Computer Network I
Reti di Calcolatori I

Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Informatica

Riccardo Caccavale
(riccardo.caccavale@unina.it)

Università degli Studi di Napoli Federico II

Transport Layer
From Application to Transport

- A transport-layer protocol mostly provide **logical communication between application processes** running on different hosts.
  - From the application's perspective, hosts running **the processes looks directly connected** as if they were on the same machine (even if they are on opposite sides of the planet).

- The **transport layer converts the application-layer messages into** one or more transport-layer packets, called **segments** or **datagrams** (the latter is mainly used for UDP packets):
  - If application messages are broken into smaller chunks (segments/datagrams), **each chunk is provided with transport-layer header**.
  - Segments are **passed one by one to the network layer** to be transmitted.

Transport Layer

Responsibilities of Transport Layer

- Transport-layer protocols (UDP and TCP) have four main responsibilities:
  1. **Process-to-process delivery:** messages are delivered independently of where these processes are.
     - Note that this is **different from host-to-host delivery** (i.e., between two different machines), which is responsibility of the lower-level IP protocol.
  2. **Integrity checking:** by including error detection fields into the segments' headers.
  3. **Reliable data transfer**: ensuring that data is delivered from sending process to receiving process, correctly and in order.
  4. **Congestion/flow control**: it prevents connection from swamping network devices (links or routers) with an excessive amount of traffic. This service is useful to improve performance and **is very beneficial to the whole (internet) network**.

- In particular, UDP (faster but simpler) provides **only the first two services**, while TCP provides all the four ones.

Transport Layer
Process-to-process Delivery

- At the application level, **several processes can be accessing the network at the same time**.
    - For example: listening to Spotify music, downloading files, navigating web sites, all together.

- A **process may also be connected to multiple processes** at the same time:
    - For example: proxy servers, mail servers, etc., all may be connected to multiple remote processes.

- To solve these issues, we use sockets for process-to-process delivery:
    - Instead of delivering the message directly to a specific process, we **use sockets as end-points from which processes get the messages**.
    - This approach somehow **decouples** the number of processes from the number of connections.

Transport Layer
Process-to-process Delivery

- Since there are multiple sockets in the receiving host, we need:
  - A **unique identifier** for each socket (**port**) that must be attached to the received segments.
  - A **multiplexing/demultiplexing** process on the sender/receiver.

- **Demultiplexing**: is the process of redirecting a segment to the right socket depending on the associated identifier.

- **Multiplexing**: is the process of creating segments from different processes, assigning to them the right identifier.

- Notice that the problem of redirecting messages to multiple sources is quite common in computer networks, multiplexing and demultiplexing are also present in other layers (not only in transport).

Transport Layer

Ports

- The identifiers of sockets are called source and destination ports (or simply ports):
  - A **port is a 16-bit number** (port number) ranging from 0 to 65535.
  - The port numbers ranging **from 0 to 1023 are called well-known port numbers** and are **restricted**, which means that they are reserved for use by well-known application protocols.
- The **list of well-known port** numbers is updated by **IANA** (Internet Assigned Numbers Authority), available at http://www.iana.org.
- When we develop a new network application, we must assign port numbers to sockets (and therefore to the applications) accordingly.

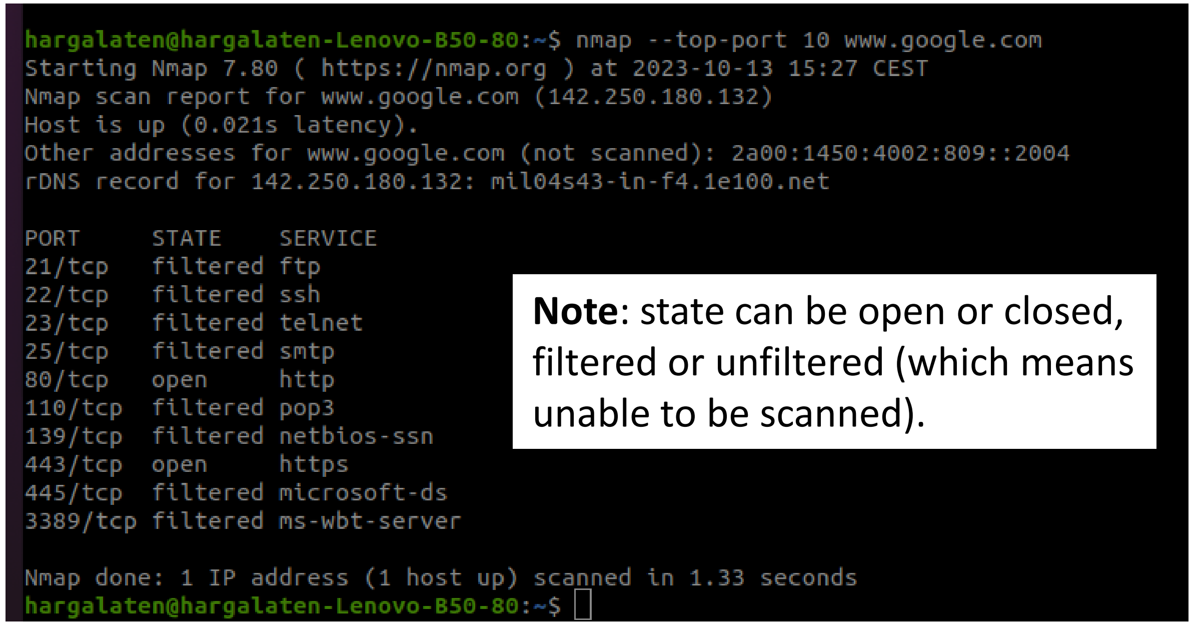
Port	Usage
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH)
23	Telnet - Remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) E-mail Routing
53	Domain Name System (DNS) service
80	Hypertext Transfer Protocol (HTTP) used in World Wide Web
110	Post Office Protocol (POP3) used by e-mail clients to retrieve e-mail from a server
119	Network News Transfer Protocol (NNTP)
123	Network Time Protocol (NTP)
143	Internet Mail Access Protocol (IMAP) Management of Digital Mail
161	Simple Network Management Protocol (SNMP)
443	HTTP Secure (HTTPS) HTTP over TLS/SSL

