ITH508 컴퓨터망

Transport Layer

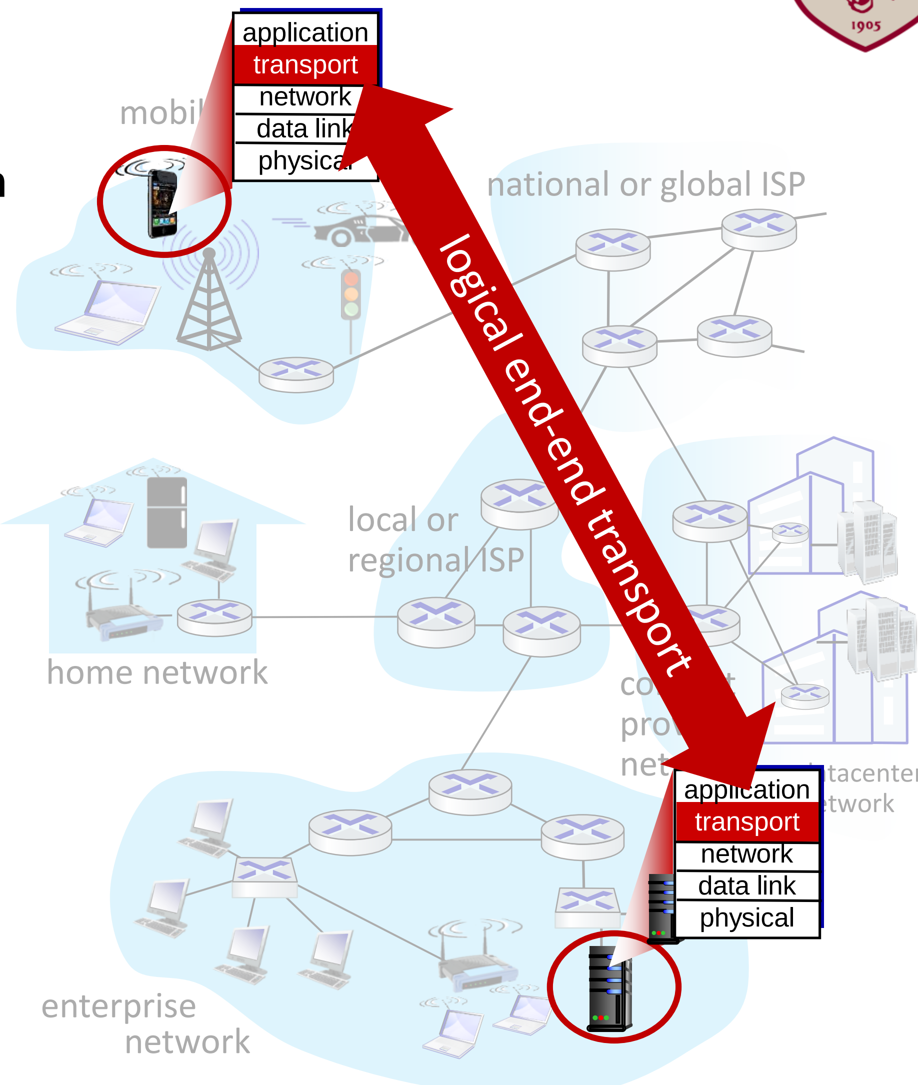
Hwangnam Kim
hnkim@korea.ac.kr
School of Electrical Engineering
Korea University

Transport Layer

- **Principles behind transport layer services:**
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- **Transport layer protocols in the Internet:**
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Transport Services and Protocols

- **Provide *logical communication* between app processes running on different hosts**
- **Transport protocols run in end systems**
  - Sender side:
    - Breaks app messages into **segments**
    - Passes to network layer
  - Receiver side:
    - Reassembles **segments** into messages
    - Passes to app layer
- **More than one transport protocol available to apps**
  - Internet: TCP and UDP

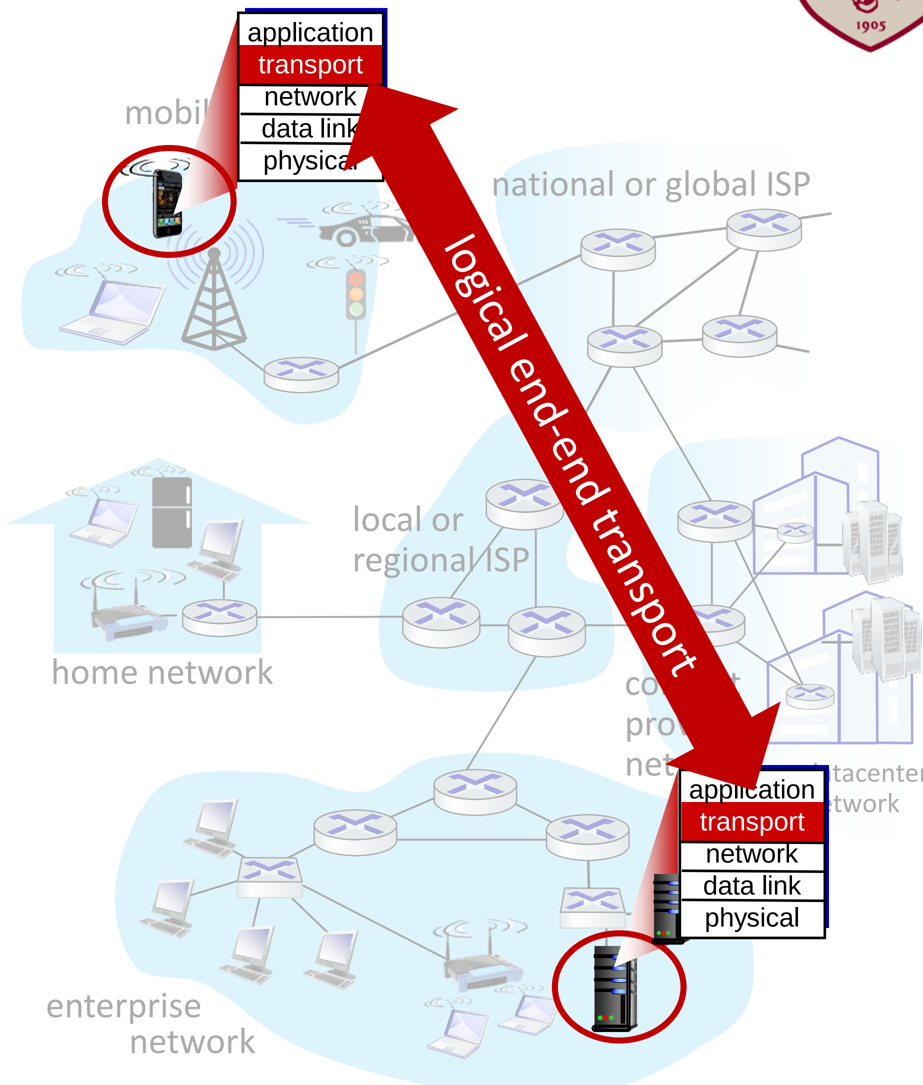


Transport vs. Network Layer

- ***Network layer:*** **logical communication between hosts**
- ***Transport layer:*** **logical communication between processes**
  - Relies on, enhances, network layer services

Internet Transport-layer Protocols

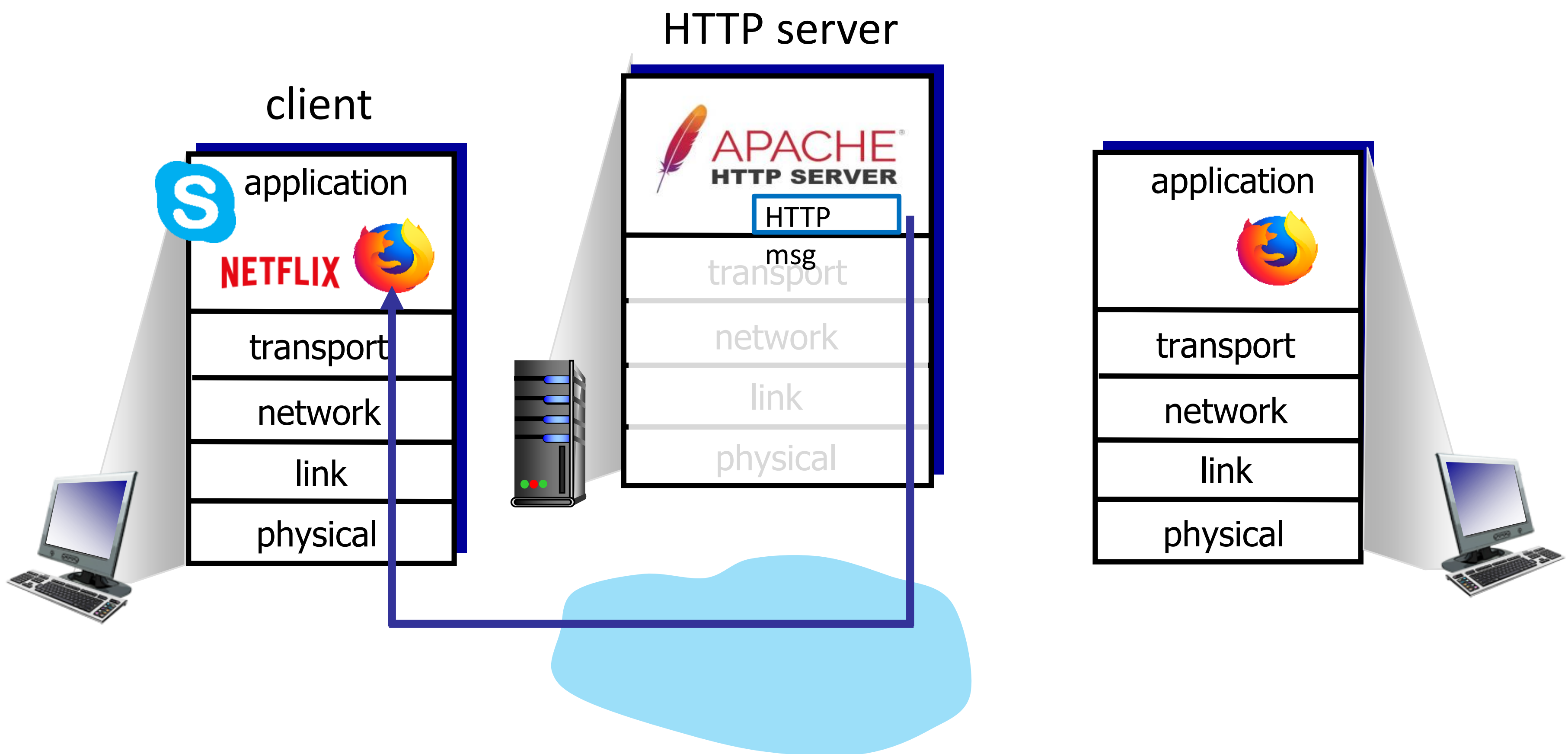
- **Reliable, in-order delivery: TCP**
  - Congestion control
  - Flow control
  - Error control
  - Connection setup
- **Unreliable, unordered delivery: UDP**
  - Simple extension of "best-effort" IP
- **Services not available:**
  - Delay guarantees
  - Bandwidth guarantees



