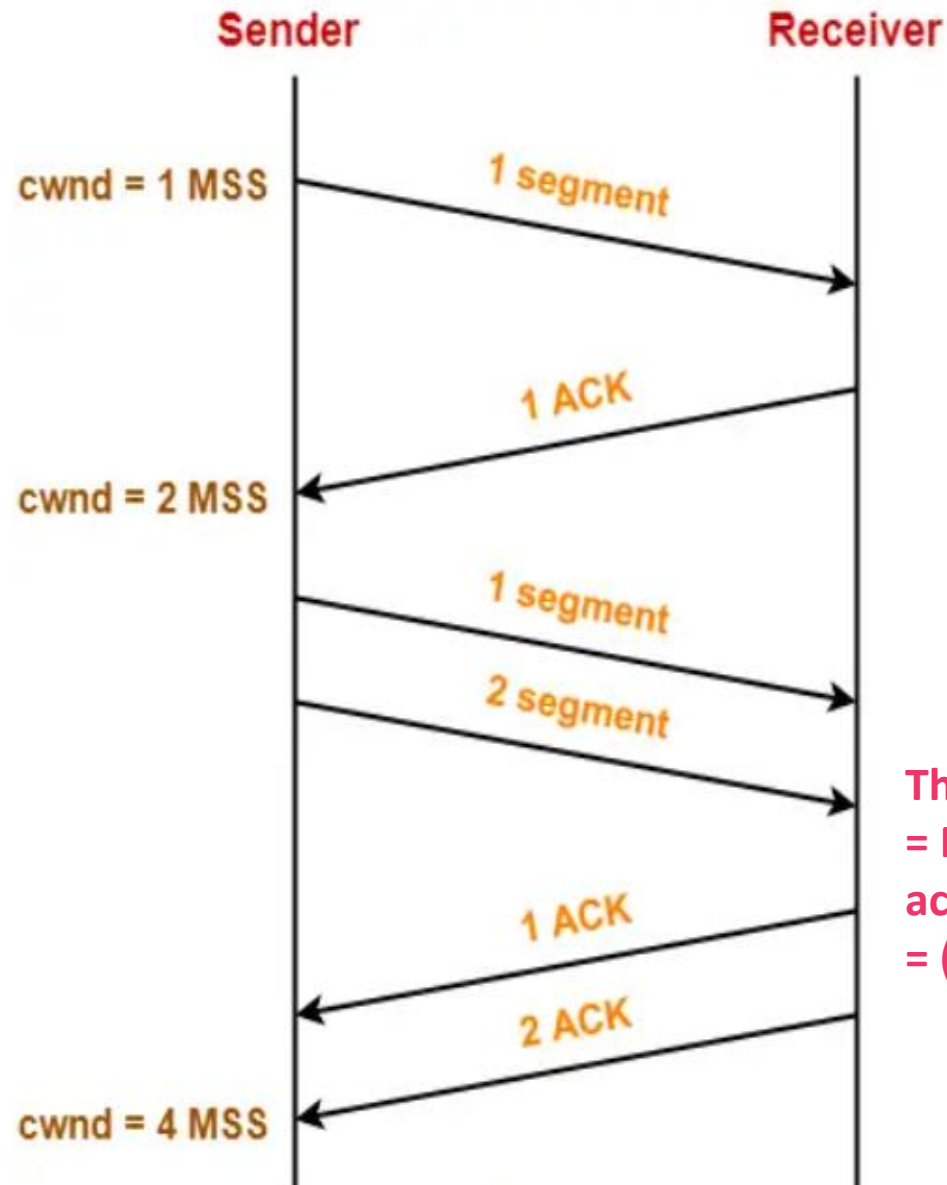
- Initially, sender sets congestion window size = Maximum Segment Size (1 MSS).
- After receiving each acknowledgment, sender increases the congestion window size by 1 MSS.
- In this phase, the size of congestion window increases **exponentially**.

Congestion window size = Congestion window size + Maximum segment size

TCP slow start

- when connection begins, increase rate exponentially until first loss event:
 - initially **cwnd** = 1 MSS
 - double **cwnd** every RTT
 - done by incrementing **cwnd** for every ACK received
- *summary*: initial rate is slow, but ramps up exponentially fast





The general formula for determining the size of the congestion window is $(2)^{\text{round trip time}}$

- After 1 round trip time, congestion window size = $(2)^1 = 2 \text{ MSS}$
- After 2 round trip time, congestion window size = $(2)^2 = 4 \text{ MSS}$
- After 3 round trip time, congestion window size = $(2)^3 = 8 \text{ MSS}$ and so on.

This phase continues until the congestion window size reaches the slow start threshold.

Threshold

= Maximum number of TCP segments that receiver window can accommodate / 2

= (Receiver window size / Maximum Segment Size) / 2

2. Congestion Avoidance Phase

After reaching the threshold,

- Sender increases the congestion window size **linearly** to avoid the congestion.
- On receiving each acknowledgement, sender increments the congestion window size by **1**.
- The followed formula is-

$$\text{Congestion window size} = \text{Congestion window size} + 1$$

- **Additive increment:** The size of cwnd(congestion window) increases additive. After each RTT **$cwnd = cwnd + 1$** .
- **Example:-** if the congestion window size is 20 segments and all 20 segments are successfully acknowledged within an RTT, the congestion window size would be increased to 21 segments in the next RTT. If all 21 segments are again successfully acknowledged, the congestion window size would be increased to 22 segments, and so on.