Transport Layer

Ports: Nmap

- To **check port usage** (and more) on a Linux machines we can use the nmap command.
- Nmap (Network Mapper) is a **network scanning utility** basically created to map (discover) hosts and services on a network.
- It can be used to probe ports of hosts **in order to detect open ones** (on which an application is listening) and possibly the associated services.

- Usage:
  - Port-scan of a given target:
    - $ sudo nmap [target address]
  - Probe port to determine services:
    - $ sudo nmap –sV [target address]
  - Check first N top used ports:
    - $ sudo –top-port [N] [target address]



```
hargalaten@hargalaten-Lenovo-B50-80:~$ nmap --top-port 10 www.google.com
Starting Nmap 7.80 ( https://nmap.org ) at 2023-10-13 15:27 CEST
Nmap scan report for www.google.com (142.250.180.132)
Host is up (0.021s latency).
Other addresses for www.google.com (not scanned): 2a00:1450:4002:809::2004
rDNS record for 142.250.180.132: mil04s43-in-f4.1e100.net

PORT     STATE    SERVICE
21/tcp   filtered ftp
22/tcp   filtered ssh
23/tcp   filtered telnet
25/tcp   filtered smtp
80/tcp   open     http
110/tcp  filtered pop3
139/tcp  filtered netbios-ssn
443/tcp  open     https
445/tcp  filtered microsoft-ds
3389/tcp filtered ms-wbt-server

Nmap done: 1 IP address (1 host up) scanned in 1.33 seconds
hargalaten@hargalaten-Lenovo-B50-80:~$
```

Note: state can be open or closed, filtered or unfiltered (which means unable to be scanned).

Transport Layer
Multiplexing and Demultiplexing

- Assume we have a process in Host A (port 19157) that wants to send a message to a process (port 46428) in Host B.

- Multiplexing:
  - The transport layer in **Host A creates a segment/datagram that includes the application data**, the source port number (19157), the destination port number (46428).
  - The transport layer then **passes the resulting segment/datagram to the network layer** that encapsulates it in an IP datagram, which will make a best-effort attempt to deliver the segment to the receiving host.
- Demultiplexing:
  - If the segment/datagram arrives at the Host B, **the transport layer receives and decapsulates** it.
  - The transport layer then **passes the segment/datagram to the appropriate socket** by examining the segment's destination port number.

Transport Layer

Multiplexing/Demultiplexing in UDP (C/C++)

