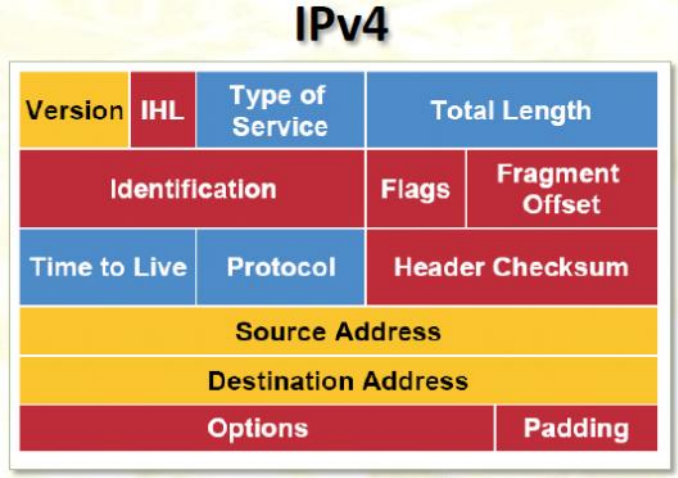
IPv6

Introduction

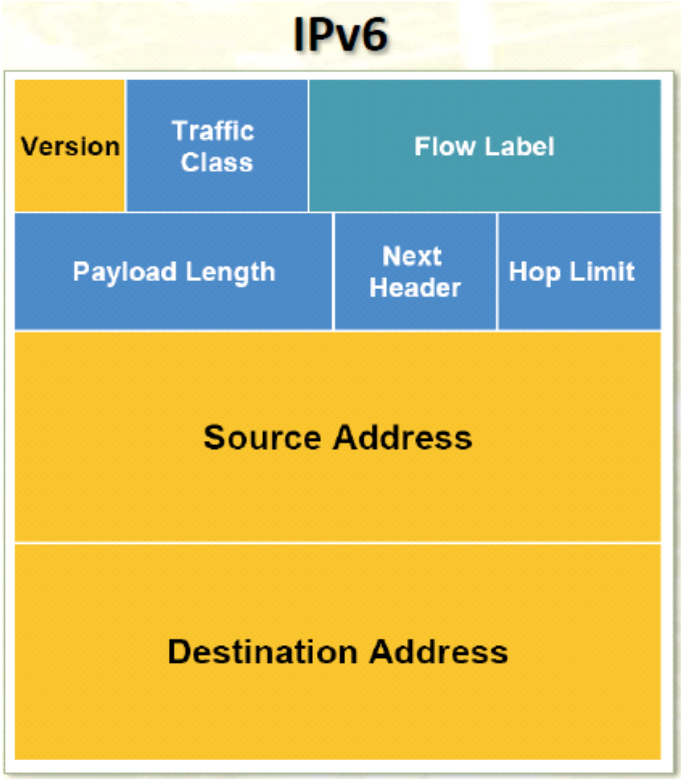
Motivations for IPv6:

- IPv4 address space limitations
- Some applications are difficult to operate through NAT
- Need for autoconfiguration
    - Purely stateless
    - Necessary for future IoT
    - e.g. DHCP is not ok
- Need for a simplified and faster to process header format
    - Fixed header size
- Need for extensibility

IPv4 vs IPv6 headers



Some notes on the IPv4 header:
* *IHL*: Internet Header Length
* *Type of Service*: for QoS
* *Identification*, *Flags*, *Fragment Offset*: for fragmentation
* *Options*: optional, variable length

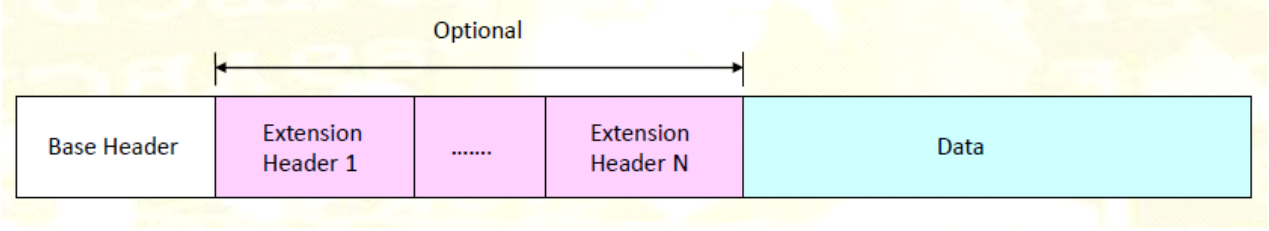


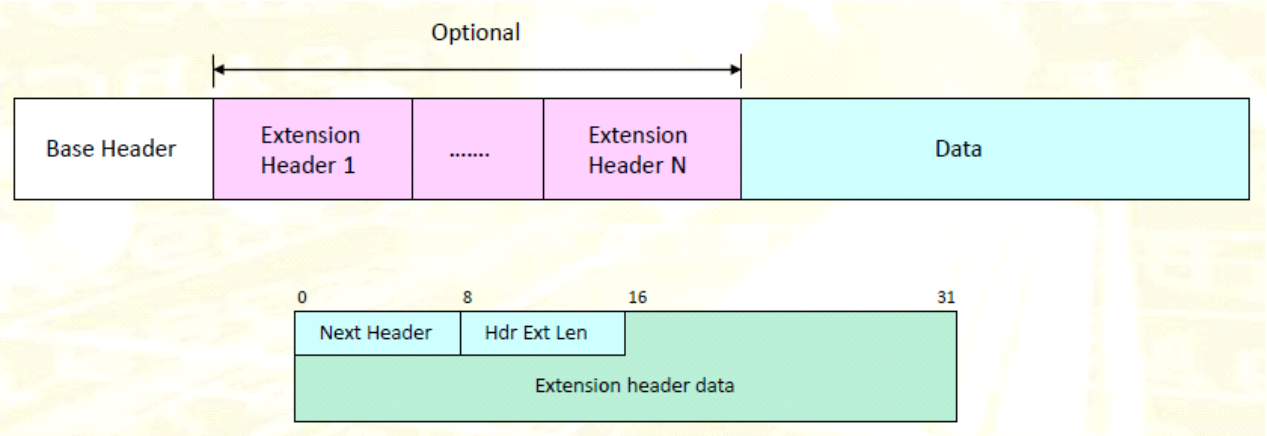
IPv6 header, news and differences w.r.t. IPV4 header:

- *IHL* no more needed: **fixed-size header**
- *Traffic Class*: replaces Type of Service
- *Flow Label*: randomly generated number, equal for all the packets belonging to the same flow of data, used to identify packets of the same flow and provide them a similar treatment
- Fields for fragmentation no more needed: there is **no fragmentation in IPv6**
  - There is a MTU allowed
  - Larger packets are discarded
  - In order to send packets with size larger than the MTU, either fragment them at higher layers or use special IPv6 extensions (discussed later)
- *Hop Limit*: replaces TTL
- Header checksum no more needed: now, links are reliable enough
  - Improved processing speed
  - Mandatory checksum check in UDP
- Fixed header size + no header checksum + no fragmentation --> **fast processing**, which can also be implemented in hardware
- Options replaced by extensions
- *Next Header*: specifies the higher layer protocol or the extension header type of the message contained in the payload (discussed hereafter)

IPv6 Extension Headers

IPv6 provides an extensibility mechanism based on **Extension Headers** (EH).





IPv6 header is made up of the base header (seen before), zero or more EHs and eventually the higher layer packet. Headers are organized in a **chain**:

- The field *Next Header* in the base header and in each EH specifies the type of the next EH
  - Eventually (in the last EH), the field will specify the type of the higher layer packet
- The field *Hdr Ext Len* (Header Extension Length) specifies the size of the current extension header (so that it is possible to know where the next extension header starts)

Here are some examples of EHs:

Next Header Value	**Meaning**
0	Hop-by-Hop Options Header
4	IPv4 (*tunneling*)
6	TCP (higher layer protocol)
17	UDP (higher later protocol)
41	IPv6 (*tunneling*)
43	Routing Header
44	Fragment Header
…	…

Some notes on EHs:

- Processed by the destination node, with the unique exception of Hop-by-Hop Options Header, which is processed by every node in the path
- A specific header order is *recommended*
- EHs are **open to further extensions**

In the following, we will analyze some EHs in more detail.

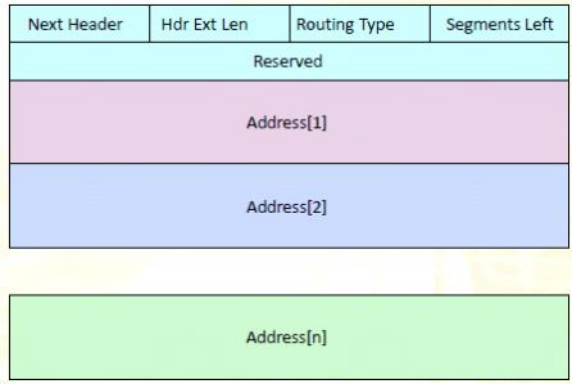
Hop-by-Hop Options Header

Contains **optional information** that has to be processed by **every node in the path**. Options (one or more) are included as *Extension Header Data* and expressed in TLV (*Tag-Length-Value*) form:

- *Action* (2 bits) specifies what to do if the option is not recognized:
  - 00: skip and continue processing
  - 01: discard the packet
  - 10: discard the packet and send an ICMP Parameter Problem message to the source
  - 11: discard the packet and send an ICMP Parameter Problem message to the source only if the destination is not a multicast address
- *C* (1 bit):
  - 1: the option information can be modified along the path
  - 0: the option information can't change
- *Type*: specifies which type of option is contained in this message - For instance:
  - 194 (*Jumbo Payload*): used to send **very large packets** (> 64K), which can't be fragmented and couldn't be sent in the IPv6 payload otherwise
    - The IPv6 base header *Payload Length* field is set to 0 and no IPv6 payload is sent
    - The packet is sent, instead, in the *Option Data* field of this *Jumbo Payload* option
  - 5 (*Router Alert*): the *Option Data* field contains information for this node, such as a request to **reserve resources** for a specific flow of packets, in order to provide QoS guarantees

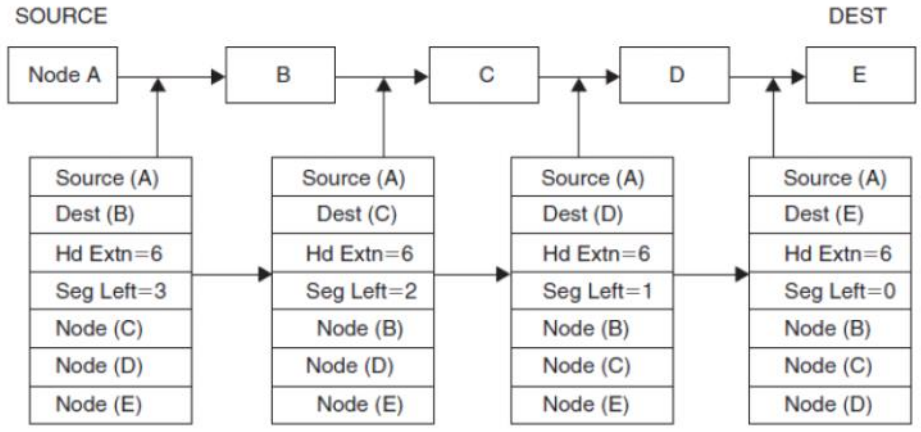
Routing Header

Gives a list of intermediate nodes to be visited on the path to the destination, allowing to do what is called **source routing**.



- Routing type:
  - 0: default
  - 2: Mobile IPv6
  - 3: RPL
- Segments left: number of nodes left to be visited

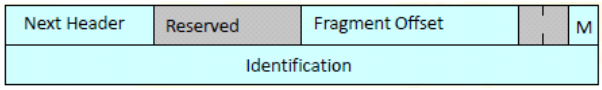
Example:



Problems:

- A malicious node could insert its address in a Routing Header in order to intercept packets
- A malicious node could insert the address of another node in the routing header of every message, so that all the traffic converges towards it and the node collapses (DoS attack)

Fragment Header



Can be used **fragment packets** larger than the path MTU:

- IPv6 hosts use a *Path MTU Discovery Procedure* to discover the path MTU:
  - The source host sends a large packet towards a destination
  - If the MTU along the path is smaller than the packet dimension, the packet will be discarded by an intermediate hop
  - This intermediate hop will reply back with an ICMP error message in which the MTU value will be included
  - IPv6 supports a minimum MTU of 1280 bytes
- Fragmentation occurs at the source host only and reassembly occurs at the destination only

Other Extensions Headers

Next Header Value	**Meaning**	**Brief Description**
50	Encapsulating Security Payload Header	Used for security purposes (like IPSec in IPv4)
51	Authentication Header	Again for security purposes
59	No Next Header	Used when there is no next EH
60	Destination Options Header	Like Hop-by-Hop Options Header, but for the destination only

IPv6 Addressing

There are three address categories:

- A **unicast** address uniquely identifies an interface of a node
- A **multicast** address identifies a set of IPv6 interfaces; a packet sent to an IPv6 multicast address is

processed by all the members of the multicast group
- An **anycast** address is assigned to multiple interfaces; a packet sent to an anycast address is delivered to only one of these interfaces
    - Anycast addresses are useful, for instance, when there are multiple equivalent servers that can process a request, therefore the request can be sent to any of these servers; in this way, anycast addresses can be used for **load balancing**.

A broadcast address category is missing; in fact, in IPv4 there was the special broadcast address which identified all the interfaces in the internet, but this was not actually used, because border routers block this kind of packets from going outside the local subnet. In IPv6, the broadcast address has been replaced by a special link-local multicast address.
Broadcast addressing was used, for instance, in DHCP, because a host joining the network doesn't know the actual address of the DHCP server. In IPv6, this problem is solved by using default multicast addresses for special services like DHCP. The advantage of this approach is that a non-DHCP node receiving an IPv6 packet containing a DHCP request will check that its own address doesn't match the packet destination address and therefore discard the packet. With the IPv4 approach, instead, all the nodes have to process the packet at layer 3 and they will be able to recognize the DHCP request and discard the packet only at layer 4. In conclusion, the IPv6 approach allows to save resources.

IPv6 **links** are abstractions: a link is a set of interfaces that can communicate directly with each other.
Assumptions about links:
- Stable over time
- Single link-layer broadcast domain
- Transitive (if A --> B and B --> C, then A --> C)

Implications:
- Network prefixes:
    - Each IPv6 subnet is associated to a unique link; from the assumption of transitivity, all the host on that link, and therefore in that subnet, can directly communicate
        - This is justified by the fact that at layer 2 we might have a network composed of multiple physical links and switches, but at layer 3 it is like all the hosts can directly communicate; only a router can separate different subnets and different links
    - Each subnet, and the corresponding link, is identified by a network prefix
    - Follows that network prefixes can be used to determine in an interface is attached to a given link
- Procedures like neighbor discovery, address resolution and duplicate address detection become easy to implement (discussed later)

Address scope:
- **Global**: the address uniquely identifies a host in the whole internet
- **Link-local**: the address is unique only inside a link (and should never be routed outside)
- **Unique Local**: The address can be used only inside the local link, but it is unique in the whole internet
    - The purpose of this kind of address is for business reasons:
        - If an enterprise wants to join two subnetworks and there are interfaces with the same address, then the network manager has to reconfigure all the addresses
        - Using Unique Local addresses it is guaranteed that it is possible to join the two networks without having conflicting addresses and without the need for reconfiguration

Address notation:
- **Format**: **x:x:x:x:x:x:x:x**
    - Where x is a block of four hexadecimal digits
    - **128 bits** total
- Abbreviation rules:

    - Leading zeroes can be skipped
    - A block of all zeroes can be represented with a single zero
    - A set of consecutive zeroes can be replaced by "::" (this rule can be applied only once)
- Prefix notation
    - Similar to IPv4: [IPv6 address]/[prefix length]
    - Identifies a set of addresses (e.g. belonging to the same subnet)
- Special addresses:
    - **Unspecified address**: **0:0:0:0:0:0:0:0** (::)
        - Used for instance in DHCP, in place of the source address when the source host doesn't know its address yet
    - **Loopback address**: **0:0:0:0:0:0:0:1** (::1)

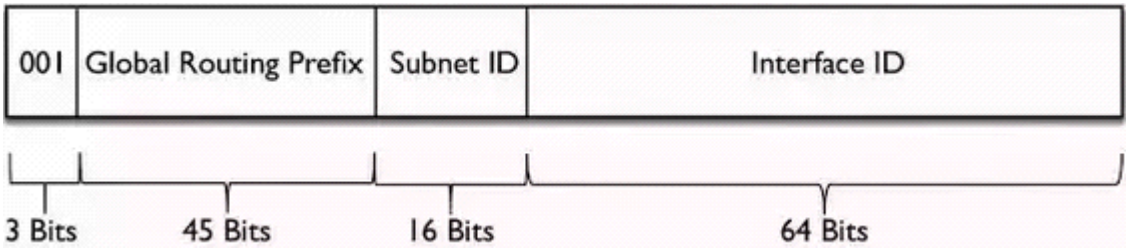
Prefix Allocation

Allocation	**Prefix binary**	**Prefix hexadecimal**
Global unicast	001	2000::/3
Link-local unicast	1111 1110 10	FE80::/10
Unique-local	1111 110	FC00::/7
Private administration	1111 1101	FD00::/8
Multicast	1111 1111	FF00::/8

Note: private administration is a subset of unique local addresses

- In pure unique local addresses, addresses are assigned by an authority and therefore unicity is guaranteed
- In private administration addresses, an enterprise autonomously assigns addresses and therefore unicity is not guaranteed
    - However, this scheme works in practice, because even if two addresses are assigned autonomously the probability of conflict is very small
- The 8th bit of the address distinguishes pure unique local from private administration addresses

Global unicast addresses:



- The first 48 bits (the first three of which are fixed to 001), called **Global Routing Prefix**, identify a **site** (corresponding, for instance, to an organization) and they are assigned by an authority
- The successive 16 bits identify a **subnet** within the site
- The last 64 bits are called **Interface ID** and they identify a specific IPv6 interface in a subnet

Link-local addresses:

Unique Local addresses:

- Structure similar to global unicast addresses
- The 8th bit (L) specifies if the address is pure unique local (0) or private administration (1)
- Global ID:
    - In case of pure unique local, it is assigned by an authority
    - Otherwise it is autonomously generated at random

Interface ID:

- Unique identifier within a link
- Default approach: Interface ID obtained from the layer 2 address
- Privacy issues with the former approach: if a node moves from a network to another, the prefix of its address changes but the Interface ID remains the same; therefore, it is possible to track the node's movements
- Other approaches for Interface ID generation are possible

Possible approaches for Interface ID generation:

- **Stable**: don't change over time (at least, as long as the host remains connected to the subnet)
    - Static:
        - Manually configured
        - **Stateless** configuration - SLAAC (State-Less Address Auto-Configuration): Interface ID generation from layer 2 address falls here
    - Dynamic: DHCPv6
- **Temporary transient**: randomly generated by the node and change at regular intervals
    - This approach is not preferred because some networking features are based on the assumption that addresses are constant over time (for instance, filtering packets with a given destination address)
    - When an address is autonomously generated, the node must check that no one else is using that address; this can be done with the IPv6 Duplicate Address Detection procedure (discussed later)
- **Stable privacy address**: stable within a subnet but changes when the host moves to another subnet

Note: Duplicate Address Detection is performed also when the Interface ID is generated from the L2 address, because, even though L2 addresses are unique, IPv6 prefers not to rely on this assumption.

Multicast addresses: the following table shows some well-known link-local multicast addresses

Address	Description
FF02:0:0:0:0:0:0:1	All-nodes address
FF02:0:0:0:0:0:0:2	All-routers address
FF02:0:0:0:0:0:0:5	OSPFIGP
FF02:0:0:0:0:0:0:9	RIP routers
FF02:0:0:0:0:0:0:A	EIGRP routers
FF02:0:0:0:0:0:0:B	Mobile agents
FF02:0:0:0:0:0:1:2	All DHCP agents
FF02:0:0:0:0:0:1:4	DTCP Announcement
FF02:0:0:0:0:1:FFXX:XXXX	Solicited-node address
…	…

Anycast addresses: there is no special prefix for anycast addresses, this is because an anycast address is just a unicast address assigned to different interfaces.

ICMPv6

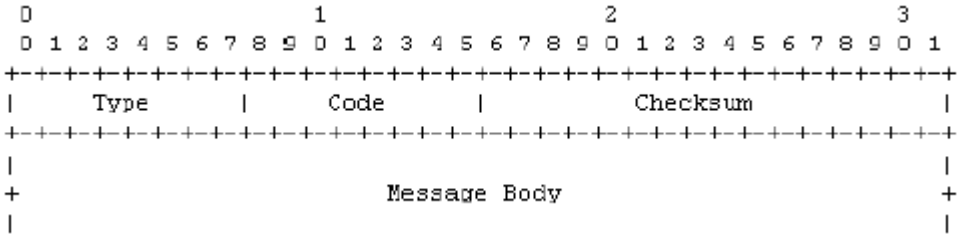
Higher layer protocol w.r.t. IPv6 for error management, configuration and diagnostic functions. ICMPv6 messages are wrapped into IPv6 packets with *Next Header* field value set to 58.
ICMPv6 also provides support for the IPv6 Neighbor Discovery protocol, which offers procedures for

- Finding routers
- Link-layer address resolution
- Detecting possible duplicate addresses

Message classes:

- Error messages
  - Destination Unreachable
  - Packet Too Big (used for the Path MTU Discovery procedure)
  - Time Exceeded (used when the hop count limit of a packet expires)
  - Parameter Problem (used when there is some error in the format of the packet)
- Informational messages
  - For instance, Neighbor Discovery
  - Echo Request, Echo Reply

Message format:



- *Type* : specifies how to interpret the message
  - Type 1: Destination Unreachable
  - Type 2: Packet Too Big
  - …

- *Code*: provides additional information about the message
  - For instance: Type = 1, Code = 0 --> Destination Unreachable, No route to destination
    - When the destination is unreachable, the Code provides information on why the destination was unreachable
- *Checksum*
- *Message Body*
  - In case of error messages, contains the original packet that caused the error
  - In case of informational messages, contains further information

Informational messages:

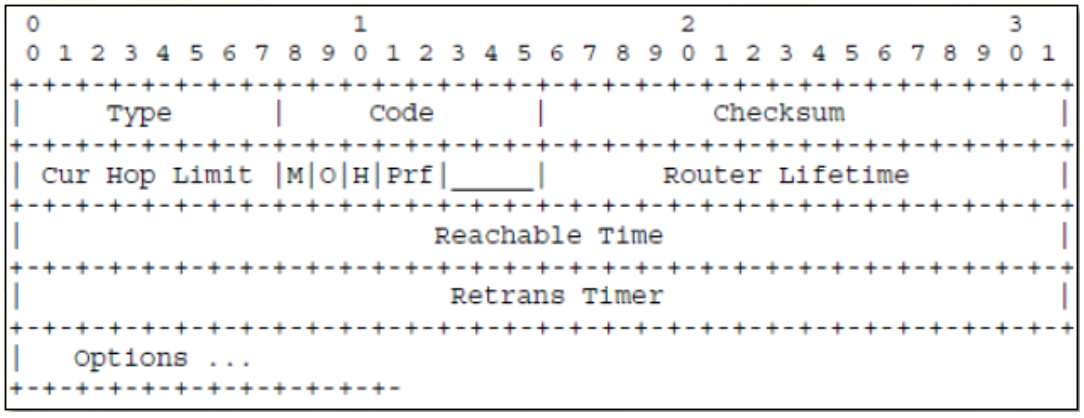
Message number	Message type	Description
128	Echo Request	RFC 4443. Used for the ping command.
129	Echo Reply	
...		
133	**Router Solicitation**	RFC 2461. Used for neighbor discovery and autoconfiguration.
134	**Router Advertisement**	
135	**Neighbor Solicitation**	
136	**Neighbor Advertisement**	
137	**Redirect Message**	
...		