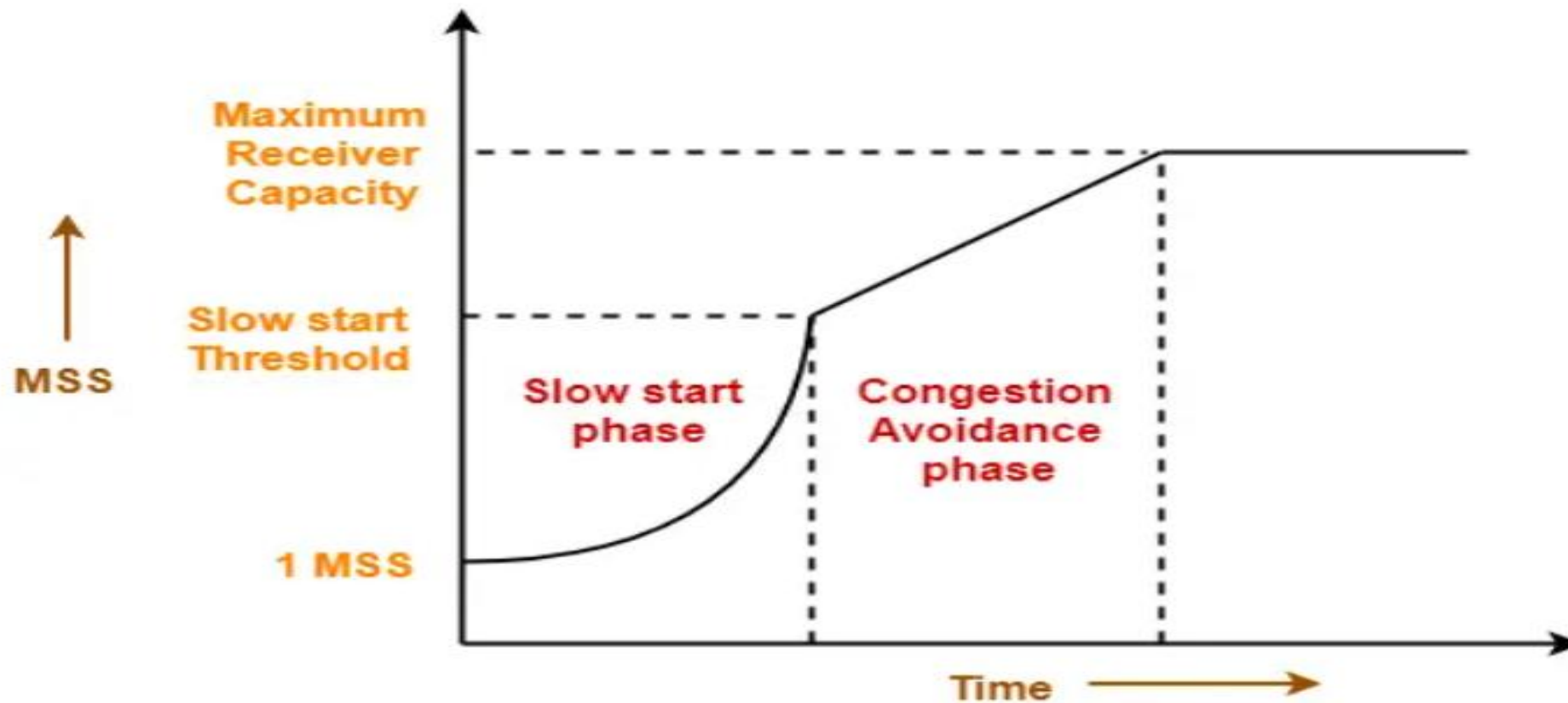
Initially cwnd = i

After 1 RTT, cwnd = i+1

2 RTT, cwnd = i+2

3 RTT, cwnd = i+3

This phase continues until the congestion window size becomes equal to the receiver window size.



3. Congestion Detection Phase

- **Multiplicative decrement:** If congestion occurs, the congestion window size is decreased. The only way a sender can guess that congestion has happened is the need to retransmit a segment. Retransmission is needed to recover a missing packet that is assumed to have been dropped by a router due to congestion.
- **Retransmission can occur in one of two cases:**

Case-01: Detection On Time Out-

- a) Timer expires before receiving the acknowledgement for a segment.
- b) Stronger possibility of congestion in the network.
- c) There are chances that a segment has been dropped in the network.

Reaction-

- In this case, sender reacts by-
 - a) Setting the slow start threshold to half of the current congestion window size.
 - b) Decreasing the congestion window size to 1 MSS.
 - c) Resuming the slow start phase.

Case-02: Detection On Receiving 3 Duplicate Acknowledgements

- a) Sender receives 3 duplicate acknowledgements for a segment.
- b) Weaker possibility of congestion in the network.
- c) There are chances that a segment has been dropped but few segments sent later may have reached.

Reaction-

- In this case, sender reacts by-
 - a) Setting the slow start threshold to half of the current congestion window size.
 - b) Decreasing the congestion window size to slow start threshold.
 - c) Resuming the congestion avoidance phase.

Problem #1

- Consider the effect of using slow start on a line with a 10 msec RTT and no congestion. The receiver window is 24 KB and the maximum segment size is 2 KB. How long does it take before the first full window can be sent?

Solution

Given:

- Receiver window size = 24 KB
- Maximum Segment Size = 2 KB
- RTT = 10 msec

- **Receiver window size in terms of MSS**

= Receiver window size / Size of 1 MSS

= 24 KB / 2 KB

= **12 MSS**

Slow Start Threshold:

- Slow start Threshold = Receiver window size / 2
- = 12 MSS / 2
- = 6 MSS

Slow Start Phase:

- Window size at the start of 1st transmission = 1 MSS
- Window size at the start of 2nd transmission = 2 MSS
- Window size at the start of 3rd transmission = 4 MSS
- Window size at the start of 4th transmission = 6 MSS
- Since the threshold is reached, so it marks the end of slow start phase.

Congestion Avoidance Phase:

- Window size at the start of 5th transmission = 7 MSS
- Window size at the start of 6th transmission = 8 MSS
- Window size at the start of 7th transmission = 9 MSS
- Window size at the start of 8th transmission = 10 MSS
- Window size at the start of 9th transmission = 11 MSS
- Window size at the start of 10th transmission = 12 MSS
- From here,
- Window size at the end of 9th transmission or at the start of 10th transmission is 12 MSS.
- Thus, 9 RTT's will be taken before the first full window can be sent.

