Introduction

We can see the Internet as a network of networks. The **Internet** is an infrastructure that provides services to applications and also a programming interface to distributed applications. Whenever we want to connect to a device, we send a request and then we get a response according to some protocols. A **protocol** establishes the packet format, the order of requests and how the content within a message is organized. If a packet does not arrive the protocol tells if and after how much time we should make another request.

The Internet structure is divided in network edges, so hosts and servers, access networks (wired or wireless), and network core (interconnected routers and network of networks).

When we have a cable that brings different types of data (TV and Internet or phone and internet) we can use two different types of transmission: either we assign to each type a frequency, or we give some timing to them.

Network

The **host** has a sending function: it takes application message, breaks into smaller chunks, called **packets**, of length L bits and it transmits them into access network, with transmission rate R. The packet transmission delay (time needed to transmit an L-bit packet into link) is:

$$\frac{L}{R}\frac{(bits)}{(bits/sec)}$$

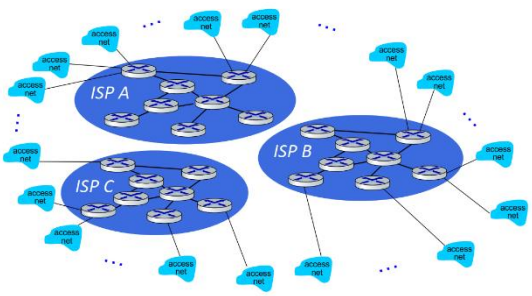
So, if we change the size of the packets, the delay will change too. If we want to transmit these information, we could use a guided or unguided media. Sometimes longer bands lengths have a greater range, but may bring less information.

A **network core** is a mesh of interconnected routers, that forwards packets to the next router. The **router** has two functionalities: **routing** (an algorithm, usually Dijkstra, that, based on the topology, decides which is the path to follow) and **forwarding** (choosing the right router). There's also a forwarding table in which are encoded all the addresses in binary code; in this way we can retrieve data using fewer bits (for example, if we want to know the continent, we may only need two bits instead of all of them).

In the **packet switching**, every time that we forward, we have to wait for the entire packet to arrive (store and forward). If a router receivers too many packets ($T_{arrival} > Trans_{rate}$) then we have the **queueing**: the router has a buffer inside that can store the packets. If the velocity is too high and the router is filled up, packets can be dropped, so we should put a stop to the origin. Packet switching is ideal whenever we have data that aren't sent all the time.

An alternative to packet switching is **circuit switching**, where we have a dedicated circuit for two specific ends. If the two devices are not sharing, then the circuit segment is left idle. To share the segment, we can give to each user a narrow frequency band (Frequency Division Multiplexing, **FDM**) and it can transmit at max rate, or we can divide in time slots (Time Division Multiplexing, **TDM**), and it can transmit at maximum rate of a wider band only in that time.

The hosts are connected to Internet via access **Internet Service Providers** (ISPs). The ISPs must be interconnected in turn, so that any two hosts can send packet to each other. If we want to connect all of them directly, we'll have $O(N^2)$ connections, so we use global and regional ISPs. There are also **Internet exchange points** (IXP) to connect the global ISPs and other types of networks like the content providers ones (Google, Microsoft, etc.).



Performance

There are many types of delay in the packet:

$$d_{nodal} = d_{proc} + d_{queue} + d_{trans} + d_{prop}$$

Name	Description	Formula	Variables
d_{trans}	Transmission delay: time needed to send a packet through the link.	$\frac{L}{R}$	L: packet length (bits) R: link transmission rate (bps)
d_{prop}	Propagation delay: time needed for the packet to travel in the link.	$\frac{d}{s}$	d: length of physical link s: propagation speed $\left(2 \cdot 10^8 \frac{m}{s}\right)$
d_{queue}	Queue delay: time for waiting in queue.		
d_{proc}	Processing delay: router time to read and analyse the packet and choose the right path.		

If we want to find the traffic intensity we can do:

$$\frac{L \cdot a}{R} \frac{arrival\ rate\ of\ bits}{service\ rate\ of\ bits}$$

Where a is the average packet arrival rate, L the packet length and R the link bandwidth (bit transmission rate). In real Internet delay we have a traceroute program, so we send three packets to a router and, when sent back, we're able to measure the delay between two ends.

In some cases, we can have a specific route for packets, so that they don't have to "stop" through all the routers, but you can reach directly the destination. This way avoids the overflow of packets arriving in a router.

We define with **throughput**, the rate at which bits arrive at the receiving endpoint. It can be instantaneous if it's at a given point in time, or average, if it's over a longer period of time. If we have an end-end connection with a link inside, there's often a bottleneck: the two links indeed, may have a different throughputs.

Network security

Some types of cyber-attacks are:

- **Packet sniffing**: a network interface reads and records all the packets passing in the link.
- **IP spoofing**: we are injecting a packet with a false source address.
- **Denial of Service** (DoS): we take control of many hosts and start overwhelming a resource with bogus traffic.

In order to avoid that we have many measures: **authentication** (providing you are who you say you are), **confidentiality** (via encryption), **integrity checks** (digital signatures prevent/detect tampering), **access restrictions** (password-protected VPNs), **firewalls** ("middleboxes" in access and core networks to restrict packet arrivals and DoS attacks).

The network is divided in to layers. Some structures are repeated among layers (modularization) so that, whenever there is an issue, instead of changing the entire network, we just change a specific protocol. In the layered internet protocol stack the layers are:

- **Application**: the message exchange needed to realize the application, so http. http protocol provides an interaction between client and server; DNS establishes the translation between name and IP address.

The seven layer OSI/ISO reference model

- **Transport**: provides the capability of establish a connection between host and server, done in a way that is generic, so not for a specific server; communication between processes.
- **Network**: protocol layer that contains all the functionalities that let your data to travel efficiently along the destination.
- **Link**: regulates the communication over a link. So, it's not an end-to-end, but only of one point to one point. Here we take care of the specific used technology.
- **Physical**: how to transmit bits on a wire or on a wireless network.

Other two layers missing and implemented if needed in the application level, are **presentation**, that allows application to interpret meaning of data (encryption or compression) and **session**, the synchronization, checkpointing or recovery of data exchange.

Application layer

A network app must run on different end system and communicate over network. In a network-core device we don't have to write software.

We have two types of networks: in the **client-server paradigm**, there are always-on **servers** with a permanent IP address (often in data centers) and a **client**, that has to pass through a server in order to communicate with another client; it has a variable IP and may be intermittently connected. In the **peer-peer architecture** there are no always-on servers, but some arbitrary end systems can directly communicate. **Peers** request service from other peers, provide service in return and new peers bring new service capacity and demands. Peers are intermittently connected and change IP addresses.

A **process** is a program running within a host; we can divide between a client process (that initiate communication) and a server process (that waits to be contacted). Within the same host two processes communicate using **inter-process communication** (defined by the OS), while processes in different hosts communicate by exchanging **messages**. A process can send/receive message to/from its socket. The **socket** is like a door between the application and the transport layer. It allows to prepare a message to be sent and it receives the messages.

Every host has a unique 32-bit IP (Internet Protocol) address and every process in it has the same IP. This is why, if we have to communicate a message between different processes we need the port number too, that, together with the IP address makes the **identifier**. The port number 80 notifies that the message is coming from a browser (so all browsers are in port 80).

An application-layer protocol defines:

- **Type of messages exchanged**: request, response
- **Message syntax**: fields in messages
- **Message semantics**: meaning of information in fields
- **Rules for when and how processes send/respond to messages**
- **Open protocols**: defined in RFCs, everyone has access to protocol definition; so, we have interoperability
- **Proprietary protocols**: Skype, Zoom

Transport layer requirements for the Application layer

In order to have a network app we need 4 transport service requirements:

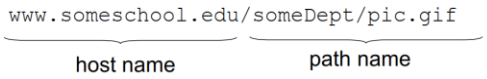
- **Data integrity**: some apps, like file transfer or web transactions, require 100% reliable data transfer, others, like audio or video calls, can tolerate some loss.
- **Timing**: some apps require low delay to be effective (Internet telephony, interactive games).

- **Throughput**: some apps require minimum amount of throughput to be effective (multimedia), others are "elastic apps" and accept every throughput they get.
- **Security**: encryption, data integrity, etc.

We have two types of transport protocol services. One is the TCP, with reliable transport, flow and congestion control, but it does not provide timing, minimum throughput and security. The other type is the UDP, that is an unreliable data transfer and does not provide reliability, flow and congestion control, timing, throughput guarantee, security and connection setup.

HTTP

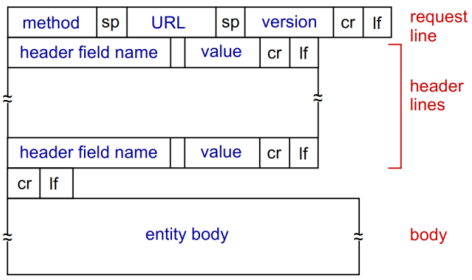
A web page is made of objects, each of which can be stored on different Web servers. Every object can be an HTML file, or an image, audio, etc. A web page is made of a base **HTML**-file and several referenced objects, addressable by a **URL**, like:



HTTP uses TCP, so asks to a server through port 80, to load the objects of a web page (usually we ask for an index and the request is automatically done by the browser). HTTP is **stateless**, so we have no connections with past states (we can open two specific pages of a web site, without having problems); this easier to manage, indeed we don't have to save the state and if a server/client crashes we don't have problems to reconcile states. In general, the connection is divided in three stages: TCP connection is open, then we send one object if we are in the non-persistent HTTP (downloading multiple objects requires multiple connections) or multiple if we are in the persistent HTTP and then we close the TCP connection. The information is stored in three different places: client, server and in the path in between them.

We define with Round Trip Time (**RTT**) the time spent to send a packet from host to destination and from destination to host. In the **non-persistent HTTP**, you first send a packet to initialize the connection, and then you start transmitting a file; the time is $2RTT + T_{transmission\ file}$. This is repeated for every object. Here, the browser often open multiple parallel TCP connections to fetch referenced objects in parallel. In the **persistent HTTP** (HTTP1.1) we leave the connection open so that the client sends requests as soon as it encounters a referenced object.

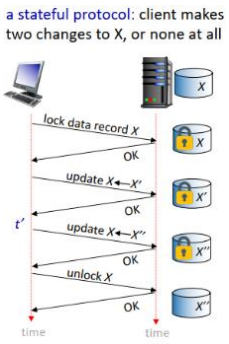
We have two types of HTTP messages: request and response. The format of **request** message is:



We have 4 different methods for requests:

- `POST`: usually in case of form inputs in web pages. The user input is sent from client to server in entity body of HTTP POST request message. It's not visible, has no length limits and it can't be bookmarked or reloaded. It may modify the server's state.
- `GET`: this method is used to retrieve data from a server. In the request message we add a "?" (like "`www.somesite.com/animalsearch?monkeys&banana`"). The request has a length limit (usually up to 240 characters), it's visible and can be bookmarked and reloaded.

- HEAD: requests headers (only) that would be returned if specified URL were requested with an HTTP GET method. Used for example if we want to check the status of a resource without downloading the resource itself.
- PUT: uploads new file (object) to server. Completely replaces file that exists at specified URL with content in entity body of the request message.



The HTTP response contains a code that points at the status: 200 is ok, 301 means that the requested object has been moved in a new location specified later, 400 bad request, 404 not found, 505 HTTP version not supported.

HTTP GET/response interaction is stateless, so we don't have multi step-exchanges. In this way we don't need to save the state of a request and we don't have to recover one in case of a partially completed transaction.

Cookies

In order to maintain some state between transactions, web sites and client browser use cookies.

There are three ways to deal with cookies in a session management. We could ask every time for a specific object, for example:

```
GET /miaservlet?Nome=Giovanni&Cognome=Severi
```

Will give back the cookie:

```
cookie("Giovanni","Severi") <html>… …</html>
```

If we have persistent objects located on the server side, we'll get for example the cookie:

```
cookie(SessionId=51) <html>… …</html>
```

That this time is referring to a table with key and value. The key is 51 and the value is, in this case "Giovanni Severi".

In this case we could directly refer to the key, so:

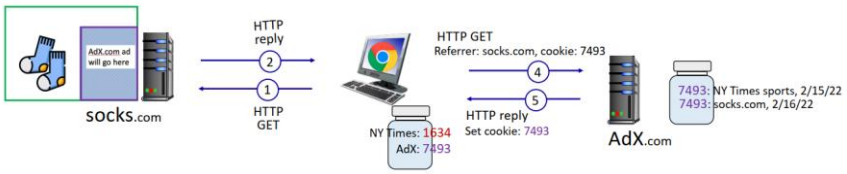
```
GET /miaservlet cookie(SessionId=51)
```

We have four components:

1. A header line of HTTP response message
2. A header line in next HTTP request message
3. The cookie file is kept on user's host and it's managed by user's browser.
4. The back-end database in in the Web site.

When we access a new site, we set a new cookie, with an ID. Every time that we want to do a cookie-specific action, we send a request message to the server, that will send it into the backend database. The database will answer to the server, that will in turn send a cookie with the answer.

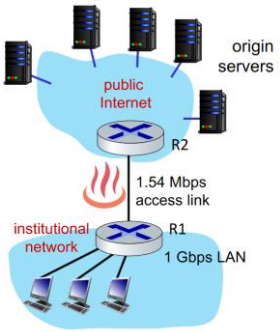
We can use cookies for authorizations, recommendations, shopping carts or user session state (in Web e-mail). In general, cookies are used from websites to learn more about the user. The third-party persistent cookies are tracking cookies that allow common identity (cookie value) to be tracked across multiple websites. The GDPR regulates all the cookies that may identify an individual, so when they may contain personal data.



For example, here when we go to the website socks.com, the ad from AdX.com catches the cookie from the user 7493 and send it into the database of AdX.com. Now, whenever I'll enter into another

website, I may receive an ad from AdX.com that is suggesting me something from socks.com, even if I have never visited AdX.com.

Web cache



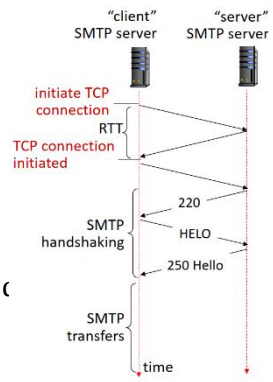
To improve efficiency in response time to client request and reduce traffic on an institution's access link, we use a web cache. The browser sends all HTTP requests to the cache: if the object is in the cache, it returns it to the client; else the cache will request it to the server. So, web cache acts both as a client (for the original server) and a server (for the client). The server tells the constraints of the data to the cache (like if it's allowable or it'll be cached only for a while). For example, in the conditional GET, we must specify a date for the cached object; if the object is not/hasn't been modified before date, then we'll get as response 304 Not Modified, else, the object.

A practical example could be the one in figure: We make requests from the institutional network to the server, but, with a high utilization, we'll have large queueing delays (1Gbps>>1,54Mbps). A solution could be simple buying a faster access link, but it's expensive. Another one is to use a web cache (cheap); at the beginning we'll have some delay for the misses.

HTTP versions

	HTTP1.1	HTTP/2	HTTP/3
Introduced:	Multiple pipelined GET over single TCP connection	More flexibility at server in sending object to client	
Server responds:	FCFS (first-come-first-served scheduling)	We transmit requested object according to priority	
Head-of line (HOL) blocking	With FCFS smaller objects may have to wait for transmission behind large objects.	We divide objects into smaller frames.	
Loss recovery (retransmitting lost TCP segments)	It stalls object transmission.	It stalls object transmission.	
Security	No security	No security	Security
Other		Same methods of HTTP1.1. We push unrequested (embedded) objects to client avoiding the delay for extra requests.	Per object error- and congestion-control (more pipelining) over UDP

Email



We have three major components: user agents (the one that compose, edits and reads mail messages), mail servers and simple mail transfer protocol (SMTP), the protocol between client and server to exchange e-mail messages, defined in RFC 5321 (like RFC 7231 defines HTTP). RFC 2822 defines syntax for e-mail message itself (like HTML defines syntax for web documents), like, header, "To:", "From:", "Subject:", etc. All the

sent/received messages are stored in the server's mailbox. The mail server has also the message queue of outgoing (to be sent) messages.