In particular, messages 133-137 constitute the set of procedures supporting IPv6 Neighbor Discovery. The **Neighbor Discovery** (ND) protocol consists of a set of operations, among which:

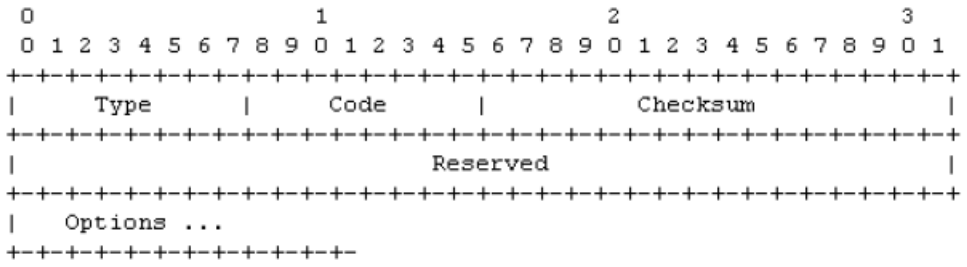
- Router Discovery
- Address Resolution
- Duplicate Address Detection

Router Discovery (RD):

- Periodically, routers send out *Router Advertisement* messages to the multicast *all-nodes* address (FF02::1), with the following format
  - *Router Advertisement* messages contain the necessary information for auto-configuration of a host joining the network
- *Router Advertisement* message format:



    - *Type* = 134, *Code* = 0
    - *Cur Hop Limit*: specifies a recommended value to be used by the hosts to set the *Hop Count Limit* field of their packets
        - This value depends on the network size: in general, it should be larger than the *diameter* of the network; nonetheless, this value is not known a priori by a host joining the network and that's why it is useful to report it in *Router Advertisement* messages
    - *Router Lifetime*: specifies how long a router remains in charge (unless a new *Router Advertisement* is received)
    - *M* flag: if set to 1, stateful address configuration is enabled (i.e. there is a DHCP server), otherwise address configuration is stateless
    - *Prf* (preference) flag (2 bits): in case there are multiple routers in the subnet, this field provides priority information, i.e. it specifies which router has higher authority
        - There might be multiple routers providing contradictory information, because one, for instance, was just re-configured, while the other's configuration is not up to date; in this case, we assign higher *Prf* to the first router, so that hosts will configure using its *Router Advertisements*
    - *Options*: optional information, among which
        - Source link-layer address: layer 2 address of the router
        - MTU: maximum transmission unit admitted on this link
        - Prefix Information Option (PIO): provides information on the prefix to be used on this link (there might be multiple PIOs in a *Router Advertisement*)
- *Router Advertisements* can be explicitly requested by a node by sending *Router Solicitation* messages to the multicast *all-routers* address (FF02::2)



    - *Type* = 133, *Code* = 0

Stateless Address Configuration:

- Interface ID autoconfigured as we have seen before
- The prefix is learned from *Router Advertisements*
- The address goes through different stages:
    - **Tentative** address: the host is verifying that no other host is using that address
        - Packets addressed to a tentative address are discarded, except for Neighbor Discovery packets
        - Once the host has verified the uniqueness of the address, with a Duplicate Address Detection procedure, the address becomes preferred
    - **Preferred** address: can be used by upper layer protocols without restrictions
        - After a given timeout, the address becomes deprecated
    - **Deprecated** address: use is discouraged, but not forbidden
        - After a given timeout, the address becomes invalid and cannot be used anymore
- Overall procedure:
    - A node joins the network
    - A link-local address is generated appending the Interface ID to the link-local prefix (FE80::/10)

        - The link-local address is tentative
        - Duplicate Address Detection is performed
        - If no duplicates are found, the address becomes preferred until expiration
    - Optionally, a *Router Solicitation* message is sent to the *all-routers* address
    - *Router Advertisements* are received, containing one (or more) prefix piece of information
        - From a given prefix, an address is generated by appending the Interface ID
        - The address is tentative, therefore Duplicate Address Detection should be performed
        - If no duplicates are found, the address becomes preferred until expiration

Duplicate Address Detection (DAD) is performed as follows:

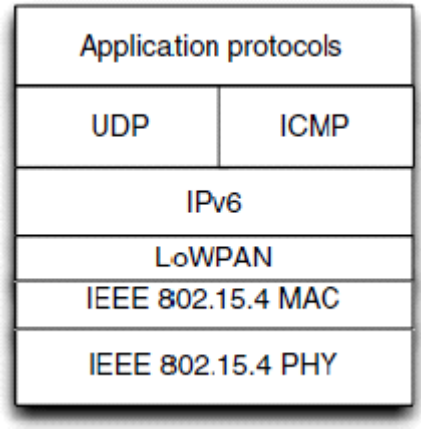
- A *Neighbor Solicitation* message is sent out, containing the query address
    - When the host hasn't a link-local address yet, the source address is set to :: (unspecified)
    - The destination address is the *solicited-node* multicast address (FF02::1:FFXX:XXXX), where the X's are the last 24 bits of the Interface ID that this node is going to use
        - On layer 2, the message is sent to the L2 multicast address 33:33:FFXX:XXXX
        - Whenever a host wants to use a given address, it must first "subscribe" to the corresponding *solicited-node* multicast group (i.e. enable reception of messages addressed to the layer 3 *solicited-node* multicast address and also to the corresponding multicast address on layer 2)
        - In this way, the message is processed only by the interested hosts, i.e. those hosts with the 24 less significant bits of their Interface ID equal to XX:XXXX, and this technique allows to avoid massive use of pure broadcast
    - Solicited nodes whose address matches the query address will reply with a *Neighbor Advertisement* message
    - The node performing DAD proceeds as follows:
        - If a *Neighbor Advertisement* message is received, the address can't be used, because someone else is already using it
        - If no *Neighbor Advertisement* is received, the address can be safely used

Address Resolution is the process of obtaining the L2 address of a node of which we know the L3 address; it can be performed as follows:

- A *Neighbor Solicitation* is sent out to the *solicited-node* address, containing the query L3 address
- The consequent *Neighbor Advertisement* reply will contain the L2 address of the solicited node whose address matches the query address

6LoWPAN

Up to now, IoT research has developed layer 2 protocols specifically designed for energy efficiency and computationally constrained nodes. Currently, the standard is 802.15.4, but new solutions are up to come out. As a layer 3 technology, research aims to place IPv6 in the IoT stack, but there are some problems; in fact, IPv6 is not compatible with 802.15.4. In order to face this challenge, an **adaptation layer** is placed in the networking stack, between layer 2 and layer 3, with the purpose of providing compatibility between the two solutions. The standard protocol for the adaptation layer is **6LoWPAN** (IPv6 for Low-power Wireless Personal Area Network).



A first problem to solve to achieve compatibility between IPv6 and 802.15.4 is that IPv6 makes the assumption that **a subnet is associated with a single link**, and therefore all the hosts on the same subnet can directly communicate with each other.
This assumption is not always true in multi-hop wireless networks. For instance, a host A could communicate with B, and B with C, but A could not communicate with C because they are too far away. In this case, the underlying link model changes: there isn't a single link-layer broadcast domain, but multiple **overlapping broadcast domains** --> *undetermined* link model.
In this kind of scenario, in order to achieve compatibility with IPv6, an L2 protocol for multi-hop wireless networks should support **multi-hop link-layer forwarding** (i.e. it should implement some routing algorithm for mesh networks at layer 2), which 802.15.4 does not. There are two solutions to cope with this problem:

- **Mesh-Under**: multi-hop forwarding is implemented either at L2 or at the adaptation layer
  - A possible way to do this is the following: let's assume we have a frame directed from A to C, but C is not directly reachable

Src: A	Dst: C	Data

  - Let's assume that C is reachable through B; then, A can encapsulate the frame in another header containing B as destination address

Src: A	Dst: B	Src: A	Dst: C	Data

  - When B receives the frame, it removes the outer header, check the inner header and determines that the packet is directed to C, encapsulate the packet with another header and sends it to C

Src: B	Dst: C	Src: A	Dst: C	Data

    - Finally, C receives the packet, removes the outer header, check that the inner packet is from A to C itself and therefore passes the Data to the higher layer
    - Of course, the choice of the next hop for a given destination is made on the base of a routing algorithm specified a priori
    - Problem: lack of interoperability between L2 and L3, route repairs are slow
        - Example: a node wants to send a packet to a border router
        - At layer 2, a link is broken
        - The node realizes that the link is broken after several failed retransmissions and then starts a route repair (i.e. it looks for another path towards the destination)
        - This process takes a lot of time
        - In the meantime, at layer 3, the node decides that the destination is unreachable and chooses to send the packet to a second border router
- **Route-Over**: in this case, we do not expect support for multi-hop forwarding from layer 2, therefore we explicitly face the problem at layer 3, considering an undetermined underlying link model
    - Problem: higher overhead w.r.t. mesh under, because packets have to be processed (possibly incurring into fragmentation/reassembly) up to layer 3 at each hop

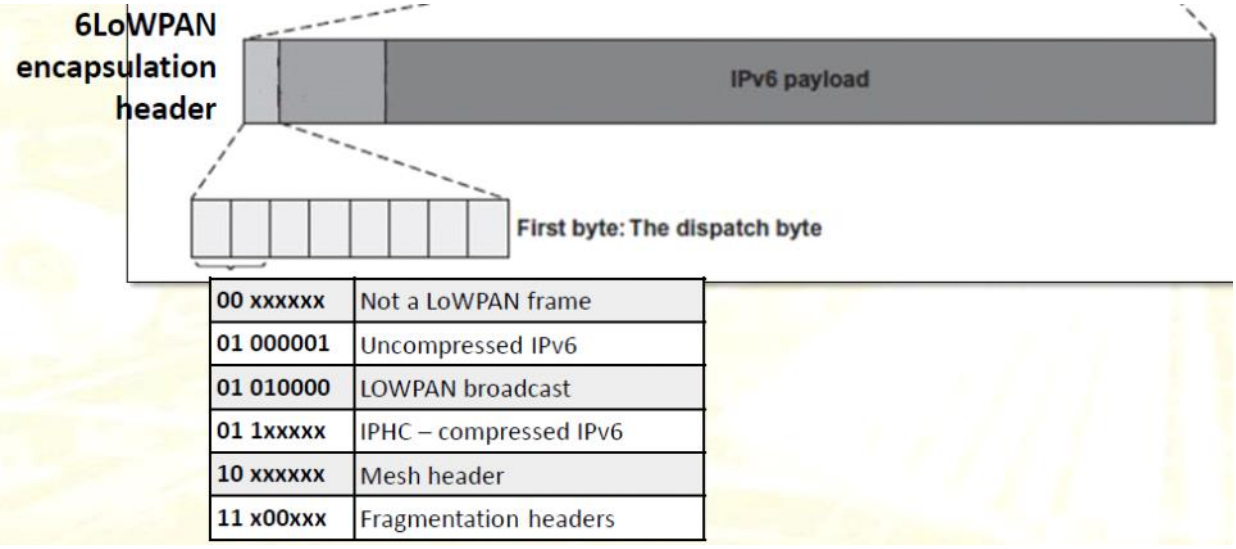
Another compatibility problem between IPv6 and 802.15.4 is the fact that IPv6 requires a minimum MTU of 1280 bytes, but 802.15.4 supports only an MTU of 127 bytes. The solution to this problem, provided by 6LoWPAN, is called **packet adaptation** and it is achieved by implementing fragmentation at the adaptation layer.

Finally, we want to implement some optimizations:

- **Addressing**:
    - Since network prefixes are obtained from *Router Advertisements* and are the same for all the network, follows that
        - For in-network communication, it is possible identify a node with the 64-bits Interface IDs only
        - It is also possible to identify a node by using 16-bits short identifiers (valid only in the local network), assigned by 802.15.4 when the node joins the network
        - We would like to replace IPv6 full addresses with short addresses, when possible, in order to make the addressing scheme more lightweight
    - Interface IDs generated from layer 2 identifiers
        - In this way, address resolution is made easier
- **Header compression**: we want to reduce the IPv6 header size (short addresses are also helpful in this case)

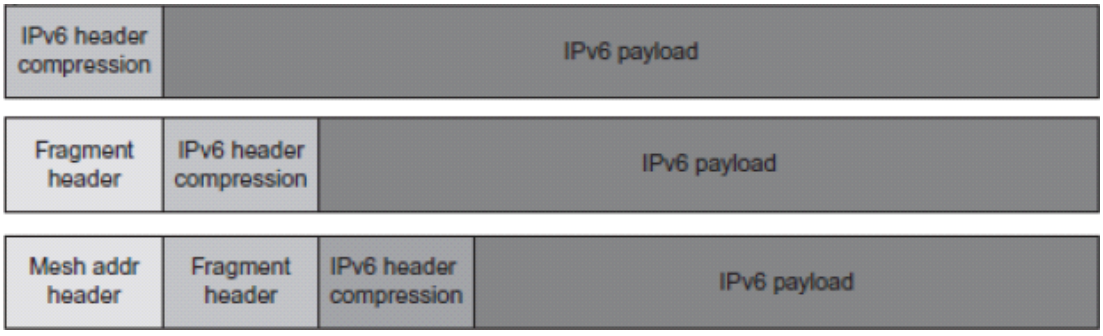
6LoWPAN header:

- Multiple headers organized as a stack, like in IPv6, but with a more rigid structure

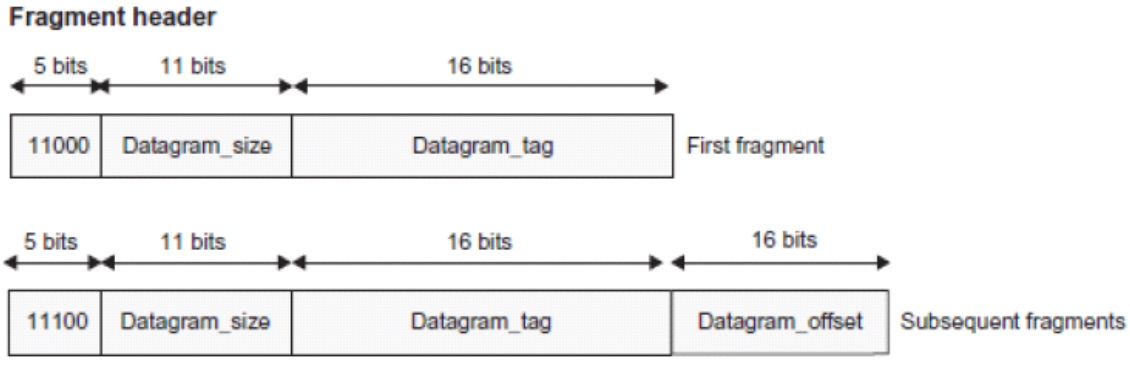


00 xxxxxx	Not a LoWPAN frame
01 000001	Uncompressed IPv6
01 010000	LOWPAN broadcast
01 1xxxxx	IPHC – compressed IPv6
10 xxxxxx	Mesh header
11 x00xxx	Fragmentation headers

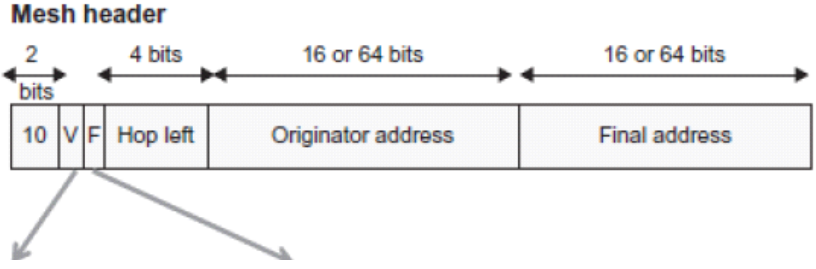
For instance, we may have

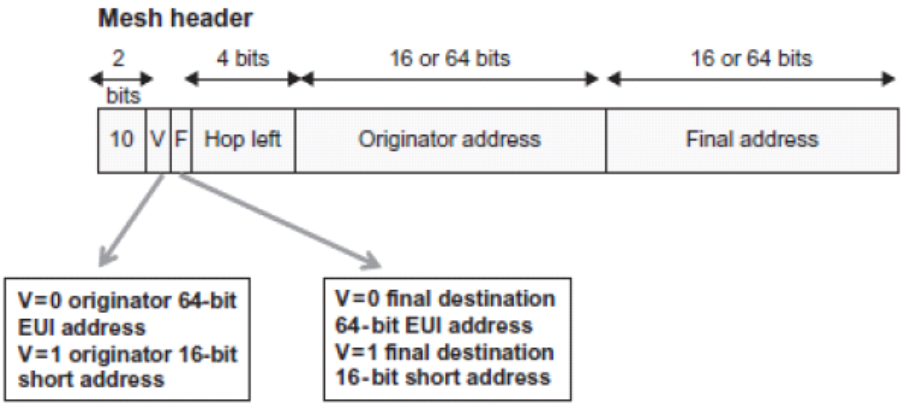


- IPv6 compression header implements header compression (discussed in more detail later)
- The fragment header implements packet adaptation through fragmentation



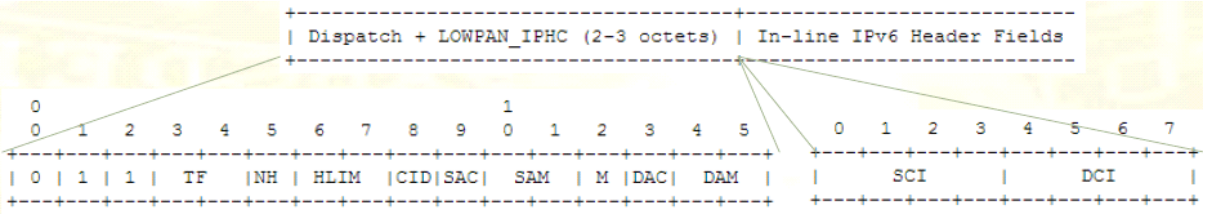
- The mesh header is used to implement a mesh under solution, although, in this case, the routing algorithm to be used for multi-hop link-layer forwarding is not specified by the standard





Header compression:

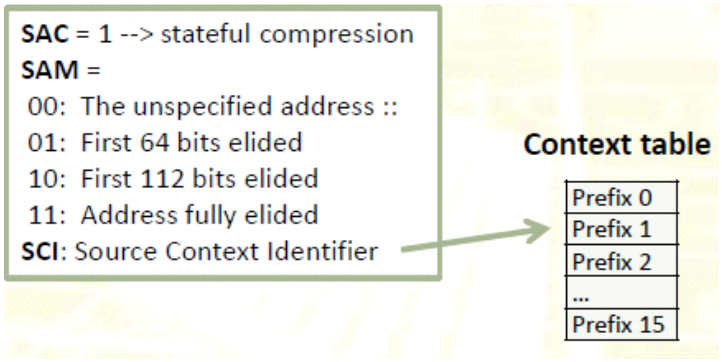
- **Stateless header compression** is a very simple scheme in which there is no shared information (*context*) between the source and the destination
    - Compression simply comes from the observation that several fields in the IPv6 header typically take the same values or they contain information that can be retrieved from layer 2 header
        - For instance, the *Version* field is always 6
        - *Traffic Class* is usually 0
        - The *Flow Label* is typically not considered
        - The *Next Header* field is typically UDP or ICMP
        - The *Payload Length* can be inferred from the layer 2 frame size
        - *Source* and *Destination Addresses* can be inferred from layer 2 frame
    - Therefore, stateless compression consists in removing redundant information from the header
    - However, this scheme is not typically used
- **Context-based header compression**: some shared context between source and destination is used to achieve better compression
    - Standard: Internet Protocol Header Compression (IPHC)
    - 6LoWPAN replaces IPv6 header with the following compressed header



    - TF specifies if *Traffic Class* and *Flow Label* have default values; if they haven't, their values are passed as *In-line IPv6 Header Fields*
    - NH specifies the same for the *Next Header* field
    - HLIM specifies possible values for the *Hop Limit*
    - SAC (Source Address Compression) specifies whether the source address is statefully or statelessly compressed
    - SAM (Source Address Mode) specifies how many bits of the compressed source address are elided, while the remaining are carried in-line

SAC = 0 --> stateless compression
SAM =
00: The full address is carried in-line
01: First 64-bits elided
(link-local prefix + XXXX:XXXX:XXXX:XXXX)
10: First 112 bits elided
(link-local prefix + 0000:00ff: fe00:XXXX)
11: Fully elided
(link-local prefix + IID derived from L2)

In case of stateful compression, the prefix can be found in a context table (of 16 entries), shared between the sender and the receiver, at the offset specified by the SCI (Source Context Identifier) carried in-line



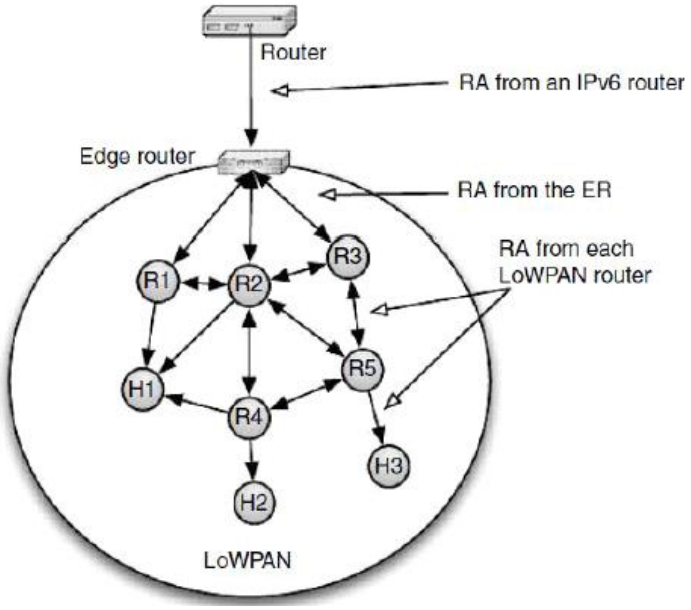
- DAC, DAM, DCI: same for the destination address

The above-mentioned undetermined link model has also some implications concerning the **problem of address configuration and the Neighbor Discovery procedure**:

- We want unique IPv6 addressing across the whole network
  - Nodes in the same subnet share the same network prefix
  - However, nodes in the same subnet might not be in direct communication range
- DHCP doesn't work --> stateless address auto-configuration is needed
- On-link subnet prefix configuration is not possible, because the link-local scope of a node is limited to those nodes in direct radio communication range and not to the whole subnet
  - Multi-hop dissemination mechanisms for Router Advertisements must be considered
  - More in general, all the Neighbor Discovery procedure must be adapted
- There is also another implicit problem: Neighbor Discovery assumes that nodes are always active, while at the underlying levels they might switch off the radio to save power
- In conclusion, we must re-design the whole address configuration and Neighbor Discovery procedure

We consider a network made up of three types of devices:

- 6LoWPAN Hosts: regular nodes that simply gather information and forward it to a router
- 6LoWPAN Routers (6LR): forward information to other routers up to a border router
- 6LoWPAN Border Routers (6LBR):
  - Are connected to the traditional internet
  - Gather information from the network
  - Provide initial configuration to the nodes of the network



Neighbor Discovery and address configuration are modified as follows:

- *Router Advertisements* and *Router Solicitations* exchanges occur between hosts and routers within transmission range
    - No periodic transmission of RAs
    - The host transmits the RS to the *all-routers* multicast address
    - Each router replies with a unicast RA, so that other hosts avoid to process RAs they are not interested to
- At startup, 6LRs act like hosts, gathering information about the initial configuration
    - Initially, only 6LBRs have the configuration information
    - Information is flooded from 6LBRs to all the other nodes in the network
    - Configuration information consists of network prefixes and context information (the context tables used in context-based header compression)
- Authoritative option:
    - Different routers could reply to the same RS , providing different configuration information (for instance, because the configuration of a router is up to date while the other is not)
    - The authoritative option is a special option added to RAs, used by hosts when different configuration information is received by different routers, in order to decide which one to use
    - In particular, this option contains a version field which is used to decide which information is the most up to date, corresponding to the router with higher authority
- Address Registration option:
    - Carried in Neighbor Solicitation messages directed to 6LRs
    - This option is used by hosts to register at a specific router, therefore this is a way for a host to tell a router: "I'm in your neighborhood"
    - Routers maintain a cache of neighboring hosts
    - Proactive solution: hosts help routers maintain a cache of neighbors by explicitly informing about their presence, instead of having the routers perform a reachability procedure to check which nodes are in their neighbors
        - Especially useful in case of mobility
    - This registration mechanism provides support, among the other things, for reachability tests even with nodes going to sleep
        - It is not the router which sends the NS, this is useful because it allows hosts to go to sleep without caring about the fact that a router might perform a reachability check and not find it
    - This scheme also provides partial support for DAD: if two hosts register at a router with the

same address, the router can detect the duplicate (however, it is still possible to have hosts with the same address under different routers)
    - The reply to a NS containing an Address Registration option is a Neighbor Advertisement containing the outcome of the registration
        - A "registration successful" reply is received when the registration has been performed correctly
        - Another possible reply is the "update successful", if the node was already registered but it just wanted to refresh the registration
        - If this is the first time a node registers at a router, but it receives an "update successful", it means that another node is already using its address, and therefore a duplicate address is detected
- DAD-related assumptions
    - The typical assumption related to duplicate addresses leads to the *optimistic approach*: we assume that L2 addresses are globally unique and Interface IDs are generated from them; therefore, no duplicate addresses occur and DAD is not performed Layer 2
    - Otherwise, DAD is checked by means of a multi-hop registration (6LoWPAN DAD), with a centralized state kept at the 6LBRs
        - New ICMPv6 messages to support this procedure*: Duplicate Address Request* (DAR) and *Duplicate Address Confirmation* (DAC)
        - Address Registration from a host is relayed by routers to the border router in DAR messages
        - A notification of the outcome is sent back through DAC messages
    - If Interface IDs are obtained through DHCP, DAD is optional
- Address resolution
    - Only link-local addresses can be considered on-link, while all the other prefixes are considered off-link
        - In particular, addresses with the same network prefix, and therefore in the same network, cannot be considered on the same link, due to the undetermined link model
    - Routers can reach off-link addresses through neighboring routers
    - Routers can reach on-link addresses directly, but the L2 address of the destination must be resolved
        - Node registration allows a router to maintain the state of the neighborhood, therefore address resolution could be performed from the neighbor cache rather than with a multicast-based approach

RPL

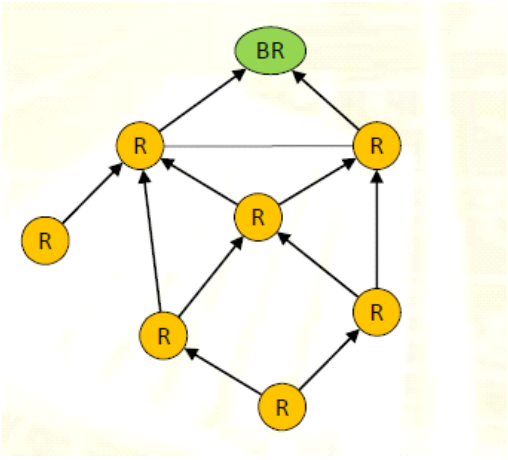
Traditional routing protocols like RIP or OSPF are not suitable in IoT, because there are several **new factors to consider**:

- Undetermined link model:
    - Traditional routing protocols are designed for the traditional IP link model (one link-one subnet hypothesis)
    - If a route-over solution is used, this hypothesis does not hold anymore and the routing protocol must adapt consequently
- New cost metrics, not only shortest path:
    - Loss of the communication medium
    - Energy efficiency
    - Delay constraints
- Adaptability to dynamic changes in the network topology
- Need for multiple paths towards a destination, to improve communication robustness
- Computationally limited nodes

RPL (Routing Protocol for Low power and lossy networks) is a layer 3 routing protocol (route-over) which represents the current standard solution for the scenario presented so far.

