University of Nottingham
UK | CHINA | MALAYSIA

COMP1047: Systems and Architecture

Dr. Fazl Ullah (Khan)

AY2023-24, Spring Semester
Week 8
Computer Networks: Protocols in TCP/IP Suite

Introduction

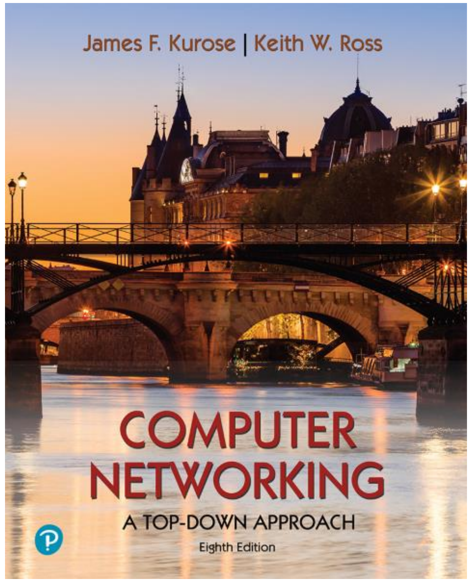
- Most of the slides are based on the Books

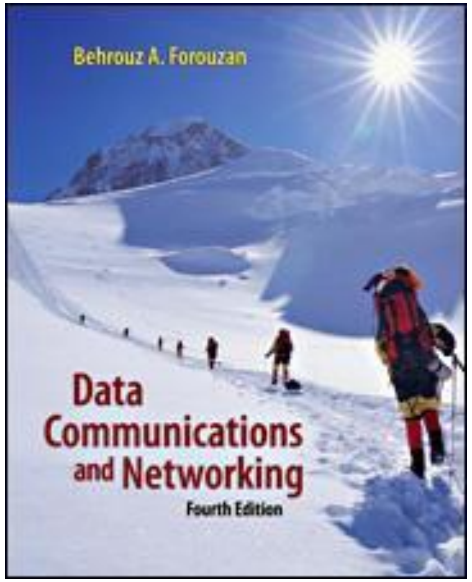
1. Computer Networking: A Top-Down Approach

8th edition by Jim Kurose, Keith Ross

and

2. Dr. Shun Yan Cheung Home Page Emory College of Arts and Sciences





Overview-Protocols in the TCP/IP Suite

Learning Outcomes:

- Understand principles behind transport layer services
- Understand the transport layer protocols
  - UDP: connectionless protocol
  - TCP: connection-oriented protocol

Overview/roadmap:

- Transport-layer services
  - Multiplexing and demultiplexing
- Connectionless transport: UDP
- Connection-oriented transport: TCP

Recap from the Last Week

Last Week, we discussed

- OSI model by the ISO
- Why layered architecture is needed
- OSI model: layers in the OSI Model and their functions
- Protocols at each layer of the OSI
- Functions at each layer provide
- OSI vs. TCP/IP Model
- Processes
- Ports
- Socket

Any question in previous lecture?

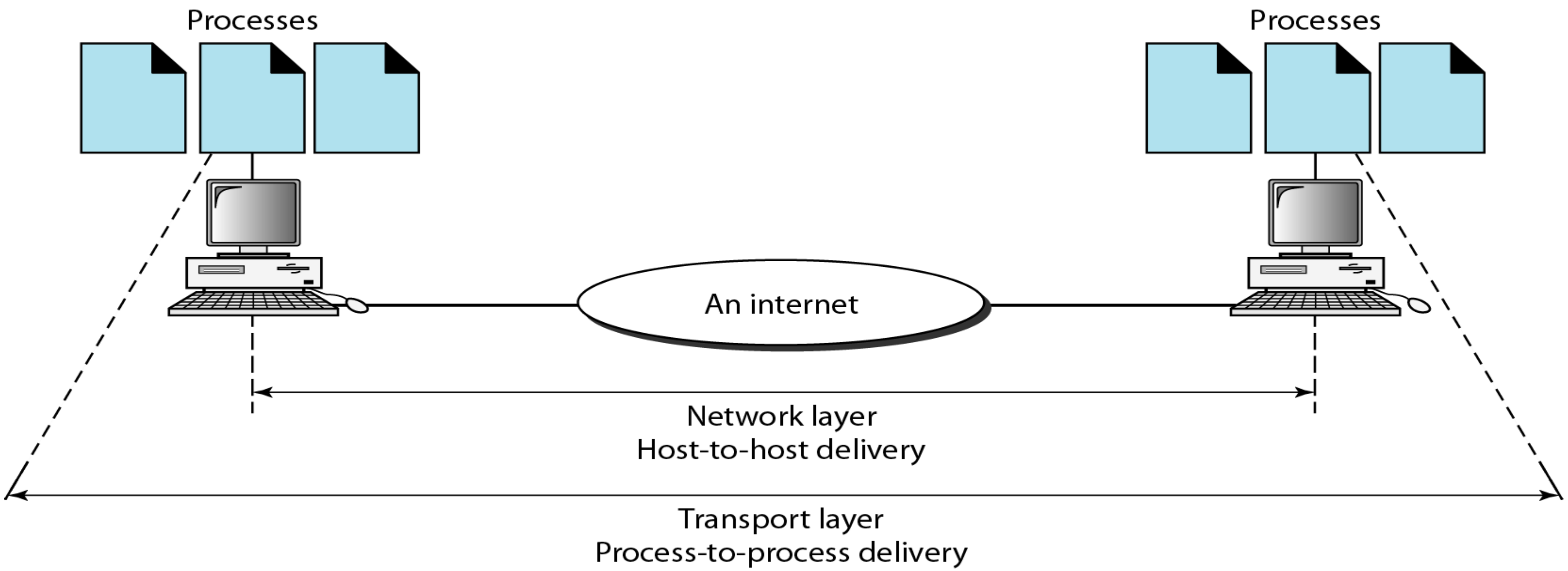
Transport Layer: Introduction

- *Networking function* lower layers provide

- **Physical layer:**
  - Transmits data (signals) between neighbor nodes
- **Datalink layer:**
  - Ensure *reliable* transfer of **data** between neighbor nodes
- **Network layer:**
  - Transmits data (packets) between *any pair* of nodes

Transport Layer

- **The 4th layer in the OSI model**
  - the *first user* level layer in the *OSI architecture*
  - the Transport layer provides communication services that a user can use



Transport Layer- different needs

- To **understand** the **services** provided by the **Transport Layer**, you need to **keep** this in **mind**
- Transport layer provide communication functions to user programs
- *Different* (users) programs have *different* needs
- *Most* program require *reliable* data communication
  - e.g. web browser, email, etc.
- *Some* programs can tolerate some data loss but need *speedy* packet delivery
  - e.g., audio chat tool
- There are *multiple* user programs running on a *single* computer

What programs run on your Computer?

Transport Layer- Protocols

- Catering to *different* needs of the user programs
  - *multiple* Transport Layers protocols available

- **The well-known Transport Protocols are:**
- **TCP (Transmission Control Protocol)**
  - **A heavy-weight (lots of overhead) protocol that provide reliable transfer over the IP protocol**

- UDP (User Datagram Protocol)
  - A light-weight (minimal overhead) protocol that allow users to transmit packets as *quickly* as possible over the IP protocol

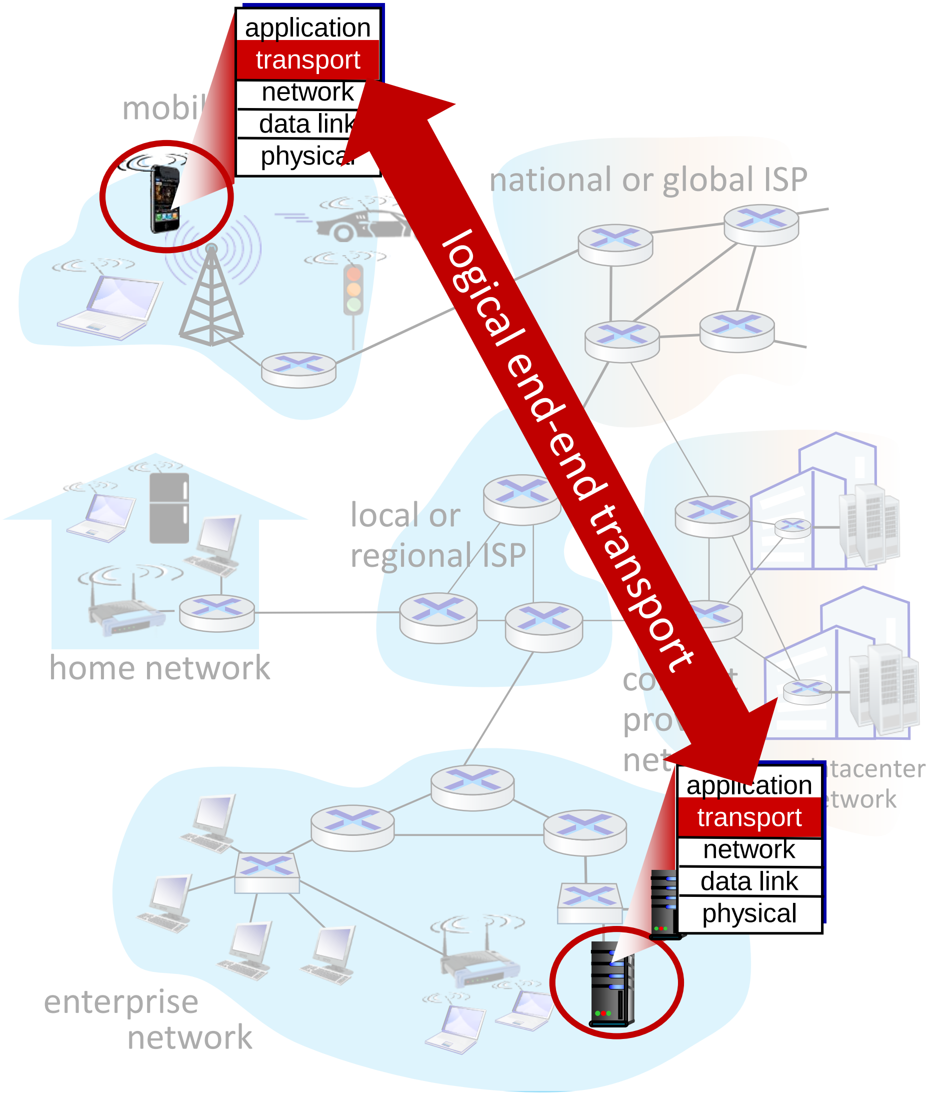
- **A *lesser* know Transport Protocol:**
- **XTP (Xpress Transport Protocol):**
  - **A high speed protocol that was promoted by the XTP forum to replace the TCP protocol**
  - **Wikipedia: click here**

Transport Layer-Services

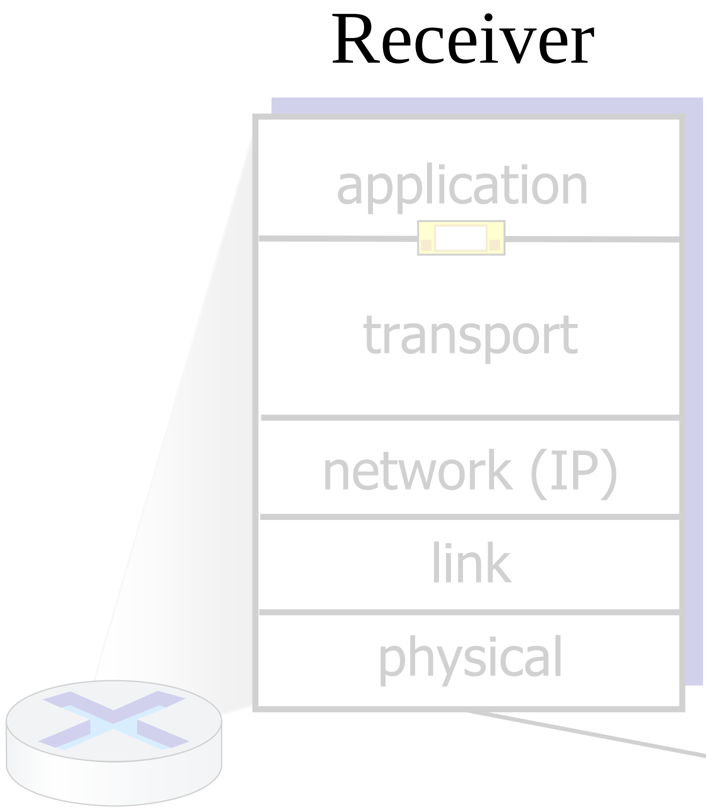
- provide *logical communication* between application processes running on different hosts

- transport protocols actions
  - 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

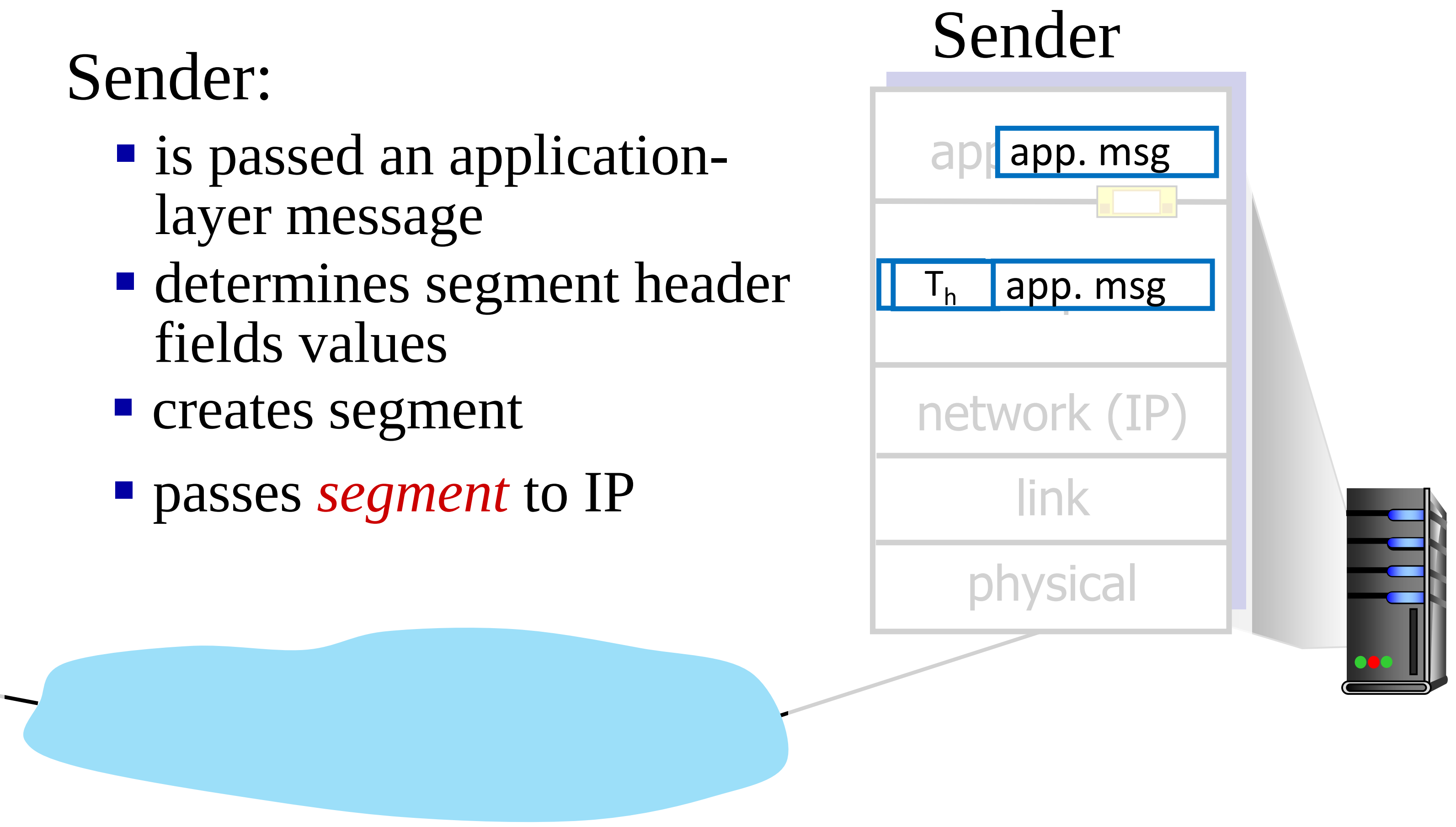


Transport Layer- Actions

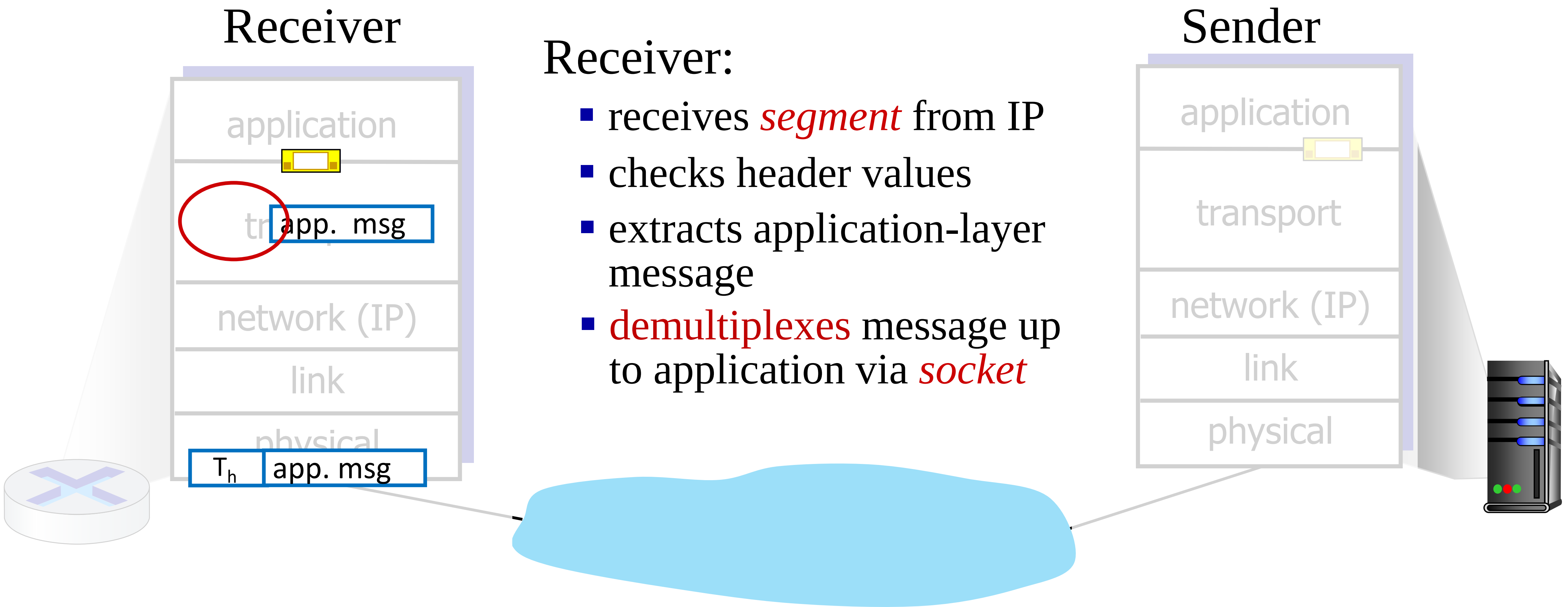


Sender:

- 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*

- *Recall* Socket number
  - The combination of an IPv4 address and a port number
  - A pair of sockets, one at the client and other at the server side, define the TCP/UDP connection end points

University of Nottingham UK | CHINA | MALAYSIA

Transport Layer-Functions

- Useful functions provided by the Transport Layer

Reliable Transfer

- The TCP provides *reliable* communication between 2 user application programs

Multiplexing

- *Every* transport protocol provides multiplexing

Flow Control

- Ensure that the sender's transmission speed do *not* exceed the receive capability of the receiver
- We make sure that the receiver buffers are *not* depleted.

Congestion Control

- The congestion control mechanism will adjust the transmission rate (of the sender) according to the current state of the network
- The goals of congestion control: Transmit as *fast* as possible
- While *not* causing *congestion* in the network.