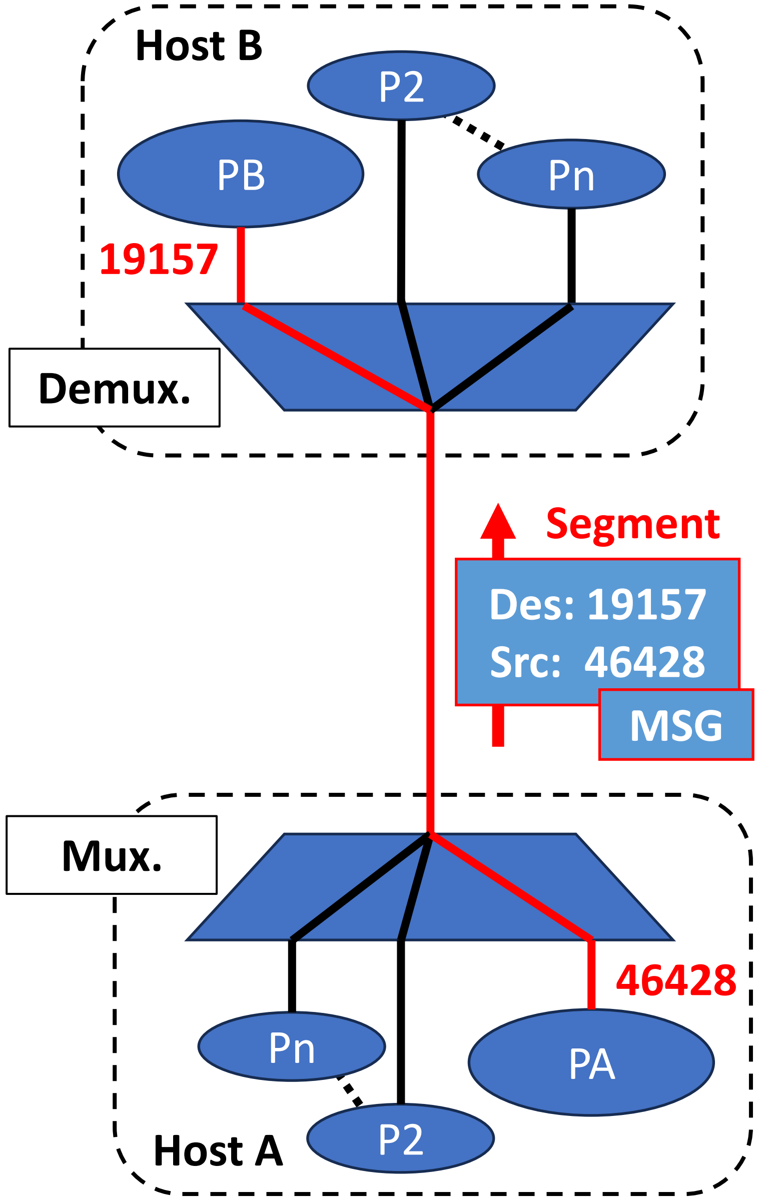
- In a C++ UDP client, socket is created as follows:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
```

- The socket is typically not bound to a specific source port, it is the OS that selects a free one (e.g., 46428) but we can always bind it to a specific port.

- Destination port/address is set by passing a suitable sockaddr_in structure to the sendto function:

```
servaddr.sin_port = htons(19157);
servaddr.sin_addr.s_addr = inet_addr( address_of_B );
...
sendto(sockfd, (const char *)msg, strlen(msg), 0,
            (const struct sockaddr *) &destaddr,
            sizeof(destaddr);
```



Transport Layer

Multiplexing/Demultiplexing in UDP (C/C++)

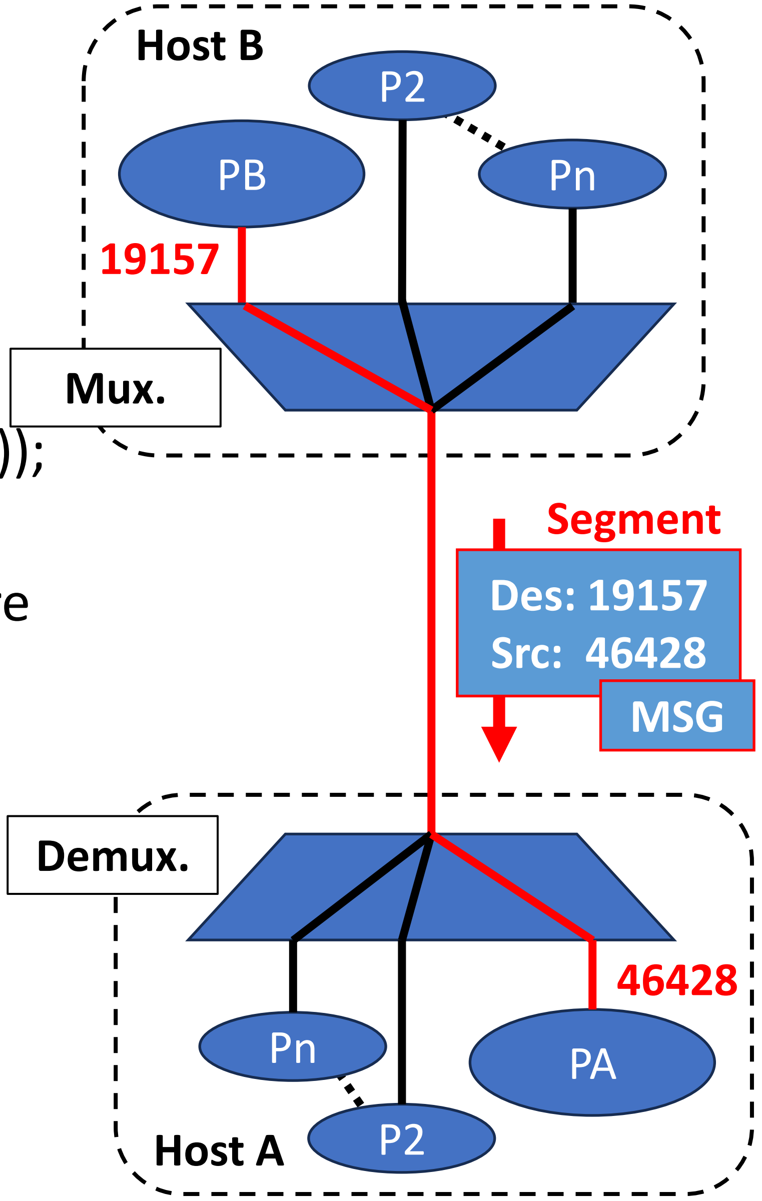
- In a C++ UDP server, socket is created and bound as follows:

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
…
servaddr.sin_addr.s_addr = INADDR_ANY;
servaddr.sin_port = htons(19157);
…
bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr));
```