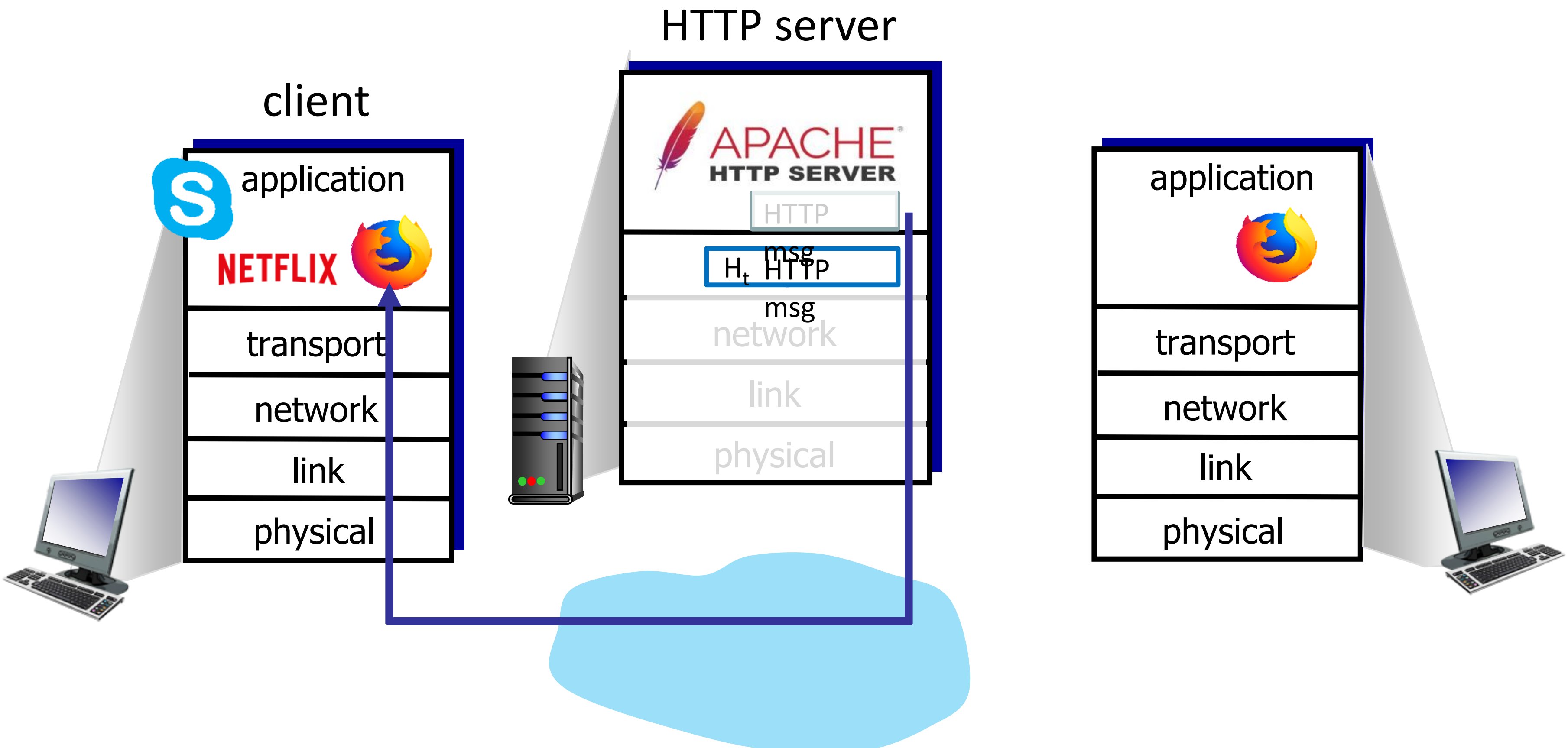
Common feature

MULTIPLEXING/DEMULTIPLEXING

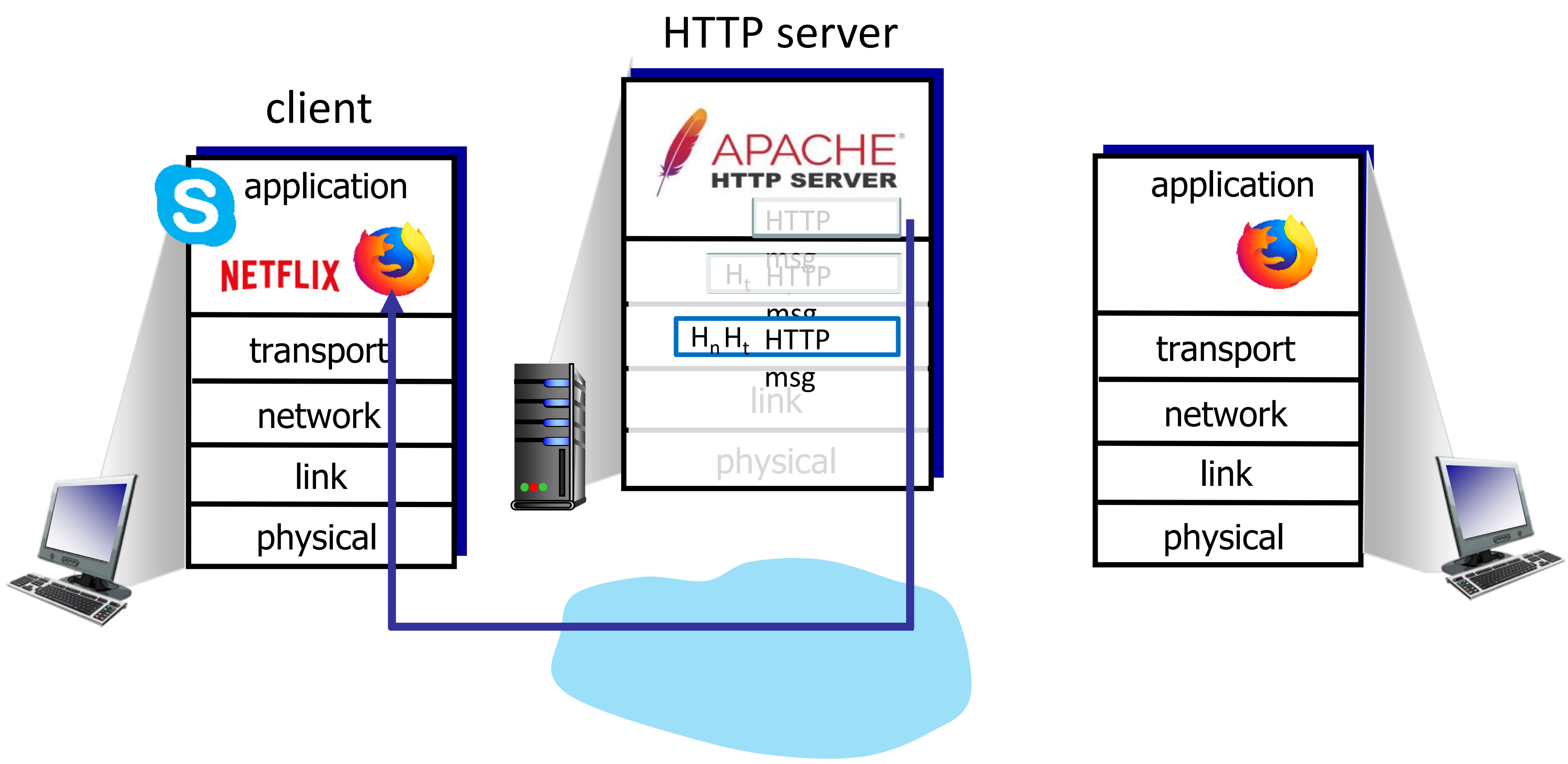
Multiplexing/Demultiplexing



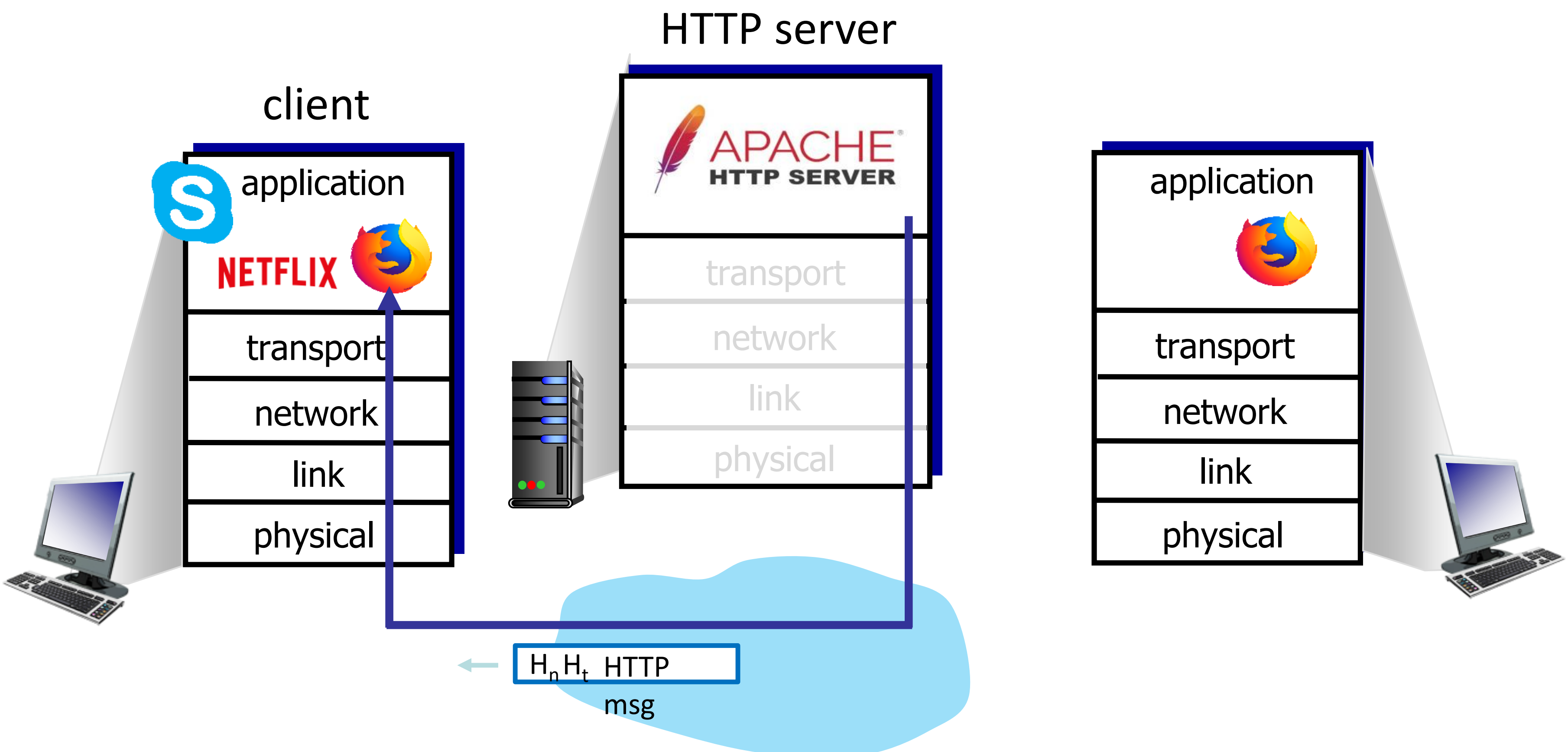
Multiplexing/Demultiplexing



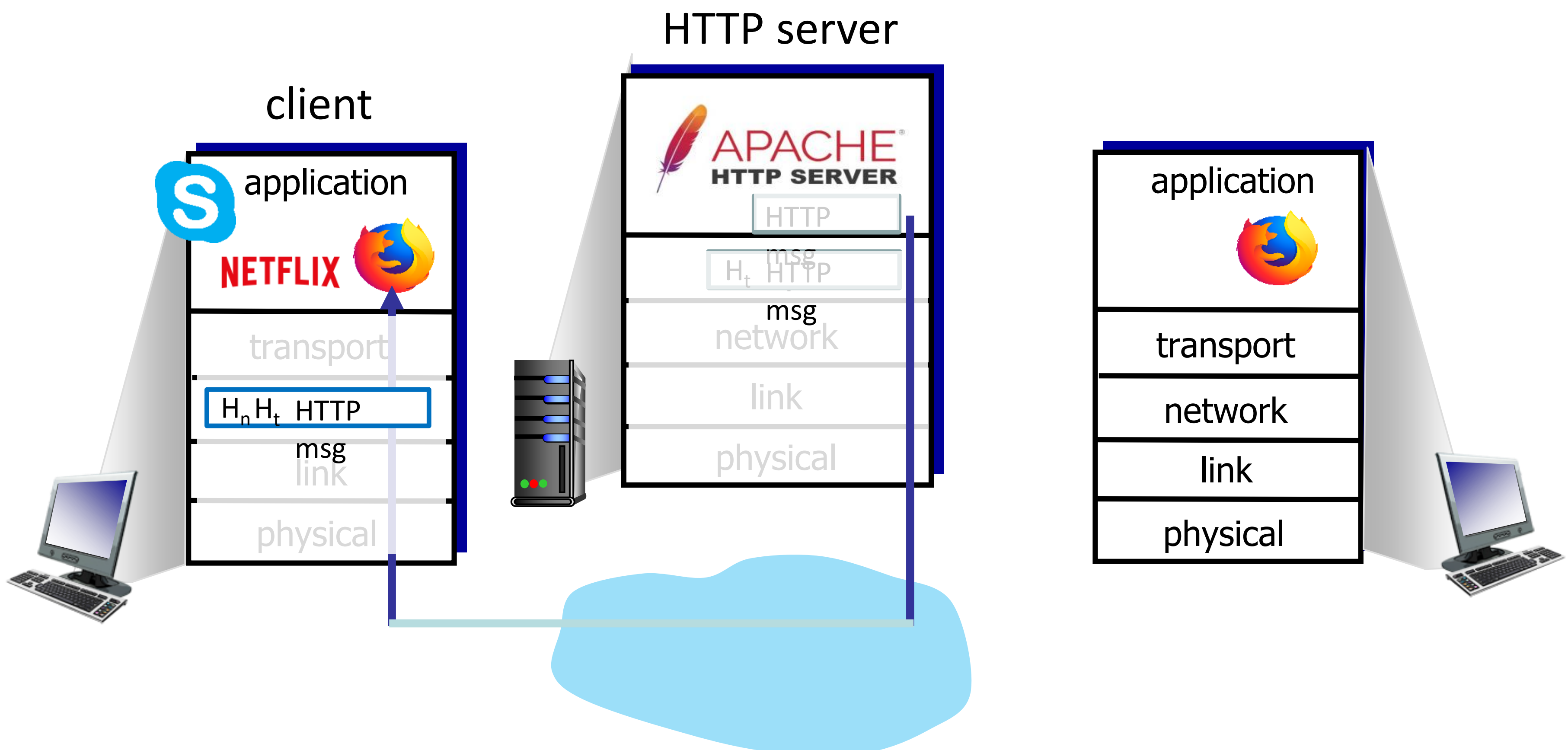
Multiplexing/Demultiplexing



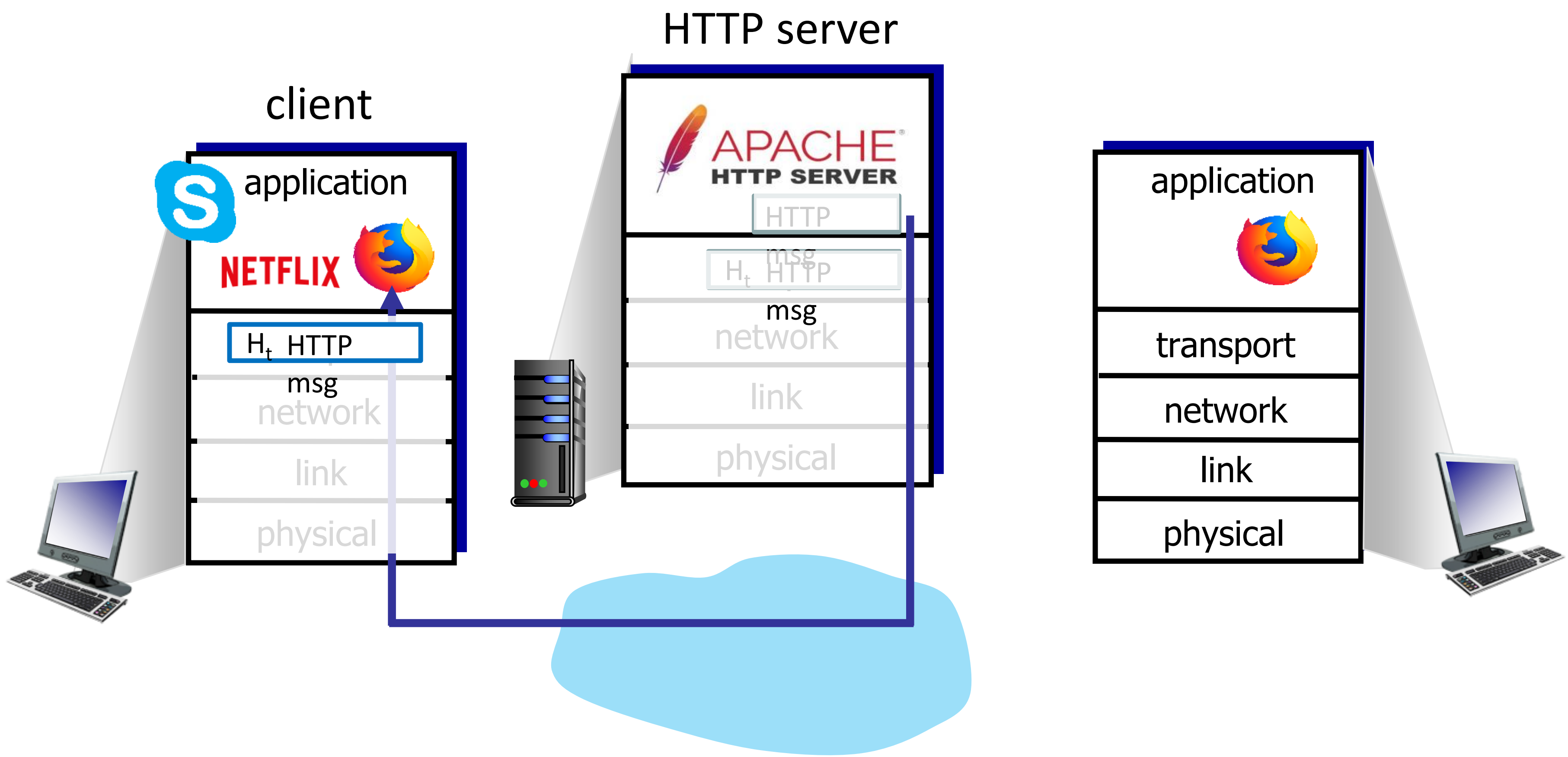
Multiplexing/Demultiplexing



Multiplexing/Demultiplexing

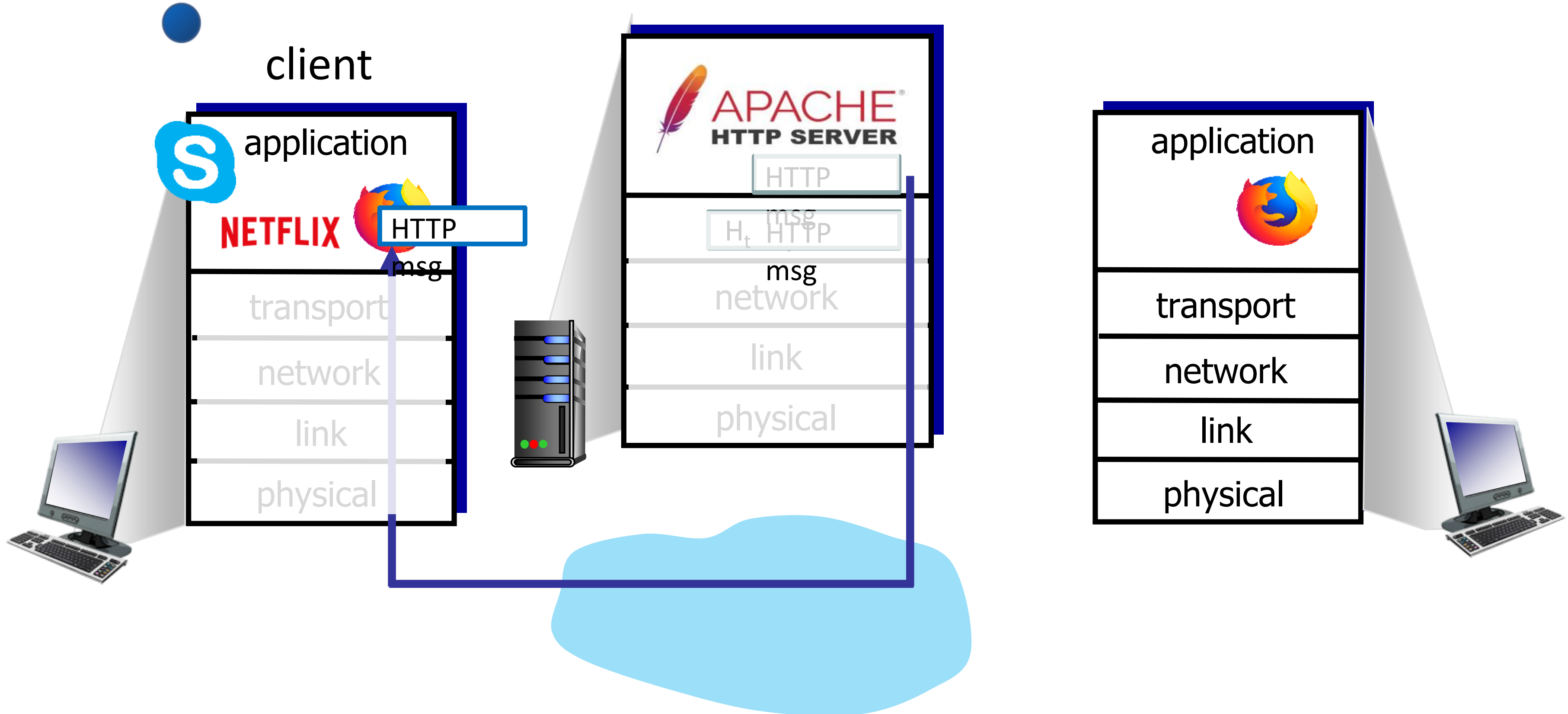


Multiplexing/Demultiplexing



Multiplexing/Demultiplexing

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



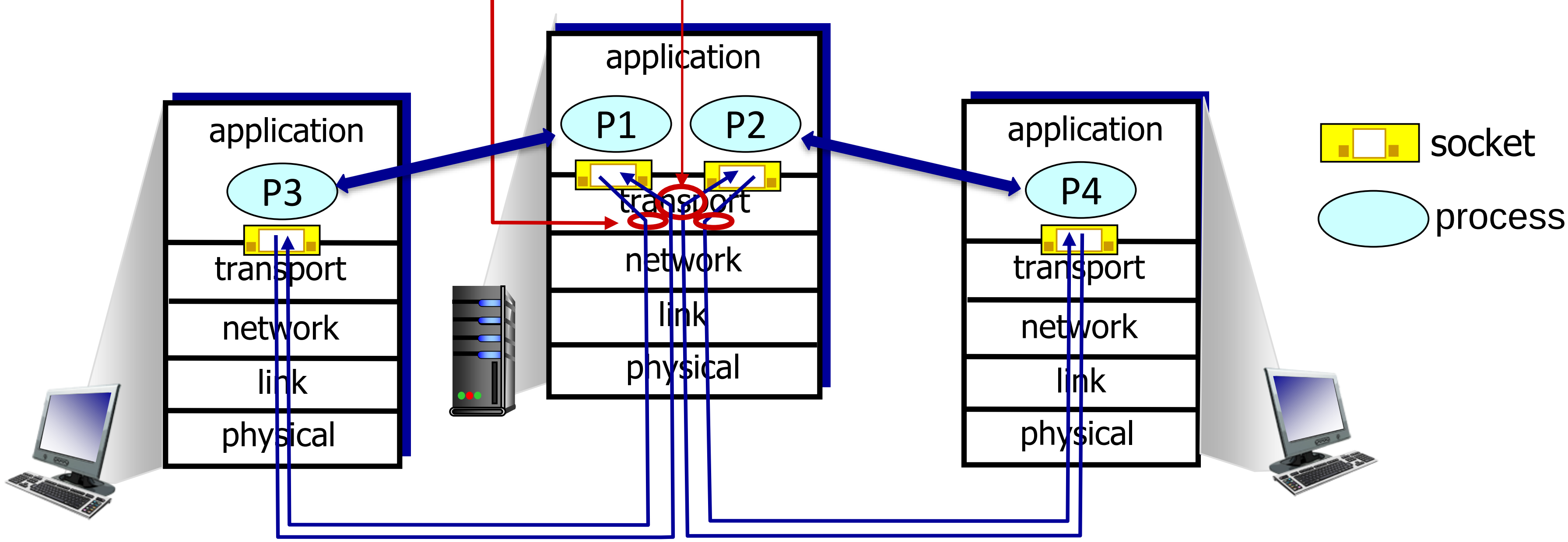
Multiplexing/Demultiplexing

multiplexing as sender:

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

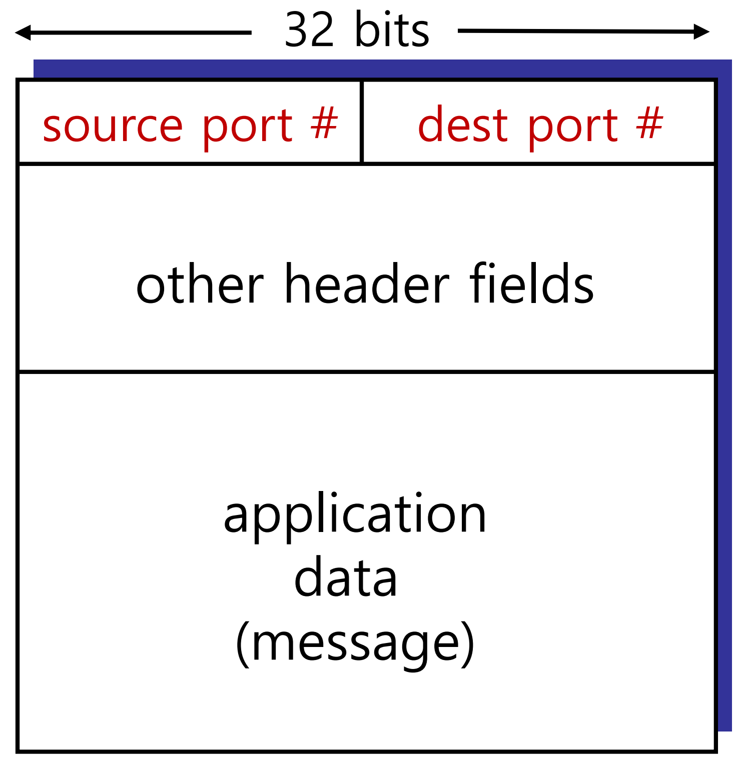
demultiplexing as receiver:

use header info to deliver received segments to correct socket



Multiplexing/Demultiplexing

- **Multiplexing at send host**
  - Gathering data from multiple sockets, enveloping data with header
- **Demultiplexing at receive host**
  - Delivering received segments to correct socket
  - Host receives IP datagrams
    - Each datagram has source IP address, destination IP address
    - Each datagram carries 1 transport-layer segment
    - Each segment has source, destination port number
      (recall: well-known port numbers for specific applications)
- **Host uses IP addresses & port numbers to direct segment to appropriate socket**



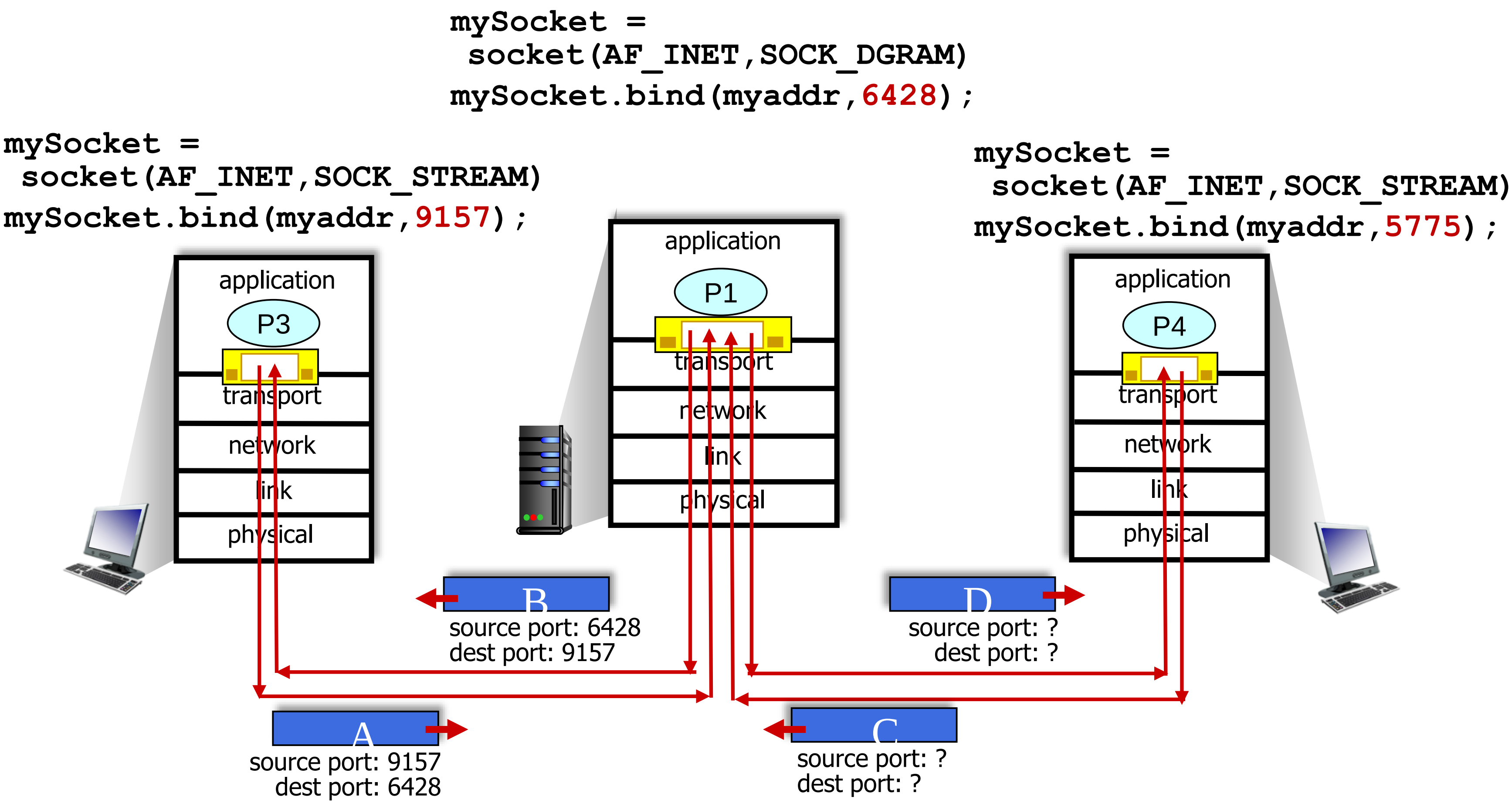
TCP/UDP segment format

Connectionless Demultiplexing

- **Create sockets with port numbers**
- **UDP socket identified by two-tuple:**

  (dest IP address, dest port number)
- **When host receives UDP segment:**
  - Checks destination port number in segment
  - Directs UDP segment to socket with that port number

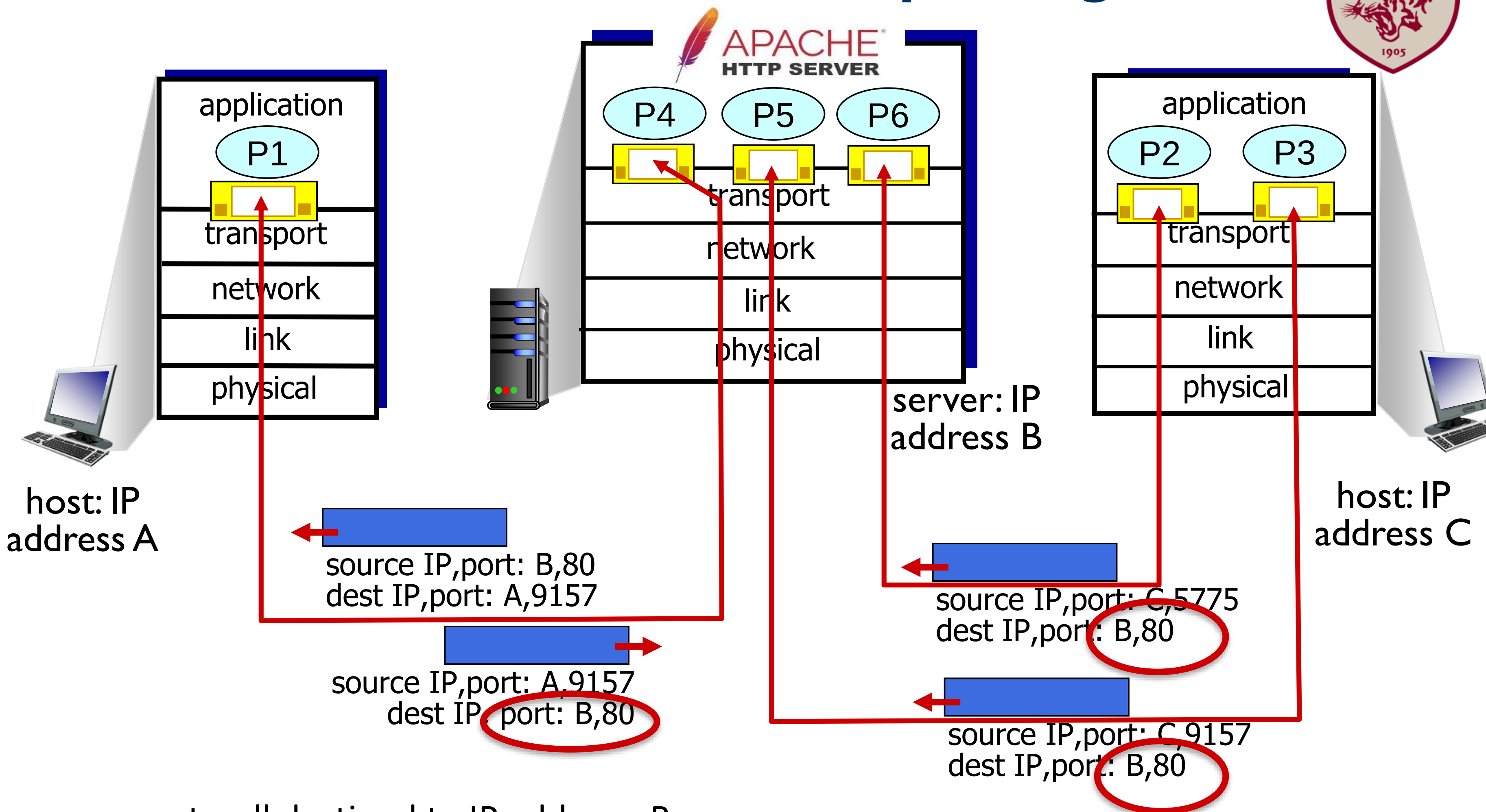
Connectionless Demultiplexing: an Example