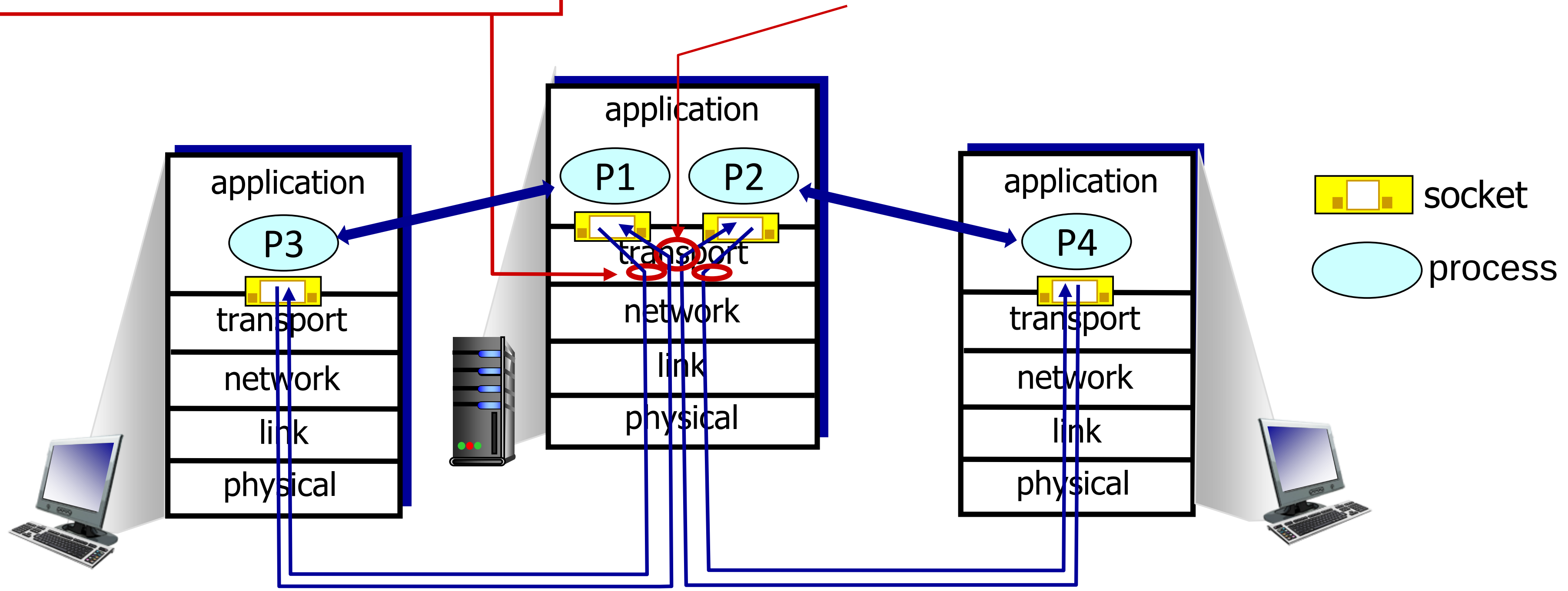
Multiplexing/demultiplexing

multiplexing as sender:

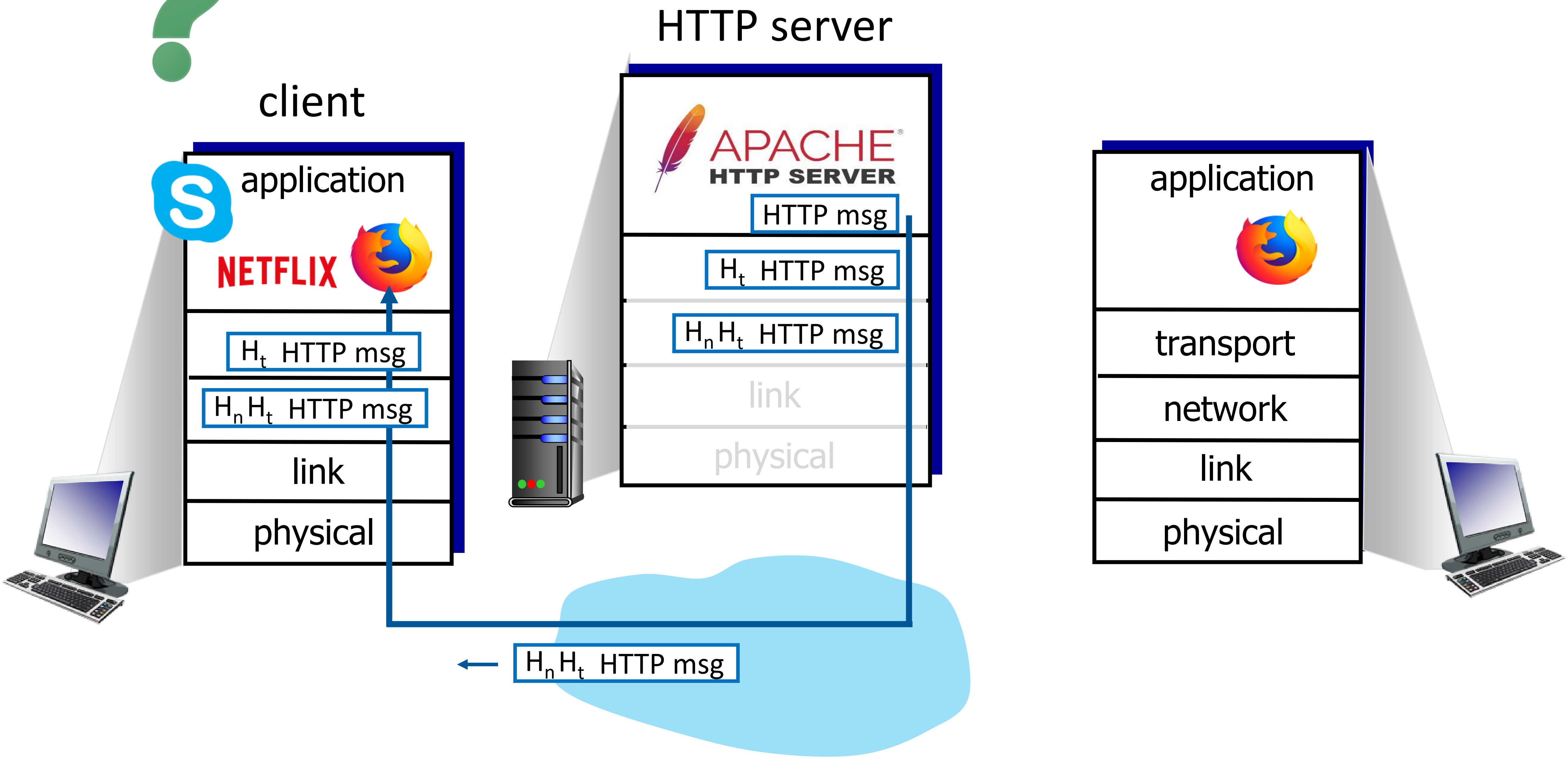
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing as receiver:

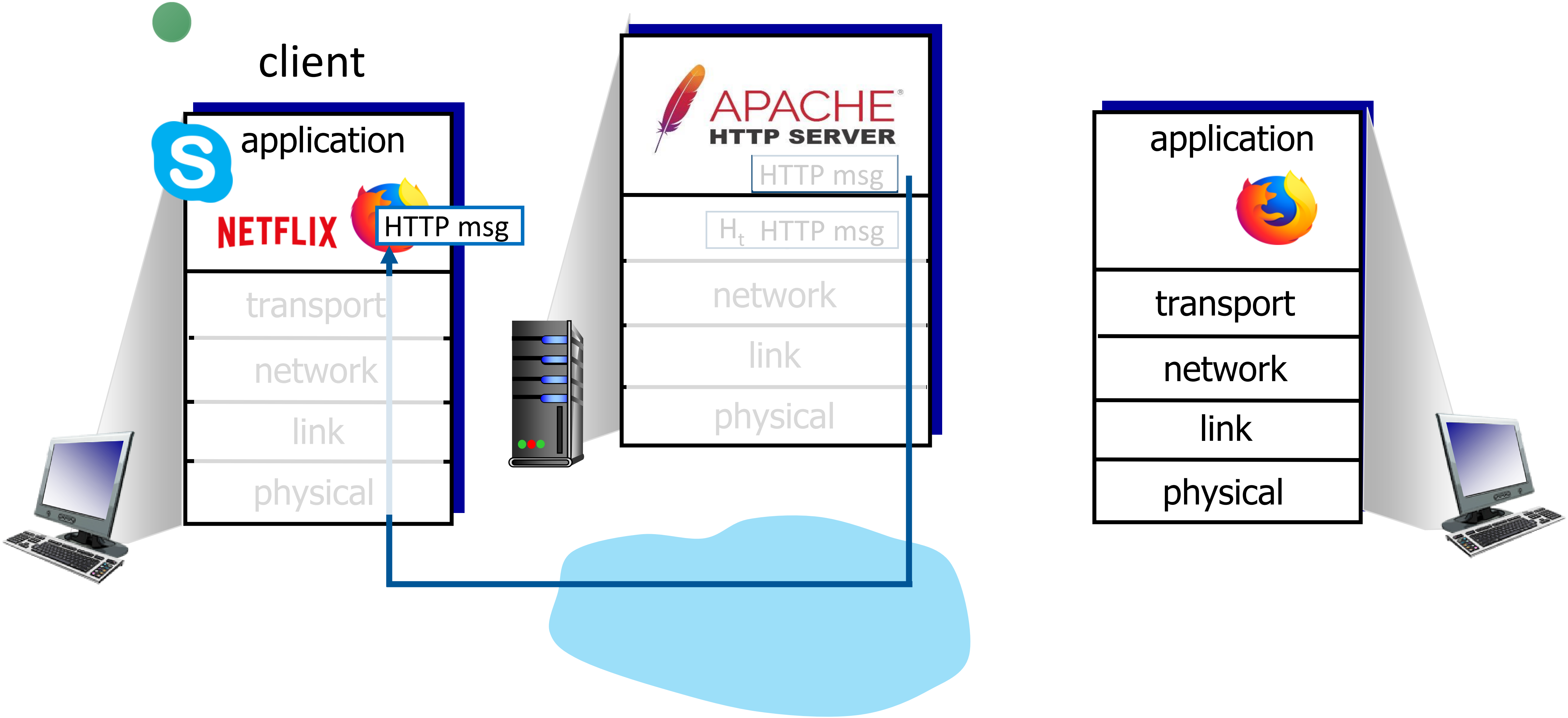
use header info to deliver received segments to correct socket



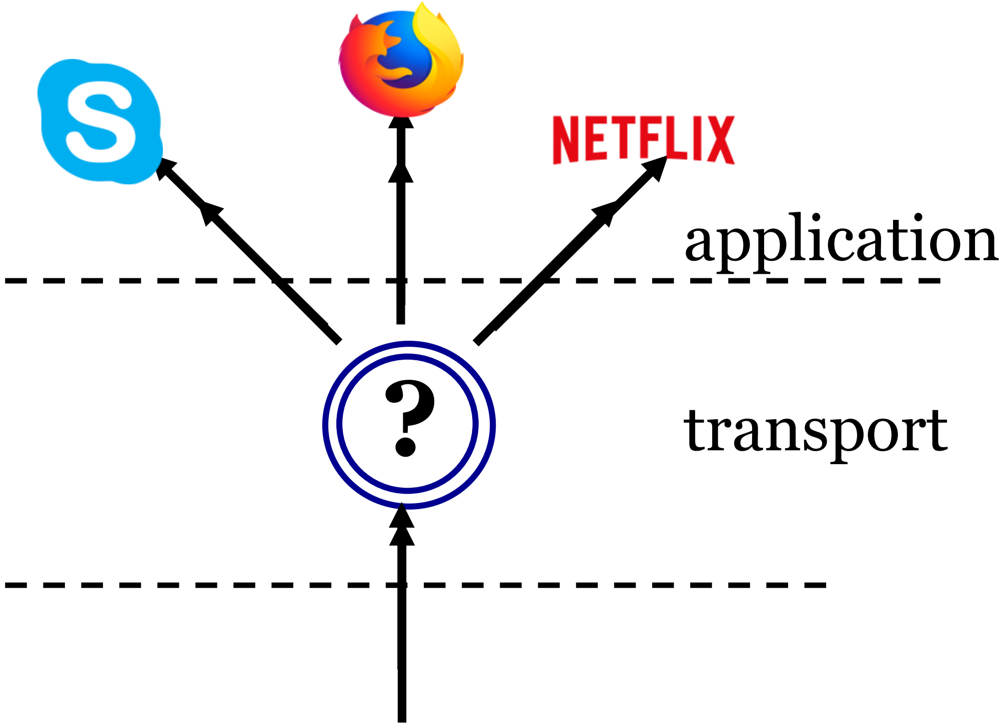
Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?



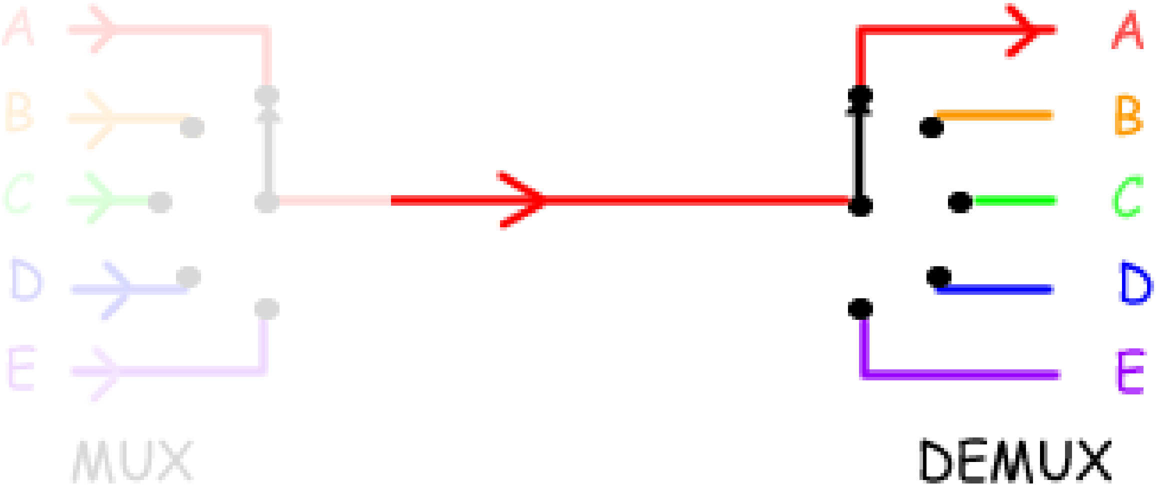
Q: how did transport layer know to deliver message to Firefox browser process rather then Netflix process or Skype process?



De-Multiplexing



de-multiplexing

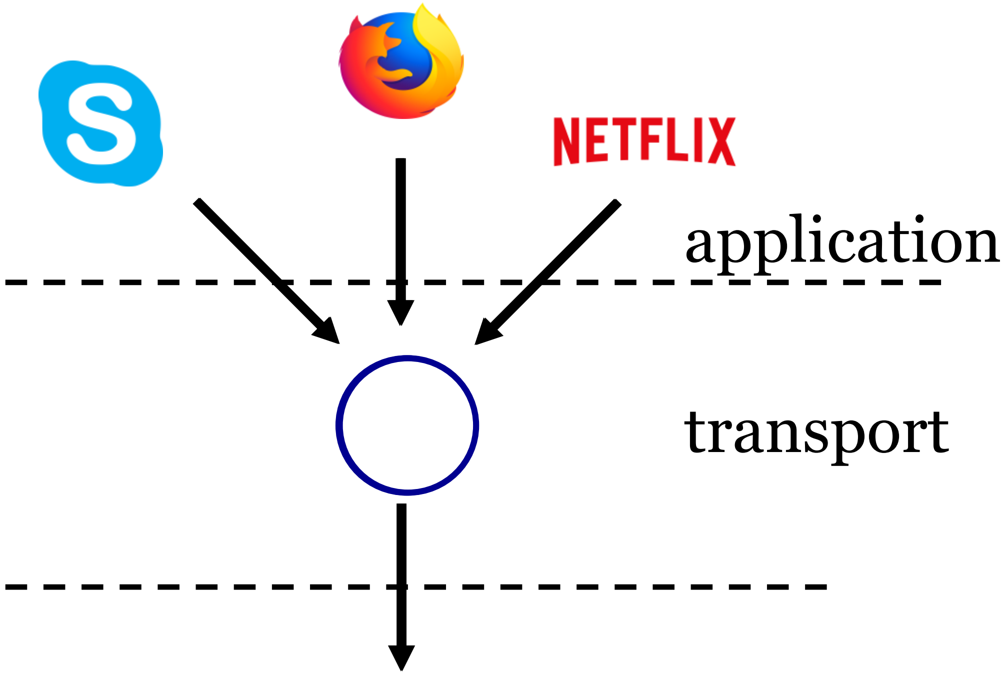


Demultiplexing

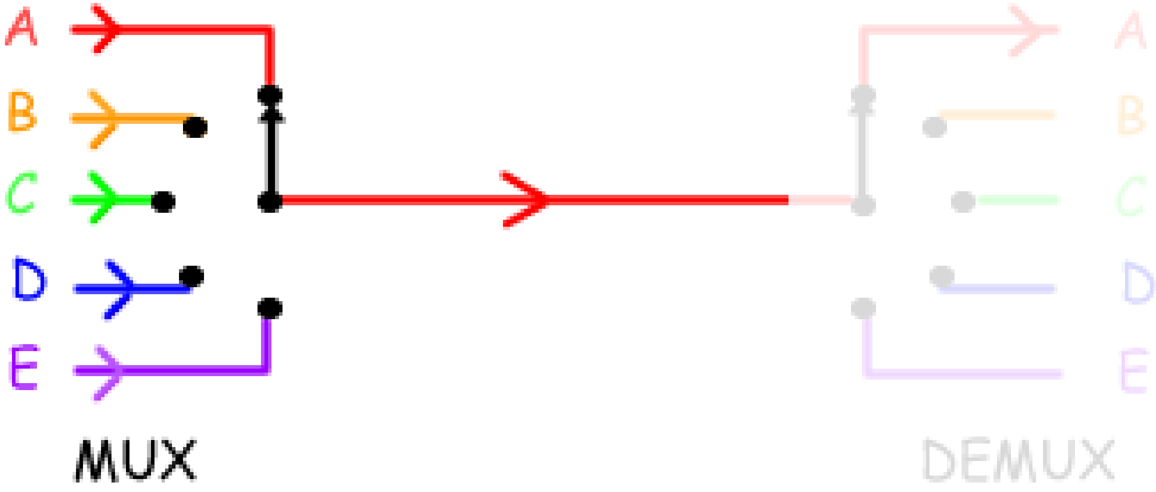

AIRFRANCE
ECONOMY
SKYTEAM
AIRFRANCE
SKY PRIORITY™
SKYTEAM


TSA Pre✓
TSA PRE✓
TSA PRE✓
Transportation Security Administration
tsa.gov
Main Checkpoint
Transportation Security Administration
tsa.gov

Multiplexing



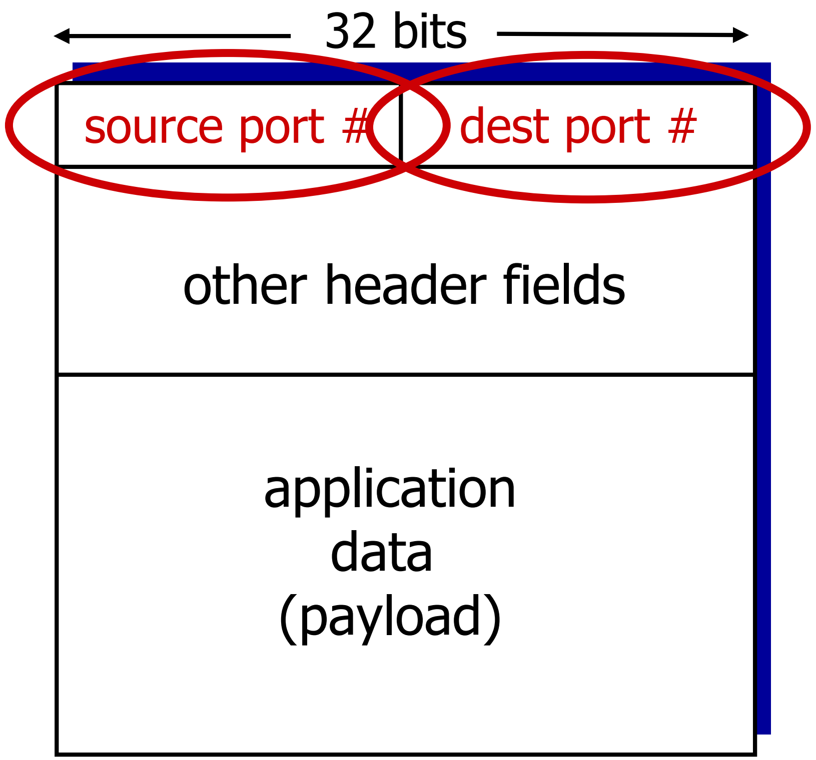
multiplexing



Multiplexing

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

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

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

- When creating datagram to send into UDP socket, must specify
  - destination IP address
  - 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

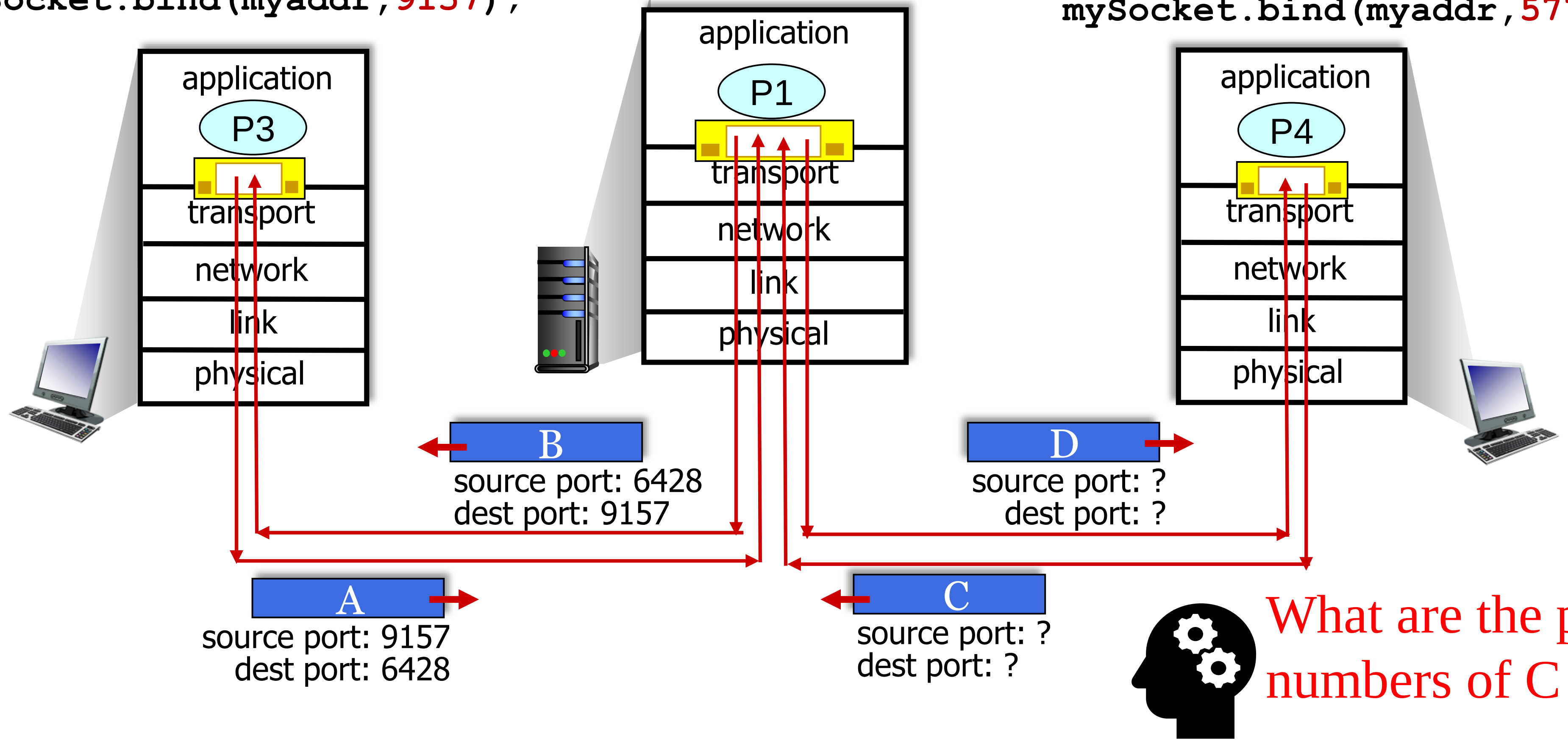
UDP: demultiplexing using destination port number (only)

Connectionless demultiplexing

```
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);
```