RPL characteristics:

- Based on Distance Vector
- Routing topology: **Destination Oriented Direct Acyclic Graph (DODAG)**, i.e. a graph whose edges converge to a special node, called *sink*, which corresponds to the border router



    - Topologies with **multiple sinks** are possible; in this case:
        - Different sinks are assumed to be directly connected
        - Each sink builds its own DODAG
        - Nodes choose only one DODAG to join (this is necessary so that the next point holds)
        - The final structure is equivalent to a single DODAG, where the multiple physical sinks are merged into a unique virtual sink
    - This kind of topology is convenient for wireless sensor networks, because, in this kind of networks, the traffic typically converges towards the sink: **multi-point to point** communication
    - A directed acyclic graph is used, instead than a tree, because this topology allows to have multiple paths toward a destination, thus improving robustness
- Paths determined on the base of a **cost metrics**, which is defined depending on the application requirements, plus some **constraints** (e.g. if we care about energy management, we may want paths not passing through battery-powered nodes)
- **Multiple RPL instances** can coexist in a network
    - For instance, one for energy management, whose paths do not pass through battery-powered devices, and another for delay-critical communication, using the shortest paths also passing through energy-constrained nodes
    - This is like having different routing tables at the routers, one for each RPL instance, each associated to a "color"; packets are marked with "colors" corresponding to the tables to be used to process those packets

The **Objective Function (OF)** defines which metrics/constraints to use for building paths in a RPL instance. Some examples of OFs defined by the standard are:

- *OF1*: low latency paths for delay-critical applications
- *OF2*: paths with minimum number of hops, not traversing battery-powered devices

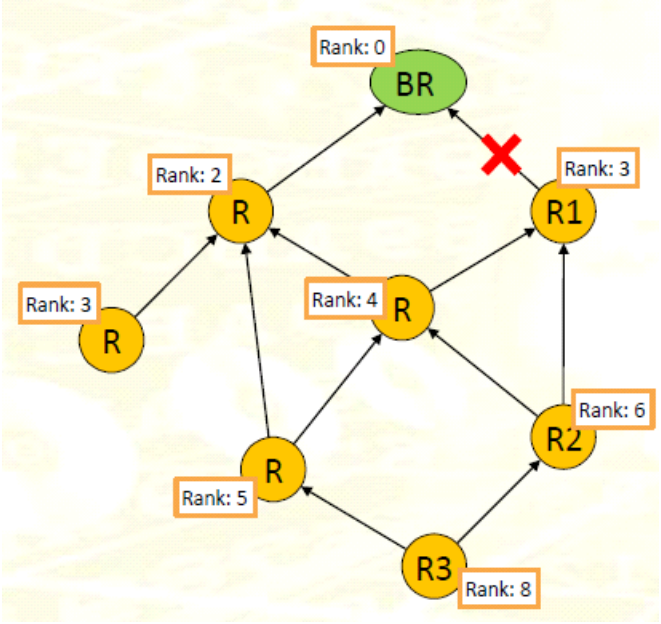
The **rank** is a scalar representation of the node position in the graph:

- It must monotonically decrease on each path towards the sin; in other words, the rank of a node is always larger than the rank of its parent nodes in the graph
- When a node joins the network, it receives the messages from its neighbors containing their ranks, then it chooses some parents among these neighbors and it has to set its rank so that it is larger than its parents' rank
- The rank provides a *partial ordering* among nodes in a graph, i.e. given two nodes A and B on the

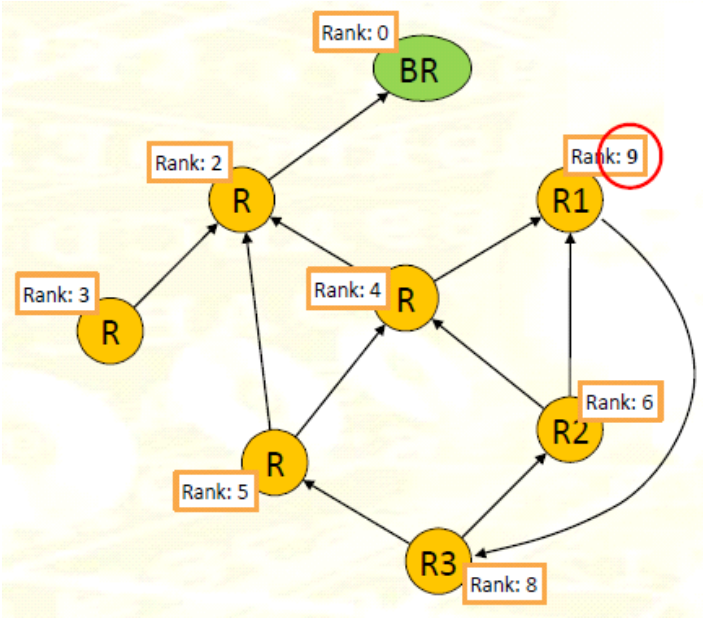
same path towards the sink, node A is closer to the sink than node B iff rank(A)<rank(B) (but if the two nodes are not on the same path towards the sink, no conclusion can be reached, that's why we talk about *partial* ordering)
- How the rank is computed is defined by the OFs
- Ranks can be used to avoid loops
- The rank is not intended to be a measure of the cost of the path towards the root:
    - A node may have more paths towards the root, hence a single measure like the rank is not suitable to represent the path cost
    - Moreover, it is better to keep the two metrics separate; in fact, the path cost is a metric which might vary frequently, while the rank is a measure that we would like to remain stable

DODAG formation:
- DODAG root starts advertising its presence by sending **DODAG Information Object (DIO)** messages to the link-local multicast address; these messages contain different pieces of information, among which:
    - The RPL instance identifier
    - The DODAG identified within the RPL instance
    - RPL instance OF
    - The rank of the sender
    - …
- RPL nodes listen to DIO messages and forward them to advertise their presence and propagate the information
    - By receiving DIO messages, a RPL node learns the set of nodes in the one-hop neighborhood
    - Within such set, based on the RPL instance OF, the node
        - Determines its candidate neighbor set
        - Selects one or more parents from the neighbor set
        - Selects a preferred parent from the parent set as the default route towards the DODAG root
    - Based on the rank received from the parents, the node determines its own rank
- Periodic broadcast of DIO messages, to maintain routing information up to date
- DIO messages propagation is regulated by the **Trickle** algorithm:
    - Gossiping algorithm, allows to propagate information with a controlled flooding
    - Each node maintains a time interval I, initially set to I_{min}
    - At the beginning of the interval, each node sets a timer to a random value in the range [I/2, I]
    - When the timer expires, a node broadcasts the message
    - Each node counts the number of *consistent* messages received in the current interval; when the timer expires, the node performs the broadcast only if this number is smaller than a *redundancy threshold* (fixed a priori)
        - This feature, called **broadcast suppression**, allows to limit the amount of flooding
    - When an interval terminates, a new one is started, whose duration is doubled w.r.t. the duration of the previous interval, up to a maximum value I_{max}
        - However, if an *inconsistent* message is received and if I != I_{min} , the current interval is immediately terminated and a new one is started, whose duration is reset to I_{min}
        - This behavior, called **adaptive periodicity**, allows to reduce the frequency of broadcasting when the network topology is static, and increasing it when the network topology changes (in which case, *inconsistent* messages are exchanged) and fast adaptation is needed
- The first half of the period [0, I/2] is a **listen-only period**: nodes do not perform broadcast
    - However, messages from other nodes can be received during the listen-only period, because the intervals at different nodes may go out of synch
    - This period is used to avoid the *short-listen* problem, which occurs when a node selects a

small random timer and therefore it transmits right after the beginning of its interval
    - In this case, it might happen that a node starts its interval and immediately transmits, then another node does the same and so on
    - This might happen for several nodes consecutively, thus producing a burst of transmissions which might result in unbounded bandwidth usage, especially in very dense networks
    - A listen-only period prevents this problem, because it imposes a separation on the transmissions in a burst, thus limiting the bandwidth usage
- **Route inconsistencies**:
    - Trickle uses the concepts of *consistent* and *inconsistent* messages: in our case, *inconsistent* messages correspond to changes in the network topology
    - The sink might detect route inconsistencies, in which case it can send new DIO messages to order the construction of a new DODAG
        - DIO messages include another piece of information, i.e. a DODAG version
        - New DIO messages include a fresher DODAG version number
        - When a node receives this DIO message with the fresher DODAG version number, it starts the procedure to join the new DODAG; moreover, it behaves as if an *inconsistent* message is received, resetting the Trickle interval, to allow a fast construction of the new DODAG
    - Also a simple node may detect route inconsistencies
        - For instance, a node might observe that the rank of a parent is changed
        - If the new rank is larger than the current rank of this node, an *inconsistency* is detected
        - In this case, the node adapts to the new topology and the Trickle interval is reset
        - Information about the node rank can be included in the messages generated by the node, by inserting it in the IPv6 header as a hop-by-hop option
- **Loop detection** can be performed by nodes as they observe inconsistencies in the ranks of their parents
    - Let's consider the following example



        - Link BR-R1 breaks (for instance, because R1 is moving downward)



- R1 attaches to R3
- Consequently, it updates its rank to 9
- However, R1 is still parent of R2, which is, in turn, parent of R3 --> a loop is formed
- In this condition, the loop can be detected when R2, for instance, sends a message to the root, using R1 as next hop
- R1 will receive a message in which the source rank is smaller than its own rank, but this is an inconsistency because messages traversing a path towards the root should always see decreasing ranks --> loop detected

- A **counting to infinity** problem arises when loops are formed:
  - In the previous example, R1 will start sending DIO messages with its updated rank, which will be received by R2
  - R2 doesn't know that there is a loop and it will just increase its own rank in order to maintain the rule that parent's rank must be smaller than children's ranks
  - Then, R2 will start sending DIO messages with its updated rank, which will be received by R3
  - R3, upon receiving the message from R2, will do the same: increment its rank and start sending updated DIO messages
  - R1, in turn, will receive updated DIO messages from R3 and increase its rank again
  - The procedure repeats over and over and nodes keep incrementing their rank indefinitely
- Loops can be broken and the counting to infinity problem can be avoided by defining a *Max Depth* value:
  - Ranks cannot be larger than this value
  - When a node sees that its parent has reached the maximum rank value, which might happen after a count to infinity, it just detaches from the parent, thus breaking a possible loop
  - The counting to infinity can be avoided by imposing that a node set its rank to the maximum value as soon as it observes that a path towards the root has failed (this is like *route poisoning*, when a router sets its routing cost towards a destination to infinity)
  - In the former example, R1 should set its rank to the maximum value when the link with the BR fails

Until now we have considered only communication from nodes to the root (multipoint-to-point). In RPL, in is also possible to enable **point to multi-point** and **point-to-point** communication, i.e. messages from

the root to the nodes or from a node to another in the DODAG. This features are optionally supported and make use of another special message called **Destination Advertisement Object (DAO**), which are used by nodes to advertise that they are reachable through a given path.
There are two possible ways to implement these features in RPL, a **storing mode** and a **non-storing mode**:

- Storing mode:
  - DAO messages contain the IPv6 address of their source nodes and they are sent in unicast from nodes to their parents
  - Upon receiving a DAO message, a node add an entry to an internal routing table, which associates the address contained in the DAO message to the child from which the message has been received: that child is the next hop to reach the destination specified by the DAO
  - Then, the same DAO is forwarded by the node to its parents, and so on up to the root
  - Traffic is routed by a node on the base of the information contained in the routing table; if no path is found towards the destination, the message is just routed toward the root
- Non-storing mode:
  - In this case, nodes do not store information, only the root does
  - DAO messages, generated by all the nodes, are made to converge to the root
  - In this case, DAO messages contain the list of all the nodes they have traversed along their path toward the root
  - In this way, the root gets to know all the paths to reach any destination
  - Point to multi-point traffic is routed from the root along the desired path by means of source routing
  - Point to point traffic is first sent to the root and then routed like point to multi-point traffic

CoAP

Constrained Application Protocol (CoAP): application-layer protocol for web applications, similar to HTTP but optimized for constrained devices. The protocol can be used to interact with IoT devices, which become actual web servers, in order to obtain information (for instance, temperature measurements or similar).
CoAP features:

- UDP transport-layer protocol (TCP is too expensive, lightweight reliability mechanisms can be implemented at application-layer, if needed)
- Request/Response model, like HTTP, but with low header overhead:
  - HTTP header is encoded as human-readable text, but it is not lightweight (the reason for this encoding is that in the past there was the need for easy interpretability and debuggability)
  - CoAP header is encoded in binary notation

CoAP is divides the application layer into **two sub-layers**:

- The top layer is the *Request/Response Layer*: messages are encoded as requests or responses (similar to HTTP) and specify the identifier of the target resource and further information
- Below this layer there is the *Asynchronous Message Layer*, which provides four types of messages in which requests/responses can be enveloped:
  - *Confirmable* messages (CON): requests or responses which require an acknowledgement
  - *Non-confirmable* messages (NON): requests or responses which do not require acknowledgements
  - *Acknowledgment* messages (ACK)
  - *Reset* messages (RST) (discussed later)

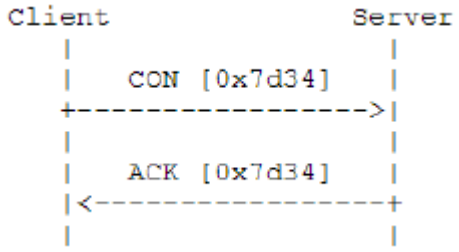
CoAP uses UDP at transport layer, but a **lightweight reliability mechanism** is implemented at the *Asynchronous Message Layer*:

- ACKs + retransmissions with Stop-and-Wait approach and exponential back-off for the retransmission timer for CON messages
- Duplicate detection is based on 16 bits identifiers assigned to messages (like sequence numbers),

called *Message IDs*, both for CON and NON messages

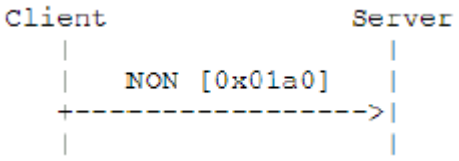
Reliable transmission:

- Initiated by sending a CON message
- The recipient must either
  - Acknowledge the message with an ACK
  - Reject the message, either explicitly, by sending an RST, or implicitly, just ignoring it
- CON and ACK messages are matched by the same Message ID



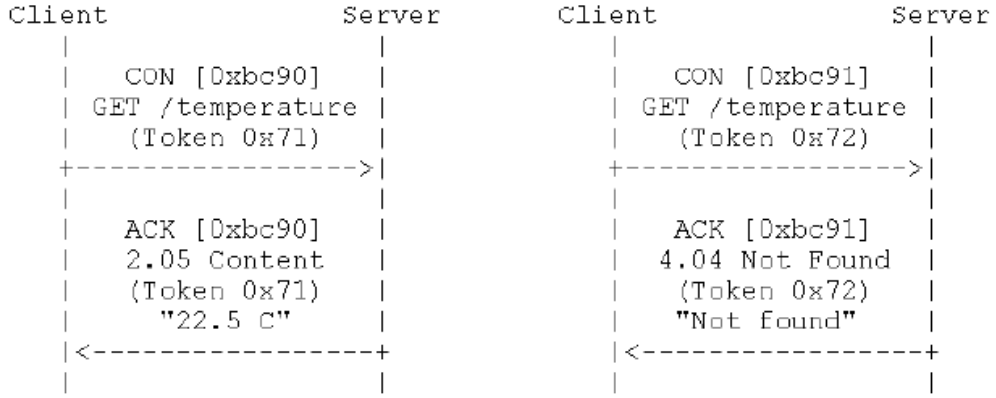
Unreliable transmission:

- Initiated by sending a NON message
- The recipient must not acknowledge the message

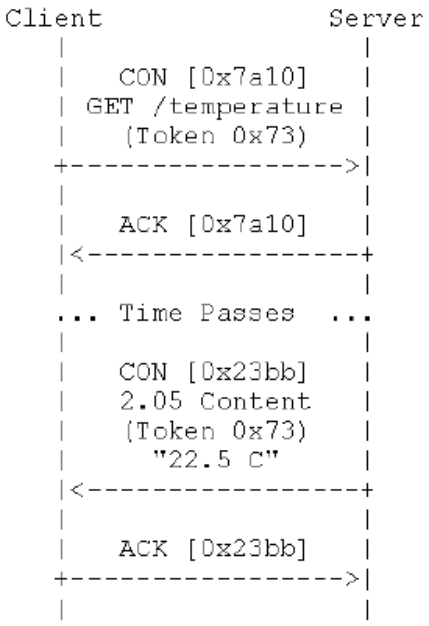


Requests/Responses:

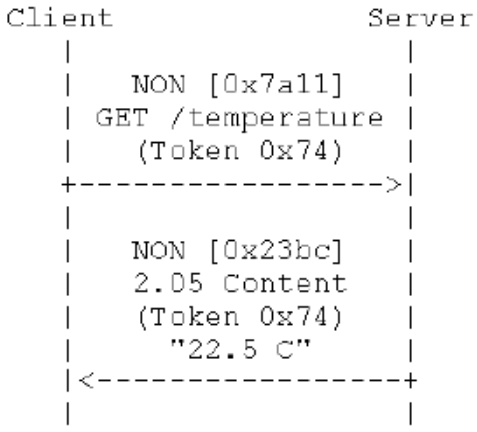
- Same as the HTTP model:
  - Requests specify method, target URI, payload and other optional information
  - Responses specify response code, payload and other optional information
- A *Token* matches requests to responses (not to be confused with the Message ID)
  - In HTTP the Token is not needed because requests and responses are already matched by means of the TCP connection
- If a Request is carried in a CON message
  - The Response may be piggybacked in the corresponding ACK



  - Or it may be sent in a separate CON message

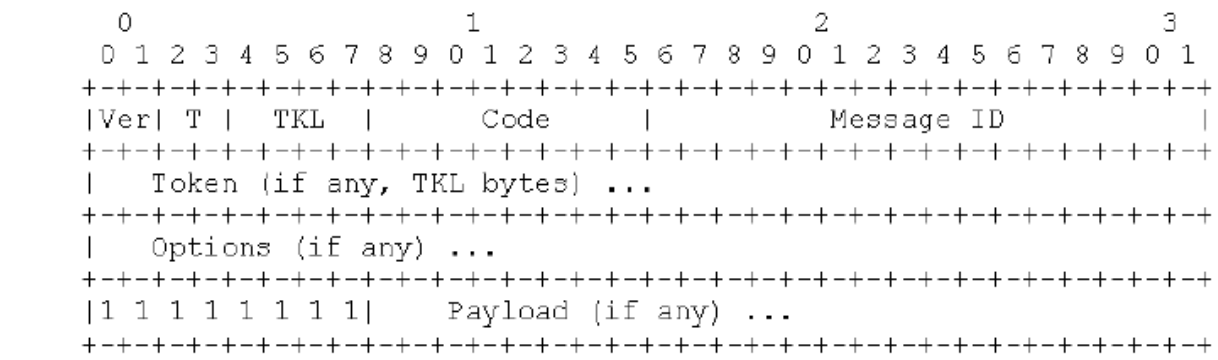


- If a Request is carried in a NON message
  - The Response is carried in a new NON message



- There is a third special type of message: the *Empty* message
  - The Empty message is neither a Request nor a Response
  - It is identified by a special code
  - An Empty message in a CON is used to solicit an RST; this feature is the so-called **CoAP ping**
- **Request methods**: same operations as in HTTP are available, although only four of them are actually part of CoAP
  - GET
  - POST
  - DELETE
  - PUT
- **Response codes**: same numbers as in HTTP are used, although HTTP had five classes, while here there are only three
  - Success: 2.xx
  - Client Error: 4.xx
  - Server Error: 5.xx

Message format:

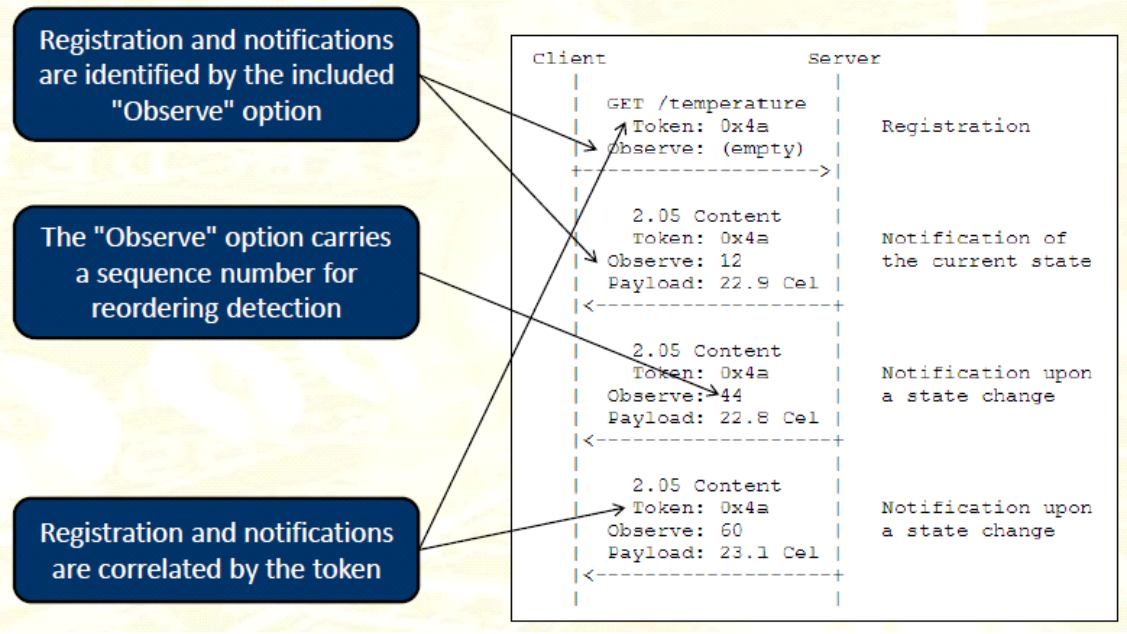


- Four bytes fixed-size header:
  - *Ver*: protocol version (currently = 1)
  - *T*: message type
    - 0: CON
    - 1: NON
    - 2: ACK
    - 3: RST
  - *TKL*: Token length (Token may have variable size)
  - *Code*:
    - Class (first three bits): Request = 0, Success = 2, Client Error = 4, Server Error = 5
    - Detail (last five bits): specifies the method (e.g. 0.01 is GET) or the response code (e.g. 2.04 is 'Created', which notifies the success of a PUT)
    - Special Code 0.00 denotes an Empty message
  - *Message ID*
- **Options** encode further information
  - Among which, the target URI
  - For example: *"coap://host[:port]/path?query-options"*
  - The URI is encoded in four fields:
    - *Uri-Host*
    - *Uri-Port*
    - *Uri-Path*
    - *Uri-Query-Options*
  - Using four fields is more efficient than using a single field:
    - With a single field we should parse the content to obtain the separate pieces of information
    - With four fields, the information is already split --> low parsing complexity
  - Other information encoded in the Options is used for **proxying**:
    - *Proxy-URI*: identifies the proxy server that we want to use
    - *Proxy-Scheme*: encodes either "CoAP" or "HTTP"; if the value is different from the type of the current request, it means that we are asking the proxy to perform translation from one protocol to another (*cross-protocol proxy*)

    Note: a proxy is a server that acts as intermediary, forwarding requests on behalf of clients and sending back responses on behalf of servers; they may also perform HTTP to CoAP translation and vice-versa and perform traffic shaping. In particular, traffic shaping is useful on traffic directed to a wireless sensor network. Sensor nodes typically use uIP (micro-IP), which is a lightweight implementation of IPv6 in which incoming messages are stored in a one-packet-size buffer; therefore, it is easy for a CoAP request to be lost due to a buffer overhead when multiple requests arrive at a node. In order to reduce losses and retransmission, a proxy could perform traffic shaping by keeping requests pending until they can actually be served. This avoids retransmission overhead and actually improves performance.

Let's assume that an client needs to detect when there is a change in a given CoAP resource (e.g. a

change in the temperature measured by a sensor). A simple way to do this is to monitor the resource by sending periodic requests to the server, but CoAP provides also a specific feature called **resource observing**. With resource observing, a client can send a *Registration* request to a CoAP server which, from now on, will send back a *Notification* whenever a change in the target resource occurs.
Example:



Once a client registers for resource observation, the standard doesn't specify how to **unregister**, but there are different possible strategies:

- A possible solution would be to send a new request for the same resource including a new option which specifies that the client wants to unregister
- Another solution, instead of using a new option, is to send an RST message which is interpreted as an unregistration request
- Notifications are followed by ACKs: based on this feature we could implement some garbaging logic that interpret the situation in which no ACK is received for multiple times as an implicit unregistration (this might also happen, for instance, because of the failure of a client)
    - Of course, this solution could lead to mistakes: an observer should always renew the registration after some time during which no notification is received
    - For this purpose, it is possible to use an option to allow the observer to obtain a Notification every x seconds even if the observed resources hasn't changed: this allows the client to know that it hasn't been removed from the set of the observers by the CoAP server by mistake
        - More in general, it is possible to have QoS options to receive a Notification every/at least every/at most every x seconds
- It is also possible to use a *Max Age* parameter to set the maximum duration of a registration a priori

Intermediaries can perform resource observing in place of end clients: a client could perform registration for a resource at a proxy and the latter, in turn, register itself at the final CoAP server. The proxy can handle multiple clients at once, but it registers itself at the CoAP node just once; in this way, the burden on the constrained node is reduced. At the worst case, if the CoAP server doesn't support resource observing, the proxy can obtain information about the resource by means of polling, masking the incapacity of the CoAP server to the end clients.

