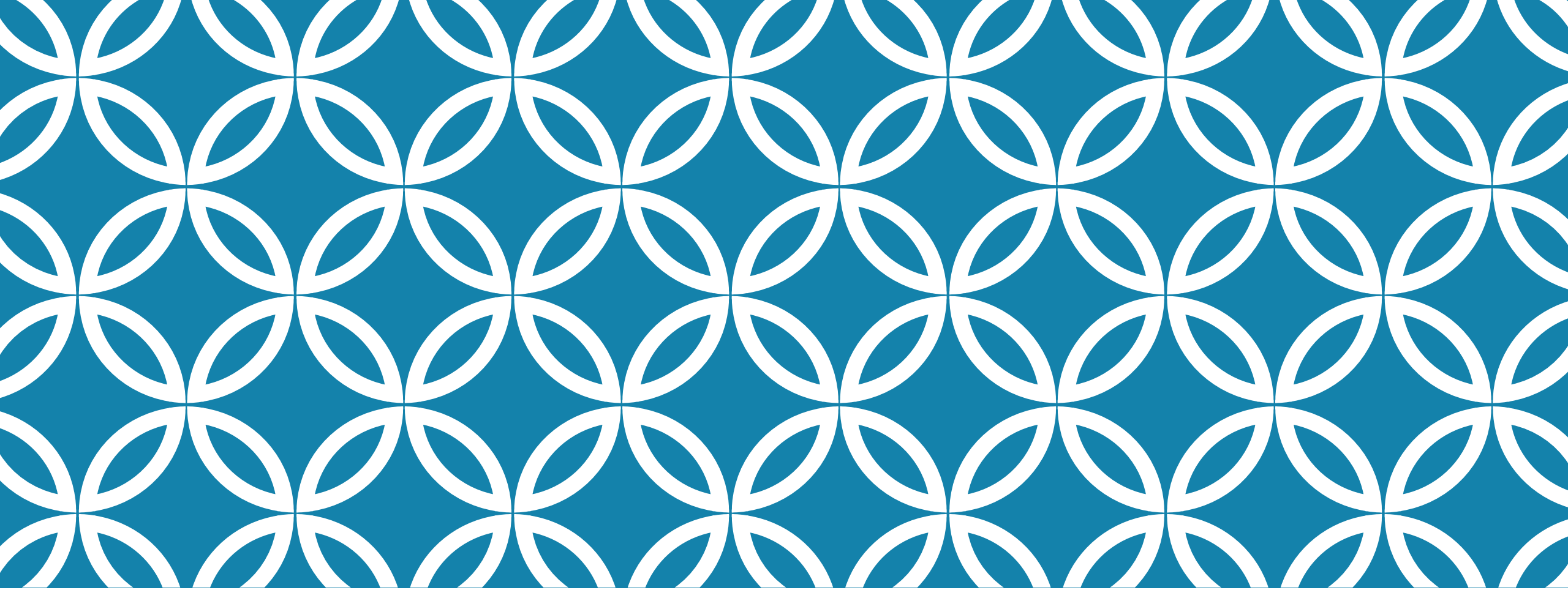


UNIT 2 – TRANSPORT LAYER

SYLLABUS

Unit 2

Transport layer: Multiplexing and Demultiplexing, Connectionless Transport-UDP: UDP Segment Structure, UDP Checksum,Go-Back-N, Selective Repeat, Connection-OrientedTransport-TCP: The TCP Connection, TCP Segment Structure, Round-TripTime Estimation and Timeout, Reliable Data Transfer, Flow Control, TCPConnection Management, TCP congestion control.

INTRODUCTION

- Located between the Application layer and the Network layer.
- A transport layer protocol provides for logical connection between application processes running on different hosts.
- Process to process communication.

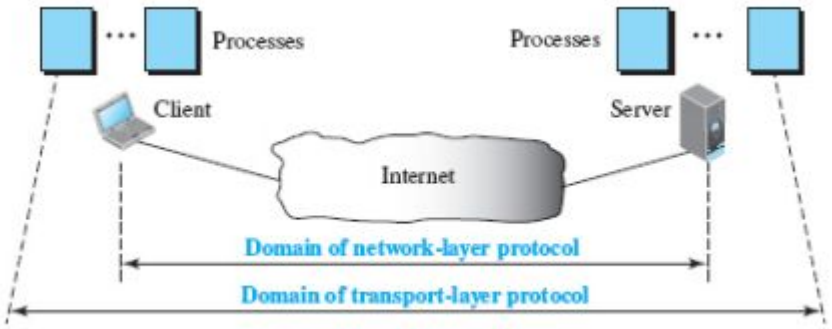
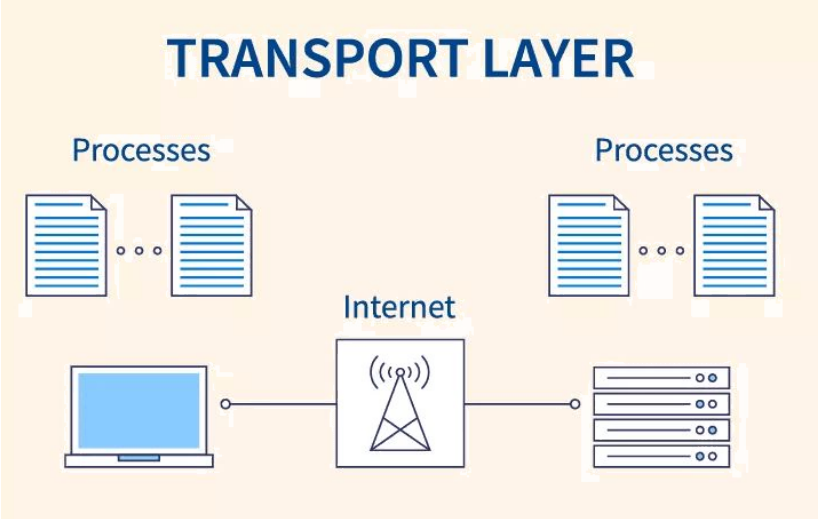
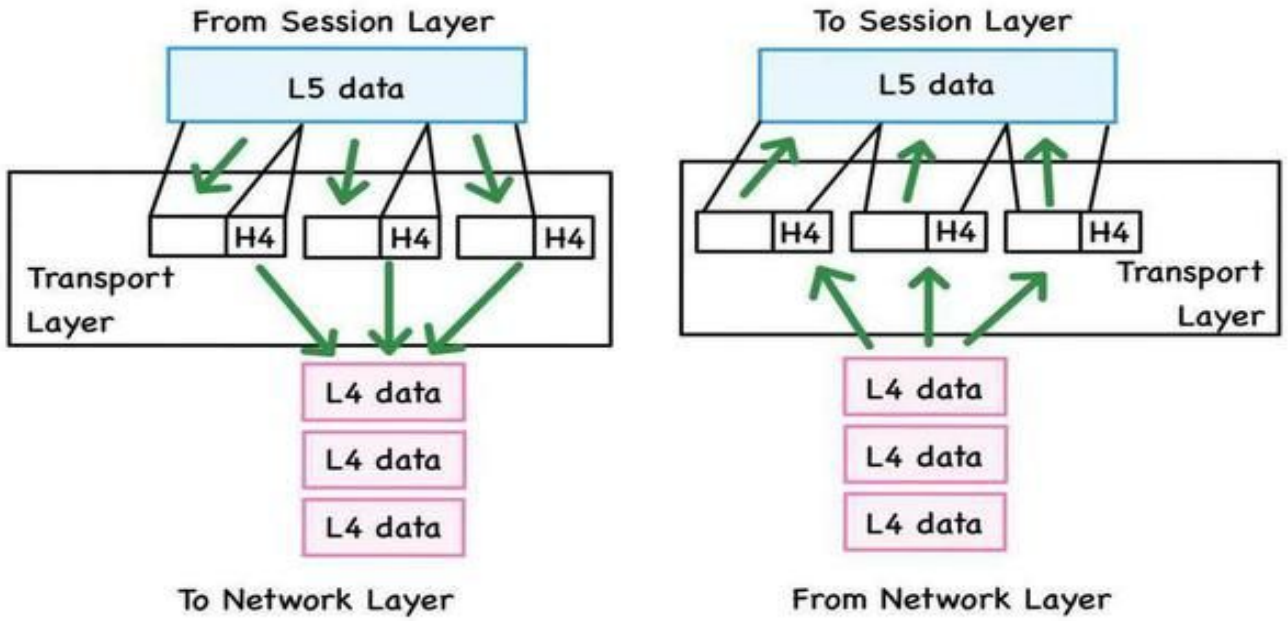


Figure :Network layer versus transport layer



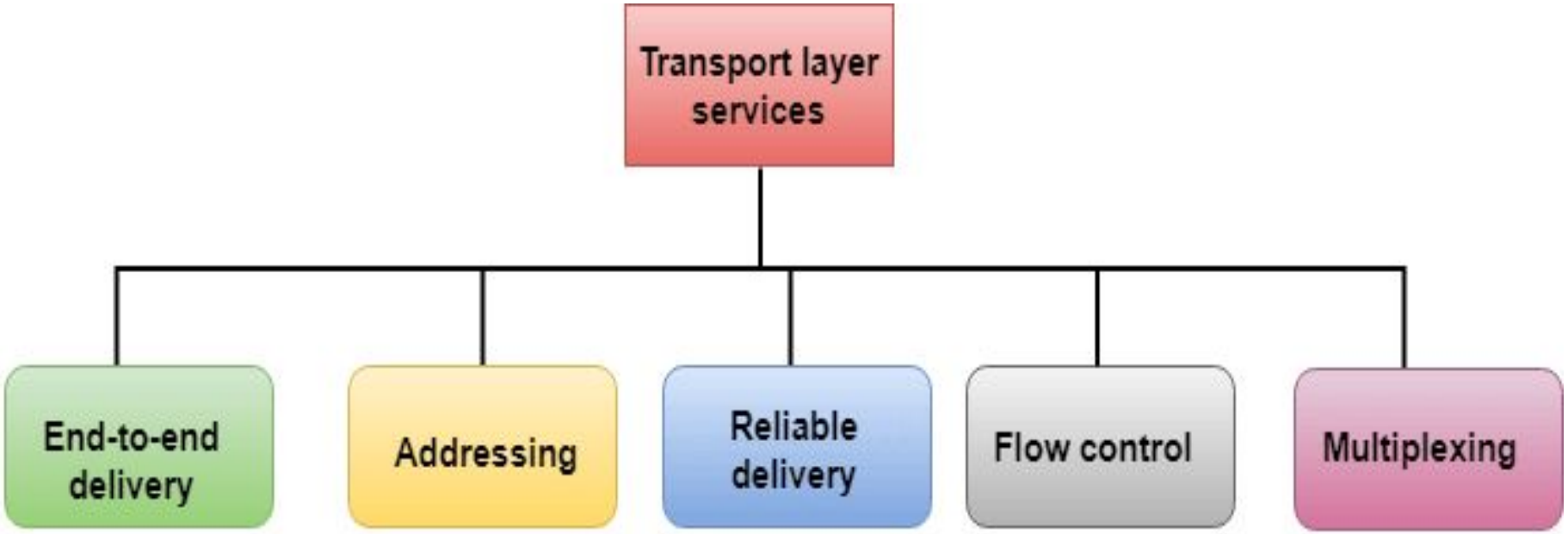
- The transport layer converts the application layer messages it receives from a sending application process into transport layer packets, known as transport layer **Segments.**
- Transport layer header is added.



Overview of the Transport Layer in the Internet

- Two distinct transport layer protocols:
- UDP (User Datagram Protocol):
  - Unreliable, Connectionless Service.
- TCP(Transmission Control Protocol):
  - Reliable, Connection oriented.
- When designing a network application, the application developer must specify one of these two transport protocol.

Transport Layer Services

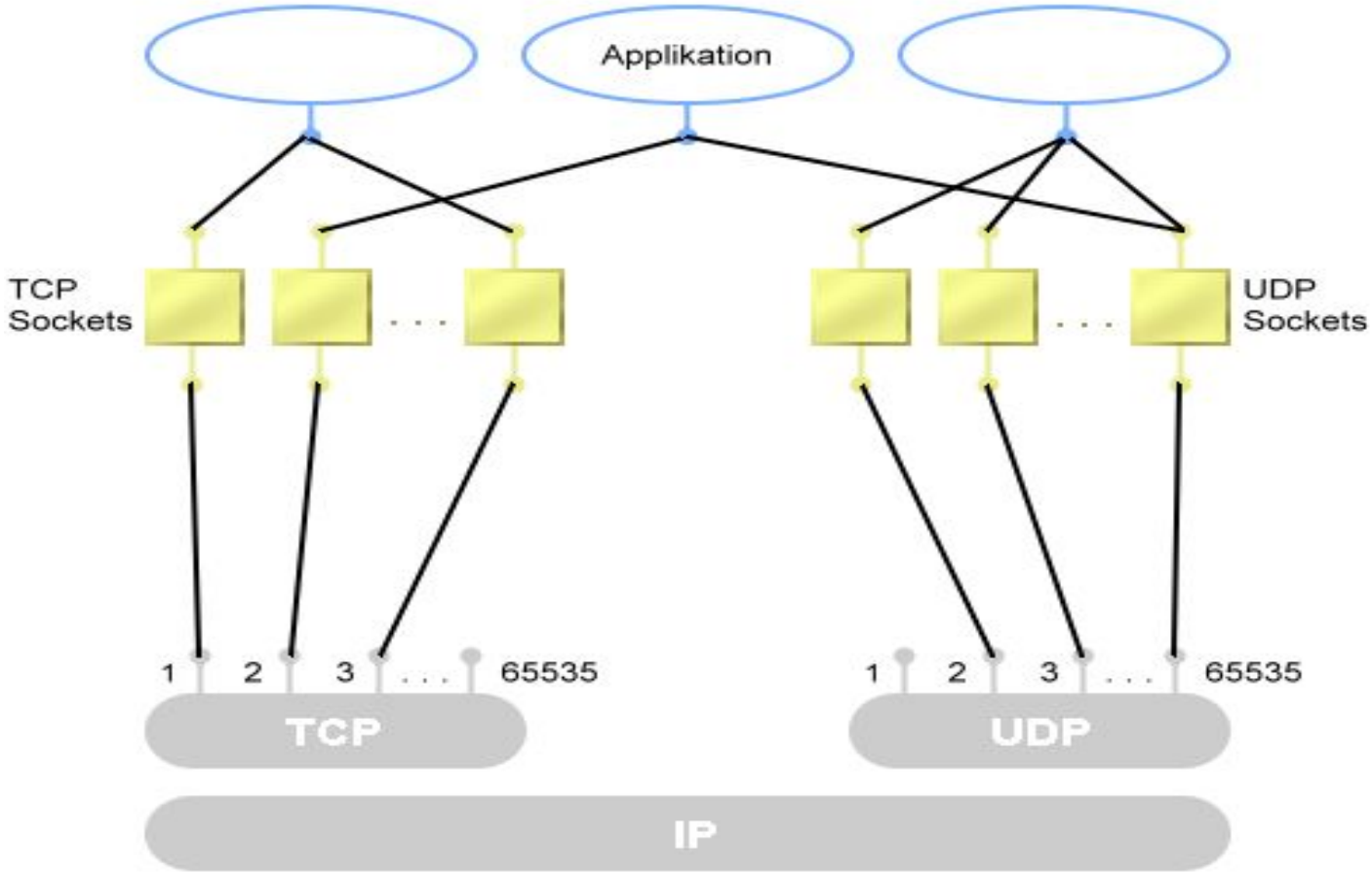


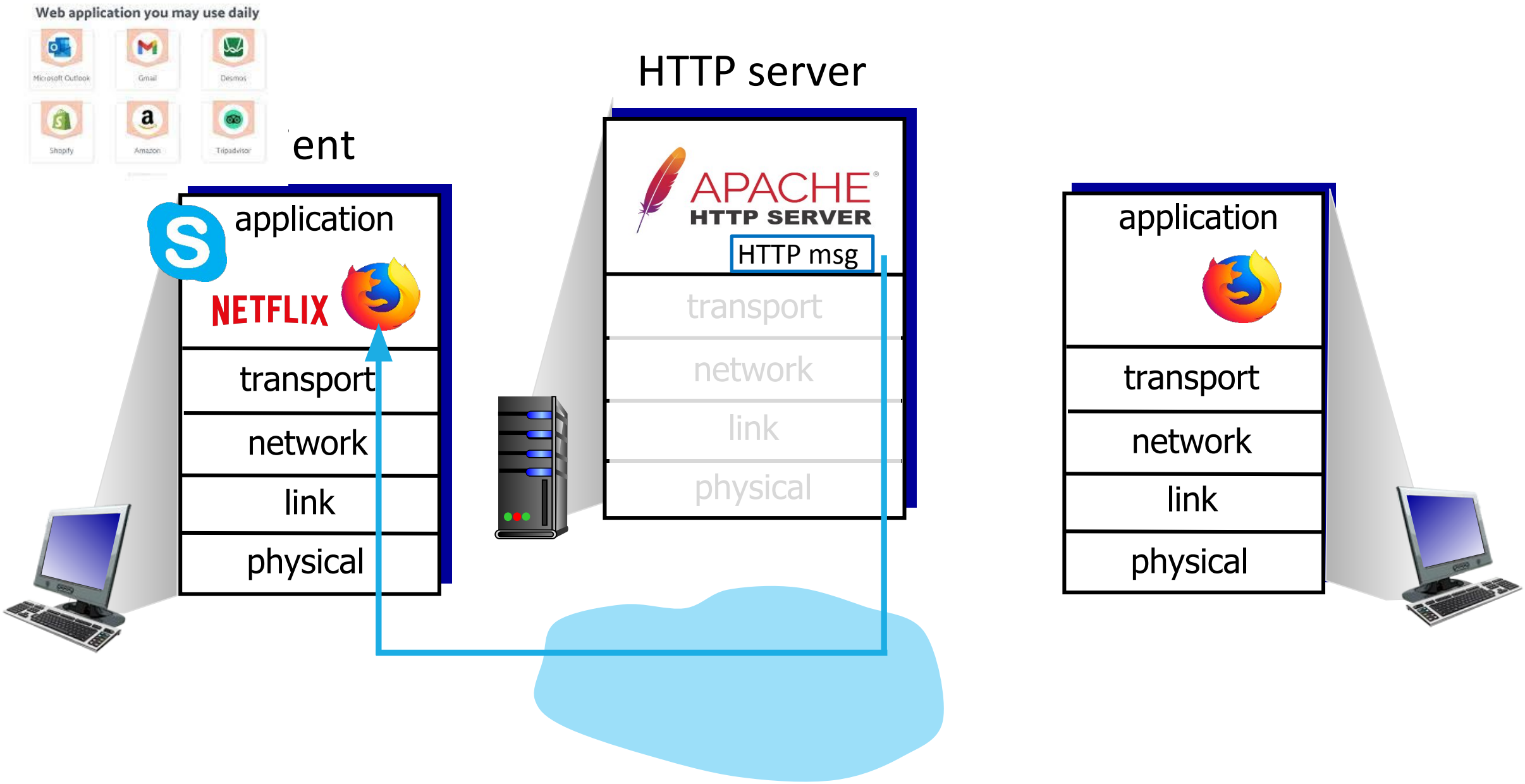
MULTIPLEXING AND DEMULTIPLEXING

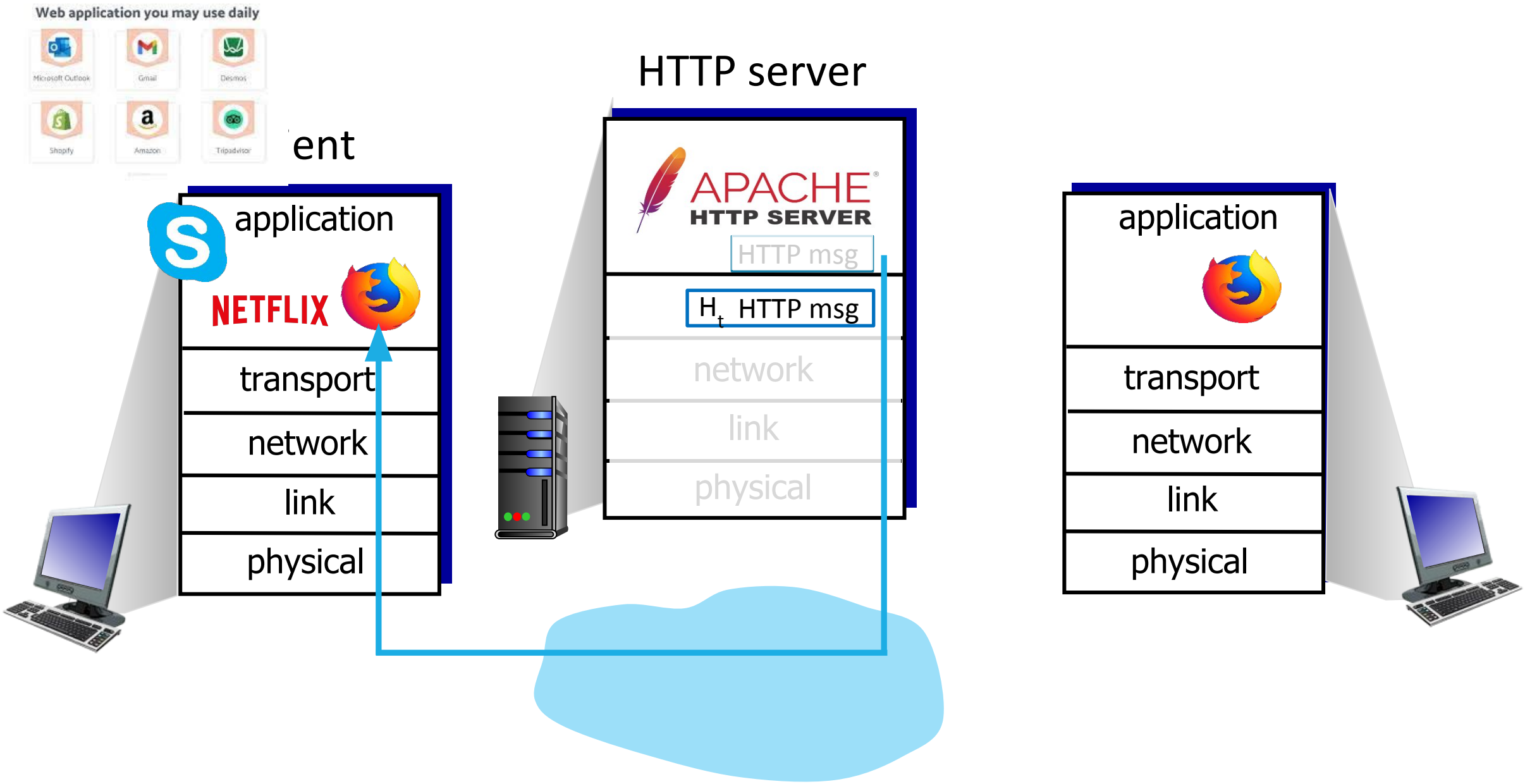
- The transport layer has the responsibility of delivering the data in these segments to the appropriate application process running in the host.