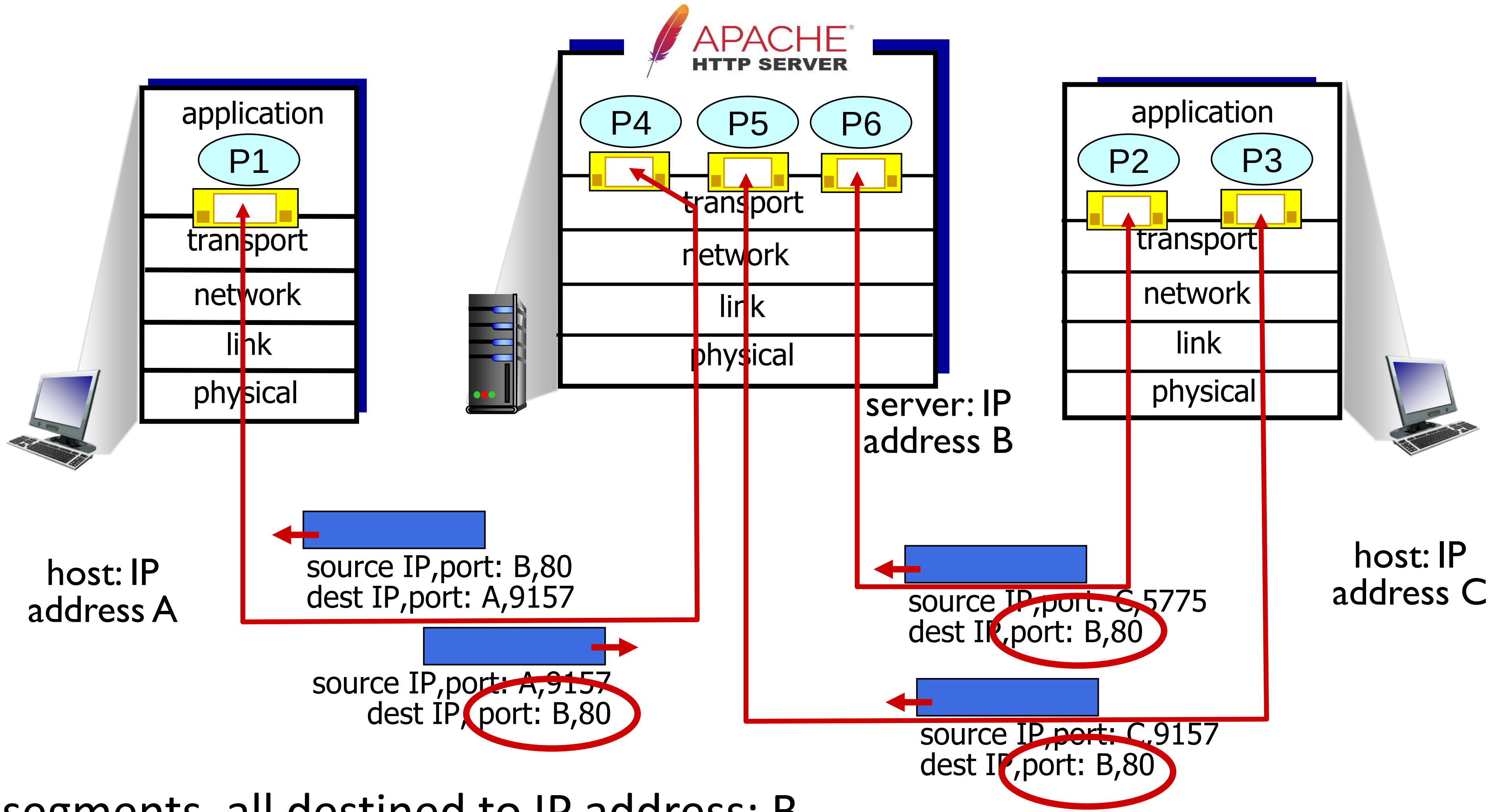
What are the port numbers of C and D?

Connection-oriented demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- server may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
  - each socket associated with a different connecting client

- demux: receiver uses *all four values (4-tuple)* to direct segment to appropriate socket

Connection-oriented demultiplexing



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

University of Nottingham
UK | CHINA | MALAYSIA

(De)Multiplexing-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

Reliability

- Two layers provide reliability
- Data Link Layer and
- Transport Layer

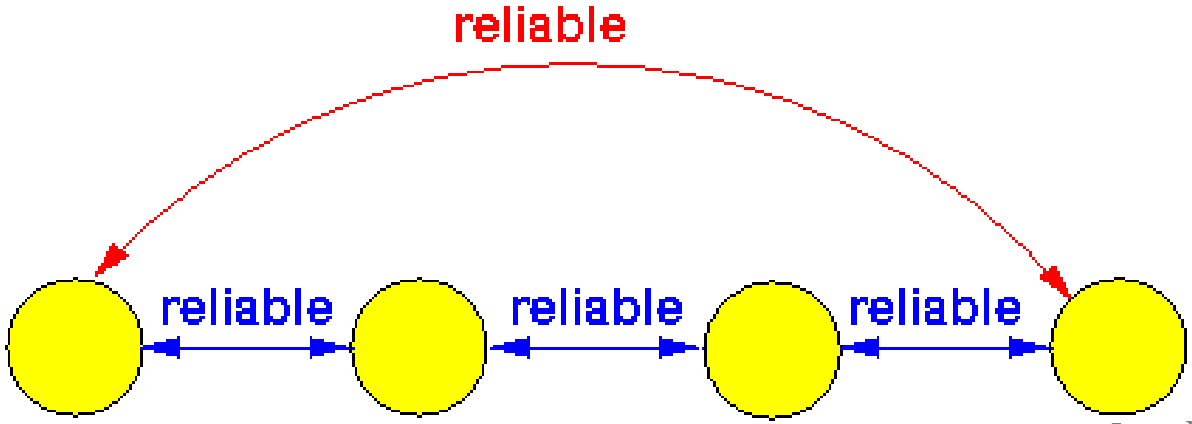
We discussed in last lecture that
The data link layer is responsible for moving frames from one hop (node) to the next.

We discussed in last lecture that
The transport layer is responsible for the delivery of a message from one process to another.

Hop-to-hop vs. End-to-end reliability

- Difference between hop-to-hop reliability (Datalink layer) and end-to-end reliability (Transport layer)
- A common misconception is:

  - If **messages** are **received *reliably*** on a **hop-to-hop** basis:
- Then, the **messages** will be **received *reliably*** on an **end-to-end** basis



- This **claim** is ***false*** !!!

Overview-Protocols in the TCP/IP Suite

Learning Outcomes:

- Understand principles behind transport layer services
- Understand the transport layer protocols
  - UDP: connectionless protocol
  - TCP: connection-oriented protocol

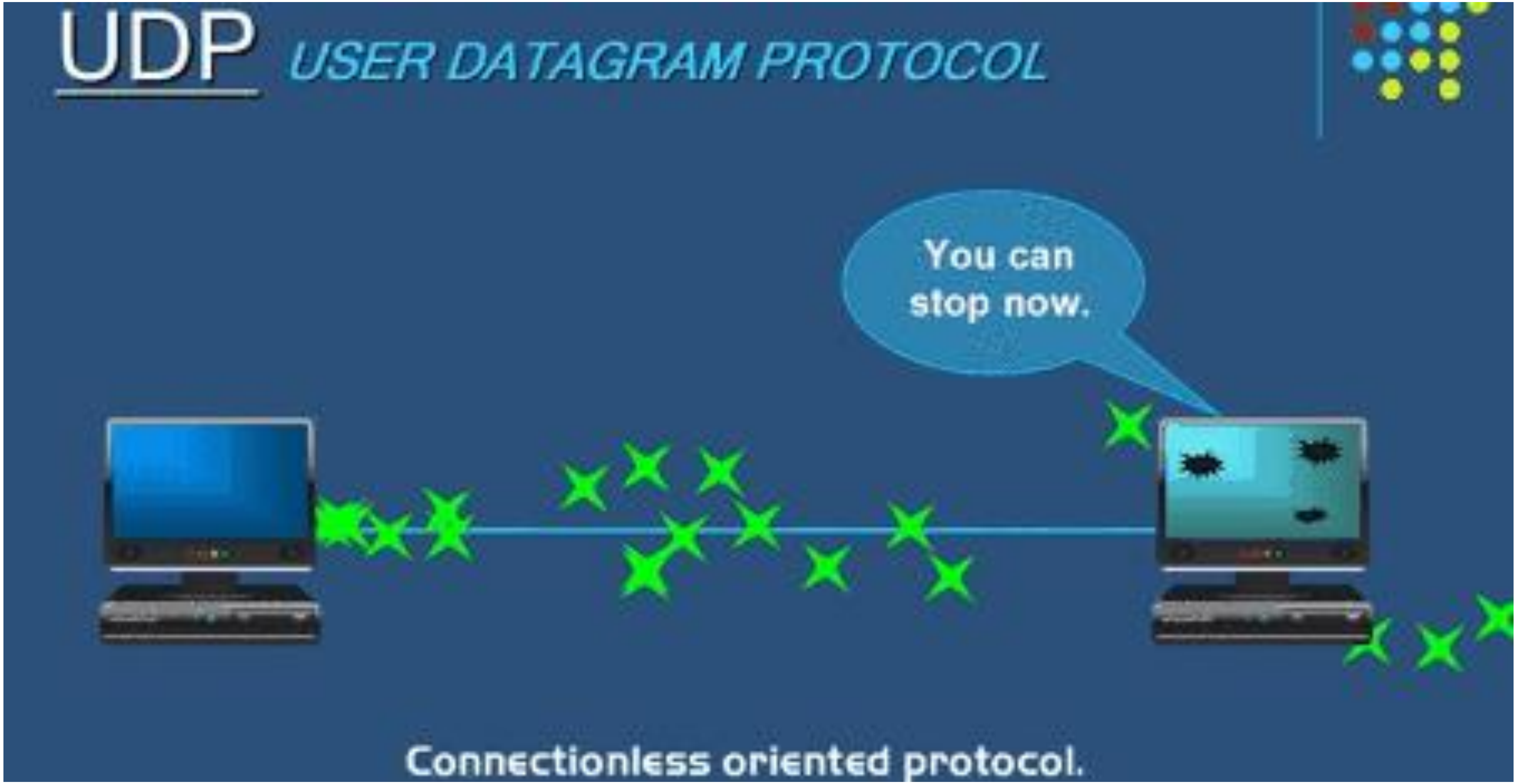
Overview/roadmap:

- We talked about
  - Transport layer actions
  - Transport layer services
    - Multiplexing and demultiplexing
- Next we discuss
  - Connectionless transport: UDP
  - Connection-oriented transport: TCP

UDP: User Datagram Protocol

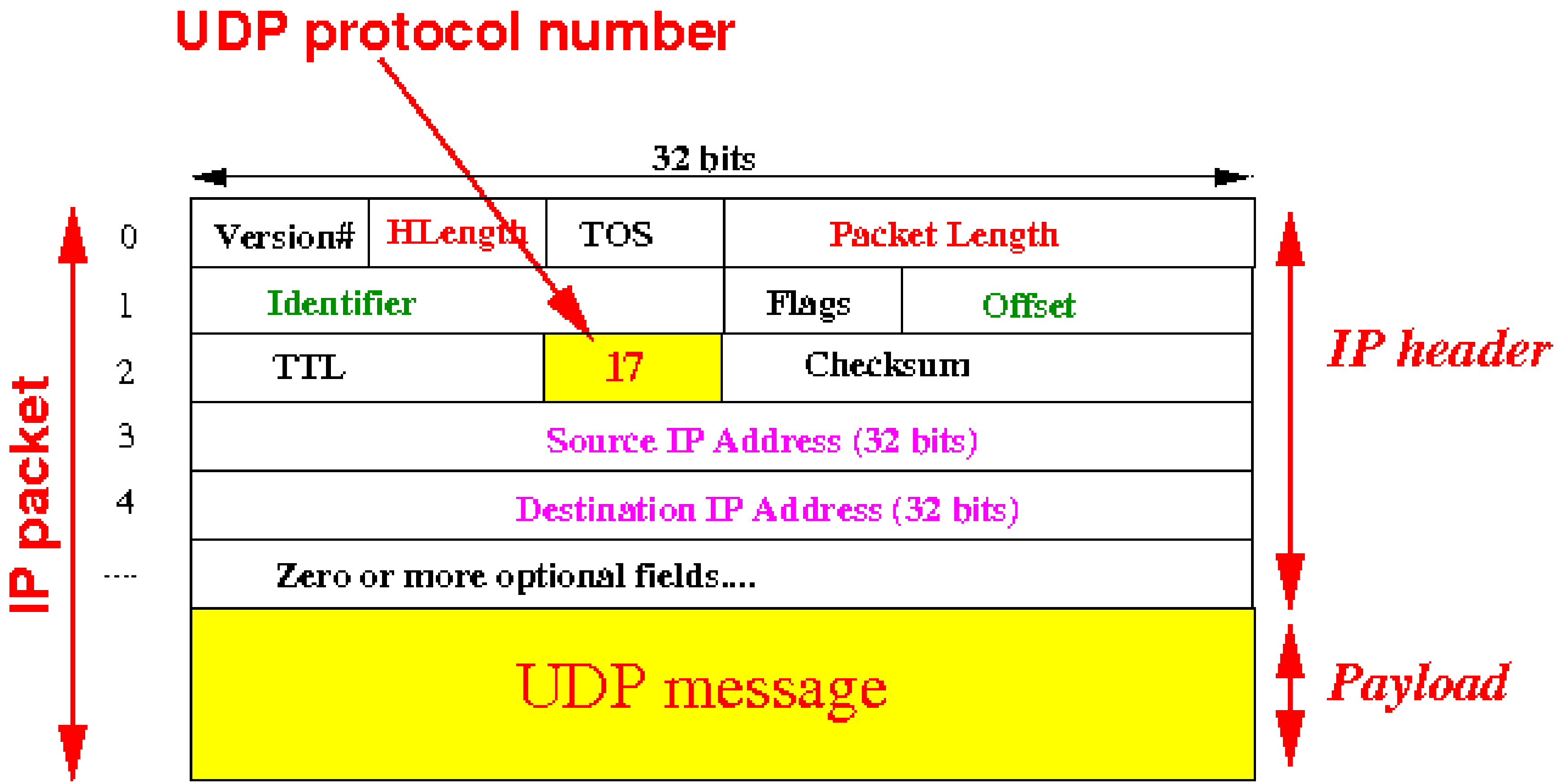
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS- This protocol runs over UDP and uses port 53
  - SNMP- Simple Network Management Protocol
  - HTTP/3
- if reliable transfer needed over UDP
  - add needed reliability at application layer
  - add congestion control at application layer

UDP: User Datagram Protocol



UDP: User Datagram Protocol

- **Service**:
  - Only **Multiplexing** provided by the **UDP** protocol:
- **Messages**:
  - **UDP messages** are **carried** ***inside*** **IP packets**:



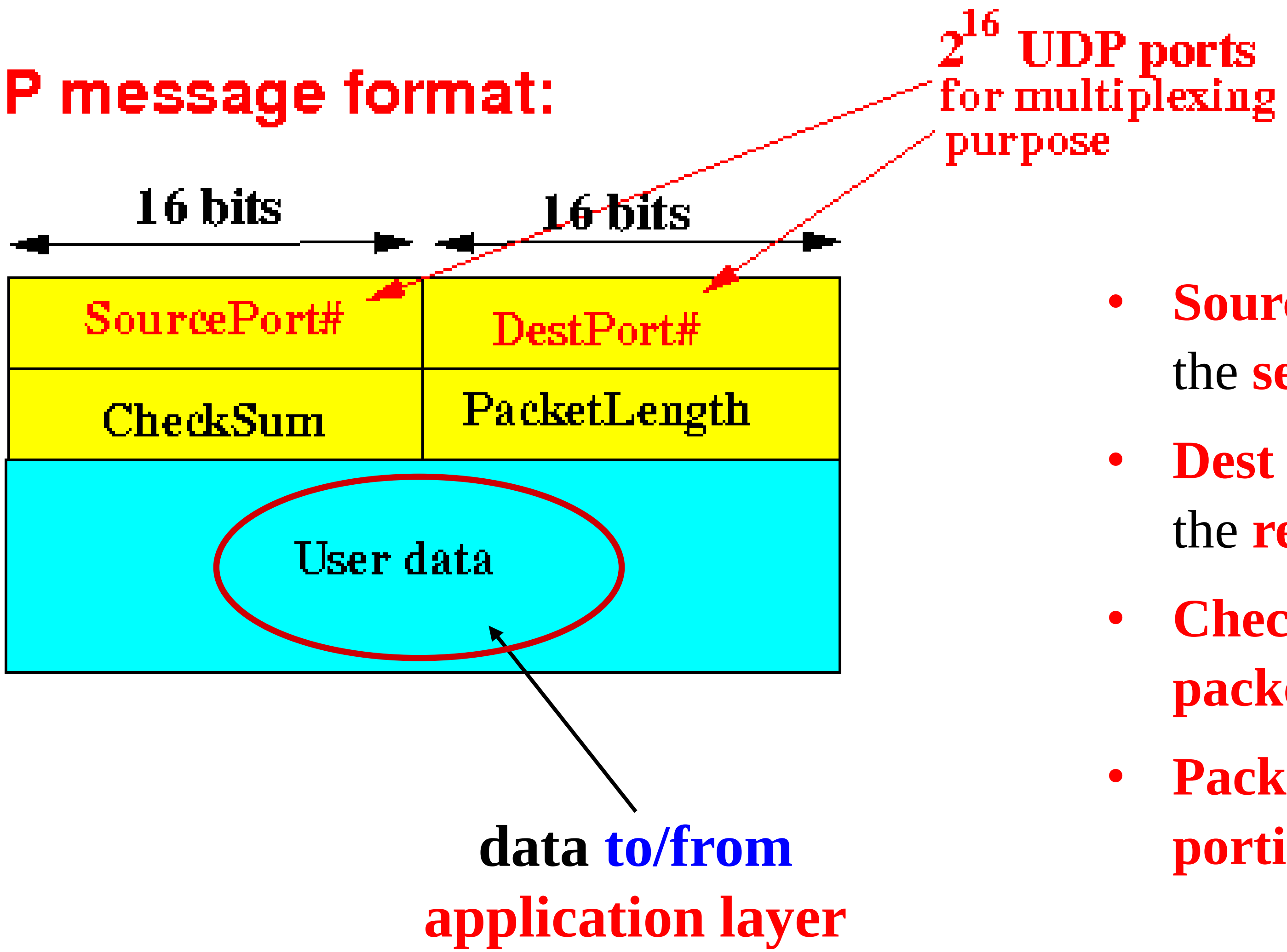
- **Yes**, **yet** another ***encapsulation***....

UDP: Message Format

- **Each transport layer defines** its **own message format**

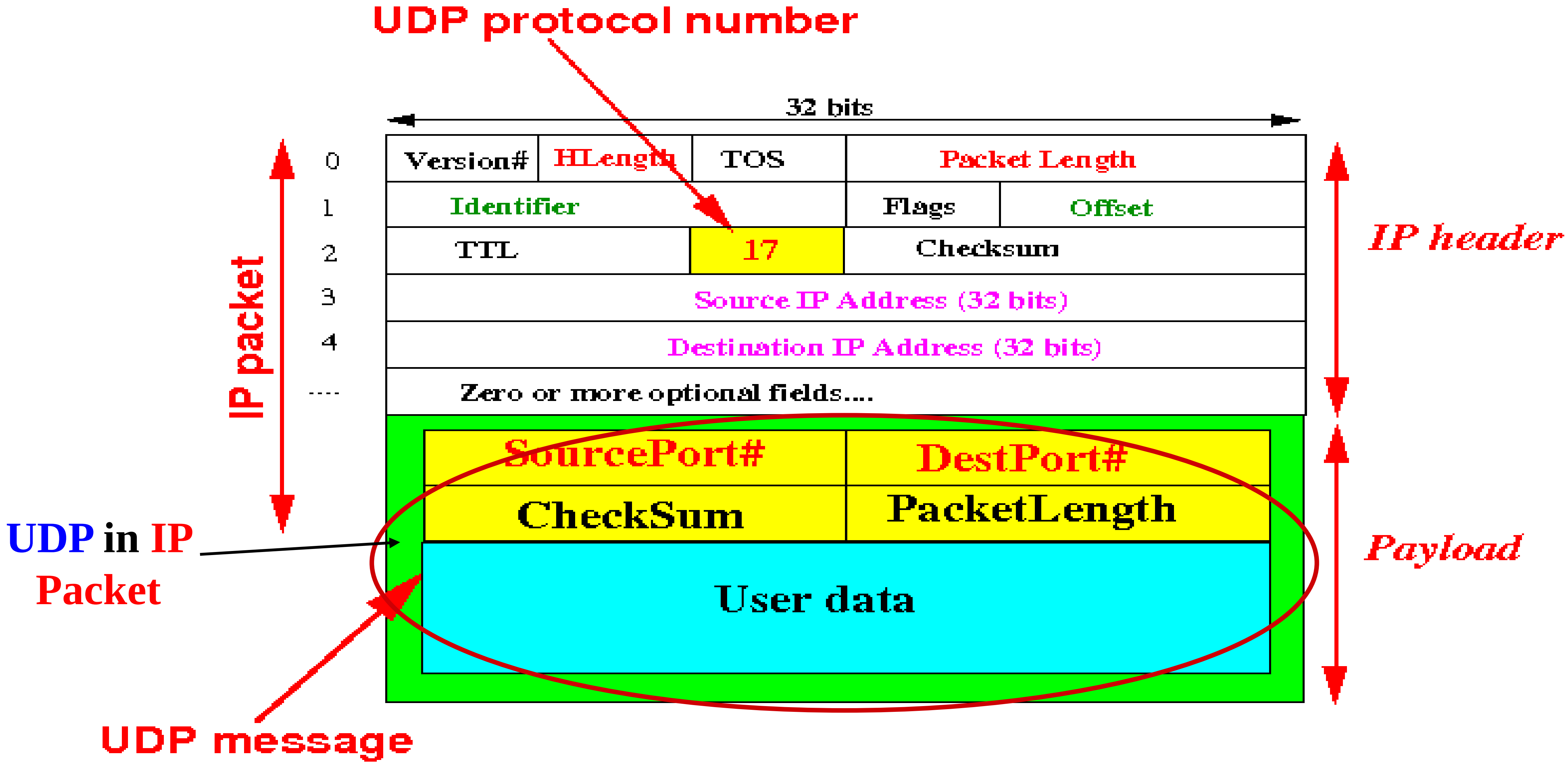
What is the total number of Ports?