Resource discovery:

- At the beginning, it is possible to discover CoAP servers in a wireless sensor network by sending a special CoAP message: Multicast CoAP Discovery Request

    - Addressed to the "All CoAP Nodes" special IPv6 multicast address
    - Discovery is bounded to the link-local scope
    - Well-known default port number: 5683
- At this point, it is possible to discover which resources are exposed by a CoAP server
    - Well-known relative URI: *"/.well-known/core"*
        - Default entry point for requesting the list of resources hosted by the server
    - The list of resources is represented with a special language called Link Format, developed by the CoRE (Constrained Restful Environment) working group
    - Link Format: collection of links (a link is a relative URI of a resource, plus a number of attributes) separated by commas
    - Link attributes:
        - *Resource Type* ("rt")
        - *Interface Description* ("if")
        - …
        - *Opaque* attributes: the standard does not define specific values for some attributes; instead, they are defined by the specific application
    - Link example: *"</sensor/temp>; if="sensor""*
    - It is possible to filter the resources returned after a request to the well-known relative URI
        - For example *"GET /.well-known/core?href=/foo*"* returns all the resources whose name starts with foo
        - *"GET /.well-known/core?rt=temperature"* returns all the resources associated to the resource type "temperature"
    - Resources can be grouped in **containers**, for example:
      "REQ: GET/.well-known/core"
      "RES: 2.05 Content
      </sensors>;ct=40"
        - The last line contains the link attribute *Content Type* ("ct"), which specifies the format of the content returned by the resource (for instance "text/plain" or "application/json")
        - The value 40 stands for "application/link-format", which means that the resource returns a list of other resources in Link Format
        - In other words, the resource */sensors* is a container of other resources: a query to a container returns a Link Format description of the resources in that container
        - For example:
          "REQ: GET/sensors"
          "RES: 2.05 Content
          </sensors/temp>;rt="temperature";if="sensor",
          </sensors/light>;rt="light";if="sensor""
        - */.well-known/core* is the container of all the resources
        - This grouping of resources in containers and sub-containers leads to a hierarchical organization of resources (actually there is no hierarchy in a CoAP server, resources are flat, the hierarchy is just what is seen from the point of view of the query)

QoS

Packet Scheduling

Quality of Service (QoS) aims at providing guarantees to flows of packets in terms of delivery delay or assigned rate. Such guarantees are especially needed for specific classes of applications, such as multimedia or critical applications. The traditional internet approach provides only soft guarantees, according to the best-effort approach, through mechanisms implemented at the communication end-points (e.g. lowering down the video encoding quality at the sender, when the available rate in the network decreases, or increasing the playback delay at the receiver so that more packets can fill the receiver buffer before being played). In order to provide better guarantees, we want to implement new mechanisms directly in the network, which allow to reserve a specific amount of resources for given classes of packets (identified by a special *Traffic Class* field in the packet header) or even for single flows (each packet in a flow is identified by the *Flow Label)*. Specifically, we want a protocol that allows to negotiate and reserve the necessary network resources dynamically.
Packets can receive differentiated service at different levels:

- Different input queues associated to different priorities
- At processing time, information about priority found in the routing tables
- Scheduling algorithm at the output queue

Definitions:

- *Flow*: stream of related packets
- A flow is *backlogged* at a router if there is at least one packet of that flow queued at that router
- *Scheduling algorithms* at routers determine how packets from different backlogged flows share the link capacity

Desirable properties of scheduling algorithms:

- Isolation of flows
    - A scheduling algorithm should isolate a flow from undesirable effects of other (possibly misbehaving) flows
    - Ideally, a scheduling algorithm should provide each flow with a (virtually) dedicated link of capacity equal to the minimum guaranteed rate required by the flow
    - Unfortunately, a minimum interference from other flows is unavoidable, due to packetization
        - While the router is sending a packet, the link is busy and cannot serve other flows; therefore, other flows don't see complete isolation, but a channel that sometimes is available and sometimes is busy
        - Packet size should be kept as small as possible
- High utilization of link capacity
    - If we have invested in a lot of network resources, we would like to fully utilize those resources
- Fairness
    - See next paragraph
- End-to-end delay and rate guarantees for each flow
    - Either deterministic (e.g. delay always smaller than x secs) or probabilistic (e.g. delay smaller than t secs 99% of the times)
    - Delay guarantees required by different classes of applications are reported in standard tables (e.g. 150ms for audio conversations)
    - Delay is linked to the rate assigned to a flow by the Little's Law:

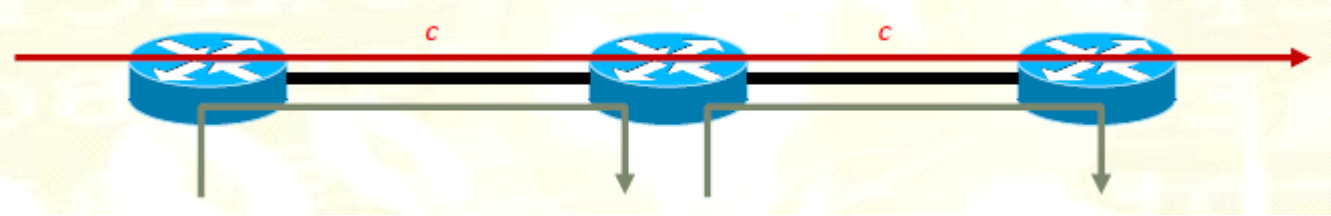
E[delay] = E[#queued-packets]/transmission-rate

- Simplicity of implementation
  - The computational complexity of a scheduling algorithm should be low (O(N), O(log(N)), O(1))

Note: often these properties cannot be satisfied together and we need to find trade-offs; for instance we could reserve a fixed bandwidth to different flows in order to guarantee maximum delivery delay and isolation, but if a flow doesn't transmit for a while its bandwidth is wasted and we do not achieve a high link capacity utilization.

Fairness

Consider the following scenario:



- There is a number of consecutive routers (say, R1, R2, …) connected with links of capacity c (denoted in black)
- There is one flow of packets following the path denoted in red, and a number of flows following the grey paths
  - Let's denote with x0 the rate associated to the red flow
  - Let's denote with x1, x2, … the rate associated to the first, second, … grey flow, respectively

A possible scheduling solution is the one which maximizes the overall throughput. This is obtained by assigning x0 = 0, x1 = c, x2 = c, .., xk = c (assuming there are k grey flows), with an overall throughput of kc. The problem with this solution is that x0 gets no throughput at all, therefore it is not a fair solution.

A possible solution that considers fairness could be to divide the available capacity, at each link, equally among the flows. This kind of fairness can be quantified by means of the **Jain's Index**:

$$f(x_1,\ldots,x_n)=\frac{\left(\sum_{i=1}^{n}x_i\right)^2}{n\sum_{i=1}^{n}x_i^2}$$

- This metric is between 0 and 1
  - It approaches 0 when a single flow gets the whole capacity --> maximum unfairness
  - It approaches 1 when all the flows get an equal share of the capacity --> maximum fairness
- However, this is not always optimal if we look at other metrics, like the utilization network resources
  - If we assume, in the previous example, that the red path is traversed by one flow (n0 = 1), the grey paths except the last one are traversed by three flows (n1 = n2 = … = 3) and the last grey path is traversed by nine flows (nk = 9), the capacity division on the last link is 0.1c to each flow, while on the other links it is 0.25c for each flow
  - However, the flow traversing the red path will have an actual throughput equal to the minimum capacity assigned on the links along its path, which is 0.1c on the last link, while the extra capacity on the previous links will be wasted
  - This allocation is fair, all the flows are happy, but the network owner is not, because we are not fully utilizing the network resources

Another solution which copes with the problems of the previous one is the **Max-Min Fairness**, in which the unused capacity on each link is reassigned to other flows. This is achieved by means of the **progressive filling algorithm**:

- We start increasing the rate of each flow uniformly
- When a flow reaches a bottleneck, we stop increasing it
- We continue increasing only the non-bottlenecked flows until all the flows are bottlenecked

Property of Max-Min Fairness: a rate allocation is Max-Min fair if, in any other allocation, flows getting larger rate do it at the cost of decreasing the rate of an already smaller flow. Formally:

$A\ rate\ allocation\ X\left(x_1, x_2, \dots\right)\ is\ Max\text{-}Min\ Fair\ iff$

$\forall\ rate\ allocation\ Y\left(y_1, y_2, \dots\right) \neq X\ if\ \exists\ j\ s.t.\ y_j > x_j \Rightarrow \exists\ i\ s.t. y_i < x_i\ and\ x_i < x_j$.

We can better see the rationale behind Max-Min Fairness if we consider an example:

- Let's try to increase flow j by a variation δ --> yj-xj = δ
- Let's assume that this is done at the cost of decreasing an already smaller flow i by the same quantity --> yi-xi = -δ
- Let's consider the variation of flow j normalized w.r.t. the original rate xj allocated to that flow: (yj-xj)/xj = δ/xj
- Let's consider the variation of flow i normalized w.r.t. the original rate xi allocated to that flow: (yi-xi)/xi = -δ/xi
- Since xj > xi, we see that δ/xj < δ/xi, which means that the benefit seen by flow j after the variation is smaller than the cost sustained by flow i; therefore, it is not fair to apply this variation
- If, instead, flow i were increased and flow j were decreased, the relation δ/xj < δ/xi would tell us that the benefit seen by flow i is higher than the cost sustained by flow j after the variation; therefore, applying this variation is good

In some sense, Max-Min Fairness privileges flows with smaller rate.

Another aspect to point out is that Max-Min fairness considers only the rate assigned to each flow and not also the overall amount of network resources reserved to that flow. The two things, in fact, do not necessarily coincide. For instance, in our example about red and grey flows, all the flows traversing a link obtain a fair portion of its capacity, but the red flow traverses several links, while each grey flows traverses only one. Therefore, the overall amount of network resources assigned to the red flow is larger than that assigned to anyone of the grey flows. Now, let's assume that a rate allocation assigns a small rate to the red flow and higher rates to the other flows: if we apply a variation that increases the red flow, this is done necessarily at the expense of decreasing many other grey flows. In Max-Min Fairness this is possible, because it's allowed to increase a smaller flow at the expense of higher flows. However, we might not want to always allow that, in which case we need an alternative solution.
In general, if a flow occupies a lot of network resources (traverses several links), an increase of this flow could produce a decrease of several other flows which occupy just a portion of those resources. In this scenario, we may want to understand if the total benefit seen by the increased flows overcomes the overall cost sustained by the decreased flows. In the example above about Max-Min Fairness, we have shown how the incidence of a variation on a flow can be seen in terms of normalized variation (yi-xi)/xi, but Max-Min Fairness considers each normalized variation separately. We may think to consider the normalized variations of all the flows together, in the so-called *proportional balance*:

$$\sum \frac{y_i - x_i}{x_i}$$

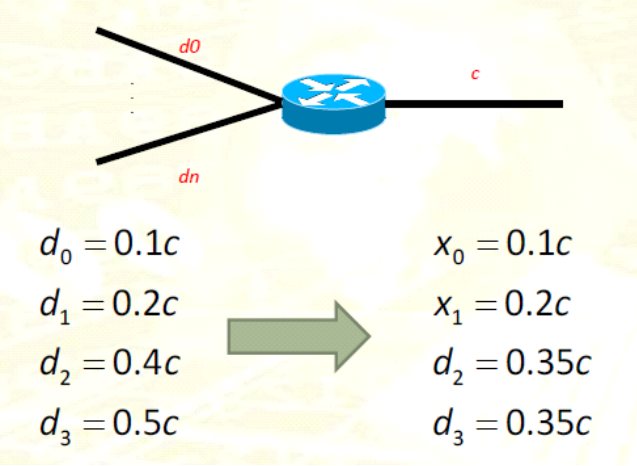
The rationale behind this sum is that if we apply a variation to the rate allocation and we obtain a positive balance it means that overall benefit obtained by increased flows overcomes the overall cost sustained by decreased flows, and therefore the new allocation is good.
This approach is called **Proportional Fairness**.
Formally, Proportional Fairness is defined by means of the following **property**:

$A\ rate\ allocation\ X\left(x_1, x_2, \dots\right)\ is\ Proportionally\ Fair\ iff$

$\forall\ rate\ allocation\ Y\ (y_1, y_2, \dots) \neq X, \sum \frac{y_i - x_i}{x_i} \leq 0$.

This property aims at allocating rate uniformly, like Max-Min Fairness, but it also implicitly limits those flows which occupy more network resources. The conceptual difference between the two solutions is that Max-Min Fairness considers each flow separately, while in Proportional Fairness all the normalized variations are considered together in the proportional balance.
It has been shown that the solution to the problem of Proportionally Fair rate allocation, provided by the operative research, consists in maximizing the sum of ln(xi), under the constraints of not exceeding the capacity of any link.

It is also possible to **consider that flows have different rate demands**, in which case the previous solutions can be easily adapted. For instance, in Max-Min Fairness, if we don't consider the demand of each flow in the scenario in the figure below, we end up with an allocation (shown in the figure) which satisfies the flows with lower demand, but not those with higher demand. In this case, it would be fairer to assign rate to flows proportionally to their demands.



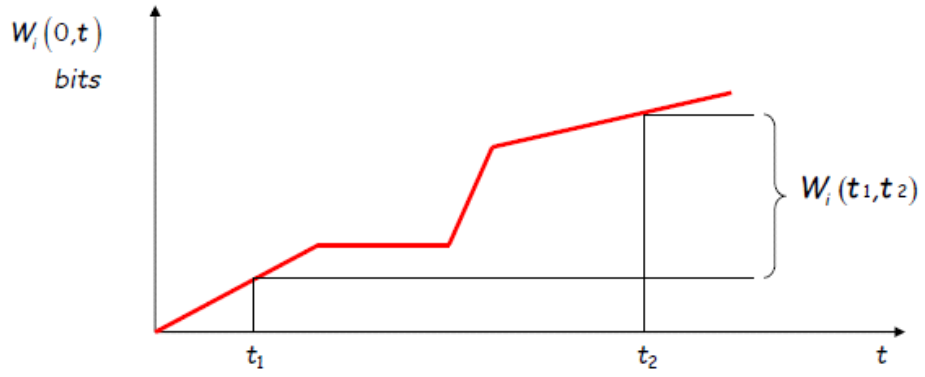
The algorithm of proportional filling for Max-Min Fairness can be easily adapted to this scenario: instead than increasing the rate of all the flows uniformly, we increase it proportionally w.r.t. the demand of each flow. The property of Max-Min Fairness becomes that allocation X is fair if there is no other allocation Y!=X s.t.:
$if\ \exists\ j\ s.t.\ y_j > x_j \Rightarrow \exists\ i\ s.t. y_i < x_i\ and\ x_i/d_i < x_j/d_j.$
Instead, Proportional Fairness can be adapted by weighting the terms in the sum of the proportional balance by their corresponding rate demands and by solving the problem of maximizing the sum of di*ln(xi), instead than just the sum of ln(xi).

We have not considered yet how a scheduling algorithm allocates the link over time. Example:

- Two flows sharing the link capacity and getting 50% of the bandwidth each
- Due to packetization, the link is not simultaneously used by the two flows, but they use it alternatively, and each one uses the link for the 50% of the time in total
- There are several ways to schedule packets in order to fulfill this bandwidth requirement:
    - One packet of the first flow and one of the second, alternatively
    - Two packets of the first flow and two of the second, alternatively
    - …

The previous definitions of fairness didn't take time into consideration. The better solution is the one in which one packet of the first flow and one of the second are sent alternatively, i.e. the link capacity is shared with a more fine-grained granularity. In this way the isolation between the flows is maximized, because each flow sees the channel busy for the minimum time while the other flow is being served.
Golestani's **Service Fairness Index** (SFI) is a fairness metric that considers also the time problem presented so far.

Definition: maximum difference between the normalized service received by two flows over an interval in which they are both continuously backlogged.
The amount of service received by a given flow i (in terms of units of traffic, e.g. bits) under scheduling discipline S during time interval (t1, t2] is denoted with $W_i^S(t1, t2)$.



The measure for the SFI is the *relative fairness*. The relative fairness of two flows i and j, during the time interval (t1, t2] in which they are both continuously backlogged and under any scheduling discipline S, is defined as:

$$RF_{(i,j)}(t_1,t_2) = \left| \frac{W_i^S(t_1,t_2)}{r_i} - \frac{W_j^S(t_1,t_2)}{r_j} \right|$$

where ri and rj are the service rates allocated to flows i and j respectively.

The relative fairness with respect to a flow i over time interval (t1, t2] is defined as

$$RF_i(t_1,t_2) = \max_{\forall j} RF_{(i,j)}(t_1,t_2)$$

The relative fairness over time interval (t1, t2] is defined as

$$RF(t_1,t_2) = \max_{\forall i} RF_i(t_1,t_2)$$

The relative fairness bound is defined as

$$RFB = \max_{\forall (t_1,t_2]} RF(t_1,t_2)$$

The lower the relative fairness, the fairer the service is. However, due to packetization, it is not possible to realize arbitrary short-term fairness: theoretical results tell us that the relative fairness is lower-bounded

$$RF_{(i,j)}(t_1,t_2) \geq \frac{1}{2}\left(\frac{L_i}{r_i} + \frac{L_j}{r_j} \right)$$

Where Li and Lj are the maximum packet sizes of flow i and flow j respectively.

Instead, the relative fairness bound is unbounded for scheduling algorithms that present long-term unfairness

- RFB increases indefinitely with t2-t1
- One flow is not service at all for an arbitrary long period (it is starved)
- Example of unfair scheduler: static priority

Characteristics of Scheduling Algorithms

Work-conserving vs non-work-conserving

- Work-conserving: always start a new transmission when the link is idle and the buffer is not empty
- Non-work-conserving: may defer a transmission even if the link is idle
    - Rationale: it is possible that the link is idle and there is a low-priority packet in the buffer
    - If the router starts sending this packet and, in the meantime, a higher-priority packet arrives, the transmission of the high priority packet has to be deferred, because the link is occupied by the low-priority packet transmission
    - A non-work-conserving policy might allow to defer the transmission of the low priority packet, so that, if a high priority packet arrives, it can be immediately transmitted

Sorted-priority vs frame-based priority:

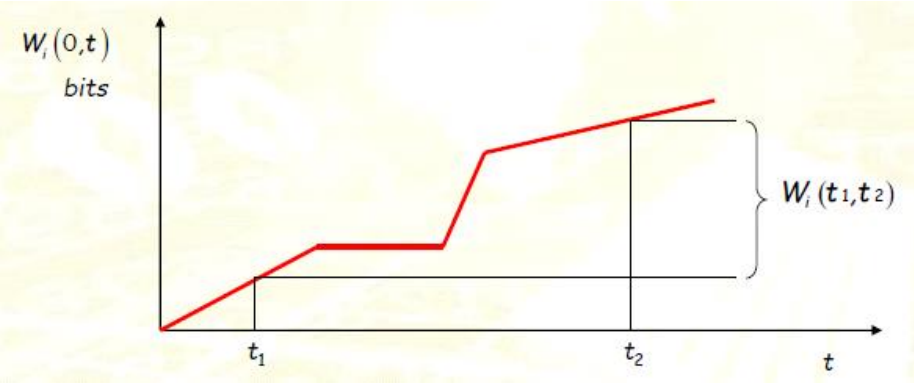
- Sorted priority: a *stamp* is applied to each incoming packet, which defines an order in which packets have to be processed
    - Stamps are applied and ordered according to some specific policy
    - Packets will be transmitted by increasing stamp order
- Frame-based priority: time is divided into slots (frames) and each flow has some slots allocated to it
    - Each flow can transmit only during the allocated time slots

Generalized Processor Sharing

Let's assume that we have N flows [1, 2, …, N] sharing a link of capacity r. Each flow i is associated to a weight ϕ_i (real positive number) that represents the service demand requested y that flow.
Generalized Processor Sharing (GPS) is a scheduling algorithm that considers the traffic as following the **fluid model**:

- Packets are indefinitely divisible
- All the flows can be served simultaneously, as long as the total throughput doesn't exceed the overall capacity

Flows are served with a relative share of the total link capacity proportional to their respective demands.
This is an **ideal algorithm**, it can't be realized in practice because packets are not indefinitely divisible and only one flow can be served at a time (only one packet at a time can be sent on the link and when the link is sending a packet from one flow it is busy for all the other flows). However, this algorithm can be used as a reference to evaluate all the other algorithms.
In the following, we denote with $W_i(t1, t2)$ the amount of service (measured in traffic units, e.g. bits) received by flow i in the time interval (t1, t2]



GPS formal definition: work conserving discipline such that

$$\frac{W_i(t_1,t_2)}{\Phi_i} \geq \frac{W_j(t_1,t_2)}{\Phi_j}, j = 1, \dots, N$$

For any flow i continuously backlogged in the interval (t1, t2].

We notice that if two flows i and j are simultaneously backlogged, both the inequalities

$$\frac{W_i(t_1,t_2)}{\Phi_i} \geq \frac{W_j(t_1,t_2)}{\Phi_j}$$

$$\frac{W_j(t_1,t_2)}{\Phi_j} \geq \frac{W_i(t_1,t_2)}{\Phi_i}$$

must hold. Follows that

$$\frac{W_i(t_1,t_2)}{\Phi_i} = \frac{W_j(t_1,t_2)}{\Phi_j}.$$

In fact, the inequality comes out when a flow is not backlogged, in which case no service is provided to it and the unused capacity is reassigned to the other flows.

GPS is perfectly fair according to SFI:

$$RFB_{i,j} = \underset{t_2>t_1}{Max}\left|\frac{W_i(t_1,t_2)}{\Phi_i} - \frac{W_j(t_1,t_2)}{\Phi_j}\right| = 0$$

$$RFB = \underset{i,j}{Max}\left(RFB_{i,j}\right) = 0$$

GPS provides a **minimum guaranteed rate** that can be computed as follows:

- Rewrite the inequality that defines GPS as

  $$\frac{W_i(t_1,t_2)}{W_j(t_1,t_2)} \geq \frac{\Phi_i}{\Phi_j}, \quad j = 1,2,..,N$$

  and sum over all the j's

  $$W_i(t_1,t_2)\sum_{j=1}^{N}\Phi_j \geq \Phi_i\sum_{j=1}^{N}W_j(t_1,t_2)$$

- Since GPS is work-conserving and since flow i is continuously backlogged in (t1, t2], follows that the throughput on the link during (t1, t2] is r; from this consideration we can write

  $$\sum_{j=1}^{N}W_j(t_1,t_2) = r(t_2 - t_1)$$

  which can be replaced in the previous formula, thus obtaining

$$W_i(t_1,t_2) \geq g_i(t_2 - t_1)$$

where

$$g_i = \frac{\Phi_i}{\sum_{j=1}^{N} \Phi_j} r$$

is the minimum guaranteed rate of flow i.

In general, the **instantaneous rate** dedicated to a flow can be larger than g_i, because, when another flow is not backlogged, the unused capacity is reallocated to the backlogged flows, in proportion to their weights.

To be precise, we can consider a time interval (t, t + Δt]. The instantaneous rate is the variation of service provided to the flow in this time interval over the length of the interval itself, when Δt becomes infinitesimal, i.e. the derivative of W_i(t, t + Δt) w.r.t. time.

We define the set B(t) as the set of all the backlogged flows at time t. When the time interval is sufficiently small, we can consider the set B not changing in that period.

We can sum up again the previous inequality, as before, but only for those j ϵ B(t). Taking the limit for Δt --> 0, we obtain the instantaneous rate

$$r_i(t) = \frac{\Phi_i}{\sum_{j \in B(t)} \Phi_j} r$$

Vice-versa, the amount of traffic units that flow i transmits according to GPS in a time interval (t1, t2] can be computed as the integral of the instantaneous rate over this time interval:

$$W_i(t_0,t_1) = \int_{t_0}^{t_1} r_i(t)dt = r \int_{t_0}^{t_1} \frac{\Phi_i}{\sum_{j \in B(t)} \Phi_j} 1_{i \in B(t)} dt$$

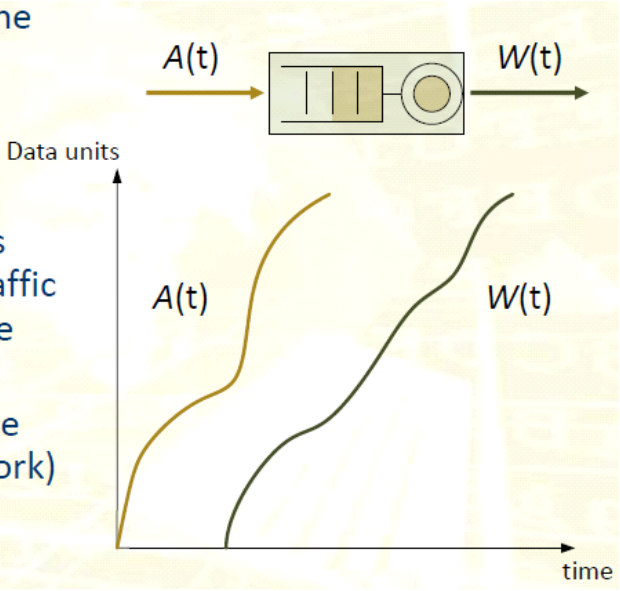
GPS can also provide delay guarantees to flows: a unit of traffic arriving at time t will experiment a delay bounded as follows:

$$(Delay)_i \leq \frac{Q_i(t)}{g_i}$$

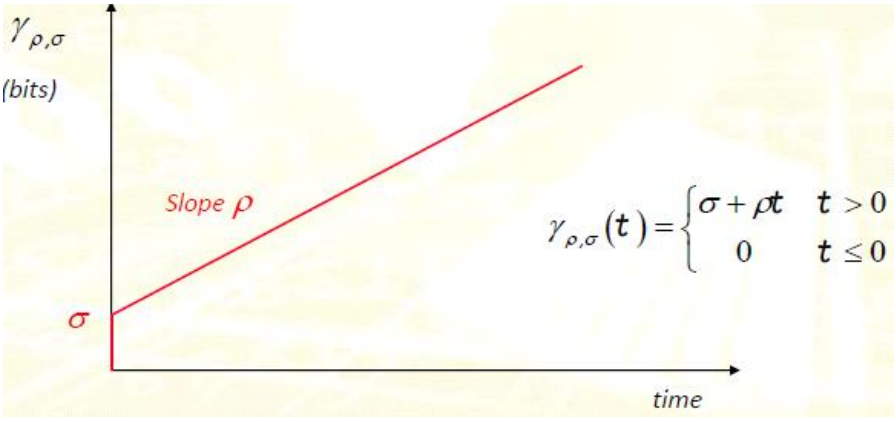
Where Q_i(t) is the size of the backlog queue of flow i at time t. Note that the delay bound is independent of the other flows.