SMTP RFC uses TCP to reliably transfer email message from client to server and it uses port 25. We have a direct connection between servers. There are three phases of transfer: SMTP handshaking (greeting), SMTP transfer of messages and SMTP closure. Usually, when sending a mail, we have these phases:

- We use the user agent (UA) to compose the email message.
- The UA sends the message to the address using SMTP, so it's placed in message queue.
- Client side of SMTP at mail server opens TCP connections with the receiver's server.
- SMTP client sends the message over the TCP connection.
- The receiver's mail server places the message in its mailbox.
- The receiver invokes its UA to read the message.

Comparison with HTTP:

HTTP	SMTP
Client pulls	Client pushes
ASCII command/response and status codes	ASCII command/response and status codes
Each object is encapsulated in its own response message	Multiple objects sent in multipart message
	Persistent connections
	The header and the body must be in 7-bit ASCII
	SMTP server uses CRLF.CRLF to determine end of message

In retrieving emails, we have three protocols:

- **SMTP**: delivery/storage of e-mail messages to receiver's sender.
- **IMAP** (Internet Mail Access Protocol, RFC 3501): provides retrieval deletion and folders of stored messages on server.
- **HTTP**: (Gmail, Hotmail or Yahoo!) provides web-based interface on top of STMP (to send), IMAP (or POP) to retrieve e-mail messages.

Domain Name System

The IP address is composed of 4 bytes; it corresponds to a string made up of 4 integers between 0 and 255 separated by a dot. It's structured in a hierarchical way, so, by reading it from right to left we obtain more information on the host location. The Domain Name System (DNS) is a distributed database that has three main functions:

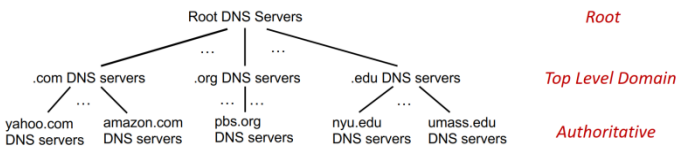
- **Translates the IP with the host name** (the "usual name" is the human one, while the IP is used in the network): it's the DNS resolution.
- **Aliasing**: it obtains simpler names from complex ones. For example, `relay1.west-coast.enterprise.com` (canonical name) may have two nicknames (aliases) `enterprise.com` and `www.enterprise.com`. This is useful when we want to distinguish mail and web server of a company that same names, but different canonical ones.
- **Load distribution**: When a site has a lot of requests, it may happen that it's hosted in replicated servers, each one with a different IP address but same canonical name. When

requesting to the DNS we'll get a list, whose order is varied at each new client request, allowing rotation between server and distributing the traffic. There's a main problem: the address caching. With caching the name resolution is much faster, but not fully controllable, indeed only the ≈ 5% of requests reaches the authoritative server. We have to add a TTL field that, once elapsed, should generate a cache miss in the next request. This is not always honored by local DNS servers and may be a problem for Content Delivery Networks (CDNs), which is why they resort to alternative machanisms.

The DNS is implemented as application-layer protocol and it's used by other application layer protocols (HTTP, SMTP, FTP) by requesting through port 53. Usually, when we are going to do a request to a website, the browser extract the name, the DNS client sends a query to the DNS server to translate the host name into the IP and, when we have the IP, the browser can start the connection with that host. The DNS is executed by end system using a client-server paradigm; it utilizes the UDP to transfer messages and it pushes the complexity towards the edge of the network. It's not a proper application because the client cannot interact with the DNS servers directly. Every host is configured with at least one DNS address.

DNS servers don't have the whole Internet mapping. They're organized in a three-level hierarchy: **root**, **top-level domain** (TLD) and **authoritative**. When we want to access an IP address for a website like www.test.com, the client queries root server to find .com DNS server, from that to get the test.com DNS server and then to the www.test.com server.

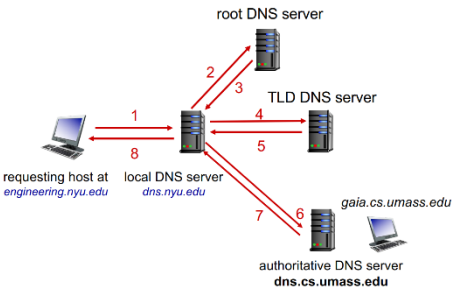
The reasons why we don't have a centralized DNS is because of the huge number of requests that a DNS receives and because of the reliability and security.



The Root DNS servers are official, contact-of-last-resort by name servers that cannot resolve name. There are 13 root servers, replicated for sake of security in 247 root servers. The root server queries a DNS TLD if it doesn't know how to map an address, obtains the required mapping and send it to the local DNS. Here the ICANN (Internet Corporation for Assigned Names and Numbers) manages root DNS domain, while the DNSSEC provides security (authentication, message integrity).

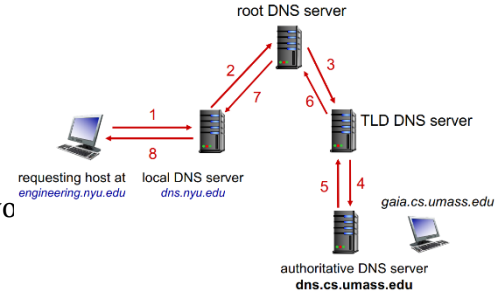
The TLD servers are responsible for all the top-level country domain (`.it`, `.uk`, etc.) and for all the `.com`, `.org`, `.net`, etc. The authoritative DNS servers are the organization's own DNS servers providing authoritative hostname to IP mappings for organization's named hosts. This can be maintained by organization or service provider.

Local DNS name servers: when host makes DNS query, it's sent to its local DNS server, that searches it into his cache of recent name-to-address translation pairs (possibly out of date). The local DNS doesn't strictly belong to hierarchy, but it may forward the request into DNS hierarchy for resolution/translation. Each ISP has a local DNS name server (ex: MacOS: `% scutil -dns`; Windows: `>ipconfig /all`).



When asking for a name into the Local DNS server, there are two types of query, the iterated one and the recursive one. In the iterated query the local DNS server contacts a single server that may respond with the answer or with the name of another server to ask. In the recursive query we are using the entire hierarchy from the root to the authoritative DNS server.

Caching DNS information: once any server learns mapping, it caches mapping, so that every time that someone will ask for the same information, the answer will arrive faster. TLD server are typically cached in local name servers. After some



time, the cache entries disappear by timeout (TTL; time to live); also, they could become out-of-date, for example if a named host has changed its IP and all the TTLs are not all expired. In this case the cache server flags its response as non-authoritative.

Inside a DNS record we have a distributed database storing resource records (RR). The RR format is (name, value, type, ttl). Depending on the type we can have:

- Type **A**: request for address, so name is the host name and value the IP address
- Type **CNAME**: request for alias, so name is the alias and value the canonical name.
- Type **NS** (Name Server): used when routing along a query chain, name is the domain name (ex. foo.com) and value the host name of the authoritative DNS server which manages the domain (dns.foo.com).
- Type **MX**: request for mail server, so name is the name and value the alias.

identification	flags
# questions	# answer RRs
# authority RRs	# additional RRs
questions (variable # of questions)	
answers (variable # of RRs)	
authority (variable # of RRs)	
additional info (variable # of RRs)	

Protocol messages: DNS query and reply messages have both the same format with a message header of 12 bytes. The identification is a 16-bit # for query (the reply uses the same #); the flags specify some details about the request (query/reply, recursion desired, recursion available, reply is authoritative); in questions we'll put name, type fields for a query.

If you want to put your info into the DNS you have to register the name at a **DNS registrar** and create authoritative servers locally.

DNS security: in the DDoS attacks there could be a bombarding of requests to the root servers (not successful also for the traffic filtering) or to the TLD servers (potentially more dangerous). In the spoofing attacks, there is an intercept of DNS queries, returnin bogus replies.

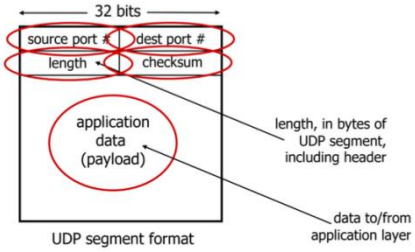
Content distribution networks (CDN)

If we want to stream content to a very large number of simultaneous users, we have two options: either we store everything on a single large server (but the solution doesn't scale), or we store multiple copies of videos at multiple geographically distributed sites. In this second option we could "enter deep", in which we push CDN servers deep into many access networks, or "bring home", in which we have a smaller number of larger clusters in PoPs near access nets. When doing a request, we contact a dispatch server that will answer us with the right server to contact, based on the distance, traffic, etc.

Demultiplexing

In multiplexing we handle multiple data from application layer trough sockets. In demultiplexing the host receives a datagram with source/destination IP and source/destination port number and thanks to this, we can direct the segment to the appropriate socket. IP/UDP datagrams with same destination port #, but different source IP addresses and/or source port numbers will be directed to same socket at receiving host. In the connection-oriented demultiplexing, TCP socket is identified by 4-tuple (source/destination IP address and source/destination port number). The server may support many simultaneous TCP sockets, each associated with a different connecting client. So, in UDP we demultiplex only by using destination port number, while in TCP by using the 4-tuple. Multiplexing/Demultiplexing happen at all layers.

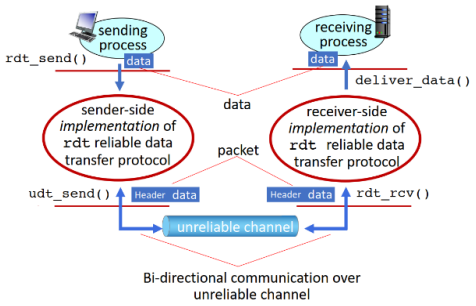
In UDP (User Datagram Protocol) we don't have handshaking between sender and receiver (so it's connectionless). It's simple because it has no connection state, small header size and it has no congestion control (UDP can blast away as fast as desired) but the



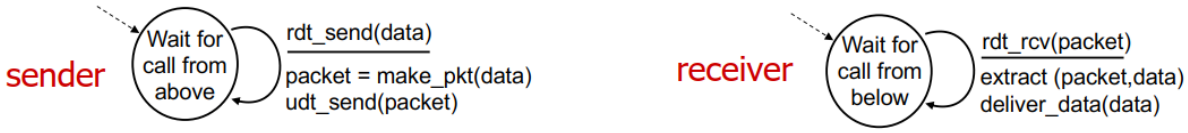
segments may be lost or delivered out-of-order to app. If we need, we have to add reliability and congestion control at application layer. When we receive a message in UDP, we check the checksum header value, extract the application layer message and demultiplex it up to application via socket.

The UDP **checksum** goal is to detect error in transmitted segment; the sender takes the content of the UDP segment, treating them as a sequence of 16-bit integers, sum them and then stores the sum into the UDP checksum field. The receiver computes again the sum and checks if it's equal to the checksum; if it's different, then we have an error, if it's equal probably not, because it still can happen that the numbers change in such a way that the sum doesn't change.

Principles of reliable data transfer

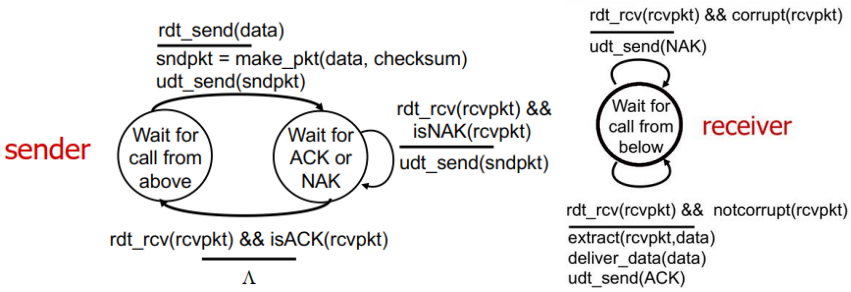


Let's assume that we are going to send some data to a receiver. Both the sender and the receiver side of the transport protocol are reliable, but the channel is not. So, the sender and the receiver don't know the state of each other when the message has been delivered. This is why we have the reliable data transfer protocol (rdt). When the application wants to deliver to receiver's upper layer some data, it sends the message `rdt_send()`; when the packet is about to travel along the unreliable channel the rdt calls `udt_send()`; `rdt_rcv()` is called instead when the packet arrives on receiver side of channel and `deliver_data()` when the data is about to be delivered to the upper layer. We can imagine the sender and receiver as FSMs. If the channel is reliable (rtd1.0), we simply have:



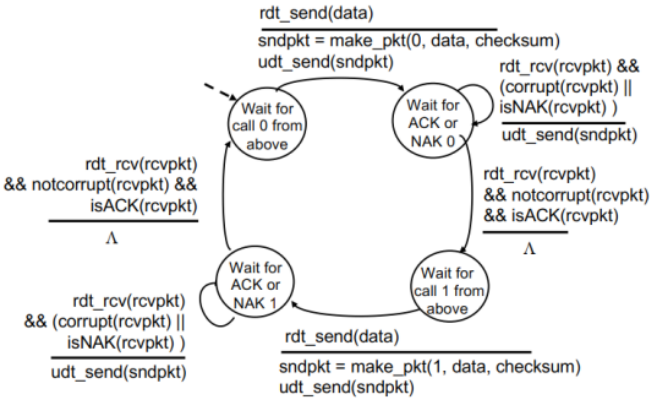
So, for example, the sender sends a packet and then waits for the next packet to arrive.

But usually, the channel is not always reliable, so we have to take a countermeasure to recover these errors (rtd2.0). In rtd.2.0 we are adding the error detection, receiver's feedback and retransmission. We have so the **ARQ protocols** (Automatic Repeat reQuest). When the receiver receives a packet, it can tell to the sender that the packet is OK (acknowledgment, ACK) or not OK (negative acknowledgment NAK); in this last case, the sender retransmits the packet. Whenever a sender sends one packet, it waits for the receiver response.

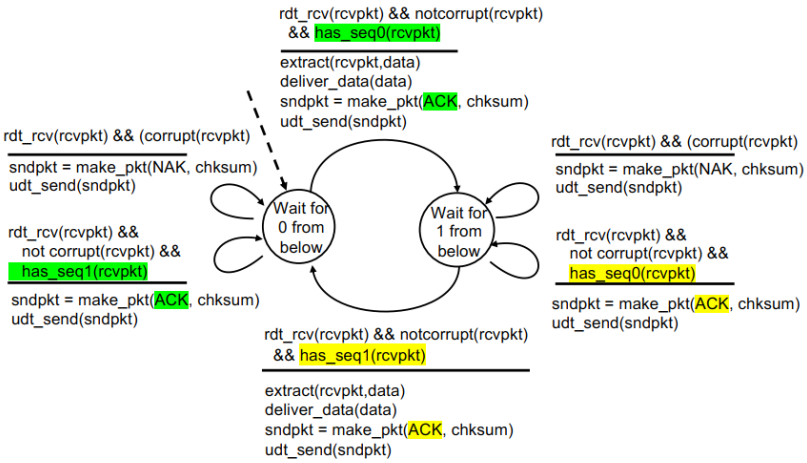


In this new FSM the sender is waiting for the receiver to send an ACK/NAK.

But what happens if even ACK/NAK are corrupted? The sender doesn't know what happened at receiver, so it'll retransmit the current packet adding a sequence number. In this case the FSM is doubled, because we have to consider both the first and the second packet.