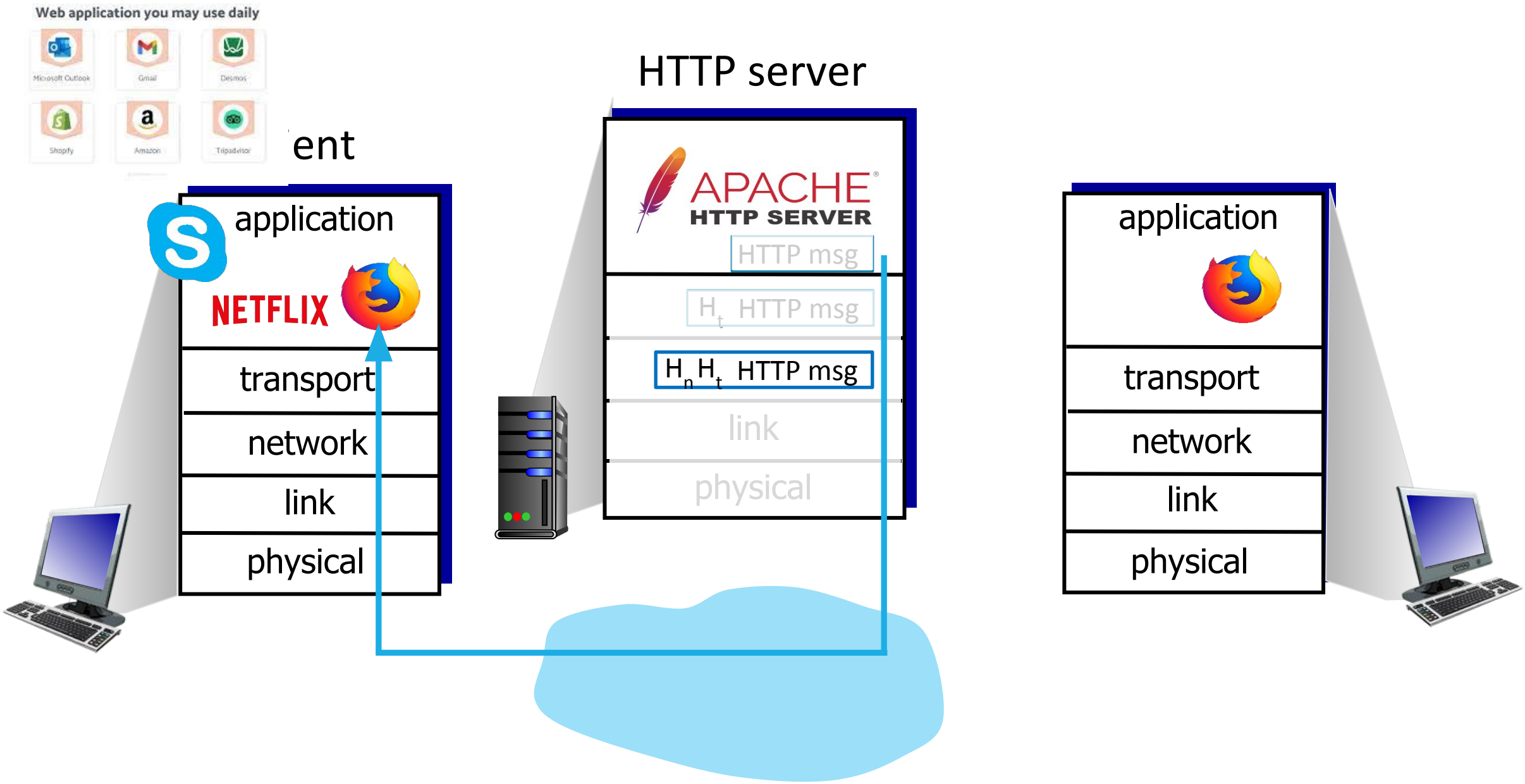
- Example:

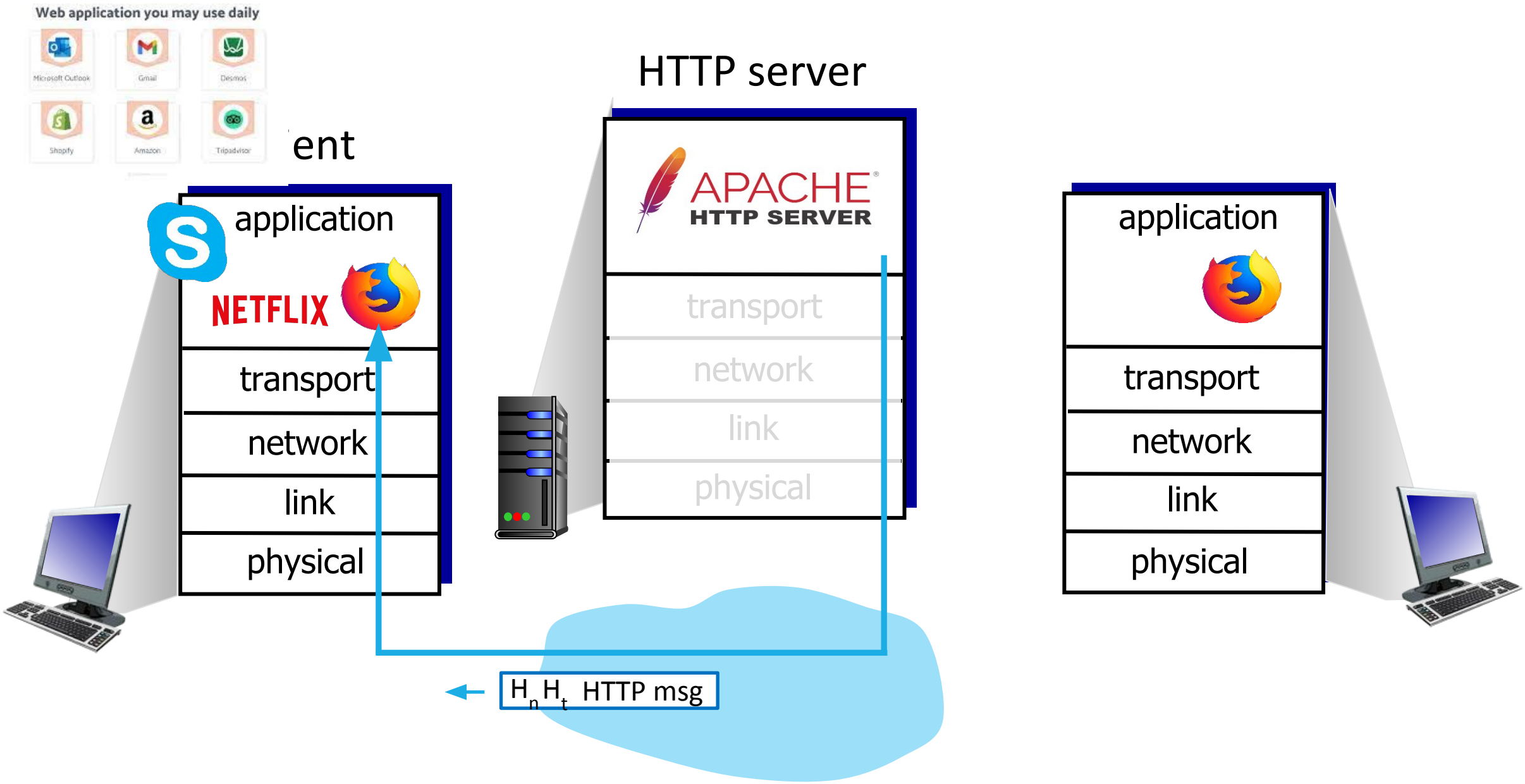
- Suppose you are sitting in front of your computer, and you are downloading web pages while running one FTP session and two Telnet sessions. You have four application process running – two telnet processes, one FTP process and one HTTP process.
  - When transport layer in computer receives data from the network layer, where it has to be directed??


Applikation
TCP
Sockets
UDP
Sockets
1
2
3
65535
1
2
3
65535
TCP
UDP
IP


Web application you may use daily
Microsoft Outlook
Gmail
Desmos
Shopify
Amazon
Tripadvisor
ent
application
NETFLIX
transport
network
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
transport
network
link
physical
application
transport
network
link
physical


Web application you may use daily
Microsoft Outlook
Gmail
Desmos
Shopify
Amazon
Tripadvisor
ent
application
NETFLIX
transport
network
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
network
link
physical
application
transport
network
link
physical


Web application you may use daily
Microsoft Outlook
Gmail
Desmos
Shopify
Amazon
Tripadvisor
ent
application
NETFLIX
transport
network
link
physical
HTTP server
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
H_n H_t HTTP msg
link
physical
application
transport
network
link
physical


Web application you may use daily
Microsoft Outlook
Gmail
Desmos
Shopify
Amazon
Tripadvisor
ent
application
NETFLIX
transport
network
link
physical
HTTP server
APACHE
HTTP SERVER
transport
network
link
physical
application
transport
network
link
physical
H_n H_t HTTP msg

- At the receiving end, the transport layer examines these fields to identify the receiving socket and then directs the segment to that socket. This job of delivering the data in a transport-layer segment to the correct socket is called **Demultiplexing**.

- The job of gathering data chunks at the source host from different sockets, encapsulating each data chunk with header information (that will later be used in demultiplexing) to create segments, and passing the segments to the network layer is called **Multiplexing**.

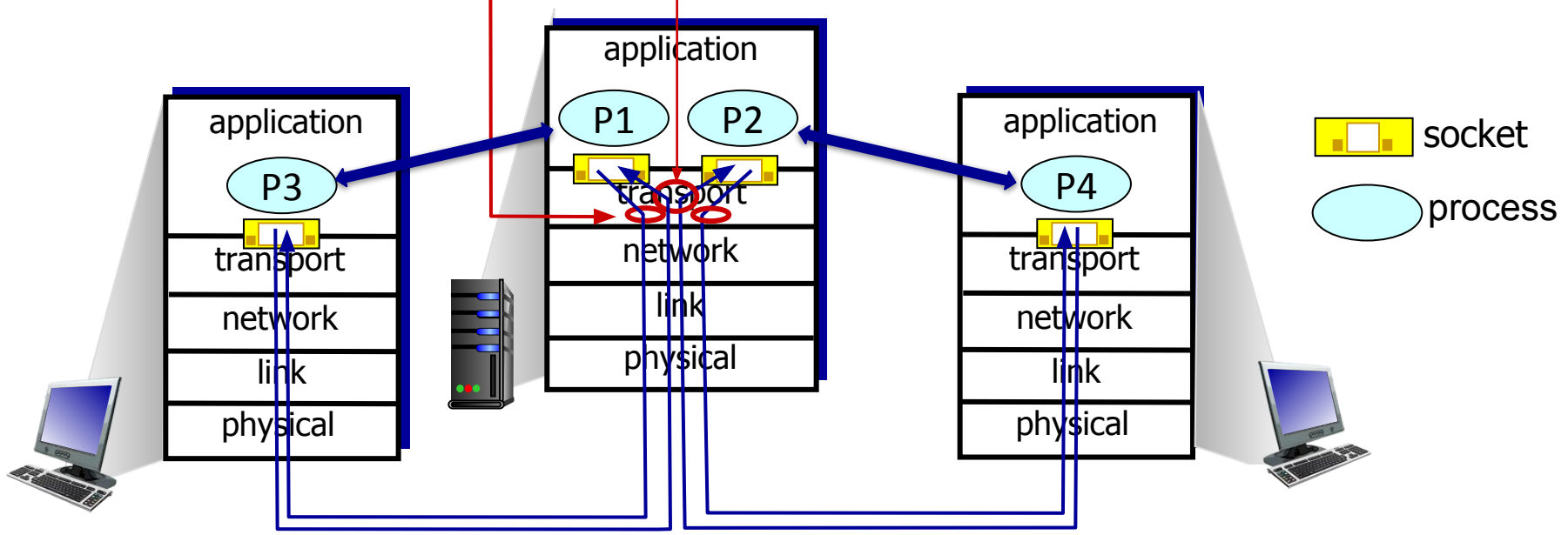
MULTIPLEXING/DEMULTIPLEXING

Multiplexing at sender:

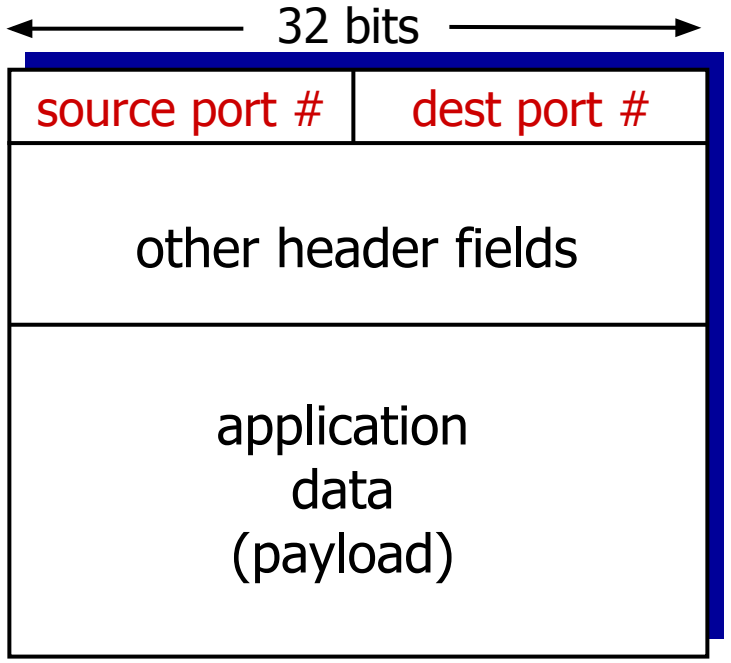
Handle data from multiple sockets, add transport header (later used for demultiplexing)

Demultiplexing at receiver:

Use header info to deliver received segments to correct socket



- Transport-layer multiplexing requires
  (1) That sockets have unique identifiers, and
  (2) That each segment have special fields that indicate the socket to which the segment is to be delivered.
- These special fields, are the source port number field and the destination port number field.



TCP/UDP segment format

- Each port number is a 16-bit number, ranging from 0 to 65535.
- The port numbers ranging from 0 to 1023 are called **Well-known port numbers** and are restricted.
  - Example: HTTP (which uses port number 80) and FTP (which uses port number 21).

- The Demultiplexing service requires:
  - Each socket in the host could be assigned a port number, and when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket.

Connectionless Multiplexing and Demultiplexing :

- A host can create a UDP socket with the line:

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

- UDP socket via the socket bind() method:

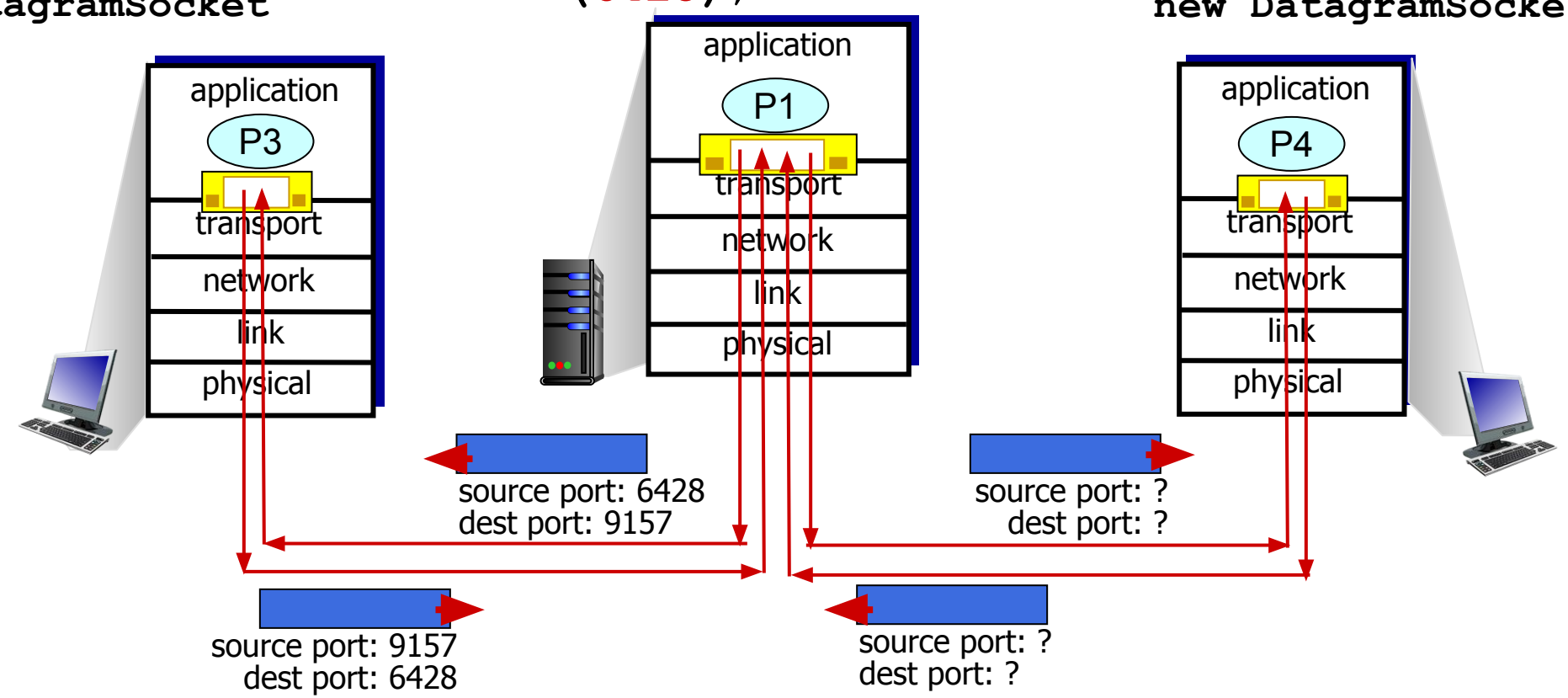
```
clientSocket.bind((’’, 19157))
```

CONNECTIONLESS DEMULTIPLEXING: AN EXAMPLE

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

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

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



CONNECTION-ORIENTED DEMULTIPLEXING

- 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

TCP Client – Server program Example:

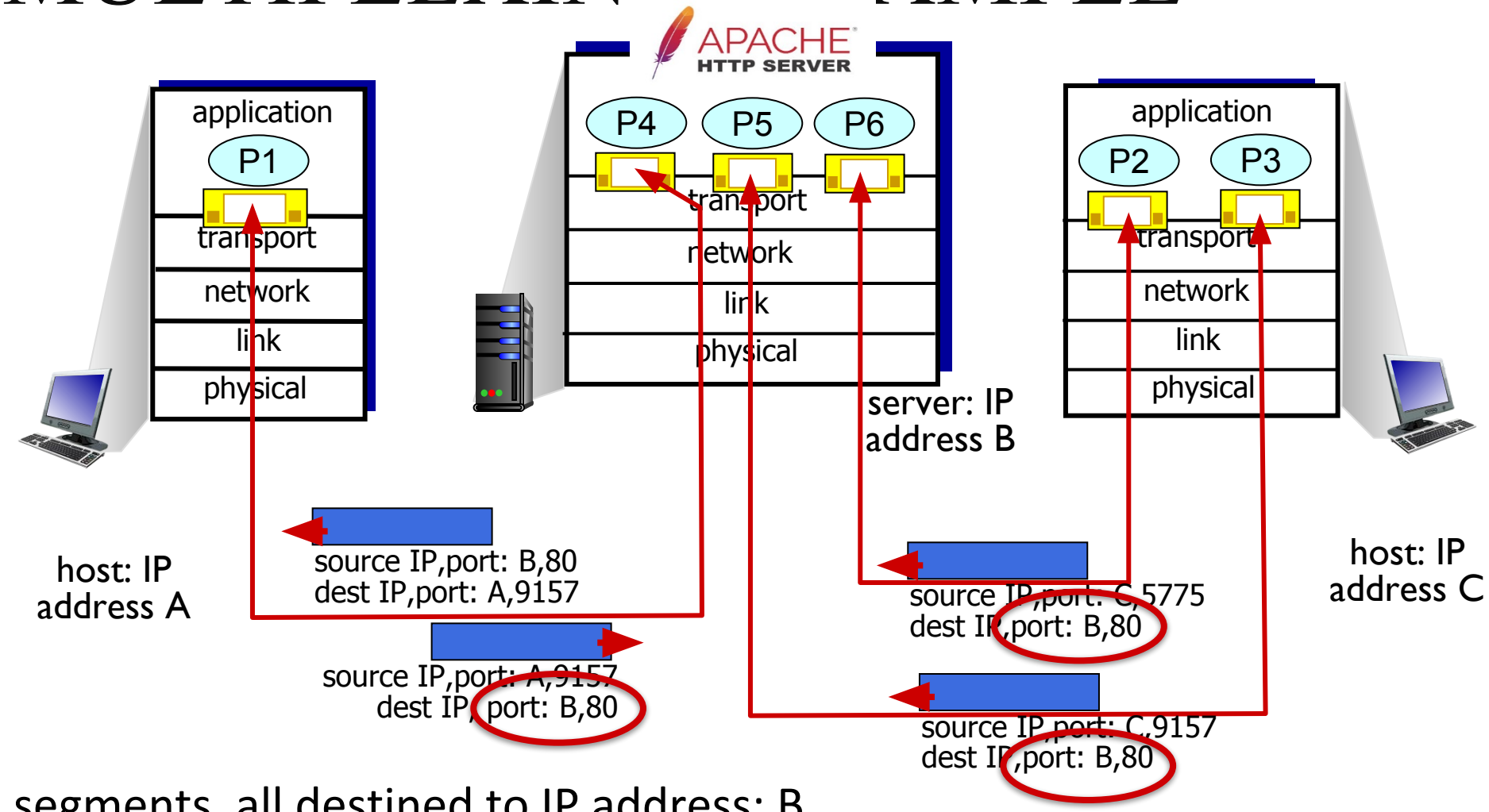
The TCP server application has a “welcoming socket”, then waits for connection-establishment requests from TCP clients on port number 12000.