Connection-Oriented Demultiplexing

- **TCP socket identified by 4-tuple:**
  - **Source IP address**
  - **Source port number**
  - **Destination IP address**
  - **Destination port number**
- **Receiving host uses all four values to direct segment to appropriate socket**
- **Server host may support many simultaneous TCP sockets:**
  - Each socket identified by its own 4-tuple
- **Web servers have different sockets for each connecting client**
  - Non-persistent HTTP will have different socket for each request

Connection-Oriented Demultiplexing (Cont)



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

UDP

UDP: User Datagram Protocol

- **"bare bones" Internet transport protocol**
- **"best effort" service, UDP segments may be:**
  - Lost
  - Delivered out of order to app
- ***Connectionless:***
  - No handshaking between UDP sender and receiver
  - Each UDP segment handled independently of others

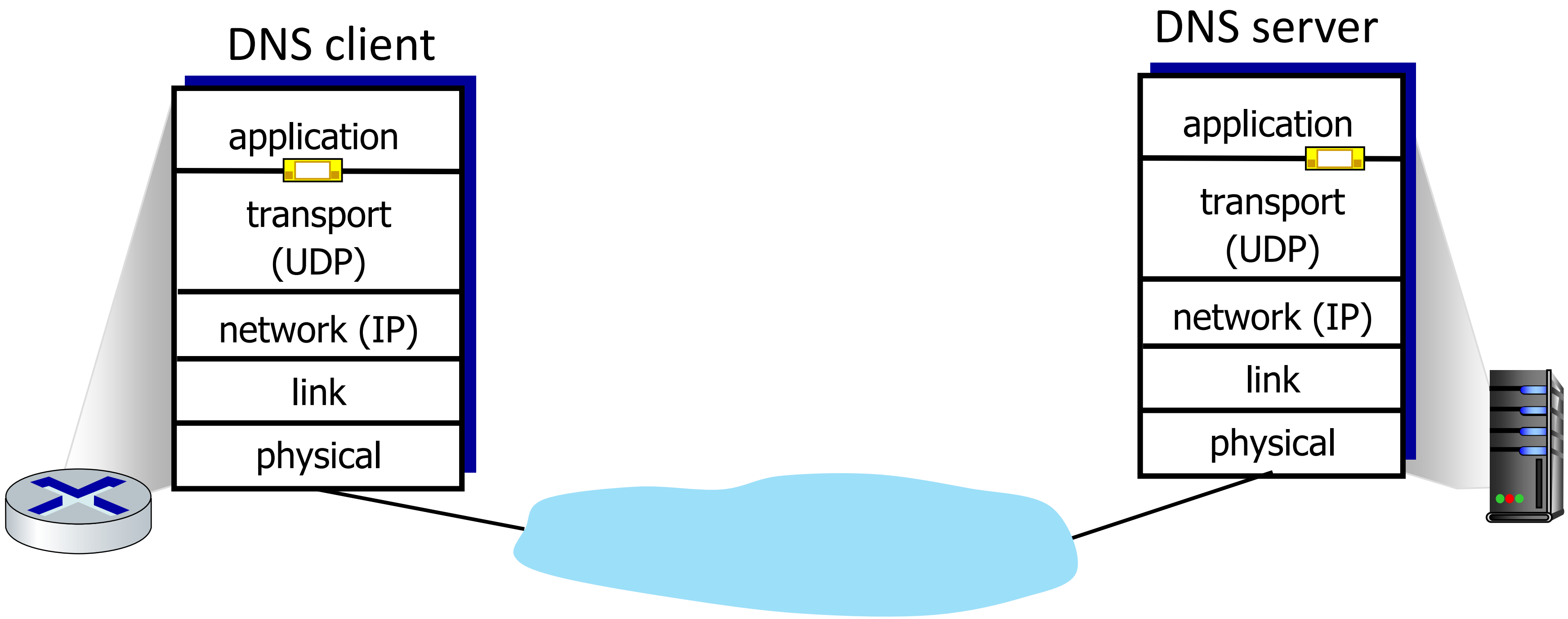
Why is there a UDP?

- **No connection establishment (which can add delay)**
- **Simple: no connection state at sender, receiver**
- **Small segment header**
- **No congestion control:**

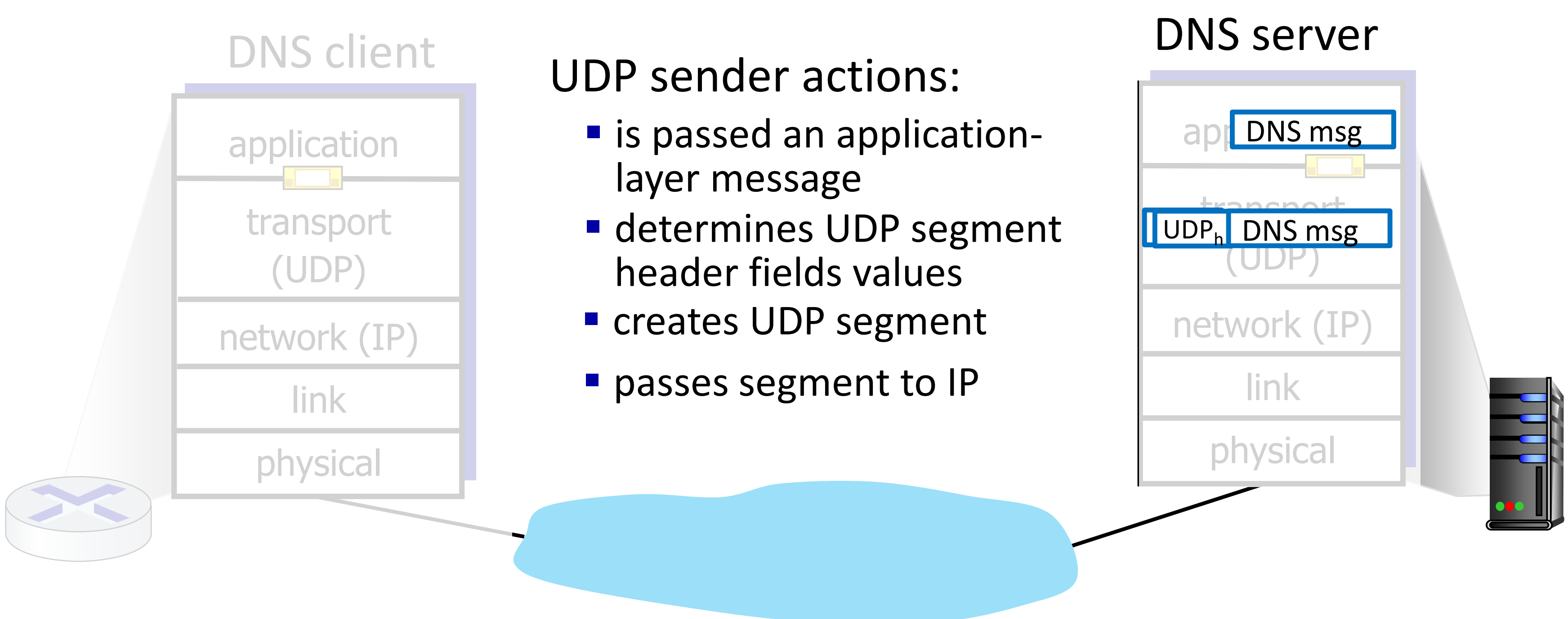
UDP: User Datagram Protocol

- **Other UDP uses**
  - DNS
  - SNMP
  - HTTP/3
- **Reliable transfer over UDP: add reliability at application layer**
  - Application-specific error recovery!

UDP: User Datagram Protocol



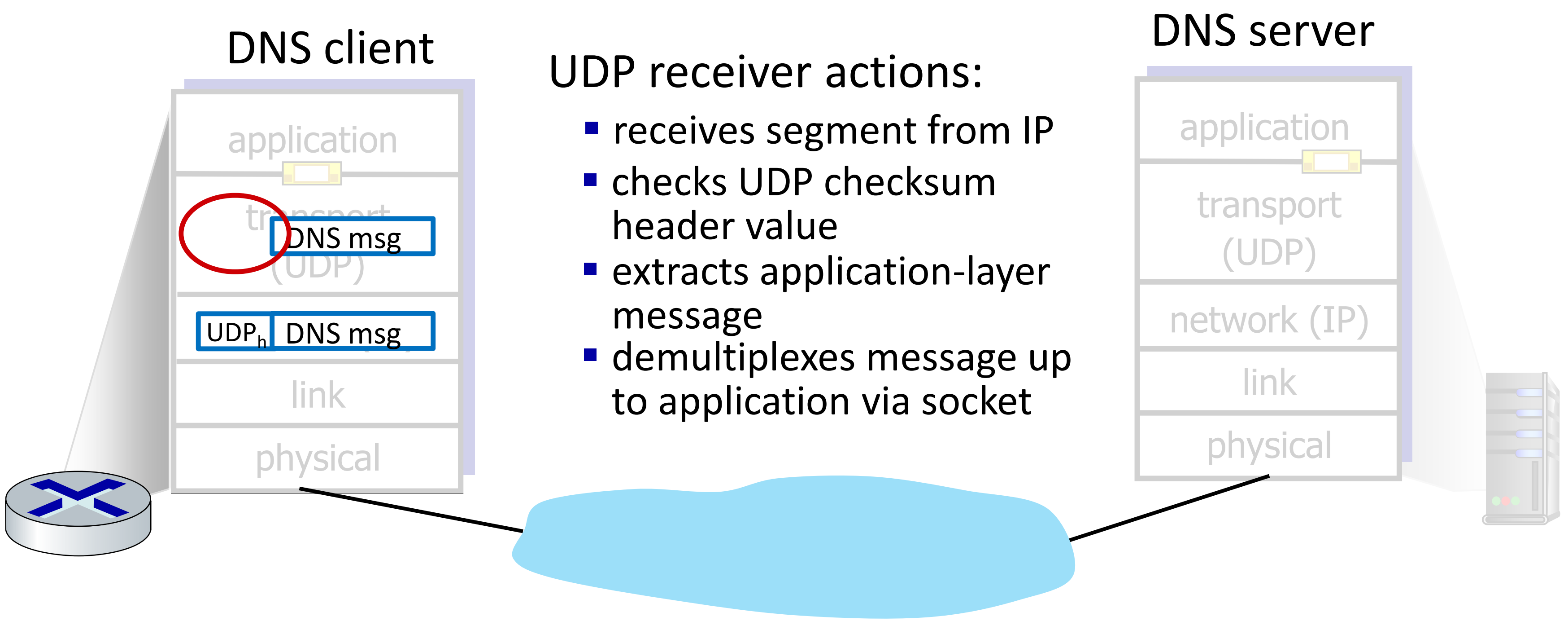
UDP: User Datagram Protocol



UDP sender actions:

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

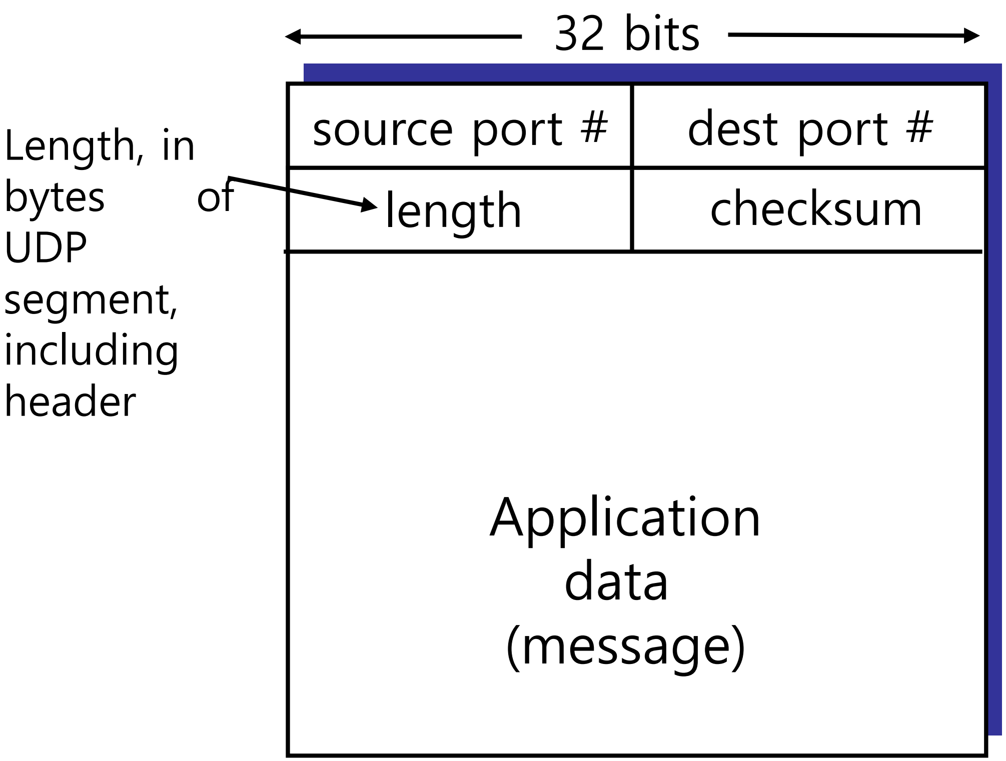
UDP: User Datagram Protocol



UDP receiver actions:

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

UDP: User Datagram Protocol



UDP segment format

UDP Checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- **Treat segment contents as sequence of 16-bit integers**
- **Checksum: addition (1's complement of the sum) of segment contents**
- **Sender puts checksum value into UDP checksum field**

Receiver:

- **Compute checksum of received segment**
- **Check if computed checksum equals checksum field value:**
  - NO - error detected
  - YES - no error detected.

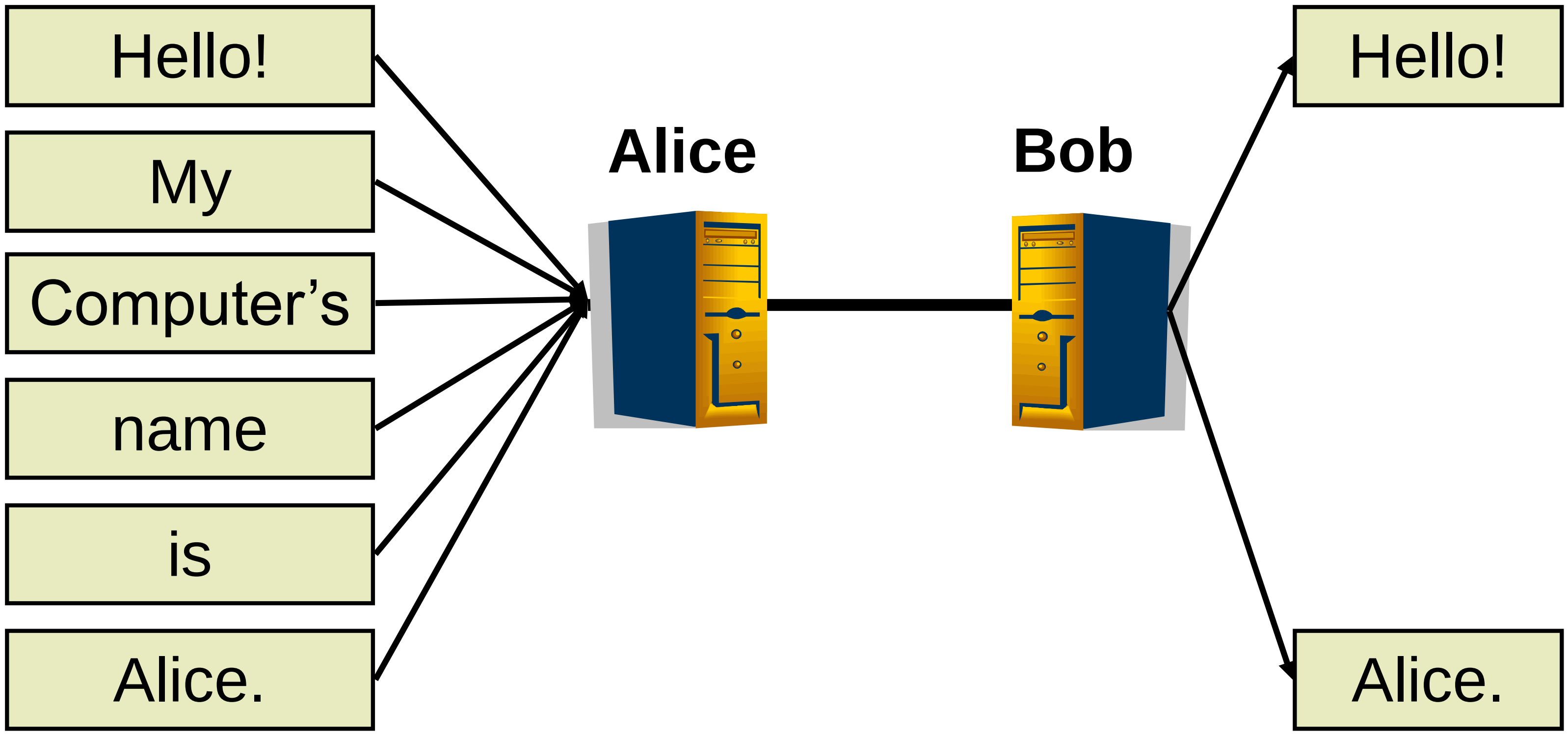
Checksum Example

- **Note**
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
- **Example: add two 16-bit integers**

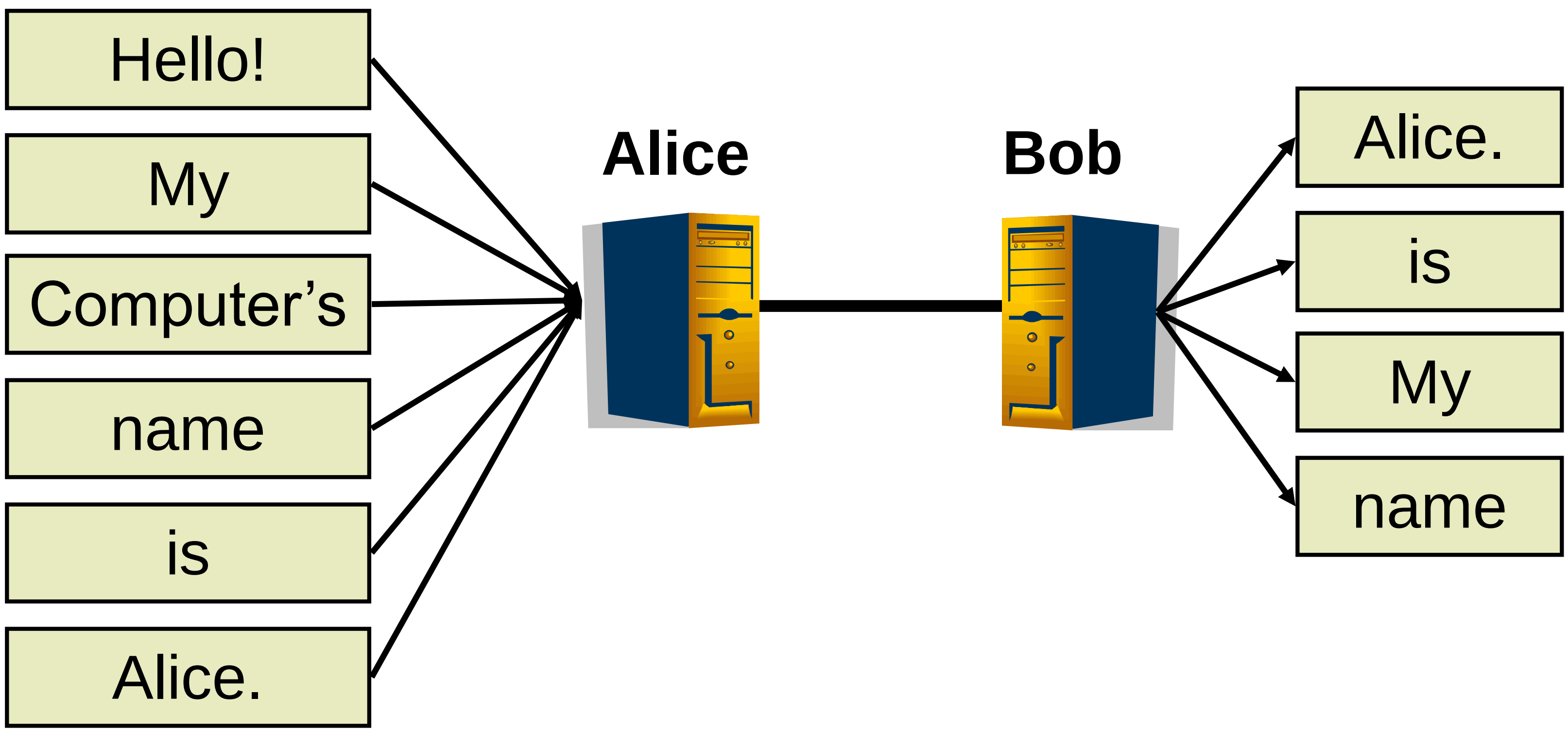
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

TCP

Reliable Transmission



Reliable Transmission



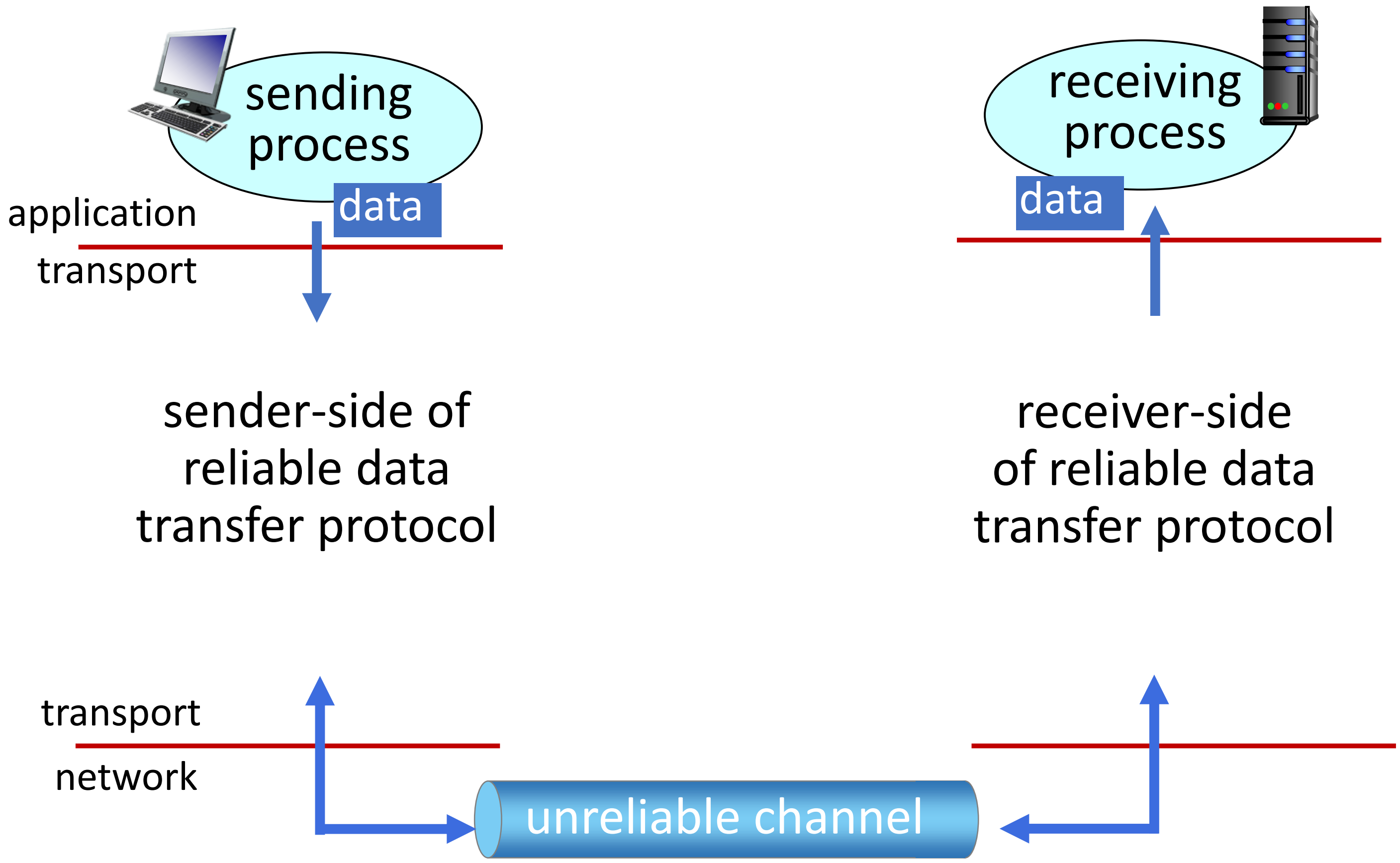
Reliable Transmission

- **Suppose error protection identifies valid and invalid packets**
- **Can we make the channel appear reliable?**
  - Insure **packet delivery**
  - Maintain **packet order**
  - Provide reliability at full link capacity