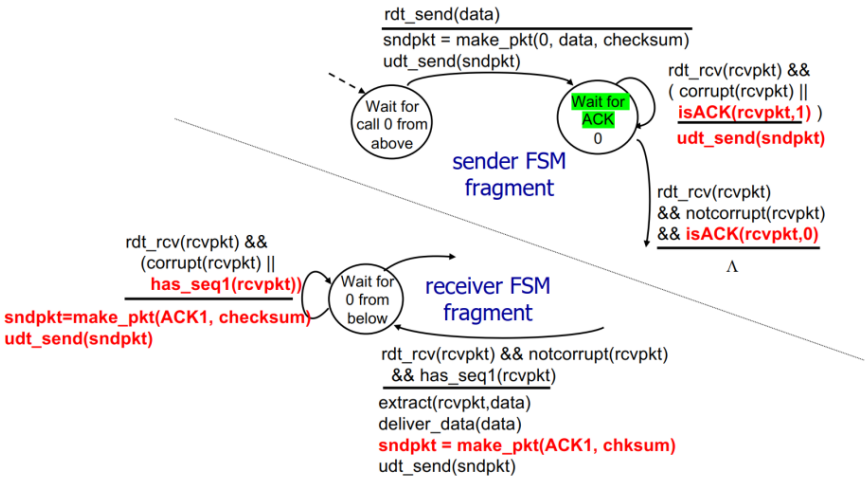


And for the receiver:

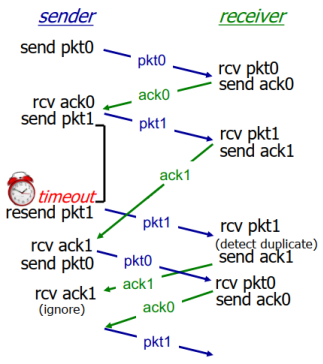


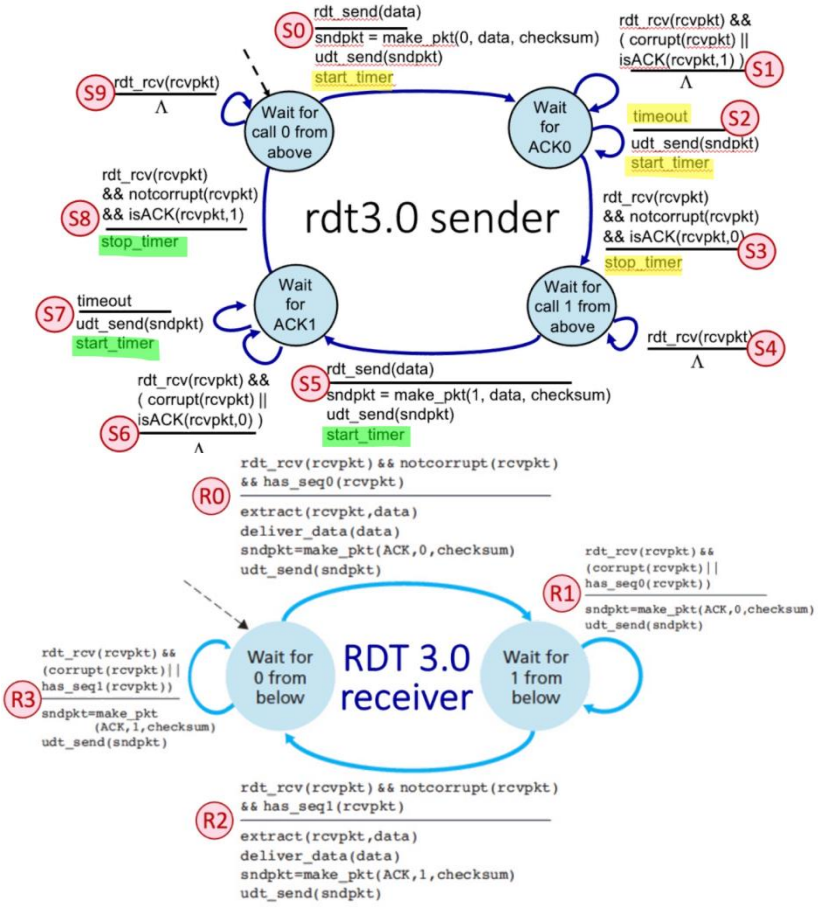
So, if I'm waiting for the packet number 0 and I receive a packet number 1 I discard it, because it's a duplicate and it's the same for the opposite.

In the rdt2.2 we only use ACKs. In this case if the duplicate packet is sent with an ACK, we are giving the same information as a NAK, so retransmit current packet.



In the rdt3.0 we make a new assumption: the underlying channel can also lose packets, both for data and for ACKs. In order to solve this, we can put a timer and after that the sender will retransmit if no ACK has been received. In this case we can handle the lost packet, but also the delayed one, indeed, in this last case, the receiver can discard the duplicate thanks to the sequential number.





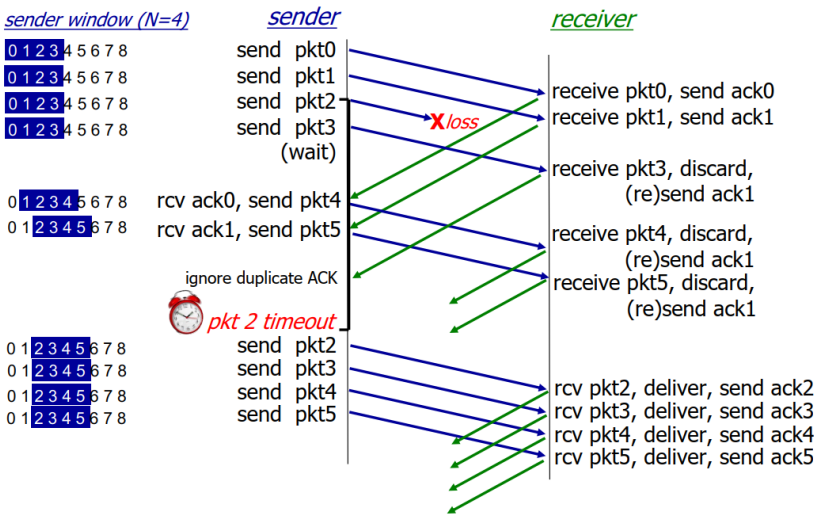
Performance of rdt3.0 and Parallelism

If the sender has to wait for the response, then it'll be idle most of the time; the effective throughput will be so much lower respect to the maximum possible. If we want to compute the utilization time of the sender, then we have to divide the transmission delay $\frac{L}{R}$ with the RTT plus the transmission delay:

$$U_{sender} = \frac{\frac{L}{R}}{RTT + \frac{L}{R}}$$

For the throughput (the actual data rate, that is not the same as the actual data bandwidth) we have the actual data that we are going to transfer divided by the total time we are using to transmit it. If we use a better link, we'll increase the R but still we have the RTT, so the limit would still be $1/RTT$.

In order to solve this problem, we can add a pipeline, where we'll send k packets yet-to-be-acknowledged. The utilization will be increased k times (but k is limited because we have also to catch the response and we can't overlap them). The sender keeps track of what has been acknowledged so far and after that it has a window of N packets that can send at the same time. The sender has a cumulative ACK, each one with a sequence number. When a packet is correctly sent, then we can move the window of one packet. The receiver will always send ACK for correctly-received packets with highest in-order sequence number. It may generate duplicate ACKs and need only to remember `rcv_base`. Whenever the receiver receives some packet not in order (maybe one has been delayed) it can buffer all the others and wait for the one that is missing. If it goes in timeout, then it has to resend the window.



So, the sender sends multiple packets, keeping for each one a timer. The receiver sends individual ACKs for each packed. The sender has also a window of consecutive packets. The window size is the maximum number of packets that we can send.

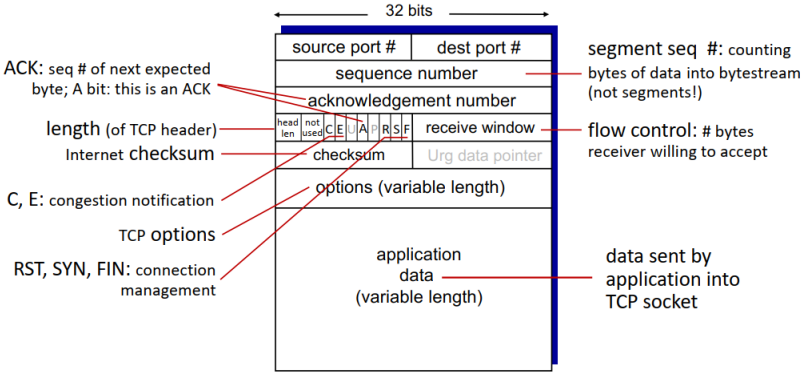
The sender receives some data in the application; if there is some sequence number available (so a packet) it will send the packet. If a specific packet goes in timeout, the sender will resend only that. When it receives an ACK, if it's in the middle it marks the packet as received, if it's at the beginning of the window, it will advance the window base of 1 packet. The receiver, when it receives a packet it sends an ACK(n), if it's out-of-order it buffers that packet. If it's in order (also buffered but in order) then it can deliver.

A major problem is that the receiver cannot see the state of the sender. If for example the sender sends 3 packets correctly and all the ACKs of the receiver are lost, then the receiver has moved its window of all the sequence numbers and, when the sender will send again the first packed because of the timeout, the receiver will just accept it as a new one, not an old one. If we increase the window size of the receiver, we can detect the duplicate packets. Since we want to prevent having packets with the same sequence number that are showing up in any moment, in high-speed networks TCP packets cannot live in the network for more than 3 minutes.

Handshaking

In TCP we have one sender and one receiver with cumulative ACKs and pipelining. In order to initialize the sender, we have to make a **handshaking**. It's also flow controlled, so the sender won't overwhelm receiver.

In encapsulation we start from the application layer and we call the data we are sending "packet", in the transport is "segment" and in the internet one is "datagram". Now we'll analyse the structure of a single TCP packet:



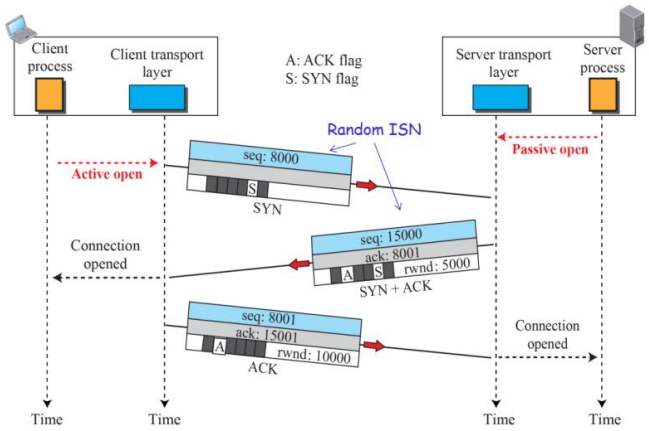
The fields in the segment are those:

- **Source Port Address**: A 16-bit field that holds the port address of the application that is sending the data segment.
- **Destination Port Address**: A 16-bit field that holds the port address of the application in the host that is receiving the data segment.
- **Sequence Number**: A 32-bit field that holds the sequence number, of the segment. It counts the number of bytes that are in the application data field. We start from 1. If the first packet has n bytes of data, the sequence number of the second packet will be $1 + n$. It's used to reassemble the message at the receiving end of the segments that are received out of order. It's different from the packet number, that is instead a sequence of number that specifies the order of the packet inside the divided message (0, 1, 2, ...)
- **Acknowledgement Number**: A 32-bit field that holds the acknowledgement number. It's the sum of all the bytes received (cumulative), so the sequence number we are expecting from the other side. When we receive the first segment with a packet of n bytes, the Acknowledgment number will be n, then $n + m$ and so on. It's an acknowledgement for the previous bytes being received successfully
- **Header Length** (HLEN): This is a 4-bit field that indicates the length of the TCP header by a number of 4-byte words in the header, i.e. if the header is 20 bytes (min length of TCP header), then this field will hold 5 (because 5 x 4 = 20) and the maximum length: 60 bytes, then it'll hold the value 15(because 15 x 4 = 60). Hence, the value of this field is always between 5 and 15.
- **Control flags**: These are 6 1-bit control bits that control connection establishment, connection termination, connection abortion, flow control, mode of transfer etc. Their function is:
    - URG: Urgent pointer is valid
    - ACK: Acknowledgement number is valid (used in case of cumulative acknowledgement)
    - PSH: Request for push
    - RST: Reset the connection
    - SYN: Synchronize sequence numbers
    - FIN: Terminate the connection
- **Congestion notification** (C, E):
- **TCP options**: One of the options is the MSS (Maximum Segment Size). In ethernet, the maximum number of bytes we can send in a single segment is 1500 bytes. Knowing that 40 bytes are only for the header, we are left with 1460 bytes of data to send.
- **Window size**: This field tells the window size of the sending TCP in bytes.
- **Checksum**: This field holds the checksum for error control. It is mandatory in TCP as opposed to UDP.
- **Urgent pointer**: This field (valid only if the URG control flag is set) is used to point to data that is urgently required that needs to reach the receiving process at the earliest. The value of this field is added to the sequence number to get the byte number of the last urgent byte.

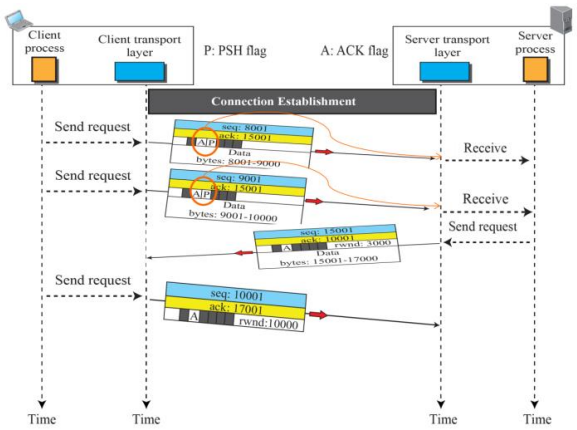
If the receiver receives out-of-order packets, it's up to the implementor to figure out how to order them and not to TCP.

In order to establish a connection with a server, we do the **3-way handshake**: the client choses a random ISN (Initial Sequence Number) x and sends it to the Server with the flag SYN. The server will answer with a message with another random sequence number y, an ack that is equal to $x + 1$ and

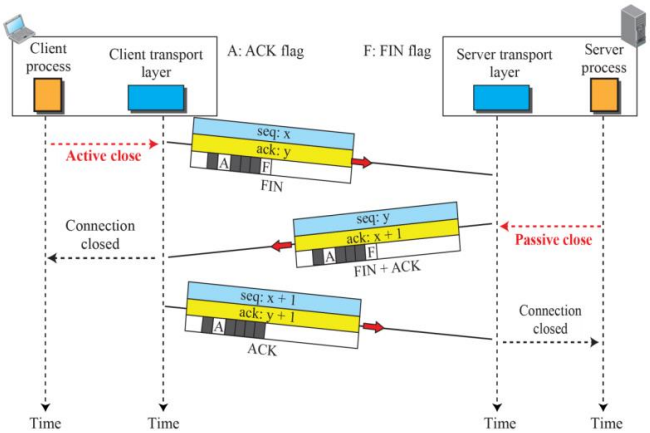
flags of ACK and SYN asserted. In the third step the client answers with a sequence that is $x + 1$, an ack $y + 1$ and only the ACK flag asserted.



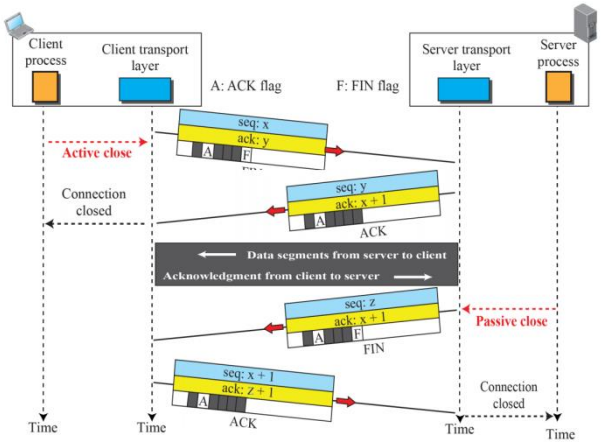
The PSH (push) flag forces the server to send request faster. When we send two or more segments with the PSH flag asserted, the server will merge all the requests and will answer with a single segment:



For connection closing we have a similar handshaking with the `FIN` flag asserted:



If the client doesn't answer (the connection is half closed because the server is still on), the server has to send another request with a different sequence number:



When exchanging segments in TCP, we have to estimate the Round-Trip-Time (RTT), but we have to do a trade-off: if it's too short, we'll go in timeout too soon and send unnecessary retransmission, while if we wait too long the reaction to segment loss is too slow. Also, in order to estimate RTT we have the $SampleRTT$, a measured time from segment transmission until ACK receipt (excluding retransmissions) or, if we want something smoother, the average of several recent measurements.