- So,
- Time taken before the first full window is sent
- = 9 RTT's
- = 9 x 10 msec
- = **90 msec**

Problem #2

- Consider an instance of TCP's Additive Increase Multiplicative Decrease (AIMD) algorithm where the window size at the start of slow start phase is 2 MSS and the threshold at the start of first transmission is 8 MSS. Assume that a time out occurs during the fifth transmission. Find the congestion window size at the end of tenth transmission.
- A. 8 MSS
 - B. 14 MSS
 - C. 7 MSS
 - D. 12 MSS

Solution

Given:

- Window size at the start of slow start phase = 2 MSS
- Threshold at the start of first transmission = 8 MSS
- Time out occurs during 5th transmission

Slow Start Phase

- Window size at the start of 1st transmission = 2 MSS
- Window size at the start of 2nd transmission = 4 MSS
- Window size at the start of 3rd transmission = 8 MSS
- Since the threshold is reached, so it marks the end of slow start phase.
- Now, congestion avoidance phase begins.

Congestion Avoidance Phase

- Window size at the start of 4th transmission = 9 MSS
- Window size at the start of 5th transmission = 10 MSS
- It is given that time out occurs during 5th transmission.

TCP reacts by

- Setting the slow start threshold to half of the current congestion window size.
- Decreasing the congestion window size to 2 MSS (Given value is used).
- Resuming the slow start phase.

- So now,
- Slow start threshold = $10 \text{ MSS} / 2 = 5 \text{ MSS}$
- Congestion window size = 2 MSS

Slow Start Phase

- Window size at the start of 6th transmission = 2 MSS
 - Window size at the start of 7th transmission = 4 MSS
 - Window size at the start of 8th transmission = 5 MSS
-
- Since the threshold is reached, so it marks the end of slow start phase.

Congestion Avoidance Phase

- Window size at the start of 9th transmission = 6 MSS
- Window size at the start of 10th transmission = 7 MSS
- Window size at the start of 11th transmission = 8 MSS
- From here,
- Window size at the end of 10th transmission
= Window size at the start of 11th transmission
= **8 MSS**

Quick TCP Math

- Initial Seq No = 501. Sender sends 4500 bytes successfully acknowledged. Next sequence number to send is:
(A) 5000 (B) 5001 (C) 5002
- Next 1000 byte TCP segment received. Receiver acknowledges with ACK number:
(A) 5001 (B) 6000 (C) 6001

The TCP Sequence Number field is always set, even when there is no data in the segment. For example, the sequence number for this packet is X. The length for this packet is Y. If this packet is transferred to another side successfully, then the sequence number for the next packet is $X+Y$.

Solve!!!

On TCP connection, consider FIN, SYN packets will take 1 byte and ACK packets will take 0 bytes. Assume client and server are working on this connection.

Client and server selected random numbers for sequence numbers **100 and 500** respectively. After the connection is established, the client sent **100 bytes of two data packets**. While giving acknowledgment to client from server for those two data packets, what are SEQ NUM and ACK NUM?

- a) SEQ NUM = 501 and ACK NUM = 300
- b) SEQ NUM = 500 and ACK NUM = 299
- c) SEQ NUM = 500 and ACK NUM = 300
- d) SEQ NUM = 500 and ACK NUM = 301

SOLUTION

- **SEQ NUM (Server) = 500** (ACK packets do not change SEQ)
- **ACK NUM (Server) = 301** (acknowledging all received data)

UNIT – III

- **Network Layer:** Switching, Logical addressing - IPV4, IPV6; Address mapping - ARP, RARP, BOOTP and DHCP-Delivery, Forwarding and Unicast Routing protocols
- **Transport Layer:** Process to Process Communication, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), SCTP Congestion Control; Quality of Service (QoS), QoS improving techniques - Leaky Bucket and Token Bucket algorithms

- **Stream Control Transmission Protocol (SCTP)** is a new transport-layer protocol designed to combine some features of UDP and TCP in an effort to create a better protocol for multimedia communication.

SCTP Services

1. Process-to-Process Communication
2. Multiple Streams
3. Multihoming
4. Full-Duplex Communication
5. Connection-Oriented Service
6. Reliable Service