UDP message format:

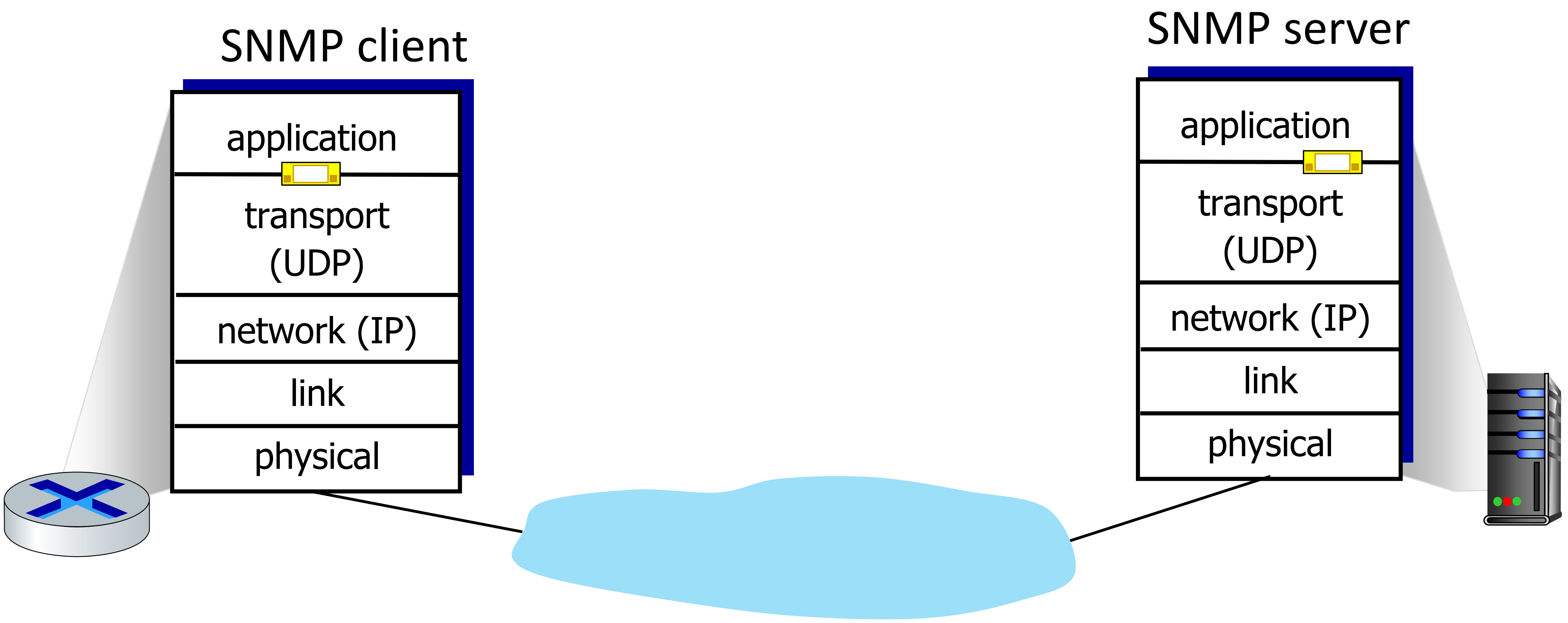


- **Source Port#** = **port number** used by the **sender** (application program)
- **Dest Port#** = **port number** used by the **receiver** (application program)
- **CheckSum** = **checksum** to protect the **UDP packet**
- **PacketLength** = **length** of **user data portion** (in **#bytes**)

UDP in IP


UDP protocol number
32 bits
0
1
2
3
4
....
Version#
HLength
TOS
Packet Length
Identifier
Flags
Offset
TTL
17
Checksum
Source IP Address (32 bits)
Destination IP Address (32 bits)
Zero or more optional fields....
SourcePort#
DestPort#
CheckSum
PacketLength
User data
IP packet
IP header
Payload
UDP in IP Packet
UDP message

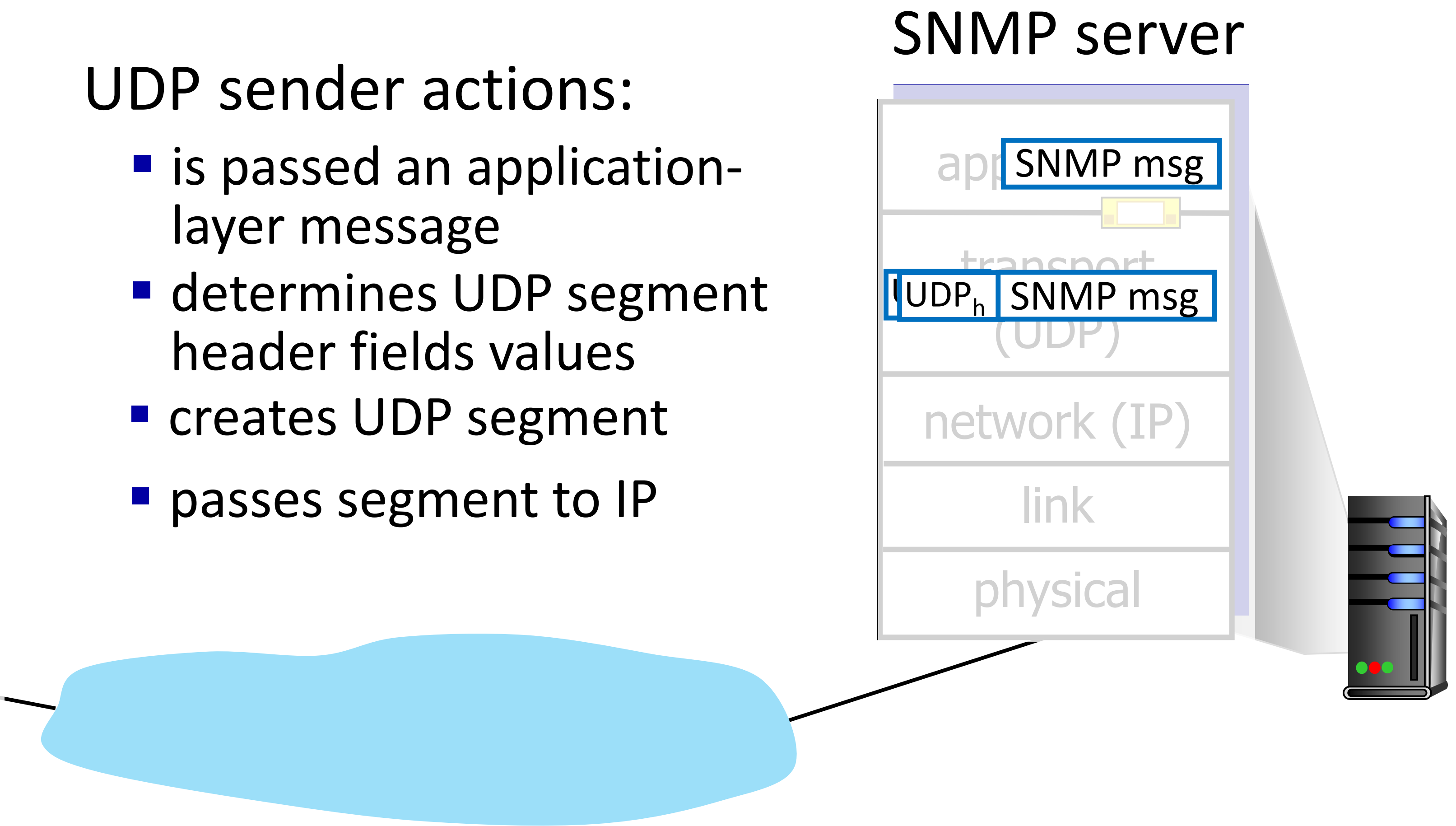
UDP: Transport Layer Actions



UDP: Transport Layer Actions

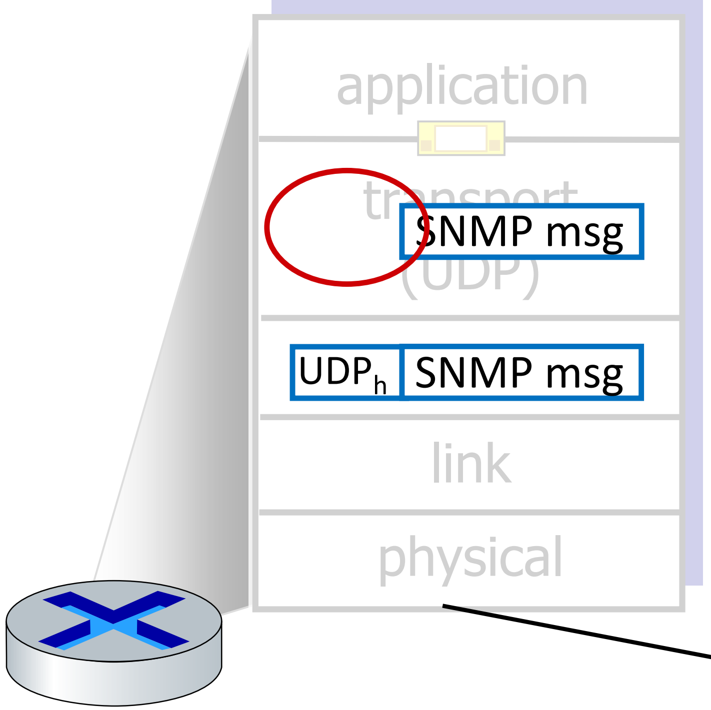
UDP sender actions:

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



UDP: Transport Layer Actions

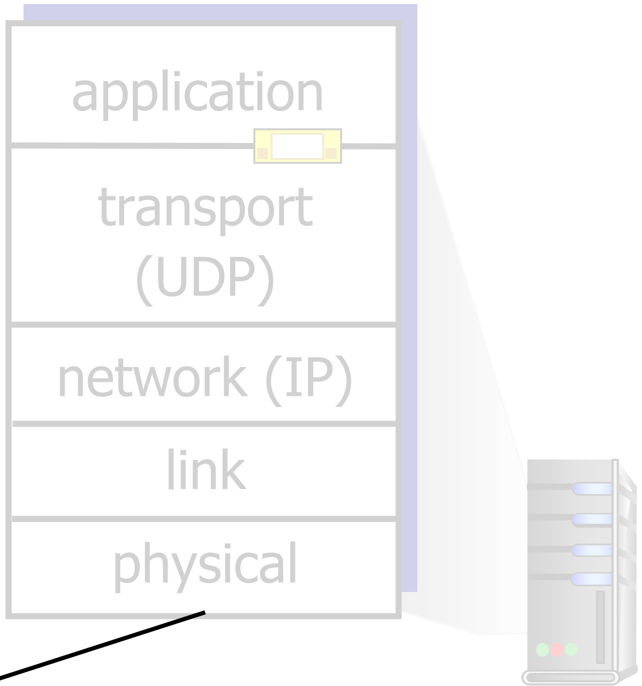
SNMP client



UDP receiver actions:

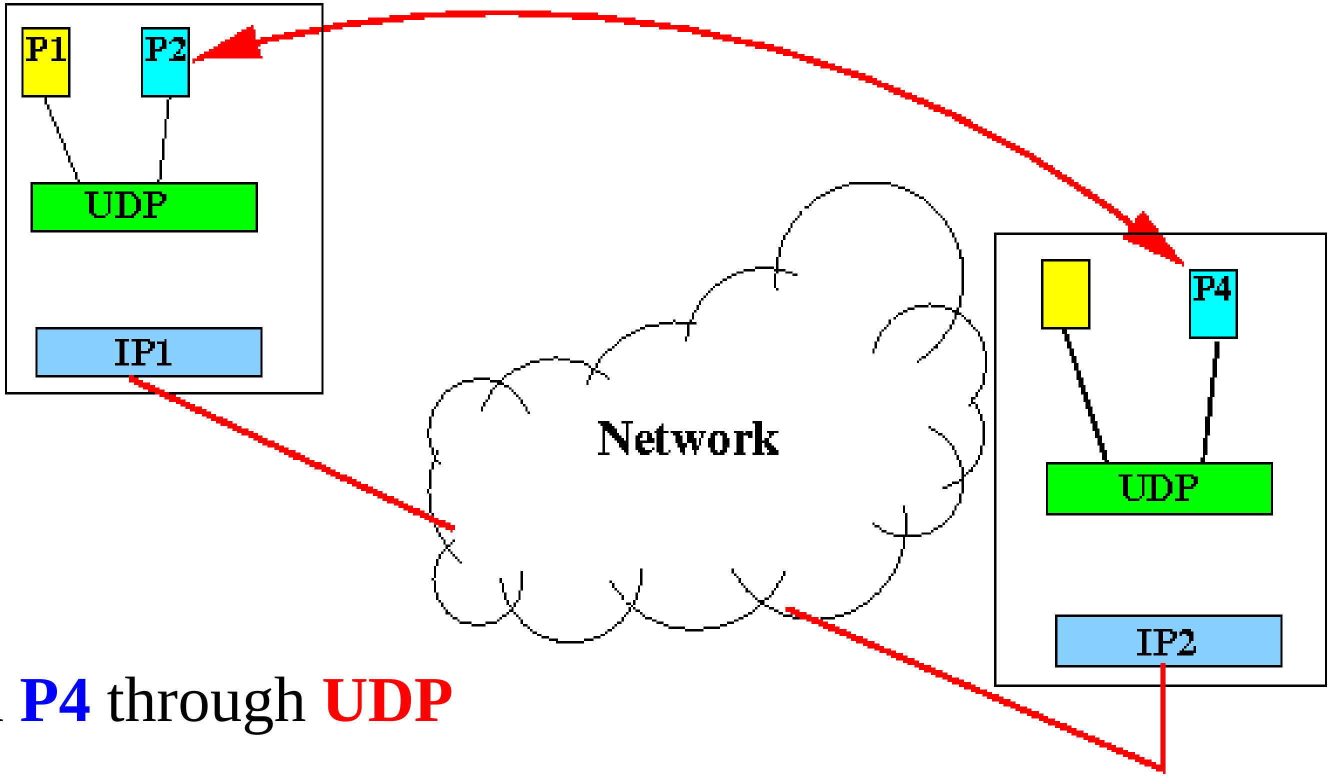
- receives segment from IP
- checks UDP checksum header value
- extracts application-layer message
- demultiplexes message up to application via socket

SNMP server



Multiplexing in UDP

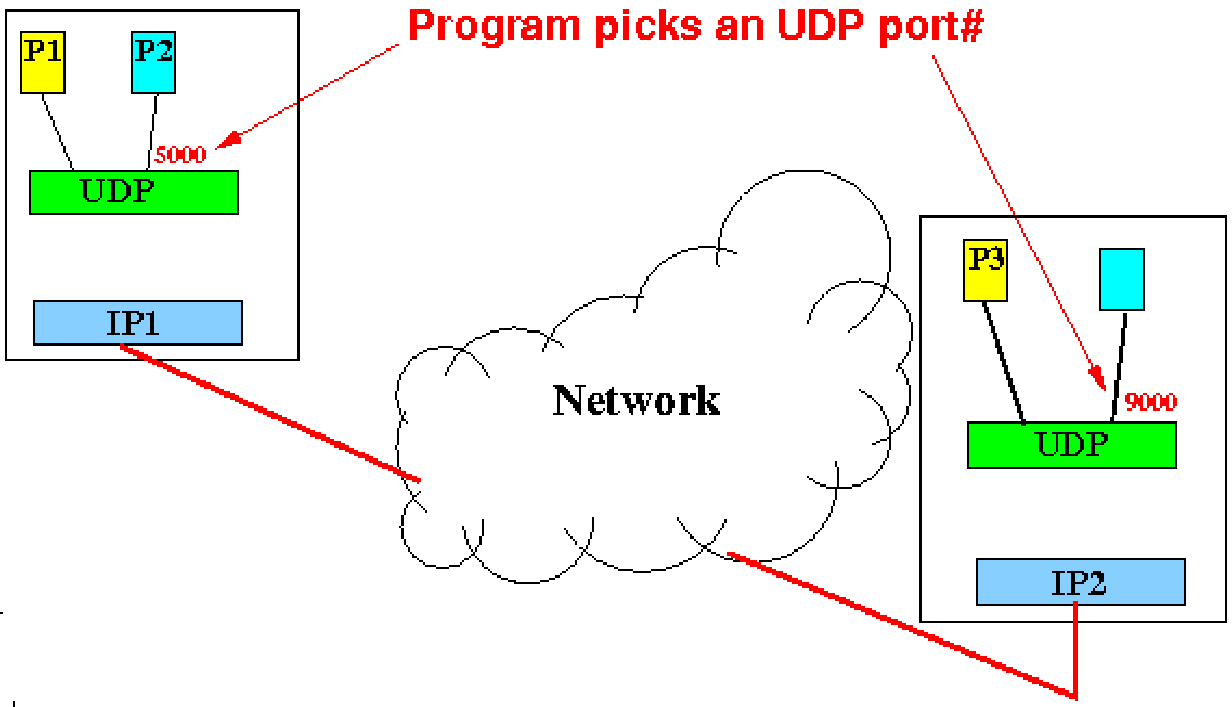
- The **UDP protocol** provide **2^{16} (65536)** different **port numbers** (16 bits)
  - **UDP messages** from a **user (application) program** is ***identified*** by a **unique UDP port number**
- and **vice versa**
  - **Received UDP messages** are ***delivered*** to the **user application program** that **own** that **UDP PortNumber**

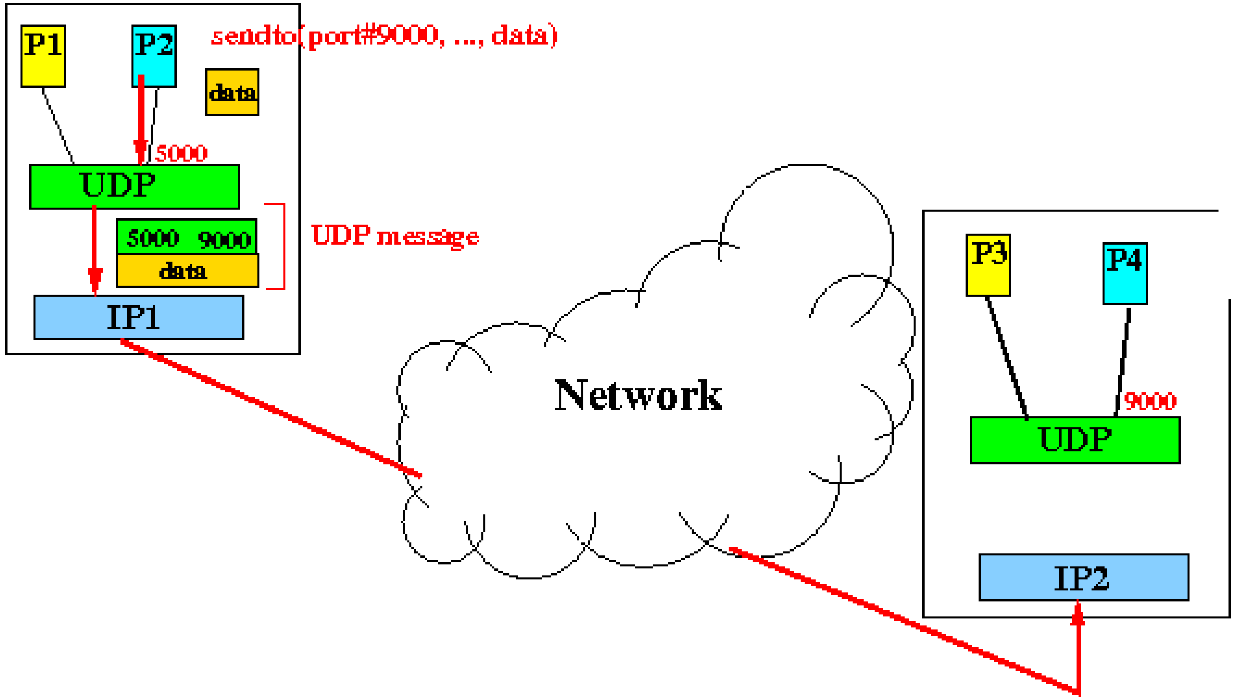


- **Example:**
- Program **P2** communicates with program **P4** through **UDP**

Multiplexing in UDP

- **Example** **Continued …**
- Program **P2** communicates with program **P4** through **UDP**
- **Each program must pick a UDP port#**