RTT estimation

(da rivedere tutto)

In order to estimate RTT, we have a formula, the **Jacobson's algorithm**, that evaluates the exponentially weighted moving average (EWMA):

$$RTT_i = \alpha \cdot RTT_{i-1} + (1 - \alpha) \cdot rtt_i$$

Where rtt_i is the time between the transmission of the i^{th} packet until the ACK of that packet, RTT_i is the average round trip time after i^{th} packet and $\alpha = 0{,}875$.

If we have a retransmitted segment, we can't use rtt_i because of the problems with the ACK. With **Karn's algorithm** we ignore rtt_i for any retransmitted segments to update RTT_i. When a segment is sent, a retransmission timer is started. If the segment is ACK'ed before the timer expires, we turn it off, otherwise we retransmit it and we set again the timer. RTO_i is the retransmission timeout interval for the i^{th} packet. In order to determine RTO_i we could use:

$$RTO_i = 2 \cdot RTT_{i-1}$$

But, when RTT_i has a high standard deviation (so the average may vary really fast, maybe one time RTT is high and then is low), this method times out too quickly, so we set:

$$RTO = RTT + a\ MDEV$$

With a that is a constant and $MDEV$ is the mean deviation:

$$RTO_i = RTT_{i-1} + 4 \cdot MDEV_{i-1}$$

We use the mean deviation because the true standard deviation is expensive to compute:

$$STDDEV = \left(\left(rtt_1 - avg(rtt)\right)^2 + \left(rtt_2 - avg(rtt)\right)^2 + \cdots \right)^{\frac{1}{2}}$$

While, for the mean deviation (another $EWMA$):

$$MDEV_1 = (1 - \rho) \cdot MDEV_{i-1} + \rho \cdot |rtt_1 - RTT_{i-1}|$$

With $\rho = 0{,}25$.

When evaluating the timeout interval, we have to use the $EstimatedRTT$ plus a safety margin (that will be higher for $EstimatedRTT$ that has a large variation):

$$TimeoutInterval = EstimatedRTT + 4 \cdot DevRTT$$

$DevRTT$ is an $EWMA$ of $SampleRTT$ deviation from $EstimatedRTT$, so we use the formula of the mean deviation:

$$DevRTT = MDEV_1 = (1-\rho) \cdot DEVRTT_{i-1} + \rho \cdot |SampleRTT_1 - EstimatedRTT_{i-1}|$$

So, at the end, the TCP works like this: if we receive some data from an application, we create a segment with *seq #*, if the timer is not already running, start it (the timer is for the oldest unACKed segment; the expiration interval is $TimeOutInterval$). If we go in timeout, we retransmit the packet and restart the timer, if we receive an ACK that acknowledges previously unACKed segments, we update what is known to be ACKed and start timer if there are still unACKed segments.

Event at receiver	*TCP receiver action*
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. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

If RTO is too short, then we'll retransmit too early. If we are going to retransmit, it may mean that the network is congested, so we need to increase RTO; this is why we use an exponential backoff when we retransmit:

$$RTO_i = 2 \cdot RTO_{i-1}$$

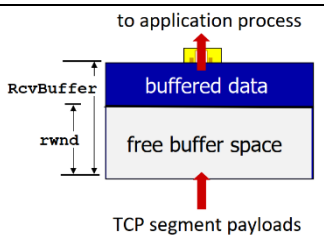
If there is no delay, but an incorrect transmission (so, the same as a NACK), after three ACK for the same data, we'll resend the segment with the smallest seq #.

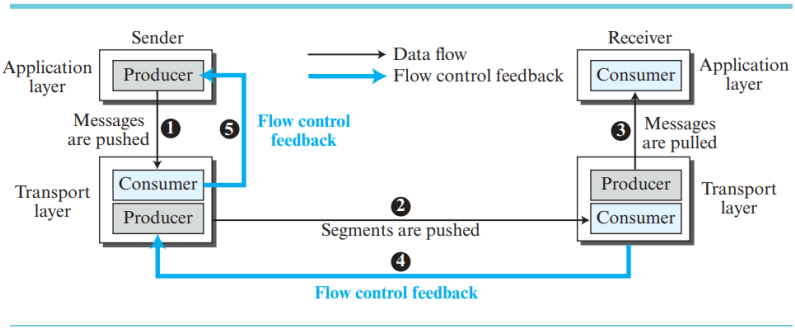
So, summarizing, the adopted mechanisms by TCP are:

- **Pipeline** (hybrid between Go-Back-N and Selective-Repeat);
- **Full-duplex** with piggybacking;
- Use of **Sequence Number** (first byte being transmitted);
- **Cumulative ACK** (it confirms all previous bytes) and delayed ACK (if all previous are ok);
- **RTT-based timeout**: unique retransmission timer (associated to the last unacknowledged segment). When an ack is received, the timer is restarted on the basis of the oldest unacknowledged segment;
- **Retransmission**:
  - **Single**-, only of the unacknowledged segment;
  - **Fast**-, when three acks are received before timeout.

Flow control

One of the components of the TCP header is the receive window or flow control, that prevent the overflow of the receiver's buffer. Indeed, it may happen that the sender transmits too many data too fast and the application process of the receiver is not able to remove data from socket buffers. TCP receiver advertises free buffer space in `rwnd` field in the TCP header, while `RcvBuffer` is the total size of the receiver's buffer and it's often auto adjusted by the OS. In this way the sender limits the amount of unACKed data.





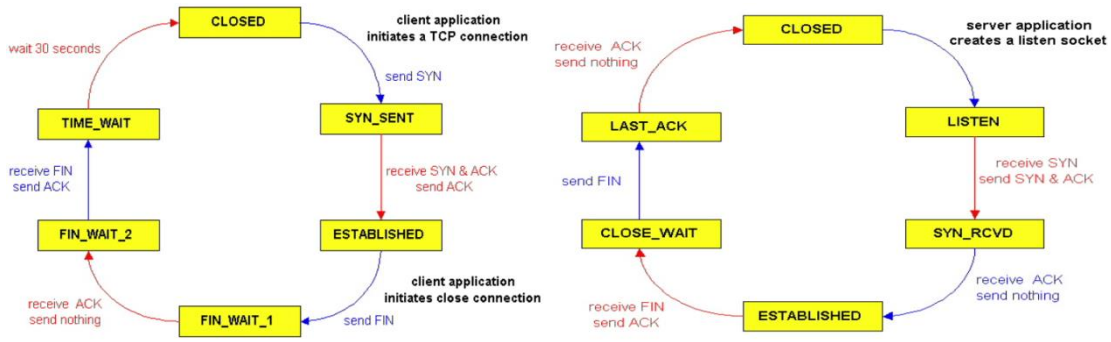
2-way handshake

There is also a **2-way handshake** (not TCP), in which the sender transmits a connection request and, after the receiver agrees, it starts to send data.

The problem with 2-way handshake is when you have losses or delays that cause retransmissions. In this last case we

Let's say that you have problems of delay in the network: you send the request but the acknowledgment is delayed, so you send again the request: after you receive the first acknowledgment you send the data and you do the same after the second acknowledgment, so you are sending the data twice without the receiver knowing it's a duplicate. With the three-way handshake, in case of delayed duplicate request, we can handle it because of the different sequence number.

The initial sequence number should change in time (according to RFC 793 we should generate ISN as a sample of a 32-bit counter incrementing at $4\mu s$ rate), be transmitted whenever we have the SYN flag active (both src and destination have their own ISN) and then, the Data Bytes are numbered from ISN+1.

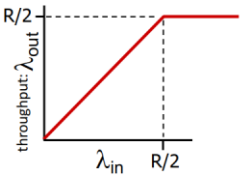


??? (immagine)

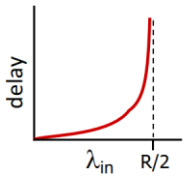
Congestion control

We talk about "**congestion**", when there are too many sources that are sending too much data too fast for the network to handle. We can experience this with long delays (queueing) or packet loss (buffer overflow). It's different from flow control because that one is referred only to a single sender and to a single receiver.

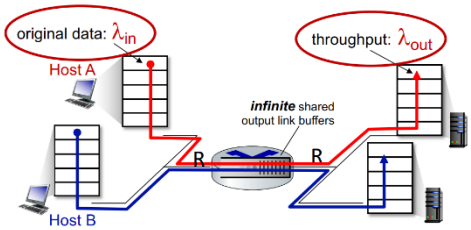
If we imagine a scenario in which we have a router with infinite buffers and each node has a link capacity for both input and output of R, then, when the arrival rate λ_{in} approaches $\frac{R}{2}$, we'll have:



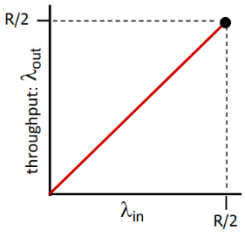
maximum per-connection throughput: R/2



large delays as arrival rate λ_{in} approaches capacity



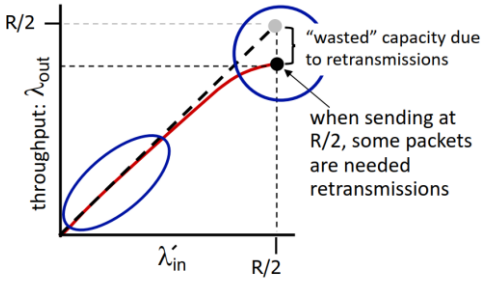
The limit is $\frac{R}{2}$ because the link is shared with both input and output.



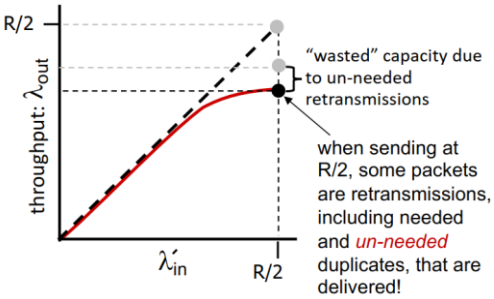
If we have only one router with finite buffers, then the sender will have to retransmit lost or timed-out packets. In this case, the application layer input is the same as the output $\lambda_{in} = \lambda_{out}$, while in the transport layer, because of retransmissions, we could have more packets sent so $\lambda'_{in} \geq \lambda_{in}$.

We can idealize this scenario with a perfect knowledge of the router, so, the sender knows exactly when the router is free and send the packets. In this case we have no problems with the throughput. So, the throughput can never exceed capacity.

If we have some perfect knowledge, so, the sender knows when the packet has been dropped and sends it again, we'll waste some time due to retransmission:

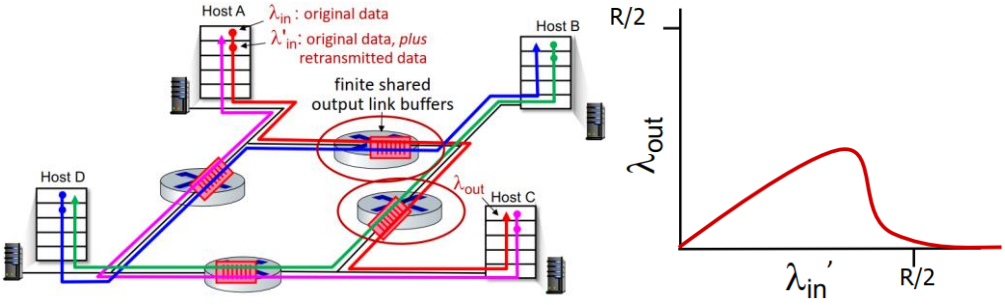


In the realistic scenario, the sender will send duplicates packets also for premature timeouts, so we spend other time in un-needed retransmissions:



The costs of congestion are the fact that the receiver will do more un-necessary work and the throughput will decrease.

If we have more paths, it may happen that a sender will block some routers, leading to the complete loss of packets of another sender:

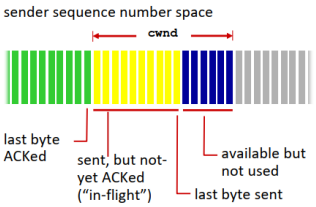


So, when packet dropped, any upstream transmission capacity and buffering used for that packet was wasted.

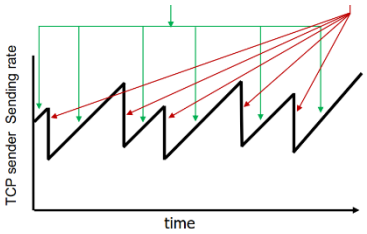
Notes by Davide Avolio, please do not obscure/remove this

An approach towards congestion control can be the TCP one, the **End-end congestion control**, in which the routers are not giving any information about the congestion and you can deduce it from the losses or delays. Another method is the **Network-assisted congestion control**, where the routers are providing direct feedback to sending/receiving hosts with flows passing through congested router. They might indicate the congestion level or explicitly set the sending rate.

To limit the sending rate, the sender fixes its windows size to the minimum between `cwnd` and `rwnd`. `cwnd` is the congestion window, while `rwnd` is the receiver window. The sender rate is bounded from above by $\frac{window\ size}{RTT}$, since it can send up to the window size and must wait an ACK before updating it.



The sender is able to detect congestion by looking at the arrival of ACKS: if it gets duplicate ACKs (weak indicator) or timeouts (strong indicator), then it reduces the window size, while if ACKs arrive with a reasonable frequency it can increase the rate by increasing the window size. For this reason, TCP is said to be self-clocking.



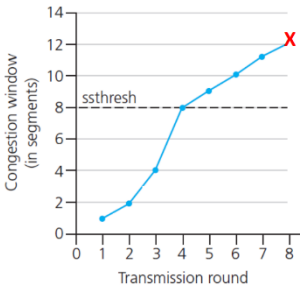
In order to regulate the window size, we use an algorithm. The most efficient one is the **AIMD** (Additive Increase, Multiplicative Decrease), in which we add 1 MSS (Maximum Segment Size) every time until we detect a loss and we cut in half the rate, getting a sawtooth graph. This algorithm is asynchronous.

In TCP Reno we cut in half on loss detected or by triple duplicate ACK; in **TCP Tahoe** and **Reno** we cut to 1 MSS when we get a loss after by timeout.

When controlling the congestion, the sender dynamically adjusts `cwnd` according to the network and limits the transmission (indeed $LastByteSent - LastByteAcked \leq cwnd$).