The TCP client creates a socket and sends a connection establishment request segment with the lines:

```
clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName, 12000))
```

CONNECTION-ORIENTED DEMULTIPLEXING: EXAMPLE



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

- If the client server are using **Persistent HTTP**, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket.

- If the client server are using **Non-Persistent HTTP**, then a new TCP connection is created and closed for every request/response, hence a new socket is created and later closed for every request/response.

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

CONNECTIONLESS TRANSPORT: UDP

- "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, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

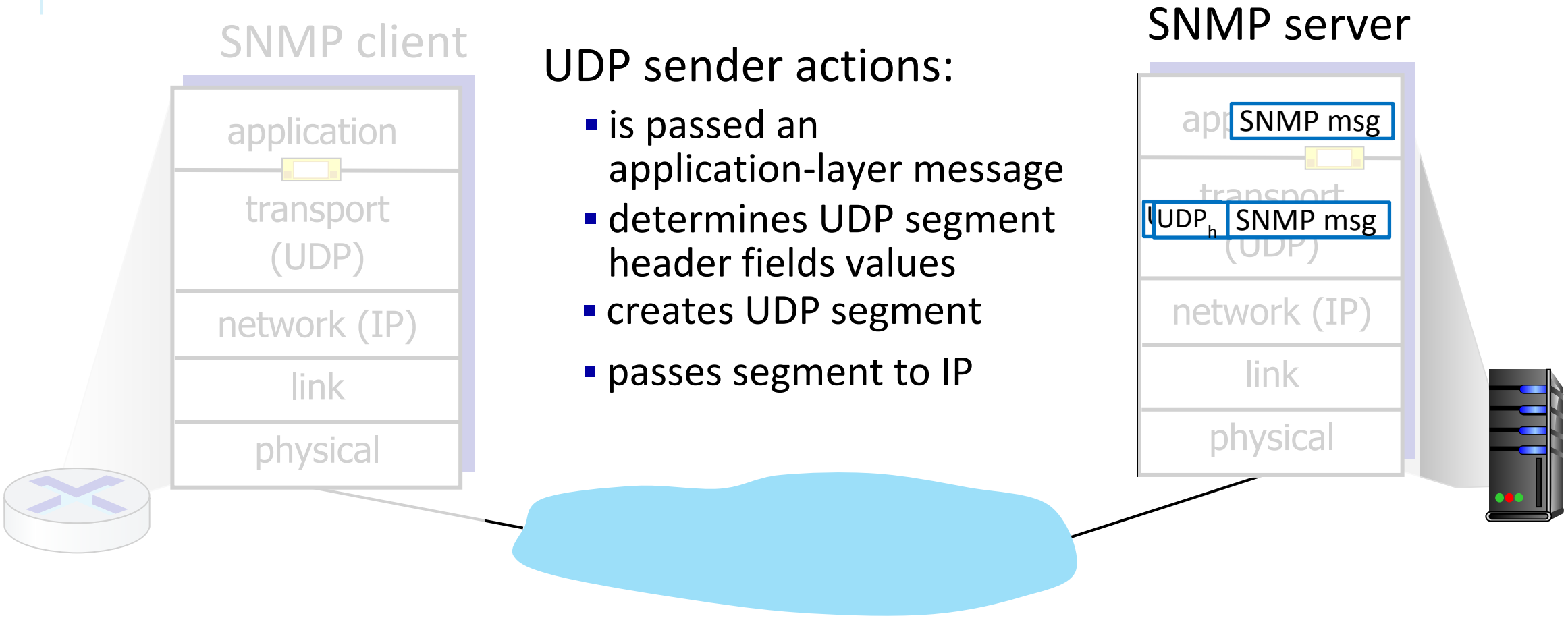
- no connection establishment (which can add RTT delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control
  - UDP can blast away as fast as desired!
  - can function in the face of congestion

UDP: USER DATAGRAM PROTOCOL

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP – network management data.
  - HTTP/3 -- UDP, providing their own error control and congestion control (among other services) at the application layer.
- if reliable transfer needed over UDP (e.g., HTTP/3):
  - add needed reliability at application layer
  - add congestion control at application layer

    (example : telephone video conferencing)

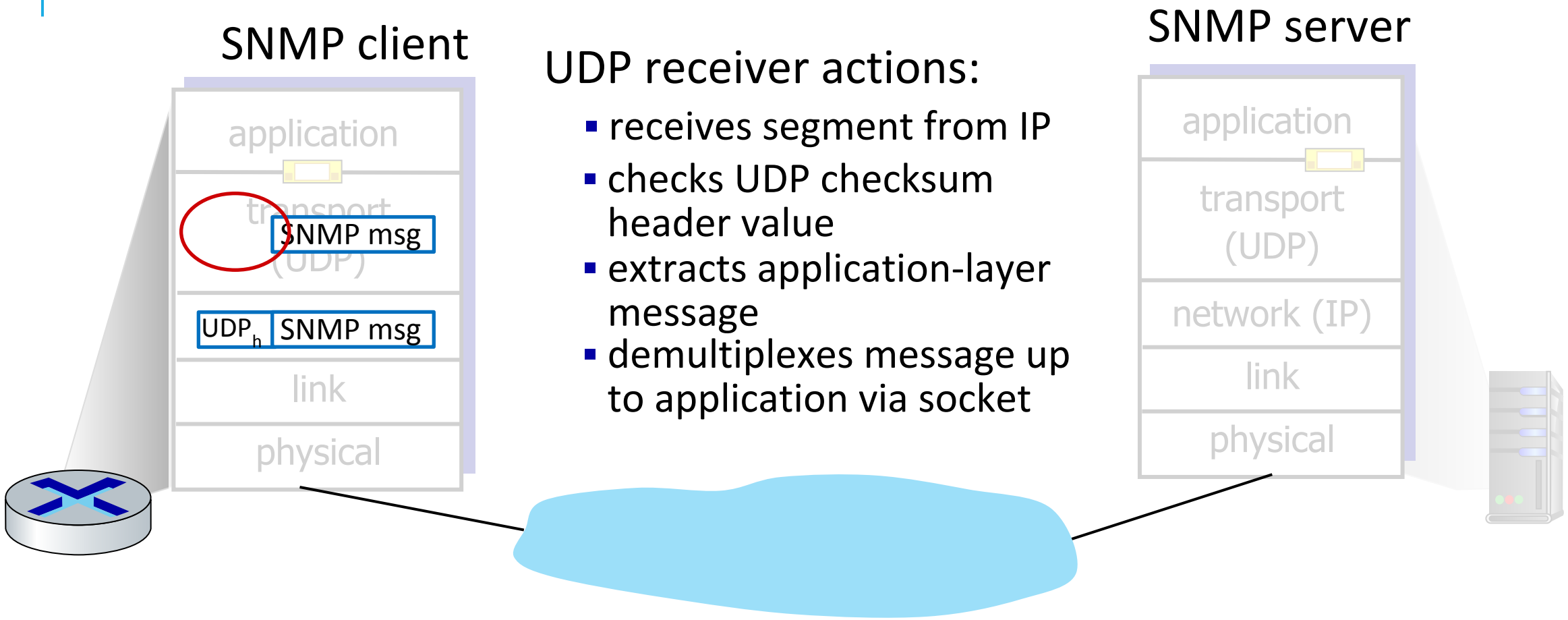
UDP: TRANSPORT LAYER ACTIONS



UDP sender actions:

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

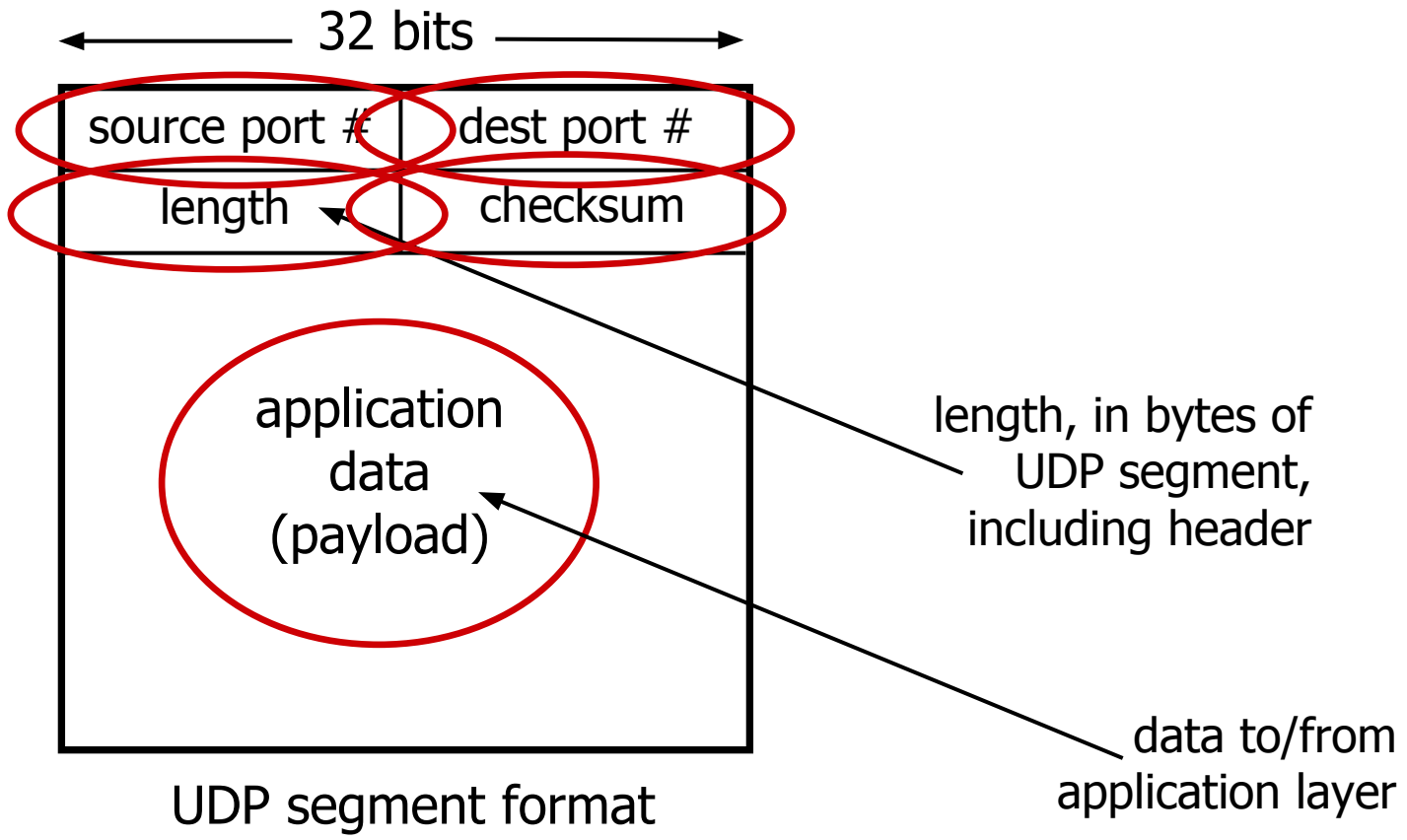
UDP: TRANSPORT LAYER ACTIONS



UDP receiver actions:

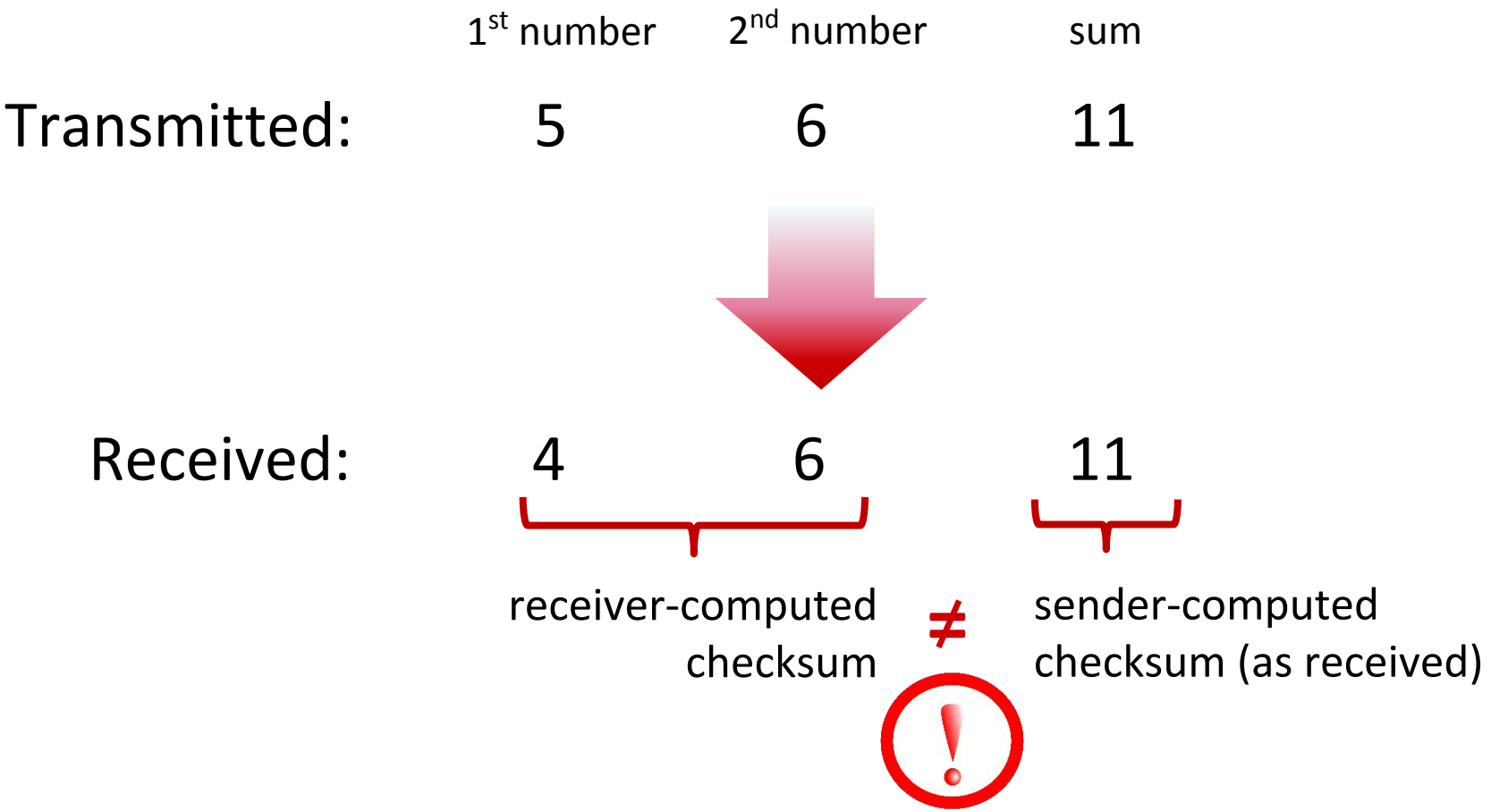
- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

UDP SEGMENT HEADER



UDP CHECKSUM

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



UDP CHECKSUM

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

sender:

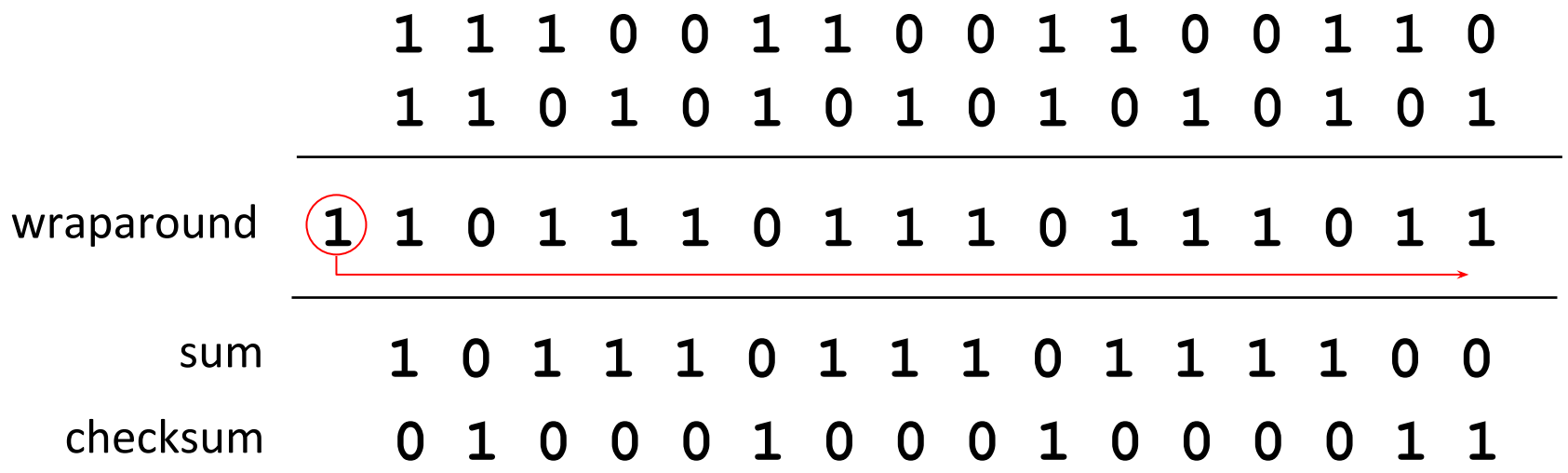
- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

INTERNET CHECKSUM: AN EXAMPLE

example: add two 16-bit integers

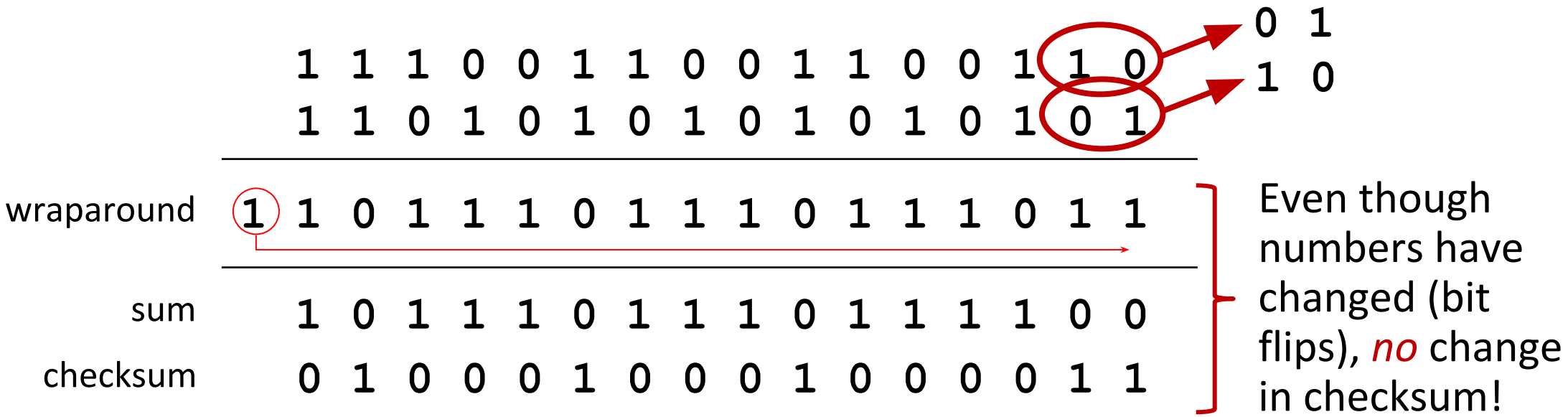


Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

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

INTERNET CHECKSUM: WEAK PROTECTION!

example: add two 16-bit integers



Even though numbers have changed (bit flips), *no* change in checksum!