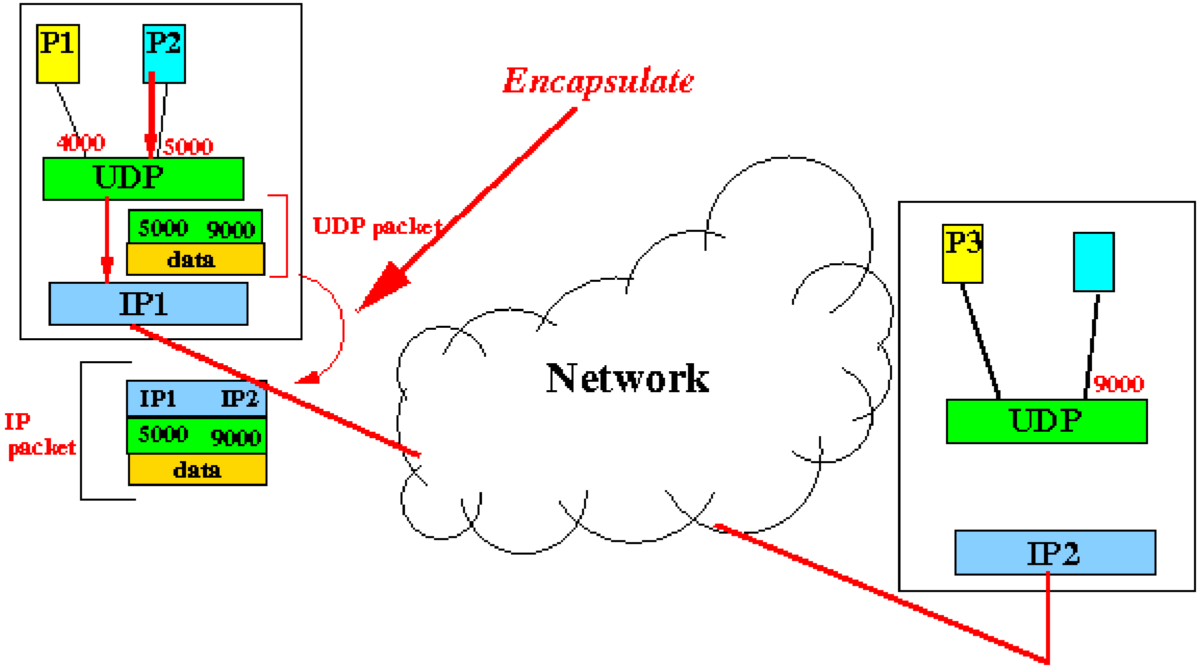


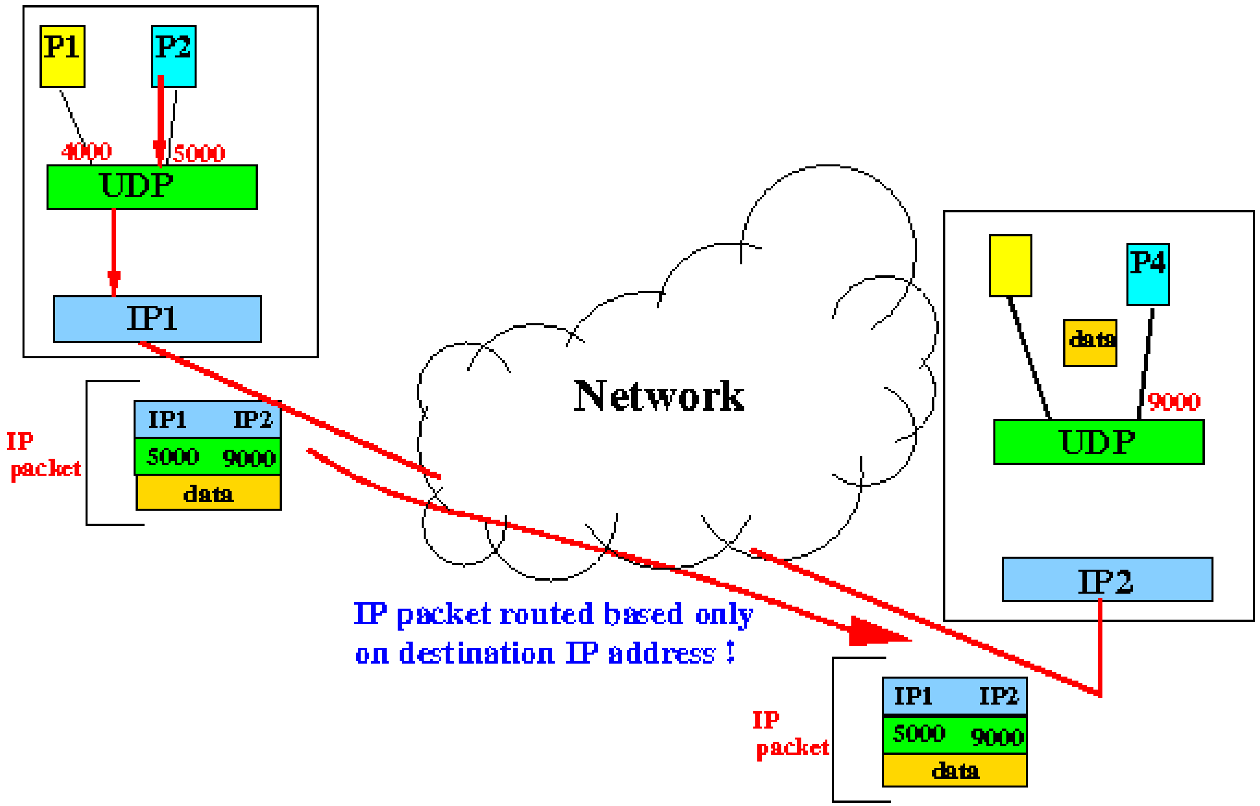
- **P2** transmits a **UDP message** to the **UDP port# (9000)** of **P4**

- The **UDP layer** will **insert** the **UDP port# 5000** used by **P2** as the **source port#**

Multiplexing in UDP

- The **UDP message** is **encapsulated** inside an **IP packet** (by the **IP layer**)

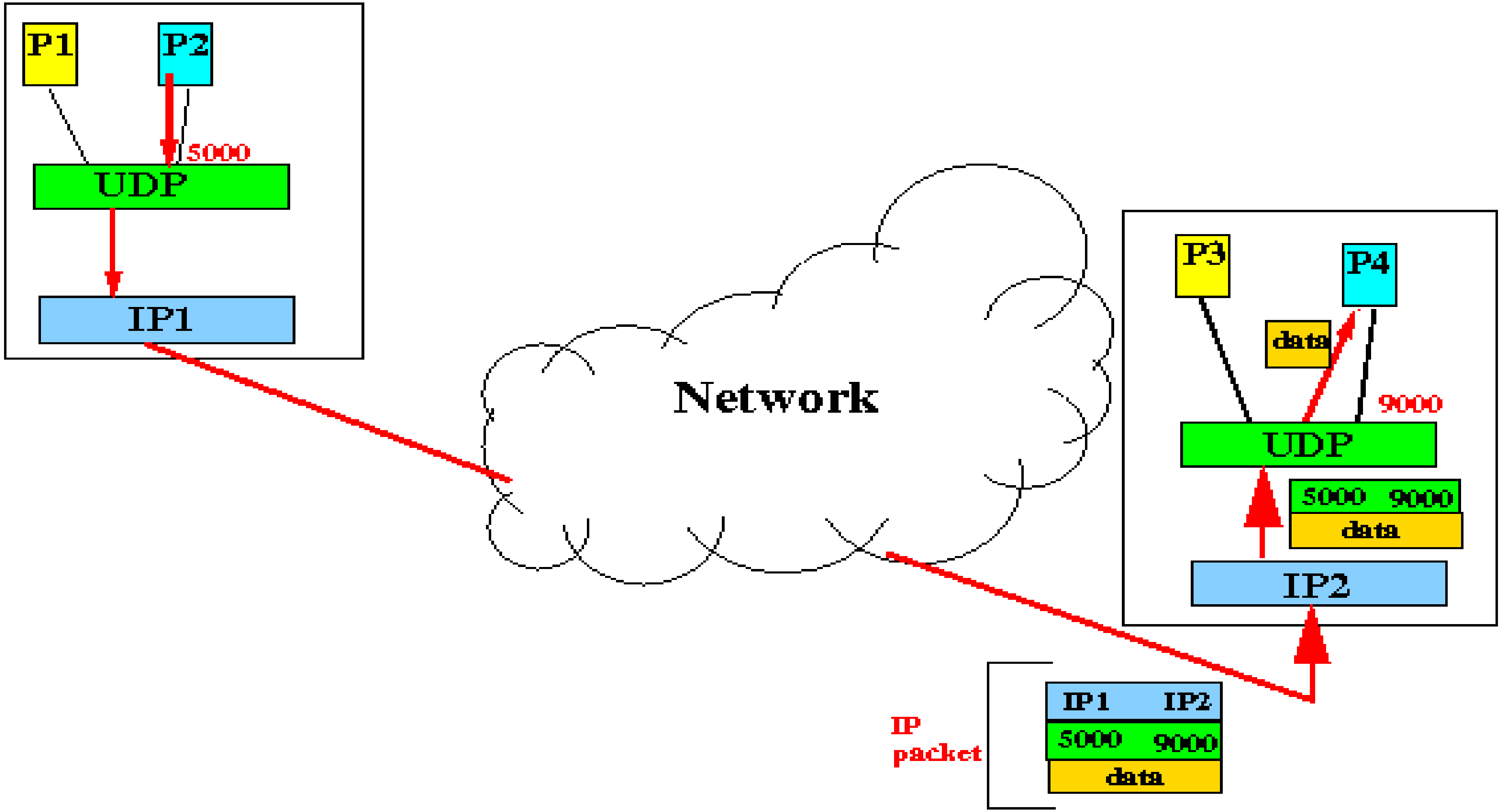




The **IP packet** is **forwarded** using ***only*** the **destination IP address**.

Multiplexing in UDP

- Here, the **UDP message** is **delivered** to the **port #9000** own by the program **P4**



UDP: Strengths and Weaknesses

- **Strength** of the **UDP protocol**
  - **UDP** is ***very* light weight** and has **almost no processing overhead**

- **Weakness** of the **UDP protocol:**
  - **UDP** does ***not*** provide ***any* additional service**
  - **Writing application program** with **UDP** is ***very* difficult**

- **Fact:**
  - The **UDP** protocol is **ideally suited** for
    - **Network research** that want to **experiment** with ***novel* network techniques**
    - the **UDP layer** adds **very little processing overhead**

- **Reason**
  - **Researchers** who wants to **implement new protocols** do **not want**
    - the **transport level software** to ***add* extra overhead** (will **give *biased* performance results**)
    - So that the **performance** of the **experimental protocol** can be **measured *accurately***

UDP checksum

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

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

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

Internet checksum

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?*

Internet checksum: an example

example: add two 16-bit integers

	1	1				1				1				1			
		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

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

Internet checksum: weak protection!

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

0 1

1 0

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)

Overview-Protocols in the TCP/IP Suite

Learning Outcomes:

- Understand principles behind transport layer services
- Understand the transport layer protocols
  - UDP: connectionless protocol
  - TCP: connection-oriented protocol

Overview/roadmap:

- We talked about
- Transport layer actions
- Transport layer services
  - Multiplexing and demultiplexing
- Connectionless transport: UDP
- Next we discuss
- Connection-oriented transport: TCP

TCP: overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries"
- 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: Transmission Control Protocol

University of Nottingham
UK | CHINA | MALAYSIA

TCP: Transmission Control Protocol

- **Services**:
  - **Multiplexing same** as **UDP** protocol
  - ***End-to-end* reliable communication**
  - **Flow control**
  - **Congestion control**

- **Multiplexing**:
- **Multiplexing** in **TCP** is **implemented** in the **same way** as in **UDP**
- The **TCPprotocol also** provides **2^{16} (65536) communi cation ports** for use to **application programs**
- **User applications** select and associate with a **TCP port#**
- **TCP packets** contains **TCP port numbers** to **identify the messages from *different* user applications**

NOTE:
- **TCP ports** and **UDP ports** are **different entities** inside **one computer**
- There are **65536 UDP ports** and **65536 TCP ports** inside **one computer**

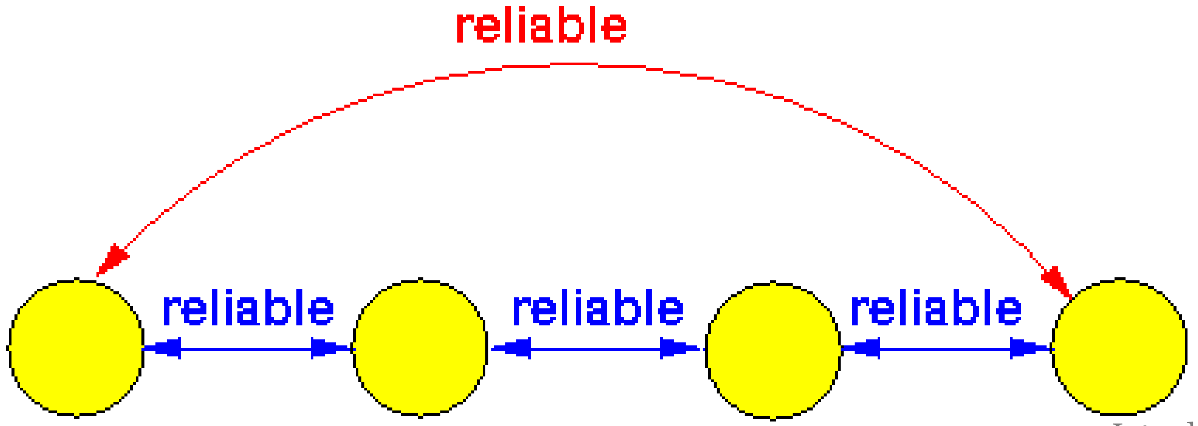
Hop-to-hop vs. End-to-end reliability

- Difference between hop-to-hop reliability (Datalink layer) and end-to-end reliability (Transport layer)

- A common misconception is:

  - If **messages** are **received *reliably*** on a **hop-to-hop** basis:

- Then, the **messages** will be **received *reliably*** on an **end-to-end** basis



- This **claim** is ***false*** !!!

TCP: End-to-end reliable communication

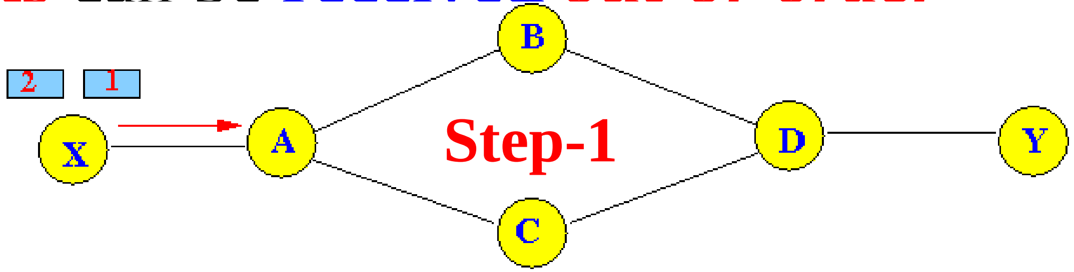
- **Reliable Transfer**
  - **TCP** uses the **sliding window** protocol to provide **reliable communication**

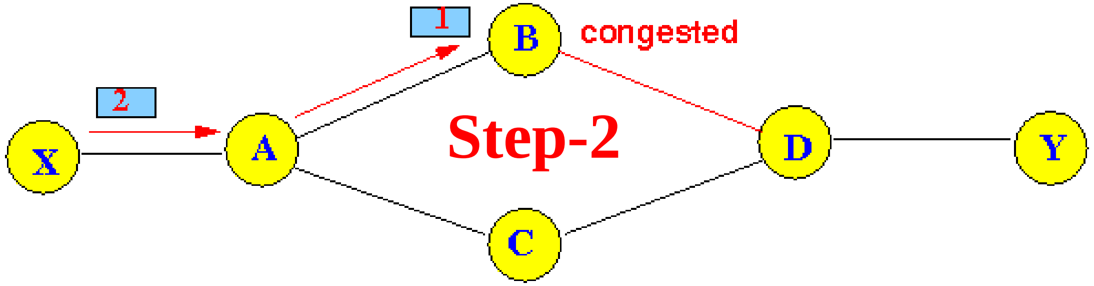
- **Delay** over a ***single* hop** in the **datalink layer** is **relatively *constant***:
  - **Timeout values** (for **frame retransmission**) can be **set *very accurately***

- **Delay** over a ***multiple* hops** in the **Transport Layer** is **extremely *variable*** (due to **congestion**):
  - **Timeout values** (for **packet retransmission**) must be ***adjusted* continuously**.

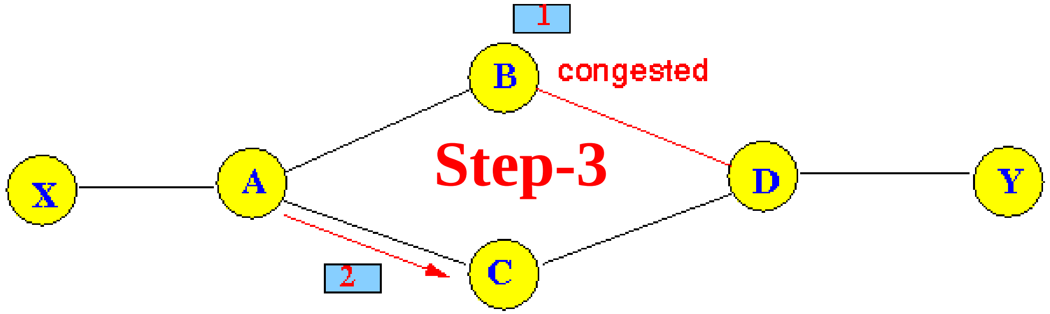
What can cause end-to-end errors

- In **end-to-end** communication, **packets** can be **received** ***out-of-order***
- **Example**
  - **Source X** transmits **2 messages**:

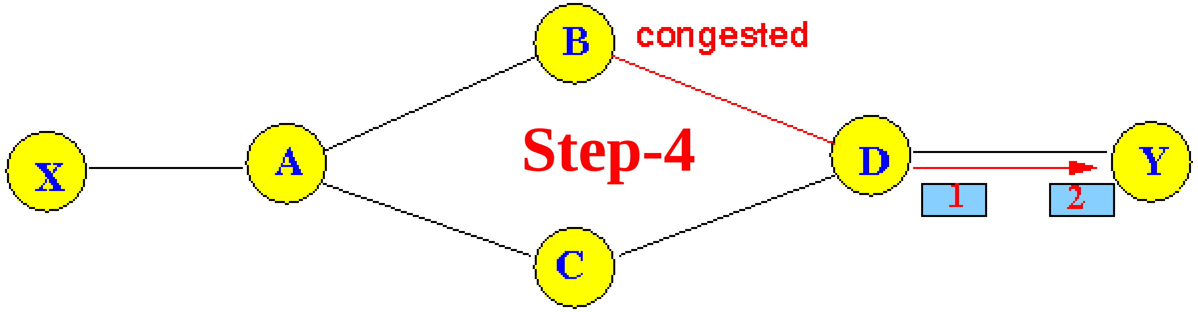




- **Packet 1** is **routed** into a **congested node B**:



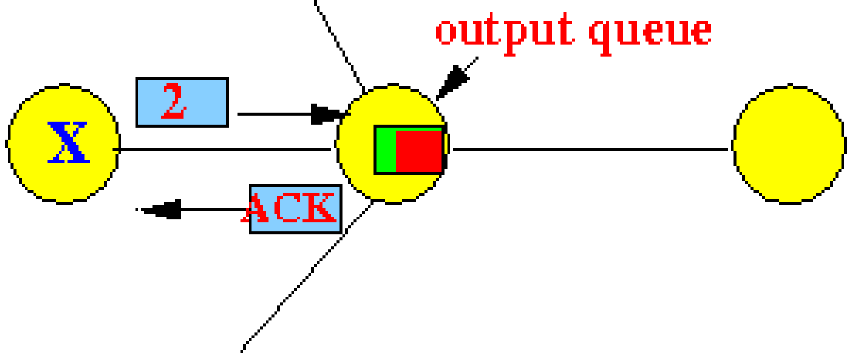
- **Router A *recomputes*** routing tables and **route packet 2** to a ***less* congested** node

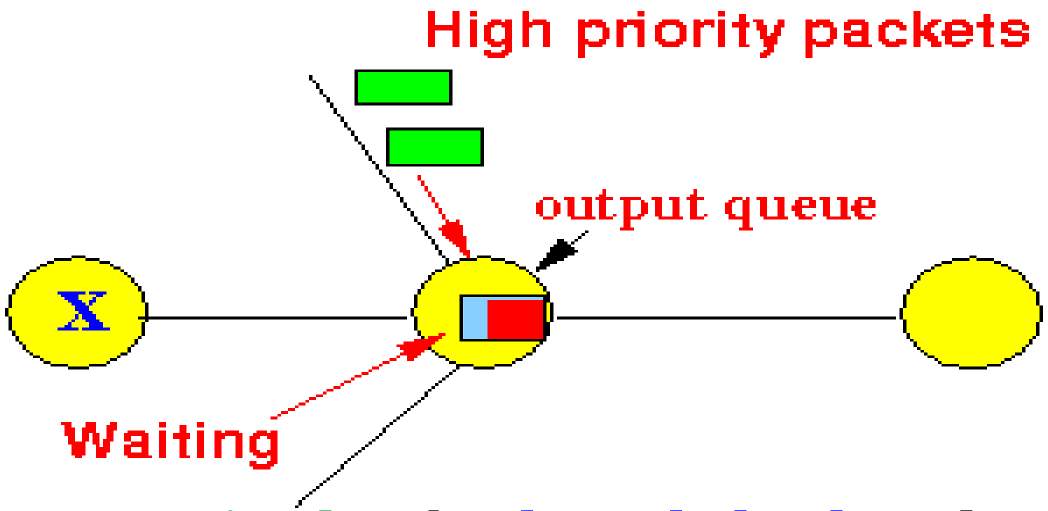


- The **packets** will **arrive *out-of-order***:

What can cause end-to-end errors

- ***Received*** (and **acknowledged**) **packets** can be **deleted** ***later*** (to **make space** for a ***higher*** **priority** packet)
- **Example**
  - A **router** received (and **acknowledged**) a **packet** from **X**
  - The **output queue** is ***almost*** **full**





- Some **received** and **acknowledged** **packet(s)** will have to be ***discarded*** !