SCTP Services

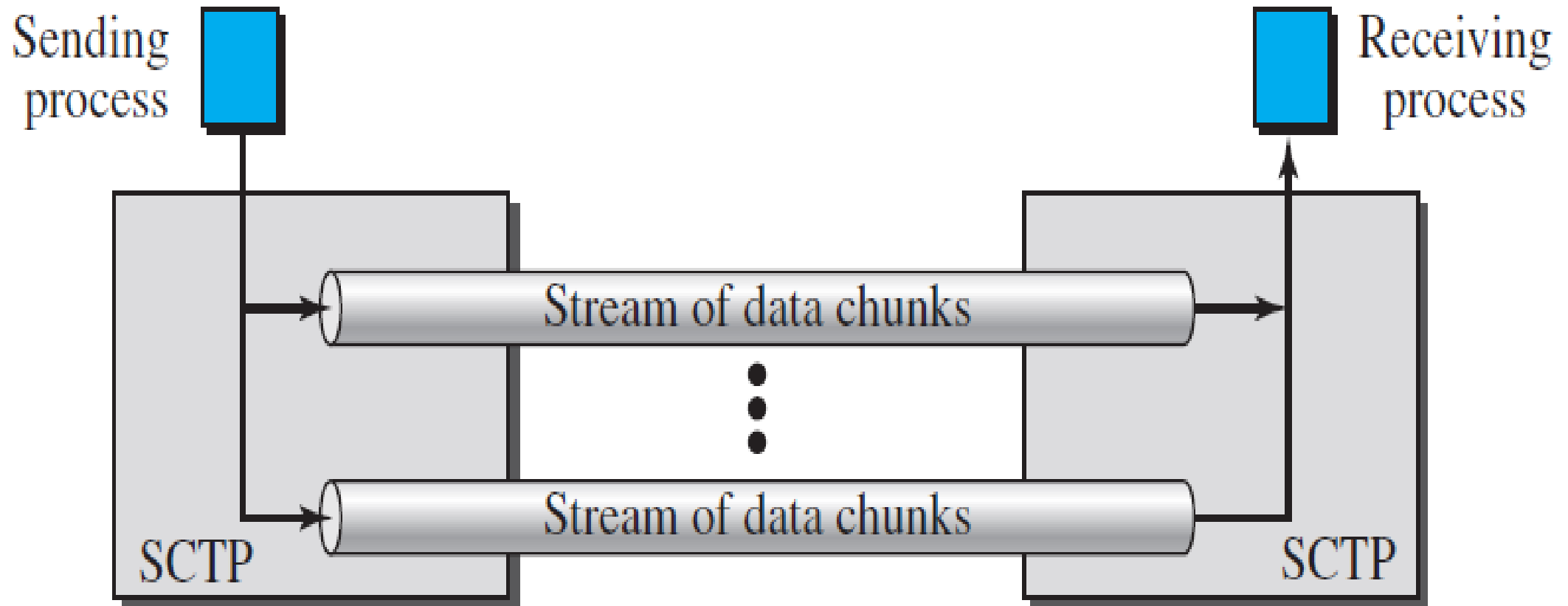
I. Process-to-Process Communication

- SCTP, like UDP or TCP, provides process-to-process communication.

2. Multiple Streams

- Each connection between a TCP client and a TCP server involves a single stream.
- The problem with this approach is that a loss at any point in the stream blocks the delivery of the rest of the data.
- This can be acceptable when we are transferring text; **it is not when we are sending real-time data such as audio or video.**
- SCTP allows multistream service in each connection, which is called association in SCTP terminology.
- If one of the streams is blocked, the other streams can still deliver their data.

Multiple-stream concept

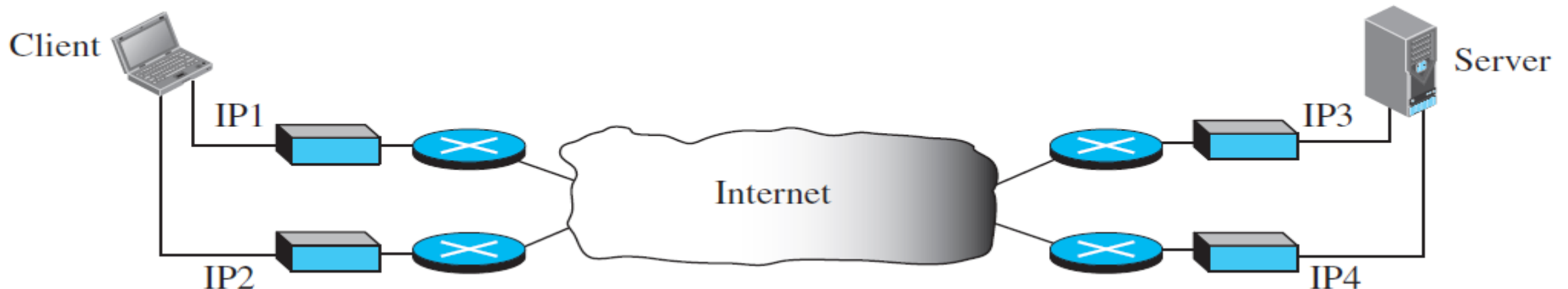


3. Multihoming

- A TCP connection involves one source and one destination IP address. This means that even if the sender or receiver is a multihomed host (connected to more than one physical address with multiple IP addresses), only one of these IP addresses per end can be utilized during the connection.
- An SCTP association supports multihoming service. The sending and receiving host can define multiple IP addresses in each end for an association.
- In this fault-tolerant approach, when one path fails, another interface can be used for data delivery without interruption.
- This fault-tolerant feature is very helpful when we are sending and receiving a real-time payload such as Internet telephony.

Multihoming concept

- The client is connected to two local networks with two IP addresses.
- The server is also connected to two networks with two IP addresses.
- The client and the server can make an association using four different pairs of IP addresses. However only one pair of IP addresses can be chosen for normal communication; the alternative is used if the main choice fails.



4. Full-Duplex Communication

Like TCP, SCTP offers full-duplex service, where data can flow in both directions at the same time. Each SCTP then has a sending and receiving buffer and packets are sent in both directions.

5. Connection-Oriented Service

Like TCP, SCTP is a connection-oriented protocol. However, in SCTP, a connection is called an association.

6. Reliable Service

SCTP, like TCP, is a reliable transport protocol. It uses an acknowledgment mechanism to check the safe and sound arrival of data.

An SCTP Association

- SCTP, like TCP, is a connection-oriented protocol.
- However, a connection in SCTP is called an association to emphasize multihoming.

A connection in SCTP is called an *association*.

SCTP Features

1. Transmission Sequence Number (TSN)
2. Stream Identifier (SI)
3. Stream Sequence Number (SSN)
4. Packets
5. Acknowledgment Number

SCTP Features

I. Transmission Sequence Number (TSN)

- The unit of data in SCTP is a data chunk
- Data transfer in SCTP is controlled by numbering the data chunks.
- SCTP uses a transmission sequence number (TSN) to number the data chunks. In other words, the **TSN in SCTP plays a role analogous to the sequence number in TCP.**
- TSNs are 32 bits long and randomly initialized between **0 and $2^{32} - 1$.**
- Each data chunk must carry the corresponding TSN in its header.

2. Stream Identifier (SI)

- Each stream in SCTP needs to be identified using a stream identifier (SI).
- Each data chunk must carry the SI in its header so that when it arrives at the destination, it can be properly placed in its stream.
- **The SI is a 16-bit number starting from 0.**