- Destination port/address is retrieved through the sockaddr_in structure from the recvfrom function and used in the reply message:

```
recvfrom(sockfd, (char *)msg, 1024, 0,
         ( struct sockaddr *) &cliaddr, &len);
…
sendto(sockfd, (const char *)msg, strlen(msg), 0,
         (const struct sockaddr *) &cliaddr, sizeof(cliaddr);
```

- The reply will be sent back **whatever the client is**.



Transport Layer
Multiplexing/Demultiplexing in UDP

- We can say that a UDP socket can be identified **by two elements**:
  - Destination address (IP).
  - Destination port.
- In fact, we can use the same socket to send a return message **back to multiple source ports/addresses**.
- In the example, the cliaddr structure is filled by the recvfrom function, so we can use it as a destination address for the following sendto function, independently on who the host is.

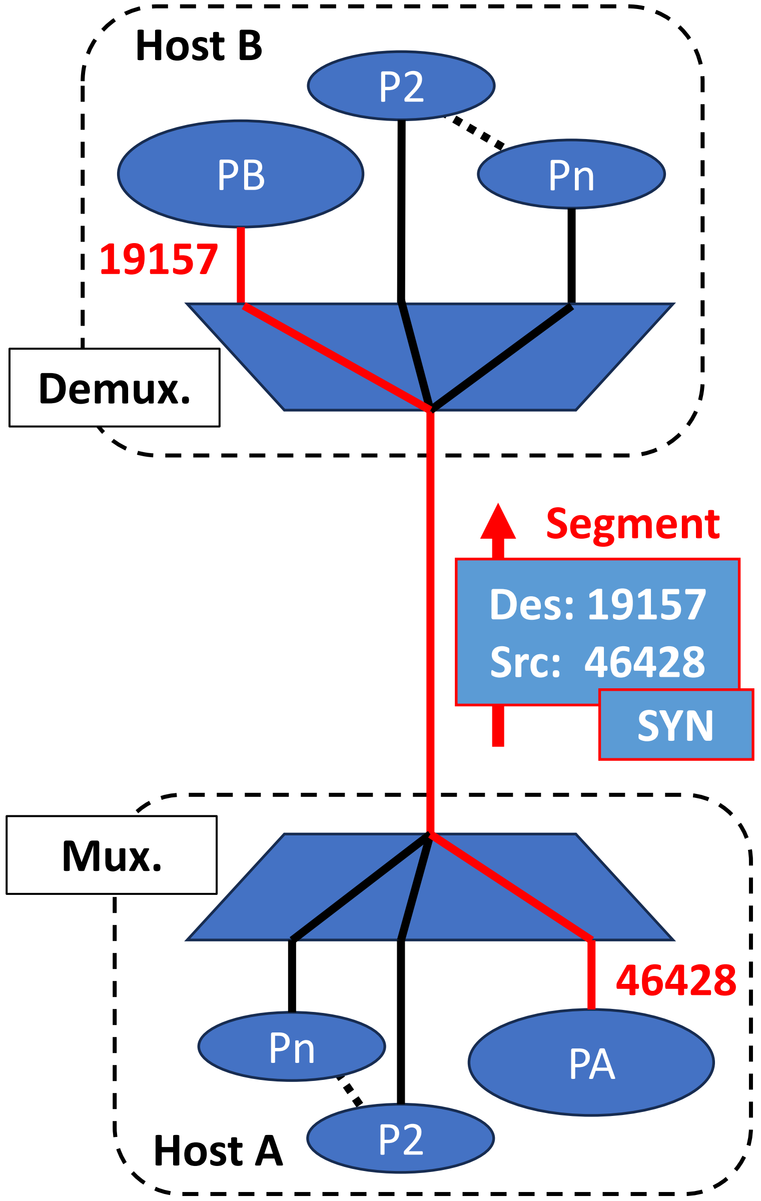
Transport Layer
Multiplexing/Demultiplexing in TCP (C/C++)

- In TCP communication the server application **has a welcoming socket**, that waits for connection establishment requests from clients on a specific port number (19157 as before).