Conclusion

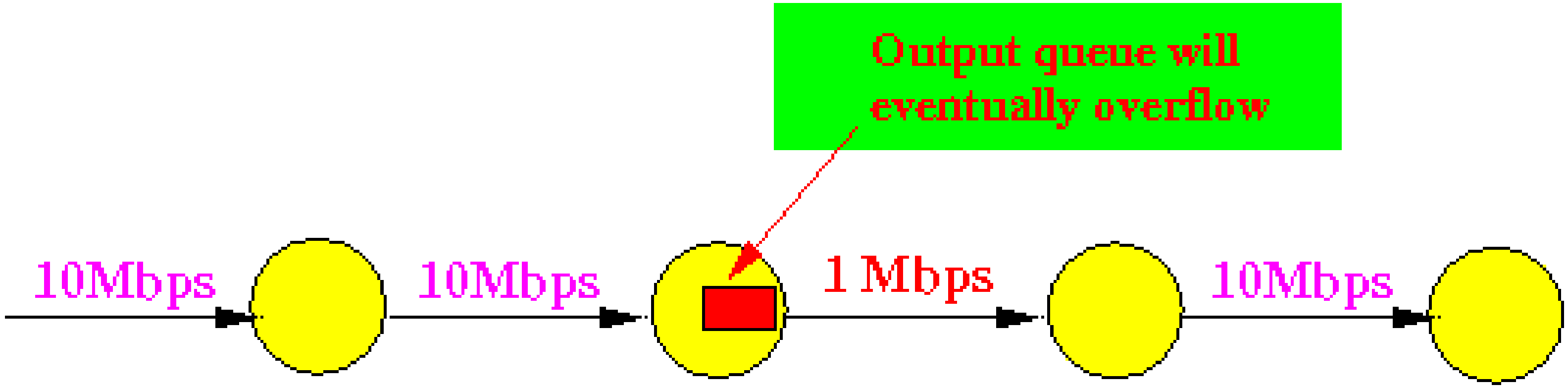
Hop-to-hop reliability **does** ***not*** ensure **end-to-end reliability**

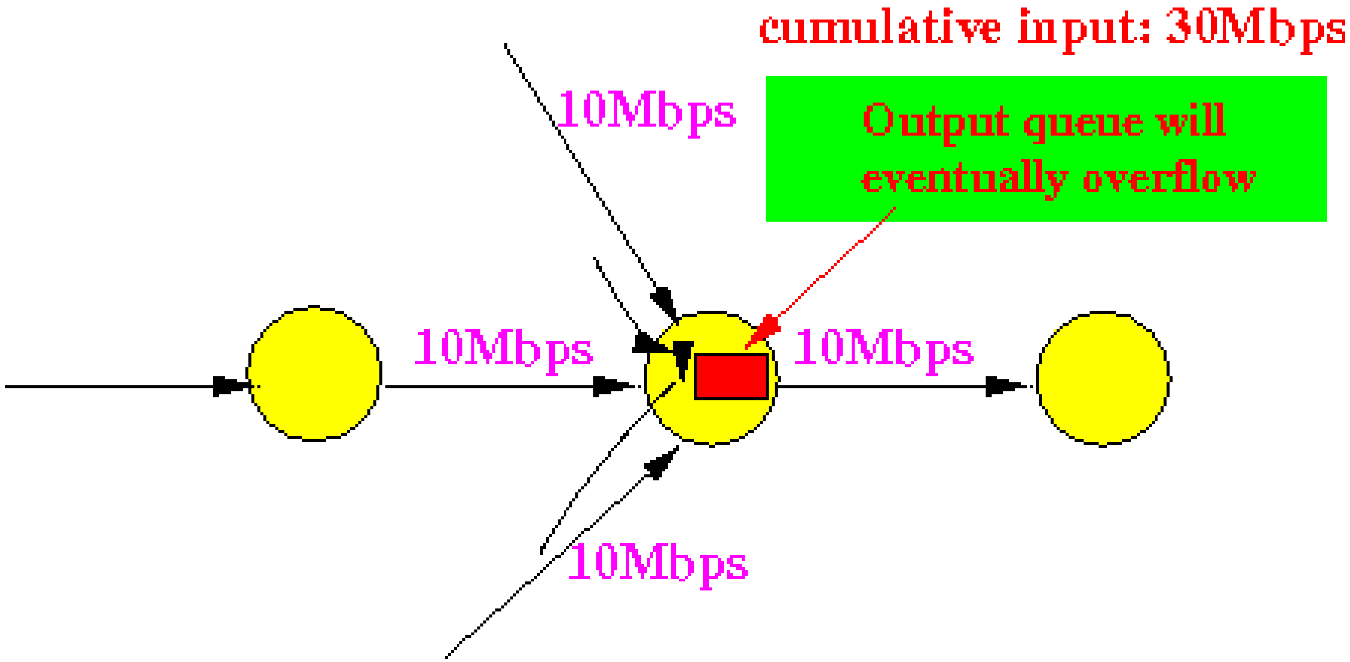
TCP: Flow Control

- **Flow control** = controlling the **transmission rate** at the **source** so that the **packet arrivals** can be **buffered** by the **receiver**
- **Example:**
- **Suppose** the **receiver** can process ***maximum* 1000 packets per second**
- **If** the **sender** transmits ***more* than 1000 packets per second** (for a prolonged period of time), then:
  - The **receiver's buffer** will eventually **overflow**

TCP: Congestion Control

- **Congestion** occurs when the **input rate** exceeds the **output rate** for some **period** of **time**
- **Example 1:** a **slower link**





- **Example 2: concentrated flows**

Solving Congestion Problem

- The **answer** on **how to solve** the **congestion problem** is ***easy***
  - **Reduce** the **transmission rate**....

- **Unfortunately**, we do ***not* know** the **answers** to

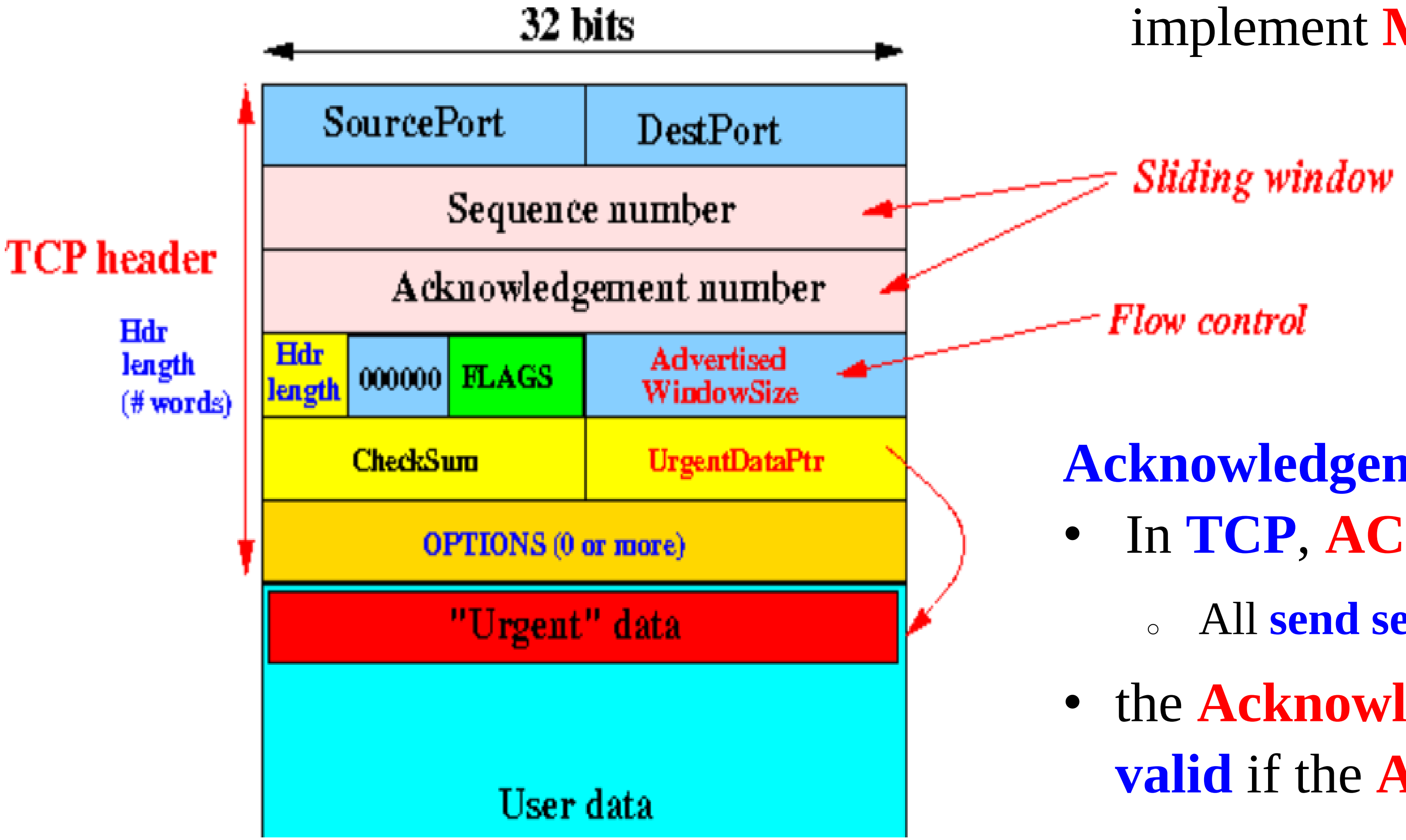
- ***When*** do we **reduce** the **transmission rate**?
  - **Congestion *detection*** !!!

- **How much** ???
  - **Transmission rate *adjustment***

- If we **reduce** the **transmission rate** by **too much**, the **bandwidth** will be ***wasted* (idle !!!)**

TCP: Message Format

- **TCP message** and **header format**

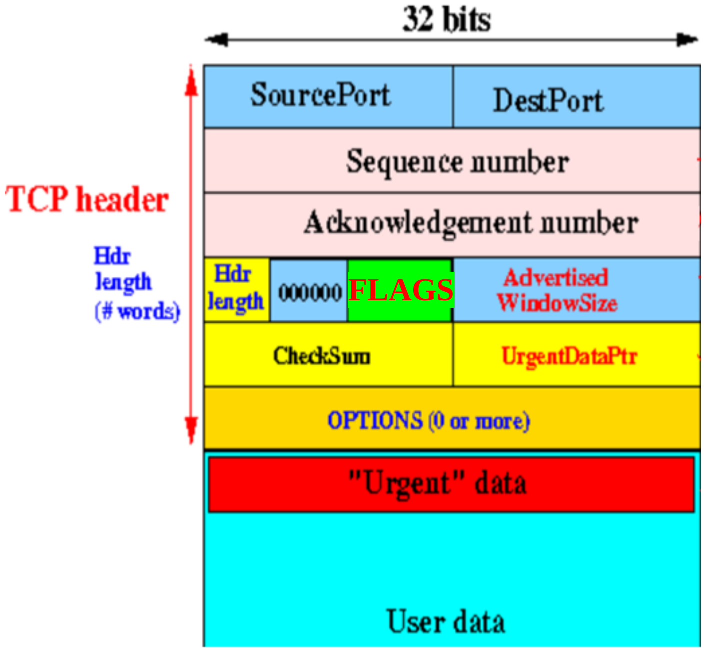


- **Source Port and Dest Port** are used to implement **Multiplexing** just like **UDP**

Acknowledgement Number:

- In **TCP**, **ACK(n)** will **acknowledge**
  - All **send sequence numbers** that are **< n**
- the **Acknowledgement Number** is ***only* valid** if the **ACK flag bit** is **set**

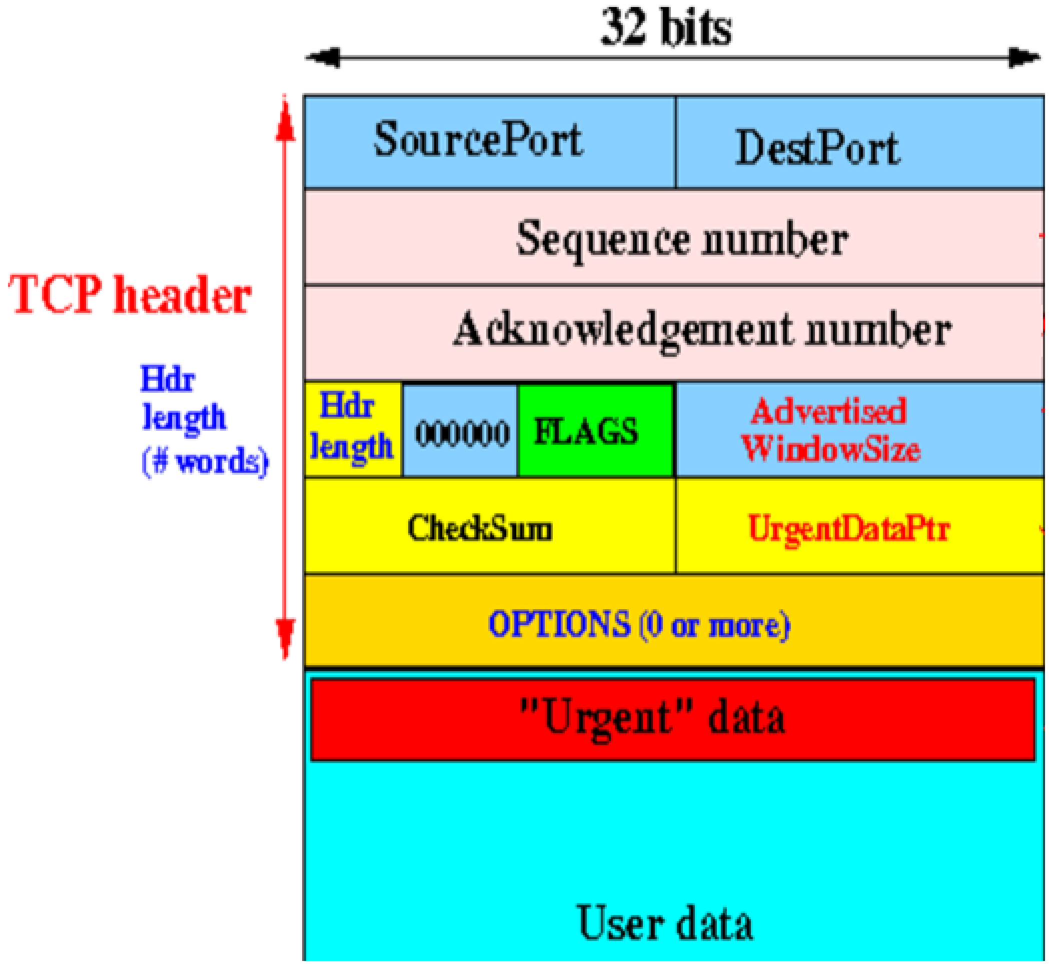
TCP: Message Format



- **FLAGS**: used to **indicate** a **TCP control message**
- **SYN flag:** (**synchronize**)
  - **SYN** is **set** when the **TCP source** wants to ***establish* a connection**
- **FIN flag:** (**finish**)
  - **FIN** is **set** when the **either TCP party** wants to **close** an existing **TCP connection**
- **RESET flag:** (**reset**)
  - **RESET** is used to **abort** a **TCP connection** (Abnormal exit)
- **PUSH flag:** (**flush** the connection)
  - The **PUSH bit** is **set** when the **user application** invoked the ***push* operation** that **flushes/clears** the **transmission buffer**.
- **URG flag:**
  - indicates that **urgent data** is **sent inside** the **TCP payload**
  - The **Urgent pointer** marks the **end** of the **urgent data**

- **ACK flag:** (**Acknowledgement**)
- **If** the **ACK bit** is **set**:
  - The **Ack (receive) sequence number** in the **TCP header** is ***valid***
- **Otherwise**:
  - The **Ack (receive) sequence number** in the **TCP header** is **invalid** and must be **ignored**

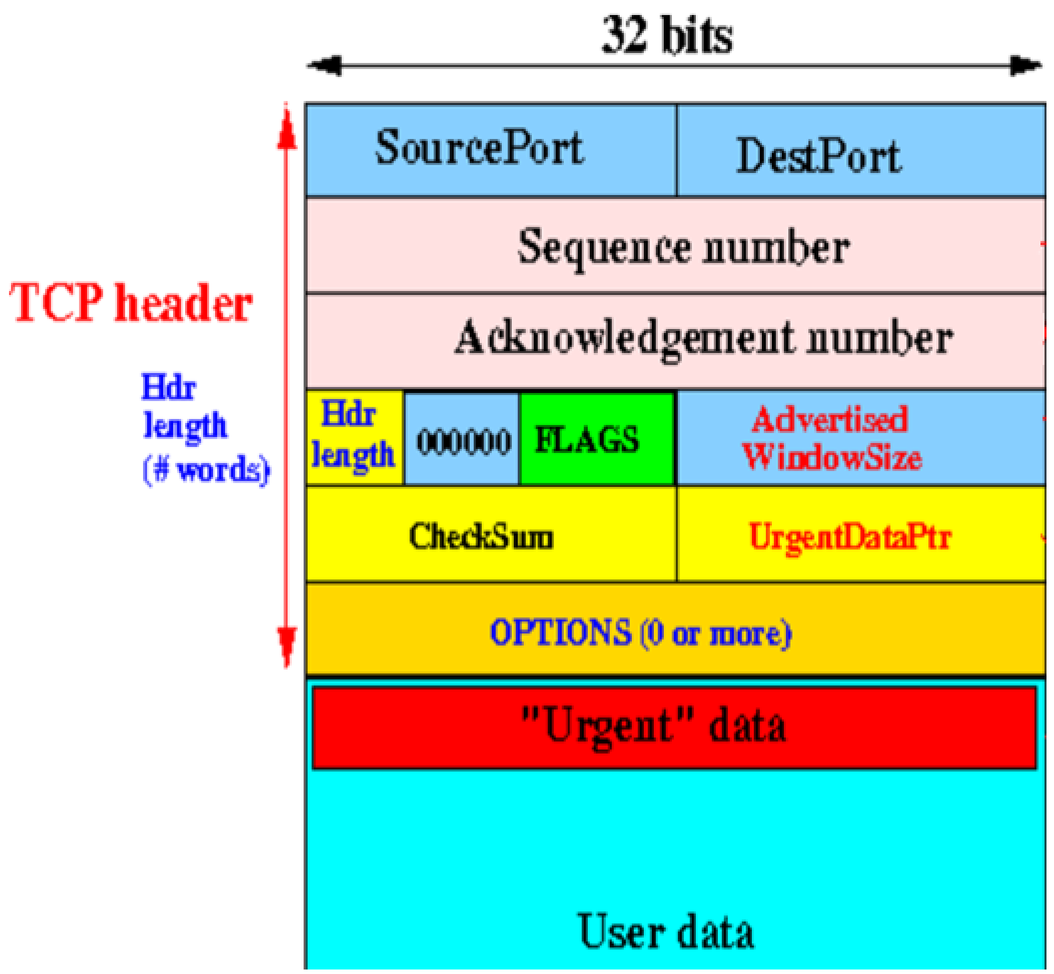
TCP: Message Format



CheckSum: checksums the TCP header & message

- **Sequence Number: Every *item*** that require ***reliable* transmission** must be
  - **identified** by a **unique (send) sequence number**
- **Example:**
  - A **SYN request** (used to **establish a TCP connection**)
  - A **FIN request** (used to **tear down** a **TCP connection**)
  - Each **byte** that the **user application** transmits

TCP: Message Format



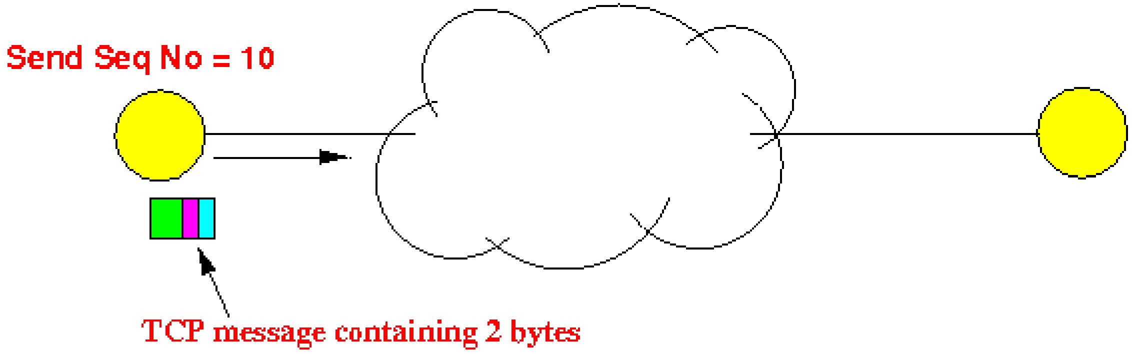
- **Advertised WindowSize:**
  - Use in **flow control**
  - **Advertised window size** = the **many bytes (free space)** that is **available** in the **receiver buffer**
- **Note:**
  - The **sender cannot** transmit **more data** than it is given in the **Adv. window size**

- **UrgentDataPtr:** The **Urgent pointer** marks the **end** of the **urgent data**

Unit of data in the TCP protocol

- **TCP** is a ***byte* oriented protocol**
  - The **unit of data transmitted** in **TCP** is **1 byte**
- **In other words:**
  - **Each byte** is **assigned** a (unique) **send sequence number**
  - **Each byte** will be **acknowledged *individually***
- It's **not** too **bad** since **TCP** uses ***cumulative* ACK**

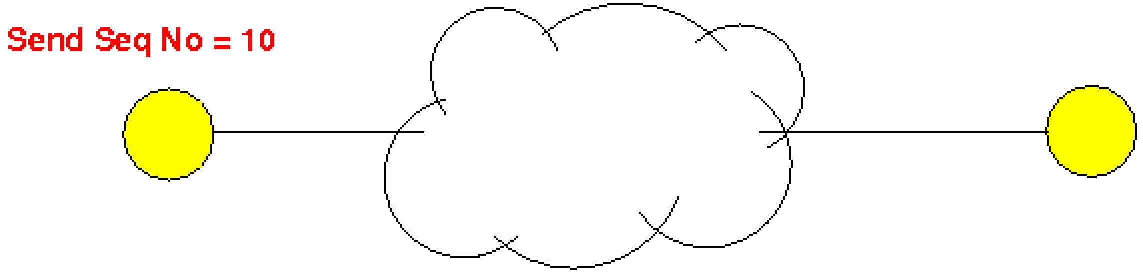
2. The **sender transmits 1 (one) TCP message** contains **2 bytes**:

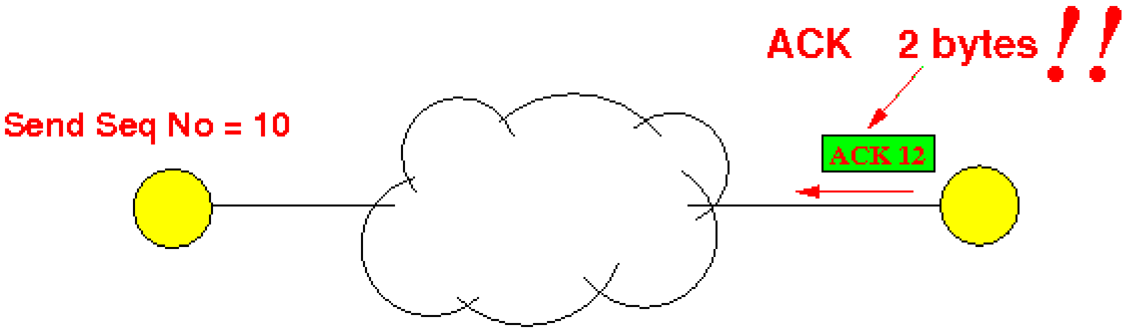


3. The **receiver** must **acknowledge** the reception of ***2* bytes**

Example:

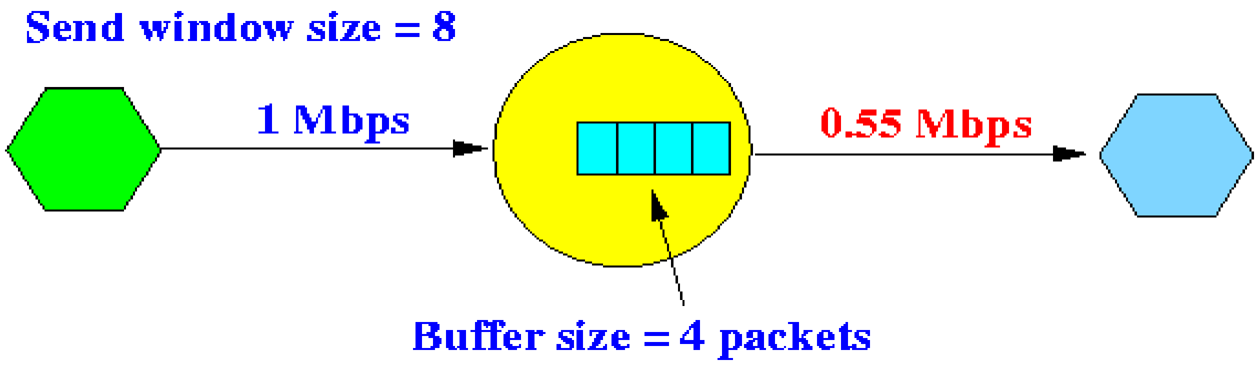
1. Suppose the **send sequence number = 10**:

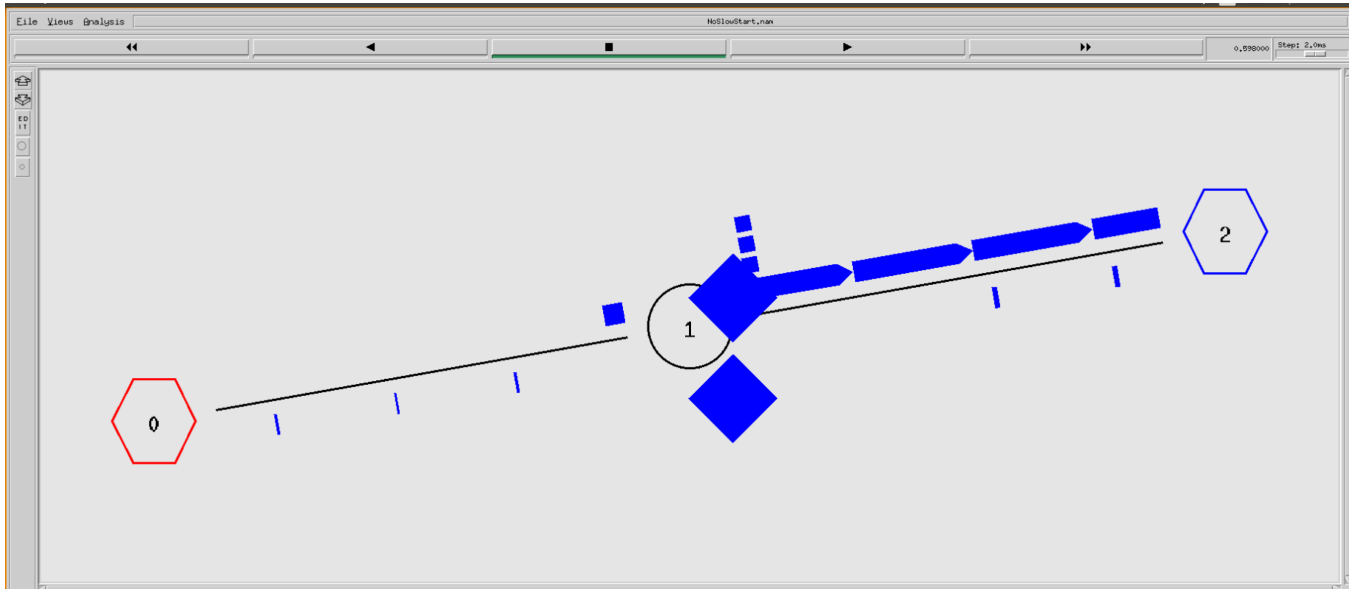




Congestion, Buffer overflow and Packet Drop

- **Example Network**





- **What is the** **reason** **of packets drop**

Overview-Protocols in the TCP/IP Suite

Any question?