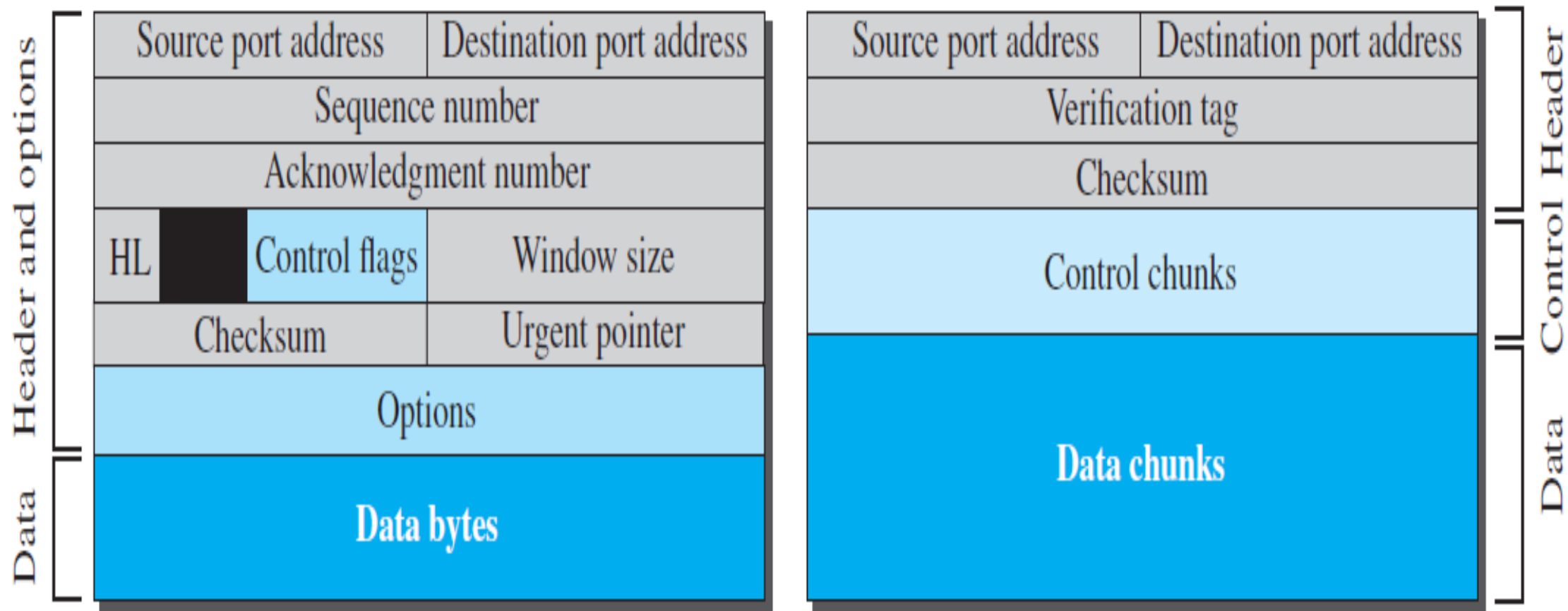
3. Stream Sequence Number (SSN)

- When a data chunk arrives at the destination SCTP, it is delivered to the appropriate stream and in the proper order.
- SCTP defines each data chunk in each stream with a stream sequence number (SSN).

4. Packets

- In TCP, a segment carries data and control information.
- Data are carried as a collection of bytes; control information is defined by six control flags in the header.
- In SCTP, Data are carried as data chunks, control information as control chunks.
- Several control chunks and data chunks can be packed together in a packet.
- **A packet in SCTP plays the same role as a segment in TCP.**

Comparison between a TCP segment and an SCTP packet



A segment in TCP

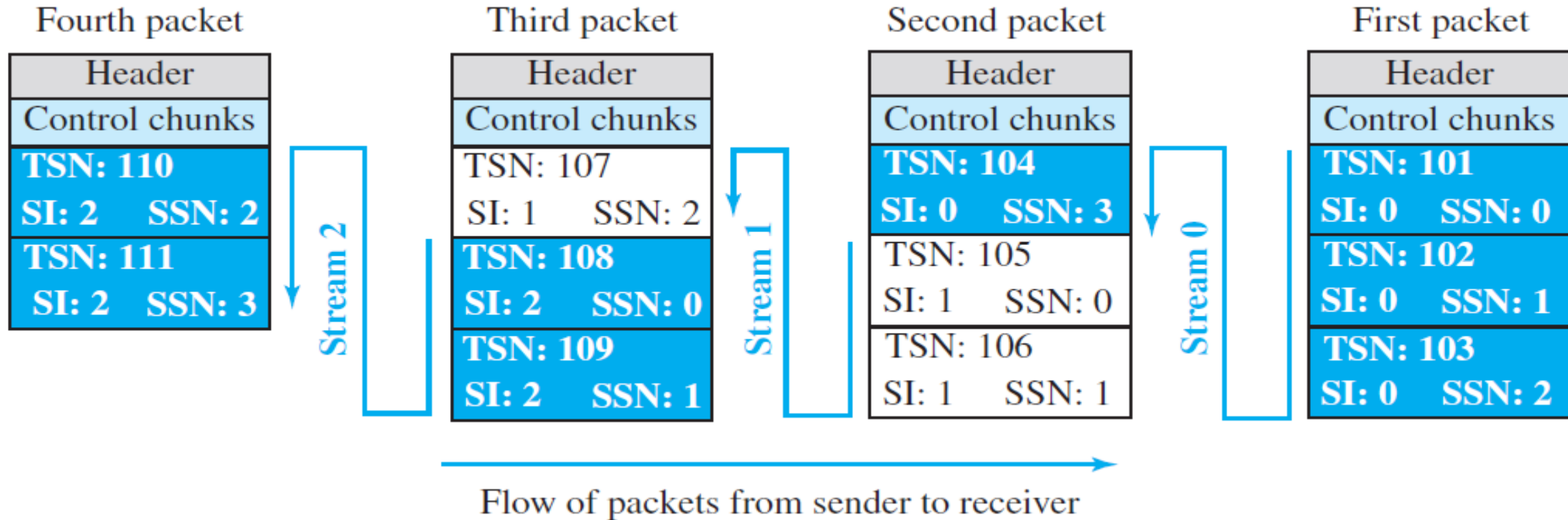
A packet in SCTP

The Verification Tag is a 32-bit random number used to uniquely identify an association between two endpoints. It helps in validating incoming packets and preventing old or malicious packets from interfering with an active connection.