In order to fully characterize this delay we need to characterize Q(t)

- *A(t) input function* is the amount of data traffic that arrives into the system in the time interval [0, *t*]
- *W(t) output function* is the amount of data traffic transmitted in the time interval [0, *t*]
- *Q(t)* := *A(t)* – *W(t)* is the backlog (unfinished work) at time *t*.

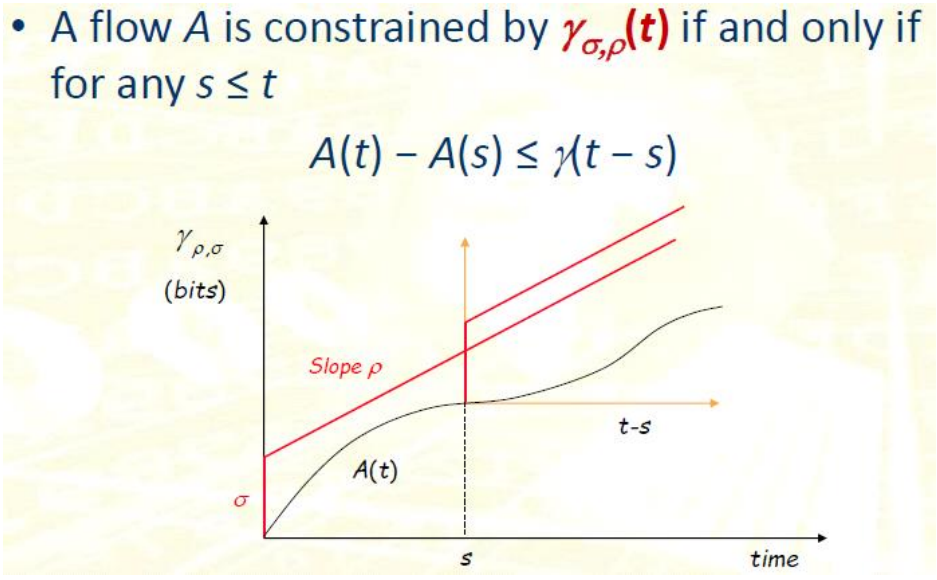


The arrival process can be characterized with some specific models. There can be stochastic characterizations, but also models that define deterministic bounds to the traffic. Among the latter class of models, a popular one is the **(σ, ρ) traffic model**



- A flow *A* is constrained by $\gamma_{\sigma,\rho}(t)$ if and only if for any $s \le t$

$$A(t) - A(s) \le \gamma(t - s)$$

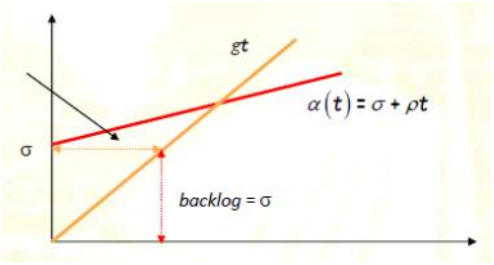


- In this model, the exact traffic stream is unknows; however, the traffic has to meet the following constraints:

- It will never exceed σ + ρΔt units of data over any time interval Δt
- The long-term average arrival rate is ρ
- The source is allowed to send bursts of traffic into the network, but the maximum burst size is σ units of traffic

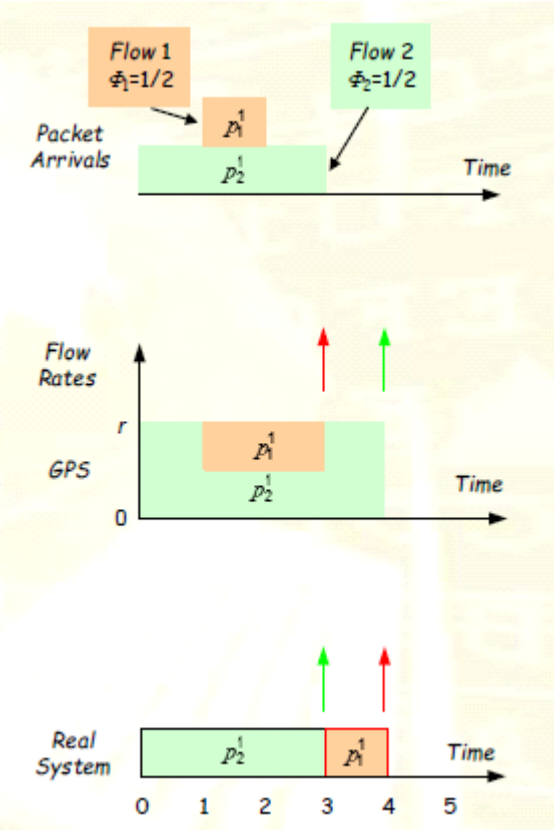
Consequences of assuming a (σ, ρ) traffic model:

- In order to keep a flow stable, the minimum guaranteed rate provided to it has to be $g > \rho$
- The maximum delay is $delay_{max} = \sigma/g$



Weighted Fair Queueing

Since GPS cannot be implemented in practice, we look for another solution that tries to approximate GPS as much as possible. This solution is called **Weighted Fair Queueing (WFQ)**: work-conserving scheduling discipline that outputs packets by increasing order of GPS departure times $d_i^{k, GPS}$.
The idea is to simulate the behavior of GPS for all the incoming packets, computing the departure time of each packet in the GPS system. Packets in the real system are transmitted in the same order of GPS departure times that we have just computed. Following this approach, we should be able to output packets in the same order as GPS.
Actually, this solution is impossible to implement, because such a system would not be causal. Let's clarify this point with an example:



Here, we see that next packet to serve according to GPS is the red one, but it is not arrived yet at time 0 and there is no way for the scheduler to know that it is going to arrive in the future. Instead, WFQ starts serving the green packet (the scheduler must always transmit something, because it is work-conserving). When the red packet arrives, GPS simulation tells the scheduler that this packet should have been served first, but the green packet is already in transmission and cannot be stopped: the final outcome is inevitably different from GPS. In general, the next packet to serve might have not arrived yet: WFQ needs information about what packets are going to arrive in the future in order to simulate GPS (information about the future --> non-causality).
After the previous considerations, we redefine WFQ as a work-conserving scheduling discipline that outputs packets by increasing order of GPS departure times, assuming that no additional packets were to arrive. Basically, we know that our scheduling algorithm could produce a different outcome w.r.t. GPS, but we accept this.

It is possible to quantify the **difference between WFQ and GPS**. The departure time of a packet in WFQ cannot go beyond the corresponding departure time in GPS more than the time necessary to send a maximum-size packet:

$$d_i^{k,WFQ} - d_i^{k,GPS} \leq \frac{L_{max}}{r}$$

L_{max} is the maximum packet length.
Note that this bound prevents packets from leaving too late in WFQ w.r.t. GPS, although the reverse does not hold: some packets can leave much earlier in WFQ than in GPS (see example on WFQ fairness later).
Follows that the worst-case delay in WFQ, under the (σ, ρ) traffic arrival model, is equal to that of GPS, with the addition of a term accounting for the difference presented above:
$delay_{max} = \sigma/g + L_{max}/R$.

Now, we present how to simulate GPS in real-time, but first we need to define the concept of **virtual**

time. Let's assume that we are in a GPS system and let's consider the set of backlogged flows at time t, B(t); this set varies when at discrete time instants, when some event occur (a flow changes its state from empty to backlogged or vice-versa). Let's assume that these events occur at times t1, t2, …
Virtual time is a function of time, V(t), defined as:

$$V(0) = 0$$

$$V\left(t_{j-1} + \tau\right) = V\left(t_{j-1}\right) + r \frac{1}{\sum_{i \in B(t_{j-1}+\tau)} \Phi_i} \tau$$

where

$$\tau < t_j - t_{j-1}$$

and tj-1 and tj are any two consecutive time instants among those in which events that change B(t) occur.
Equivalently, the virtual time in a period [0, t] can be computed as:

$$V(t) = r \int_0^t \frac{1}{\sum_{j \in B(\tau)} \Phi_j} d\tau$$

The virtual time is related to the amount of service provided to a flow i during time interval (t', t''] in which it is continuously backlogged:

$$W_i(t', t'') = \int_{t'}^{t''} r_i(t) dt = r \int_{t'}^{t''} \frac{\Phi_j}{\sum_{j \in B(t)} \Phi_j} dt$$

$$W_i(t', t'') = \Phi_i \left[V(t'') - V(t') \right]$$

Now we consider the following quantities:

- Arrival time of the k-th packet from flow i: a_i^k
- Its length: L_i^k
- Its departure time in the GPS system: $d_i^{k, GPS}$
- The departure virtual time: $F_i^k = V(d_i^{k, GPS})$
- The start-of-service virtual time, which is the maximum between the virtual arrival time of this packet and the virtual departure time of the previous one: $S_i^k = \max(V(a_i^k), F_i^{k-1})$

Our goal is to obtain $d_i^{k, GPS}$ from all the other quantities.
Since flow i is continuously backlogged during $[a_i^k, d_i^{k, GPS}]$, we can write

$$W_i\left(a_i^k, d_i^{k,GPS}\right) = \Phi_i \left(F_i^k - S_i^k\right)$$

And then, since $W_i(a_i^k, d_i^{k, GPS}) = L_i^k$, follows that

$$F_i^k - S_i^k = \frac{L_i^k}{\Phi_i}$$

$$F_i^k = S_i^k + \frac{L_i^k}{\Phi_i}$$

At this point we have the virtual departure time of the packet and we should obtain back the GPS departure time. However, this is not necessary. In fact, we can notice that the function V(t) is monotonically increasing w.r.t. time, therefore it is identical to order packets according to $d_i^{k, GPS}$ or F_i^k.
WFQ is redefined equivalently: work-conserving scheduling discipline transmitting packets by increasing order of GPS virtual departure times.
Practical WFQ implementation:

- Keep track of virtual departure times of head-of-line packets per each flow (it is not necessary to keep track of the virtual departure times of all the packets, just the head-of-line packets per each flow are enough) with the formula

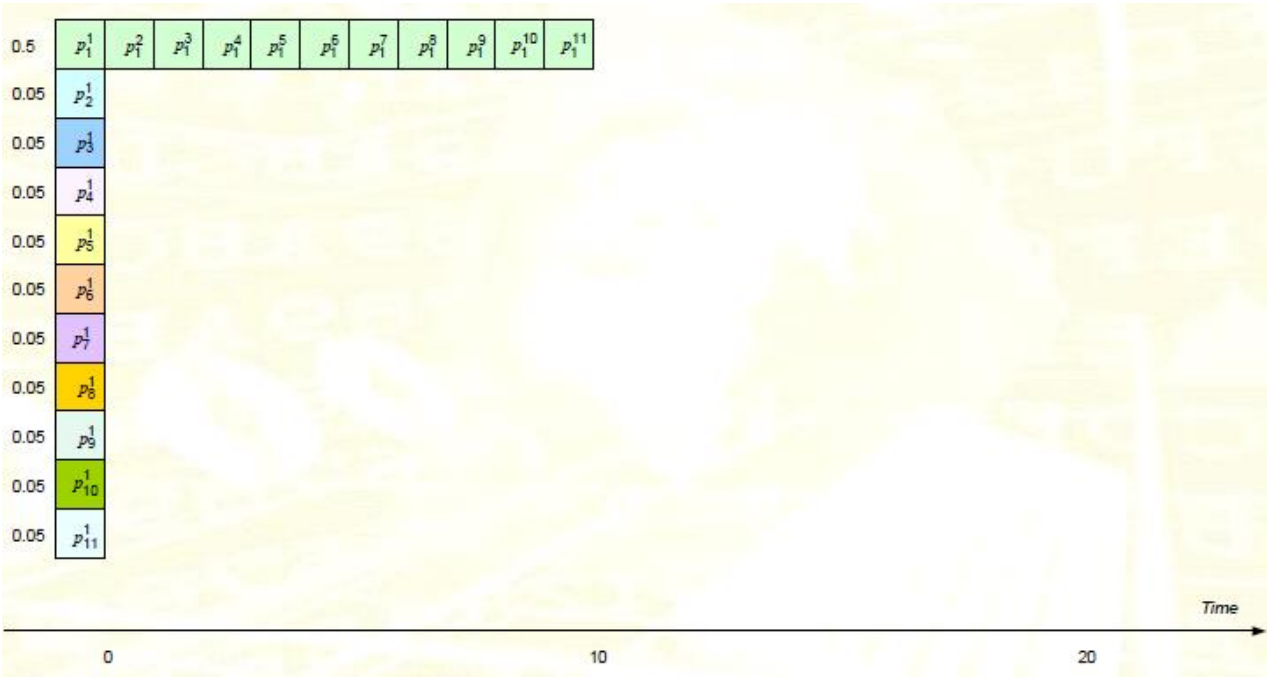
$$F_i^k = \max\left(F_i^{k-1}, V\left(a_i^k\right)\right) + \frac{L_i^k}{\Phi_i}$$

- Keep track of the GPS system virtual time in real time with the formula

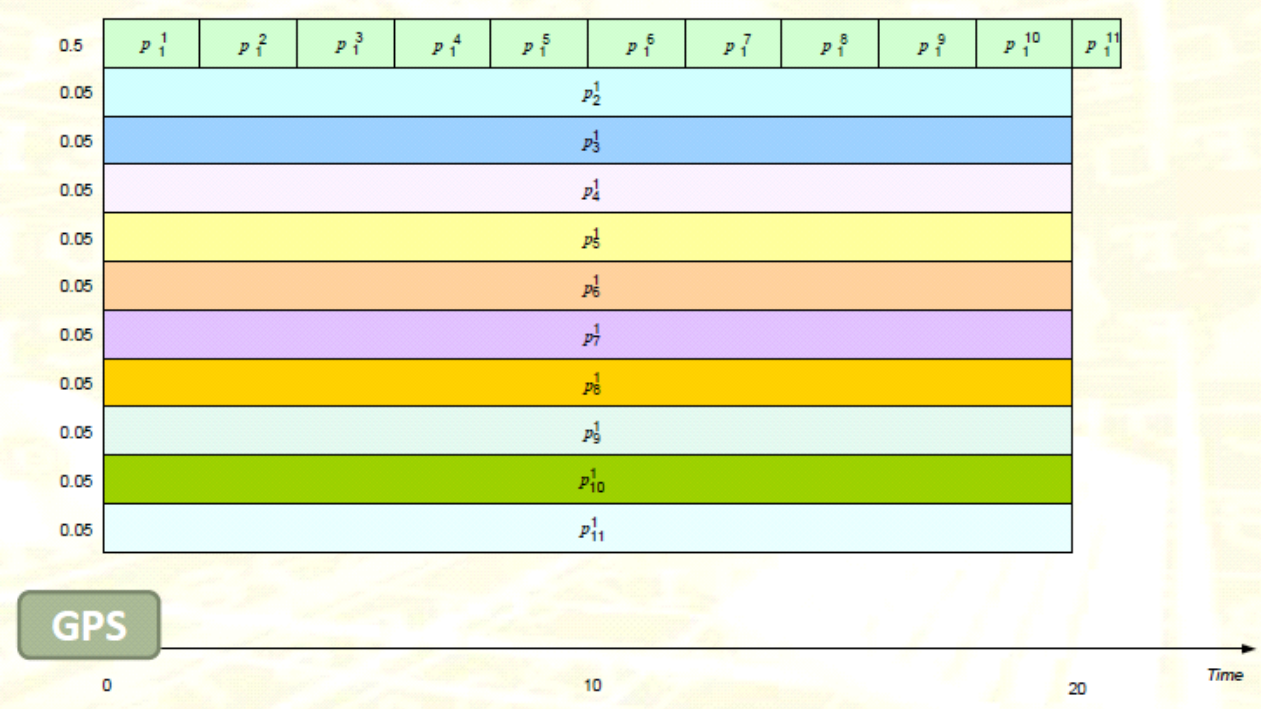
$$V\left(t_{j-1} + \tau\right) = V\left(t_{j-1}\right) + r \frac{1}{\sum_{i \in B\left(t_{j-1}+\tau\right)} \Phi_i} \tau$$

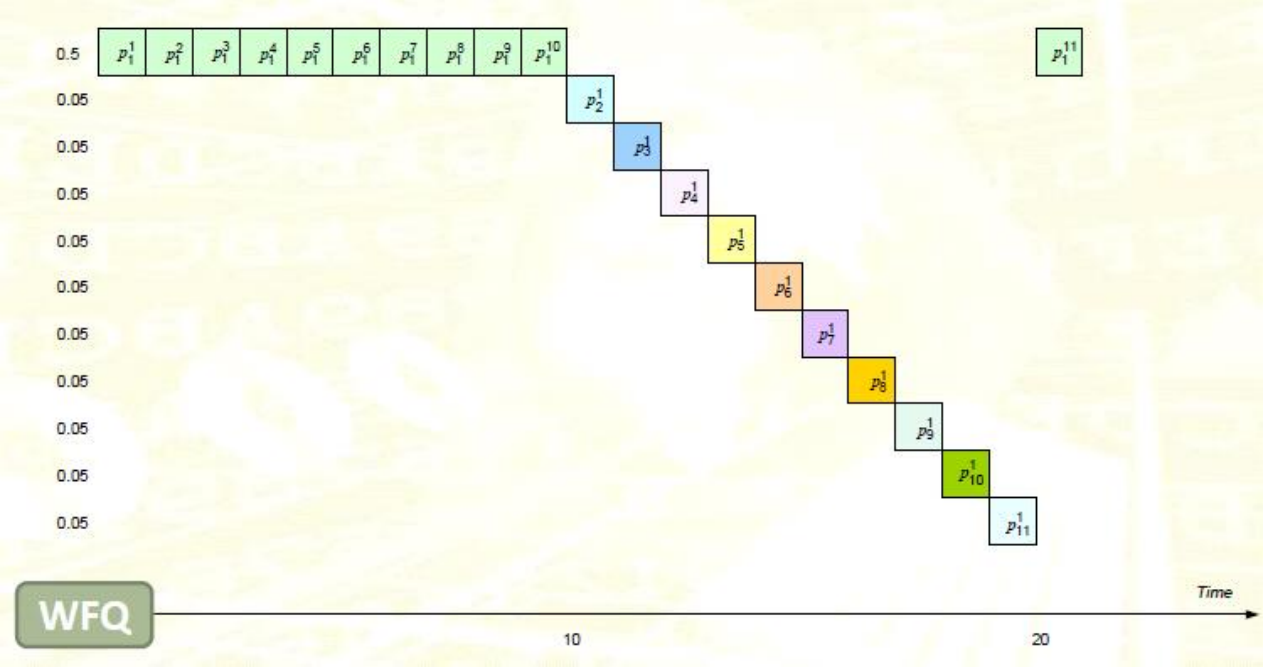
- The latter point requires keeping track of B(t) over time
  - Traditionally, this operation used to have linear complexity
  - An O(log(n)) solution has been found

We have already pointed out that, in WFQ, packets can leave the system much earlier than in GPS. This may produce **fairness issues with WFQ**. Let's consider an example:



There are eleven flows; the first one has demand 0.5 and the others have demand 0.05. The figure above shows the packet arrivals at each flow over time: eleven packets arrive at the first flow and one at each other flow. Let's see how GPS and WFQ schedule these packets:

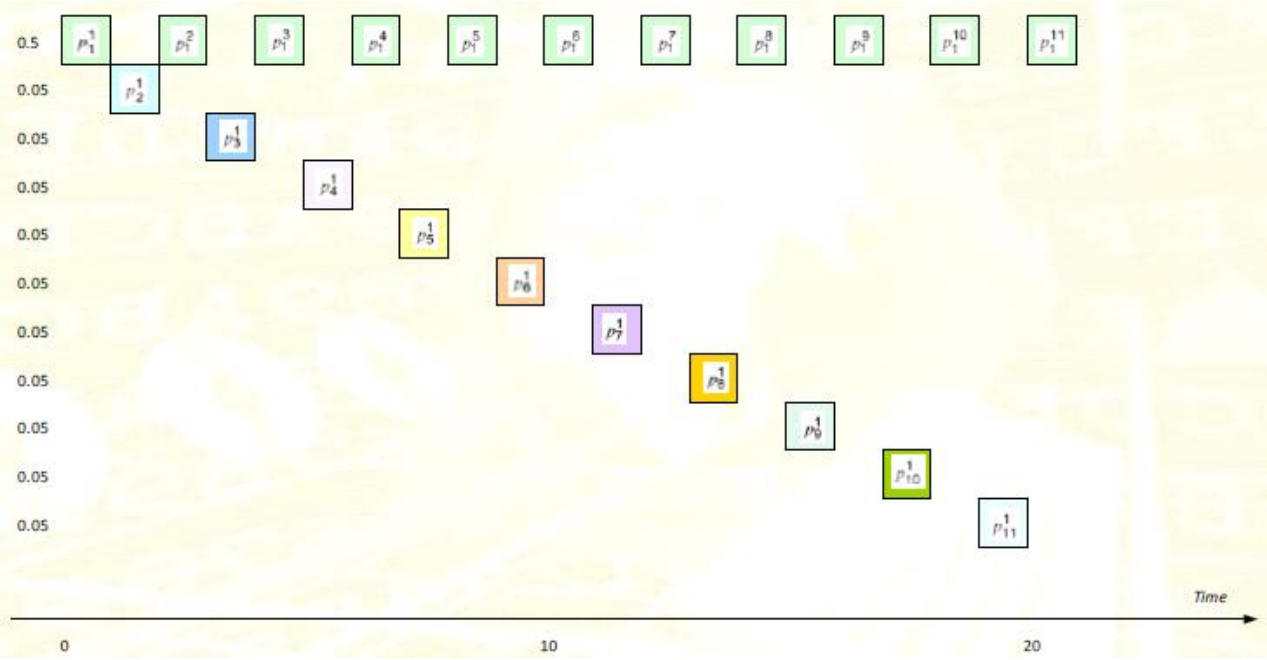




According to the rule of WFQ scheduling, ten packets from the first flow are served at the beginning, and then the packets from the other flows are served, but this way of distributing service over time is very unfair.

To cope with this fairness problem, a variant of WFQ, called **Worst-Case Weighted Fair Queueing** (denoted with **$(WF)^2Q$** or **WF2Q** for brevity), has been proposed. The solution is very simple: when, at time t, the server has to transmit the next packet, it chooses the first one that would complete the service in the corresponding GPS system, but only from those that have already started receiving service in the corresponding GPS system (and not from all the packets queued at time t).

The result of these new approach applied to the previous case is much fairer, as shown in the following figure:

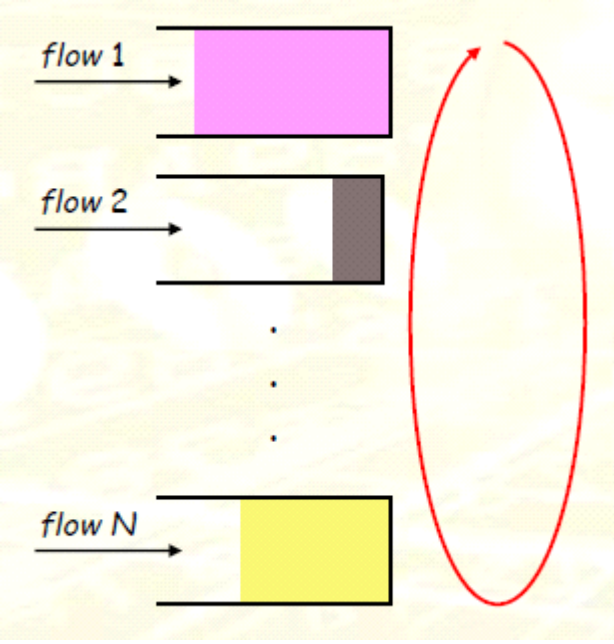


WF2Q implementation:

- Virtual time is still used
- A packet is *eligible* for service the current virtual time is >= its virtual start-of-service time
- Next packet to serve is chosen from the eligible ones only

Deficit Round Robin

Deficit Round Robin (DRR) is a work-conserving frame-based packet scheduling algorithm in which flows are served in order and each flow is allowed to transmit a bounded amount of traffic at each turn.