SUMMARY: UDP

- "no frills" protocol:
  - segments may be lost, delivered out of order
  - best effort service: "send and hope for the best"
- UDP has its plusses:
  - no setup/handshaking needed (no RTT incurred)
  - can function when network service is compromised
  - helps with reliability (checksum)
- build additional functionality on top of UDP in application layer (e.g., HTTP/3)

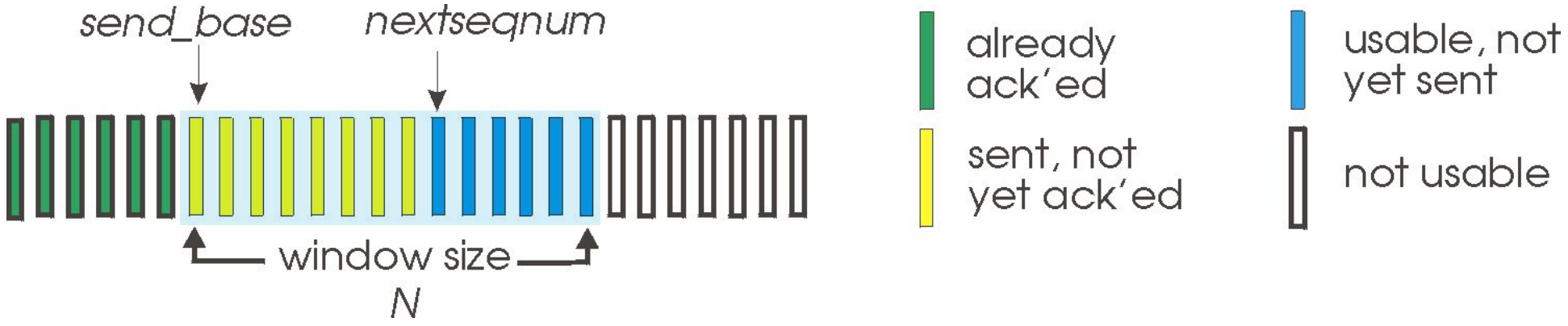
GO-BACK-N (GBN)

An awesome interactive animation :

https://computerscience.unicam.it/marcantoni/reti/applet/GoBackProtocol/goback.html

GO-BACK-N: SENDER

- sender: "window" of up to N, consecutive transmitted but unACKed pkts
  - k-bit seq # in pkt header

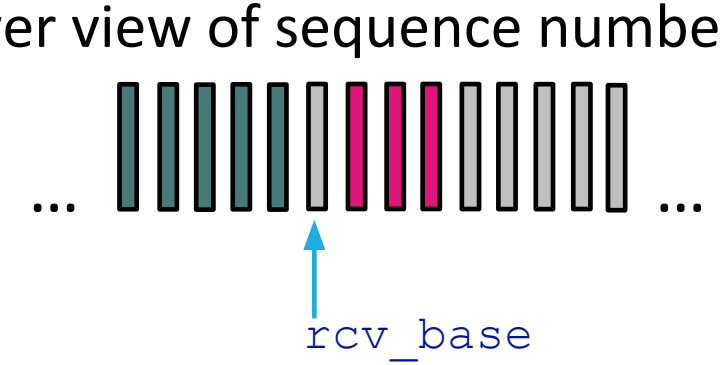


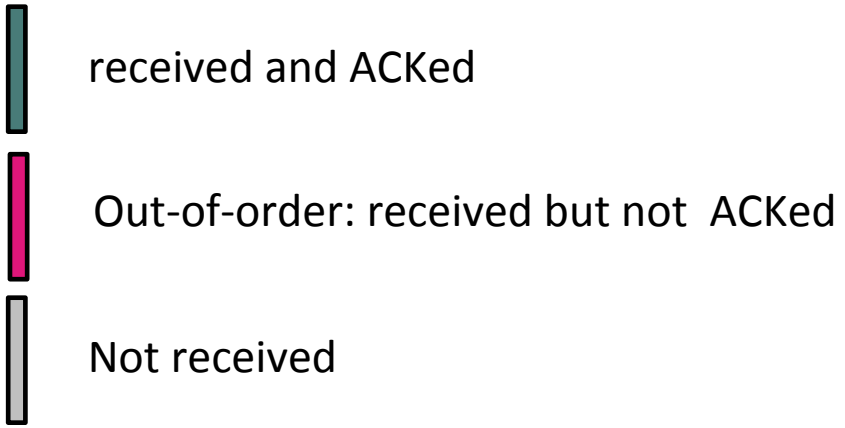
- *cumulative ACK:* ACK(*n*): ACKs all packets up to, including seq # *n*
  - on receiving ACK(*n*): move window forward to begin at *n+1*
- timer for oldest in-flight packet
- *timeout(n):* retransmit packet n and all higher seq # packets in window

GO-BACK-N: RECEIVER

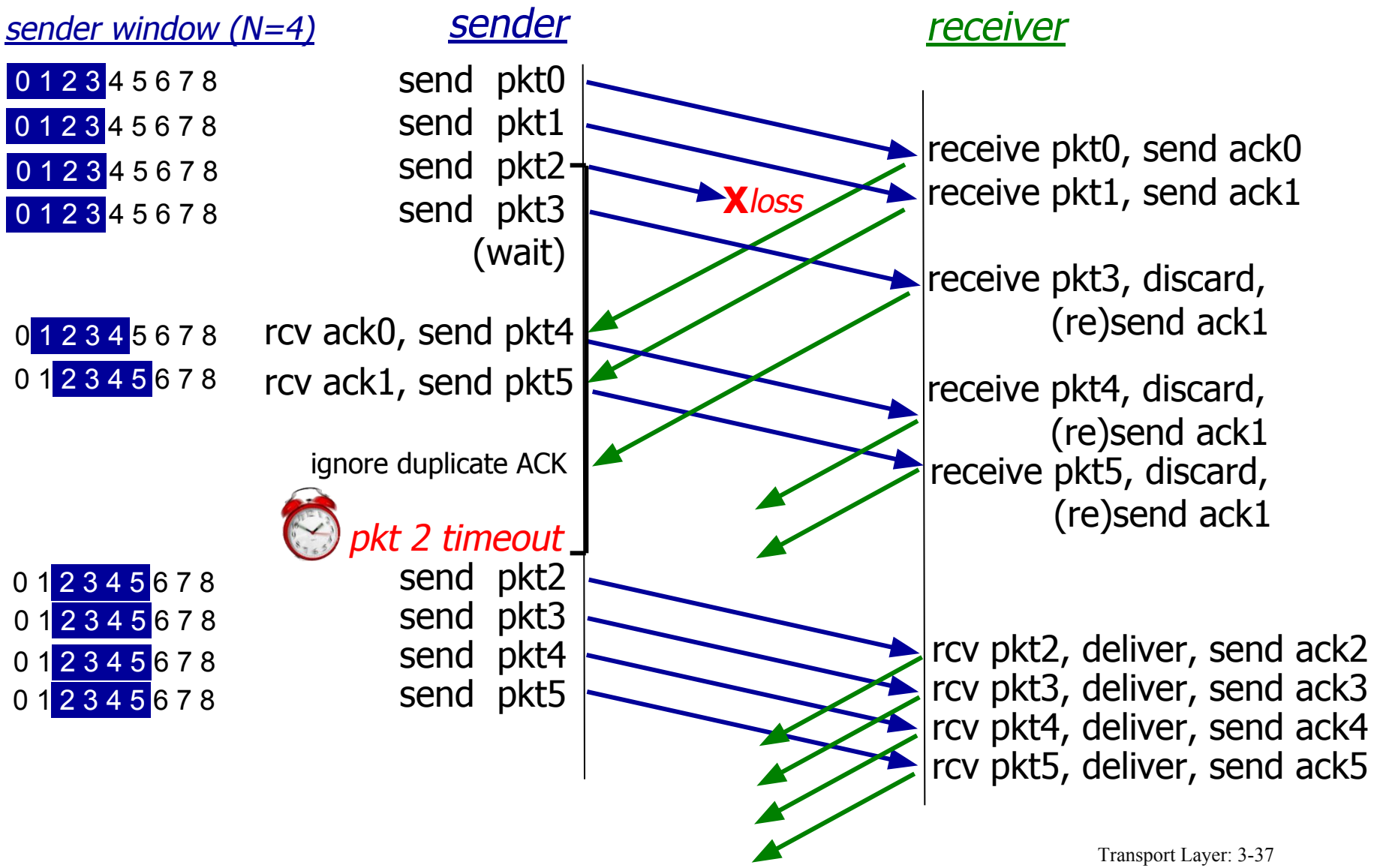
- ACK-only: always send ACK for correctly-received packet so far, with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember `rcv_base`
- on receipt of out-of-order packet:
  - can discard (don't buffer) or buffer: an implementation decision
  - re-ACK pkt with highest in-order seq #

Receiver view of sequence number space:

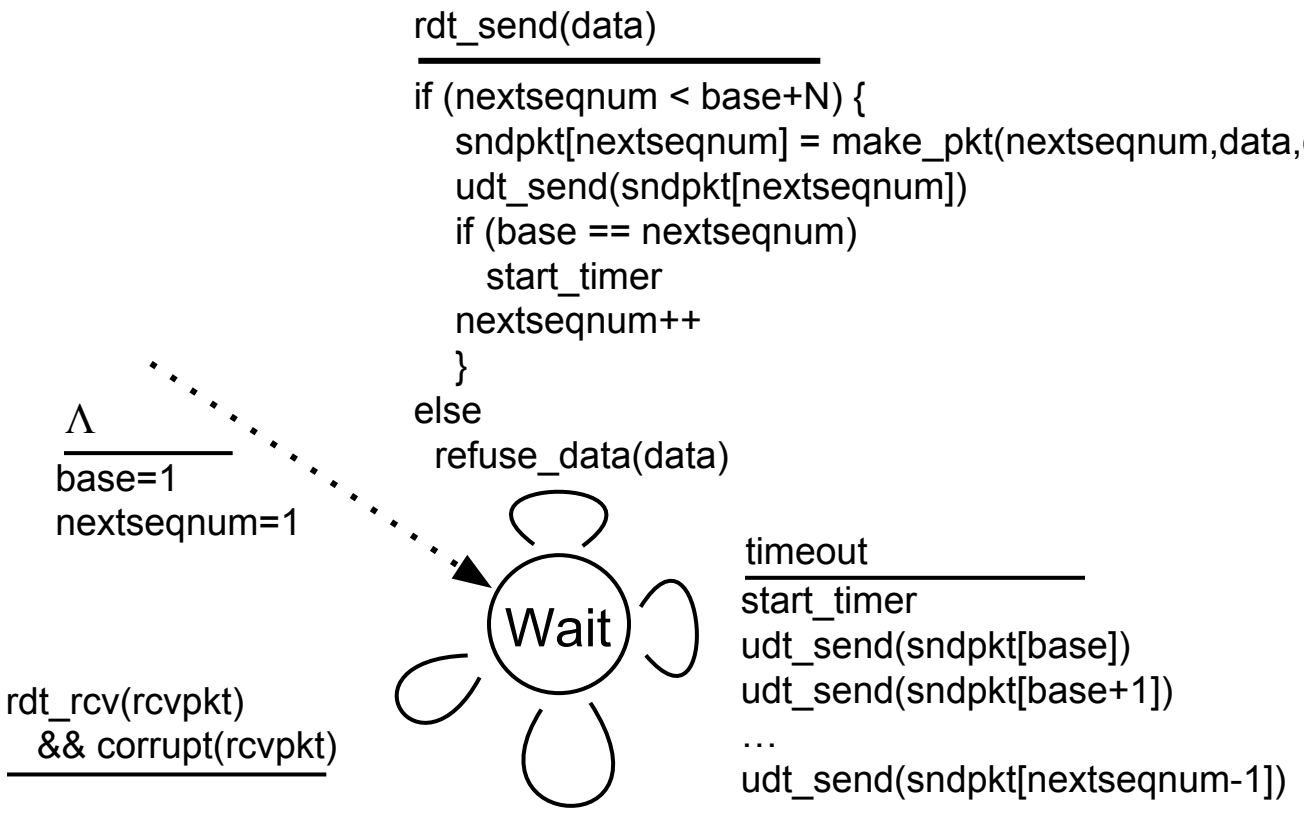




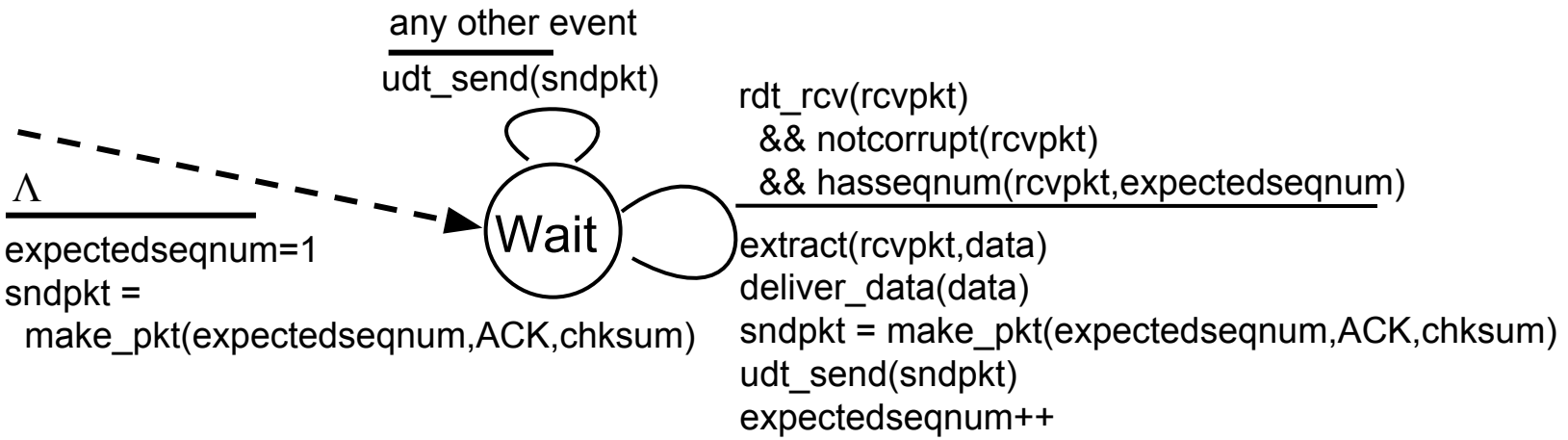
GO-BACK-N IN ACTION



GO-BACK-N: SENDER EXTENDED FSM



GO-BACK-N: RECEIVER EXTENDED FSM



ACK-only: always send ACK for correctly-received packet with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

- out-of-order packet:
  - discard (don't buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #

SELECTIVE REPEAT

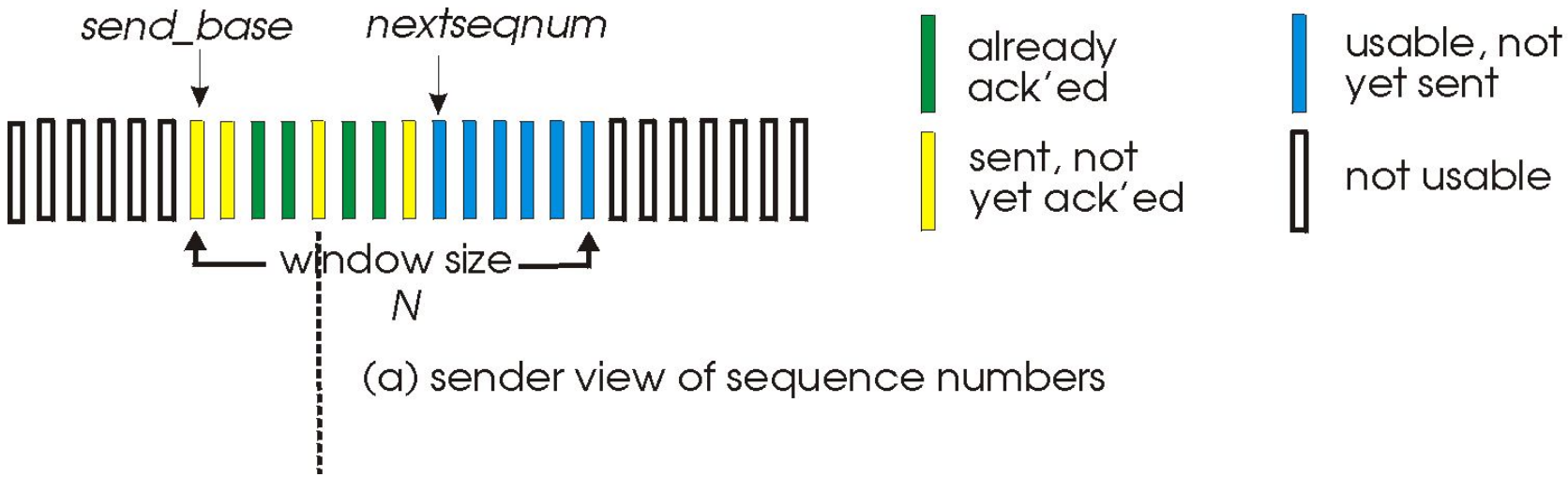
- The GBN protocol allows the sender to potentially “fill the pipeline” .
- GBN itself suffers from performance problems.
- When the window size and bandwidth-delay product are both large, many packets can be in the pipeline.
- A single packet error can thus cause GBN to retransmit a large number of packets, many unnecessarily.

An awesome interactive animation :

https://computerscience.unicam.it/marcantoni/reti/applet/SelectiveRepeatProtocol/selRepProt.html

- Selective repeat protocols avoid unnecessary retransmissions.
- Receiver individually acknowledge correctly received packets.
- A window size of N will again be used to limit the number of outstanding, unacknowledged packets in the pipeline.

SELECTIVE REPEAT: SENDER, RECEIVER WINDOWS



SELECTIVE REPEAT

- Receiver *individually* acknowledges all correctly received packets
  - buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender times-out/retransmits individually for unACKed packets
  - sender maintains timer for each unACKed pkt
- Sender window
  - *N* consecutive seq #s
  - limits seq #s of sent, unACKed packets

SELECTIVE REPEAT: SENDER AND RECEIVER

Sender

Data received from above:

- if next available seq # in window, send packet

Timeout(*n*):

- resend packet *n*, restart timer

ACK(*n*) in [sendbase, sendbase+N]:

- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

receiver

Packet *n* in [rcvbase, rcvbase+N-1]

- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

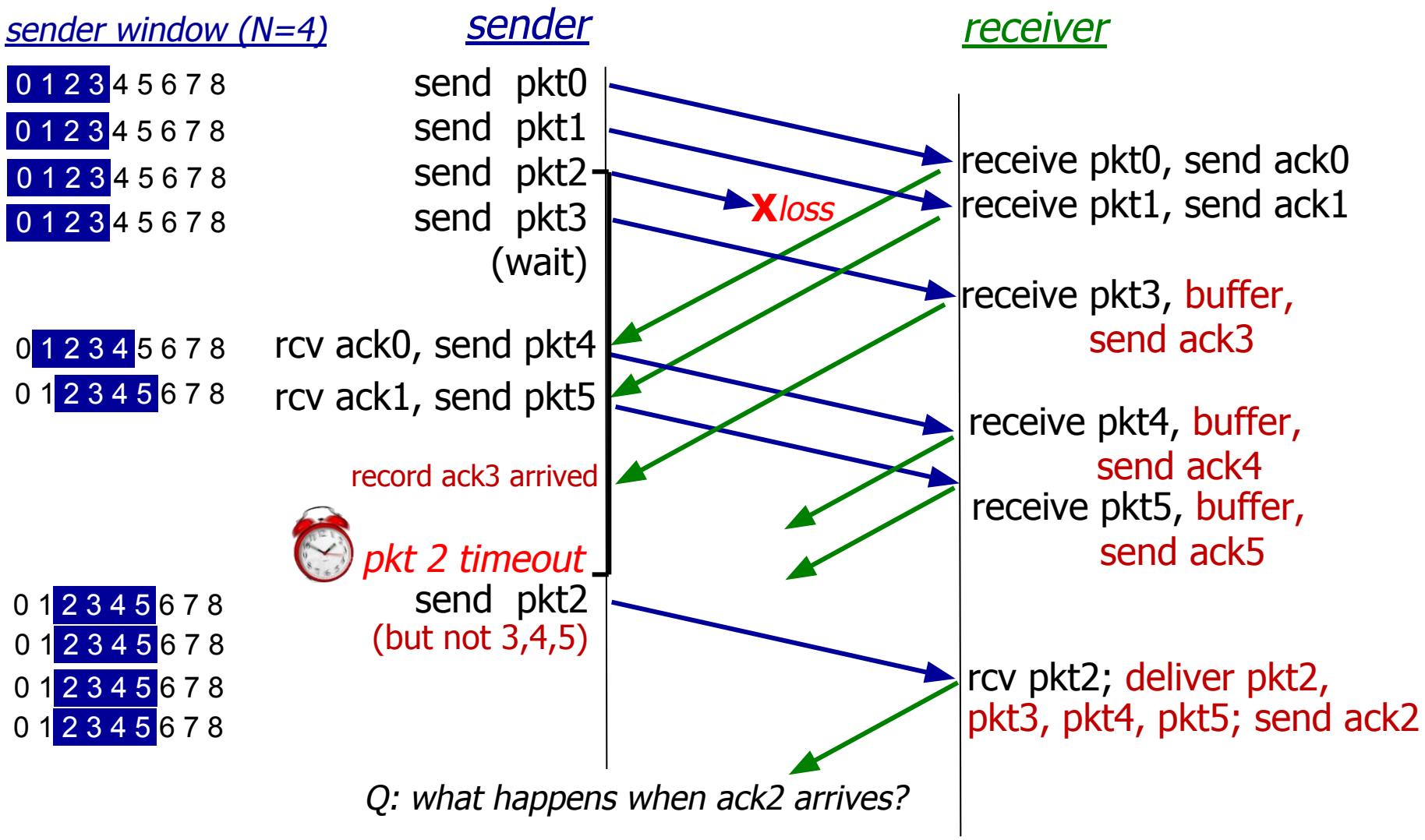
packet *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

otherwise:

- ignore

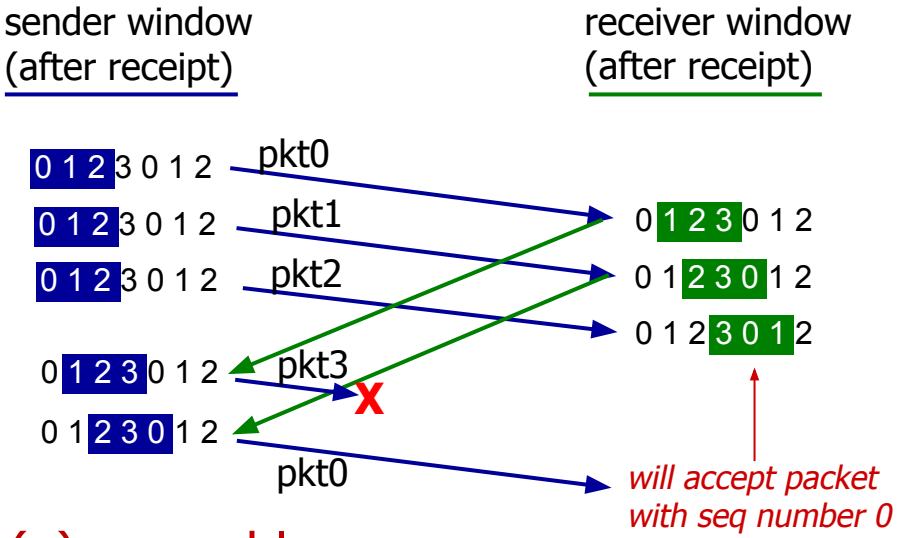
SELECTIVE REPEAT IN ACTION



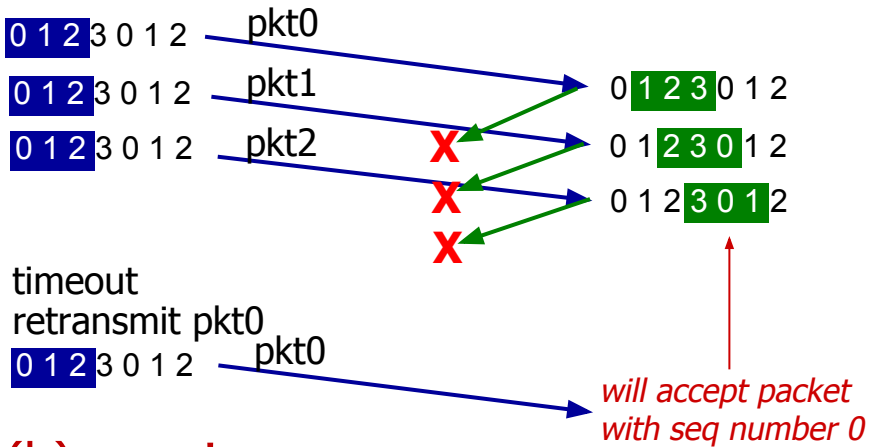
SELECTIVE REPEAT: A DILEMMA!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3