Overview/roadmap:

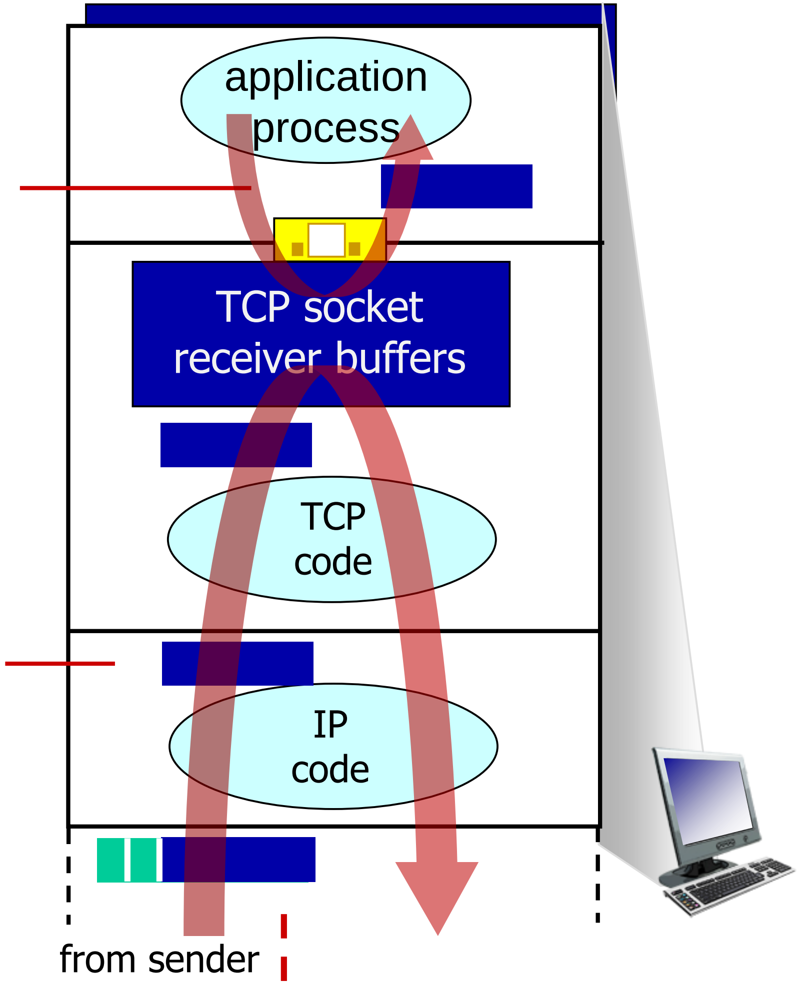
- So far, we talked about
  - TCP Services
  - TCP Message Format
  - Congestion Buffer Overflow
  - Next we discuss
  - Connection-oriented transport: TCP
    - TCP flow Control

TCP flow control

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

Application removing data from TCP socket buffers

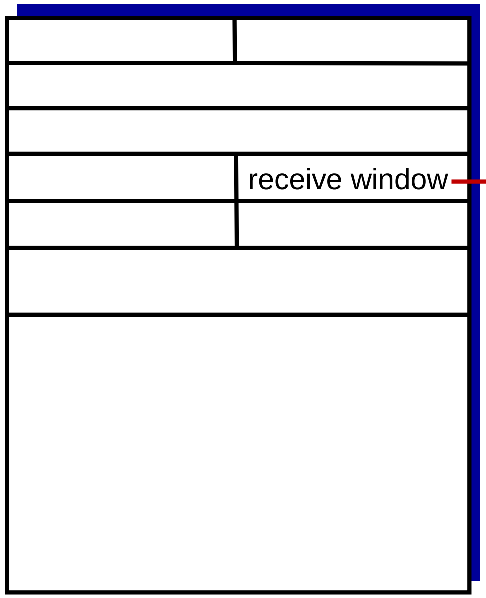
Network layer delivering IP datagram payload into TCP socket buffers



receiver protocol stack

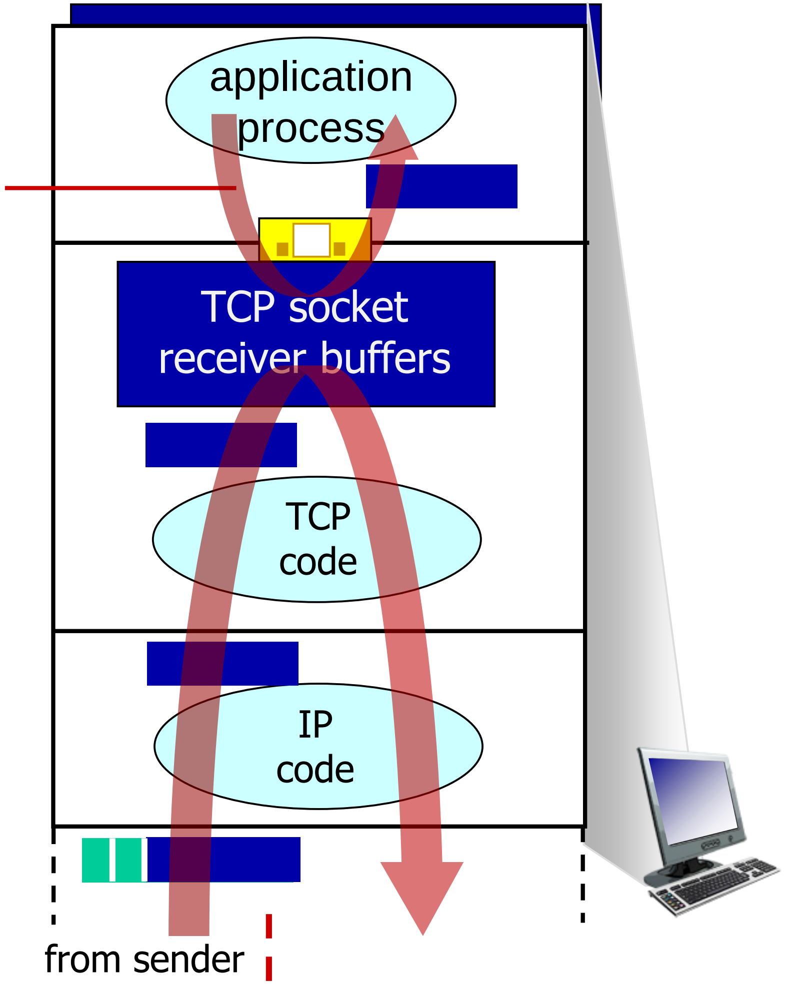
TCP flow control

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

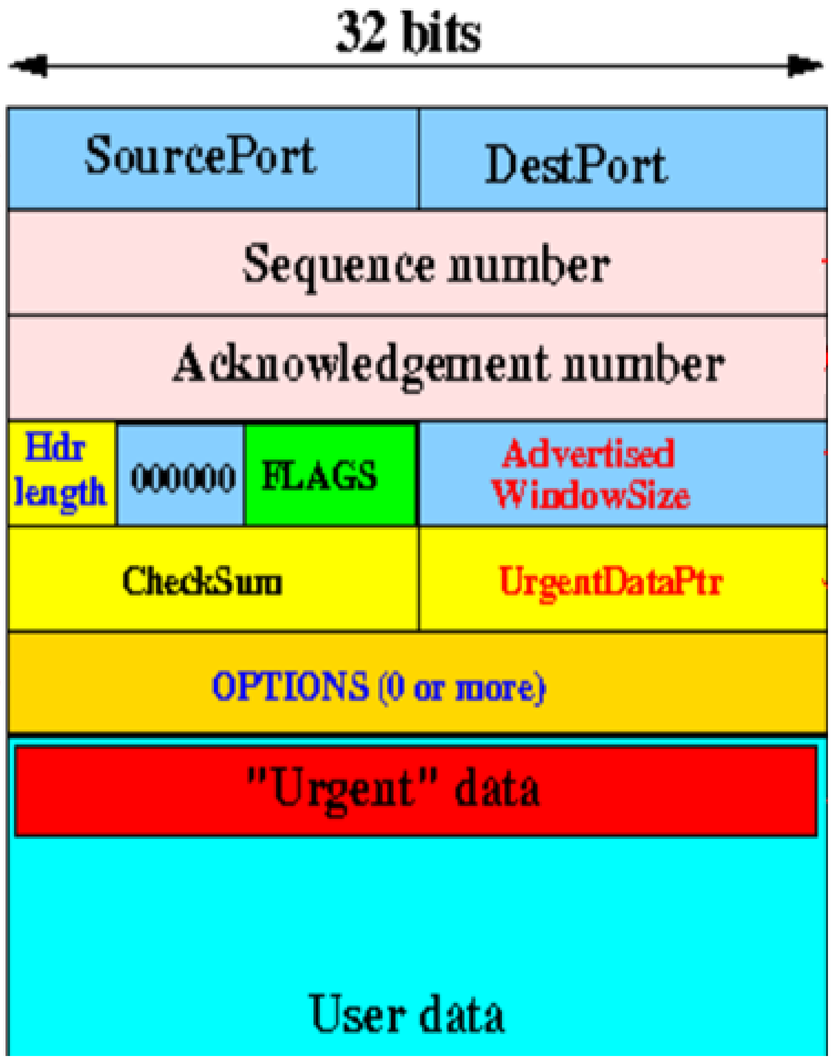


receiver protocol stack

TCP flow control

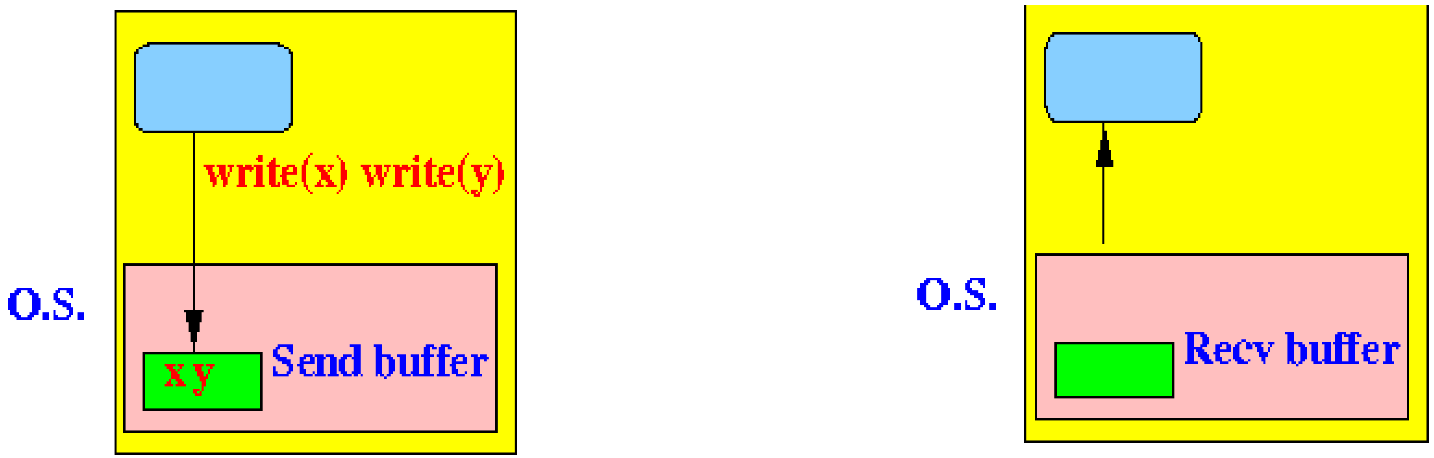
- TCP receiver "advertises" free buffer space in **`rwnd`** field in TCP header
- sender limits amount of unACKed ("in-flight") data to received **`rwnd`**
- guarantees receive buffer will not overflow

flow control: # bytes receiver willing to accept



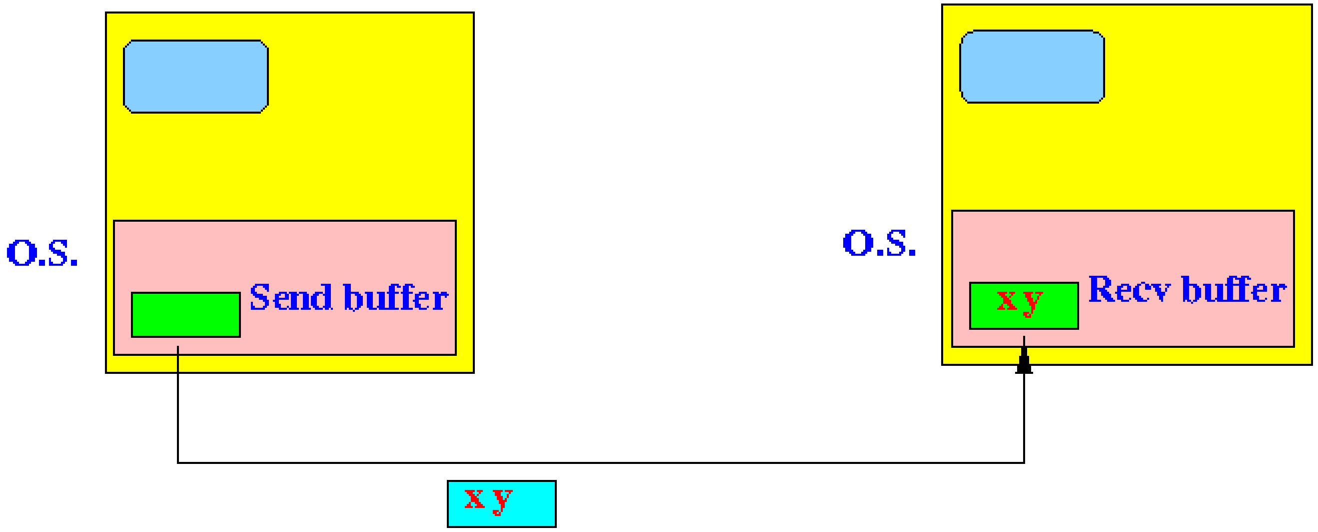
Flow Control in TCP- Send and Receive buffers

- When a **user application sends data** to another **user application**
- the **data** is ***first* stored in a *send buffer*** inside OS kernel



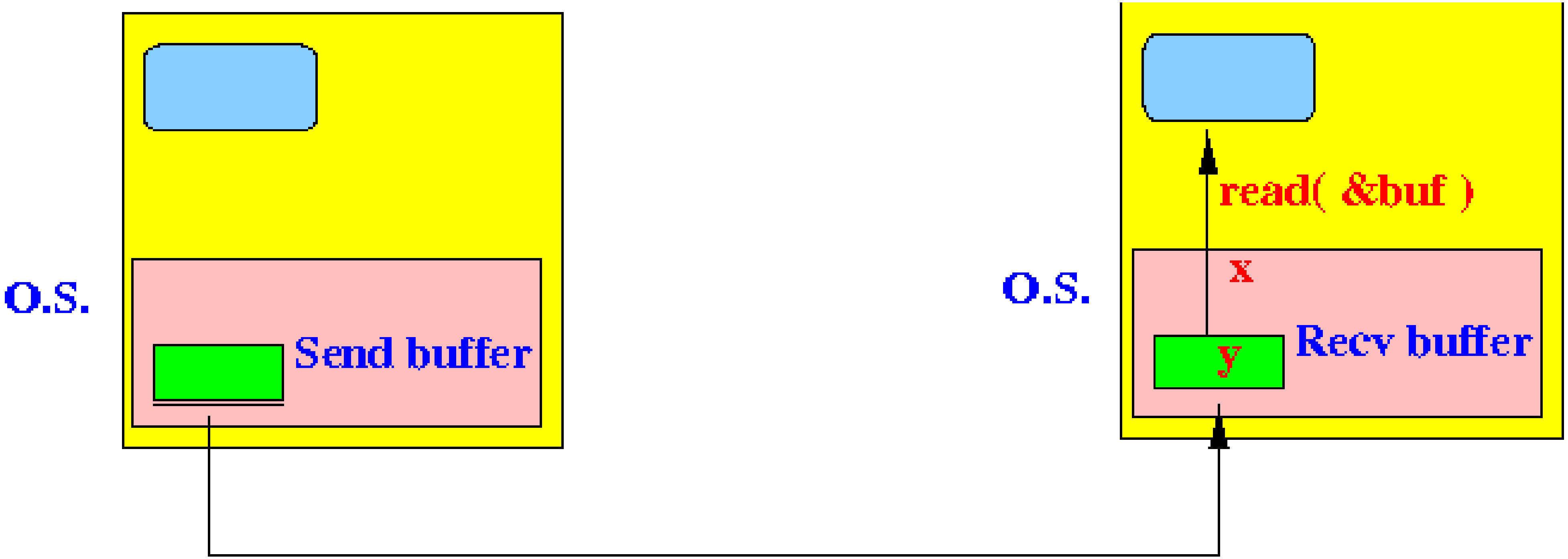
- When the **send buffer** contains **more than MSS bytes** or when the **send time *expires***, the **TCP module** will **send the data** in a **TCP segment** to the receiver

- The **received data** is stored in a **receive buffer** inside the OS kernel of the receiver computer



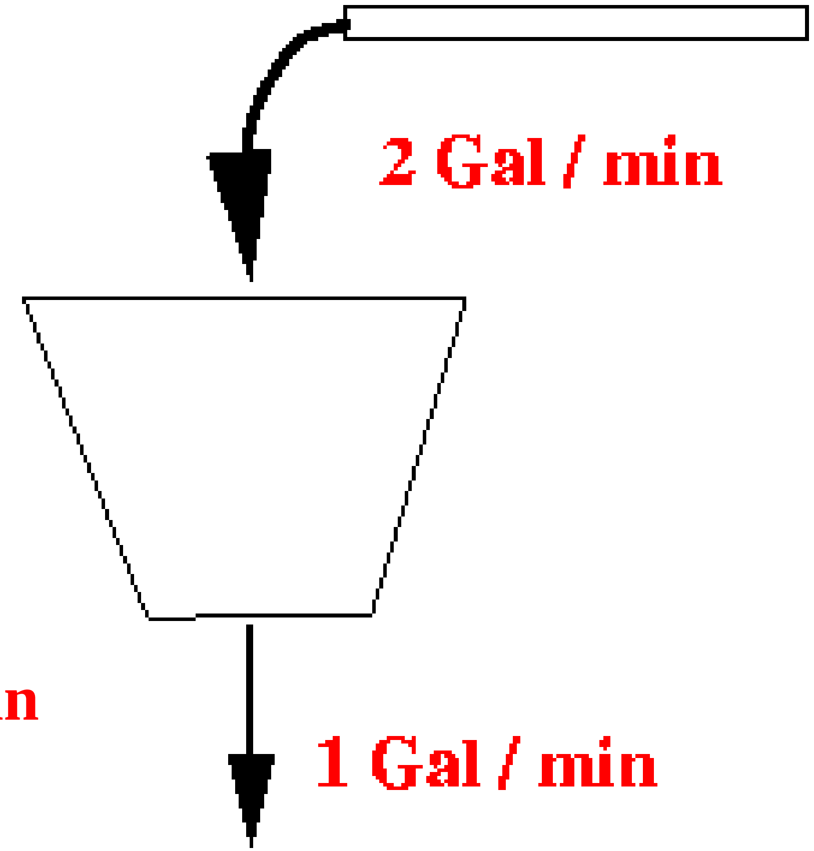
Flow Control in TCP- Send and Receive buffers

- The **receiving application** can use the **read( ) system call** to read the **received data** from the **receive buffer**
  - **The receive buffer becomes empty**



Flow Control in TCP- Flow Control

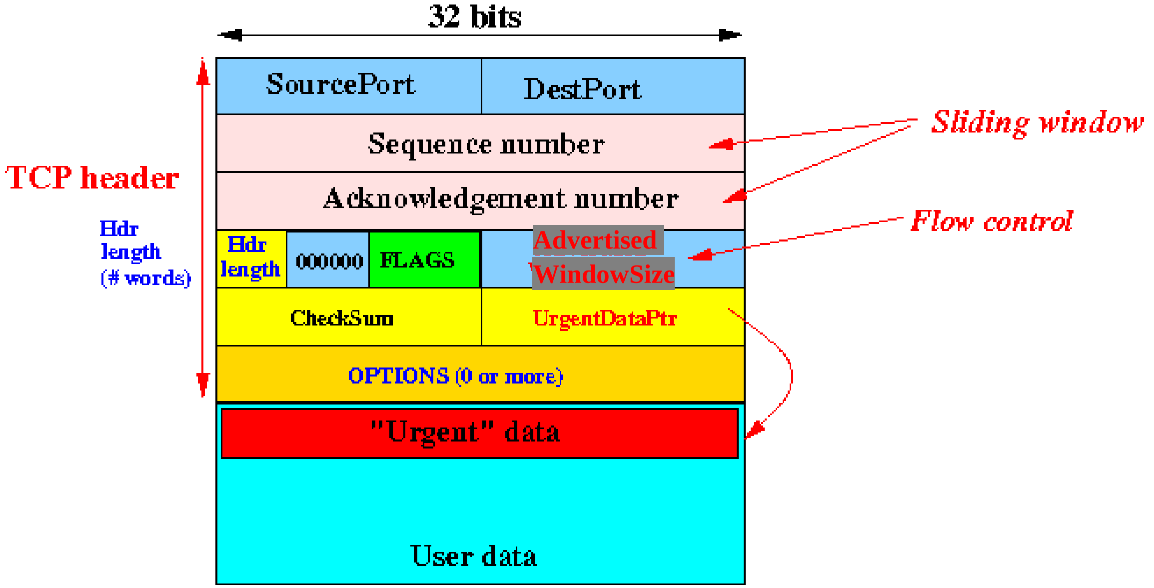
- **Flow control** is a **mechanism** to ensure that the **transmission rate** of the **sender do *not* exceed** the **capacity** of the **receiver**
- Fact:
- If the **transmission rate *exceeds* the capacity of the receiver**, then **sooner or later**, there is **no buffer space left** to store **arriving packets**
  - **When this happens, newly arriving packets are discarded**
- **ANALOGY**
- The bucket has a **hole** that let **water drain** at **1 Gal/min**
- A pipe fills the bucket with water at **2 Gal/min**
- Sooner or later, the bucket will **overflow**.