Each flow is characterized by two quantities: a *quantum* and a *deficit counter*. The quantum of flow i, say Qi is the amount of traffic that flow i should ideally transmit during a round. Ideally, because it might happen, for instance, that there are two packets of 700 bytes to transmit and a quantum of 1KB. In this case, the first packet can be transmitted normally, but the second cannot, because by doing so we would exceed the quantum. In this situation, the turn is terminated beforehand, but the unused portion

of quantum is stored in the deficit counter, DCi, and it will be recovered in the next round. DCi represents the credit in terms of service towards flow i. In this way, we guarantee that, in the long run, the average amount of service provided to flow i per turn is Qi (as long as flow i is backlogged). Another possibility is that there is just one packet of 700 bytes to transmit, after which the queue of flow i becomes empty. In this case, the turn is also terminated beforehand, but the deficit counter is set to 0, so that no service credit is given to non-backlogged flows and more resources can be saved for the backlogged ones.
A larger quantum will be associated to flows with higher demand.
The operations of DRR are the following:

- Flow i becomes backlogged
- Init: DCi = 0
- Turn:
    - DCi = DCi + Qi
    - Transmit up to DCi bytes:
      while(*more packets to transmit*)
          p = *next packet to transmit*
          if(p.length <= DCi)
              transmit p
              DCi = DCi - p.length
          else
              end turn
    - If queue of flow i gets empty:
      DCi = 0
      end turn

DRR implementation:

- Backlogged flows are kept in an *ActiveList*
- DRR worst-case complexity is O(1), provided that the maximum packet length of flow i is smaller than Qi
    - If it is, we are sure we can send at least one packet per turn
    - If it isn't, DRR can work anyway:
        - If a packet is larger than the quantum, we accumulate the whole quantum in the DCi and we finish the turn
        - Next turn, if the packet is still larger than the quantum plus the value accumulated in the DCi, we accumulate another quantum in the DCi (thus, the DCi will be equal to 2Qi), and so on
        - In this way, the DCi will keep growing until, eventually, it will be possible to serve the packet
    - However, if the maximum packet length of flow i larger than Qi, the worst-case complexity grows:
        - When we start a new turn, we move to the next flow in the list, but we find out that the size of next packet is too large
        - Therefore, we terminate the turn and we move to the next flow in the list
        - Again, we start a new turn but we might find out that next packet is too large
        - Therefore, also this turn is terminated and we move to the next flow in the list, and so on, until a packet is found that we can send
        - In the worst case, we might have to traverse the whole list of active flows before we find a packet that we can transmit, which has complexity O(N)

DRR guarantees (can be proven but we didn't):

- Rate

$$g_i = \frac{Q_i}{F} r \qquad F = \sum_{i=1}^{N} Q_i$$

- Fairness

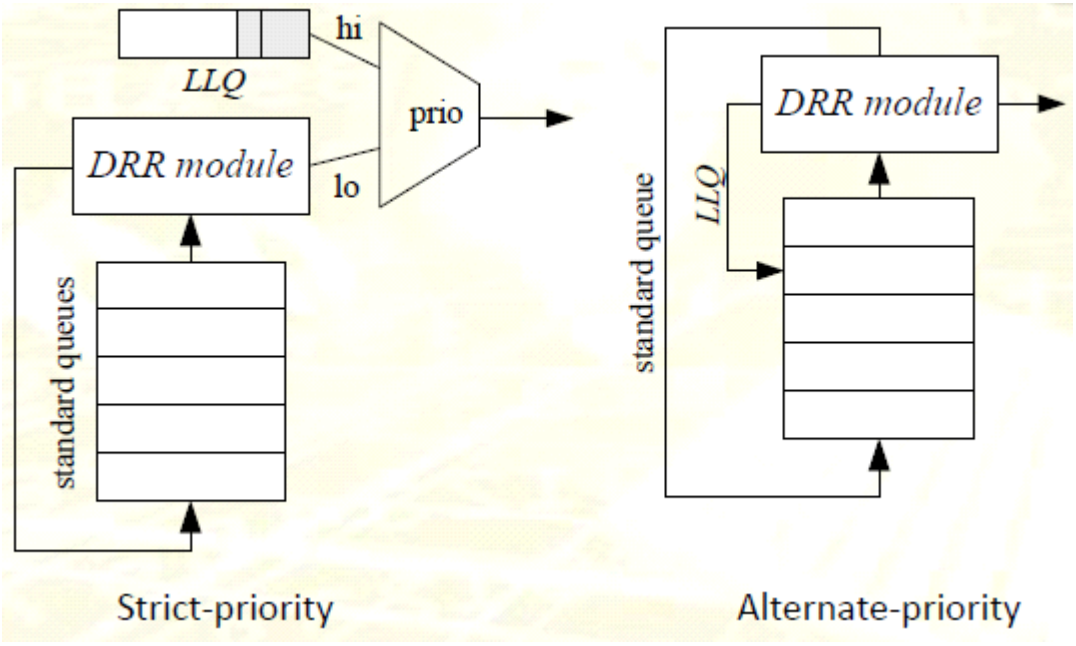
$$RFB_{i,j} = F\left(1 + \frac{L_{i,\max}}{Q_i} + \frac{L_{j,\max}}{Q_j}\right) \qquad RFB = 3F$$

- Latency

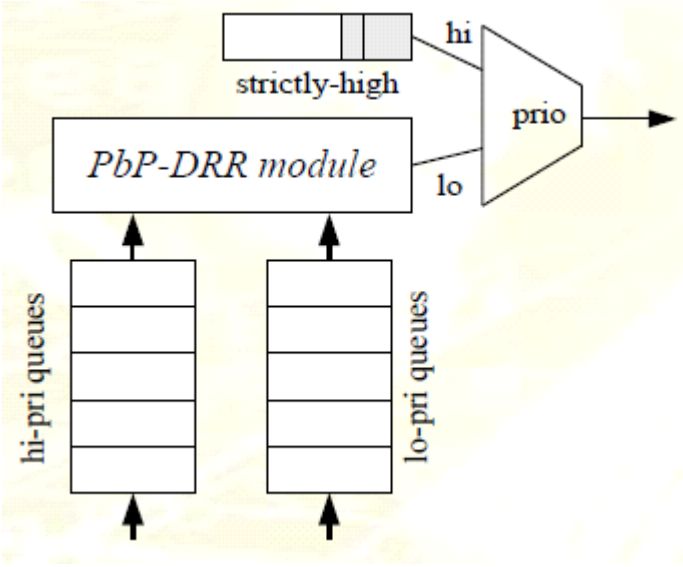
$$T_i = \frac{3F - 2Q_i}{r}$$

DRR in commercial routers:

- Cisco IoS



  - Strict priority: packets in the Low Latency Queue (LLQ) are transmitted as soon as the link is free; when the LLQ is empty, packets from the DRR module are transmitted
    - This solution allows to give stronger delay guarantees to high priority flows
  - Alternate priority: the LLQ is included in the regular queue; in this case we alternate one quantum for the LLQ and one quantum for the standard flows
    - This version smooths the differences between high-priority and low-priority flows, which is better when the high priority queue risks to make the regular queues starve
- Junos



- Junos routers mix up the two previous solutions: there is a strictly-high priority queue and a DRR module handling other high-priority queues and low-priority queues alternatively
  - The difference w.r.t. handling a single set of queues containing both high-priority and low-priority flows is that with a unique queue we cannot provide better guarantees for high priority packets: a packet arriving at a high priority queue will be served only after all the previous queues in the round are served, both high-priority and low-priority
  - With two queues, instead, since the high-priority flows are supposedly less than the low-priority flows, we can assume that rounds on the high priority flows last less; follows that high priority flows receive service more often, thus achieving stronger delay guarantees

IP QoS Architectures

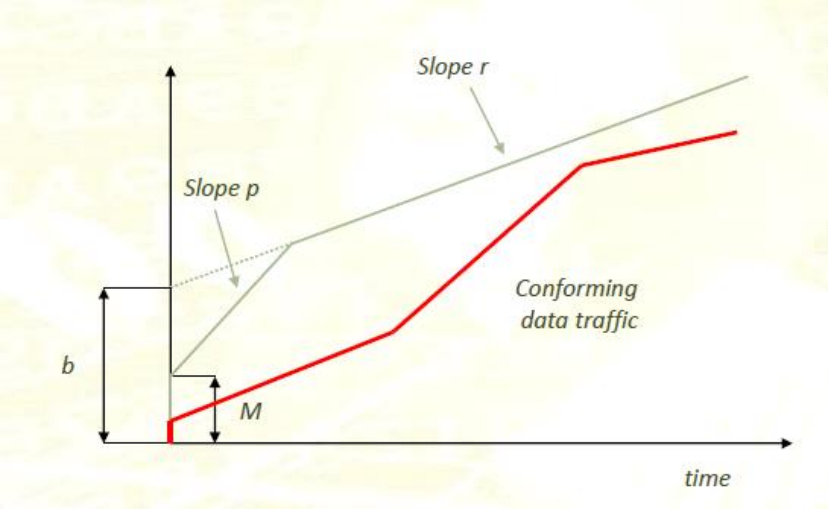
Integrated Services

Integrated Services (IntServ) refers to an architecture that aim at providing per-flow QoS guarantees.

We distinguish between two types of service classes: **guaranteed service** and **controlled-load service**.
In guaranteed service, a source that wants to send a flow of packets **specifies** the **traffic parameters** of this flow and the **desired service** in terms of delay bounds. The network will reserve the necessary resources in order to accommodate enough bandwidth for the specified traffic and in order to provide the required delay guarantees. This type of service is designed for non-adaptive real-time applications, i.e. those applications that can't adapt to variable delays and therefore need strict guarantees.
In controlled-load service, a source that wants to send a flow of packets through the network **specifies** the **traffic parameters** of this flow, while the desired service in terms of delay bounds is not specified. In this case, in fact, the network aims at reserving resources in order to accommodate enough bandwidth for the specified traffic but nothing is done to provide guarantees in terms of latency. In this sense, the type of service provided approximates that of a best effort network under unloaded conditions. This type of service is designed for real-time adaptive applications, i.e. applications that can adapt to variable network delays (for instance, by reducing their transmission rate).
For example: with guaranteed service, packets will be scheduled with an algorithm like WFQ which provides delay guarantees; with controlled-load service, packets will be scheduled with an algorithm that doesn't provide delay guarantees, although a fixed share of the bandwidth is reserved.

We talked about "specifying the traffic parameters": the description of the traffic parameters is called

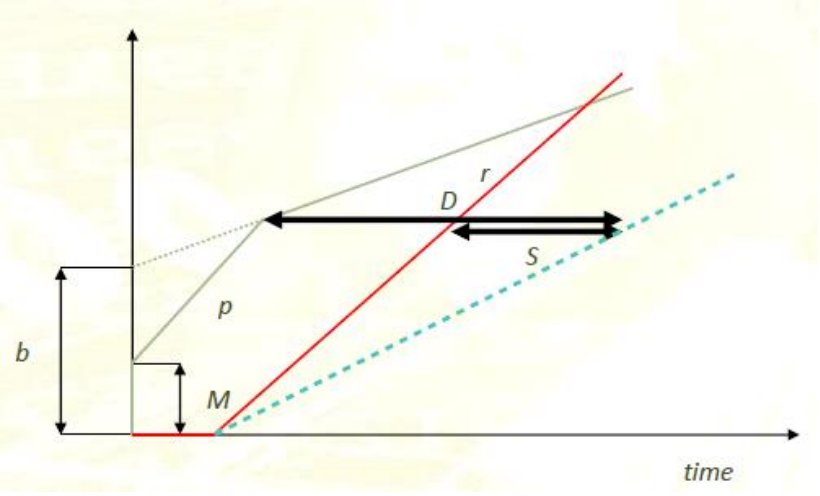
TSpec and it describes the characteristics of the traffic. This is done in a way similar to the (σ, ρ) model, i.e. by providing a deterministic bound, represented as a curve, to the traffic shape:



The following quantities are specified in the TSpec:

- The **token rate**: r
  - The long-term average rate
- The **peak rate**: p
  - The maximum short-term rate that might be sporadically produced by the source
- The **bucket depth**: b
  - Represents approximately the buffer size that should be allocated to the flow in order to absorb bursts of traffic avoiding losses
- The **maximum packet size**: M
- The **minimum policed unit**: m
  - Packets smaller than m are considered as if they had size m

We also talked about "specifying the desired service in terms of delay bounds": this information is described in the **RSpec**. In case of controlled-load service, no parameter is specified, while, in case of guaranteed service, two parameters are specified: namely the **guaranteed rate** R and the **slack** S. Given R, the maximum delay bound according to WFQ can be determined. The slack is an extra delay that can be tolerated by the application, given that the network provides the minimum requested rate R.



In the figure, the red line represents the service provided at a rate R and D is the maximum delay that the source can tolerate.
In practice, it is not necessary that the network provide a minimum guaranteed rate exactly equal to R;

in fact, any rate is fine, provided that it is between the slope of the light blue line (in the figure) and the slope of the red line.
This solution gives the network some flexibility in allocating bandwidth. For instance, the network may decide to allocate less bandwidth to a flow, in order to save resource and accommodate possible other flows. Or, if the network cannot allocate a rate equal to R, because, for example, the bandwidth available at a router is R' < R, it is possible to allocate a rate R' and accommodate the flow anyway, instead than rejecting the reservation request. This can be done as long as the limit imposed by the slack is not overcome.
TSpec and RSpec together form the **FlowSpec**, i.e. the overall specification of the traffic parameters and requirements for a given flow (in case of controlled-load service, RSpec is left unspecified, while it is properly specified in case of guaranteed service).

Another feature of IntServ is **admission control**: a control function executed before any new flow is admitted into the network, in order to verify that enough resources are available to accommodate the required service guarantees and that the other previously admitted flows are not affected by the new one.

Resource reservation is performed by means of the **Resource Reservation Protocol (RSVP)**:
- IP-based
- Extension Header code: 46
- Soft-state: reservations have to be periodically refreshed, otherwise they automatically expire
    - Provides adaptation to changes in the network topology
    - Provides robustness against failures
- Unicast and multicast support
    - Many of the first multimedia applications (which are those for which QoS mainly developed) where multicast
- When resources have to be reserved to transmit traffic from a sender to a receiver, the actual reservation request is initiated by the receiver host
    - However, resources must be reserved along the path that traffic would follow from the sender to the receiver, therefore the reservation request must traverse this same path (in reverse)
    - A preliminary message from the sender to the receiver is transmitted to set up the reverse path

RSVP evolves as follows:
- The sender generates an *RSVP Path* message and sends it along the path towards the receiver
    - Any node that receives this message allocates a structure called *Path State* where state information necessary to carry out the protocol will be stored
    - The RSVP Path message includes a *Previous Hop* (PHOP) field, containing the address of the previous node in the path that forwarded the message to the current node
        - The current node stores this information in the Path State, as it will be necessary to route the next message from the receiver along the reverse path up to the sender
    - The RSVP Path message also includes the sender TSpec and another data structure called ADSPEC, which is updated by every node in the path
        - Each node in the path updates ADSPEC by writing the maximum bandwidth and delay guarantees that it can provide to the flow
    - When a node has finished processing an RSVP Path message, it forwards the message to the next hop in the path towards the receiver, after having updated the PHOP information with its own address
    - Eventually, the message arrives at the receiver host
        - Based on the Sender TSpec and on the ADSPEC, the receiver decides the type of service to require (guaranteed or controlled-load) and the amount of resources to reserve (RSpec)
- The receiver host, after having processed the RSVP Path message, generates an *RSVP Resv*

message which is sent along the reverse path w.r.t. the one followed by the RSVP Path message, towards the sender host
    - The RSVP Resv message contains the FILTERSPEC, which is a collection of information needed to univocally identify a flow
        - Source IP address, destination IP address, source port, destination port in IPV4
        - Source IP address and Flow Label in IPv6
    - The message also includes the FLOWSPEC, containing information about the required service (guaranteed or controlled-load) and the amount of resources to reserve (RSpec)
    - RSVP Resv messages are routed hop-by-hop: each node along the path chooses the next hop using the PHOP information previously stored in the Path State
    - Before reserving the resources, each node performs policy control (i.e. it checks that the receiver is authorized to make this reservation) and admission control (discussed before)
        - If successful, resources are reserved and the message is forwarded to the next hop
    - Eventually, the message is delivered to the sender host
        - Reservation is concluded
        - Optionally, the sender can send a RSVP *ResvConf* message to the receiver as a confirmation of the success of the operation
    - Path and Resv messages are periodically retransmitted to refresh the state of the reservation
        - Reservations automatically expire after a timeout if they are not refreshed
        - This allows for graceful management of implicit session termination or route changes
        - Explicit termination is possible through *RSVP Path Tear* and *Resv Tear* messages

In case of errors an *RSVP PathErr* message is generated by the node where the error arose and sent back to the sender (if the error was caused by a Path message) or to the receiver (if the error was caused by a Resv message).
Non-RSVP routers forward RSVP messages like normal messages (they are transparent to the reservation process).

Problems with IntServ:

- Scalability: keeping per-flow state information is expensive, especially in the core network, where the number of flows can easily grow to tens of thousands
- Administration: if a flow traverses several autonomous systems (for instance, belonging to different companies), we ask each of them to reserve resources according to the RSVP protocol; however, companies want to maintain some independency and some control over their infrastructures and therefore this approach is not practical under this point of view
    - The alternative is to forget about end-to-end inter-domain QoS and focus only on intra-domain QoS, i.e. there are no end-to-end QoS guarantees, but each traversed autonomous system provides its own guarantees, which all in all result in a good overall quality

Differentiated Services

The alternative to IntServ is **Differentiated Services (DiffServ)**, i.e. instead than providing a different QoS to each flow, we provide differentiated QoS to different classes of traffic. We also consider a single network domain and not the whole Internet.

When a packet has to be processed by a network element, it first undergoes a process of **traffic classification**, i.e. it is identified as belonging to a specific traffic class, so that the appropriate service can be provided. Traffic classification is performed by analyzing a packet, following different possible approaches:

- Complex (multi-field) classification involves looking at some fields in the header:
    - IPv4: source/destination address, source/destination transport protocol port
    - IPv6: source address, Flow Label
- Deep Packet Inspection (DPI) also looks at the information carried in the payload of a packet

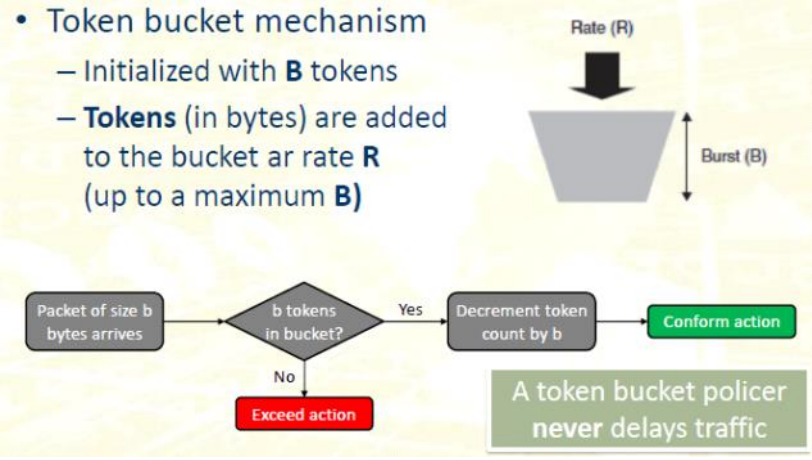
- Simple classification considers only fields in the packet header specifically designed for QoS classification
    - IPv4: Type of Service (ToS) field
    - IPv6: **Differentiated Service field**

In the following, we will consider the case in which the Differentiated Service field is used for classification. This field contains a code, named **Differentiated Service Code Point (DSCP)**, identifying the traffic class the packet belongs to (we will see more details about DSCP later). Typically, a packet is tagged with a DSCP as soon as it enters the network domain. This **packet marking** operation is performed by the edge routers by analyzing the packet and deciding the traffic class according to the network policies. Once a packet has been marked, it is easy for the core routers to identify its class and therefore provide the appropriate service.

Another feature applied to packets when they are entering the network is **traffic conditioning**. When a source of traffic asks the network for some QoS guarantees, it provides a description for the type of traffic it produces (burst/rate profile, like a (σ, ρ) model). But when the traffic generated by the source doesn't comply with the model provided, border routers reshape the traffic in order to make it compliant with the required profile. Traffic conditioning is the name of this process and it comes in two forms:

- **Policing**: the traffic profile is enforced to comply with the provided model by discarding non-compliant packets
- **Shaping**: the traffic profile is reshaped to comply with the provided model, by properly delaying incoming packets that do not meet the desired traffic profile (there is no discarding of non-compliant packets in this case)

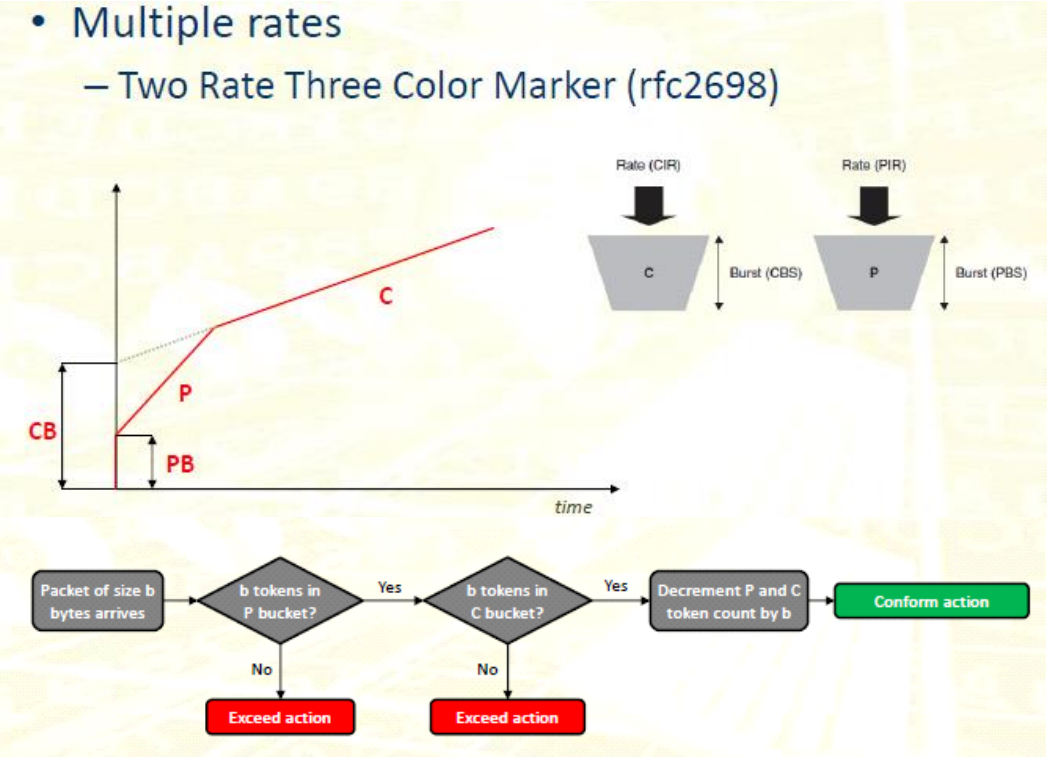
Policing can be implemented with the **token bucket mechanism**:



"Conform action" and "exceed action" may correspond to "forward" and "discard", respectively, but this is not the only possible approach; for instance, compliant packets might be marked as *in-contract* and non-compliant ones as *not-in-contract*:

- *In-contract* packets are always forwarded
- *Not-in-contract packets* can be forwarded anyway throughout the network (provided that there are enough resources available) but, if a congestion arises, these packets have a higher probability of being discarded

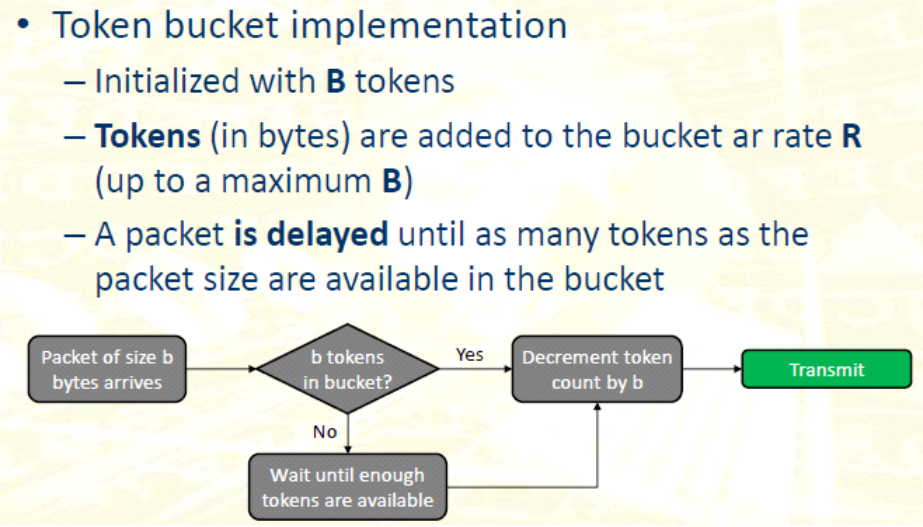
The token bucket mechanism can also be used to enforce traffic to meet more complex profiles, like with multiple rates. For example, a possible solution is the **two-rates three colors marker**:



A traffic profile like that in the figure corresponds to taking the minimum between the profile with rate C and the profile with rate P. In order to implement this kind of policer, we use two buckets, one filled up at rate C (up to a value CB) and the other filled up at rate P (up to a value PB). Different actions can be applied to conforming and exceeding packets; for instance, they can be marked with different *colors*:

- Packets conforming to both buckets are marked as *green*
- Packets conforming to one the first bucket only are marked as *yellow*
- Packets conforming to none of the buckets are marked as *red*
- Packets of different colors have different probabilities of being dropped
    - Very low for green packets, moderate for yellow packets and high for red ones

A token bucket can also be used to implement shaping:



Now, let's get back to the Differentiated Service IPv6 header field. We mentioned that it contains a code, named **Differentiated Service Code Point (DSCP)**, which identifies a traffic class. A DSCP is an 8-bit

code which is associated to a **Per Hop Behavior (PHB)**, i.e. a type of service that routers will reserve to packets marked with a given code. Mapping between DCSP and PHB can be arbitrarily chosen for each autonomous network, although a specific mapping is recommended. The mapping is many to one: many DSCP can be assigned to the same PHB, but packets marked with a given DSCP cannot receive different PHBs at a given router or a different routers in the network.
Note: guaranteeing that the same behavior be reserved to packets with the same DSCP also guarantees that packets of the same class are not reordered in-network.
The recommended DSCP assignment is the following (only six of the eight bits are used):

Codepoint	DSCP		
Default/CS0	000000		
EF PHB	101110		
CS1	001000		
CS2	010000		
CS3	011000		
CS4	100000		
CS5	101000		
CS6	110000		
CS7	111000		
AF PHB Group	Drop Precedence		
AF Class	Low	Medium	High
AF1x	001010	001100	001110
AF2x	010010	010100	010110
AF3x	011010	011100	011110
AF4x	100010	100100	100110