(a) no problem



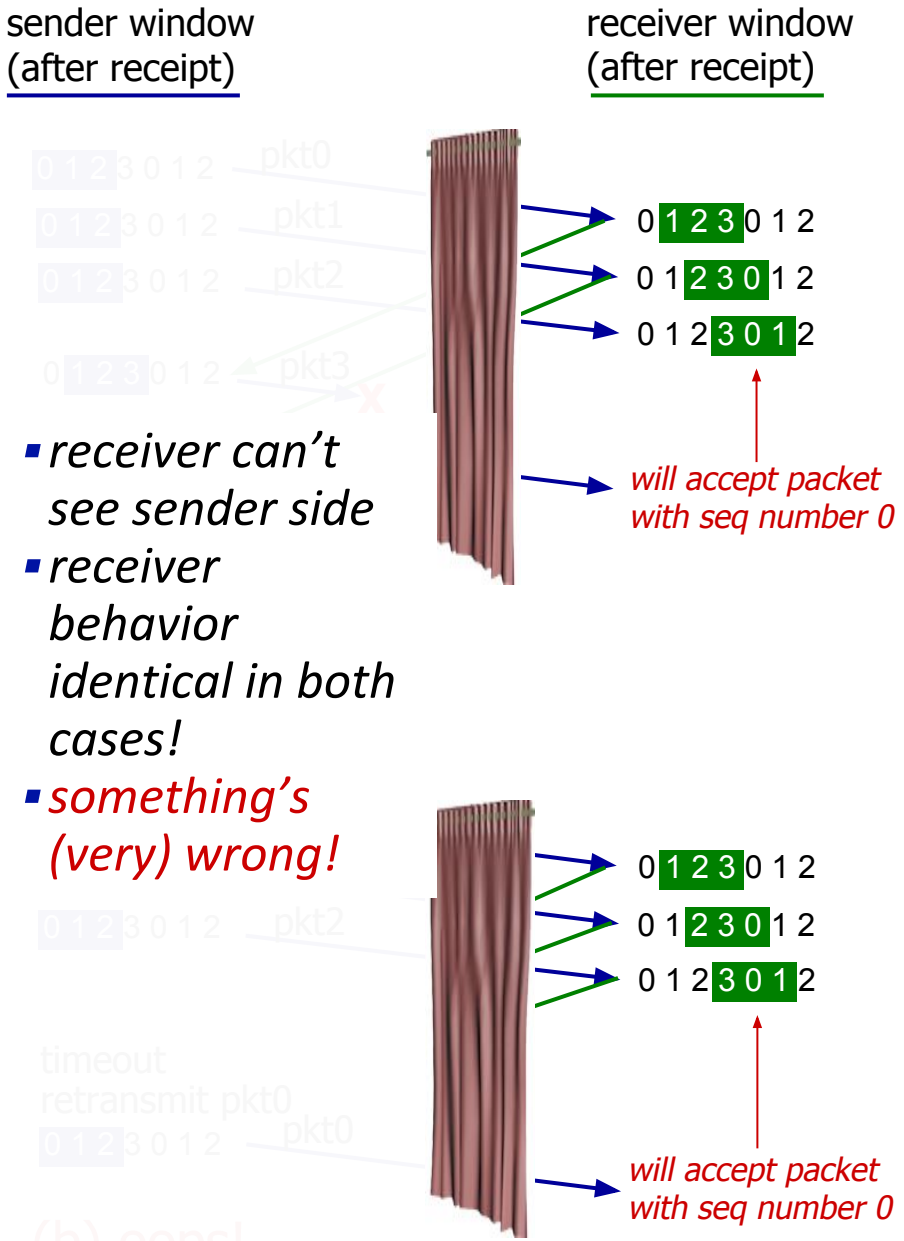
(b) oops!

SELECTIVE REPEAT: A DILEMMA!

example:

- seq #s: 0, 1, 2, 3 (base 4 counting)
- window size=3

Q: what relationship is needed between sequence # size and window size to avoid problem in scenario (b)?



SUMMARIZE

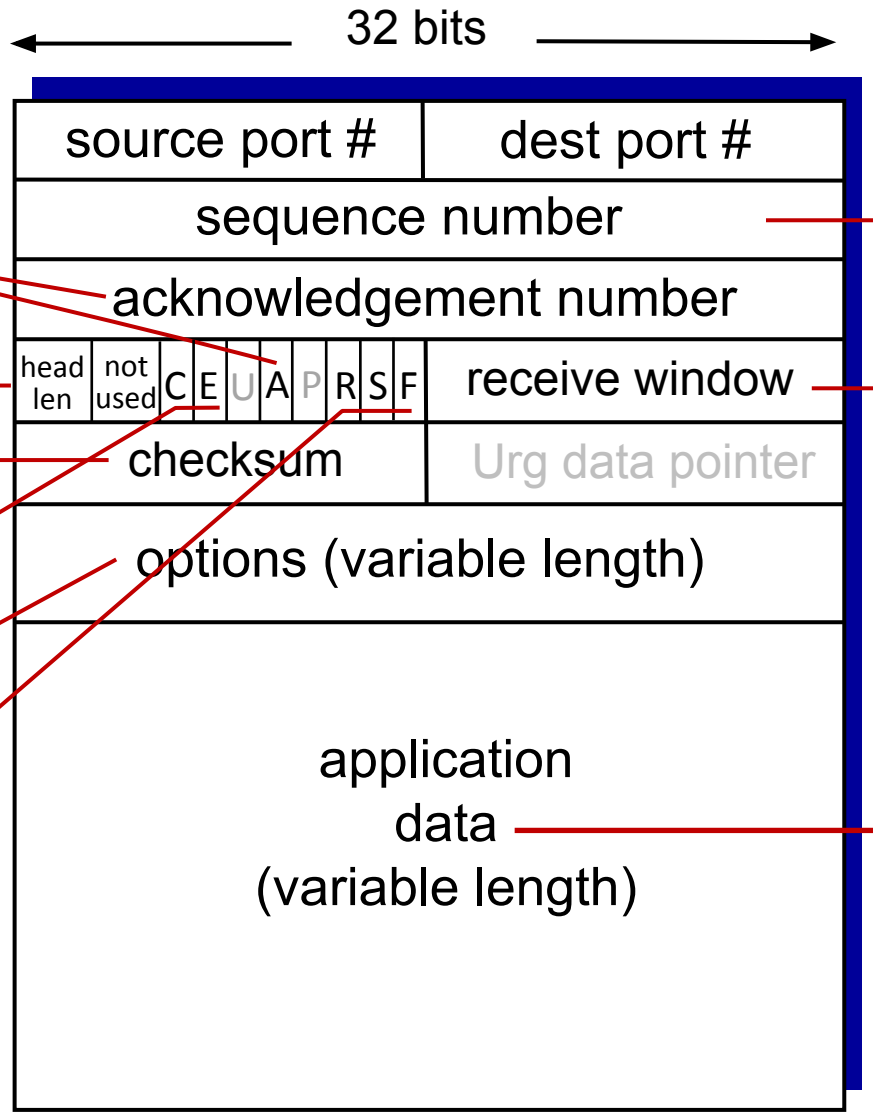
- Checksum
- Timer
- Sequence Number
- Acknowledgement
- Negative Acknowledgement
- Window, Pipelining

CONNECTION ORIENTED TRANSPORT : TCP

The TCP Connection:

- Connection-oriented:
  - handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- Full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- Point-to-Point:
  - one sender, one receiver
- cumulative ACKs
-
- reliable, in-order *byte steam:*
  - no “message boundaries"
- pipelining:
  - TCP congestion and flow control set window size
- flow controlled:
  - sender will not overwhelm receiver

TCP SEGMENT STRUCTURE



ACK: seq # of next expected byte; A bit: this is an ACK

length (of TCP header)

Internet checksum

C, E: congestion notification

TCP options

RST, SYN, FIN: connection management

segment seq #: counting bytes of data into bytestream (not segments!)

flow control: # bytes receiver willing to accept

data sent by application into TCP socket

Sequence Number and Acknowledgement Numbers:

TCP views data as an unstructured, but ordered, stream of bytes.

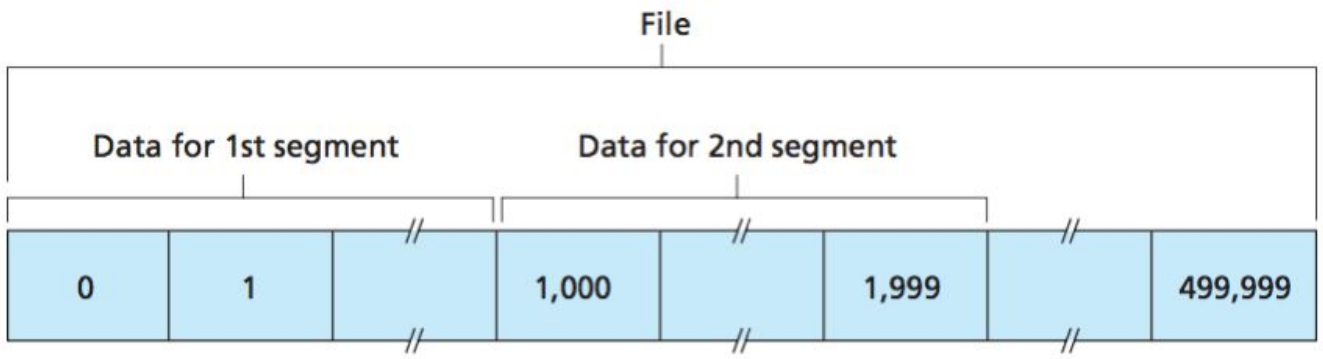


Figure 3.30 ♦ Dividing file data into TCP segments

TCP SEQUENCE NUMBERS, ACKS

Sequence numbers:

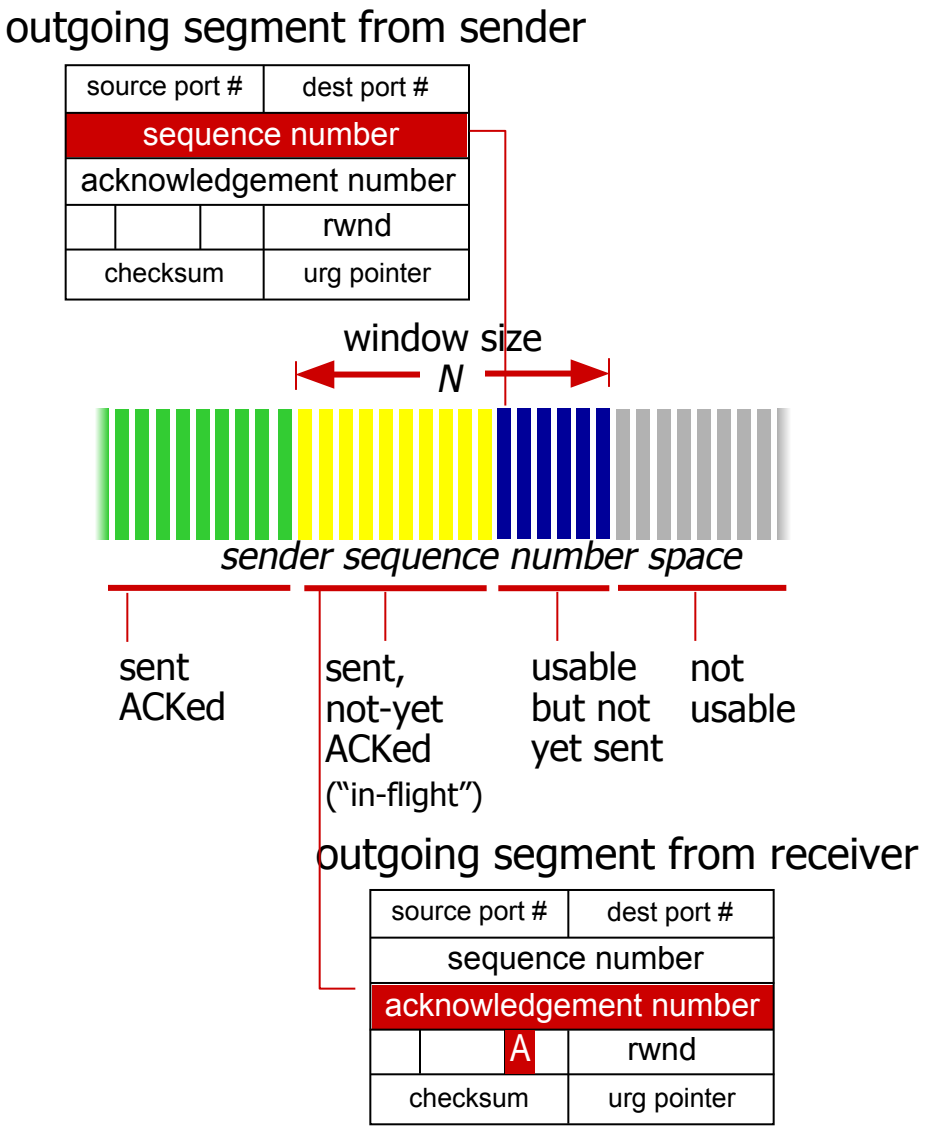
- byte stream "number" of first byte in segment's data

Acknowledgements:

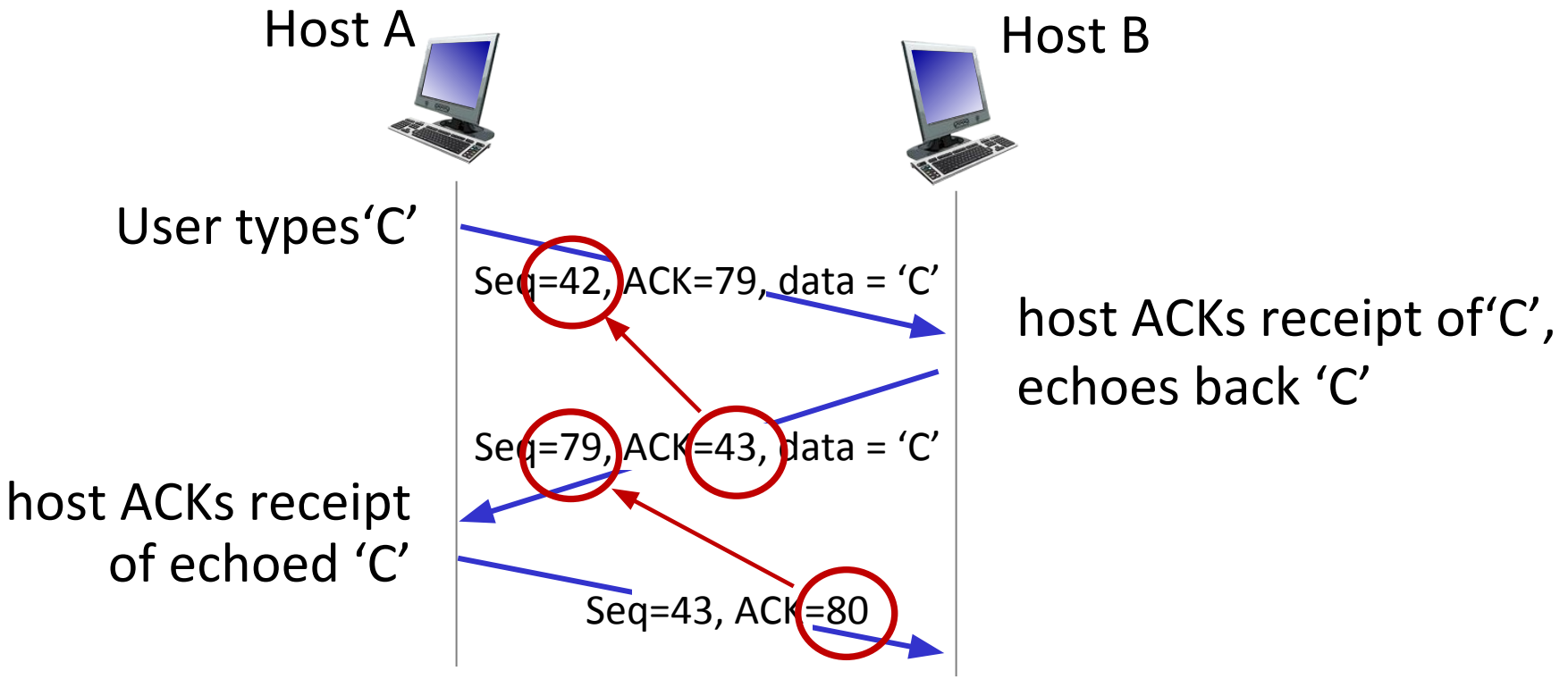
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- *A:* TCP spec doesn't say, - up to implementor



TCP SEQUENCE NUMBERS, ACKS



simple telnet scenario

ROUND-TRIP TIME ESTIMATION AND TIMEOUT

- TCP, uses a timeout/retransmit mechanism to recover from lost segments.

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 values will fluctuate from segment to segment due to congestion in the routers and to the varying load on the end systems.
- In order to estimate a typical RTT, it is therefore natural to take some sort of average of the SampleRTT values. TCP maintains an average, called EstimatedRTT, of the SampleRTT values.

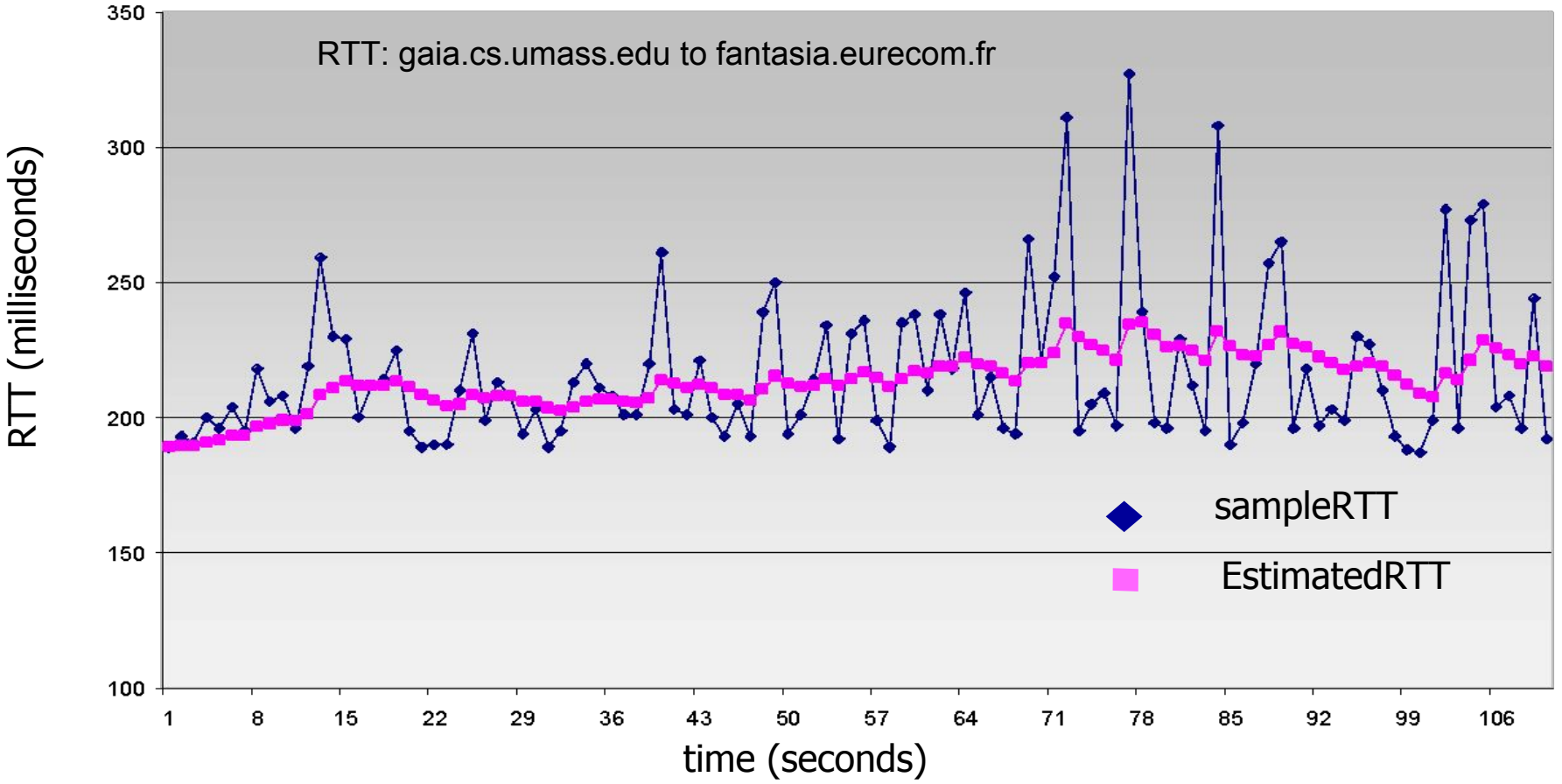


Figure: RTT samples and RTT estimates

- Upon obtaining a new SampleRTT, TCP updates EstimatedRTT according to the following formula:

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

- EstimatedRTT is a weighted combination of the previous value of EstimatedRTT and the new value for SampleRTT. The recommended value of α is $\alpha = 0.125$

$$\text{EstimatedRTT} = 0.875 \cdot \text{EstimatedRTT} + 0.125 \cdot \text{SampleRTT}$$

- In addition to having an estimate of the RTT, it is also valuable to have a measure of the variability of the RTT. DevRTT, as an estimate of how much SampleRTT typically deviates from EstimatedRTT:

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

- Note that DevRTT is an EWMA of the difference between SampleRTT and EstimatedRTT. If the SampleRTT values have little fluctuation, then DevRTT will be small; on the other hand, if there is a lot of fluctuation, DevRTT will be large. The recommended value of β is 0.25.