Reliable Transmission

reliable service *abstraction*

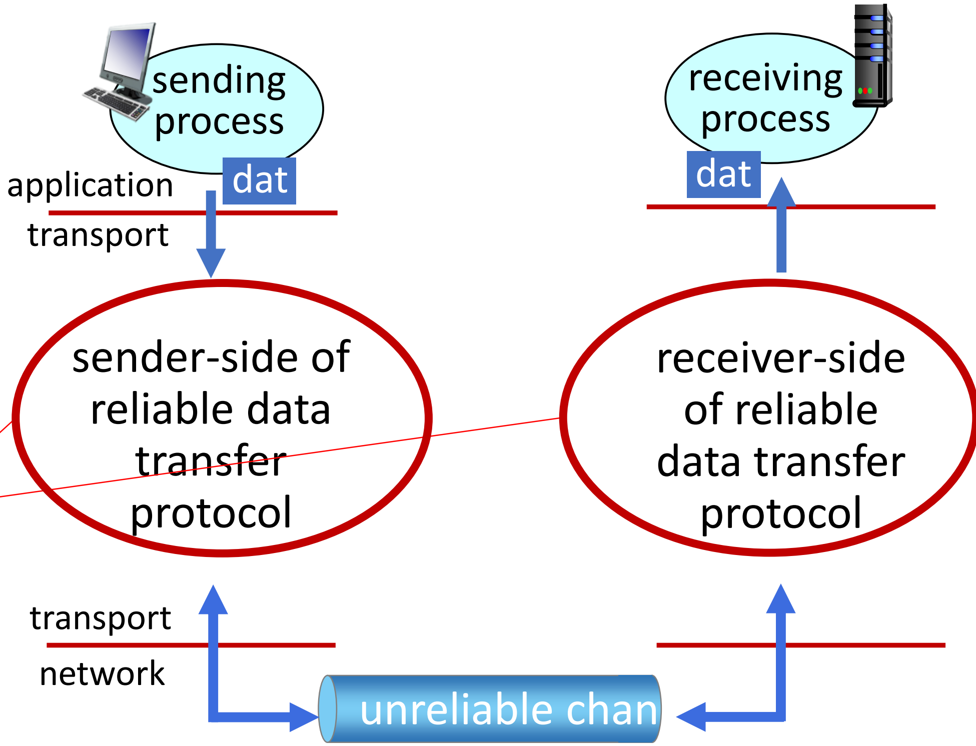
Reliable Transmission



reliable service *implementation*

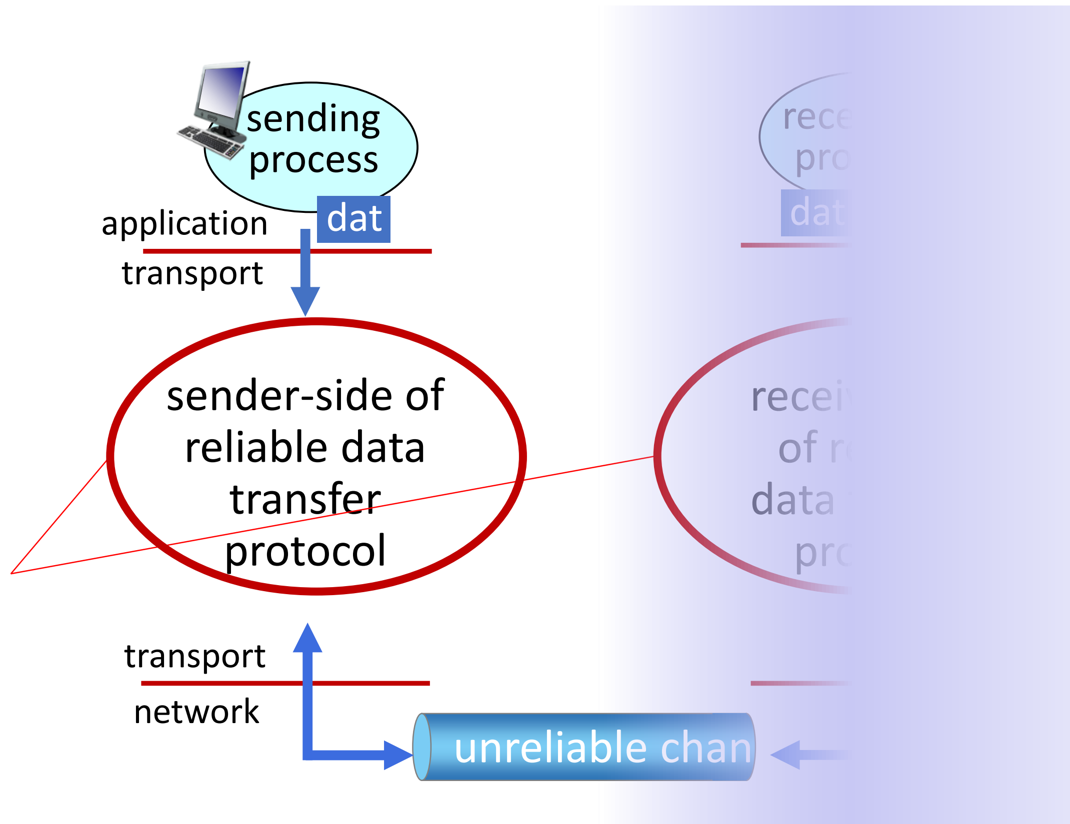
Reliable Transmission

Complexity of reliable data transfer protocol will depend (strongly) on characteristics of unreliable channel (lose, corrupt, reorder data?)



reliable service *implementation*

Reliable Transmission

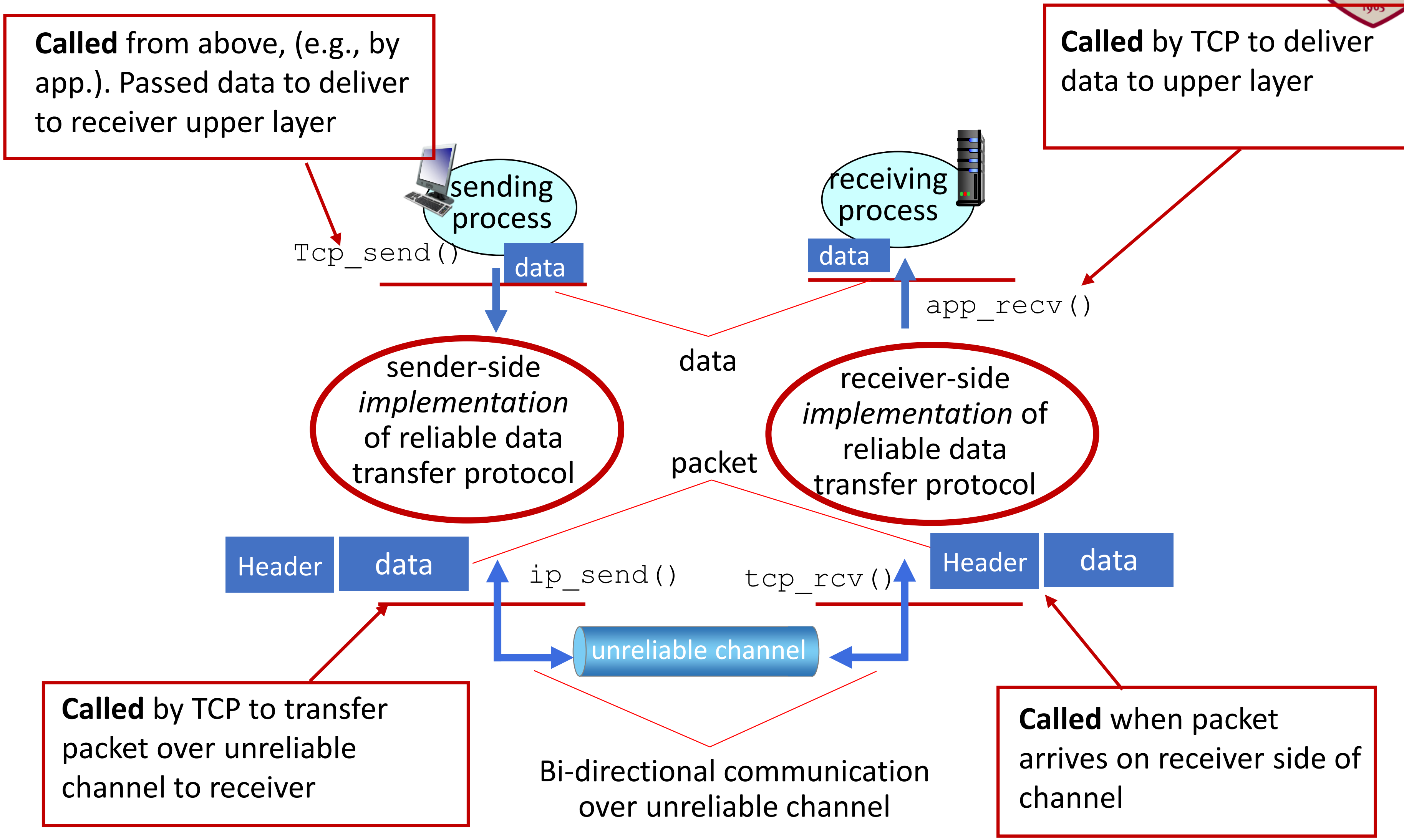


Sender, receiver do *not* know the "state" of each other, e.g., was a message received?

- unless communicated via a message

reliable service *implementation*

Reliable Transmission



Reliable Transmission Outline

- **Fundamentals of Automatic Repeat reQuest (ARQ) algorithms**
  - A family of algorithms that provide reliability through retransmission
- **ARQ algorithms (simple to complex)**
  - Stop-and-wait
  - Concurrent logical channels
  - Sliding window
    - Go-back-n
    - Selective repeat
- **Alternative: forward error correction (FEC)**

Terminology

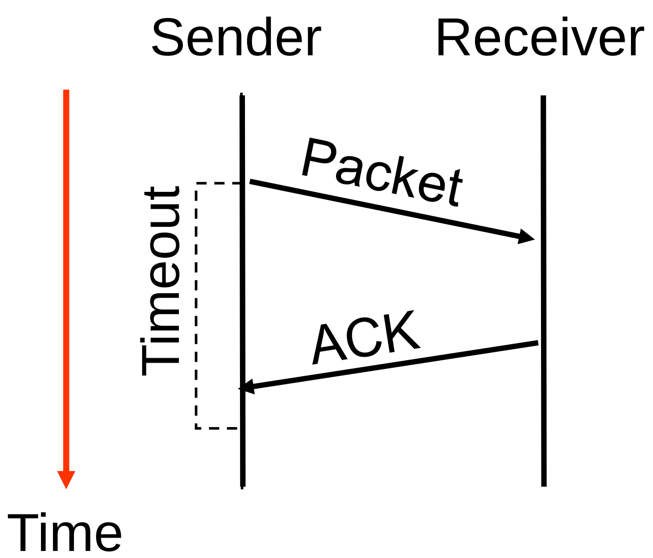
- **Acknowledgement (ACK)**
  - Receiver tells the sender when a frame is received
    - Selective acknowledgement (SACK)
      - Specifies set of frames received
    - Cumulative acknowledgement (ACK)
      - Have received specified frame and all previous
    - Negative acknowledgement (NAK)
      - Receiver refuses to accept frame now,
        e.g., when out of buffer space

Terminology

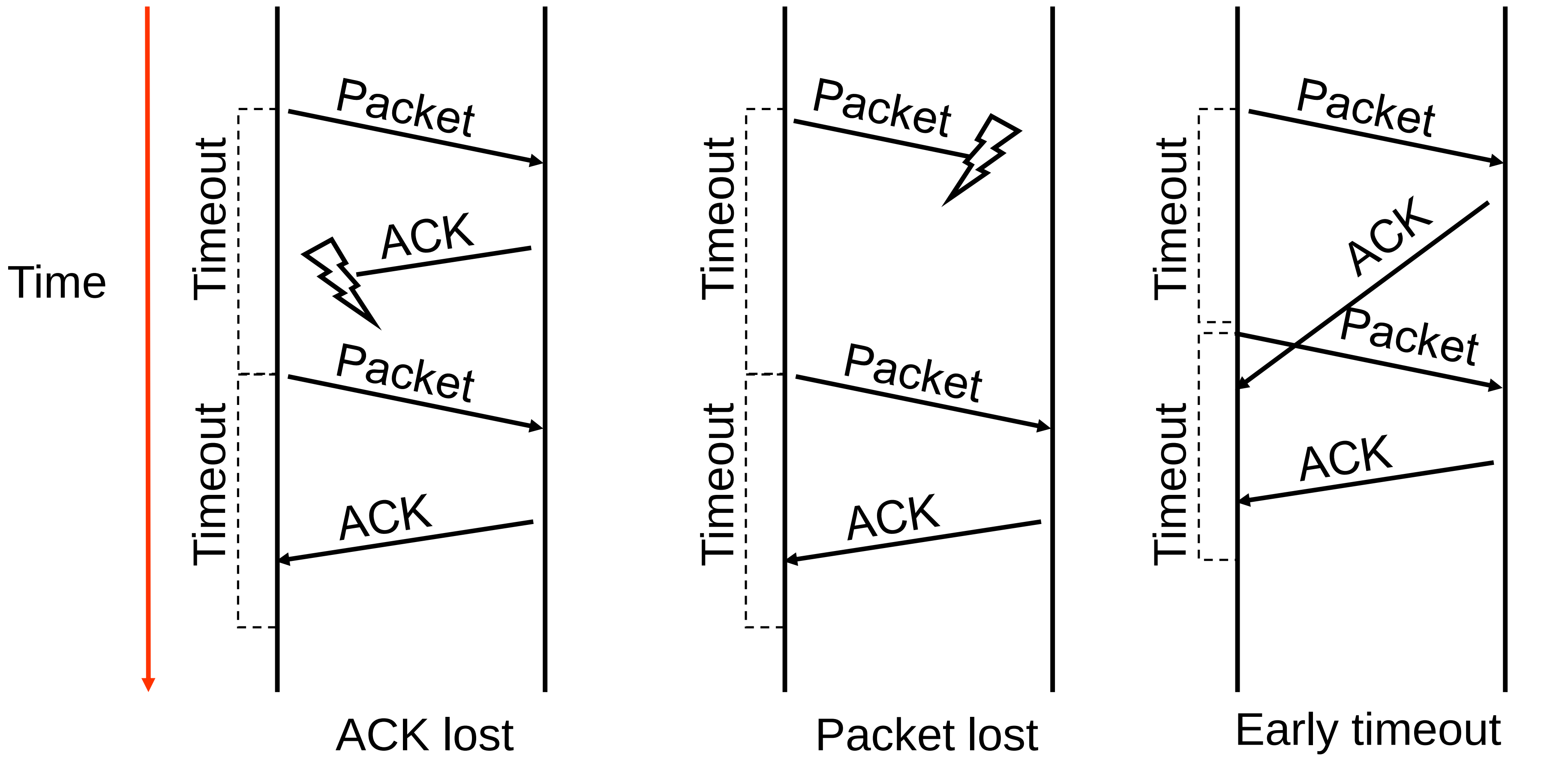
- **Timeout (TO)**
  - Sender decides the frame (or ACK) was lost
  - Sender can try again
- **ARQ also called Positive Acknowledgement with Retransmission (PAR)**

Stop and Wait

- **ARQ**
  - Receiver sends acknowledgement (ACK) when it receives packet
  - Sender waits for ACK and timeouts if it does not arrive within some time period
- **Simplest ARQ protocol**
- **Send a packet, stop and wait until ACK arrives**

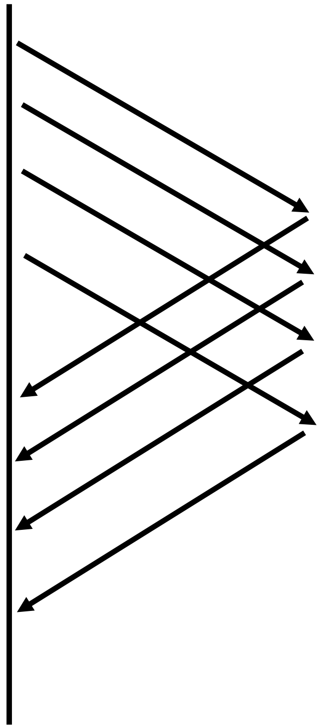


Stop and Wait



How to Keep the Pipe Full?

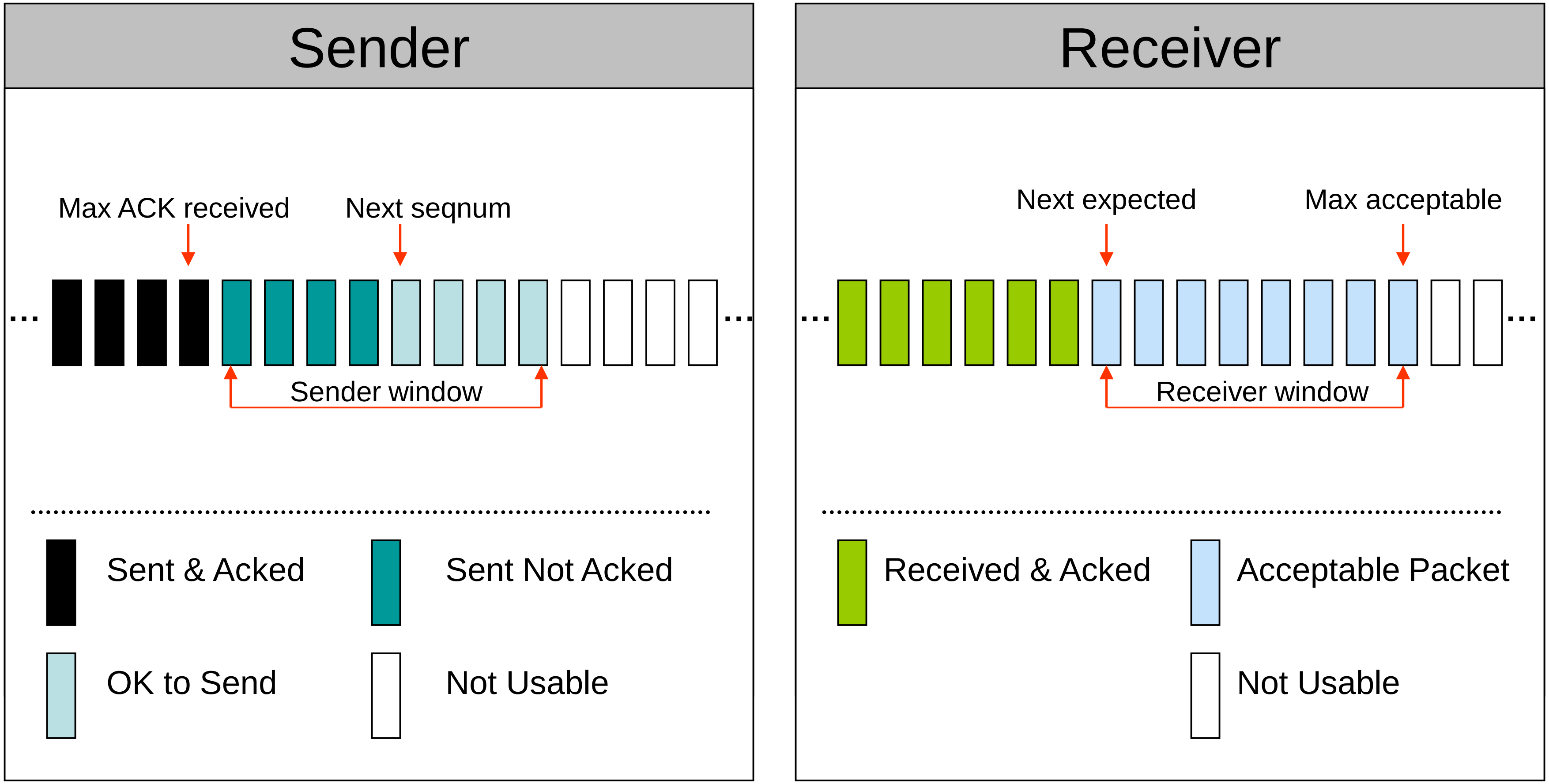
- **Send multiple packets without waiting for first to be acked**
  - Number of pkts in flight = window
- **Reliable, unordered delivery**
  - Several parallel stop & waits
  - Send new packet after each ack
  - Sender keeps list of unack'ed packets; resends after timeout
  - Receiver same as stop & wait
- **How large a window is needed?**
  - Round trip delay * bandwidth = capacity of pipe



Sliding Window

- **Reliable, ordered delivery**
- **Receiver has to hold onto a packet until all prior packets have arrived**
  - Sender must prevent buffer overflow at receiver
- **Circular buffer at sender and receiver**
  - Packets in transit $\leq$ buffer size
  - Advance when sender and receiver agree packets at beginning have been received