In TCP **slow start algorithm**, when connection begins, we increase rate exponentially. Indeed, for every ACK received, we increase `cwnd` of 1, so:

Start	→	$cwnd = 1 \rightarrow 2^0$
After 1 RTT	→	$cwnd = cwnd + 1 = 1 + 1 = 2 \rightarrow 2^1$
After 2 RTT	→	$cwnd = cwnd + 2 = 2 + 2 = 4 \rightarrow 2^2$
After 3 RTT	→	$cwnd = cwnd + 4 = 4 + 4 = 8 \rightarrow 2^3$



If we continue exponentially in this way, we'll certainly lose some packets, so we introduce a new variable, `ssthresh`: after the first loss, we set `ssthresh` to the half of `cwnd` value. When we'll start again, after the value of `ssthresh`, we'll proceed in a linear way instead of exponential. So, we decrease the growing rate of `cwnd` before getting a loss. The additive increase works in this way:

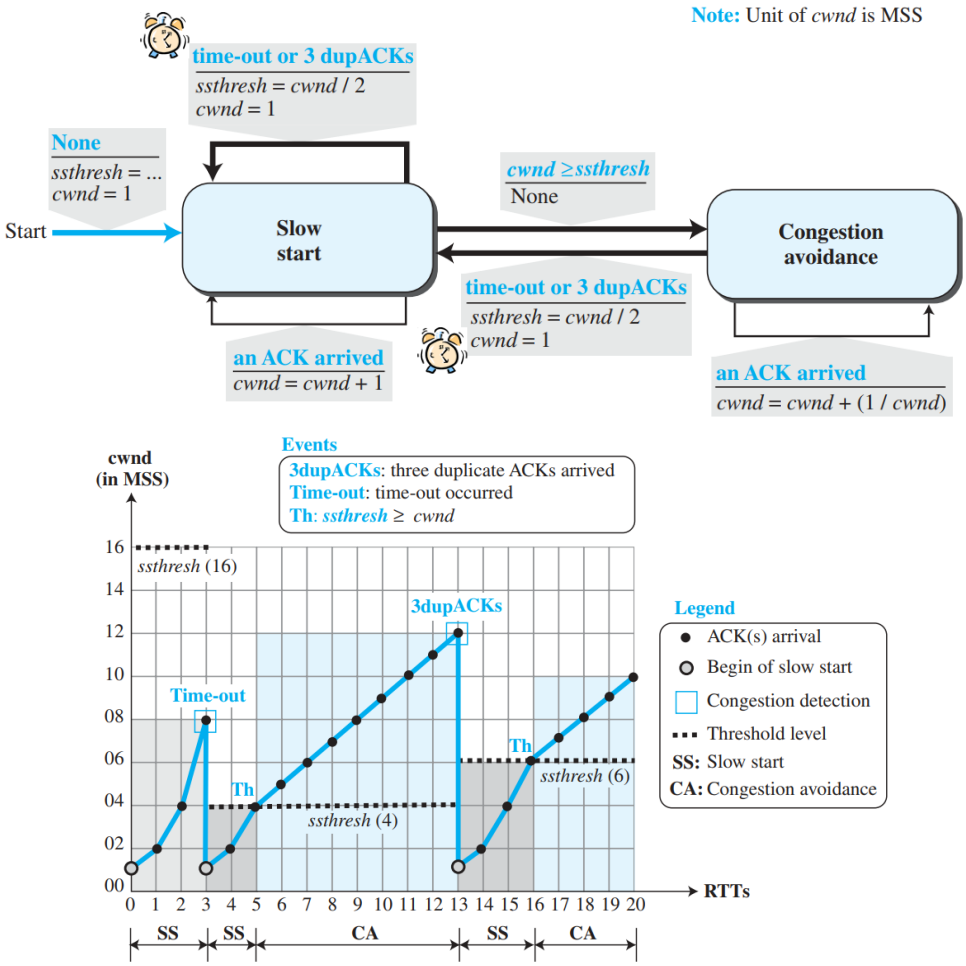
Start	→	$cwnd = i$
After 1 RTT	→	$cwnd = i + 1$
After 2 RTT	→	$cwnd = i + 2$
After 3 RTT	→	$cwnd = i + 3$

In the congestion phase we proceed in this way:

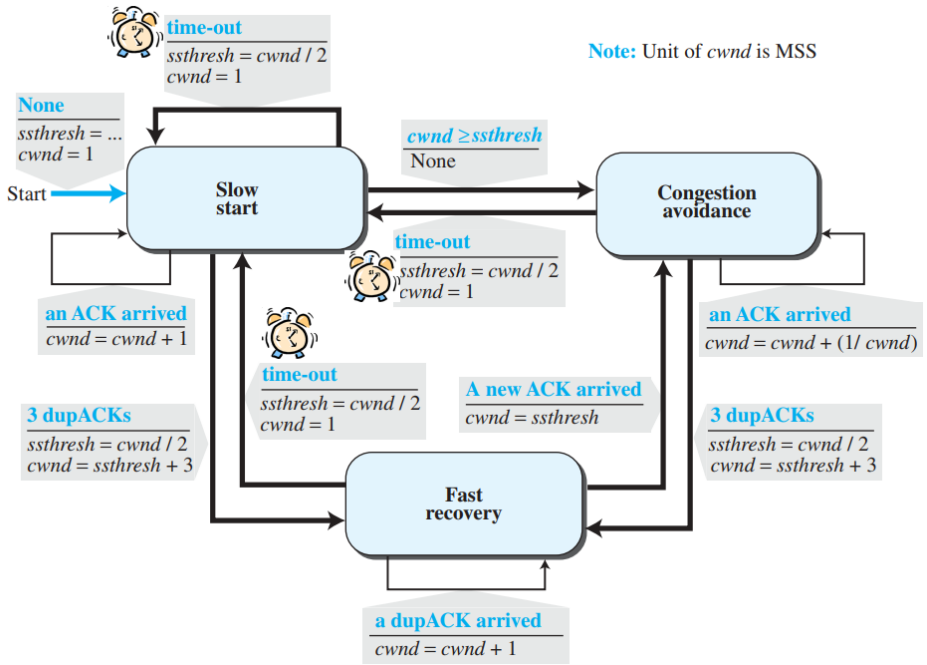
- Linear increase of `cwnd`.
- Stop linear increase of `cwnd` when we detected congestion (timeout or 3 duplicate ACKs)
- At a timeout event, `ssthresh`= `cwnd`/2 and `cwnd`=1
- After the third duplicate ACK, both in the slow start and in the congestion avoidance, it's up to the TCP implementation decide what to do:
    - **TCP Tahoe**: In the older version we consider a 3-duplicate ACK as a timeout, so we have only Slow start and Congestion Avoidance

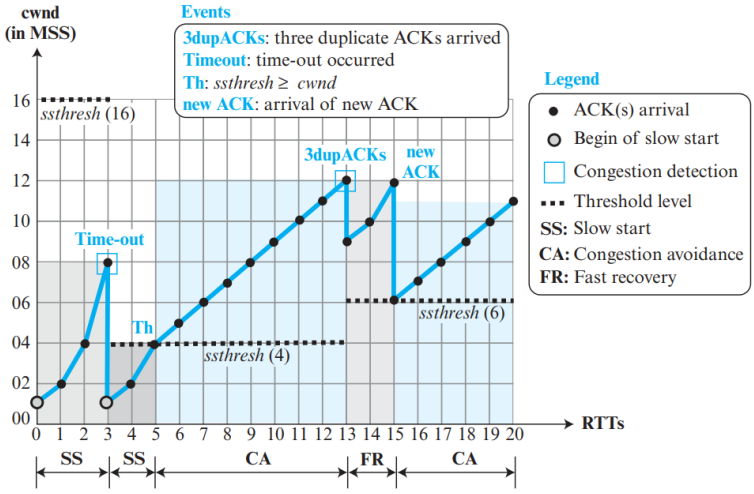
- **TCP Reno**: Here the 3-duplicate ACK is a lighter indicator of congestion rather than a timeout, so we have the fast recovery mode.

This is the structure of the **TCP Tahoe**:



In **Reno TCP**, when we find 3 duplicate ACKs, instead of starting back from `cwnd` = 1, we start from `cwnd` = `ssthresh` + 3, then with the same reasoning of slow start but from an already high rate and then with the congestion control:

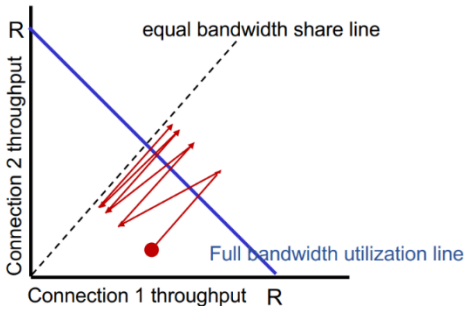




For the throughput we can ignore the slow start, because we assume we always have data to send, so we only are in the Congestion avoidance and Fast recovery phases. The lowest possible value is $\frac{W}{2}$ (because of the definition of `ssthresh`), while the highest is W (after that we'll go either in timeout or we'll get 3 duplicate ACKs), so the average is $\frac{3}{4}W$, for the throughput we have to divide by RTT and so we get an average throughput of $\frac{3}{4}\frac{W}{RTT}$.

TCP fairness

The goal is to have an average rate of $\frac{R}{k}$ with k TCP sessions. With two competing TCP sessions, with the same RTT and fixed number of session only in congestion avoidance, we've an additive increase and a multiplicative decrease, getting closer and closer to a 45° line. This means that their rates sum to the total R. In the graph below we start from the red circle and we see how after some steps, both connections get to the same throughput.



Anyway, fairness is not always wanted. In multimedia apps, we use UDP instead, because we don't want a throttled rate, but rather a constant one, even if we'll suffer from packet losses. This is why there is no "Internet police" for congestion control.

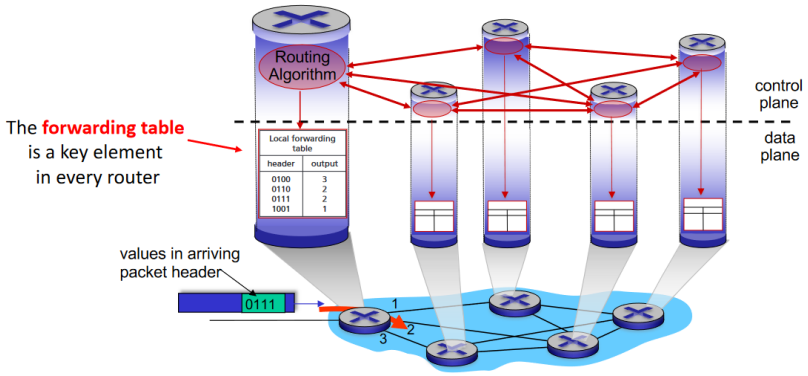
For fairness we have to consider also parallel connections: some applications can indeed open multiple parallel connections between two hosts and, in this way, they'll get more bandwidth (because in the network they'll result as different applications). For example, if we have already 9 existing connections and a new app asks for another TCP, we'll have 10 of them, each one with rate $\frac{R}{10}$. If another application comes and asks for 11 TCPs, it'll get $\frac{11}{21}R \sim \frac{1}{2}R$.

Network layer

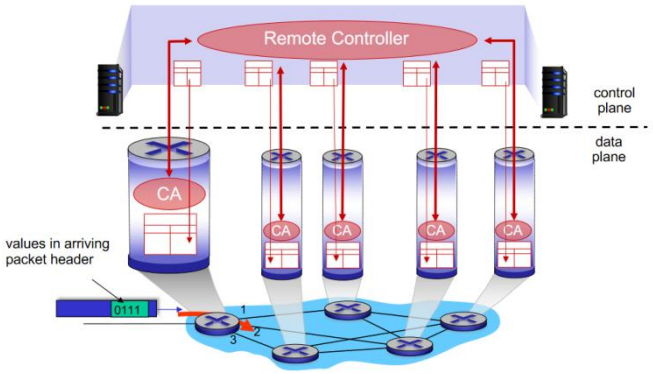
In the network layer we encapsulate segments into datagrams. The routers will examine header fields in all IP datagrams and will move datagrams from input ports to output, to transfer them along end-end path. Two key network-layer functions are **forwarding** and **routing**. These actions are performed by the **data plane** (local, it forwards an arriving router) and **control plane**, network-wide that determines how datagram is routed along end-end path from source host to destination host. In routers we implement the traditional routing algorithms, while in (remote) servers we implement the **software-defined networking** (SDN). The individual routing algorithm components in every router interact in the control plane. In SDN remote controller computes and installs forwarding tables in routers.

Notes by Davide Avolio, please do not obscure/remove this

Per-router control plane:



SDN:



Network service model

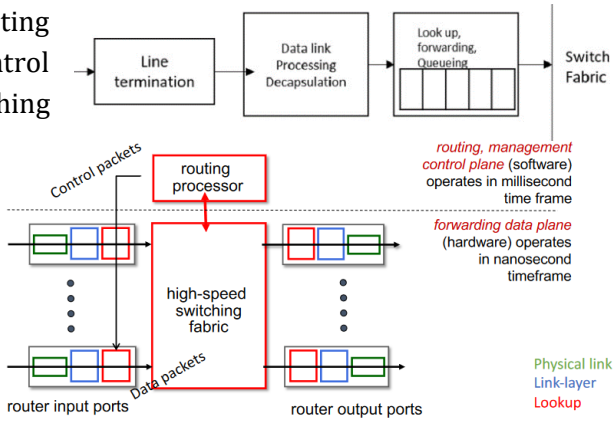
In channel transporting datagrams, we want to have some service models; for example, when sending individual datagrams, we want to guarantee delivery, or when we are sending a flow of datagrams, we want to have them in order, guarantee minimum bandwidth to flow and restrictions on changes in inter-packet spacing. The service model Internet with "best effort" doesn't guarantee datagram delivery to destination, timing/order of delivery and bandwidth available to end-end flow, but its mechanism's simplicity has allowed Internet to be widely deployed and adopted; the performance of real-time applications are "good enough" for "most of the time" with sufficient provisioning of bandwidth. Replicated, application-layer distributed services (datacentres, content distribution networks) connecting close to clients' networks, allow services to be provided from multiple locations.

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

Routers

The router is divided in 4 components: the routing processor manages in milliseconds routing and control plane at a software level; the high-speed switching fabric in nanoseconds decides the forwarding of the data; in the router input ports and output ports the data comes in and goes out.

The input port is divided in three zones: the packets go in the line termination and then in the data link processing, where the frame is removed



Notes by Davide Avolio, please do not obscure/remove this

and we are left with the datagram. These "spoiled" packets then go into the forwarding zone, where they're prepared to be switched. There's a queue in case of too many packets.

We have two different ways to compute the switching: in the **destination based** (traditional) we forward only with the destination IP address; with the **generalized forwarding** the forward is based on any set of header field values.

In the destination-based forwarding, we have to check in which range our IP will go:

forwarding table

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through 11001000 00010111 00010111 11111111	0
11001000 00010111 00011000 00000000 through 11001000 00010111 00011000 11111111	1
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

But not always we'll have simple ranges, indeed, if we want to add this range:

11001000 00010111 00010000 00000100 through 11001000 00010111 00010000 00000111	3

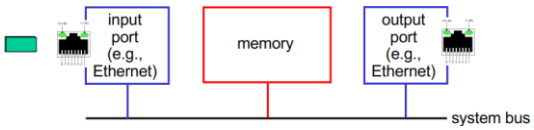
We've to split in two the range with the 0. That's why we pass to the **longest prefix matching**: we have a forwarding table only with part of the addresses. Each time that we get an IP we check if it matches the destination addresses in the table starting from the longest one to the shortest:

Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

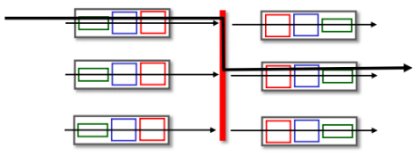
We perform the longest prefix matching by using **ternary content addressable memories** (TCAMs) that can retrieve the address in one clock cycle, regardless of table size.

Once that we know where to switch a packet, we have to compute the transferring to the proper output link. The switching fabric has a switching rate at which packets can be transferred from inputs to outputs, measured as a multiple of input/output line rate. If we have N inputs we would like to have a switch rate that is N times the line rate. We have three major types of switching fabrics.

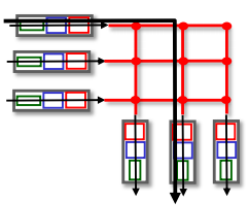
In switching **via memory** (first generation routers) we've a traditional computer with a CPU that is switching the packets. The packets are copied to system's memory and the speed is limited by memory bandwidth (2 bus crossing per datagram).

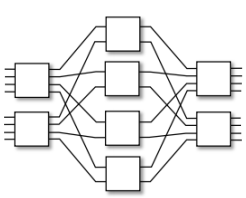


Another method is **via a shared bus**, but in this case the switching speed is limited by bus bandwidth.



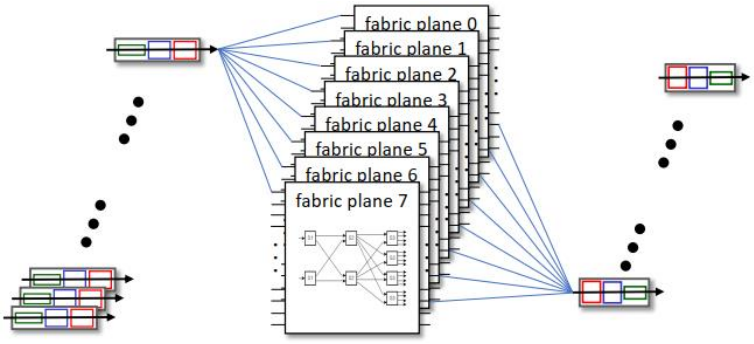
In the **interconnection network** we have an internal net made of switches. There are many types of internal networks. In the **crossbar** we have an $n \times n$ net. In the **multistage** switch we still have an $n \times n$ switch but with more interconnections. With this method we can exploit parallelism, because we split the datagrams into fixed length cells on entry and we reassemble it at exit.


3x3 crossbar

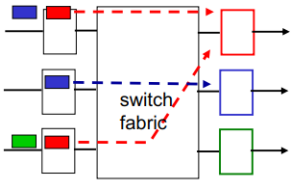

8x8 multistage switch
built from smaller-sized switches

Notes by Davide Avolio, please do not obscure/remove this

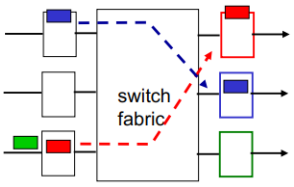
For this reason, we can use multiple switching planes in parallel (scaling).