Setting and Managing the Retransmission Timeout Interval :

- Timeout interval should not be too much larger than EstimatedRTT; otherwise, when a segment is lost, TCP would not quickly retransmit the segment, leading to large data transfer delays. It is therefore desirable to set the timeout equal to the EstimatedRTT plus some margin.
- All of these considerations are taken into account in TCP's method for determining the retransmission timeout interval:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

RELIABLE DATA TRANSFER

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`**

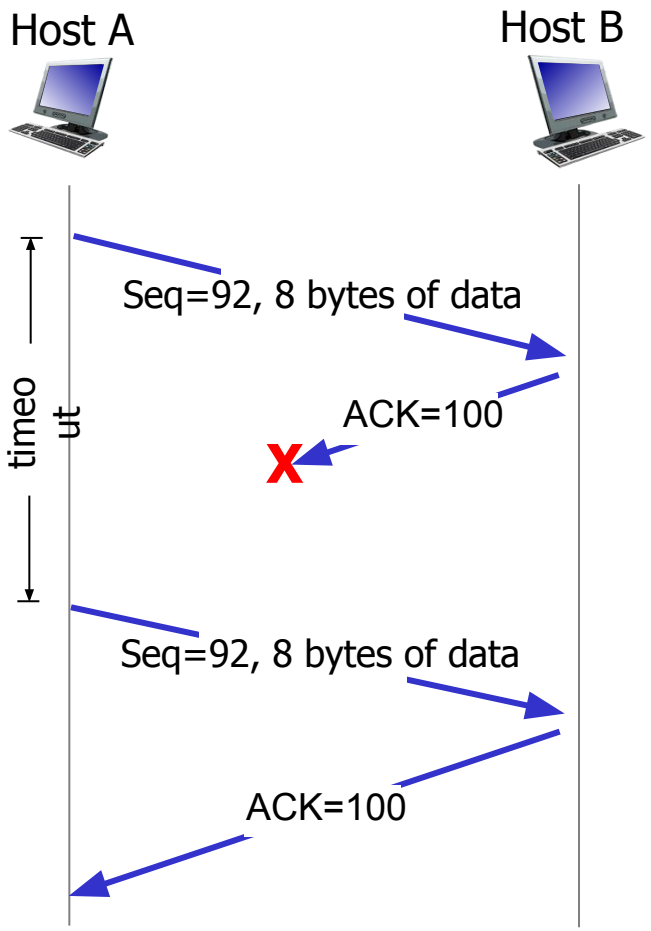
event: timeout

- retransmit segment that caused timeout
- restart timer

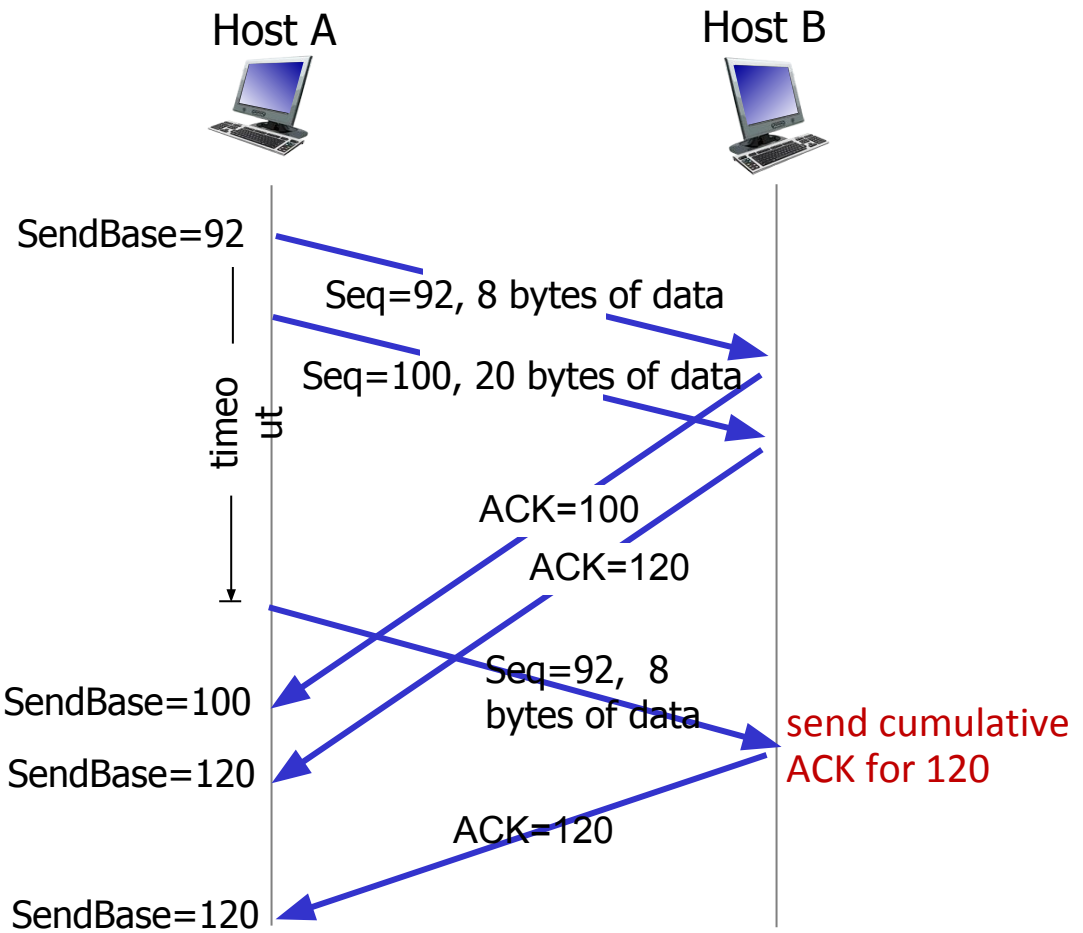
event: ACK received

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

3 SCENARIOS

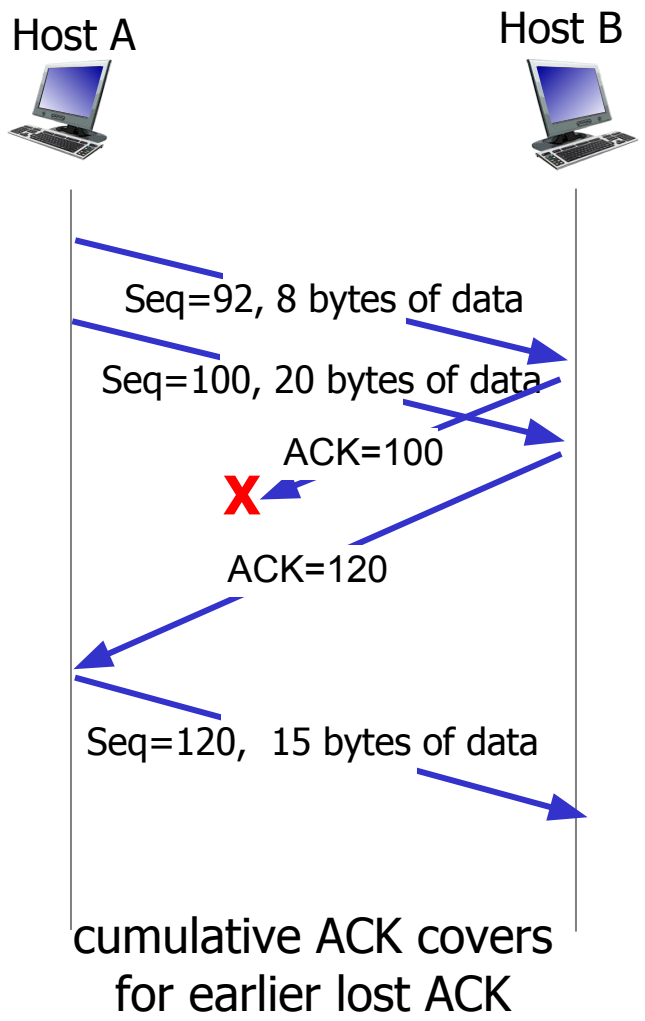


lost ACK scenario



premature timeout

TCP: RETRANSMISSION SCENARIOS



Doubling the Timeout Interval :

- A few modifications that most TCP implementations employ :
- In this modification, whenever the timeout event occurs, TCP retransmits the not-yet-acknowledged segment with the smallest sequence number. But each time TCP retransmits, it sets the next timeout interval to twice the previous value.

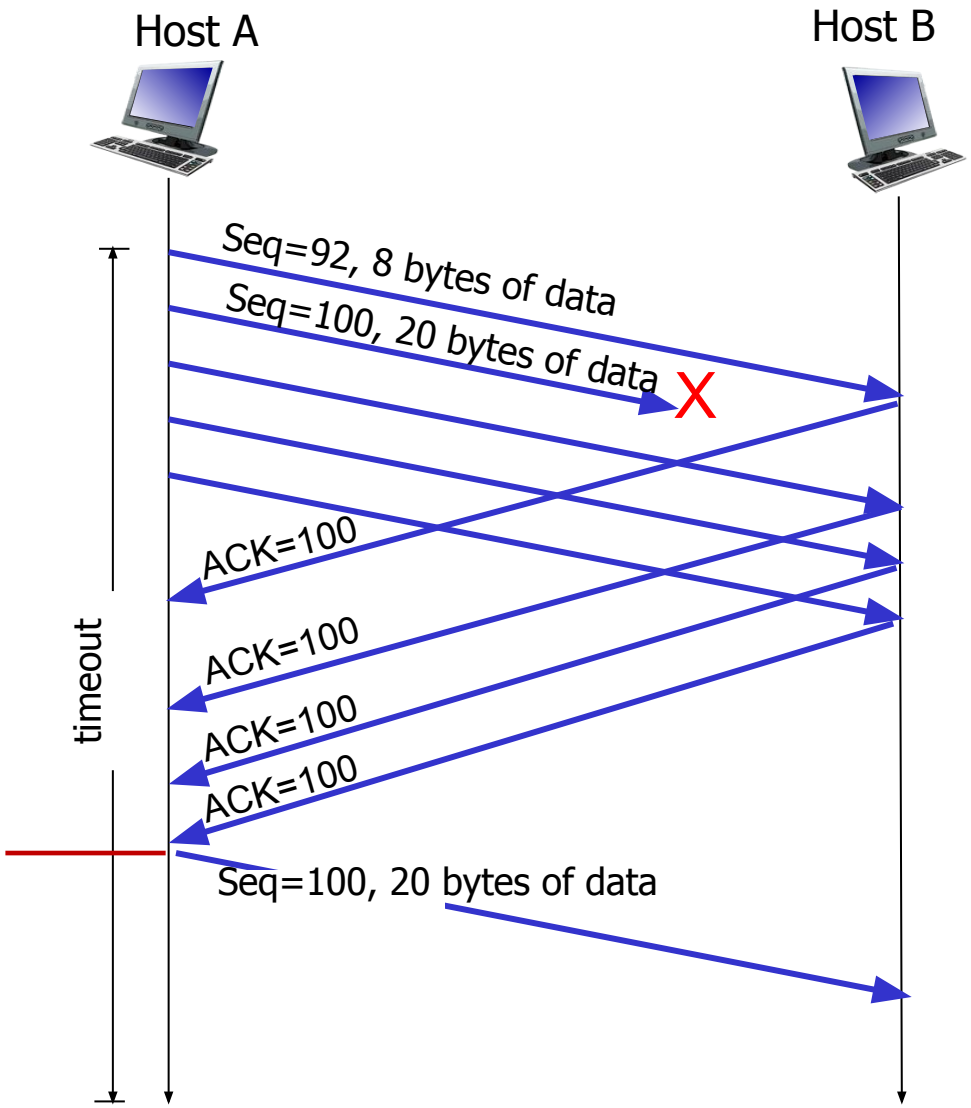
TCP FAST RETRANSMIT

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!



FLOW CONTROL

- TCP provides a flow-control service to its applications to eliminate the possibility of the sender overflowing the receiver's buffer.
- Flow control is thus a speed matching service—matching the rate at which the sender is sending against the rate at which the receiving application is reading.
- TCP provides flow control by having the sender maintain a variable called the **receive window**. Informally, the receive window is used to give the sender an idea of how much free buffer space is available at the receiver.
  - denote its size by RcvBuffer.

- The receiver reads from the buffer. Define the following variables:

  1. LastByteRead : the number of last byte in the data stream read from the buffer by the application process in B.

  2. LastByteRevd : the number of the last byte in the data stream that has arrived from the network and has been placed in the receive buffer at B.

- TCP doesn't permit to overflow the allocated buffer :

  LastByteRevd – LastByteRead <= RevBuffer

- The receive window denoted rwnd is set to the amount of spare room in the buffer

  Rwnd = RcvBuffer – [LastByteRcvd – LastByteread]

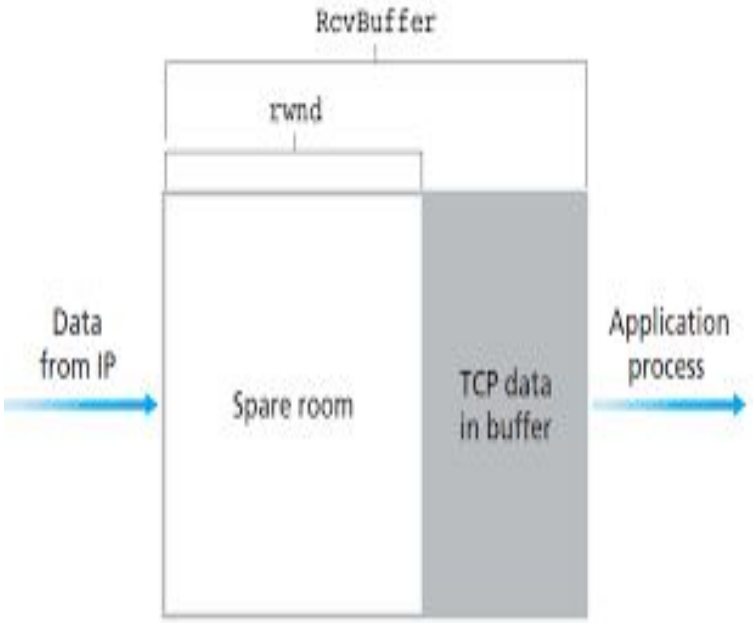


Figure : The receive window (rwnd) and the receive buffer (RcvBuffer)

- Host B tells how much spare room it has in the connection buffer
- Host A keeps track of Two variables
  1. LastByteSent
  2. LastByteAcked
- Host A makes sure that

  LastbyteSent – LastByteAcked <= rwnd

TCP CONNECTION MANAGEMENT

Let’s first take a look at how a TCP connection is established.

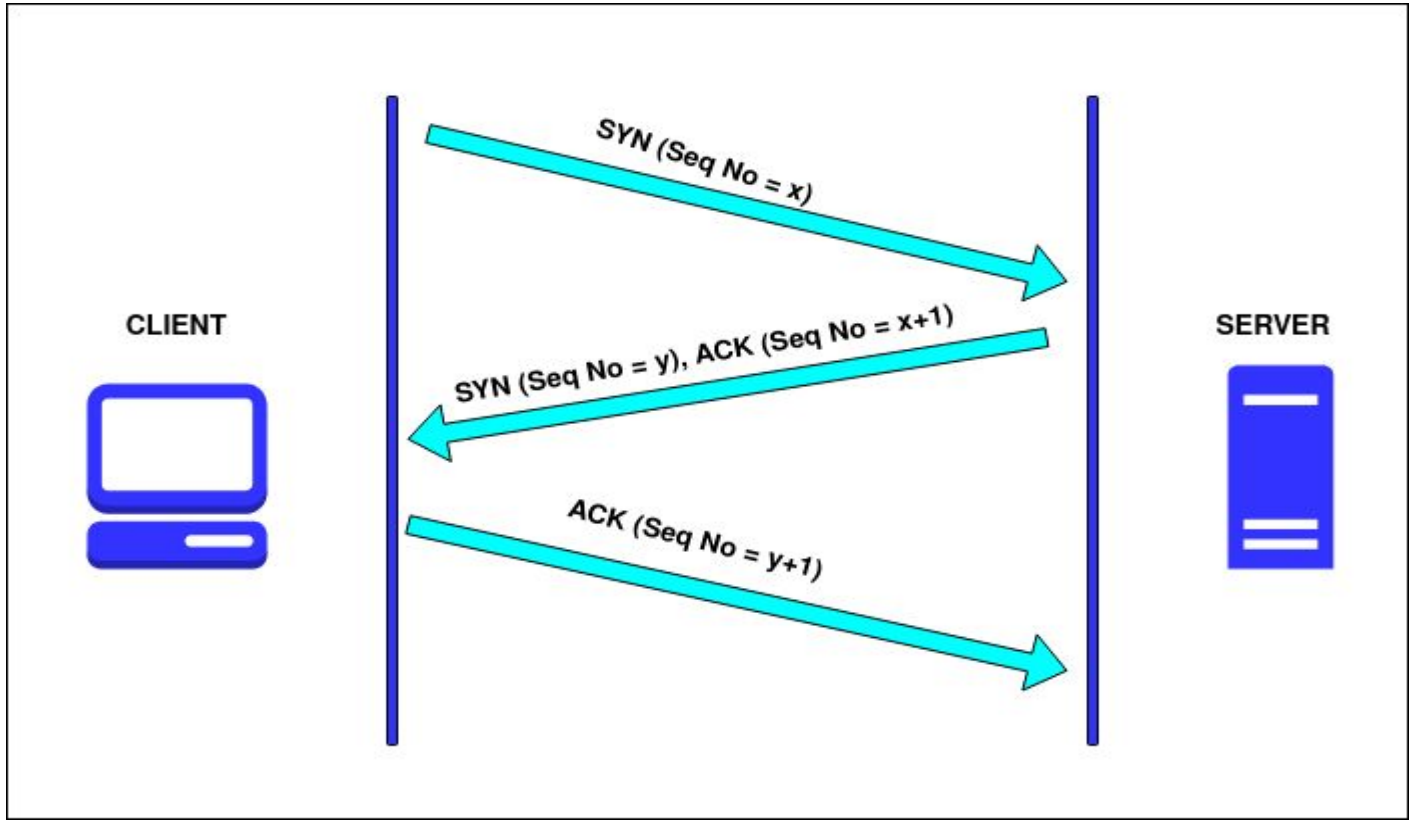


Figure : TCP three-way handshake: segment exchange

Closing the TCP connection :