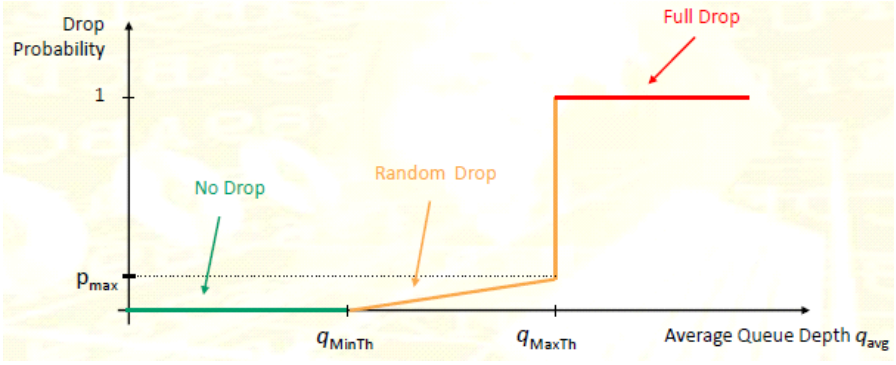
- **Default** PHB: used for traffic not explicitly mapped to other PHBs
  - A minimum capacity could be reserved to this traffic
- **Expedited Forwarding (EF**) PHB: requires a minimum guaranteed rate, independent of the load of any non-EF traffic
  - Implementation:
    - Packet scheduling: WFQ or (more typically) strict priority (in the latter case, a policer is necessary to avoid starving the other traffic)
    - Traffic conditioning: policing (discarding out-of-profile traffic is needed to ensure low delay)
  - Designed for non-adaptive applications with stringent requirements
- **Class Selector (CS)** PHBs: provide backward compatibility with previous IP QoS mechanisms
  - Packets with higher precedence (higher CS codepoint) have a lower probability of being dropped
- **Assured Forwarding (AF)** PHBs: provide a two-dimensional differentiation of traffic service
  - One dimension is represented by the *AF Class* (identified by the first three bits of the code): packets with different class have different capacity assurance
  - The other dimension is represented by the *Drop Precedence* (identified by the last three bits of the code): packets can have low, medium or high drop precedence
    - The drop precedence is decided by a policer: in-profile traffic can be assigned a low drop precedence, out-of-profile traffic will be assigned a medium drop precedence, highly out-of-profile traffic will be assigned a high drop precedence
  - This two-dimensional classification provides enhanced flexibility
  - Designed for adaptive applications
  - Implementation:
    - Packet scheduling: WFQ
    - Traffic conditioning: shaping

We talked about **dropping** packets with different drop probabilities. In the following, we see how to do

this more in detail.
Traditionally, dropping occurs at routers when a new packet arrives but there is no space in the internal queues to store it. This strategy is called **tail drop**. The queue size is limited not really because there are not enough resources to accommodate more packets, but rather because longer queues imply longer delays due to queueing.
In order to provide different treatment to different traffic classes, we could extend the previous strategy by considering two limits for the queue depth, say q1 and q2, with q1 < q2. When a low-priority packet arrives, it is discarded if the queue size is > q1, while when a high-priority packet arrives, the threshold q2 is applied. This approach is called **weighted tail drop**.
When dropping occurs, the sender which suffered the loss typically reacts by slowing down its transmission rate (TCP congestion control), which should result, at least in principle, in an optimization of the application throughput w.r.t. the network conditions. However, this is not always true. In fact, let's assume that a queue is full and multiple packets coming from different sources arrive together at that queue at the same time. The result is that they are all discarded and all the sources slow down their transmission rates simultaneously: the network condition passes from a very high to a very low utilization. This phenomenon is called **global synchronization of congestion control windows**. In order to overcome this problem, we can adopt more sophisticated dropping techniques collectively called **active queue management**.
A technique of active queue management, specifically designed to enhance TCP congestion control algorithm, is **Random Early Detection (RED)**. In RED, we try to detect congestions before queue overflows occur. This is done by computing and periodically updating an exponential average of the queue depth; when we observe that this value increases too much over time, we detect that a congestion is going to arise. The exponential average is computed as:

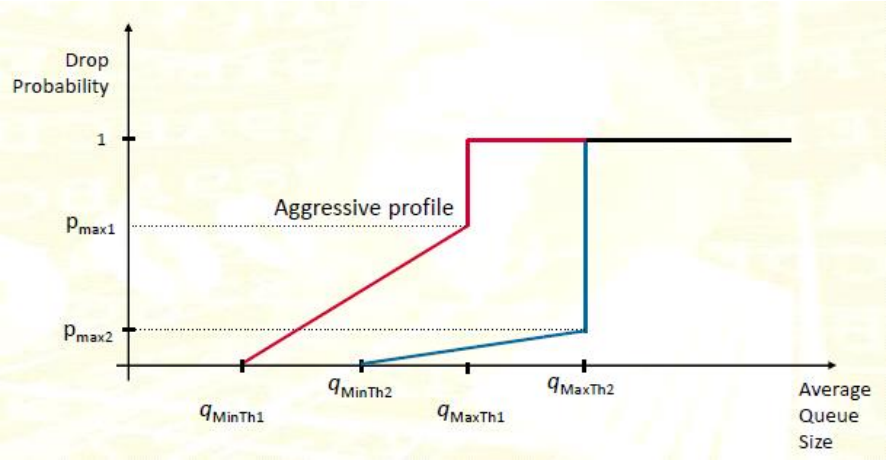
$$q_{avg} \leftarrow q_{avg}\left(1-\frac{1}{2^w}\right)+q_{curr}\frac{1}{2^w}$$

Note: the larger w the less we weight q_{curr}, which produces a smoother average over time; on the contrary, the larger w, the more we weight the current value than the past history, which produces a more varying average over time.
When RED detects that a congestion is going to occur, it reacts by discarding some packets at random (with a given probability), which provides a feedback to just some sources in order to slow down their rate. The behavior of RED in terms of drop probability as a function of the queue depth can be represented with the following graph:



- There is no drop if the queue size is below a minimum threshold
- There is a linearly increasing drop probability (up to a value p_{max}) if the queue size is between the minimum and a maximum threshold
- There is 100% drop probability when the queue size is above the maximum threshold

We can also devise a variant of RED which allow to provide different behaviors to different traffic classes, like we did for tail drop with weighted tail drop. In **Weighted RED**, we can imagine to have different profiles of the drop probability graph as a function of the queue depth and more aggressive profiles are used for packets with a higher drop precedence.



We conclude this section by describing the typical organization of a network providing DiffServ QoS (typical DS deployments). There are three types of network elements:

- Access routers: routers through which the traffic enters the network; access routers apply the edge policies, i.e. they perform traffic policing, assign drop precedence tags to packets and discard packets in case of congestion, or they perform reshaping, in order to enforce incoming traffic to meet a specific traffic profile
- Distribution routers: intermediate routers that perform traffic classification and assign code points to packets
- Core routers: apply core policies, i.e. they provide the appropriate service to packets depending on the class they belong to

Core Network Technologies

Introduction

In this section we are going to analyze some advanced technologies and protocols which find application in the core networks, enabling a higher scalability and flexibility than traditional technologies.
We want to find an improved routing solution to achieve the following goals:

- Scalability: traditional routing is poorly scalable
  - If a new destination becomes available in the network, a new entry has to be added to the routing table of each node
  - If a new external network becomes reachable, also in this case we need to add an entry to the routing table of each node: nodes need to know which egress router they have to transmit to (which one is the best one) in order to reach the new destination
  - Routing tables easily grow large
- Flexibility: traditional routing only allows to find least-cost paths towards a destination
  - When multiple paths towards a destination are available, a network administrator may want to use the least cost path for high priority traffic and another path for low priority traffic or as a backup path
  - In this light, we might want the network to provide more flexibility in the choice of routing paths both for administrative reasons and for optimizing the network utilization

We are going to address the following topics:

- First, we present the **Multi-Protocol Label Switching (MPLS)** and we see how this protocol can be used to satisfy the requirements of scalability and flexibility that we have mentioned so far
- Then, we discuss about **traffic engineering**, i.e. devising different routing paths (not necessarily the shortest path) to a given destination for different classes of traffic, and we see more in details how MPLS, in conjunction with other protocols (LDP, OSPF-TE, RSVP-TE), can be used to provide such a flexibility
- Follows a discussion on the **Border Gateway Protocol (BGP)**, a protocol which allows to implement inter-autonomous-systems routing
- Finally, we discuss **MPLS/BGP IP VPNs**, which is a solution to implement VPN services, provided by the core network, in a scalable way, without recurring to tunneling techniques like in IPSec and removing the burden of complex border router configuration on the customer side
  - We will see how both MPLS and BGP come into play in the design of this solution

MPLS

Conventional IP routing is destination-based, i.e. when a router has to process a packet, it looks up the destination address in the routing tables, in order to obtain the next hop that packet has to be forwarded to. This approach has some drawbacks:

- IP addresses are long and the look up process is expensive
- As we have already pointed out, routing tables might become large and keeping them updated is not a scalable task
- Routing tables provide only the least-cost path towards a destination (not enough flexibility)

In order to achieve a better flexibility, instead than considering just the destination address of a packet, we could devise a more general routing scheme that consider also other pieces of information (for instance, the source address and the Flow Label, or the DSCP). Therefore, we introduce the concept of **Forwarding Equivalent Class (FEC)**. A FEC is a class of packets that are routed in the same way. The FEC of a packet is determined by different pieces of information contained in the packet, as we mentioned above (and not only from the destination address).

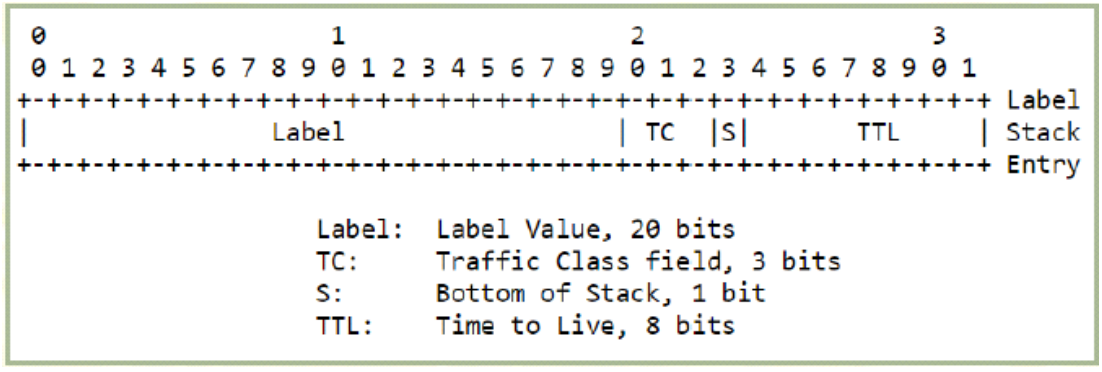
The evolution of a routing algorithm can be seen in two phases:

- Partitioning: given a packet, the router determines its FEC
- Mapping: given the FEC of a packet, the router maps it to a next hop by means of its routing tables

The next feature that we can introduce is the decoupling between these two phases: since the partitioning phase is complex (because potentially it involves the inspection of several fields in the packet), we associate this task only to the ingress routers (those from which the traffic is injected in the network), which, after the classification, apply a label to the packet to identify its FEC. Core routers only perform the mapping phase, since they already know the FEC from the label.
So, basically, when a node has to route a packet, it uses a sort of modified routing table which maps the packet label to the corresponding action to be taken. We will see later how this tables are built.
This approach provides good improvements, since we avoid to repeat the partitioning phase at every intermediate hop. The advantages relies on the fact that a class label is very short, and therefore it is much easier to inspect than other fields in the packet (for instance, the destination address, which might be quite long). This solution also reduces the scalability problems that we have mentioned so far: when a new external network becomes reachable, we only need to update the ingress routers, so that packets addressed to the new destination are mapped to a FEC associated to a path leading to the appropriate egress router. Internal routers don't need any update, because they are already able to map that FEC to the next hop, and that same FEC is already associated to the path towards a given egress router.
In order to fully implement the solution that we have presented so far, we need a way to let all the routers agree on which labels are associated to which FECs. This task could be quite hard, but fortunately we don't really need this. In fact, we can permit each router to freely choose its own label to represent a given FEC. In addition, routers exchange information just with their neighbors about the representation they use. At this point, when a router A has to transmit a packet to a neighbor B, it just changes the label associated to the packet right before the transmission, in order to comply with the representation used by B.
It is possible to see that this technique of using labels to define how a packet should be treated by routers actually defines how the packet will be routed and the path that it will follow. Therefore, we talk about *Label Switched Routing* (LSR) and *Label Switched Paths* (LSPs).

Let's see more in detail how the **Multi-Protocol Label Switching (MPLS)** implements these ideas:

- Header placed between layer 2 and layer 3
- Header format:



- Multiple headers can be placed one on top of the other, in a stack-like fashion
  - This is done for tunneling purposes, which find application in some problems like fast traffic re-routing in case of failures or VPNs implementation, that will be discussed later
  - The S bit is used to specify whether there is another MPLS header stacked below the current one (S = 0), or if the current header is the bottom of the stack (S = 1)
- The TC bit, a.k.a. EXP bit (experimental bit, it has been later renamed TC because it is used to enable the integration of QoS functionalities within MPLS), is used to specify up to eight different traffic classes
  - Eight among the DSCPs are mapped to a corresponding MPLS TC code, the mapping is

chosen by the network administrator
    - If it is necessary to map more than eight traffic classes, the mapping is performed between the DSCP and the whole label plus the EXP bits
    - The first solution is called E-LSP (EXP-LSP), while the second is called L-LSP (Label-LSP)

MPLS provides three types of **actions** to deal with incoming packets, depending on their label:

- PUSH action: a new MPLS header with a new label is placed on top of the MPLS header stack (or on top of the L3 header, if no other MPLS headers were pushed before)
- POP action: the MPLS header on top of the stack is removed
- SWAP action: the label of the top MPLS header of an incoming packet is replaced with another label; this action is used before forwarding, in order to change the current label with the one used by the next hop to represent that FEC

A typical combination of actions employed in an MPLS domain is the SWAP&PUSH: when a packet arrives, first the label of the top MPLS header is swapped and then another header is PUSHED (typically done for tunneling).

The **forwarding scheme** employed in MPLS can be modeled as follows:

- There is a first table, named *Incoming Label Map* (ILM), which maps the label of an incoming packet to an entry of another table, called *Next Hop Label Forwarding Entry* (NHLFE)
- The NHLFE table contains information on the action that has to be performed on a given packet: an entry contains the type of action (PUSH, POP, SWAP), the next hop which the packet should be routed to and the label that should be applied to the packet (with SWAP or PUSH actions) before it is transmitted
- A third table, the FEC-to-NHLFE map, is used by ingress routers through which unlabeled traffic enters the MPLS domain:
    - First, the router determines the FEC associated to a packet (depending on the network policies) and then the packets has to be labeled with the appropriate label, depending on its FEC
    - In order to do this, given the packet FEC, the FEC-to-NHLFE map is used to retrieve an NHLFE that will correspond to a PUSH action and will contain the label to be applied to the packet (and, of course, the next hop which this packet should be forwarded to)

This is a general scheme, but real implementations could use data structures organized in different ways or with different name. In general, we identify them with the generic name of *Label Forwarding Information Base*.

A little optimization called **Penultimate Hop Popping (PHP)** is typically implemented. Consider what happens when a packet reaches the last hop of an LSP: the last router looks up its internal tables and finds out that it has to POP the label. After the label is popped, the router uses the IP routing table to route the packet as it would normally do. So, we observe two look-ups. The optimization allows to perform one look up only. This is done as follows: normally, the penultimate hop in the LSP performs a SWAP and forwards the packet to the last router; with PHP, we make the penultimate router perform a POP, instead than a SWAP, before forwarding the packet to the last router. Therefore, the last router receives an already unlabeled packet which can be processed directly with a single lookup in the IP routing tables.

Now, let's discuss about how labels are managed by routers.

Different scopes may be assigned to labels:

- Per interface scope: if two packets with the same label come from different interfaces, they are interpreted differently
- Per LSR (Label Switched Router) scope: two packets with the same label are interpreted in the same way at the same router, independently of the interface they come from

Given an LSP, labels are **downstream-assigned**, i.e. a given router Rx decides the label to use for a given FEC and informs the previous hop Ry in the LSP about the decision. The latter will store an NHLFE where it is written that a packet corresponding to that FEC must be forwarded to Rx and the incoming label has to be swapped to the value received by Rx. In principle, we could have either Rx take the decision and inform Ry or Ry take the decision and inform Rx, but we opt for the first approach. In this sense, the

downstream router decides the label assignment.
Label distribution occurs by means of the **Label Distribution Protocol (LDP)**, that can take place following two possible approaches:

- *Downstream on-demand*: a router explicitly asks the next hop what label assignment it has decided to use for a given FEC and receives a reply
- *Unsolicited downstream*: the downstream router freely broadcasts the label assignment to its neighbors (LDP peers), without explicit request from them

A router R might be connected to multiple downstream routers, each one providing a different label-FEC binding. In this case, router R needs a mechanism to select only one of those bindings, and the corresponding router as next hop. For instance, it is possible to use the binding coming from the router corresponding to the shortest path toward the desired destination, information that is provided by the L3 routing tables, built using traditional routing algorithms. Specific algorithms for traffic engineering (described in the corresponding section later on) can be used to obtain the appropriate paths for FECs having complex requirements, other than just shortest paths. So, only the binding coming from one downstream router is used; however, there are different ways, **called label retention modes**, to treat the information coming from the other routers:

- Liberal label retention mode: unused information is stored anyway internally, so that it can be possibly used in the future (for instance, if the selected path fails, a backup path can be found)
- Conservative label retention mode: unused information is discarded

Finally, there are two possible approaches to setup an LSP, called **LSP setup control modes**:

- *Ordered control*:
    - With unsolicited downstream label distribution
        - Path setup starts from the egress router that will constitute the last hop of the LSP
        - This router starts broadcasting binding messages to its neighbors for a FEC, following an unsolicited downstream approach
        - Any other router, upon receiving binding messages from downstream routers, uses the L3 routing table to choose which binding to use and which router to select as next hop and generates the NHLFE; then, in turn, it starts broadcasting its own binding to its neighbors
        - The process goes on up to the ingress router from which the traffic will be injected and the path is formed
    - With downstream on-demand label distribution
        - Path setup starts from the ingress router from which the traffic is injected, as soon as it needs to discover a path for a given FEC
        - Using the L3 routing tables, it determines the next hop for that FEC; then it asks the next hop for the binding it wants to use for this FEC, following a downstream on-demand approach
        - Upon receiving the request for the binding information from the upstream router, the downstream router, in turn, needs to find a path for said FEC; therefore, in turn, it determines the next hop from its L3 routing tables and asks it for the binding it wants to use for this FEC
        - The process is repeated for all the routers until the egress router which will constitute the final hop in the LSP; this router knows how to reach the desired destination, selects a label for the FEC and communicates the binding to the upstream router that had requested it right before
        - At this point the latter router, in turn, decides a binding and communicates it to the upstream router and so on, up to the initial router: the path is finally built
- *Independent control*:
    - In this case there is the unsolicited downstream label distribution case only
        - Each router creates and distributes the bindings independently
        - At the beginning, when there is no binding yet, a FEC is associated to a POP action and the next hop address is the loopback address; in this way, the MPLS label of a packet is removed and the packet is just passed to level 3

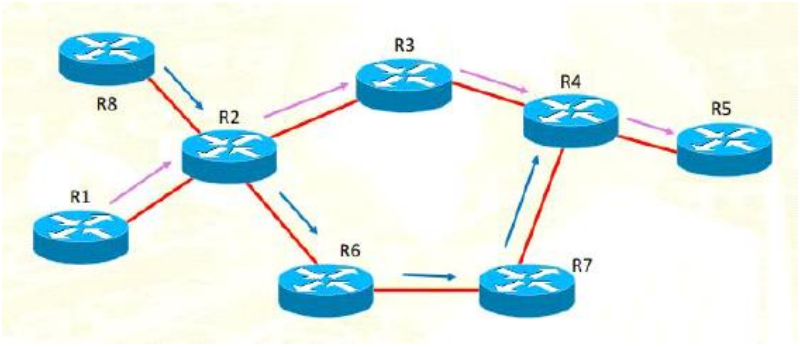
    - When the downstream router decides which binding to use and communicates it to the upstream router, the latter just changes the POP action with a SWAP to the label decided by the downstream router and set the downstream router as next hop

A final note on how a router gets to know that it is the penultimate hop so that it can perform PHP: the last router explicitly tells it so; this is done by imposing the last router in an LSP to use a reserved label for a binding. This is label number 3, a.k.a. **null label**: whenever an upstream router receives a binding to label 3, it means that next router is the last one in the LSP.

Traffic Engineering

In traditional networking, traffic always follows routes determined by shortest path routing algorithms. However, sometimes we might want to control the paths taken by traffic in different ways. For instance, the following situations may arise:

- There might be different paths towards the same destination: we don't want to use just the shortest path to route the traffic, but we want to use all the available paths in order to maximize the overall utilization of the network resources



    - For instance, in the figure there is a path passing by R3 and another passing by R6-R7
    - The shortest path (in terms of number of hops) is the one passing by R3
    - Let's assume that both paths offer 100Mb/s available capacity
    - Let's assume that there are two flows, both transmitting at 80Mb/s
    - We would like to accommodate both flows, but we can't send them both through the path passing by R3, because there is not enough bandwidth
    - Instead, we could send one flow along the path passing by R3 and the other along the path passing by R6, thus maximizing the utilization of the network resources
    - We would like a routing algorithm to consider not only the path cost, but also the available capacity on the links, so that, if a path cannot provide enough bandwidth, another one is chosen
    - A traditional routing algorithm, instead, will just route both flows through the path passing by R3, which will be congested, while the other one will remain completely unutilized
- We might need to find a path for a traffic class not passing for a specific link, which is forbidden due to administrative reasons
- We might want to devise different paths for different classes of traffic (for instance, reserving a path with higher throughput for high priority traffic and another path with low throughput for low priority traffic)

This is what is called **traffic engineering**.

MPLS can be effectively used to enforce traffic to follow a specific path, but MPLS paths are derived from the paths computed by the L3 routing protocol; therefore, only least cost paths can result from this approach. In order to enable traffic engineering we need more complex routing protocols to be used in conjunction with MPLS. These routing protocols should not only minimize a cost metric, but also consider constraints like those that we have mentioned above. Such routing schemes belong to the class of the so-called **constraint-based routing**.

The types of constraints that can be imposed are:

- Performance constraints: finding paths with a minimum available capacity guaranteed on each link
- Administrative constraints: finding paths not traversing forbidden links

Therefore, while in traditional routing links are characterized only by their cost, in constraint-based routing they will be characterized by a set of **link attributes**.
We also need a path computation algorithm considering the constraints. For example, **Constrained Shortest Path First (CSPF)** is a variation of SPF that also considers constraints when computing paths. This algorithms follows a link-state approach: routers exchange information about links (including the link attributes) with each other and then each one computes the optimal paths, that comply with the constraints, locally. Information about links is stored internally by each router in a **Traffic Engineering Database (TED)**. A typical strategy for constrained path computation is to build the network graph and then "prune" it, by removing those inks that do not match the constraints; finally, paths are computed by applying the Dijkstra's algorithm to the pruned graph.
A specific implementation of CSPF is **OSPF-TE** (OSPF for Traffic Engineering, a variation of OSPF that also includes link attributes, other than link cost, in the advertisement messages), which defines the following link attributes:

- Maximum bandwidth of the link, i.e. the link capacity
- Maximum reservable bandwidth: maximum amount of bandwidth that can be reserved on the link
    - Typically configured to be <= maximum bandwidth, unless the network administrator wants the link to be oversubscribed
- Unreserved bandwidth: amount of bandwidth still available on the link
- Administrative group (or color): up to 32 groups can be defined
    - A link can be member of multiple groups/can have multiple colors
    - Colors can be used to specify administrative constraints, e.g. "find a path not traversing red links"
    - The meaning of colors is snot specified by the standard: the network administrator can freely decide the meaning associated a color
    - The administrative group link attribute is represented as a set of 32 flags, each one associated to a group: if flag i is set, it means that the link belongs to group i

Link information dissemination can be periodic or event-based. In OSPF it is event-based, i.e. whenever a change occurs in the state of a link, new information is advertised by exchanging control messages. This is good under the assumption that changes happen unfrequently. But if changes become frequent, we risk to overload the network with control messages; in this case, periodic advertising is more scalable, even though routers may not always have 100% accurate information. In OSPF-TE there are some link attributes that are static (maximum bandwidth, maximum reservable bandwidth, administrative group), while others are dynamic (unreserved bandwidth). In particular, unreserved bandwidth changes very frequently, because it varies every time a new flow reserves some bandwidth or a flow releases its reservation. Therefore, in OSPF-TE, changes in the state of a link happen very frequently and the event-based information dissemination approach is not scalable. In practice, a hybrid approach (periodic + event-based) is used: information is exchanged event-based, but if changes become too frequent, information cannot be exchanged with a period smaller than a given threshold.

At this points we have the tools to compute paths (OSPF-TE) and enforce traffic follow a specific path (MPLS). But we may also want to allow new flows to perform resource reservation while setting up a path at the same time. For this purpose, an extension of RSVP, called **RSVP-TE** (RSVP for Traffic Engineering), exists. The ingress router of the LSP that we want to build is called *head-end*. Like in RSVP, the head-end initiates the protocol instance by sending out a *Path* message, and then the other end-point will reply with a *Resv* message. The protocol changes in that the two types of messages carry new pieces of information, called *Objects*:

- The Path message carries the *Label Request Object (LRO), Explicit Route Object (ERO), Sender TSpec* and *Record Route Object (RRO)*
- The Resv message carries the *Label Object* and *Record Route Object (RRO)*

The LRO indicates that we want to set up a path and to perform a label binding at each hop.