In the input port we can have queuing for two different reasons: if two datagrams have to get out from the same output port, we have contention; if the switch fabric is too slow, the Head-of-the-Line (HOL) is blocked, so other datagrams can't move.

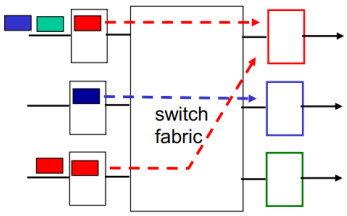


output port contention: only one red datagram can be transferred. lower red packet is *blocked*

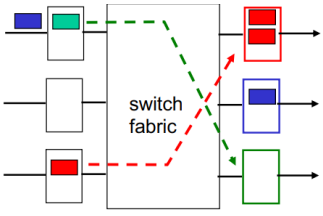


one packet time later: green packet experiences HOL blocking

In the same way we can have queuing in the output port: if two packets have to get out in the same port, we have buffering. If the line speed is too low, the packets will get lost because the output buffer is full.



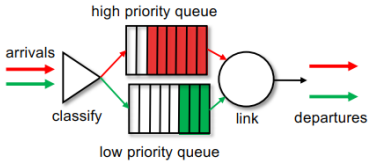
at *t*, packets more from input to output



one packet time later

The RFC 3439 rule of thumb states that the buffering should be $RTT \cdot C$, (C is the link capacity). A more recent recommendation is $\frac{RTT \cdot C}{\sqrt{N}}$, where N is the number of concurrent flows in a link. In any case we have to find a trade-off between delay and loss, because too much buffering may lead to a slower performance. When the buffer is full, we have to manage the congestion in the router: if we are about to drop a packet, we can either choose the last one (tail drop) or we can choose it by priority. We'll also mark some packets to signal congestion (ECN, RED). In this case we have to choose an algorithm to decide which packets should be sent next on the link:

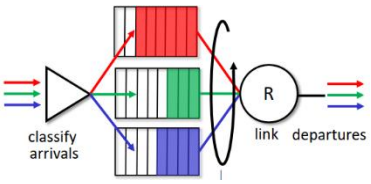
- **First come first served (FCFS)**: we transmit the packets in order of arrival to output port
- **Priority scheduling**: we classify the datagrams based on header fields and we start sending packets from highest priority queue (with FCFS) that has buffered packets.
- **Round Robin (RR)**: The arrivals are classified by header and the server cyclically scans the queues, sending one complete packet from each class (if available) in turn.
- **Weighted Fair Queuing (WFQ)**: generalized RR, where each class i has a weight w_i and gets weighted amount of service in each cycle $\frac{w_i}{\sum_j w_j}$. In this way we guarantee minimum bandwidth (per-traffic-class). So, each class will send a different number of packets per cycle.

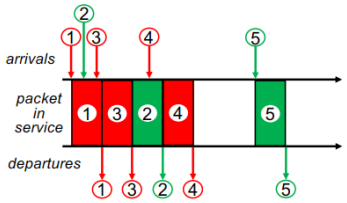


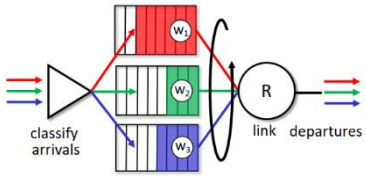
Left: priority scheduling.

Top right: RR.

Bottom right: WFQ.



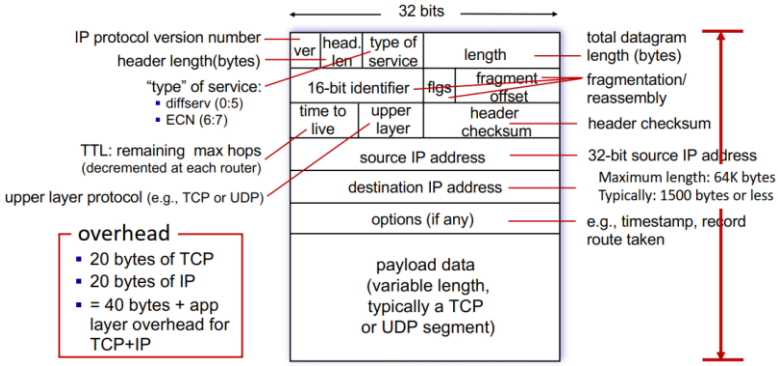




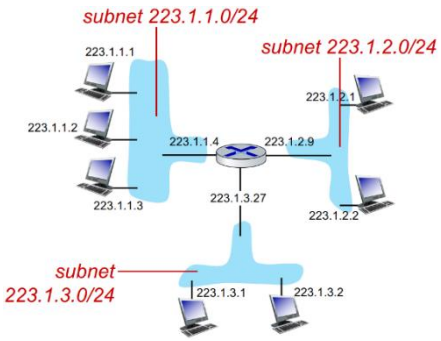
A network should maintain neutrality, so it should not block lawful content/applications/services or non-harmful devices; it shouldn't throttle or impair lawful Internet traffic; it should not engage in paid prioritization.

The Internet Protocol

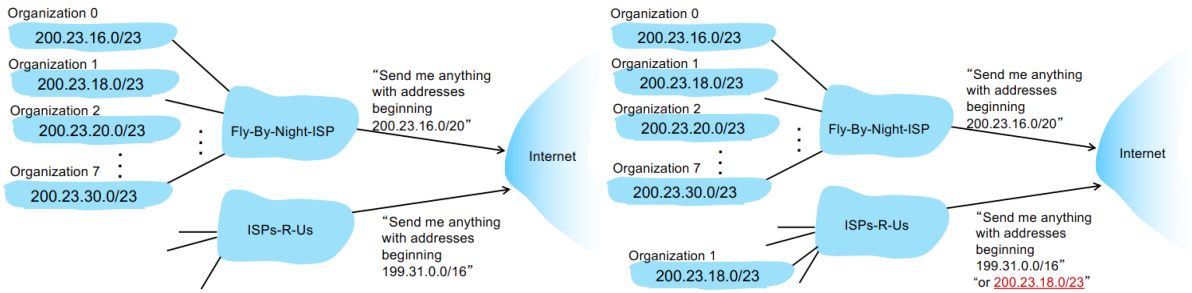
The functions of host and router in the network layer are the path-selection algorithms, forwarding tables, IP protocol and ICMP protocol. The format of an IP Datagram is:



In the connection between the host and the router (**interface**) we have to address the source and the destination with an IP address. Usually, the host has one or two interfaces, while the routers have multiple interfaces. The notation is the dotted one, so, four 8-bytes groups.



We call **subnet** a group of devices interfaces that are connected without passing through a router. In this case the IP addresses are divided in the subnet part for the high order bits and in the host part for the low order ones. The subnets are defined according to the CIDR (pronounced "cider"): we choose a number of bits *x* for the subnet part and we write it at the end, so, the format is a.b.c.d/x. In this way, we can easily manage the subnets and the ISP. Let's say that we have some organizations in an ISP, each with its IP address (the ISP will accept the requests from the internet whose IP is even more generic). If we want to move an organization from an ISP to another with a different IP, we just have to update the IP of the ISP by adding the new one:

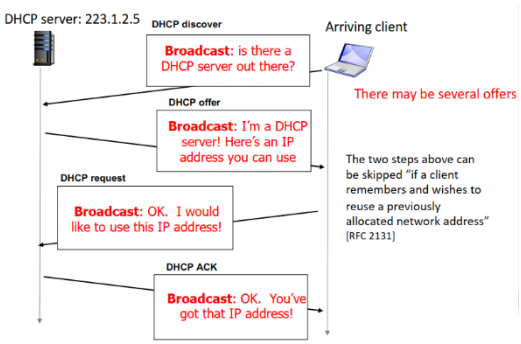


A host can get its IP address wither by hard-coding it by sysadmin in config file or dynamically with the **Dynamic Host Configuration Protocol** (DHCP) that will get the address from a server. A network

instead, will get its subnet part from a portion of the ISP's address space and the ISP will refer to the Internet Corporation for Assigned Names and Numbers (ICANN) the new IP.

When a new host join a network server, it'll communicate with the DCHP to get an IP (either a previous one or a new one). Because of the limited number of unique IPs, we can reuse addresses. DHCP is a client-server protocol. Typically, the DHCP server is co-located in a router and serves all the subnets to which the router is attached. If a new client is attached to the network, to get a new IP, it'll broadcast to all the devices nearby a DHCP discover message; if a DHCP catches it, it'll respond with an DHCP offer; after this the host can request an IP address and accept it.

The DHCP can also provide the address of the first-hop of router for the client, the name and IP of DNS server and the network mask (indicating the network versus host portion of address).



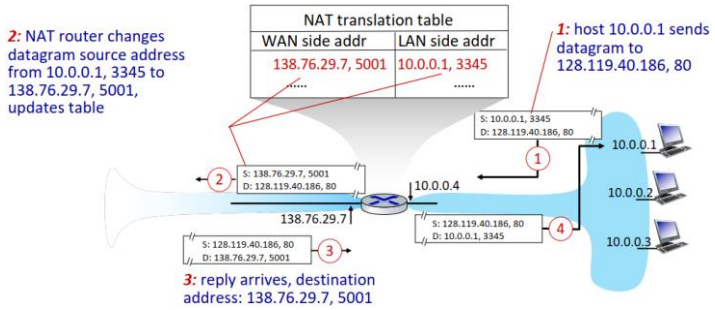
The ICANN allocates IP addresses through 5 regional registries (RRs) and manages DNS root zone, including delegation of individual TLD management. The ICANN allocated last chunk of IPv4 in 2011, so, very slowly in time we're passing to IPv6, with 128-bit address space.

NAT: network address translation

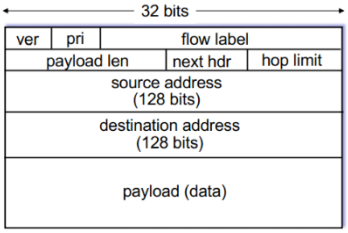
In a local network it's more convenient to have a shared IP for all the devices connected. In the local space, each device has its own IP address, while in the "outside world", we just need one single IP address for all devices in the same ISP. In this way we can:

- It's easier to change addresses of host in local network without notifying outside world;
- It's easy to change ISP without changing all the addresses of devices in local network;
- More security: the devices inside local net, are not directly addressable.

To do this, we use the NAT router, that'll replace all the outgoing datagrams (source IP address, port #) with (NAT IP address, new port #) and store these two information in a translation table so that, when receiving an incoming datagram, it can efficiently replace the original source IP address, thanks to the new port #. Because of the different usage of port # (to search for a host in the translation table instead of identifying a process) NAT has been controversial.

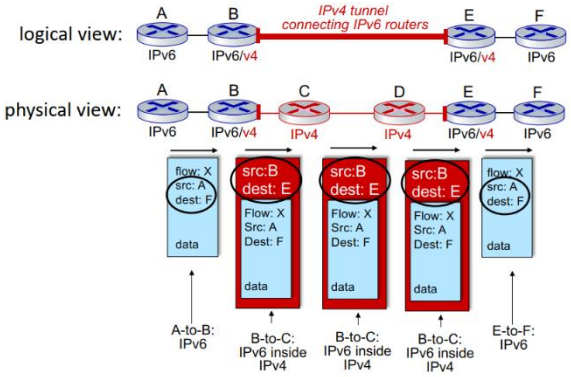


Since network links have an MTU (Maximum Transfer Unit), if we are going to send a large datagram, we have to split it into several smaller pieces. In this case we'll assert the `fragflag` and we'll update the offset. By a convention the offset is measured in octets, so we divide the length of the datagram by 8. Thanks to the offset we can recollect in the right order the pieces.



Since the IPv4 address space is completely allocated and, in order to speed processing/forwarding and to enable different network-layer treatment of "flows", the IPv6 was created. The header in this case is slightly smaller than the IPv4:

Since we can't upgrade all the routers simultaneously, we recur to tunnelling and encapsulation to deliver a message from an IPv6 router through an IPv4 one (it's also used in contexts like 4G/5G). In the moment in which an IPv6 datagram is going to enter into an IPv4 router, it'll be encapsulated inside a IPv4 datagram, it'll pass through an IPv4 tunnel and finally the IPv4 header is removed and we have back the original datagram.



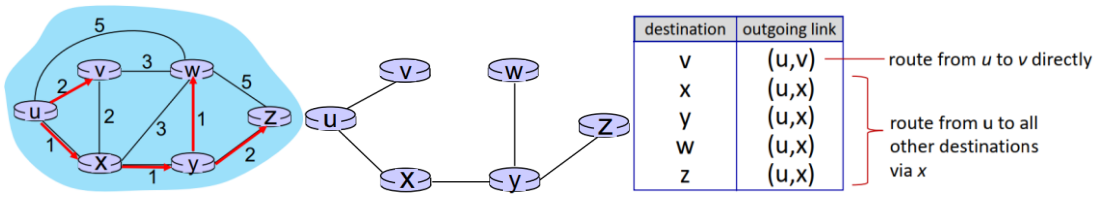
Routing protocols

An important goal in networking is finding the cheapest, fastest and least congested path between two nodes. We can represent all the network as a graph, where we have a set of nodes and a set of edges with all the possible combinations of nodes and each edge has a weight (if two nodes are not connected, the weight is infinity). We can classify routers as static/dynamic (depending on the frequency of change) and as global/decentralized (depending if all routers have to know complete topology or if they only exchange information with neighbours).

Dijkstra's link state routing algorithm: This algorithm is centralized with via "link state broadcast" and all nodes have same info. Once we know all the costs, then we can compute from one node the cheapest path, from the source to a host. This algorithm is iterative because after k iteration we know at least the path of k destinations.

```
Initialization:
N' = {u} # compute least cost path from u to all other nodes
for all nodes v
    if v adjacent to u #u initially knows direct-path-cost only to direct neighbors
        then D(v) = cu,v # but may not be minimum cost!
    else D(v) = ∞
Loop
    find w not in N' such that D(w) is a minimum
    add w to N'
    update D(v) for all v adjacent to w and not in N' :
        D(v) = min ( D(v), D(w) + cw,v )
    # new least-path-cost to v is either old least-cost-path to v or known
    # least-cost-path to w plus direct-cost from w to v
until all nodes in N'
```

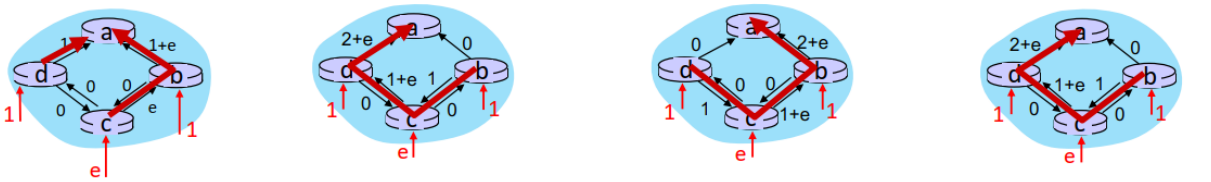
In this way, once we have completed the execution of the algorithm, we can have the forwarding tables for all the nodes. For example, if we start from this graph, we'll get the following least-cost-path tree from the node u and, in that node, we'll get the following forwarding table:



The algorithm complexity is $O(n^2)$ because at each iteration we have to check all the nodes that are not in N (a total of $\frac{n(n+1)}{2}$ comparisons). With a more efficient implementation we can reach $O(n \log n)$. Each router has to broadcast its link state information to other n routes; an efficient algorithm allows $O(n)$ link crossings to disseminate a broadcast message from one source; each router's message will cross $O(n)$ links, so we get a total complexity of $O(n^2)$ for the messages.