Sliding Window



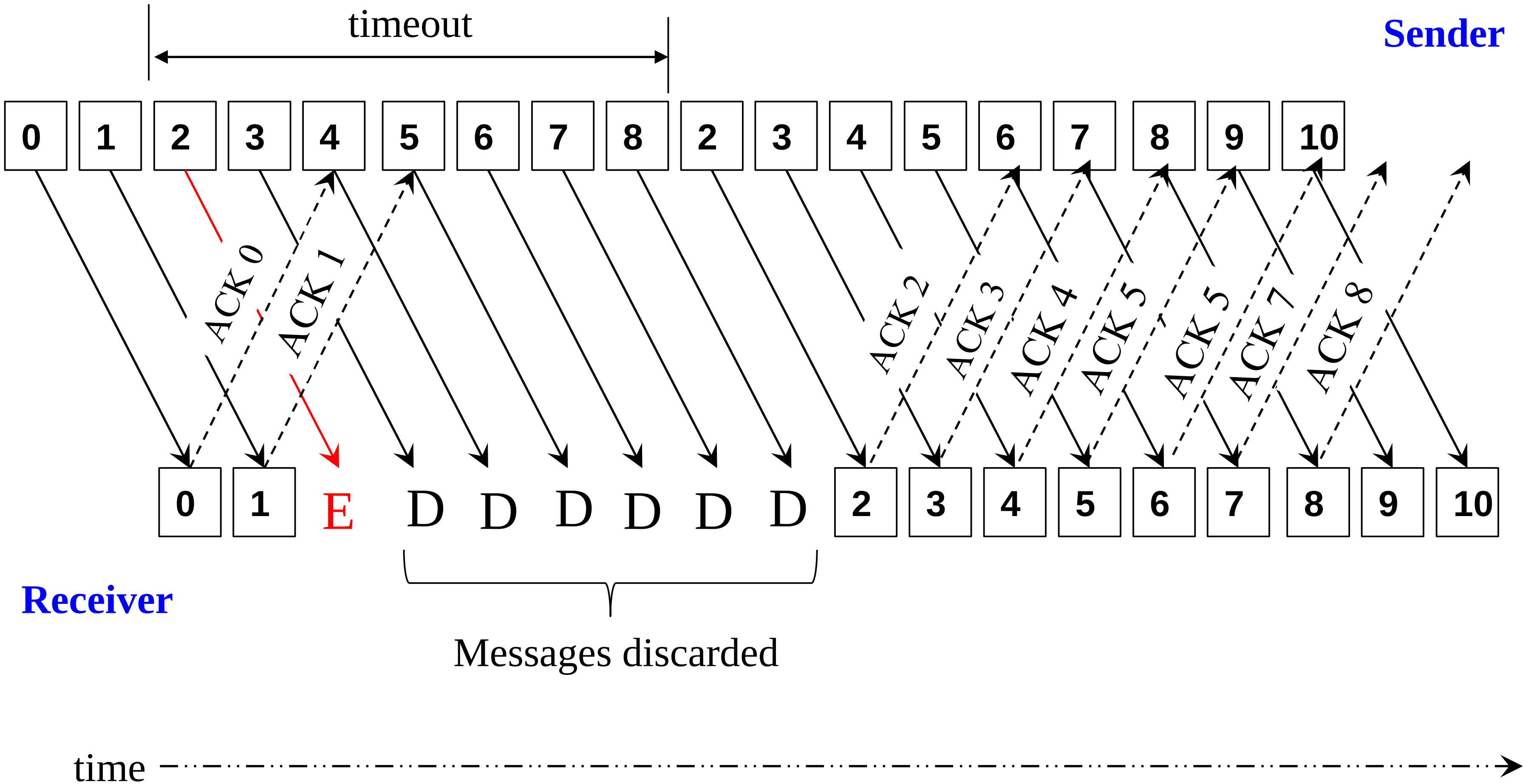
Sliding Window: Common Case

- **On reception of new ACK (i.e. ACK for something that was not acked earlier)**
  - Increase sequence of max ACK received
  - Send next packet
- **On reception of new in-order data packet (next expected)**
  - Hand packet to application
  - Send cumulative ACK – acknowledges reception of all packets up to sequence number
  - Increase sequence of max acceptable packet

Sliding Window: Loss

- **On reception of out-of-order packet**
  - Send nothing (wait for source to timeout)
  - Cumulative ACK (helps source identify loss)
- **Timeout (Go-Back-N recovery)**
  - Set timer upon transmission of packet
  - **Retransmit all unacknowledged packets**
- **Performance during loss recovery**
  - No longer have an entire window in transit
  - **Can have much more clever loss recovery**

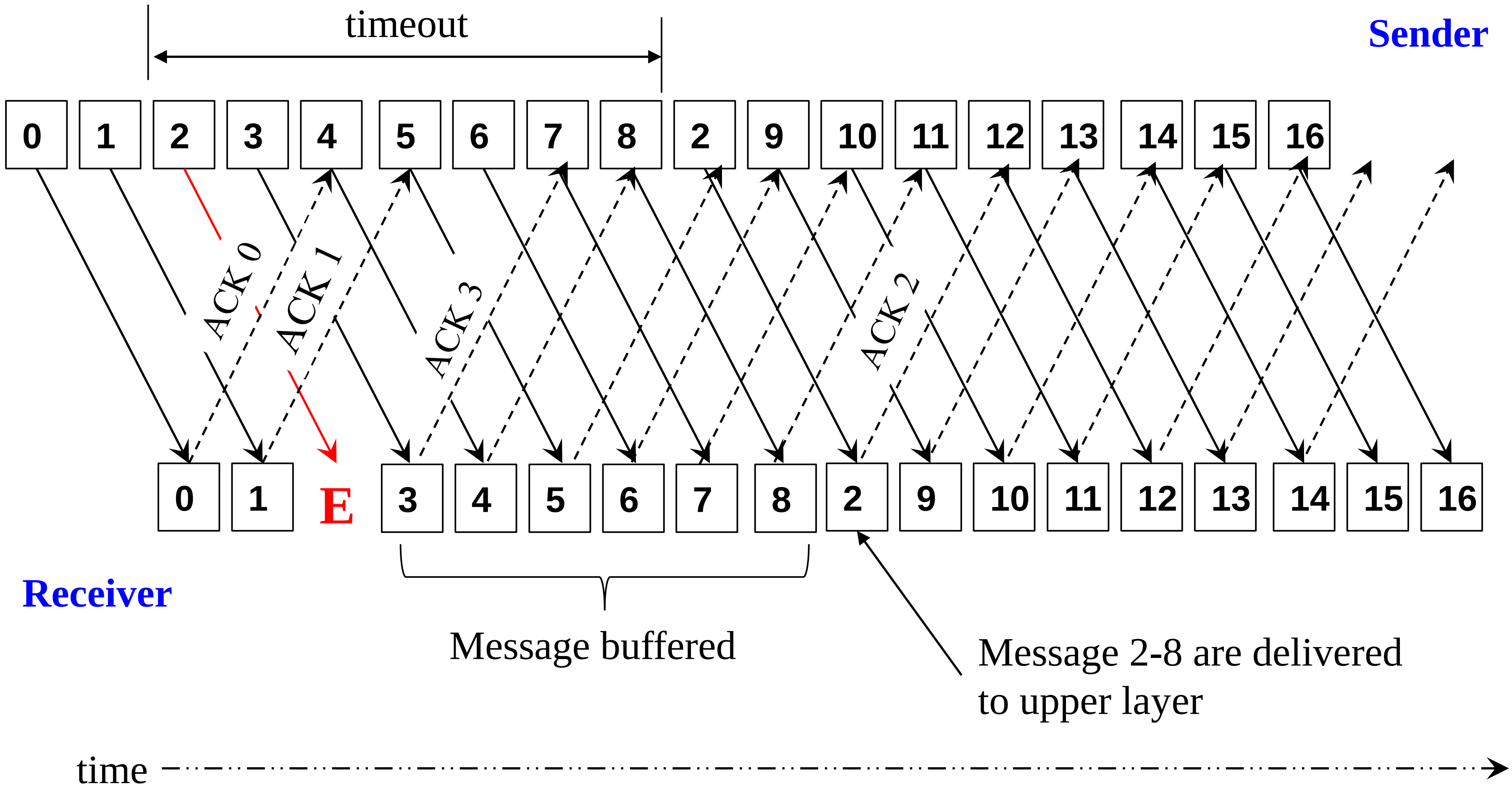
Sliding Window: Go-Back-N



Sliding Window: Selective Repeat

- **Receiver individually acknowledges all correctly received packets**
  - Buffers packets, as needed, for **eventual in-order delivery** to upper layer
- **Sender only resends packets for which ACK not received**
  - Sender timer for each unACKed packet

Sliding Window: Go-Back-N



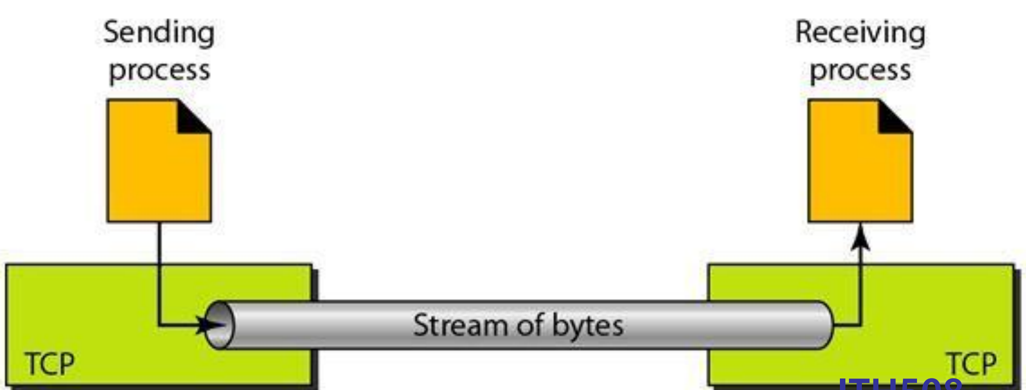
Sliding Window: Sequence Number

- **How large do sequence numbers need to be?**
  - Must be able to detect wrap-around
  - Depends on sender/receiver window size
- **E.g.**
  - Max seq = 7, send win=recv win=7
  - If pkts 0..6 are sent successfully and all acks lost
    - Receiver expects 7,0..5, sender retransmits old 0..6!!!
- **Max sequence must be ≥ send window + recv window**

TCP DYANAMICS

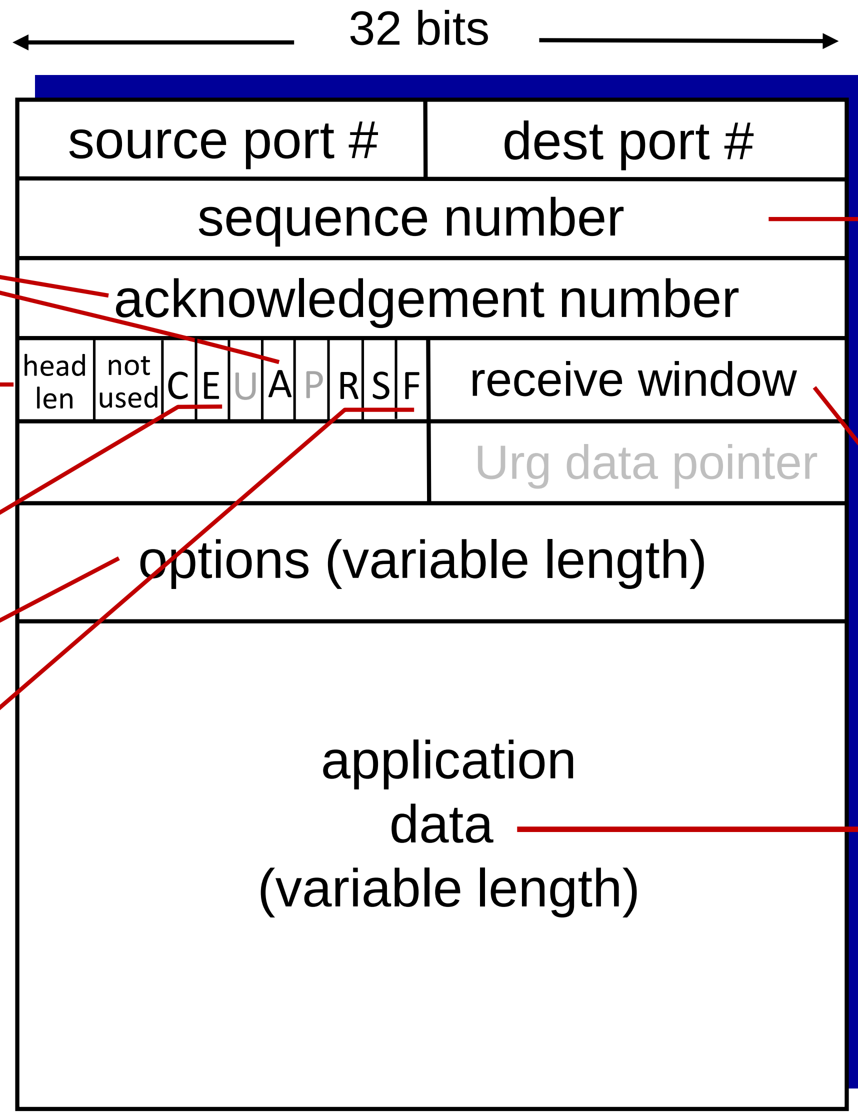
TCP: Overview

- **Point-to-point:**
  - One sender, one receiver
- **Reliable, in-order *byte steam:***
  - No “message boundaries”
- **Pipelined:**
  - TCP congestion and flow control set window size
- ***Send & receive buffers***



- **Full duplex data:**
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- **Connection-oriented:**
  - Handshaking (exchange of control messages) init’s sender, receiver state before data exchange
- **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)

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

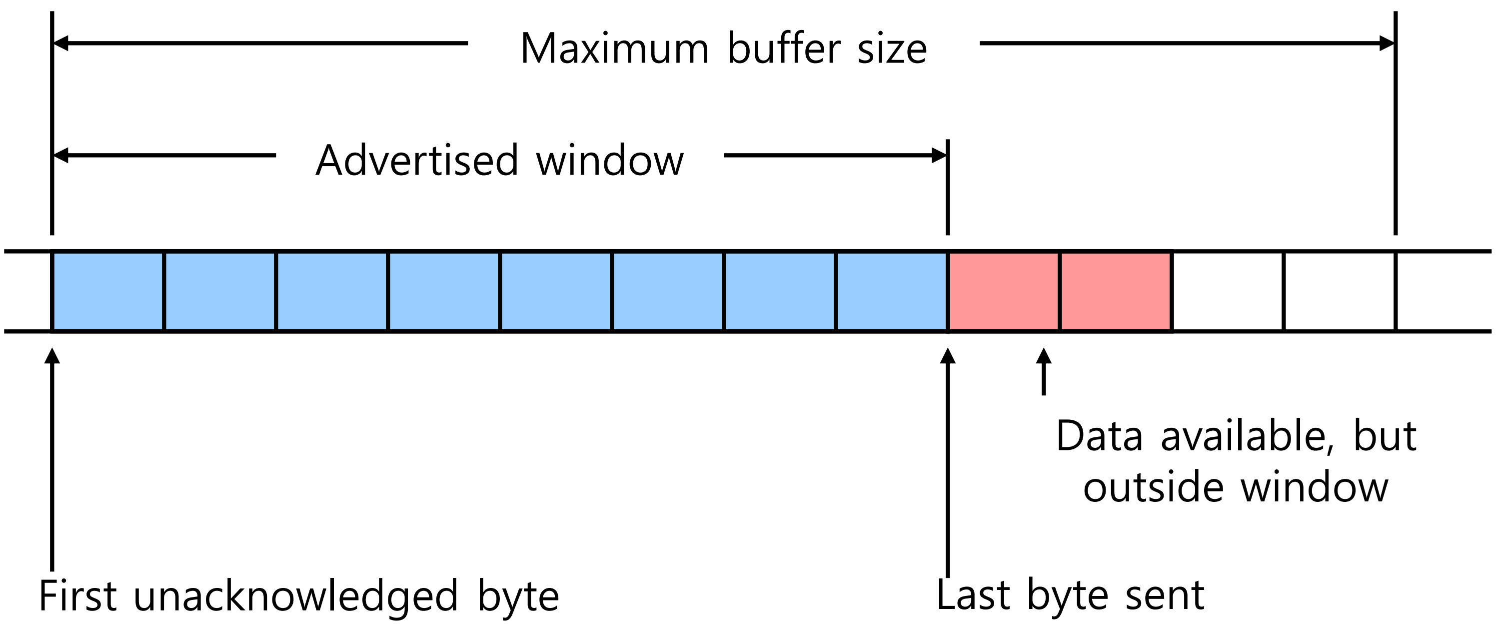
data sent by application into TCP socket

TCP Sliding Window Protocol

- **Sequence numbers**
  - Indices into byte stream
- **ACK sequence number**
  - Actually **next byte expected** as opposed to last byte received
- **Advertised window**
  - Enables dynamic receive window size
- **Receive buffers**
  - Data ready for delivery to application until requested
  - Out-of-order data out to maximum buffer capacity
- **Sender buffers**
  - Unacknowledged data
  - Unsent data out to maximum buffer capacity

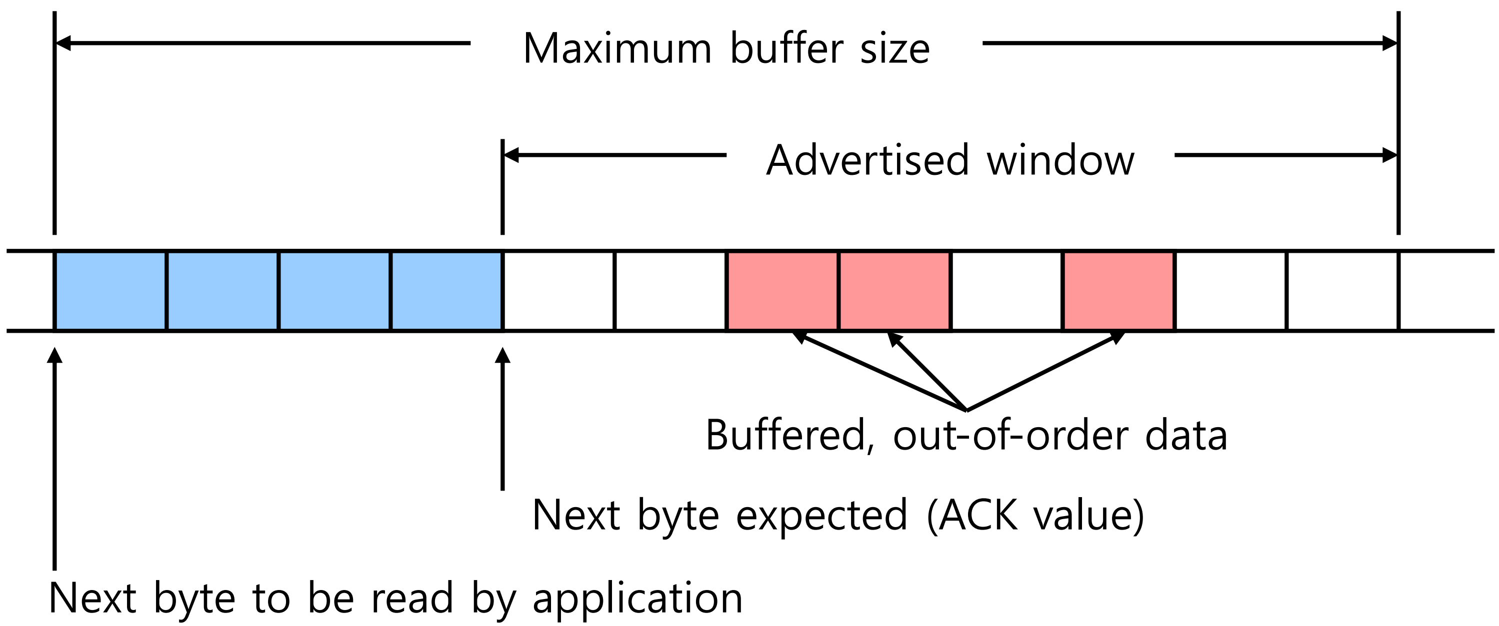
TCP Sliding Window Protocol: Sender Side

- `LastByteAcked <= LastByteSent`
- `LastByteSent <= LastByteWritten`
- **Buffer bytes between `LastByteAcked` and `LastByteWritten`**

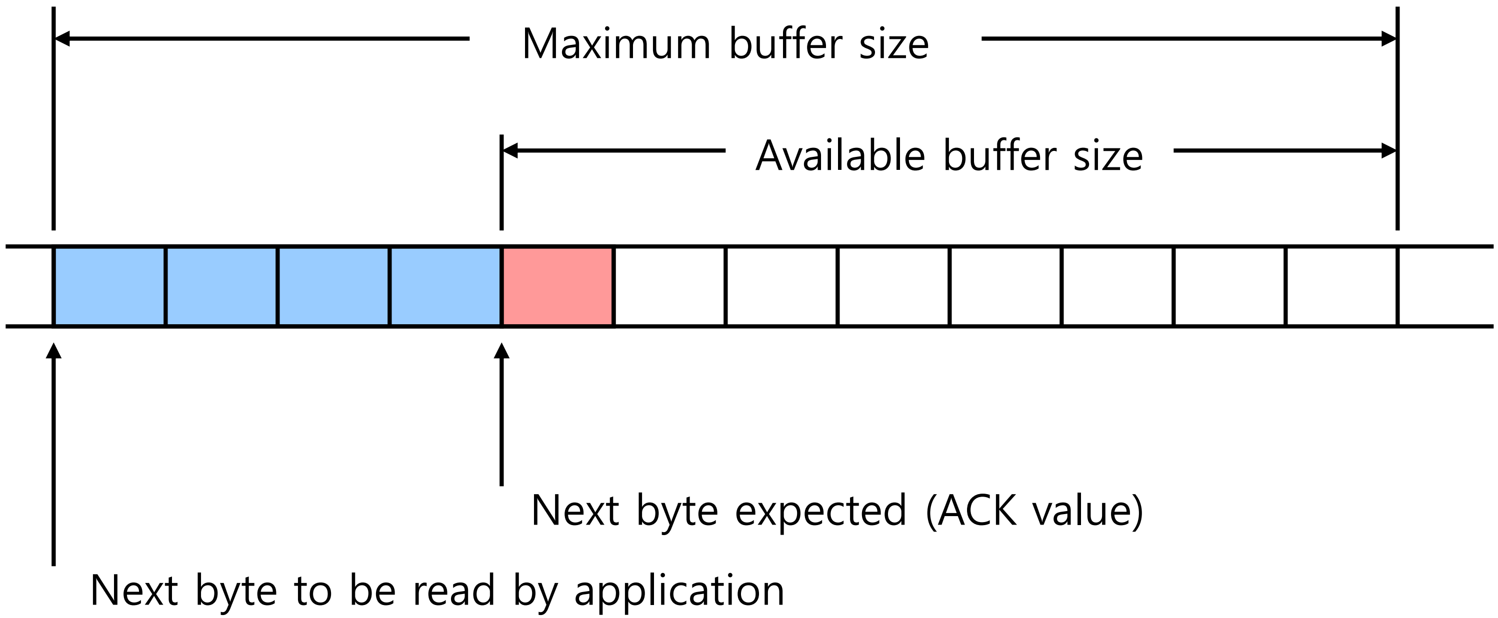


TCP Sliding Window Protocol: Receiver Side

- `LastByteRead <  NextByteExpected`
- `NextByteExpected <= LastByteRcvd + 1`
- **Buffer bytes between `NextByteRead` and `LastByteRcvd`**