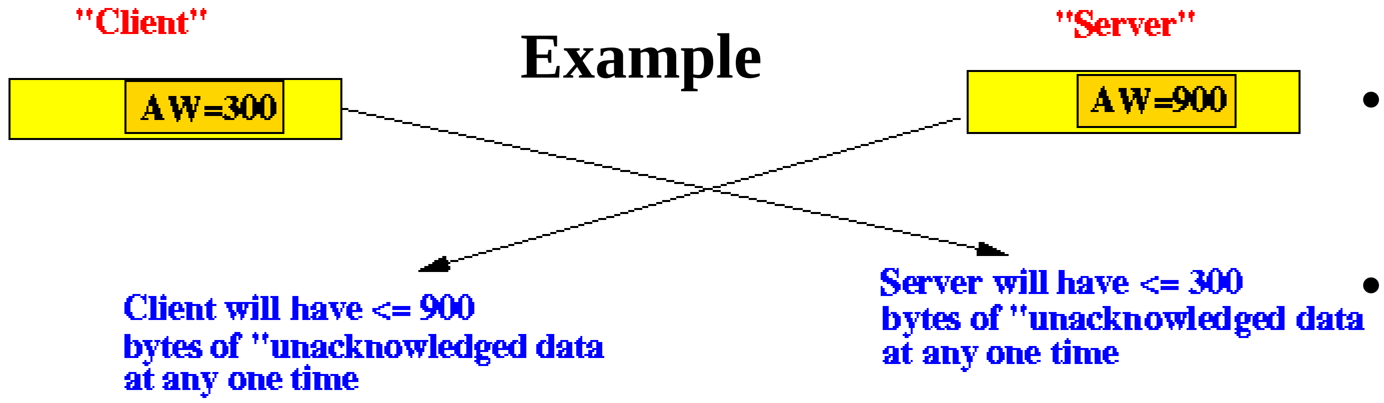
Flow Control in TCP- Implementation

- **Flow control** is implemented with the **Advertised Windowsize** in the **TCP header**

- **Using the Advertised Window Size**
  - The **adv. win. size** is the **maximum number of bytes** that the **receiver** is willing to **buffer**
  - The **sender** should **honor** the **receiver's request** and **refrain from sending *more* data** than the given **adv. win. size**



Example



- The **client** tells the **server *not* to send more than 300 bytes**
- The **server** tells the **client *not* to send more than 900 bytes**

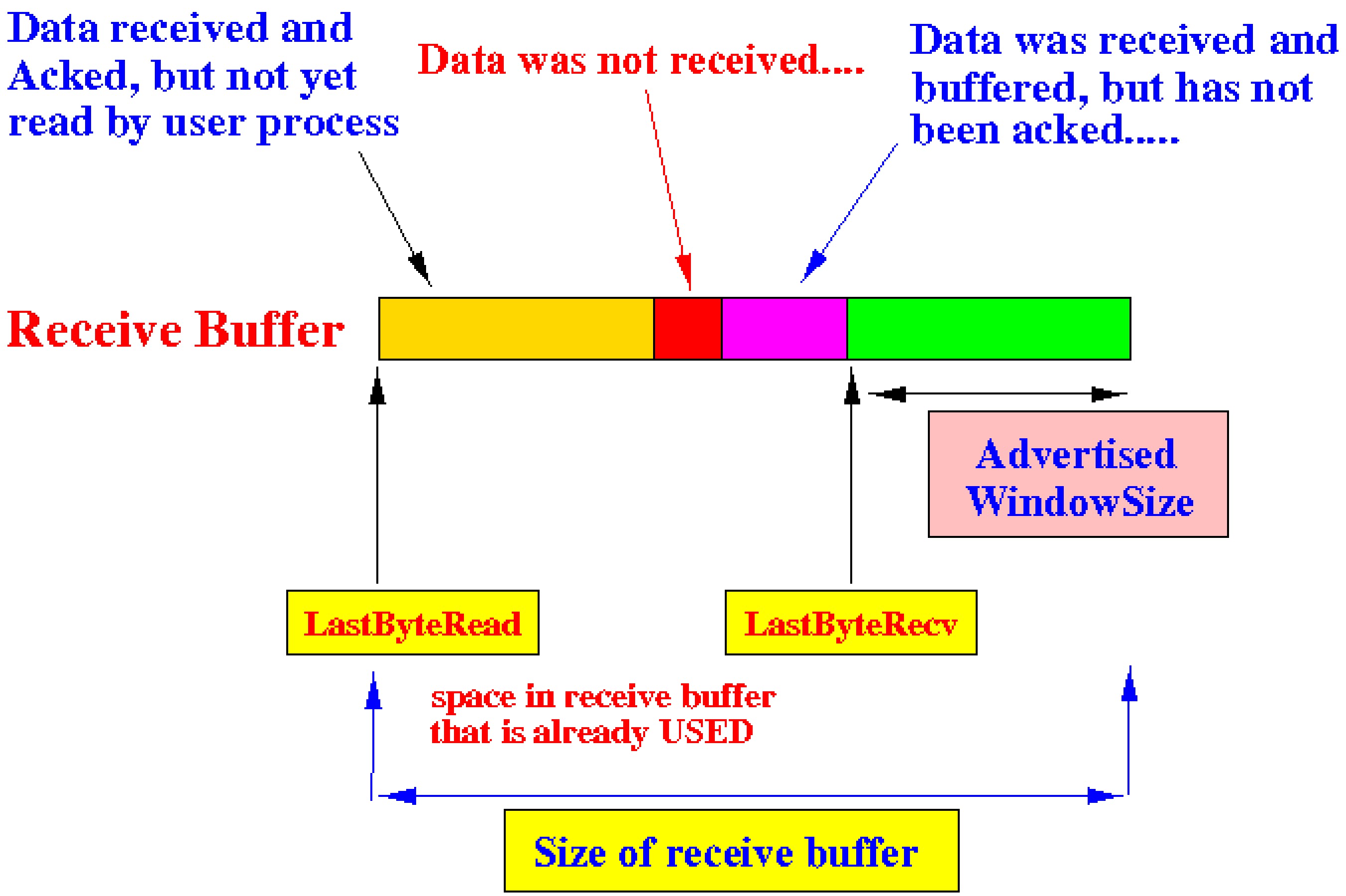
Receive buffer and Advertised Window Size

- **Factors** that determine the **Advertised Window Size**
  - The **size** of the **receive buffer** that the **TCP connection** has reserved
  - The **receive buffer size** is **fixed** at **connection establishment**
  - The **amount of data** currently **buffered** in the **receive buffer**

Receive buffer

- The **receive buffer** is located ***inside* the Operating System (kernel)**.
- The receive buffer is filled when a data segment is received
- The receive buffer is emptied (a little bit) when data is read by the user process

Computing the Advertised Window Size



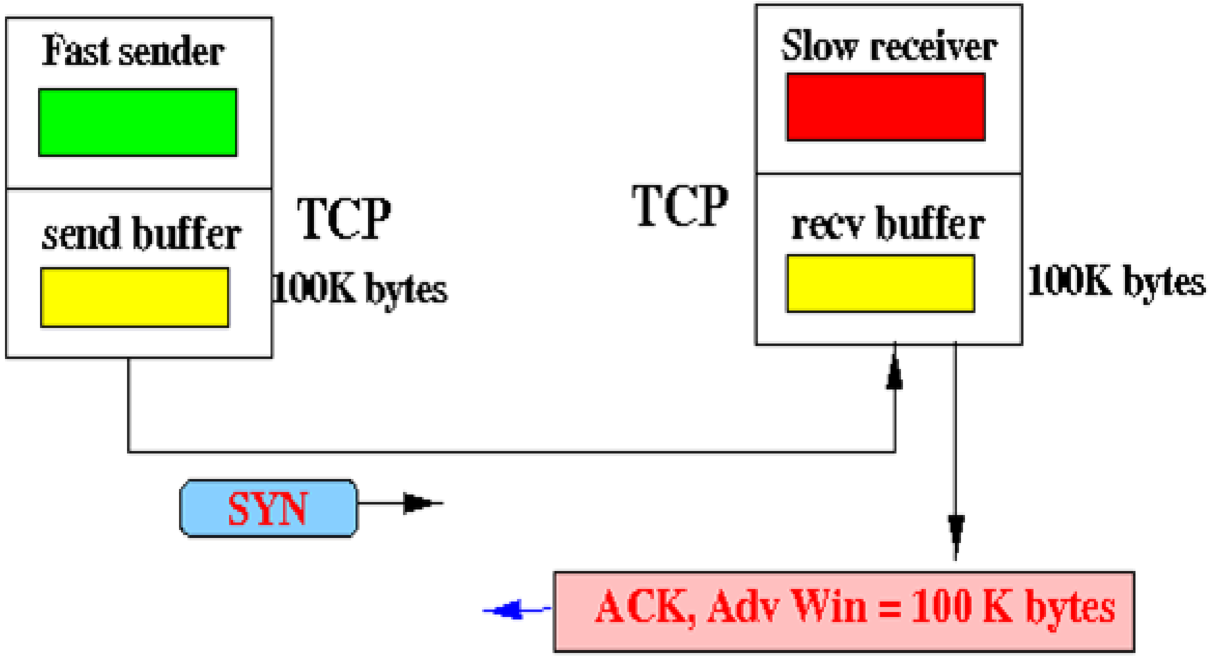
- Clearly, the **Advertised WindowSize** should be set to

Advertised Window = SizeRecvBuffer - (LastByteRecv - LastByteRead)

Flow Control in Action

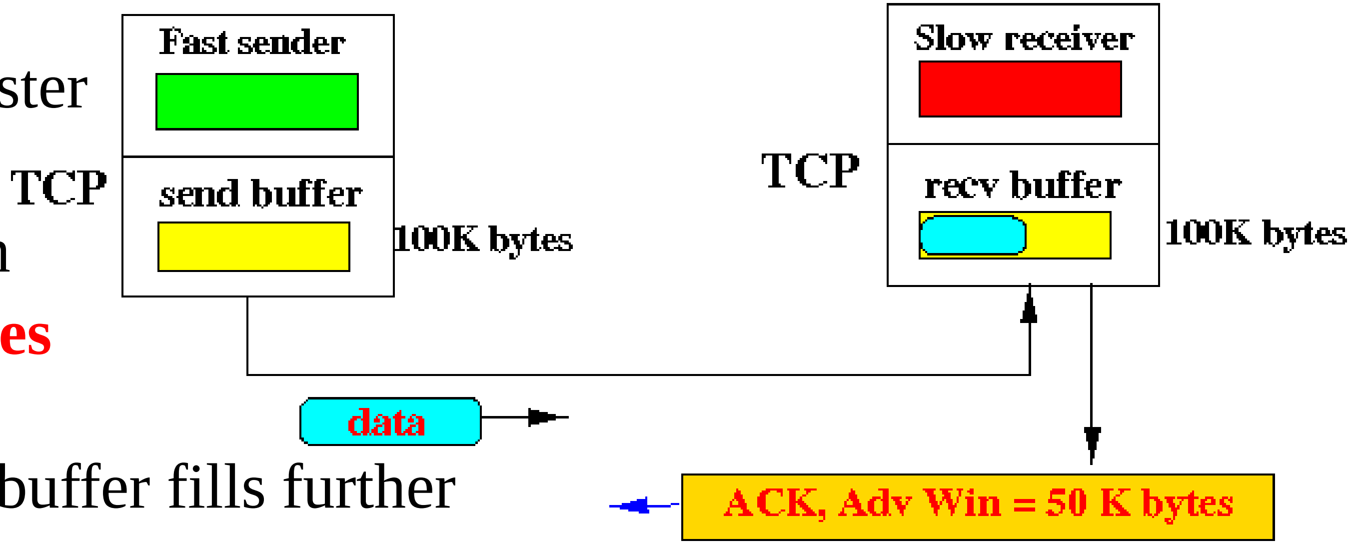
- The **effect** of **flow control** is **only evident** when a **speedier sender** transmits to a **slower receiver**
- The following example of **flow control** uses a **receive buffer size of 100 K bytes**.

- **Example**
  - Initially, the send and receive buffer are empty.
  - Receiver advertises window of **100 Kbytes**

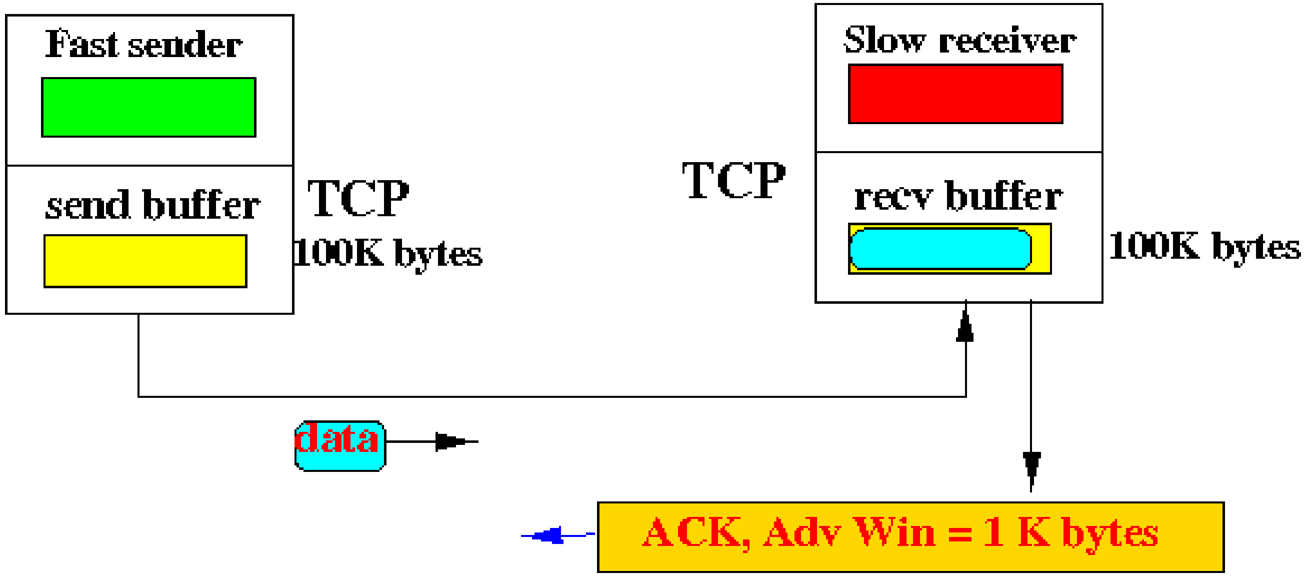


Flow Control- Example

- Sender sends many packets quickly, faster than the receiver can process
- the receive buffer starts to fill and soon the advertised window drops to **50 Kbytes**



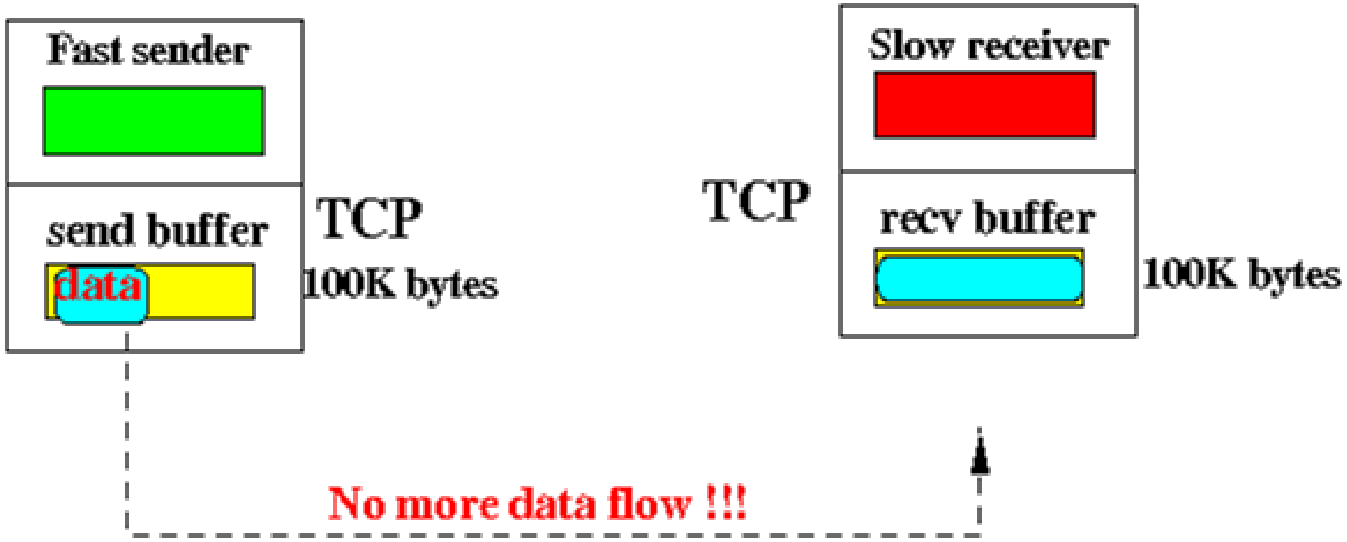
- The situation continues and the receive buffer fills further
- The ACK packets from the receiver will now have a **lower advertised window**, e.g., **1 K bytes**



- Sooner or later, the **receive buffer fills up**

Flow Control- Example Continued …

- The receiver returns an ACK packets with the **advertised window = 0 bytes**

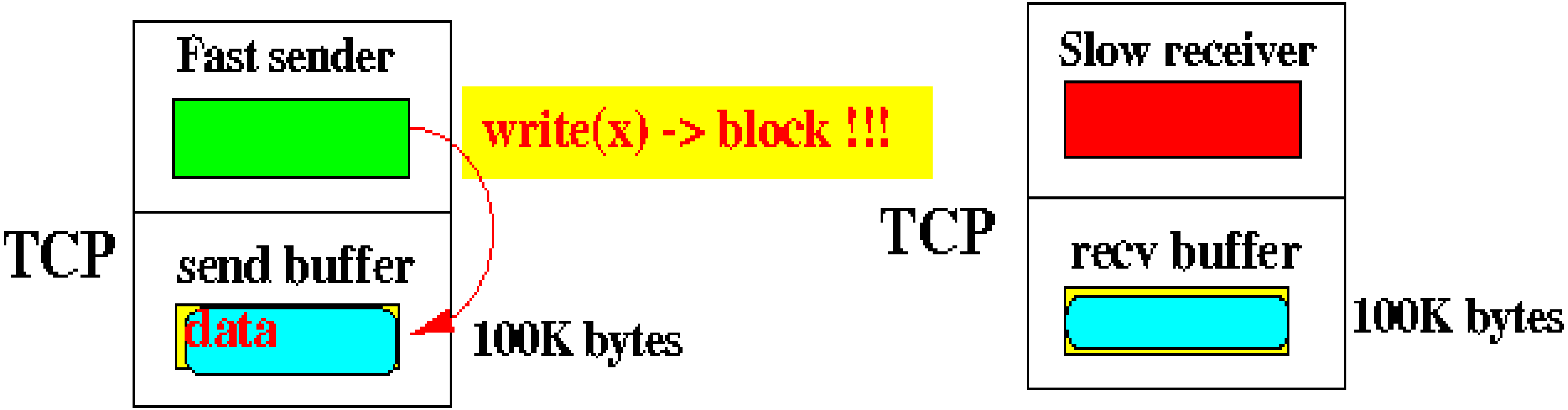


- This causes the sending TCP to stop transmitting more data
- and prevented the sending TCP to overflow the receive buffer of the receiving TCP.

- **NOTE**
  - This will not stop the **sending application** from sending more data
  - The **sending application process** can still send more data
  - but the data sent will **remains in the send buffer** (as given in the above figure)

Flow Control- Example Continued ...

- If the **sending application process** continues sending, the **send buffer will fill up**
- A **subsequent `write()` call** will cause the **sending application process** to **block**



- Now the **faster sending application process** has been **successfully throttle**...

Thanks