Notes by Davide Avolio, please do not obscure/remove this

If the link cost depends on traffic volume, we'll get route oscillations:



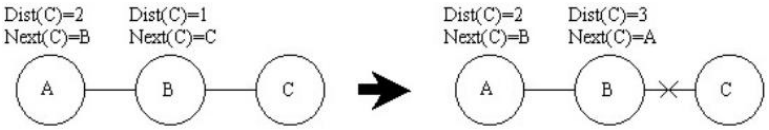
Distance vector algorithm: Based on the Bellman-Ford equation we can develop the distance vector algorithm; in particular, if $D_x(y)$ is the cost of least cost path from x to y, then:

$$D_x(y) = \min_v\{c_{x,y} + D_v(y)\}$$

Each node will wait for a change in local link cost or a message from neighbour; in case it receives a change, it will recompute the distance vector (DV) and then it'll notify the neighbours about the change only if the DV has changed.

At time one a node will receive from the others information about their neighbours with the weights of the links if the two nodes are connected or the value of infinity if they're not. After it gets all the DV from the others, it applies the Bellman-Ford equation and then at the second step, it'll notify all the neighbours with the updated DV. After some hops we have advertised all the nodes, so, unless a weight changes, the algorithm stops.

A problem with this algorithm is the case of loops: if a weight is decreased there're no problems, but if the weight is worse, the nodes will constantly exchange false information, until they reach a false high distance. The problem is solved with poisoned reverse. Here, if A is reaching C passing through B, it should not notify B of this change (we poison the notification to B).



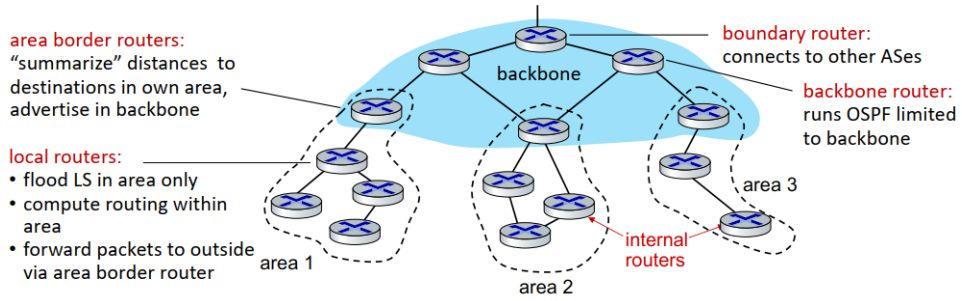
If we have three nodes x, y, z and the cost (x, y) changes, then y will say to z that it has changed, z will say to y that it has changed and so on. With poisoned reverse, after the first notification, z will say to y that its path is of ∞ directly, then computes the real path with the real weight and notifies it to y. y on reverse will update its path, notifying to z the change and that the new cost for passing through it is ∞.

Hierarchical routing

In a real network, forwarding tables would be too large to be stored if they contained all the routers, so we divide the network in smaller networks/**autonomous systems** (AS)/domains. In the **intra-AS** all routers must run the same intra-protocol (it could be different among different AS'es) and there is a **gateway router** that links other router(s) of other AS'es. In the **inter-AS** we route among AS'es (with the gateway router that performs both types of routing). Here the admin wants to control how its traffic is routed and who's routing through its network, while in the intra-AS we have less policies. Also, in the intra-AS we can focus on performance, while in the inter-AS policy dominates over performance. We have many types of intra-AS routing:

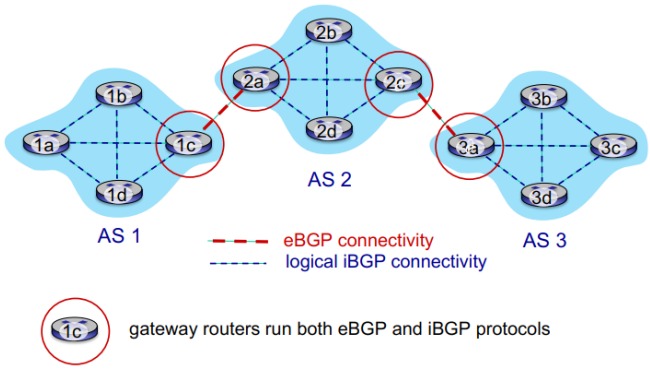
- **Routing Information Protocol** (RIP): the classic DV (DVs exchanged every 30 seconds), so **Bellman-Ford**: every router asks for the tables of the neighbours, if something changes, they send their updated table and if a router doesn't receive an update for 6 periods, then it'll set the weight to infinity. It's no longer used.
- **Enhanced Interior Gateway Routing Protocol** (EIGRP): it's **DV** based too.

- **Open Shortest Path First** (OSPF): it's open and a link state routing: each router floods the OSPF link-state (with the IP instead of the TCP/UDP). All OSPF messages are authenticated to prevent malicious intrusion. We have a two-level hierarchy: the smaller networks are the local areas and there is a zone called **backbone** that connects all the local areas. Each router has full topology of its area thanks to **Dijkstra** and only knows the direction to reach other destinations. In OSPF we have also an equal-cost multipath.



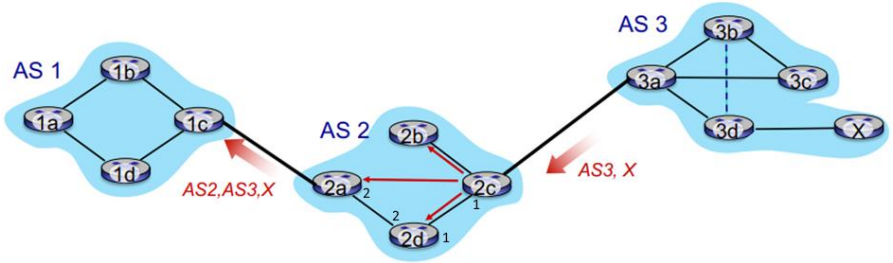
In the inter-AS routing we have the **Border Gateway Protocol** (BGP). BGP allows all the subnets to advertise their existence. It provides each AS to:

- **eBGP**: obtain subnet reachability information from neighbouring AS'es.
- **iBGP**: propagate reachability information to all AS-internal routers.
- Determine good routes to other networks based on reachability information and policy.



In BGP we've two BGP routers ("peers") that exchange BGP messages over semi-permanent TCP connection. The routers are advertising paths to different destination network prefixes (they guarantee each other that they'll send the datagrams towards the route they've advertised).

When we advertise a route, we give a prefix (the destination being advertised) and 2 attributes, **AS-PATH** (list of AS'es through which prefix advertisement has passed) and **NEXT-HOP** (the specific internal AS-router to next-hop AS). The gateway receiving route advertisement uses import policy to accept/decline path (for example, never pass through this subnet) and also the AS policy determines whether to advertise path to other neighbouring AS'es.



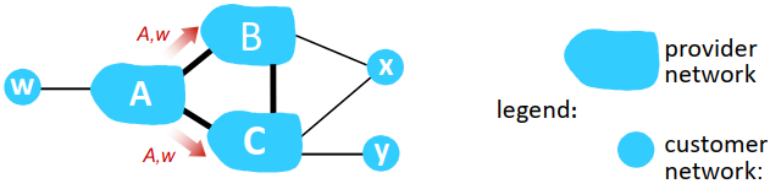
In this example, AS2 router 2c receives via eBGP path advertisement AS3, X from AS3 router 3a. Based on AS2 policy, router 2c accepts path AS3, X and propagates via iBGP to all AS2 routers. Based on AS2 policy, AS2 router 2a advertises via eBGP path AS2, AS3, X to AS1 router 1c. If a router discovers more paths, we can choose one with policy. In this example, to reach 2c the router 2d has to

choose the interface number 1, while from 2a it's the number 2. If there are more paths to reach a destination, we can apply the **hot potato routing**: we choose the local gateway that has least intra-domain cost, independently of the inter-domain cost.

There are different types of BGP messages exchanged between peers over TCP connection:

- `OPEN`: opens TCP connection to remote BGP peer and authenticates sending BGP peer.
- `UPDATE`: advertises new path (or withdraws old).
- `KEEPALIVE`: keeps connection alive in absence of UPDATES; also, ACKs OPEN request.
- `NOTIFICATION`: reports errors in the previous message; also, to close connection.

In a real-world network, we've customer and provider networks. The aim is to not carry traffic between other ISPs, so, whenever we advertise a path to a provider network, it'll only advertise to a customer network and not to other provider networks (here, B doesn't advertise BAw to C). Here x is **dual-homed** because it's attached to two networks. By policy to enforce, since x doesn't want to be in the middle of a route from B to C, it won't advertise to B a route to C.



When we discover more than one route to destination AS, we can choose the route based on:

1. Local preference value attribute: policy decision.
2. Shortest AS-PATH.
3. Closest NEXT-HOP router (hot potato routing).
4. Additional criteria.

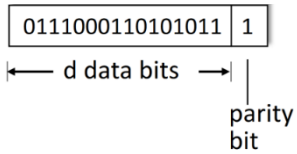
Link layer

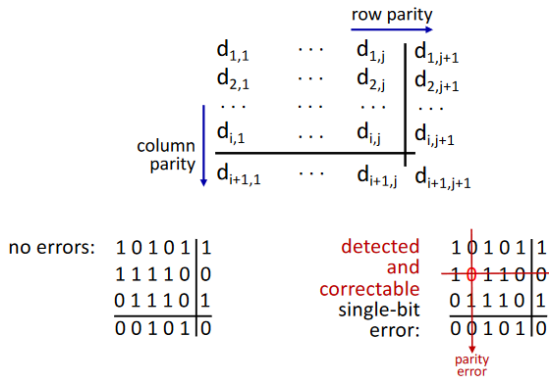
The datagram is transferred by different link protocols over different link, each one providing a different service. The services are:

- **Framing**: encapsulate datagram into frame (header + tailer)
- **Link access**: in case of shared medium (shared cables in the first versions of Ethernet). We've the **Medium Access Control** (MAC) address to distinguish source and destination.
- **Reliable delivery between adjacent nodes**: seldom used on low bit-error links. Usually, the wireless links are more prone to have high error rates.
- **Flow control**: pacing between adjacent and receiving nodes, so enough data for receiver (but we don't have congestion control, because it's not up to the link layer to control it).
- **Error detection**: we can have error for signal attenuation or noise. The receiver detects the errors and decides if retransmit or drop frame.
- **Error correction**: the receiver corrects the bit error without retransmission.
- **Half-duplex and full duplex**: With half-duplex, both links can transmit but not at the same time.

The link layer is implemented in every host on a chip (network adapter) or inside the **network interface card** (NIC), a combination of hardware, software and firmware attached into host's system buses. When two interfaces are communicating, the sender encapsulates the datagram in frame, adding checking bits, reliable data transfer, flow control, etc., while the receiving side, looks for errors, reliable data transfer, etc and it extracts the datagram, passing it to the upper layers.

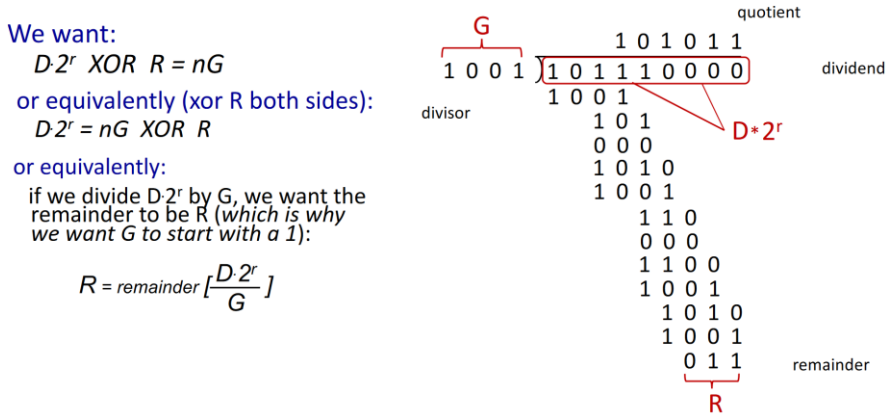
We can detect error by using some error detection and redundant correction bits. When the receiver gets some data, it checks if all the bits are fine according to the **EDC**. The larger is the EDC, the better detection we have, but still, this method does not guarantee a 100% of error detection. With the **parity checking** we always want that the number of 1 in the data is even. We can also use a two-dimensional bit parity, where we check both rows and columns.





In the internet checksum, the sender treats contents of UDP segment as a sequence of 16-bit integer, then it computes the checksum and it puts it inside the UDP checksum field. The receiver computes the checksum and it checks if it's equal to the field value. If it's not equal, we have an error, if not, then we still have to check,

A more powerful tool is the **Cyclic Redundancy Check** (CRC). We take the data to check D and we want to add some r bits (that will make a number R) such that $\langle D, R\rangle = D \cdot 2^r \; XOR \; R$ is divisible by a fixed number in advance G. G is made of $r+1$ bits and the receiver can detect for sure any error smaller of $r+1$ bits, by making again the division (if it gets a remainder, then there's an error). The division that we are going to do is the modulo 2 division. In the modulo 2 arithmetic, we only do XORS, so in addition and subtractions we don't have any kind of carry.



If we want to find R we have to compute the modulo 2 division between $D \cdot 2^r$ and G.

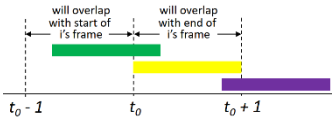
Multiple access links

There exist two types of links: in the **point-to-point** there is a communication between two single nodes, while in the **broadcast** we are sharing a wire (the old Ethernet) or a medium (like in 4G/5G, in Wi-Fi or in satellite, all shared radio). In the broadcast we have a single shared channel, with some nodes connected to it. If two or more nodes transmit at the same time, we get an **interference** and if a node receives more than two signals, we have a **collision**. In the protocol we have an algorithm that determines how nodes share channel and who can transmit; the communication about channel sharing must use the channel itself and no out-of-band channels. In an ideal multiple access protocol (MAC) with a rate R, we want a rate R if a single node is transmitting and an average of $\frac{R}{M}$ if M nodes are transmitting. It must be simple and fully decentralized, so no special nodes that coordinate

transmissions and no synchronization of clocks or slots. Effectively this is not possible, so we have three classes of MAC protocols:

- **Channel partitioning**: we divide the channel into smaller pieces and we give each piece exclusively to a node. There are two types:
  - **Time Division Multiple Access** (TDMA): we divide the access to the channel in rounds and each station gets fixed length slot. The unused slots go idle.
  - **Frequency Division Multiple Access** (FDMA): the channel spectrum is divided into frequency bands, each one given to a node. The unused go idle.
- **Random access**: the channel is not divided and when there are collisions, we recover from them. There are many random-access MAC protocols:
  - **Slotted ALOHA**: all frames have the same size; nodes are synchronized and start to transmit only at the beginning of a slot. If two or more nodes transmit in the same slot all nodes detect collision before the end of the slot. If there is no collision the node can send new frame in next slot, or if there is a collision it'll retransmit with a probability p.

    The Pros are: if there is a single node, it'll transmit at full channel rate, it's simple and highly decentralized, because we need synchronization only in slots. The Cons are: collisions, wasted and idle slots, clock synchronization and the detection of collisions before the transmission. If we have N nodes the probability of a single one to have success in a slot is $p(1-p)^{N-1}$. For any node is $Np(1-p)^{N-1}$. If we maximize this number for $N \to \infty$ we get a probability of $\frac{1}{e} \approx 0{,}37$ (max efficiency).
  - **Pure ALOHA**: Unslotted ALOHA, so no synchronization. The collision probability increases because a frame sent at t_0 may collide with other frames sent in $[t_0 - 1, t_0 + 1]$. The probability is now:

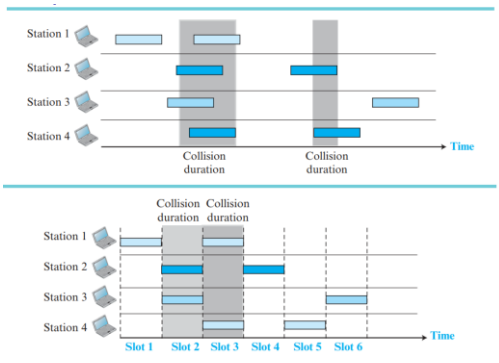


$$P(node\ transmits) \cdot P(no\ other\ node\ transmits\ in\ [t_0 - 1, t_0]) \cdot P(no\ other\ node\ transmits\ in\ [t_0, t_0 + 1])$$

    So, $p \cdot (1-p)^{2(N-1)}$. The optimum p is $\frac{1}{2e} \approx 0{,}18$.