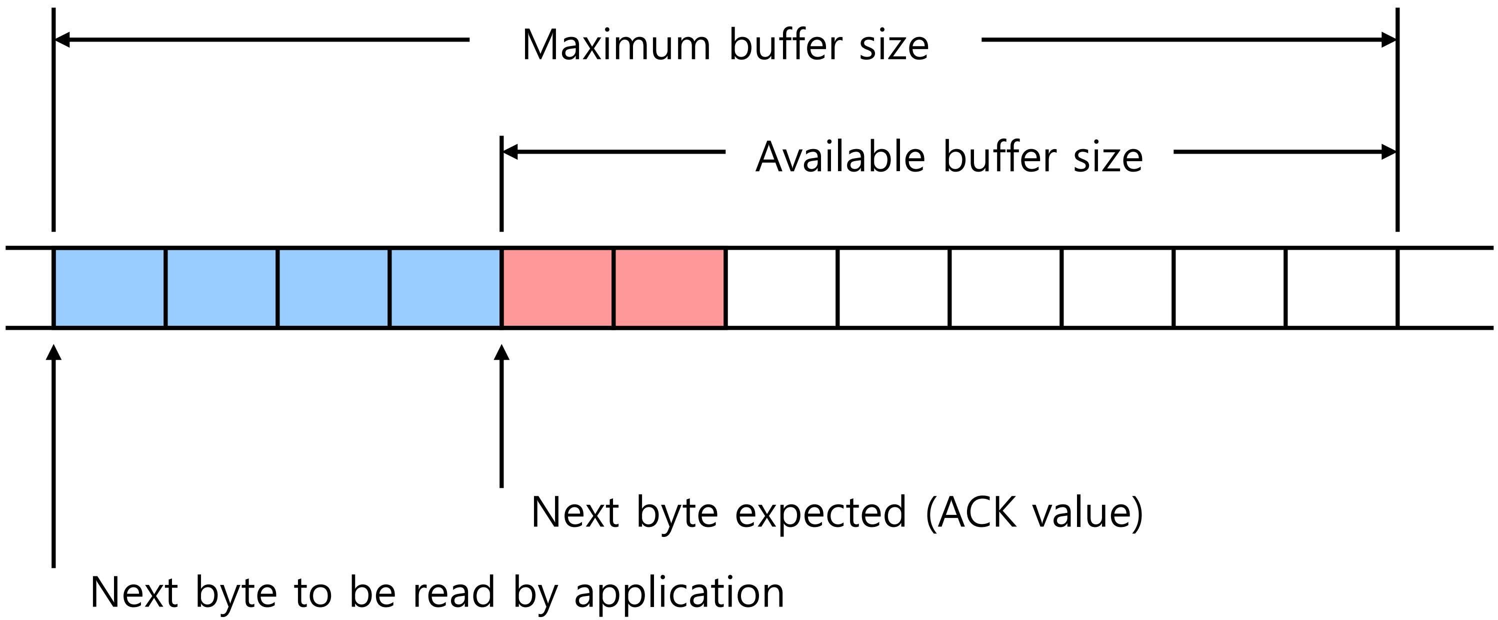


TCP ACK Generation - 1



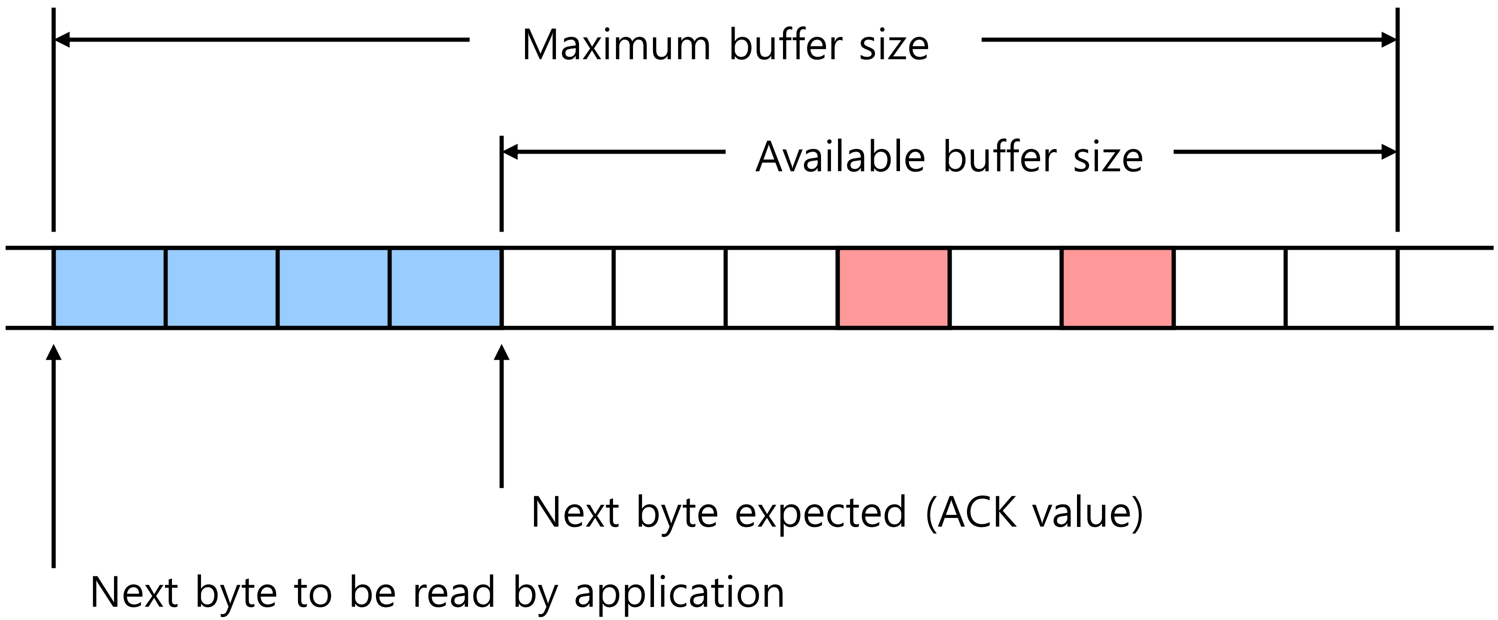
- **Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed**
  - Delayed ACK. Wait up to 500ms for next segment.

TCP ACK Generation - 2



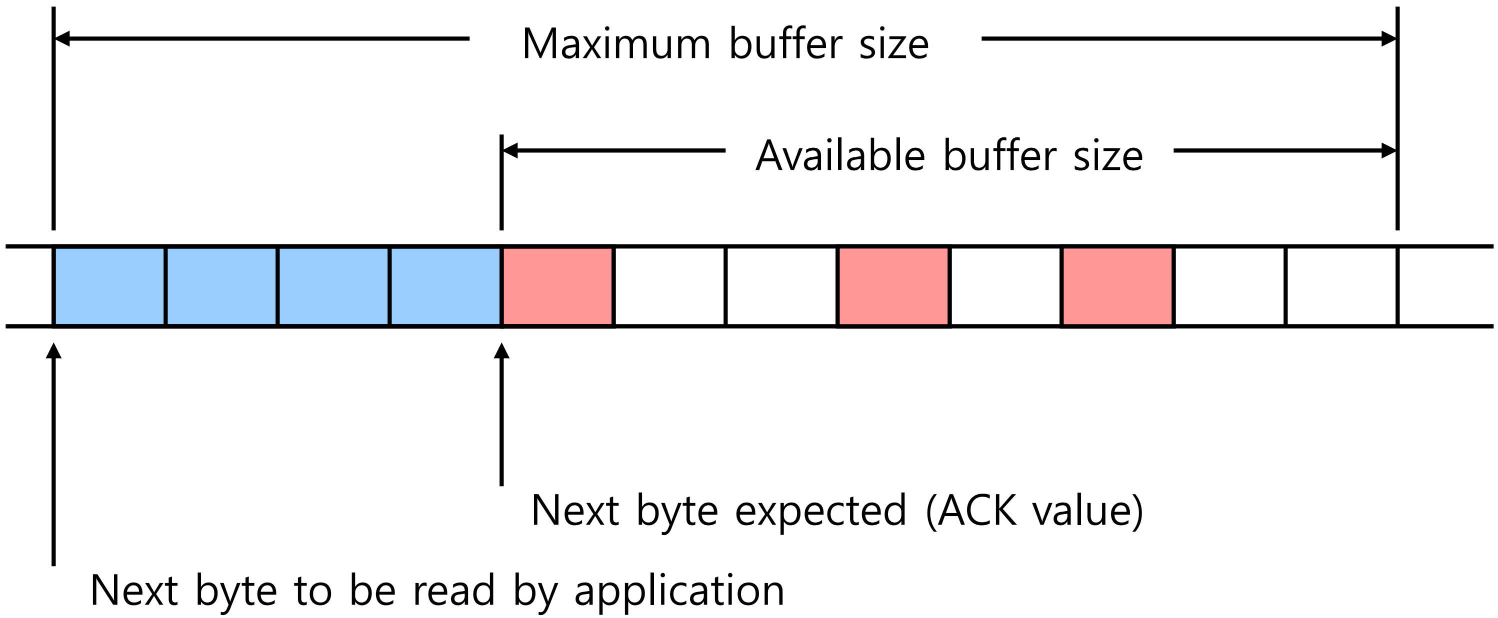
- **Arrival of in-order segment with expected seq #. One other segment has ACK pending**
  - Immediately send single cumulative ACK, ACKing both in-order segments

TCP ACK Generation - 3



- **Arrival of out-of-order segment higher-than-expect seq. # Gap detected**
  - Immediately send duplicate ACK, indicating seq. # of next expected byte

TCP ACK Generation - 4



- **Arrival of segment that partially or completely fills gap**
  - Immediate send ACK, provided that segment starts at lower end of gap

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- **Longer than RTT**
  - But RTT varies
- **Too short: premature timeout**
  - Unnecessary retransmissions
- **Too long: slow reaction to segment loss**

Q: how to estimate RTT?

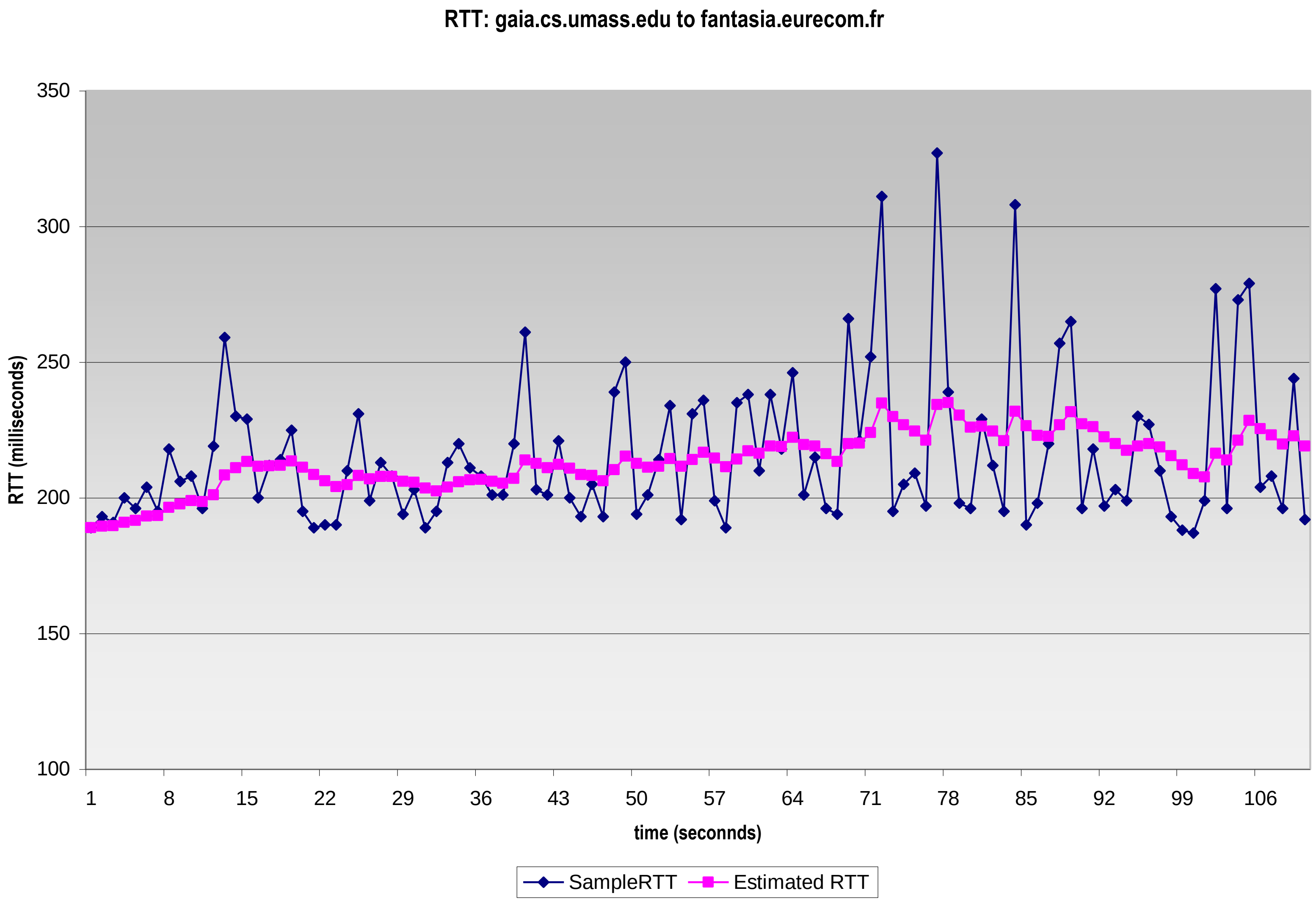
- **`SampleRTT`: measured time from segment transmission until ACK receipt**
  - Ignore retransmissions
- **`SampleRTT` will vary, want estimated RTT "smoother"**
  - Average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: α = 0.125

Example RTT Etimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimtedRTT` plus "safety margin"**
  - Large variation in **`EstimatedRTT`** -> larger safety margin
- **Variance of RTT measurement:**

  first estimate of how much **`SampleRTT`** deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|
(typically, β = 0.25)
```

Then set timeout interval:

```
Avoid unnecessary retransmissions
fast respond to changes
```

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

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`**)

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 SYN ACK segment**
- Server allocates buffers
- Specifies server initial seq. #

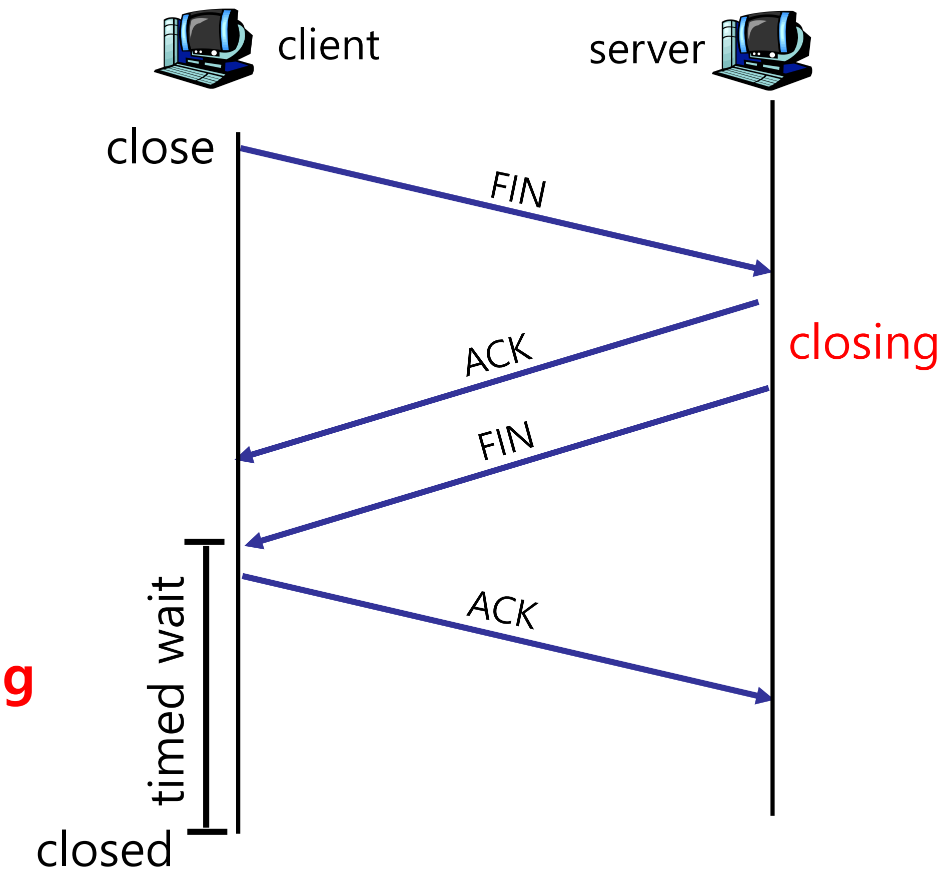
Step 3: **Client receives SYN ACK, replies with ACK segment, which may contain data**

TCP Connection Management

Closing a connection:

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

Step 2: **Server receives FIN, replies with ACK. Inform Application. Send remaining data. sends FIN.**



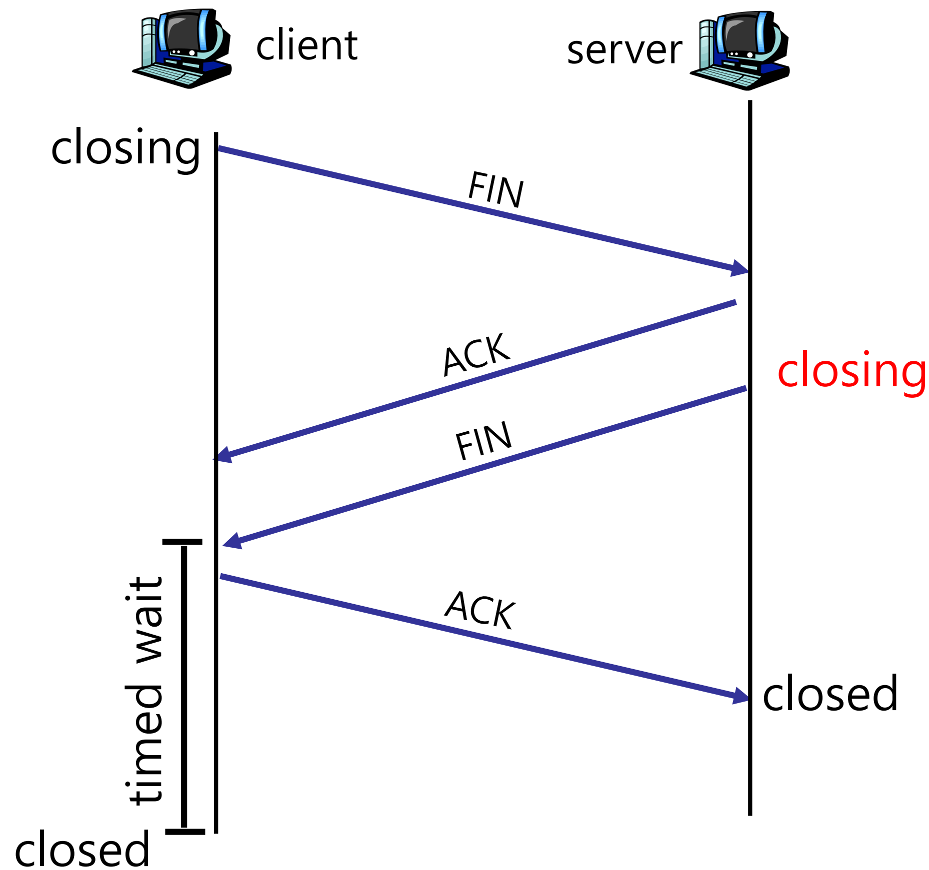
TCP Connection Management

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 Sender Events:

<u>Data received from app:</u>

- 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`

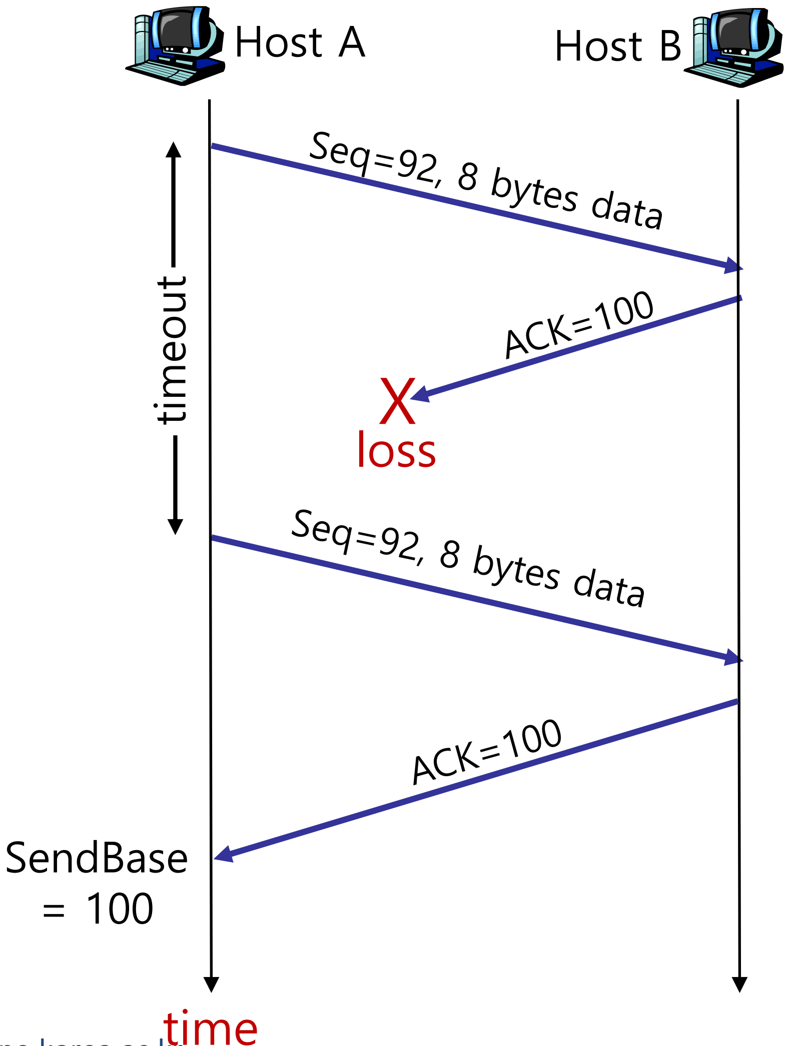
<u>Timeout:</u>

- Retransmit segment that caused timeout
- Restart timer

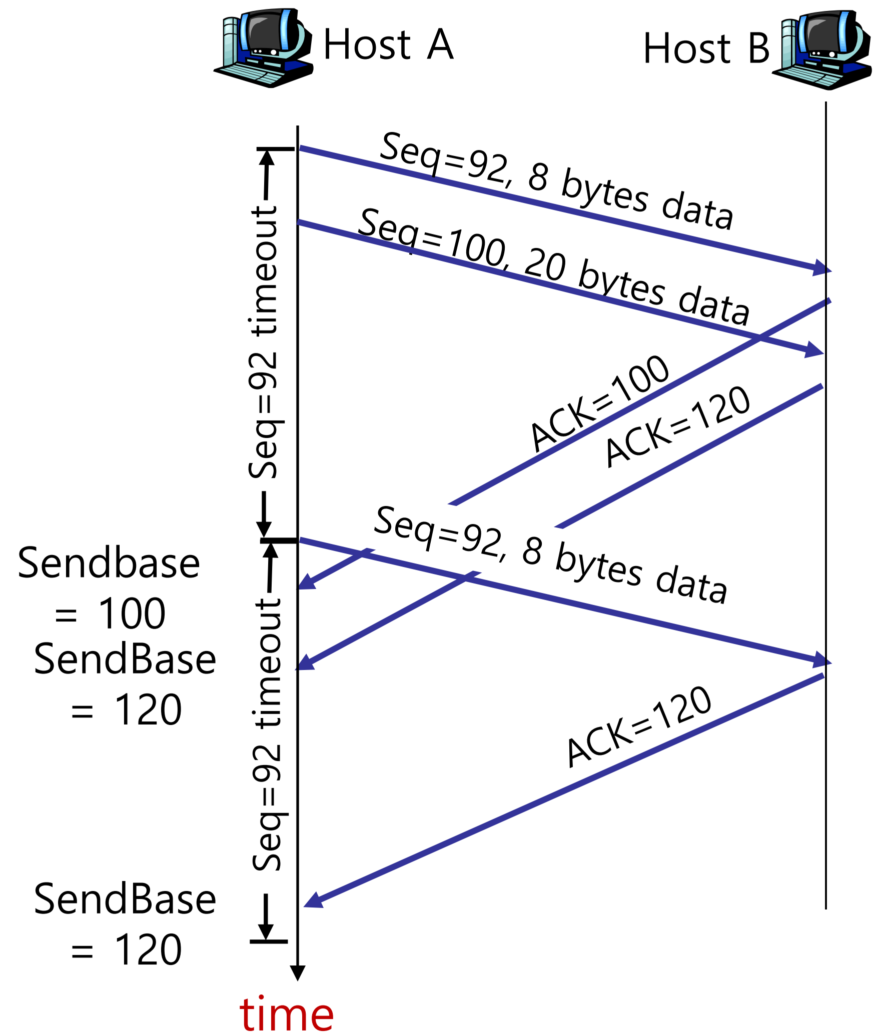
<u>Ack received:</u>

- If acknowledges previously unacked segments
  - Update what is known to be acked
  - Start timer if there are outstanding segments

TCP: Retransmission Scenarios

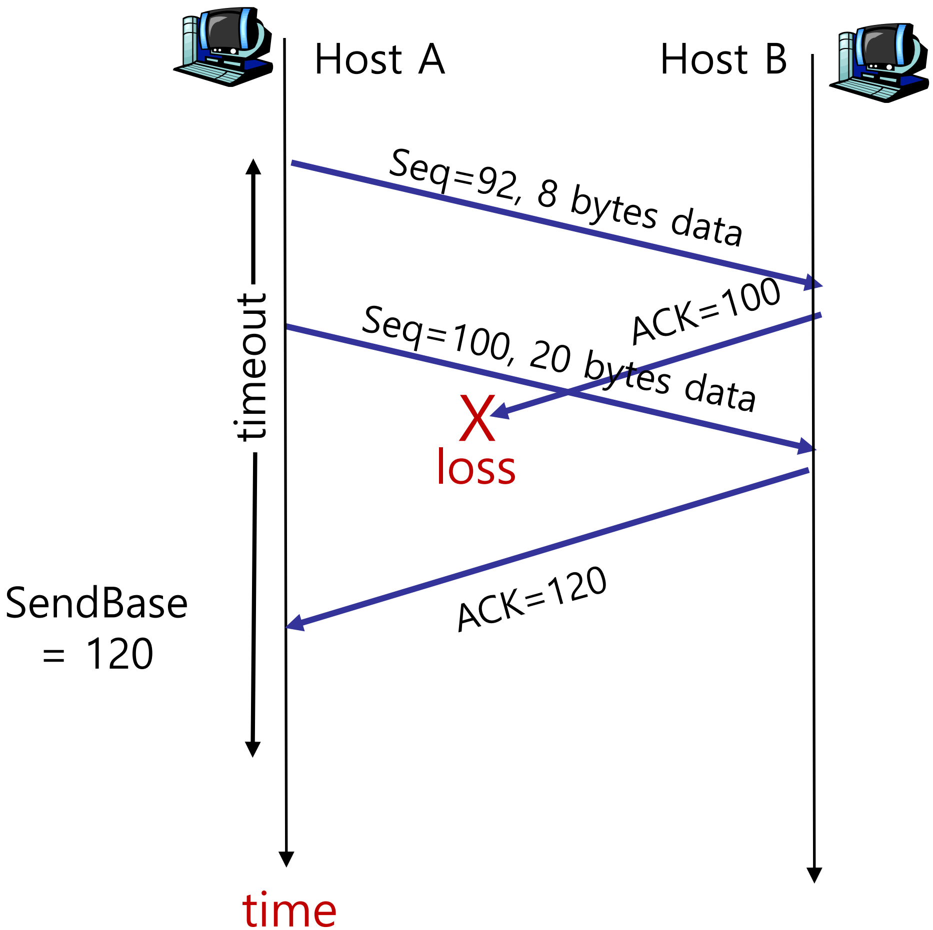


lost ACK scenario



premature timeout

TCP Retransmission Scenarios (more)



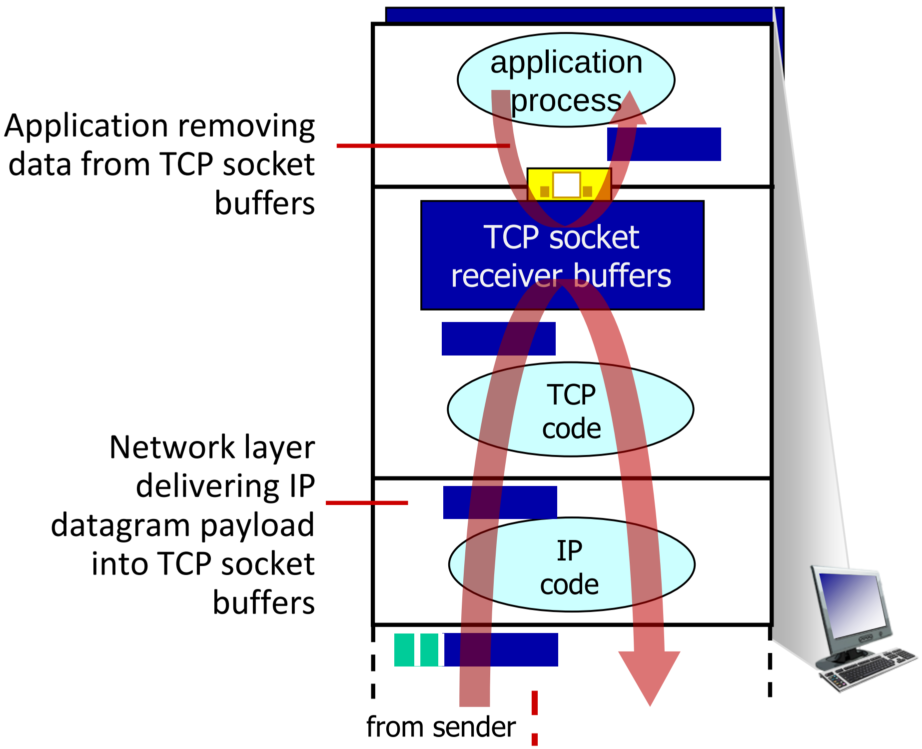
Cumulative ACK scenario

TCP Flow Control vs. TCP Congestion Control

- **Flow control**
  - Preventing senders from overrunning the capacity of the receivers
- **Congestion control**
  - Preventing too much data from being injected into the network, **causing switches or links to become overloaded**
- **TCP provides both**
  - Flow control based on advertised window
  - Fongestion control

TCP Flow Control

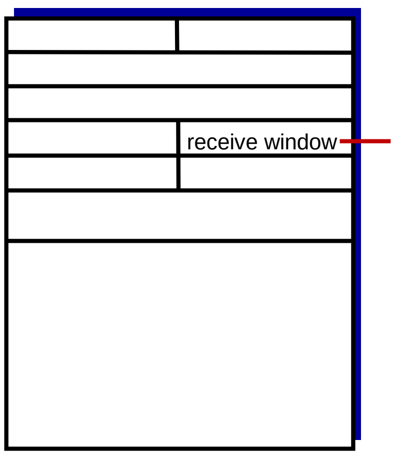
Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?



receiver protocol stack

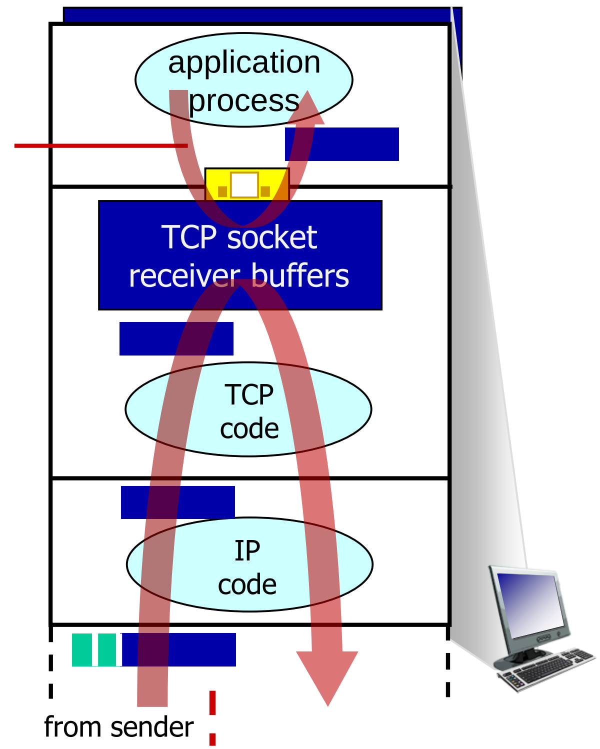
TCP Flow Control

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



flow control: # bytes receiver willing to accept

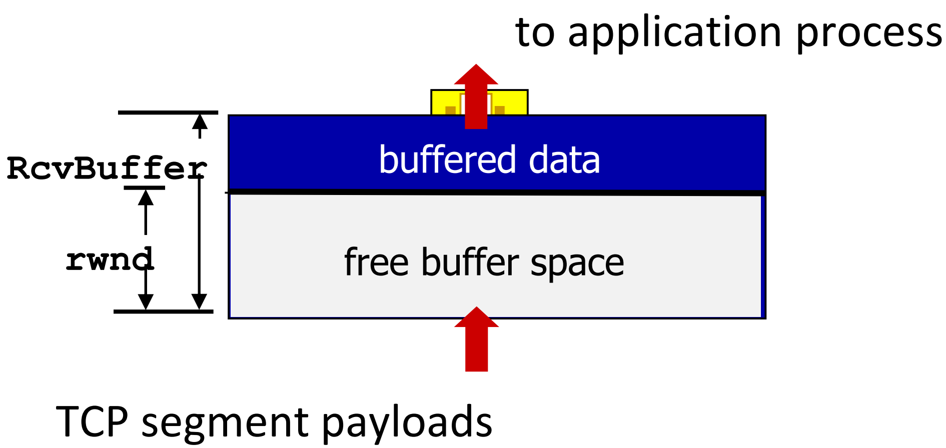
Application removing data from TCP socket buffers



receiver protocol stack

TCP Flow Control

- **Receive side of TCP connection has a receive buffer:**



TCP receiver-side buffering

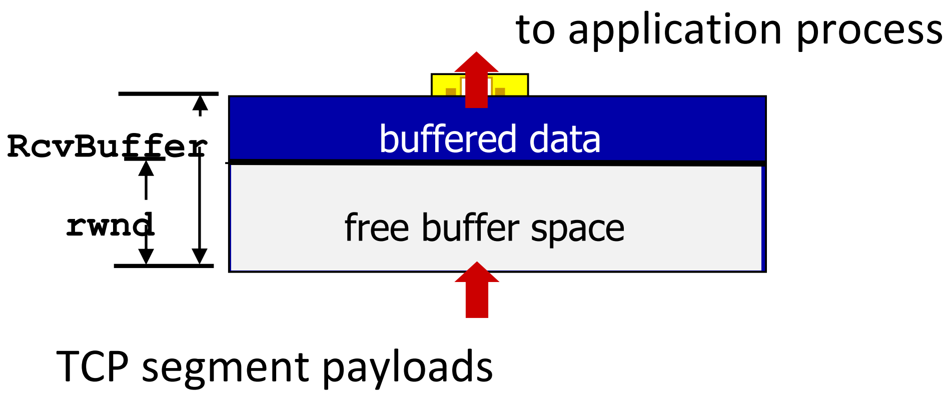
- **Application process may be slow at reading from buffer**

flow control

Sender won't overflow receiver's buffer by transmitting too much, too fast

- **Speed-matching service: matching the send rate to the receiving app's drain rate**

TCP Flow Control



TCP receiver-side buffering

(Suppose TCP receiver discards out-of-order segments)

- **Spare room in buffer**

= `RcvWindow`

= `RcvBuffer-[LastByteRcvd - LastByteRead]`

- **Receiver advertises spare room by including value of `RcvWindow` in segments**
- **Sender limits unACKed data to `RcvWindow`**
  - Guarantees receive buffer doesn't overflow

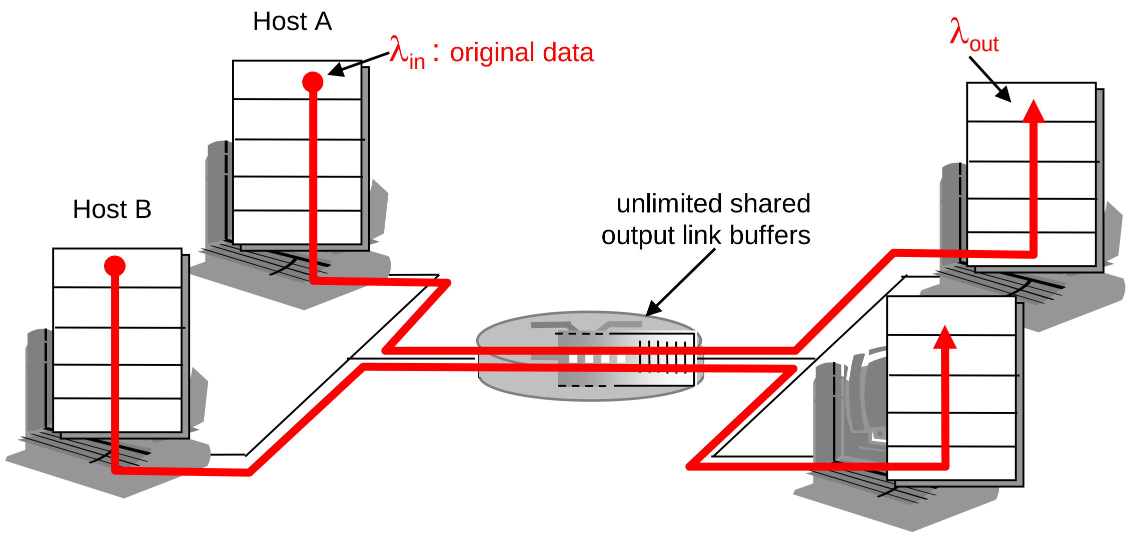
Congestion Control

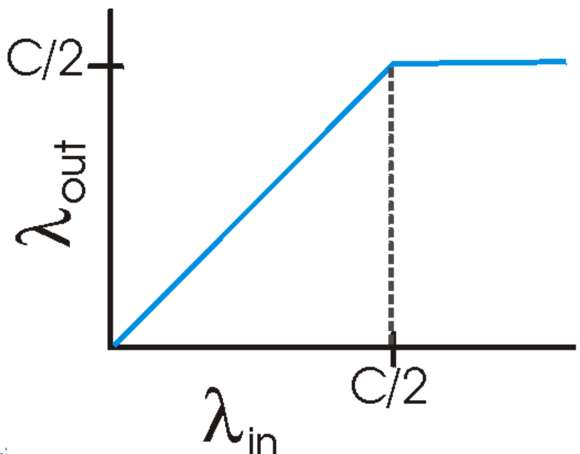
Congestion:

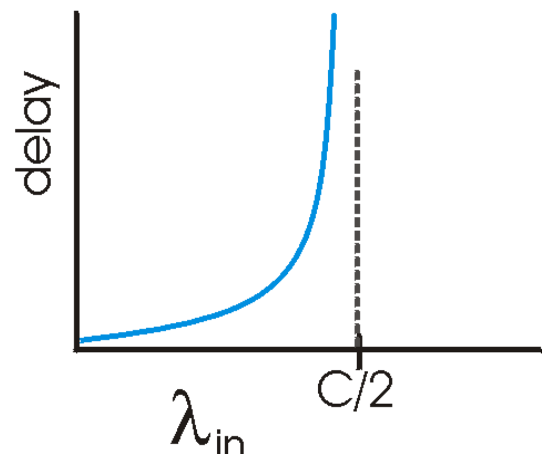
- **Informally: "Too many sources sending too much data too fast for *network* to handle"**
- **Different from flow control!**
- **Manifestations:**
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

Causes/Costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, infinite buffers
- No retransmission



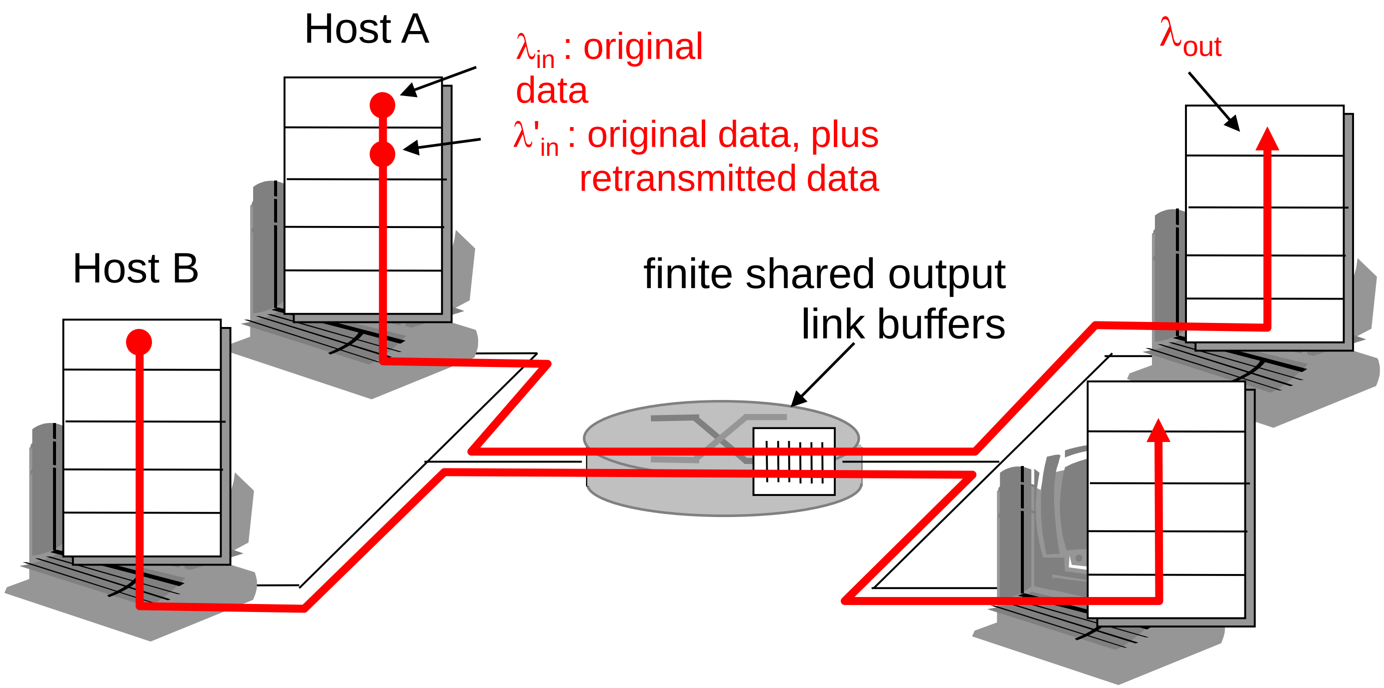




- Large delays when congested
- Maximum achievable throughput

Causes/Costs of Congestion: Scenario 2

- One router, *finite* buffers
- Sender retransmission of lost packet

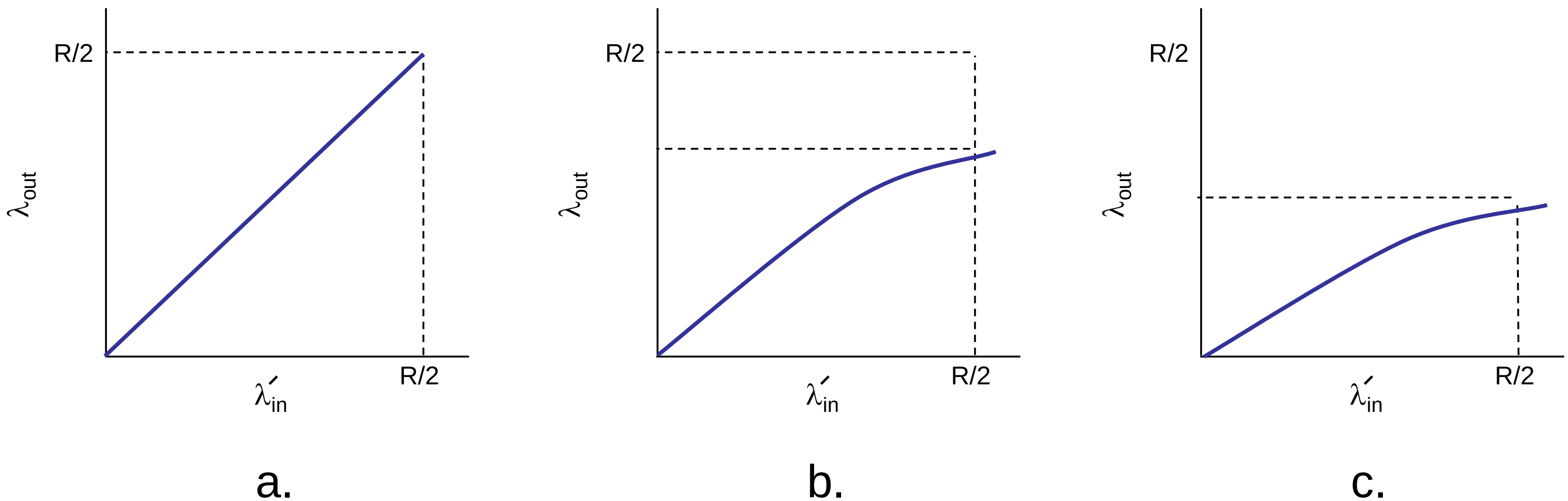


Causes/Costs of Congestion: Scenario 2

a. **Perfect case: always:** $\lambda_{in} = \lambda_{out}$ **(goodput)**

b. **Perfect retransmission only when loss:** $\lambda'_{in} > \lambda_{out}$

c. **Retransmission of delayed (not lost) packet makes** λ'_{in} **larger (than perfect case) for same** λ_{out}



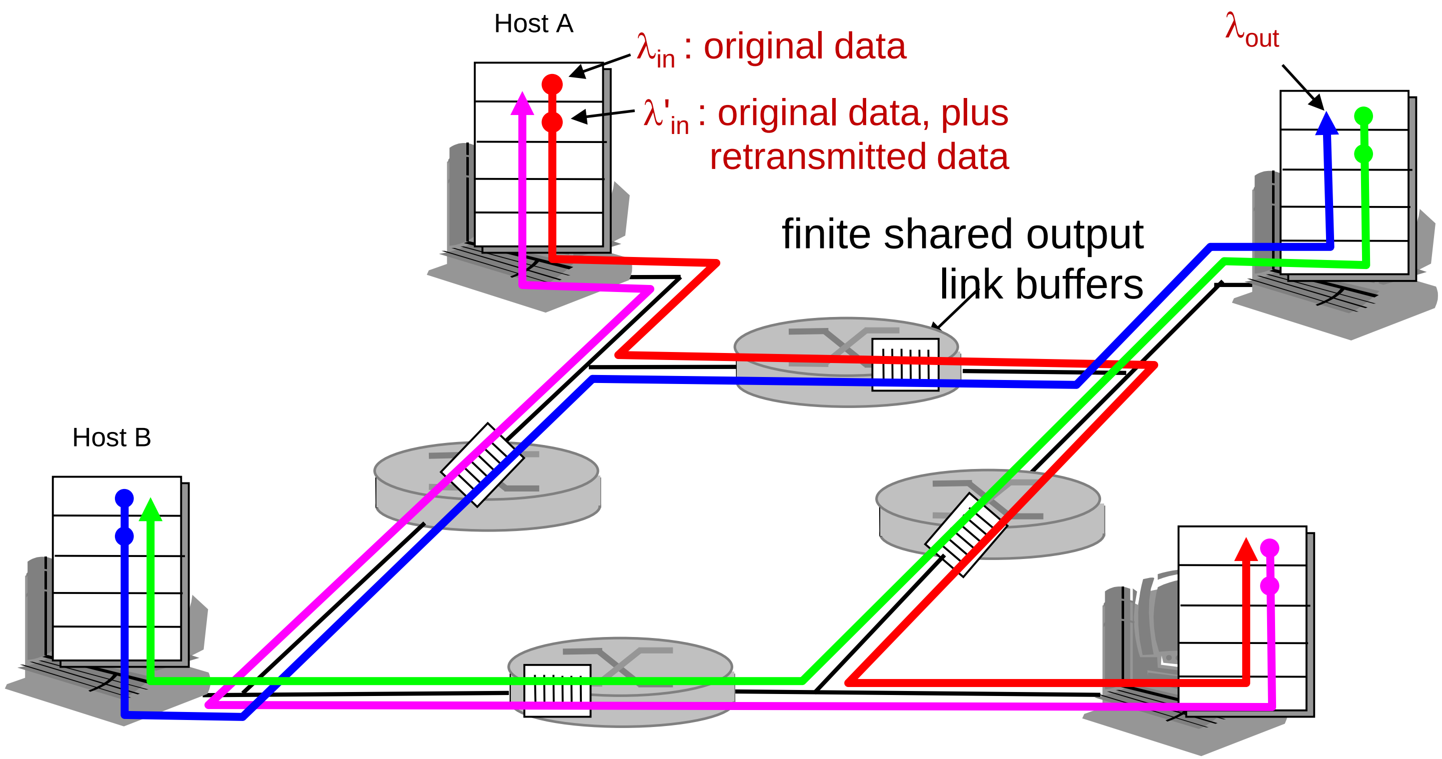
"costs" of congestion:

- More work (retransmission) for given "goodput"
- Unneeded retransmissions: link carries multiple copies of packet

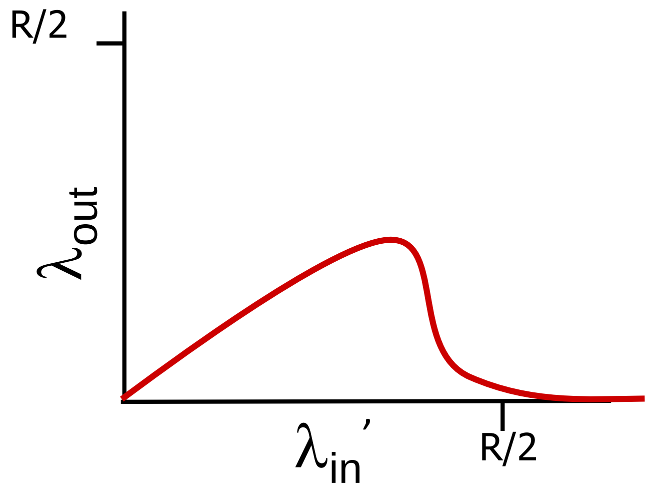
Causes/Costs of Congestion: Scenario 3

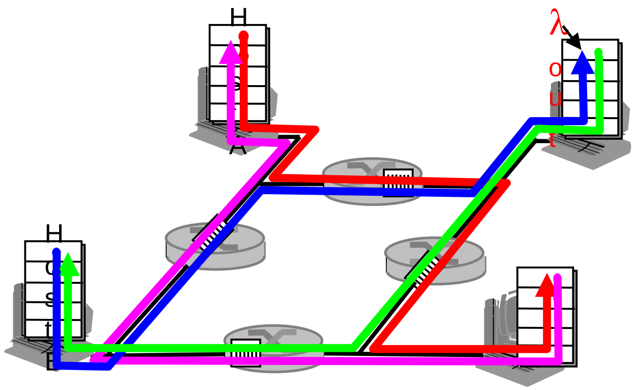
- **Four senders**
- **Multihop paths**
- **Timeout/Retransmit**

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/Costs of Congestion: Scenario 3





Another "cost" of congestion:

- When packet dropped, any upstream transmission capacity used for that packet was wasted!

TCP Congestion Control

- **Basic idea**
  - **Add notion of congestion window**
  - Effective window (for transmission) **is smaller of**
    - **Advertised window** (flow control)
    - **Congestion window** (congestion control)
  - Changes in congestion window size
    - **Slow increases** to absorb new bandwidth
    - **Quick decreases** to eliminate congestion

TCP Congestion Control

- **Specific strategy**
  - Self-clocking
    - Send data only when outstanding data is ACK'd
    - Equivalent to send window limitation mentioned
  - Initial time of TCP connection
    - **Slow Start**
    - Send data twice when outstanding data is ACK'd

TCP Congestion Control

- **Specific strategy (continued)**
  - Growth
    - Add one maximum segment size (MSS) per congestion window of data ACK'd
    - Known as additive increase
  - Decrease
    - Cut window in half **when three duplicate ACKs**
    - In practice, set window = window /2
    - Known as multiplicative decrease
  - **Additive increase, multiplicative decrease (AIMD)**
    - **Congestion avoidance + Fast recovery**

TCP Start Up Behavior

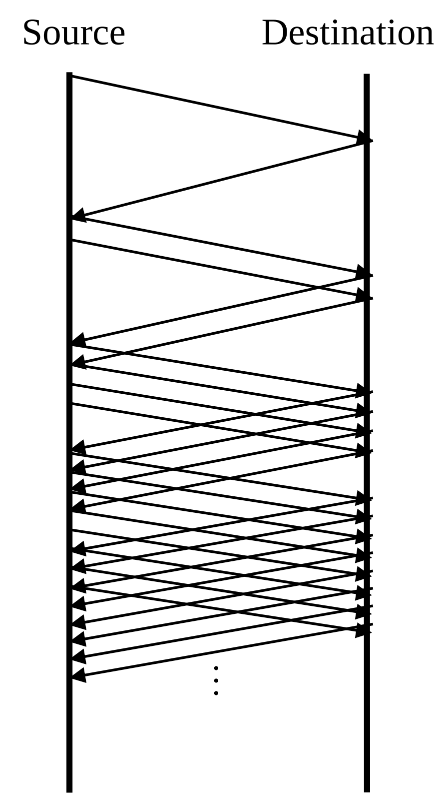
- **How should TCP start sending data?**
  - AIMD is good for channels operating at capacity