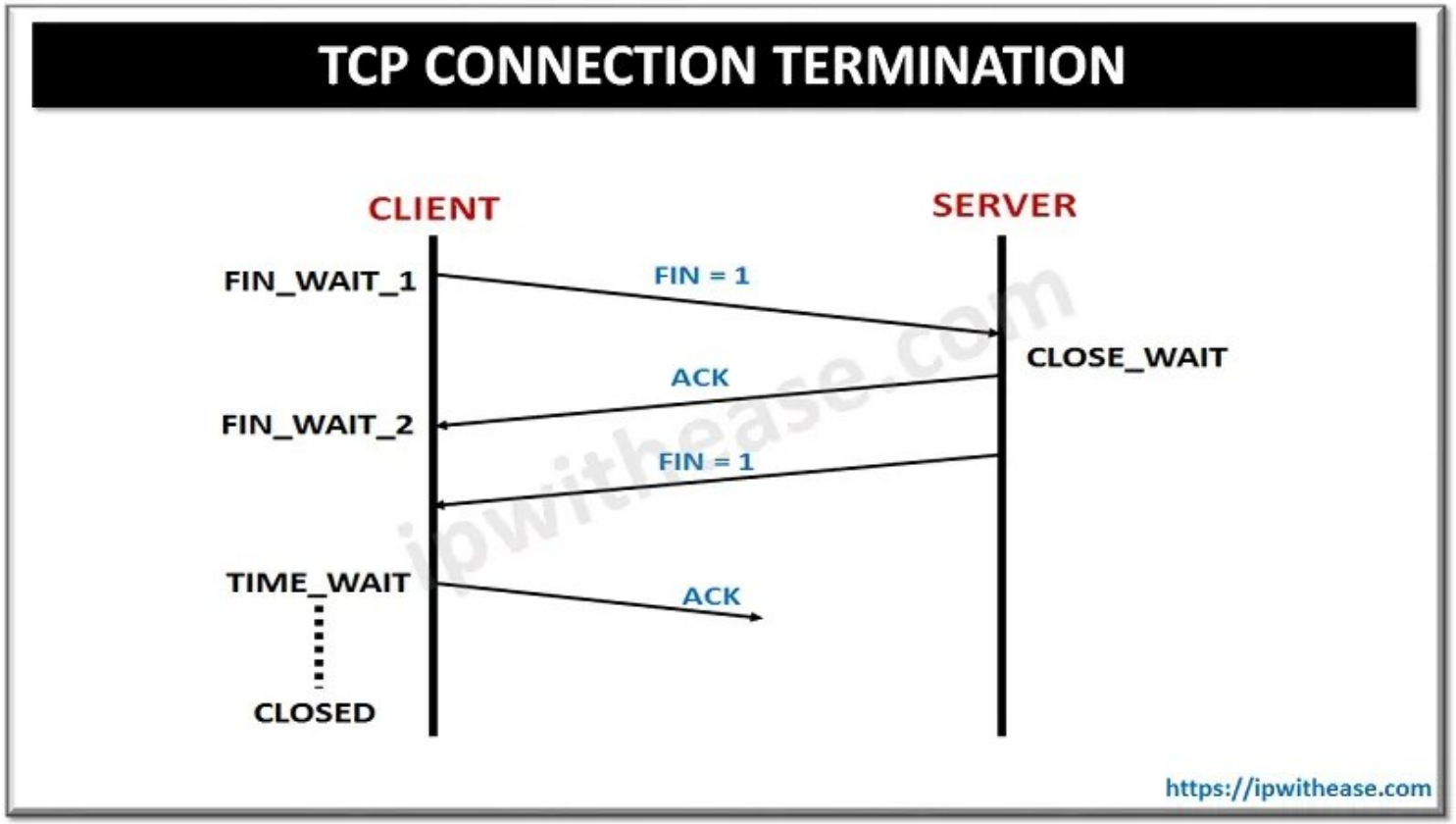
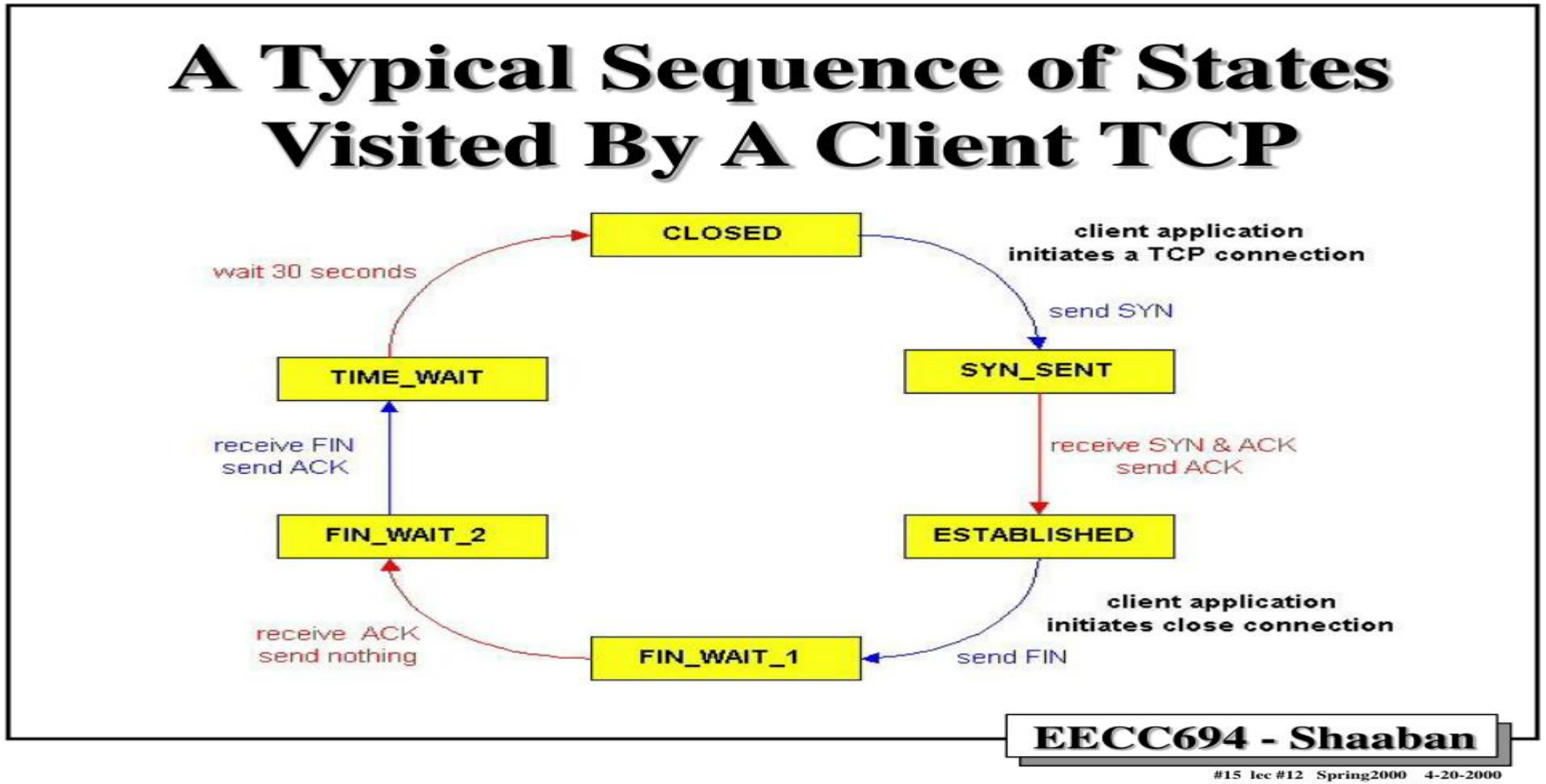


Figure : Closing a TCP connection

During the life of a TCP connection, the TCP protocol running in each host makes transitions through various TCP states.



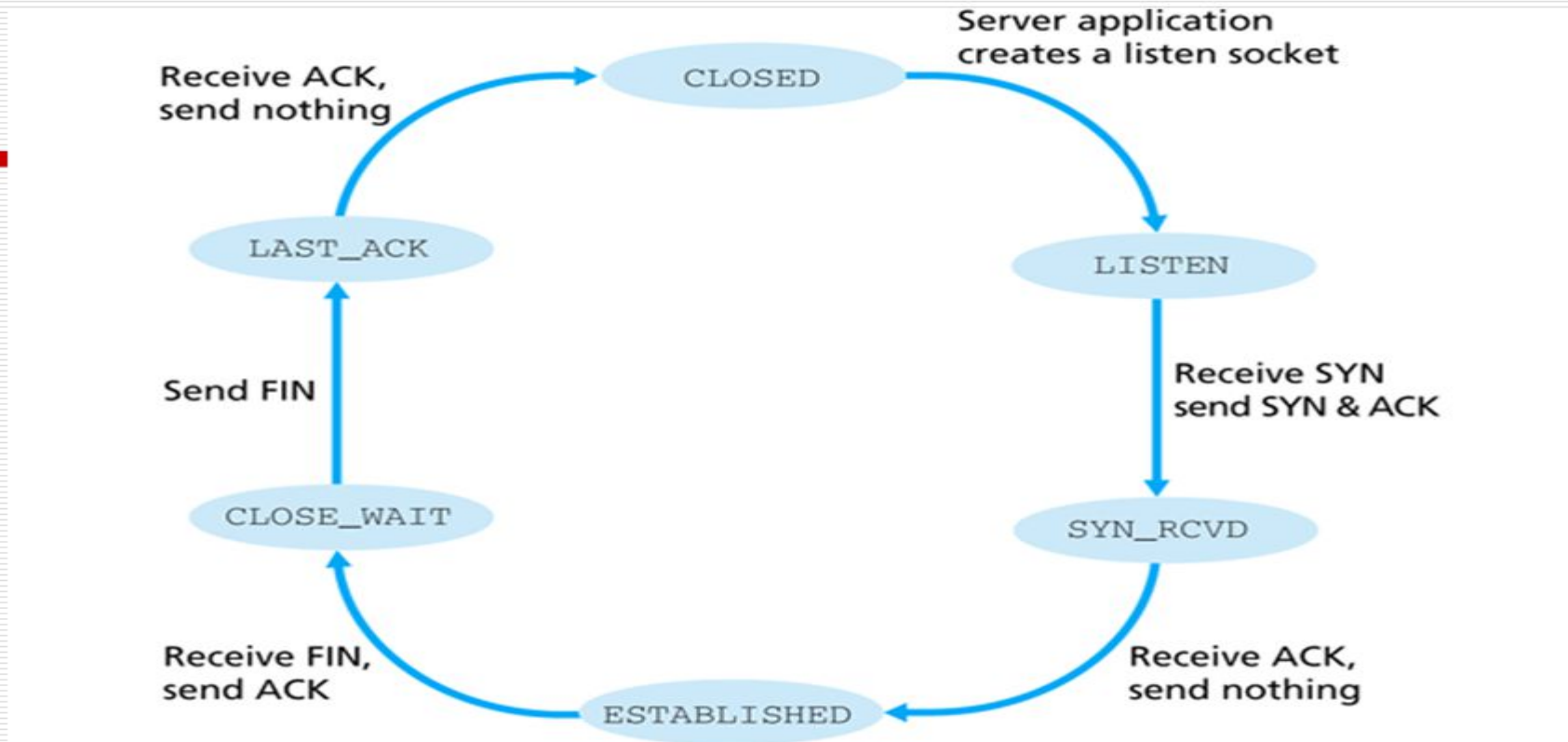


Figure 3.42 ♦ A typical sequence of TCP states visited by a server-side TCP

TCP CONGESTION CONTROL

- TCP provides a reliable transport service between two processes running on different hosts.
- Another key component of TCP is its congestion-control mechanism.

Classic TCP Congestion Control :

- The approach taken by TCP is to have each sender limit the rate at which it sends traffic into its connection as a function of perceived network congestion.
- If a TCP sender perceives that there is little congestion on the path between itself and the destination, then the TCP sender increases its send rate.
- if the sender perceives that there is congestion along the path, then the sender reduces its send rate.

this approach raises three questions.

First, how does a TCP sender limit the rate at which it sends traffic into its connection?

Second, how does a TCP sender perceive that there is congestion on the path between itself and the destination? And

third, what algorithm should the sender use to change its send rate as a function of perceived end-to-end congestion?

Let’s first examine how a TCP sender limits the rate at which it sends traffic into its connection.

- The TCP congestion-control mechanism operating at the sender keeps track of an additional variable, the congestion window.
- The congestion window, denoted cwnd, imposes a constraint on the rate at which a TCP sender can send traffic into the network.
- Specifically, the amount of unacknowledged data at a sender may not exceed the minimum of cwnd and rwnd, that is:
  - LastByteSent – LastByteAcked <= min{cwnd, rwnd}
- The sender’s send rate is roughly cwnd/RTT bytes/sec. By adjusting the value of cwnd, the sender can therefore adjust the rate at which it sends data into its connection.

Let’s next consider how a TCP sender perceives that there is congestion on the path between itself and the destination.

- Let us define a “loss event” at a TCP sender as the occurrence of either a timeout or the receipt of three duplicate ACKs from the receiver.
- The dropped datagram, in turn, results in a loss event at the sender—either a timeout or the receipt of three duplicate ACKs—which is taken by the sender to be an indication of congestion on the sender-to-receiver path.
- if acknowledgments arrive at a high rate, then the congestion window will be increased more quickly. Because TCP uses acknowledgments to trigger (or clock) its increase in congestion window size, TCP is said to be self-clocking.

guiding principles:

- A lost segment implies congestion, and hence, the TCP sender's rate should be decreased when a segment is lost.
- An acknowledged segment indicates that the network is delivering the sender's segments to the receiver, and hence, the sender's rate can be increased when an ACK arrives for a previously unacknowledged segment.
- Bandwidth probing.

TCP congestion-control algorithm :

- The algorithm has three major components:
  - (1) slow start,
  - (2) congestion avoidance, and
  - (3) fast recovery.
- Slow start and congestion avoidance are mandatory components of TCP, differing in how they increase the size of cwnd in response to received ACKs.
- Fast recovery is recommended, but not required, for TCP senders.

Slow Start :

- When a TCP connection begins, the value of cwnd is typically initialized to a small value of 1 MSS, resulting in an initial sending rate of roughly MSS/ RTT.
- the available bandwidth to the TCP sender may be much larger than MSS/RTT, the TCP sender would like to find the amount of available bandwidth quickly.
- Thus, in the slow-start state, the value of cwnd begins at 1 MSS and increases by 1 MSS every time a transmitted segment is first acknowledged.

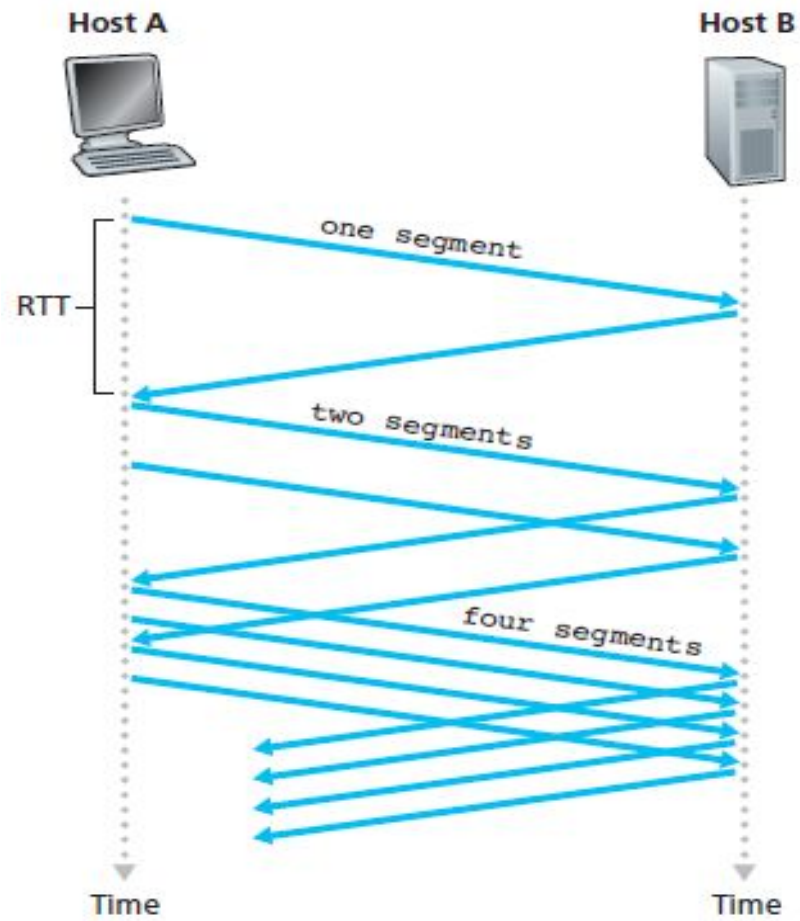


Figure : TCP slow start

Congestion Avoidance :

- On entry to the congestion-avoidance state, the value of cwnd is approximately half its value when congestion was last encountered.
- Increases the value of cwnd by just a single MSS every RTT.
- This can be accomplished in several ways. A common approach is for the TCP sender to increase cwnd by MSS bytes (MSS/cwnd) whenever a new acknowledgment arrives.
- when the triple duplicate ACKs were received. The fast-recovery state is then entered.

Fast Recovery :

- In fast recovery, the value of cwnd is increased by 1 MSS for every duplicate ACK received for the missing segment that caused TCP to enter the fast-recovery state.
- Eventually, when an ACK arrives for the missing segment, TCP enters the congestion-avoidance state after deflating cwnd.
- If a timeout event occurs, fast recovery transitions to the slow-start state after performing the same actions as in slow start and congestion avoidance.
- Fast recovery is a recommended, but not required, component of TCP.
- It is interesting that an early version of TCP, known as TCP Tahoe, unconditionally cut its congestion window to 1 MSS and entered the slow-start phase after either a timeout-indicated or triple-duplicate-ACK-indicated loss event.
- The newer version of TCP, TCP Reno, incorporated fast recovery.

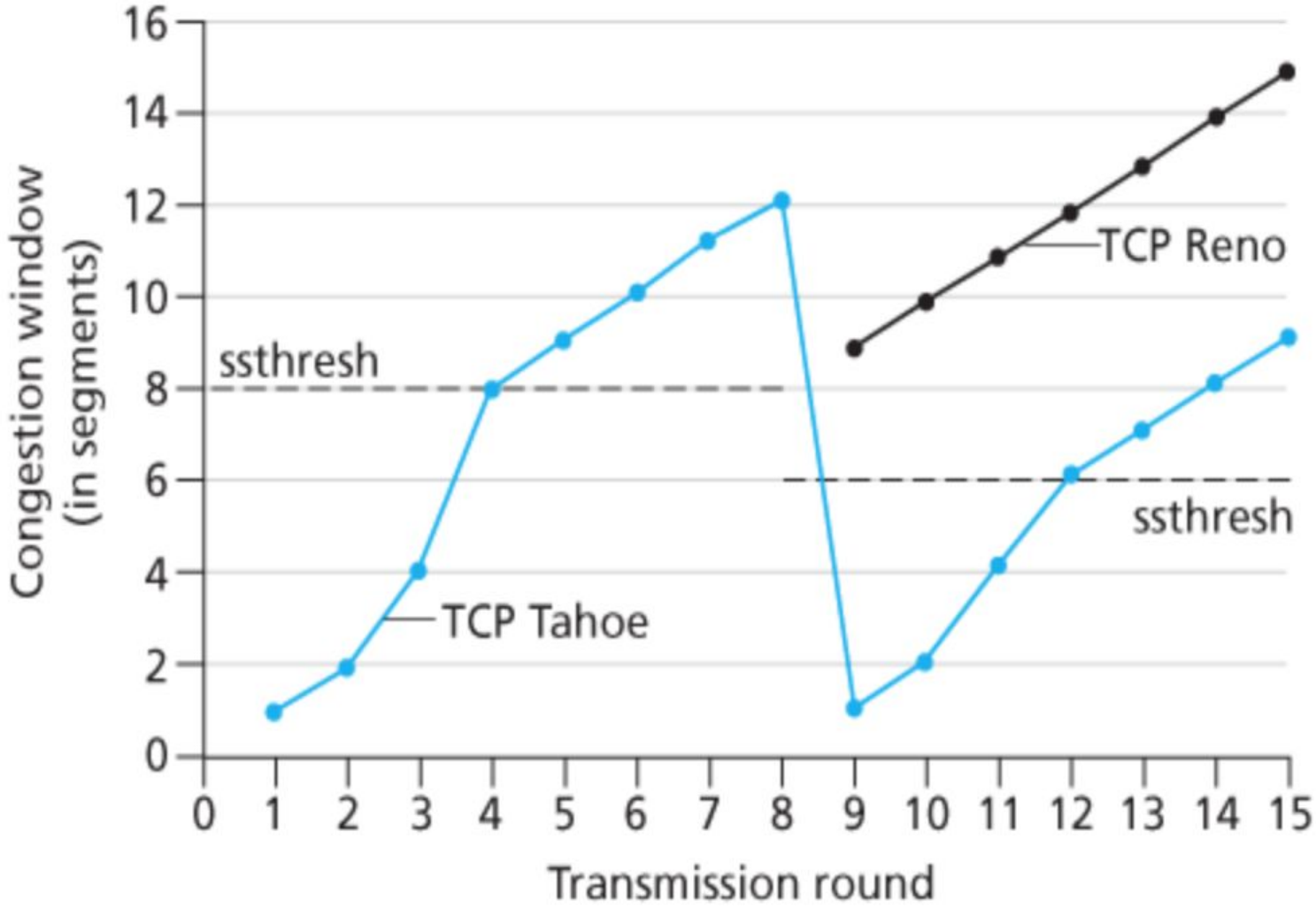


Figure : Evolution of TCP's congestion window (Tahoe and Reno)

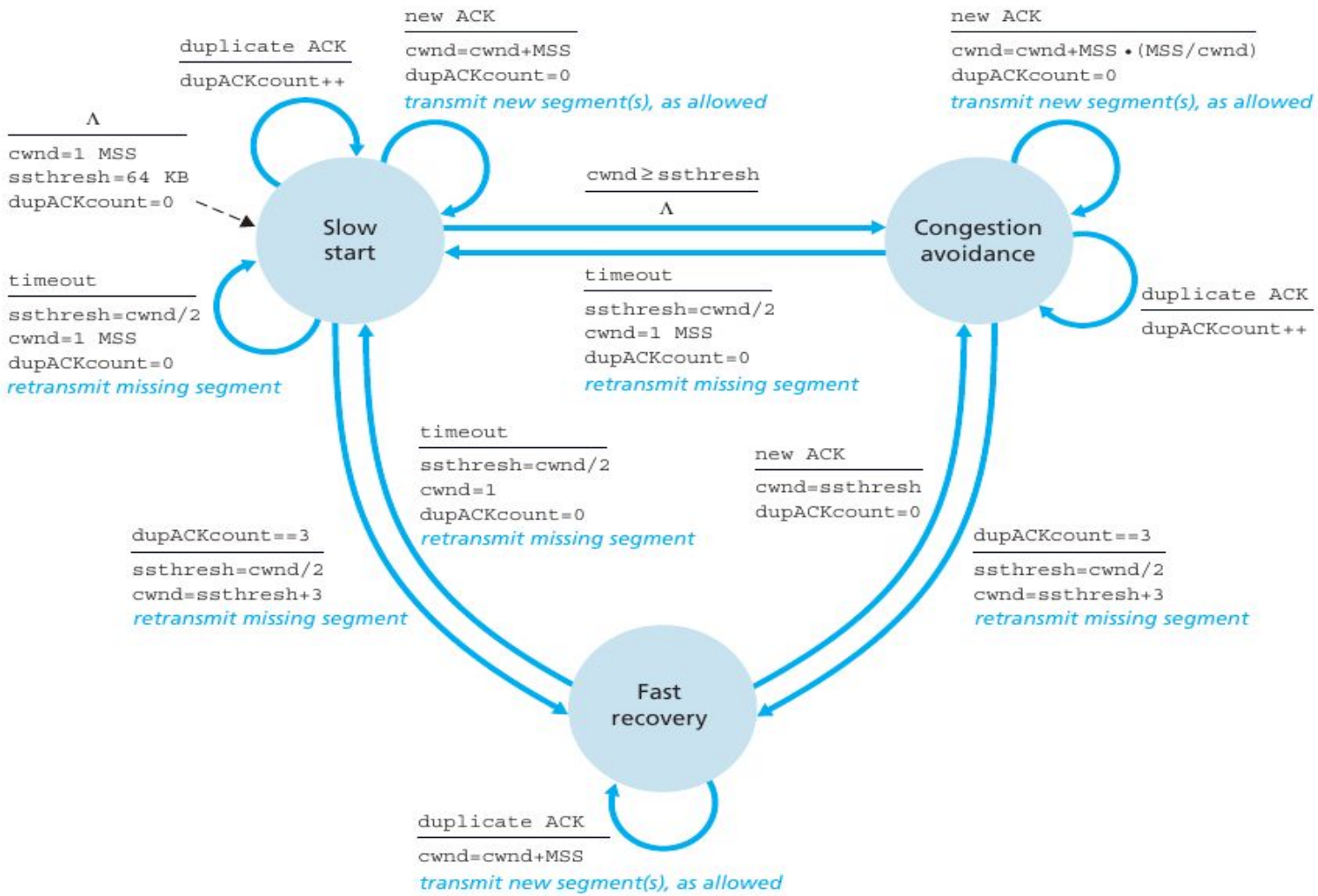


Figure 3.52 ♦ FSM description of TCP congestion control

TCP Congestion Control: Retrospective :

TCP's congestion control consists of linear (additive) increase in cwnd of 1 MSS per RTT and then a halving (multiplicative decrease) of cwnd on a triple duplicate-ACK event. For this reason, TCP congestion control is often referred to as an additive-increase, multiplicative decrease (AIMD) form of congestion control.