- The TCP client creates a socket and **sends a connection establishment request segment** to the host specified in the sockaddr_in structure (servaddr in this case):

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
…
servaddr.sin_port = htons(19157);
servaddr.sin_addr.s_addr = inet_addr( address_of_B );
…
connect(sockfd, (struct sockaddr*)&servaddr,
            sizeof(servaddr));
```

- A **connection-establishment request is just a TCP segment** with a destination port number (19157 here) and a special connection-establishment bit set in the TCP header (SYN = 1).



Transport Layer

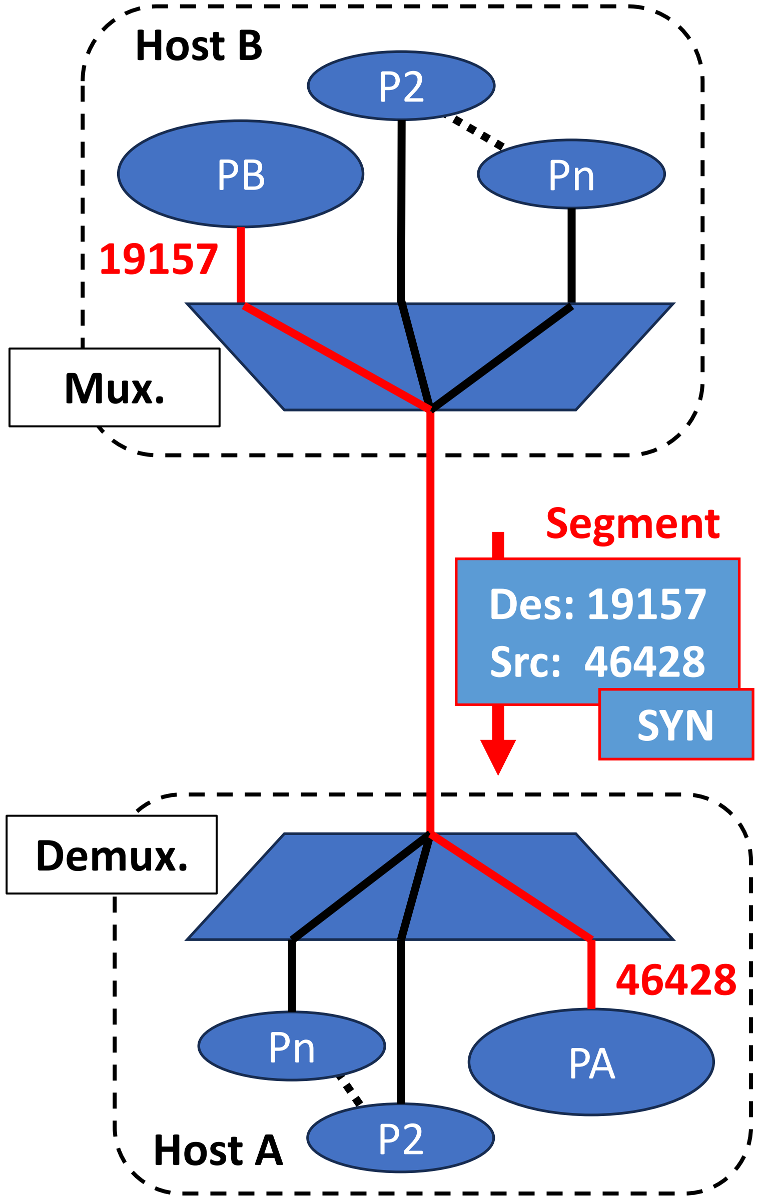
Multiplexing/Demultiplexing in TCP (C/C++)

- When the server receives the incoming connection-request segment (with destination port 19157), it **locates the server process that is waiting** to accept a connection. Then a new socket is created:

```
        new_socket = accept(sockfd,
                (struct sockaddr*)&cliaddr, (socklen_t*)&addrlen)
```

- The server-side transport layer fills the structure (cliaddr) **by using port numbers and addresses from the incoming segment** and creates a new socket (new_socket).

- All future segments having these specific source port, source IP address, destination port, and destination IP address **are redirected (demultiplexed) to new_socket**.



Transport Layer

Multiplexing/Demultiplexing in TCP (C/C++)

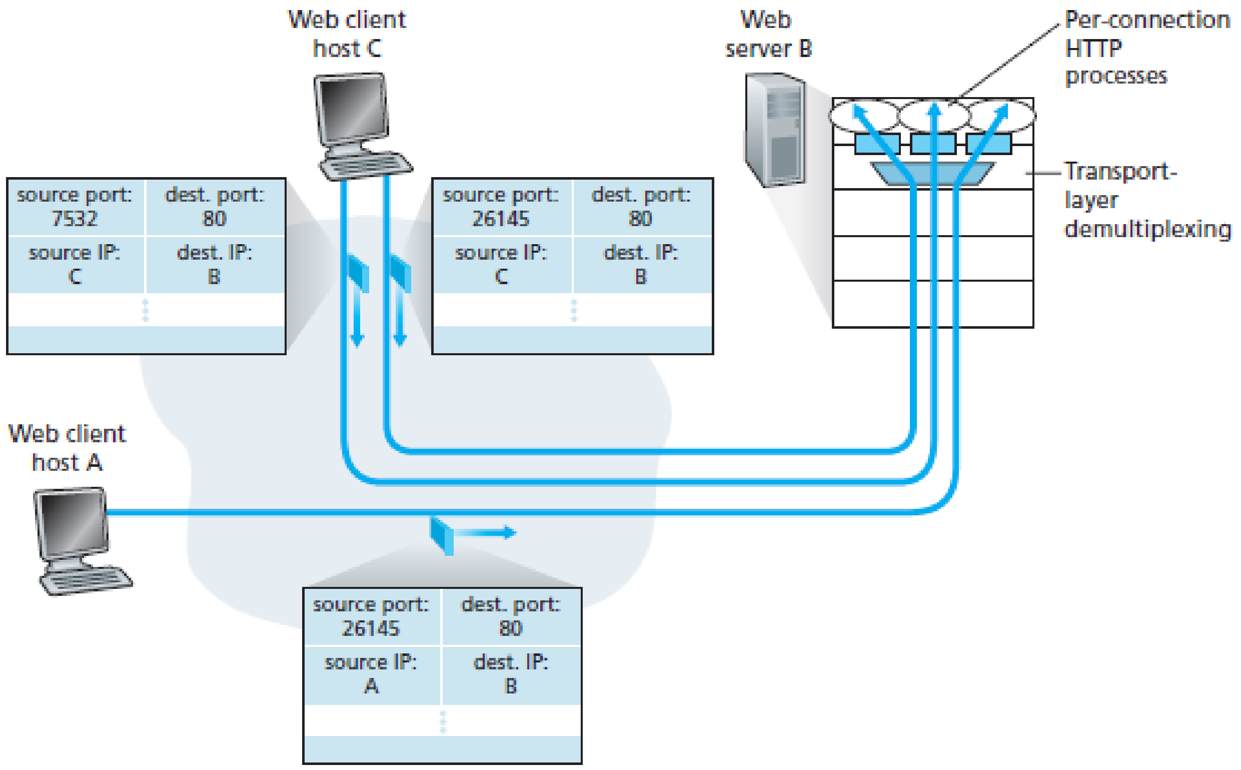
- We can say that a TCP socket can be identified **by four elements**:
  - Source address (IP).
  - Source port.
  - Destination address (IP).
  - Destination port.

- Because of the initial handshake the TCP connection **entangles the processes** of both sides of the socket.

- Only one source and one destination will use this socket, if the client or the server sides of the communications are interrupted, the socket is closed on both sides (which does not happen in UDP).

Transport Layer
Multiplexing/Demultiplexing HTTP example

- We can then have multiple processes from one machine communicating with multiple processes (or with the same process) on another.
- In this example **a host C initiates two HTTP sessions to server B**, and **host A initiates one HTTP session to B**. Hosts A and C and server B each have their own unique IP address (A, C, and B). Host C assigns two different source port numbers (26145 and 7532) to its two HTTP connections.
- Host **A may assign a source port of 26145** to its HTTP connection (not aware of C).
- This is not a problem, since **the two connections have different source IP** addresses.
- In this example, the server opens a new process (or a new thread) for each incoming connection by assigning to it the specific socket (new_socket).
- This is quite typical but opening too many processes (or threads) may impair the performance of the server.



Transport Layer
UDP

- As already explained, **UDP is useful for simple and rapid connections**. It mainly performs 2 tasks:
  - Process-to-process delivery (ports, multiplexing and demultiplexing).
  - Error Checking.

- UDP is connectionless: **there is no handshaking** between sending and receiving transport-layer entities before sending a segment.

- UDP has **no guarantee about delivery of messages** (no reliable data transfer implemented) and has **no congestion/flow control**.

- Roughly speaking, UDP is a **minimalistic protocol** that just adds “the essential” with respect to the lower-level IP protocol.