  - AIMD **can take a long time to ramp up to full capacity** from scratch
  - Use Slow Start to increase window rapidly from a cold start

TCP Start Up Behavior

- **Initialization of the congestion window**
  - ▶ Congestion window should start small
  - ▶ Avoid congestion due to new connections
  - ▶ **Start at 1 MSS**, **reset to 1 MSS** with each **timeout**
  - ▶ **Known as slow start**

Slow Start

- **Objective**
  - Determine (probe) initial available capacity
- **Idea**
  - Begin with **`CongestionWindow`** = 1 packet
  - Double **`CongestionWindow`** each RTT
    - Increment by 1 packet for each ACK
  - Continue increasing until loss
- **Result**
  - Exponential growth
  - Slower than all at once
- **Used**
  - When first starting connection
  - When connection times out



Congestion Control

- **Question**
  - How **does the source determine whether or not the network is congested**?
- **Answer**
  - **Timeout** signals packet loss
    - **Serious situation**
  - **3 duplicate ACK** means also packet loss
    - **Not that serious congestion situation**
  - **Packet loss is rarely due to transmission error** (on wired lines)
  - **Lost packet implies congestion**!

Congestion Control

- **Congestion indication**
  - Time out
  - 3 duplicate ACKs
- **How to respond to losses**
  - Reset and slow-start
  - Fast Retransmit
  - Fast Recovery
  - Selective ACK
- **Result**
  - **Congestion avoidance + Slow Start**

Fast Retransmit

- **Time-out period often relatively long:**
  - Long delay before resending lost packet
- **Detect lost segments via duplicate ACKs.**
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.

- **If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:**
  - **Fast retransmit**: resend segment before timer expires

Fast Recovery

- **Fast recovery**
  - When **fast retransmission occurs, skip slow start**
  - **Congestion window becomes 1/2 previous**
  - **Start additive increase immediately**

Additive Increase/Multiplicative Decrease

- **Objective**
  - Adjust to changes in **available capacity** (not maximal)
- **Tools**
  - React to observance of congestion (time out, 3 duplicate ACKs)
  - Probe channel to detect more resources
- **Observation**
  - On notice of congestion
    - Decreasing too slowly will not be reactive enough
    - Sometimes, need to reset TCP congestion dynamics
  - On probe of network
    - Increasing too quickly will overshoot limits

Additive Increase/Multiplicative Decrease

- **New TCP state variable**
  - **`CongestionWindow`**
    - Similar to **`AdvertisedWindow`** for flow control
  - Limits how much data source can have in transit
    - **`MaxWin = MIN(CongestionWindow, AdvertisedWindow)`**
    - **`EffWin = MaxWin - (LastByteSent - LastByteAcked)`**
    - TCP can send no faster then the slowest component, network or destination
- **Idea**
  - Increase **`CongestionWindow`** when congestion goes down
  - Decrease **`CongestionWindow`** when congestion goes up

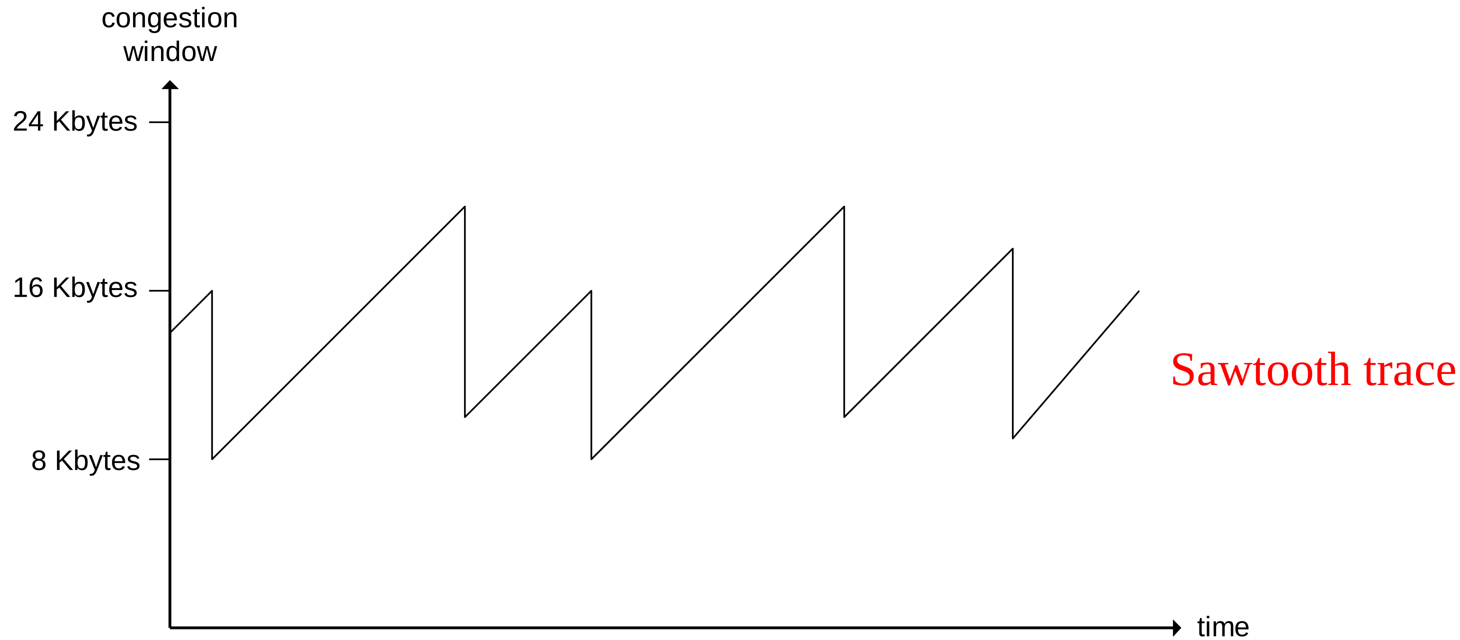
Additive Increase/Multiplicative Decrease

- **Algorithm**
  - Increment CongestionWindow by **one packet** per RTT
    - Linear increase
  - Divide CongestionWindow by two whenever a congestion occurs
    - Multiplicative decrease

Additive Increase/Multiplicative Decrease

Multiplicative decrease: cut `CongWin` in half after loss event

Additive increase: increase **CongWin** by 1 MSS every RTT in the absence of loss events: *probing*



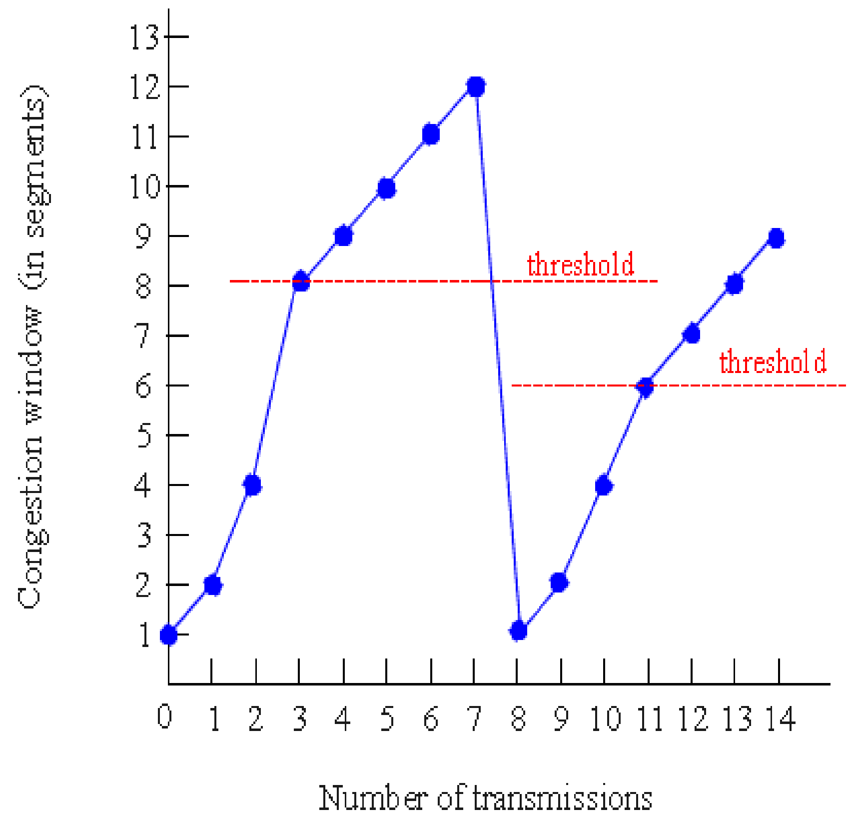
Long-lived TCP connection

Interaction between Slow Start and AIMD

- **Maintain threshold window size**
- **Use multiplicative increase**
  - When congestion window smaller than threshold
  - **Double window** for each window ACK'd
- **Threshold value**
  - **Initially set to maximum window size**
  - **Set to 1/2 of current window on timeout**
- **In practice, increase congestion window by one MSS for each ACK of new data (or N bytes for N bytes)**

Interaction between Slow Start and AIMD

- **How to control the exponential increase?**
  - Target window size is **`ssthresh`**
  - Estimate network capacity
  - When **`CongestionWindow`** reaches **`ssthresh`**
    - Switches to additive increase
- **Example**
  - Initial values
    - **`ssthresh`** = 8
    - `CongestionWindow` = 1
  - Loss after transmission 7
    - `CongestionWindow` currently 12
    - Set **`ssthresh`** = CongestionWindow/2
    - Set `CongestionWindow` = 1



TCP Congestion Control Mechanism

- **End-end control (no network assistance)**
- **Sender limits transmission:**

  `LastByteSent-LastByteAcked ≤ CongWin`

- **Roughly,**

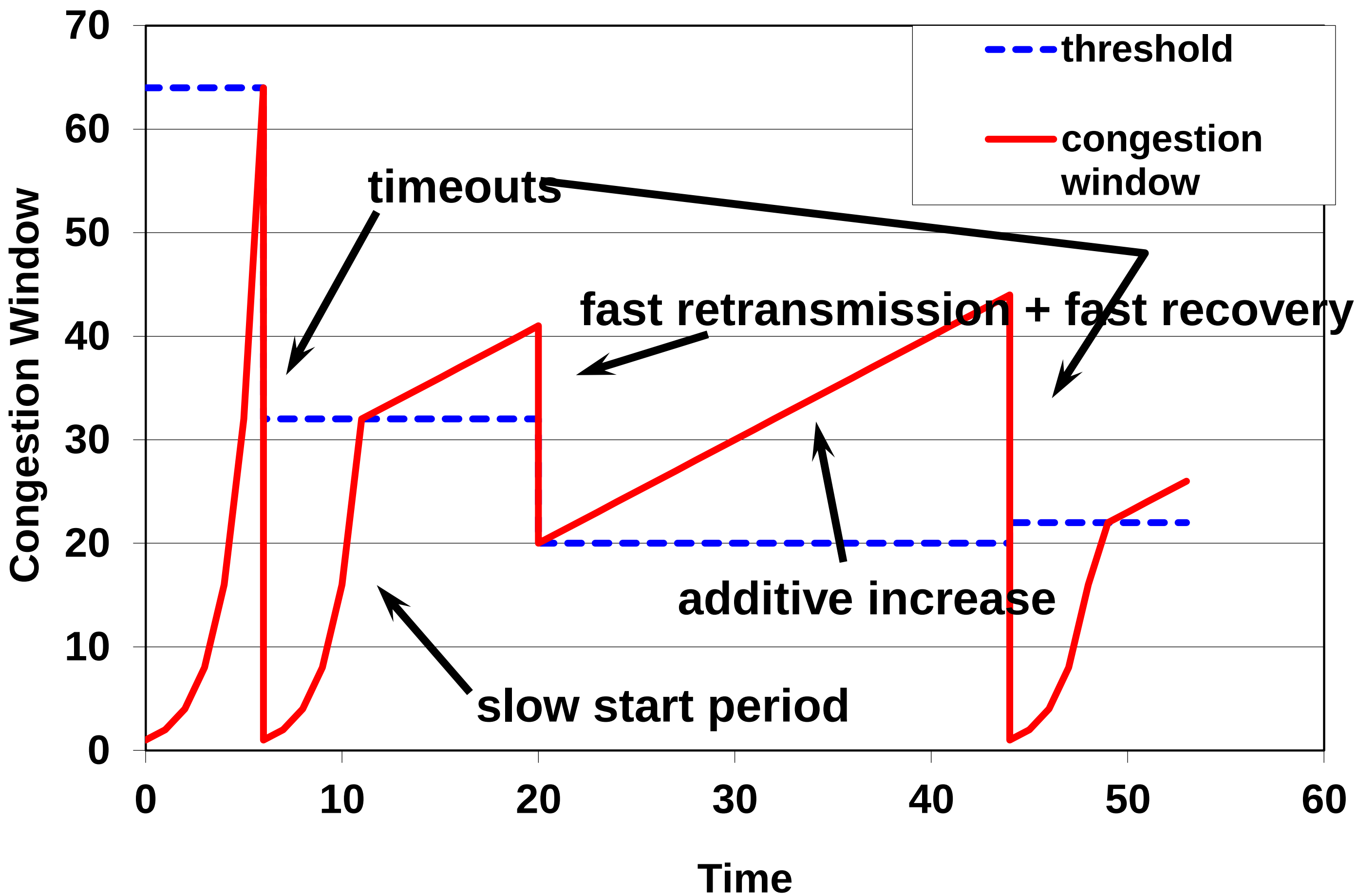
$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **`CongWin` is dynamic, function of perceived network congestion**

Main mechanisms:

- **Slow start**
- **AIMD**
- Conservative after timeout events
- Congestion window (cwnd)-based
- Congestion indication

TCP Congestion Window Trace



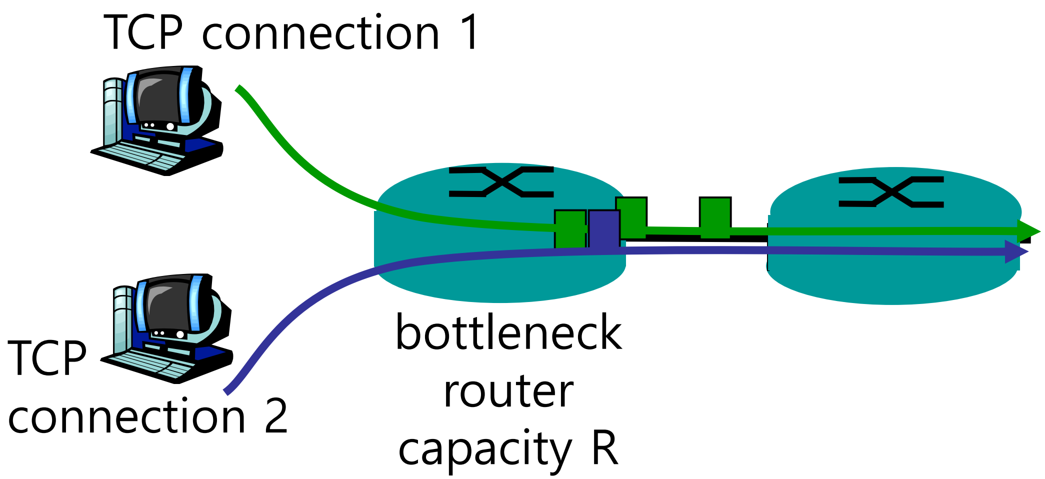
TCP Variants

- **TCP Tahoe**
  - Use 3 duplicate ACKs as well as Timeout
  - Fast Retransmit
- **TCP Reno**
  - Fast Retransmit
  - Fast Recovery
- **TCP New Reno**
  - Fast Retransmit
  - Fast Recovery
  - Partially ACK
- **TCP SACK**
  - SACK permitted
  - TCP reno
- **TCP Cubic**
  - It increases W as a function of the cube of the distance between the current time and when TCP window size will reach the maximum window
    - larger increases when further away from the maximum window
    - smaller increases (cautious) when nearer the maximum window

TCP FAIRNESS

TCP Fairness

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



TCP Fairness

Two competing sessions:

- **Additive increase gives slope of 1, as throughout increases**
- **multiplicative decrease decreases throughput proportionally**