    Another protocol is the **backoff** one: a station has a frame to send, it sends the frame and waits for $2 \cdot T_p$ (propagation time). If it receives an ACK, then it's a success, or else, it increases the counter of attempts K. If $K < K_{max}$, then it chooses an $R \in [0{,}2^K - 1]$ and then it waits (the backoff time T_B) for $R \cdot T_p$ or $R \cdot T_{fr}$ (average transmission time).

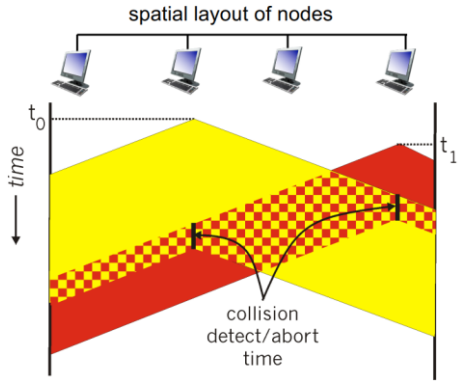
    The vulnerability time is $2 \cdot T_{fr}$, while in the slotted one is T_{fr}.



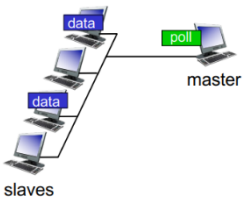
  - **Carrier Sense Multiple Access** (CSMA): here we "listen" before transmitting. If the channel is sensed idle, we send the entire frame, if it's sensed busy, we defer

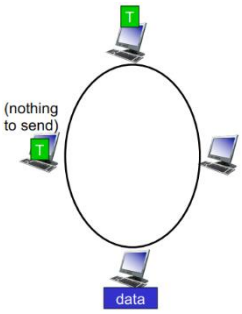
transmission. In the **CSMA/CD**, we detect collisions within short time, reducing channel wastage (easy in wired, difficult with wireless). Here we still can have collisions, because two nodes may not hear each other's just-started transmission or because of delays and distances. In CSMA/CD the time wasted in collisions is reduced because in the moment in which a transmitting node detects a collision, we abort the transmission. The vulnerable time is the same as the propagation time, so it increases with the distance. Upon collision detection, a station sends a jamming signal. If a station doesn't detect any collision, it doesn't store a copy of the frames it sent and it doesn't check for existing collisions. This is why the minimum frame size must be at least of $2 \cdot T_p$. In the worst-case scenario, we finish sending the frame to the destination and in that moment (so after T_p time) the destination starts sending a frame. After another T_p time, we are able to detect the collision because we're finishing in that moment the frame transmission, so we are able to store a copy of the frame and to send it again. The number of bits we are sending is $bandwidth \cdot 2 \cdot T_{fr}$.

  - **Ethernet CSMA/CD algorithm**: In this case the passages are the same. After a collision detection we apply the binary (exponential) backoff, so the more collisions, the more backoff interval.



- **Taking turns**: Nodes take turns, but the ones that wants to send more, will take more time. While the Channelling is efficient only at high load and the random access at low load, here we take the best. We can have:
  - **Polling**: There's a **master** that invites other nodes to transmit in turn. With too many requests, we have a polling overhead. We also have a single point of failure, the master.
  - **Token passing**: There's a **control token** that is passed from one node to the next one sequentially. Who has a token can start transmitting. The problems are again the overhead, latency and single point of failure, the token.

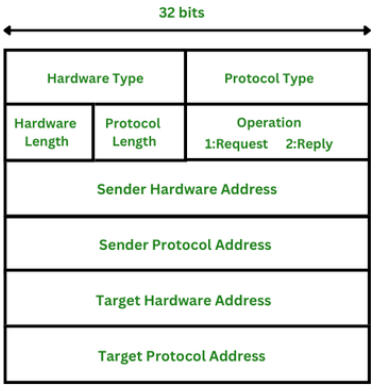




MAC addresses

Each interface has its own MAC address, that is burnt in NIC ROM (sometimes it's also software settable). While the IP is 32-bit long, it's used for forwarding in the network layer and it may change (it's unique only at a local level), the MAC address is 48-bit long (AA-AA-AA-AA-AA-AA, where each "A" is a hexadecimal number, so it carries 4 bits) and it's unique. It's used locally to get frame from one interface to another. The allocation of MAC addresses is administered by IEEE: manufacturers buy portions of MAC address space. The MAC flat address is portable, so it can be moved from one LAN to another (unlike IP addresses).

Address resolution protocol (ARP)



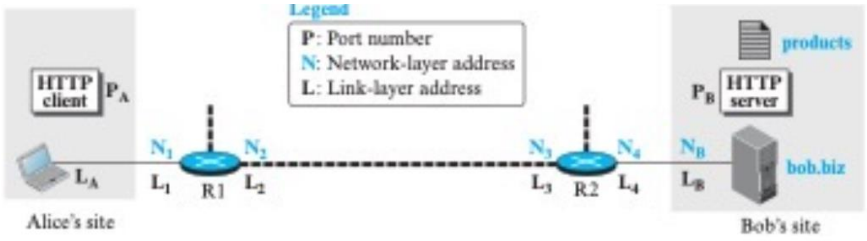
Each IP node (host, router, LAN) has an ARP table, that maps the IP address with the MAC. An entry in this table is: $\langle IP\ address; MAC\ address; TTL \rangle$, with TTL being the Time To Live, so time after which the address map is forgotten, so it has to be renewed. If we want to communicate with a node whose MAC address is unknown because is not in the table, the sender broadcasts the query to all the nodes in the LAN, using as MAC address FF-FF-FF-FF-FF-FF.

If we are referring to a node of another subnet, then we'll put as destination MAC address, the MAC address of the router. The router, when reading the frame, will remove the header and put a new one with the correct MAC address.

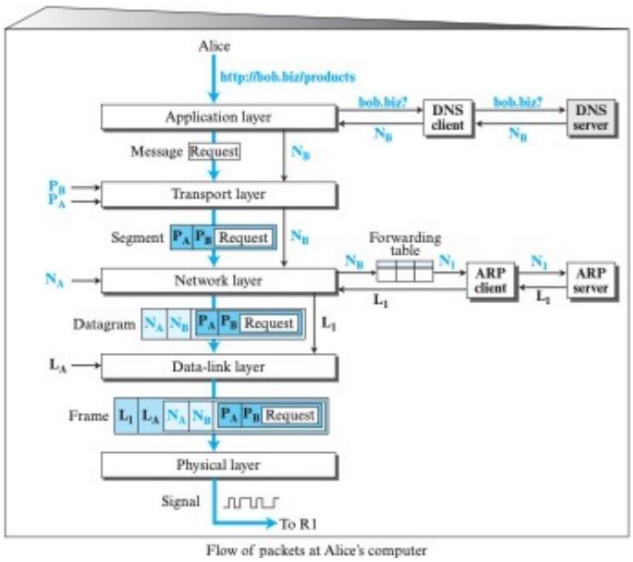
The packet format of an ARP is:

Example of address management in an HTTP interaction

1:

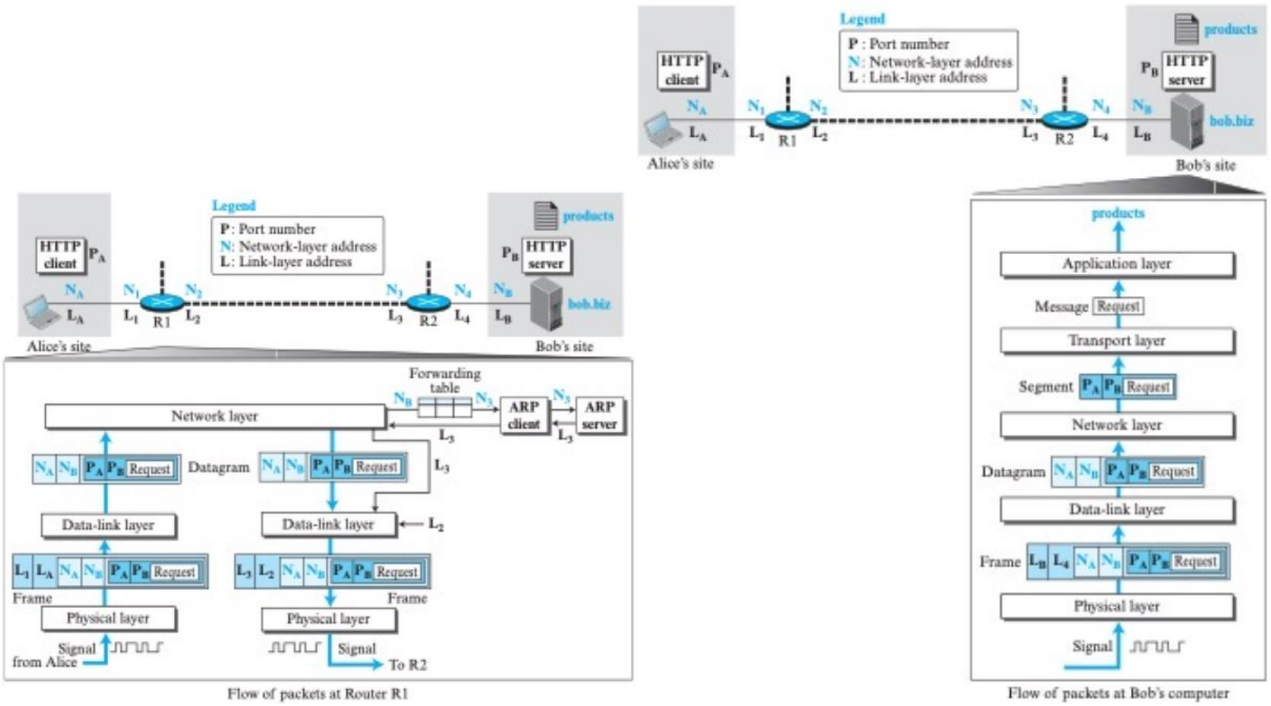


2:

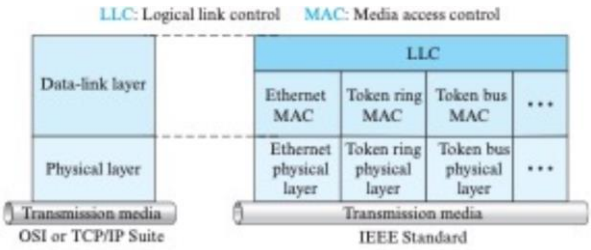


Flow of packets at Alice's computer

3 and 4:



The standards for LANs are:

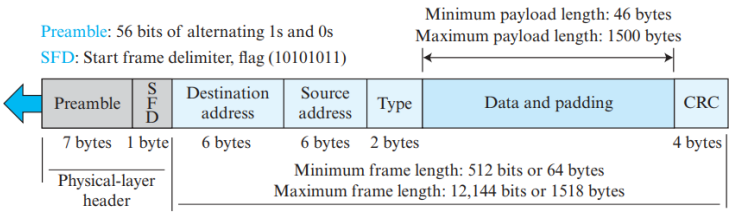


Ethernet

There are two ways of using Ethernet: the old one is with the bus, so all the nodes in the same collision domain. The modern one is the switched, in which there is a switch in centre and all the nodes are connected to it through separate Ethernet protocols, so they don't collide.

The Ethernet frame structure is made of 18 bytes: 6 for the destination MAC address, 6 for the source MAC address, 2 for the type (the protocol of the higher layer, so, IP or ARP, OSPF, used to demultiplex up at receiver) and 4 for the CRC, the redundancy check. If an error is detected, the frame is dropped. The data length goes from 46 to 1500 bytes, so the frame is from 64 to 1518.

In the physical layer we add other 8 bytes of header, the **Preamble**, that is a sequence of 10101010 repeated 7 times and the **SFD** (Start frame delimiter), a flag of 10101011.



When transmitting a frame to other nodes, we have three methods:

- Unicast: all stations receive the frame but only the intended one keeps it and handles it, while the others discard it. Since we are in the Ethernet, all the nodes will receive the message.

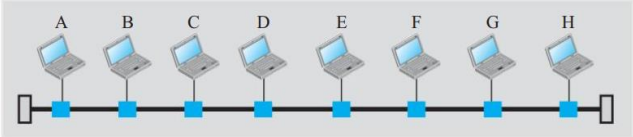
- **Multicast**: all stations receive the frame but only a group keep it and handle it, while the others discard it.
- **Broadcast**: all stations receive the frame (except the sender), keep and handle it.

If we want to distinguish between a unicast, multicast and broadcast, we have to read the reversed MAC address. We take the single bytes, we reverse the order (so, for example, from 1B we go to 00011011 and then to 11011000) and we check the new first bit (so, in the normal order, the last bit of the second letter from left). If it's 0, it's a unicast transmission (this is the real MAC address), if it's all 1, it's a broadcast, if it starts with 1 and the other digits are not all 1, it's a multicast.

The properties of Ethernet are: **connectionless** (no handshaking between sending and receiving NICs), **unreliable**, it's not up to the Ethernet to manage the data from dropped (so no ACK/NAK and if the higher layer rdt is not like TCP, the data is lost)). The Ethernet's MAC protocol is the unslotted CSMA/CD with binary backoff. We have many different Ethernet standards, that have different speeds and layer media, but same MAC protocol and frame format.

Switches

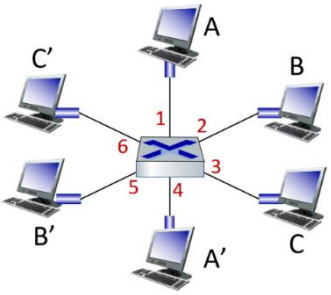
A type of interconnection devices is the LAN with bus topology using a coaxial cable:



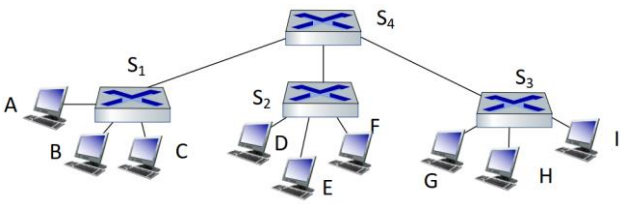
In a LAN with a star topology, instead, we may use a **hub** (so it's a broadcast LAN and it may suffer from collisions), a physical layer device that acts on individual bits (not frames). It repeats (boosts) the received signal to all its outgoing interfaces.

Another physical layer device is the **switch**: it has an active role because it stores and forward Ethernet frames to one or more outgoing links by looking at the MAC address. When the link has to forward to the segment (a physical subnet of devices, that can be made of one or more nodes in case of switches of switches), it uses the CSMA/CD. The switches are transparent to nodes, that don't know about their presence and they're also plug-and-play and self-learning. They can handle multiple simultaneous transmissions to different nodes. They can buffer packets and since they've an Ethernet protocol, we don't have collisions between different links (it's full duplex), but only on the same.

Each switch has a **switch table** that contains the MAC address of the host, the interface to reach it (the numbers in the image) and a time stamp. After the time stamp, we delete the entry. The switch table is initially empty and whenever a node is transmitting to the switch, the switch saves its MAC address and interface into the switch table. Whenever the switch doesn't have an entry for a host, it forward the request on all interfaces but the arriving one (flooding). If the destination frame is the same as the arriving one, then it'll drop the frame.



Since the switches are self-learning, with the same reasoning, we can connect together more switches:



In general, switches may propagate **broadcast storms**, that are broadcast messages that require to the receiving node the broadcasting of the message to all the other nodes, increasing exponentially the

frame traffic and hogging the network (this is caused either by switch cycles of by cyber-attacks). This is a problem because switches, unlike routers, don't support flow control.

The differences and similarities between switches and routers are:

Switches	Routers
Store-and-forward	Store-and-forward
Act on link layer	Act on network layer
The forwarding table is updated by flooding (and it can be larger than the router' one)	The forwarding table is updated with routing algorithms
In the table there are MAC addresses	In the table there are IP addresses
Plug-and-play	They need IP address configuration
The topology must be the spanning trees one, so no cycles	They can be used in richer topologies and we may have cycles in case of wrong configurations but they're prevented with the TTL field in the IPv4 and with the hop limit field in the IP6.
Used in small networks	Used in larger networks

	Hubs	Routers	Switches
Traffic isolation	No	Yes	Yes
Plug and play	Yes	No	Yes
Optimal routing	No	Yes	No