Transport Layer
UDP

- Why to use UDP instead of TCP?
  - **Application-level control**: UDP is more direct, so the application has more control on the transmission.
    - This can be useful for instance in real-time applications, where sending rate or delays are crucial while data loss can be tolerated, so less checking is preferred.
  - **Fast connection establishment**: there is no handshake, so less delay to establish a connection.
    - This is good for instance in DNS to not provide additional delays.
  - **No connection state**: there are no congestion-control parameters, no sequence nor acknowledgment numbers.
    - A server can typically support many more active clients when the application runs over UDP rather than TCP.
  - **Minimal packet overhead**: The TCP segment adds 20 bytes of header in every segment, whereas UDP only adds 8 bytes.

Transport Layer

UDP: DNS Example

- An example of application using UDP connection is DNS. The DNS client works as follows:
  - In **application layer**: when the host wants to make a query to a DNS server, it constructs a DNS query message and passes the message to UDP.
  - In **transport layer**: without performing any handshaking with the DNS server, UDP adds header fields to the message and passes the resulting segment to the network layer.
  - In **network layer**: the UDP segment is encapsulated into an IP datagram and sent to the DNS server (through the access layer).
  - The client then waits for a reply to its query. If **reply is not received** (possibly because the underlying network lost the query or the reply), it might try different approaches:
    - Resending the query.
    - Sending the query to another name server.
    - Informing the invoking application that a reply is not received.

Transport Layer

UDP: Usage

- Applications needing **reliable data transfer** such as, e-mail, remote terminal access, the Web, run over TCP.
- In this cases **we can't afford packet loss**, for instance, mails must be fully delivered we can't have missing elements.
- **SNMP** uses UDP to carry out device management. Network management applications **must often run when the network is in a stressed state** (precisely when reliable data transfer is difficult).
- **Multimedia applications** such as Internet phone, video conferencing, streaming, often use both UDP and TCP, because small amount of packet loss can be tolerated.
- In general, real-time applications react very poorly to TCP congestion control.

Service	**Application Protocol**	**Transport Protocol**
E-mails	SMTP	TCP