Packets, data chunks, and streams

- In SCTP, we have data chunks, streams, and packets.
- An association may send many packets, a packet may contain several chunks, and chunks may belong to different streams.
- Suppose that process A needs to send 11 messages to process B in three streams.
- The first four messages are in the first stream, the second three messages are in the second stream, and the last four messages are in the third stream.
- Therefore, we have 11 data chunks in three streams.
- The application process delivers 11 messages to SCTP, where each message is earmarked for the appropriate stream.
- It delivers all messages belonging to the first stream first, all messages belonging to the second stream next, and finally, all messages belonging to the last stream.
- Network allows only three data chunks per packet, which means that we need four packets, as shown in Figure

Packets, data chunks, and streams



Data chunks in stream 0 are carried in the first and part of the second packet; those in stream 1 are carried in the second and the third packet; those in stream 2 are carried in the third and fourth packet. Note that each data chunk needs three identifiers: TSN, SI, and SSN. TSN is a cumulative number and is used for flow control and error control. SI defines the stream to which the chunk belongs. SSN defines the chunk's order in a particular stream. In our example, SSN starts from 0 for each stream.

5. Acknowledgment Number

- TCP acknowledgment numbers are byte-oriented and refer to the sequence numbers. SCTP acknowledgment numbers are chunk-oriented. They refer to the TSN.
- A second difference between TCP and SCTP acknowledgments is the control information. To acknowledge segments that carry only control information, TCP uses a sequence number and acknowledgment number (for example, a SYN segment needs to be acknowledged by an ACK segment).
- In SCTP, the control information is carried by control chunks, which do not need a TSN. These control chunks are acknowledged by another control chunk of the appropriate type (some need no acknowledgment).
- For example, an INIT control chunk is acknowledged by an INIT-ACK chunk. There is no need for a sequence number or an acknowledgment number.

When to Use TCP vs. SCTP

Use Case	Recommended Protocol
Web browsing, email, file transfer (FTP), SSH	TCP
VoIP, video conferencing, real-time gaming	SCTP
Signaling in telecom networks (e.g., SS7 replacement)	SCTP
High-availability applications (e.g., banking, finance systems)	SCTP (Multi-Homing support)

UNIT – III

- **Network Layer:** Switching, Logical addressing - IPV4, IPV6; Address mapping - ARP, RARP, BOOTP and DHCP-Delivery, Forwarding and Unicast Routing protocols
- **Transport Layer:** Process to Process Communication, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), SCTP Congestion Control; Quality of Service (QoS), QoS improving techniques - Leaky Bucket and Token Bucket algorithms

Quality of Service (QoS)

- The Internet was originally designed for best-effort service without guarantee of predictable performance.
- Best-effort service is often sufficient for a traffic that is not sensitive to delay, such as file transfers and e-mail. Such a traffic is called elastic because it can stretch to work under delay conditions; it is also called available bit rate because applications can speed up or slow down according to the available bit rate.
- The real-time traffic generated by some multimedia applications is delay sensitive and therefore requires guaranteed and predictable performance.
- Quality of service(QoS) is an internetworking issue that refers to a set of techniques and mechanisms that guarantee the performance of the network to deliver predictable service to an application program.

I. Scheduling

- Treating packets (datagrams) in the Internet based on their required level of service can mostly happen at the routers. It is at a router that a packet may be delayed, suffer from jitters, be lost, or be assigned the required bandwidth.
- A good scheduling technique treats the different flows in a fair and appropriate manner.
- Several scheduling techniques are designed to improve the quality of service. Three of them here:
- **FIFO queuing, priority queuing, and weighted fair queuing.**

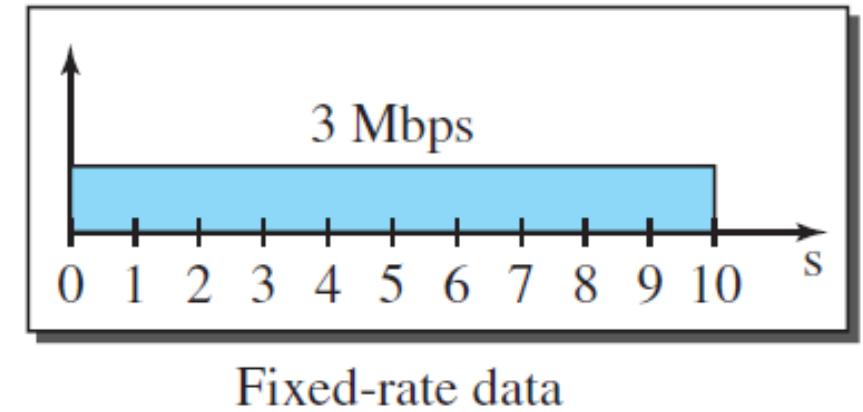
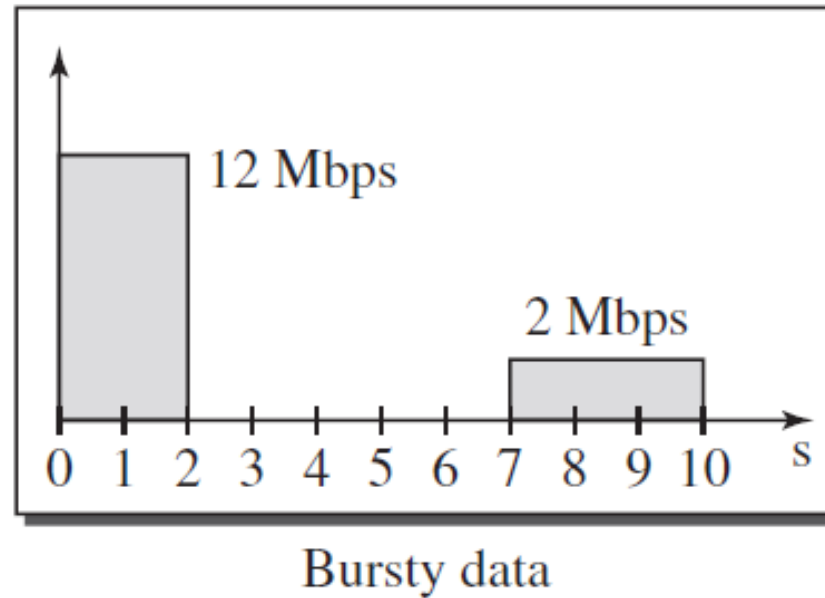
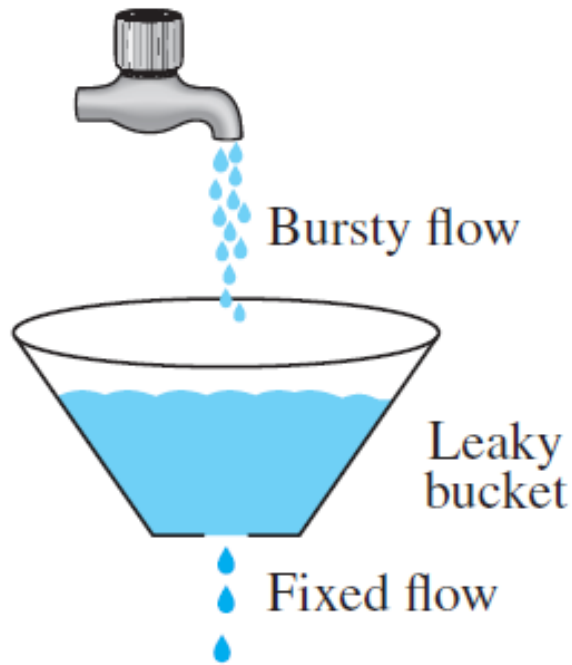
2. Traffic Shaping or Policing