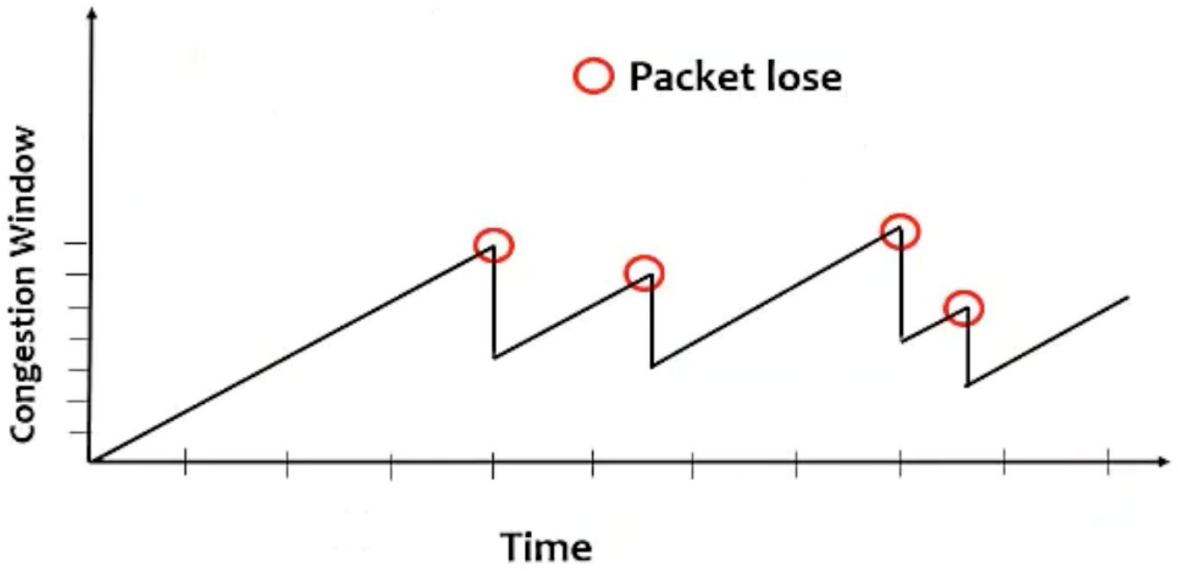
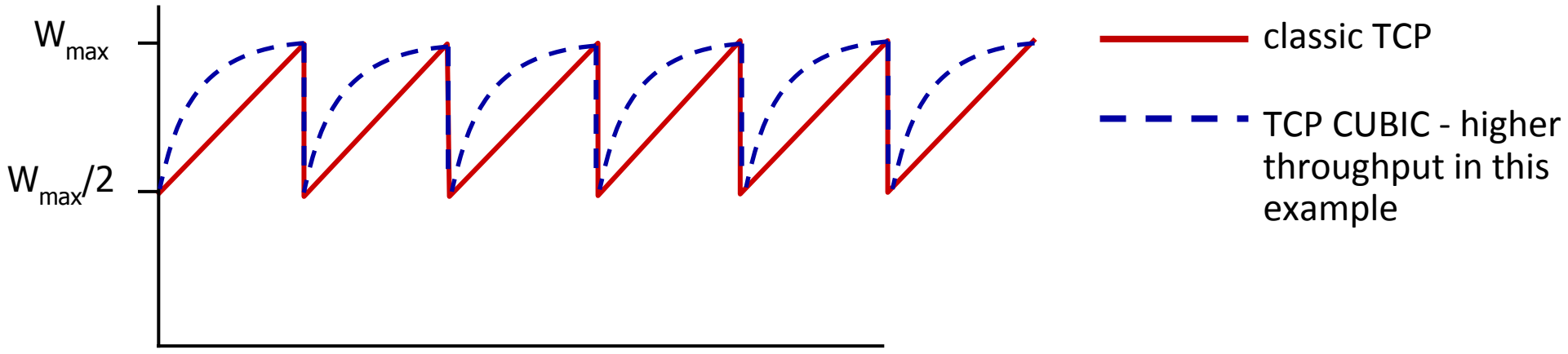


Figure : Additive-increase, multiplicative-decrease congestion control

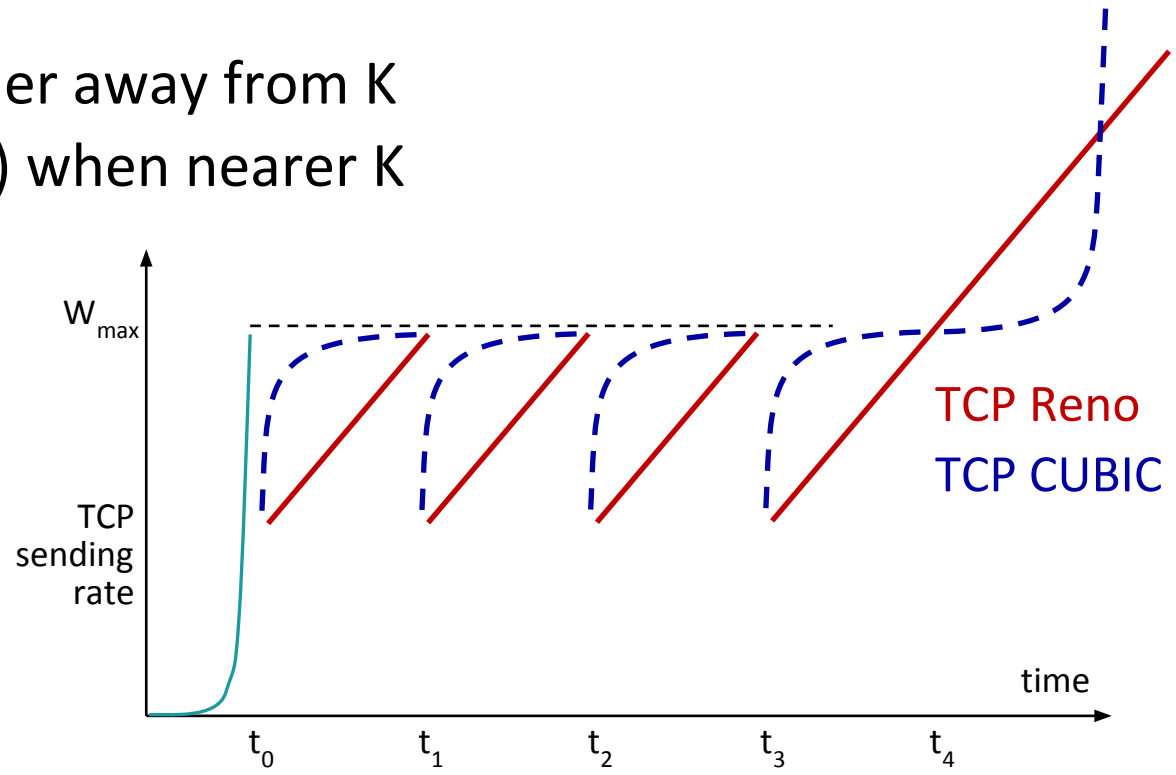
TCP CUBIC

- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



TCP CUBIC

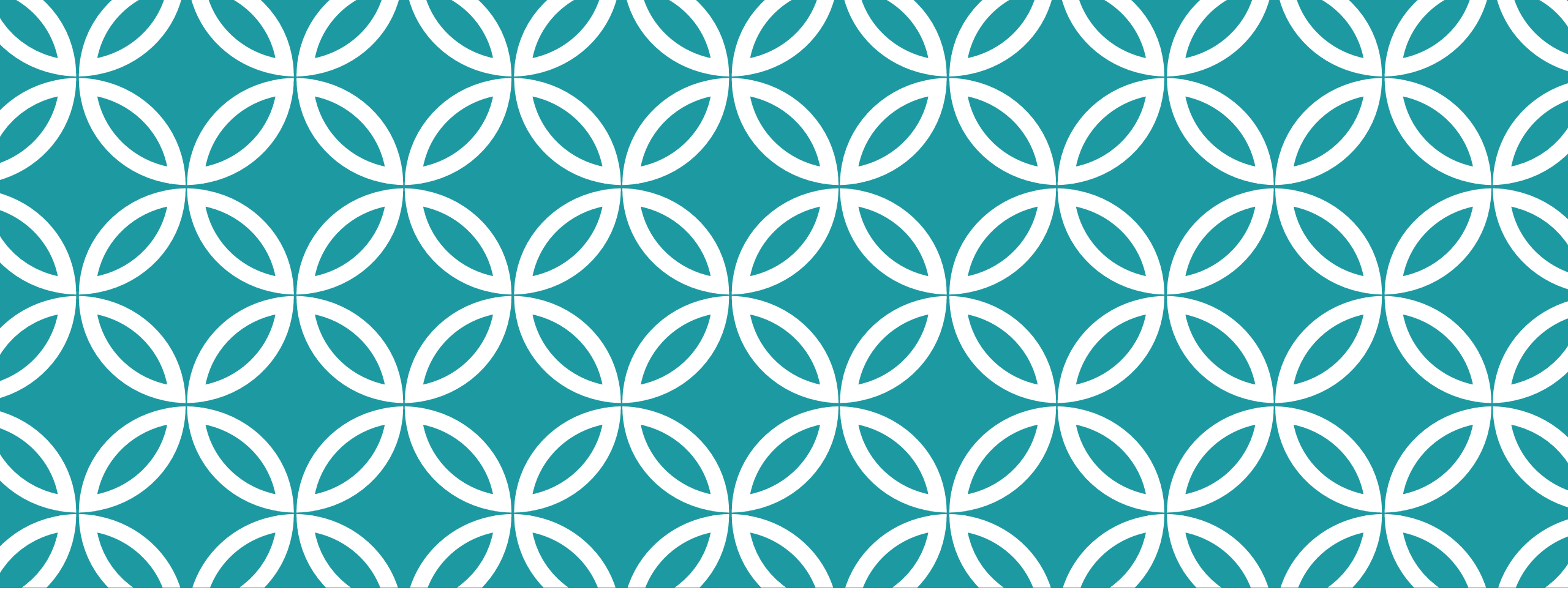
- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers



Macroscopic Description of TCP Reno Throughput

- During a particular round-trip interval, the rate at which TCP sends data is a function of the congestion window and the current RTT.
- When the window size is w bytes and the current round-trip time is RTT seconds, then TCP's transmission rate is roughly w/RTT.
- TCP then probes for additional bandwidth by increasing w by 1 MSS each RTT until a loss event occurs. Denote by W the value of w when a loss event occurs.
- Assuming that RTT and W are approximately constant over the duration of the connection, the TCP transmission rate ranges from $W/(2 \cdot RTT)$ to W/RTT.
- These assumptions lead to a highly simplified macroscopic model for the steady state behavior of TCP.

- The network drops a packet from the connection when the rate increases to W/RTT;
- the rate is then cut in half and then increases by MSS/RTT every RTT until it again reaches W/RTT. This process repeats itself over and over again. Because TCP's throughput (that is, rate) increases linearly between the two extreme values,
- we have average throughput of a connection = 0.75 . W RTT



ADDITIONAL CONTENT

Addressing: Port Numbers :

- For communication, we must define the local host, local process, remote host, and remote process.
- The local host and the remote host are defined using **IP addresses**.
- To define the processes, we need second identifiers, called **port numbers**.
- In the TCP/IP protocol suite, the port numbers are integers between 0 and 65,535.
- The client program defines itself with a port number, called the **ephemeral port number**.
- TCP/IP has decided to use universal port numbers for servers; these are called **well-known port numbers**.

Example : the daytime client process, an application program, can use an ephemeral (temporary) port number, 52,000, to identify itself, the daytime server process must use the well-known (permanent) port number 13.

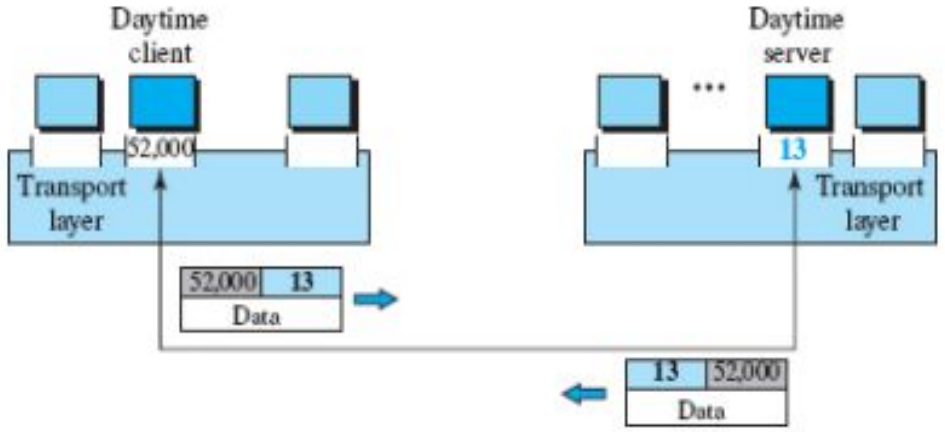


Figure :Port Numbers

IP vs Port Number

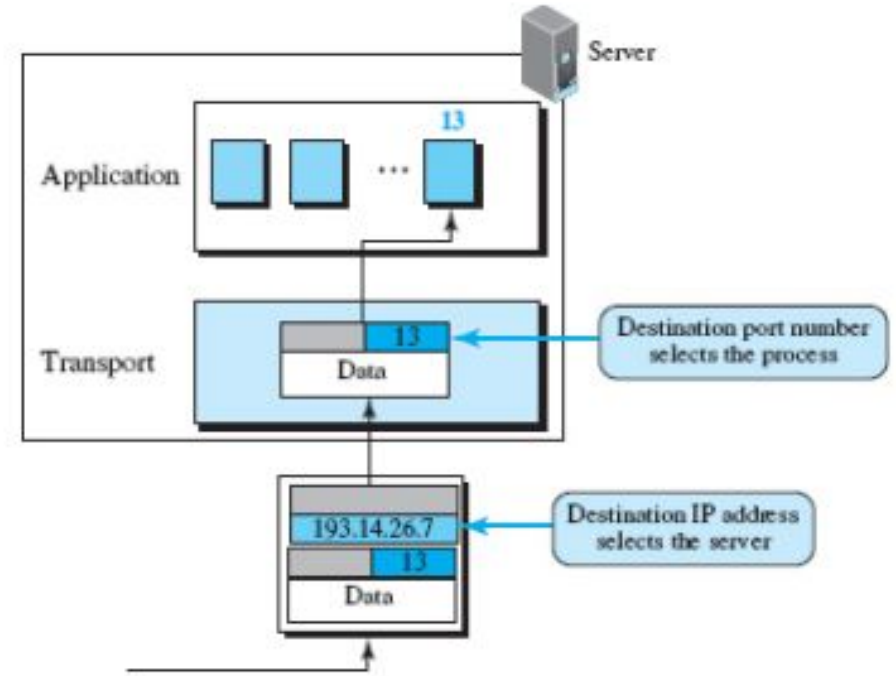
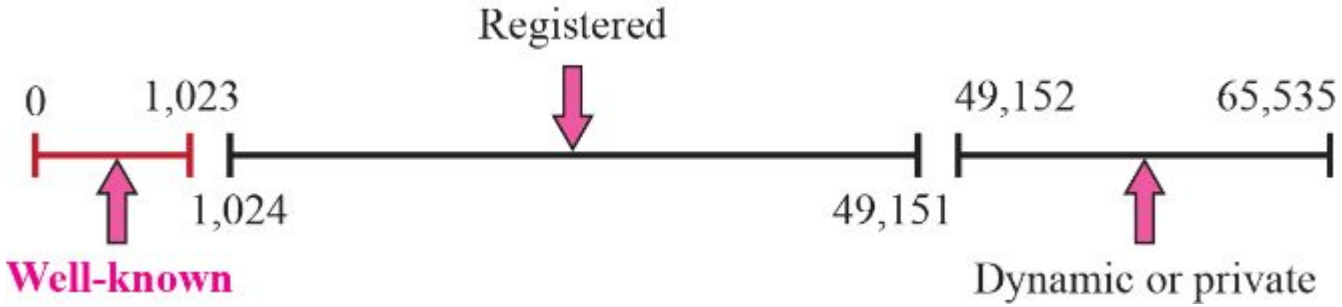


Figure :IP versus Port Numbers

ICANN Ranges :

- The Internet Corporation for Assigned Names and Numbers (ICANN) has divided the port numbers into three ranges: well-known, registered, and dynamic.



- Well-known ports : The ports ranging from 0 to 1023 are assigned and controlled by ICANN. These are the well-known ports.
- Registered ports. The ports ranging from 1024 to 49,151 are not assigned or controlled by ICANN. They can only be registered with ICANN to prevent duplication.
- Dynamic ports. The ports ranging from 49,152 to 65,535 are neither controlled nor socket addresses.

- A transport-layer protocol in the TCP suite needs both the IP address and the port number, at each end, to make a connection. The combination of an IP address and a port number is called a socket address.

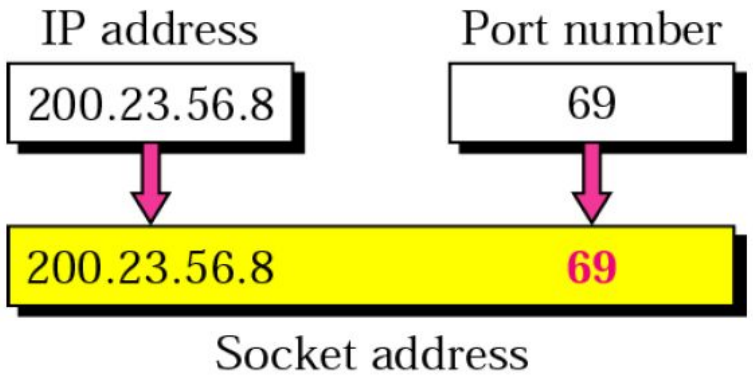


Figure : Socket address

Encapsulation and Decapsulation :

- To send a message from one process to another, the transport-layer protocol encapsulates and decapsulates messages.

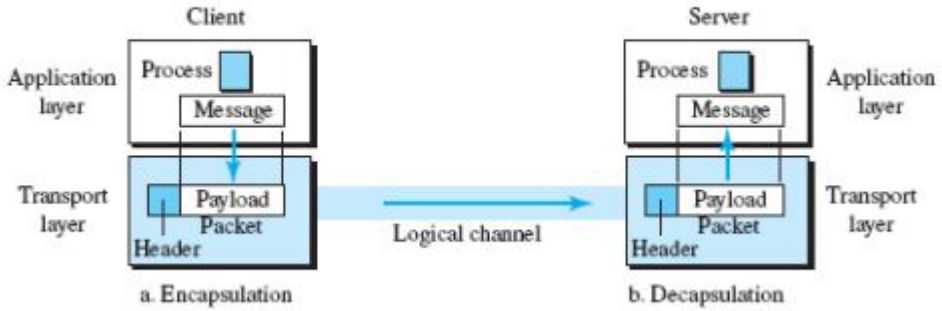
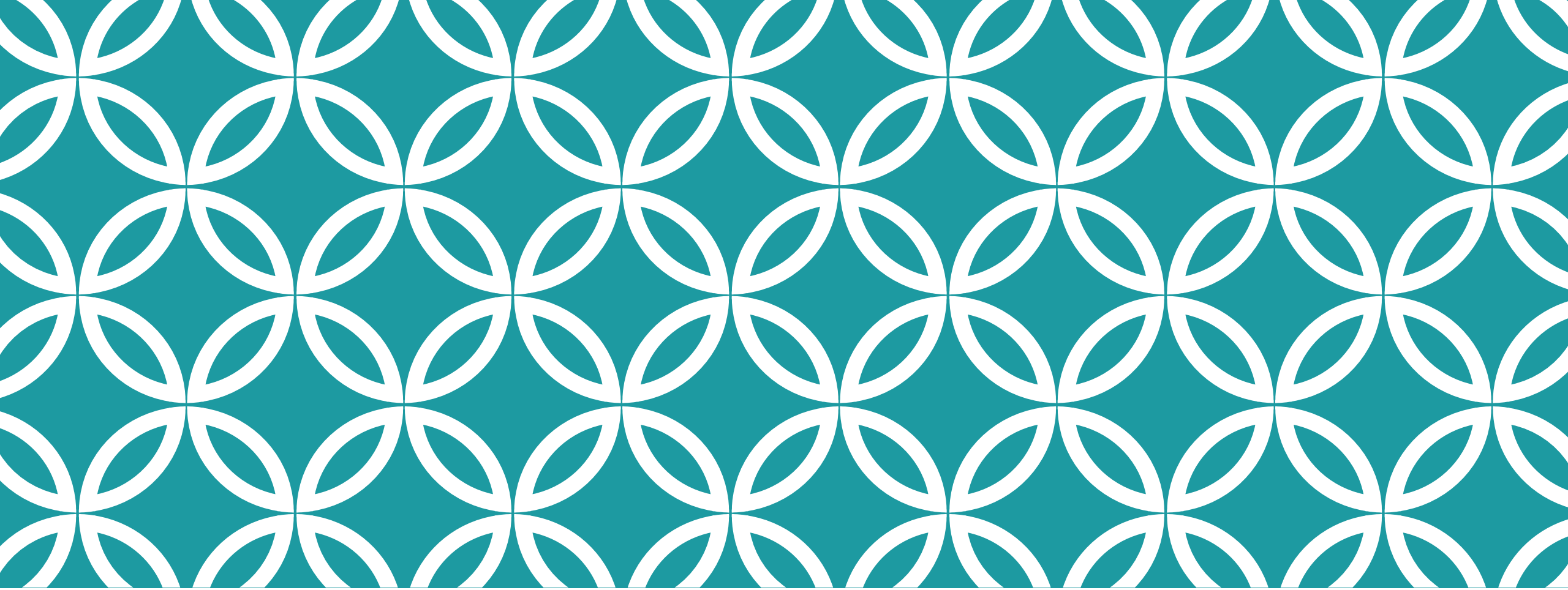


Figure : Encapsulation and decapsulation



END