Remote terminal	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Name translation	DNS	UDP
Network device management	SNMP	UDP
Multimedia streaming	Proprietary	UDP or TCP
Internet telephony	Proprietary	UDP or TCP

Transport Layer

UDP: Usage

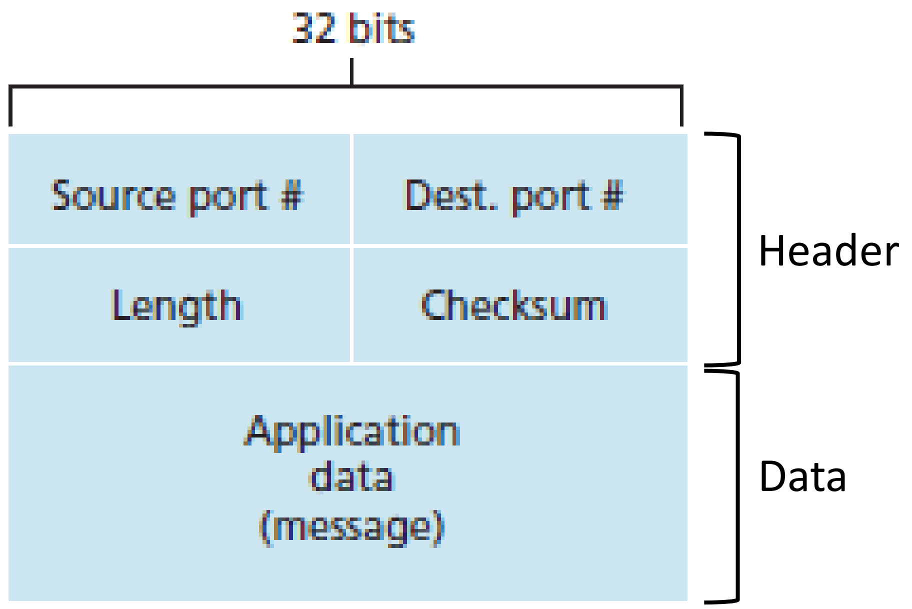
- On the other hand, running **multimedia applications over UDP is controversial** because no congestion control is performed.
- If everyone were to (selfishly) start streaming high-bit-rate video without using any congestion control, **network devices would overflow** causing more UDP packets to be lost.
- High loss rates induced by the uncontrolled UDP senders would also cause the **TCP senders to dramatically decrease their rates**.

Service	Application Protocol	Transport Protocol
E-mails	SMTP	TCP
Remote terminal	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Name translation	DNS	UDP
Network device management	SNMP	UDP
Multimedia streaming	Proprietary	UDP or TCP
Internet telephony	Proprietary	UDP or TCP

Transport Layer

UDP: Segment Format

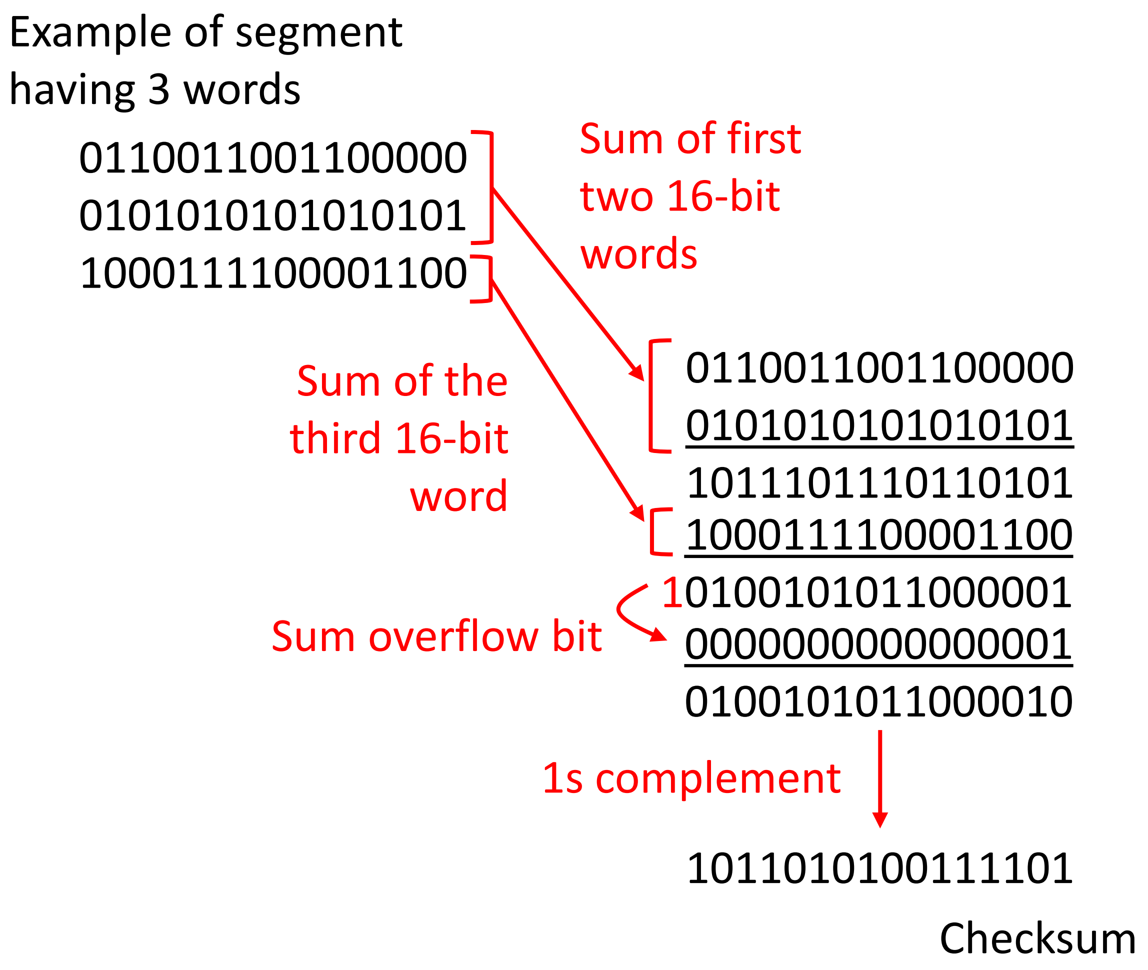
- The **UDP segment** (or datagram to be more precise) is composed by 5 elements:
  - **UDP header** (64 bits) which includes information related to the UDP protocol and is composed by **4 elements of 16 bits** each:
    - **Source port**: port number of the sending process.
    - **Destination port**: port number of the receiving process.
    - **Length**: length of the whole datagram (header+data, i.e., N + 64 bits).
    - **Checksum**: used by the receiver to check if the message is intact.
  - **Data** (N bits) which is the actual message from the application layer.



Transport Layer

UDP: Checksum

- Integrity checking is performed by using the **checksum**. The UDP checksum is a 16 bits filed used for error detection as follows:
  - UDP at the sender side performs the **1s complement of the sum of all the 16-bit words** in the datagram, with **any overflow bit being summed** itself.
  - This **result is put in the checksum field** of the UDP datagram.
  - When **the receiver sums all bits inside the message (checksum included)** the sum must be 1111111111111111 (16 ones).
  - If one bit is 0, then we know that **errors have been introduced** into the packet.


Example of segment having 3 words
Sum of first two 16-bit words
Sum of the third 16-bit word
Sum overflow bit
1s complement
Checksum

```
0110011001100000
0101010101010101
1000111100001100

 0110011001100000
 0101010101010101
 1011101110110101
 1000111100001100
10100101011000001
 0000000000000001
 0100101011000010

 1011010100111101
```