- To control the amount and the rate of traffic is called traffic shaping or traffic policing.
- The first term is used when the traffic leaves a network; the second term is used when the data enters the network.
- Two techniques can shape or police the traffic: **leaky bucket**, and **token bucket**.

Leaky Bucket

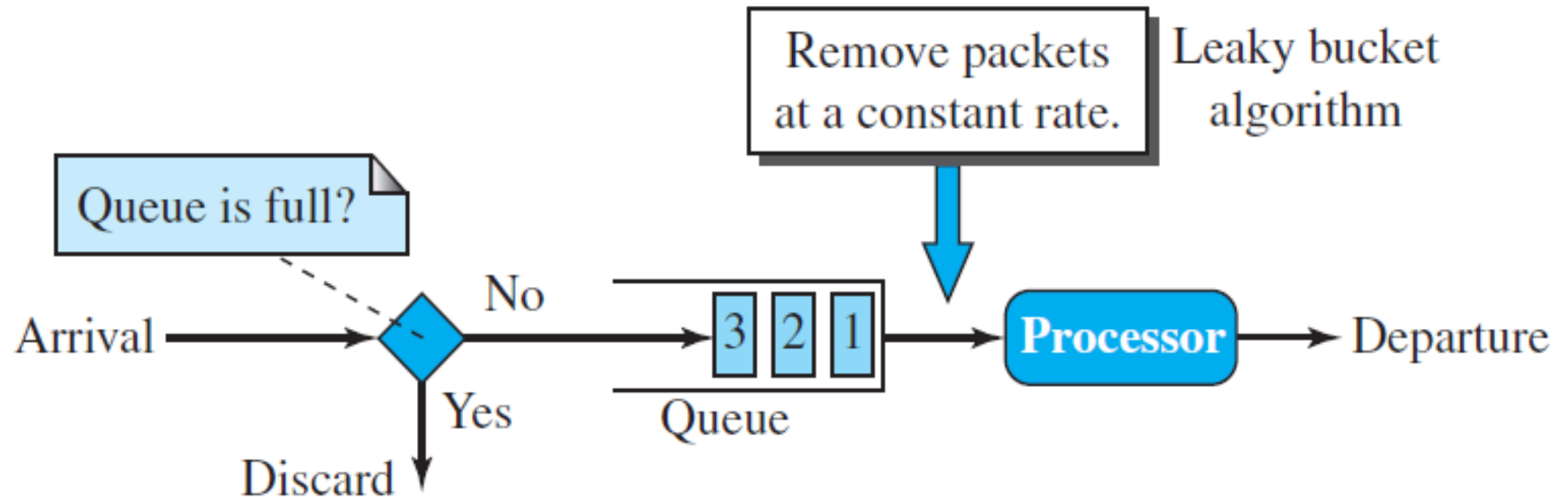
- If a bucket has a small hole at the bottom, the water leaks from the bucket at a constant rate as long as there is water in the bucket. The rate at which the water leaks does not depend on the rate at which the water is input unless the bucket is empty.
- If the bucket is full, the water overflows. The input rate can vary, but the output rate remains constant.
- Similarly, in networking, a technique called leaky bucket can smooth out bursty traffic.
- Bursty chunks are stored in the bucket and sent out at an average rate.

leaky bucket and its effects



In the figure, we assume that the network has committed a bandwidth of 3 Mbps for a host. The use of the leaky bucket shapes the input traffic to make it conform to this commitment. In Figure, the host sends a burst of data at a rate of 12 Mbps for 2 seconds, for a total of 24 Mb of data. The host is silent for 5 seconds and then sends data at a rate of 2 Mbps for 3 seconds, for a total of 6 Mb of data. In all, the host has sent 30 Mb of data in 10 seconds. The leaky bucket smooths the traffic by sending out data at a rate of 3 Mbps during the same 10 seconds. Without the leaky bucket, the beginning burst may have hurt the network by consuming more bandwidth than is set aside for this host. We can also see that the leaky bucket may prevent congestion

Leaky bucket implementation



A simple leaky bucket implementation is shown in Figure . A FIFO queue holds the packets. If the traffic consists of fixed-size packets (e.g., cells in ATM networks), the process removes a fixed number of packets from the queue at each tick of the clock. If the traffic consists of variable-length packets, the fixed output rate must be based on the number of bytes or bits.

The following is an algorithm for variable-length packets:

1. Initialize a counter to n at the tick of the clock.
2. If n is greater than the size of the packet, send the packet and decrement the counter by the packet size. Repeat this step until the counter value is smaller than the packet size.
3. Reset the counter to n and go to step 1.

A leaky bucket algorithm shapes bursty traffic into fixed-rate traffic by averaging the data rate. It may drop the packets if the bucket is full.

Token Bucket

- The leaky bucket is very restrictive. It does not credit an idle host.
- For example, if a host is not sending for a while, its bucket becomes empty. Now if the host has bursty data, the leaky bucket allows only an average rate. The time when the host was idle is not taken into account.
- On the other hand, the token bucket algorithm allows idle hosts to accumulate credit for the future in the form of tokens.
- Assume the capacity of the bucket is c tokens and tokens enter the bucket at the rate of r tokens per second. The system removes one token for every cell of data sent.
- The maximum number of cells that can enter the network during any time interval of length t is shown below.

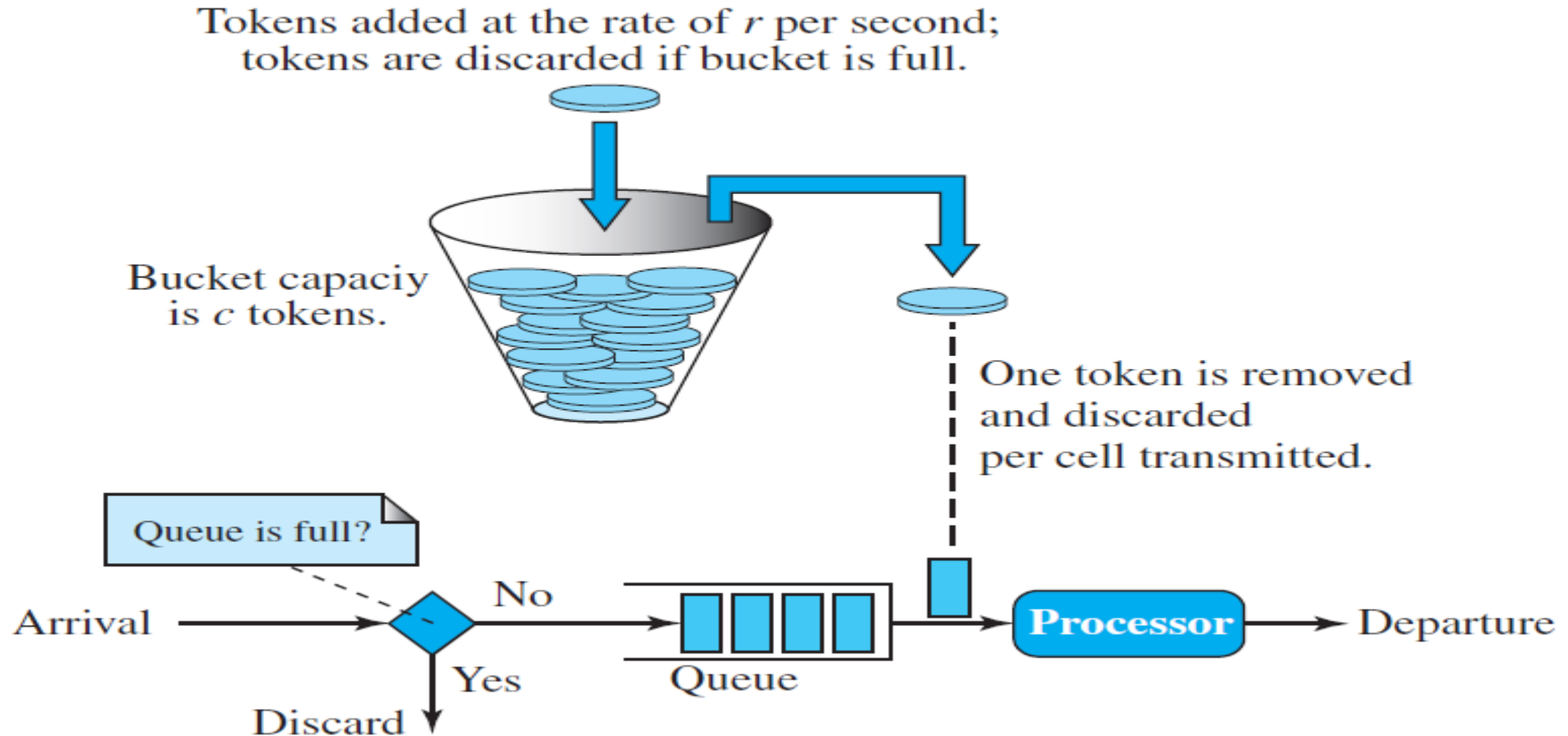
$$\text{Maximum number of packets} = r \times t + c$$

- The maximum average rate for the token bucket is shown below

Maximum average rate = $(r \times t + c)/t$ packets per second

- This means that the token bucket limits the average packet rate to the network

Token bucket



Example

- Let's assume that the bucket capacity is 10,000 tokens and tokens are added at the rate of 1000 tokens per second. If the system is idle for 10 seconds (or more), the bucket collects 10,000 tokens and becomes full. Any additional tokens will be discarded. The maximum average rate is shown below.
 - **Maximum average rate = $(1000t + 10,000)/t$**
- The token bucket can easily be implemented with a counter. The counter is initialized to zero. Each time a token is added, the counter is incremented by 1. Each time a unit of data is sent, the counter is decremented by 1. When the counter is zero, the host cannot send data.

The token bucket allows bursty traffic at a regulated maximum rate.

Combining Token Bucket and Leaky Bucket

- The two techniques can be combined to credit an idle host and at the same time regulate the traffic.
- The leaky bucket is applied after the token bucket; the rate of the leaky bucket needs to be higher than the rate of tokens dropped in the bucket.

3.Resource Reservation

- A flow of data needs resources such as a buffer, bandwidth, CPU time, and so on. The quality of service is improved if these resources are reserved beforehand.
- QoS model called Integrated Services, which depends heavily on resource reservation to improve the quality of service.

4. Admission Control

- Admission control refers to the mechanism used by a router or a switch to accept or reject a flow based on predefined parameters called flow specifications.
- Before a router accepts a flow for processing, it checks the flow specifications to see if its capacity can handle the new flow. It takes into account bandwidth, buffer size, CPU speed, etc., as well as its previous commitments to other flows.
- Admission control in ATM networks is known as Connection Admission Control (CAC), which is a major part of the strategy for controlling congestion.

The End