The Label Object contains the label selected by a router for this LSP which has to be communicated to the upstream router in the path.
The ERO contains a list of hops that will form the path to be set up. This path can be computed directly by the head-end; in this way, it is possible to avoid path computation at each single hop. There is a bit in the ERO that indicates whether the hops contained in the object are *strict* or *loose*. In case of strict hops, subsequent nodes in the ERO are directly connected, and therefore the path will be exactly the one described in the ERO. In case of loose hops, instead, the head-end provides a list of nodes such that subsequent hops might not be directly connected. In the latter case, each node in the path can compute the path to reach the next node in the list autonomously, using CSPF. We say that the list of nodes carried in the ERO is a list of abstract nodes, in the sense that an element of the list can represent one or more physical routers.
The RRO contains the state of the current operation.
Admission control is required at each hop: in principle, when a path is chosen with CSPF, it is guaranteed that there are enough resources along the path, but in practice this is not true, because the information contained in the TED could not be up to date when path computation is performed.
LSPs provide up to eight levels of **priority** (numbered from 0 to 7) in order to solve situations of resource contention. Actually, two distinct priorities are associated to LSPs: *setup priority* (for paths that are currently in construction) and *hold priority* (for paths that are already active). If two flows want to reserve resources simultaneously, but there are not enough resources for both, the setup priority is used to decide which reservation will be accepted between the two. Another possible situation is when we want to set up a path for an important flow, but there are not enough resources. In this case, we might want to tear down the reservation of another, less important flow. This is what happens when the new flow has setup priority higher than the hold priority of the other flow. We use two distinct priority to have more flexibility: we can associate a high hold priority to a flow, so that we are sure that, once the reservation is completed, the path will remain stable, but a lower setup priority, because, for instance, we want to avoid tearing down other paths in case of resource contention. Note that the hold priority of an LSP should always be larger than the setup priority, otherwise instability could occur:

- Consider two LSPs with hold priority 0 and setup priority 7
- LSP1 is initially created, then we try to make a reservation for LSP2 --> LSP1 is tore down and LSP2 is created
- At this point, the flow that was using LSP1 tries to perform another reservation --> this time, LSP2 is tore down and LSP1 is recreated
- The process goes on cyclically

Since the network conditions change dynamically, also the optimal path for a flow might change over time. Therefore, it is necessary to periodically re-run CSPF in order to update the LSPs and adapt to the new network conditions. This process is called **re-optimization**. Once we have computed the new path, we could simply tear down the old one and then build the new one from scratch; however, this is not a good approach, because we would make the network unavailable in the meantime. In practice, a *make-before-break* approach is used: first I create the new LSP and then I tear down the old one. Problem: we need double capacity to do this, because we have to accommodate twice the flows at some point in time. The solution is called **Shared Explicit (SE) reservation**. The SE is a particular type of reservation during which routers are explicitly informed that the flow for which we are making the reservation coincides with a preexisting flow. In this way, routers that have already allocated capacity for that flow will not allocate twice the capacity, but they will just share the capacity between the two LSPs, until the new LSP is completely setup and the old one is tore down.

Finally, let's discuss an example application of MPLS for traffic engineering: **fast re-routing**. In case of network failures, what happens in traditional networks is that the network element that detects the failures informs all the other router, and then they will all compute a new path towards some destination. The problem is that this approach takes too much time. Fast re-routing allows to find an alternative temporary path to cope with a network failure, until a new official path is computed. This is done by having a pre-computed backup LSP for each link: when a failure is detected, the backup LSP can be immediately used, leveraging MPLS tunneling, to bypass the broken link. Note that a backup LSP is

dedicated to the protection of a specific link and not of a specific service LSP: we have a backup LSP for each link, not a backup LSP for each regular LSP.

BGP

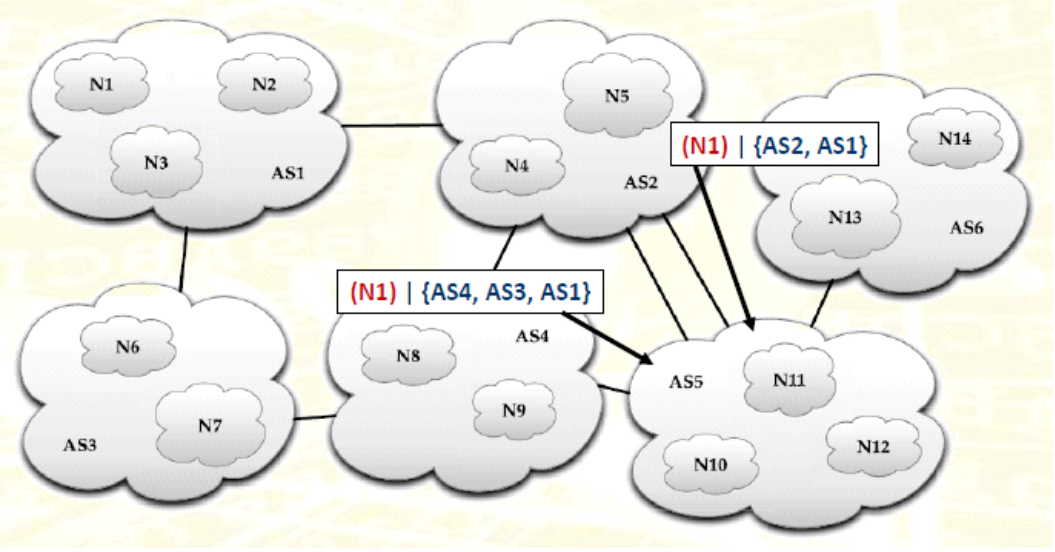
Nowadays, Internet is made up of different Autonomous Systems (ASs) interconnected with each other. The **Border Gateway Protocol (BGP)** is a solution to allow interoperation among ASs. Since Internet infrastructure is huge, it is not possible to use a unique global routing algorithm to interconnect all nodes; instead, a partitioning into ASs is necessary. BGP allow to implement inter-AS routing.
The routing solution used in BGP is neither link state nor distance vector:

- Link state: constructing the whole Internet topology at each node and computing shortest paths is not scalable
- Distance vector: there is a high possibility that loops arise in the transient phase, which are dangerous in such a big topology

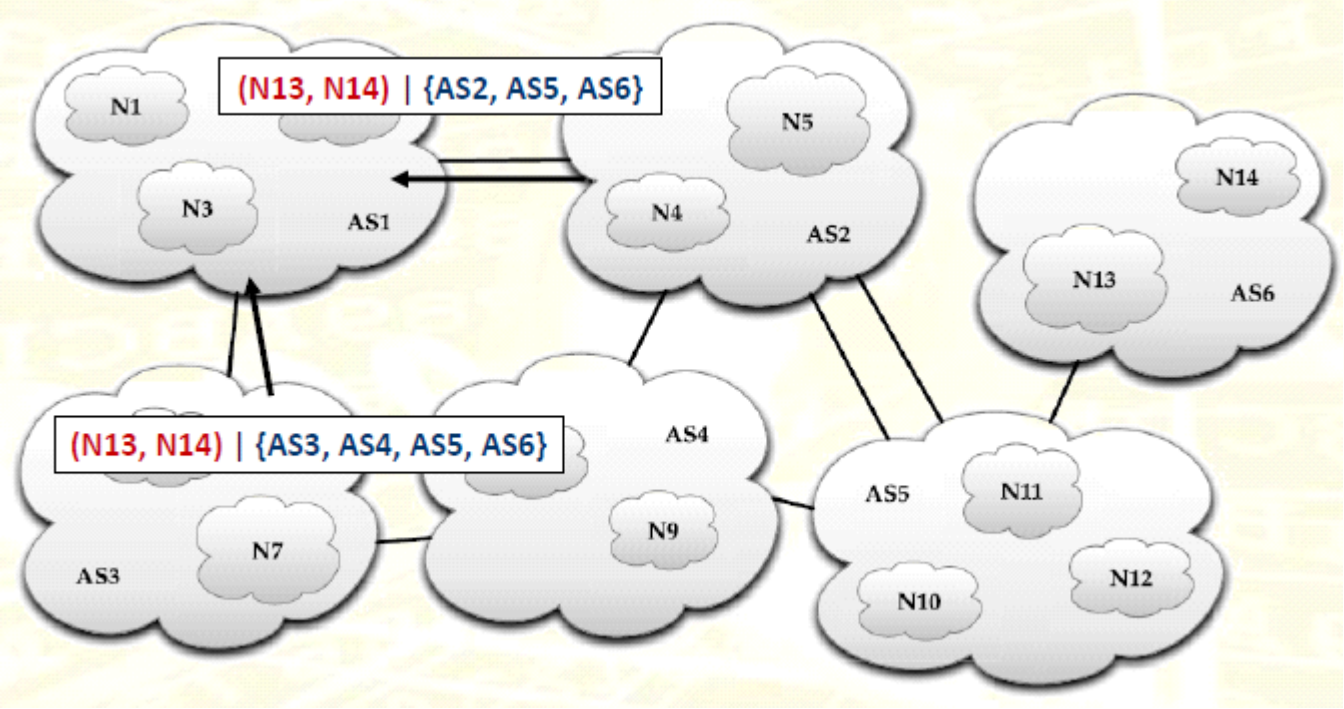
The solution employed is called **path vector**. It is similar to distance vector, but information dissemination is different

- For each known destination network N, with distance vector information about the best known path to reach N is exchanged by each single router
- With path vector, instead, information about the best known path to reach N is exchanged only by ASs

In this case, ASs are seen as fundamental units for the topology, and not routers. Each AS is a single network unit, even though, in practice, one or more Border Routers (BRs) are hidden behind this representation. Each AS exchanges with neighbor ASs messages containing a known destination network and the list of ASs that compose the best known path towards the destination (this is an important difference w.r.t. distance vector, because this feature allows to prevent cycles: if ASx receives a path vector in which ASx itself is included, the message has to be discarded, otherwise a cycle will arise if ASx uses that path).



The last identifier in the AS list is the home AS, i.e. the AS the advertised networks belongs to.
The cost associated to each hop is always one. However, a simple trick to associate a higher cost to a path is to repeat an AS identifier multiple times in a path vector (path manipulation).
If an AS contains multiple destination networks, these can all be advertised in the same message, as a list of possible destinations.



These ideas are implemented in the **BGP-4** protocol. ASs do not talk directly, but BRs in an AS talk with BRs in an adjacent AS. BRs exchanging BGP information are called *BGP peers* or *BGP speakers*. BGP peers communicate over a TCP connection. Two BGP peers can belong to different ASs or to the same AS:

- **Exterior BGP (E-BGP)**: the two peers belong to different ASs
- **Interior BGP (I-BGP)**: the two peers belong to the same AS

I-BGP is needed because a single AS is endowed with different BRs which may be connected to different other ASs; therefore, a BR receives information from an AS and another BR receives information from another AS: the two BRs need to exchange the pieces of information they have collected. One I-BGP session is established between each pair of BGP-enabled BRs in a given AS.
Now, let's discuss about the structure of BGP messages (without delving too much into details). There are different types of messages, e.g. for opening a connection, for keepalive, etc., but we are specifically interested in **UPDATE** messages, which are those periodically exchanged between BGP peers to communicate information about known paths towards some destinations. A BGP UPDATE message contains a list of **Path Attributes**. Path attributes encode the information representing the path to reach a given destination. The attributes are the following:

- ORIGIN: identifier of the home AS
- AS_PATH: list of identifiers of the ASs that represent the path towards the destination
- NEXT_HOP: this is the actual router address that has to be put in the routing tables
- …

Another piece of information included in the UPDATE message is the **Network Layer Reachability Information** containing the list of destination networks reachable through the path described in the path attribute.

VPNs

Traditional technologies for building VPNs make use of tunneling over Internet, e.g. IPSec

- Pros: the fundamental goals of VPNs are achieved (connectivity between networks of different venues of the same company, private addressing, privacy of traffic)
- Cons:
  - VPN implementation requires configuration at **Customer Edge (CE)** routers
  - If the customer wants to add a new venue to the infrastructure, all the CE routers of all the venues must be reconfigured in order to create the virtual interconnection to the new

venue --> poor scalability

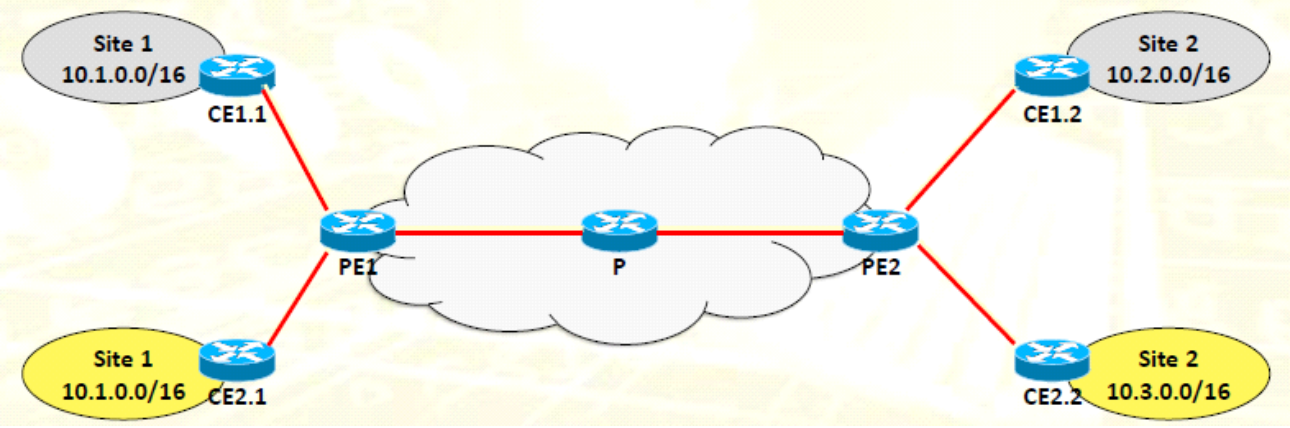
In this section we address new technologies to build VPNs without the problems mentioned above. First of all, we introduce a feature that allows to remove the burden of complex configuration at CE routers. In order to do this, instead than configuring a virtual private link between a CE and any other CE at a different venue of the company, we create just a private link between a CE and the nearest Point of Presence (PoP) of the provider network (peer-to-peer solution). A Point of Presence is a border router through which customers can attach to the backbone network, and it is also called **Provider Edge (PE)** router.

At this point, the burden of guaranteeing VPN service is on the provider. The goals to be achieved are:

- **Private addressing**: each customer uses its own addresses; possibly, different customers use overlapping addresses
- **Isolation of traffic**: traffic generated by customer A must not reach customer B's networks

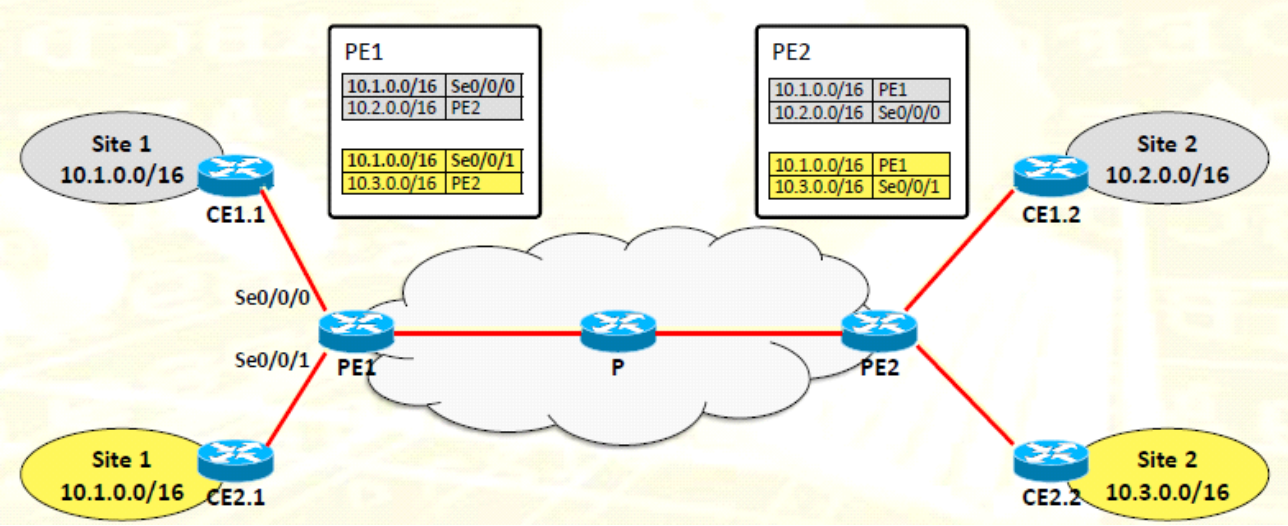
Solution: **BGM/MPLS IP VPNs**.

We will consider the example scenario depicted in the following figure:



There are two customers: the yellow one and the grey one. Given a packet addressed to 10.1.1.1, if it comes from CE1.2, then it must be delivered to CE1.1, while if it was generated from CE2.2, then is must be delivered to CE2.1.

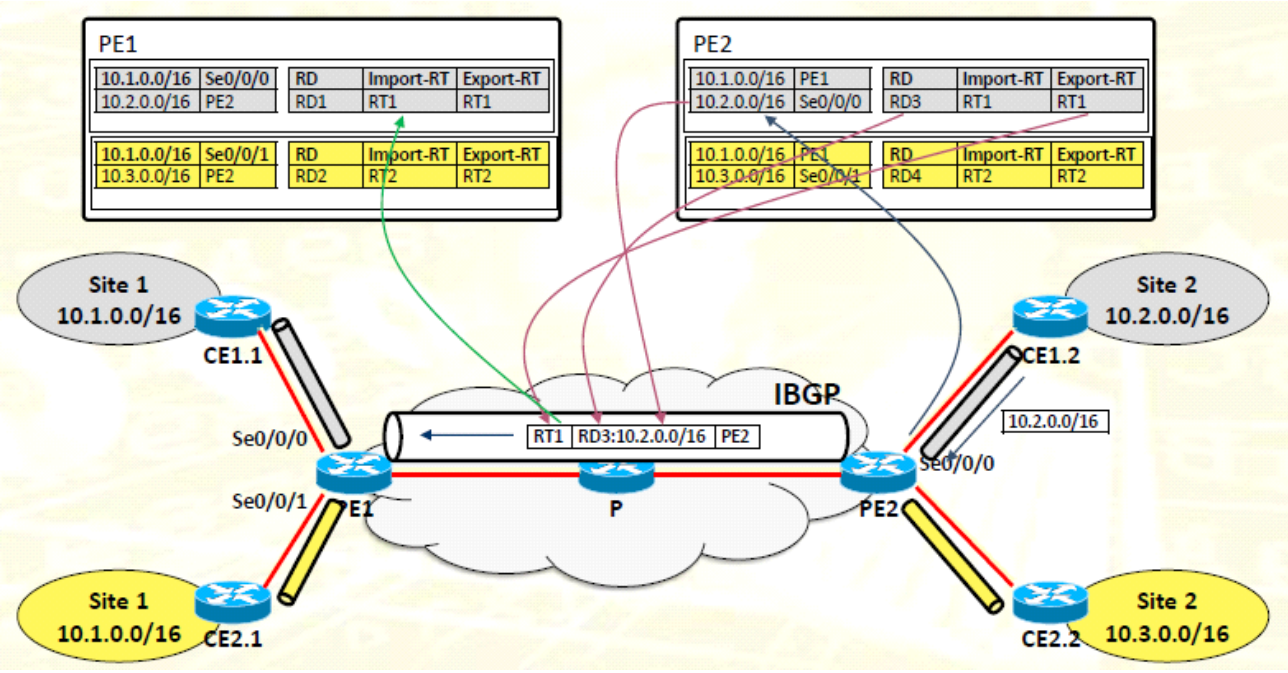
In order to achieve isolation of traffic, the solution is to use different routing tables for packets generated by different customers: they are called **VPN Routing and Forwarding (VRF) tables**. When a packet arrives at a router, it is possible to determine which customer generated the packet depending on the interface over which it was received (only a specific customer is attached to a given interface) and consequently use the corresponding VRF. VRFs are built only by PEs and not also by core routers (otherwise the algorithm would not scale well), while the problem of building a path between different PEs through the core routers is solved separately: in particular, the path is obtained by means of MPLS tunneling (we assume that an LSP exists between each pair of PEs). In the figure below it is possible to see the situation with grey and yellow customers and the corresponding grey and yellow VRFs.



PEs build VRFs with the routes initially advertised by CEs. Then, known routes are announced to other PEs by means of I-BGP (route distribution). Actually, an extended version of BGP is used, called **Multi-Protocol BGP (MP-BGP)**, which includes, among the other things, new attributes named **Route Targets (RTs)**. RTs are identifiers used to specify the target VRFs in which the PE that receives a given BGP message should insert the route contained in that message (constrained route distribution). BGP messages may contain multiple RTs, because we may want a route to be added to multiple VRFs at a PE (we will see later how this feature is used).
PEs maintain, for each VRF, a list of **Import-RTs** and **Export-RTs**:

- Import-RTs specify which routes we want to import in a given VRF: if the Import-RT set includes at least one of the RTs in the BGP message, the information contained in the BGP message is used to update the VRF
- Export-RTs is the list of RTs that must be included in a BGP message announcing routes from this VRF

RTs are configured on the PEs by the network administrator.
Consider, for example, the following figure:



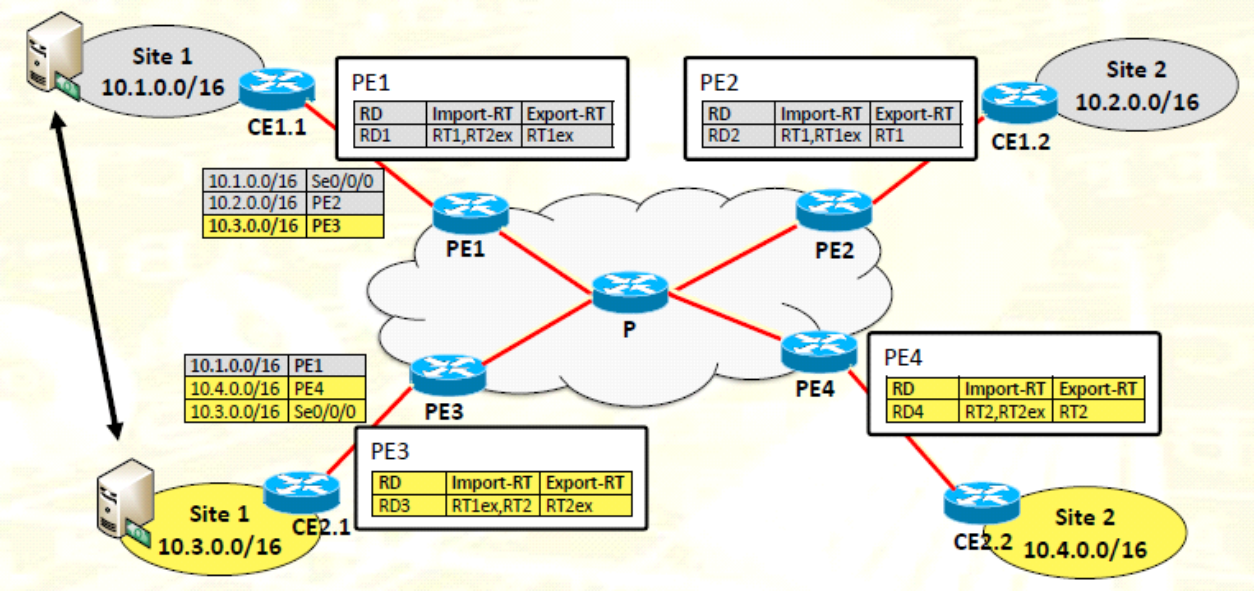
- CE1.2 announces destination 10.2.0.0/16 to PE2
- PE2 inserts the information on the grey table (because the announcements came from the grey

customer)

- Then, PE2 sends to PE1 a BGP announcement messages for the same destination, containing the Route Target RT1 (as specified by the Export-RT), which identifies the VRFs for the grey customer
- PE1, upon receiving this message, inserts the route in the grey table, because the Route Target in the message matches the Import-RT of that table

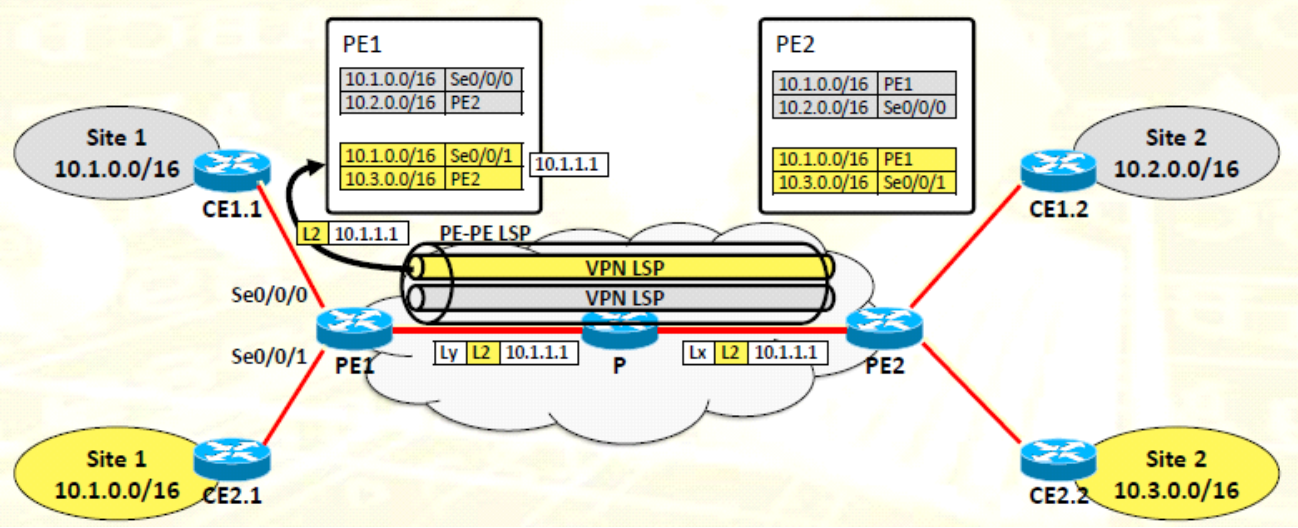
The **Route Distinguisher (RD)** field shown in the figure is used to avoid ambiguity when overlapping addresses are used by different customers. The problem is that if two messages announcing the same address are received by a PE, BGP considers the second as an update w.r.t. the former, regardless whether they refer to the same destination or to different destinations with overlapping address. In fact, BGP uses the announced address to distinguish messages related to different routes; therefore, messages containing the same address will be undistinguishable from each other. In the previous scenario, we can consider that PE2 receives two BGP messages from PE1, both announcing route 10.1.0.0/16, but one is related to the grey site 1 and therefore will contain RT1, while the other is related to the yellow site 1 and therefore will contain RT2. Let's assume that the message containing RT1 is received first and then the one containing RT2. When the message containing RT1 is received, the route is inserted into the grey VRF. When the message containing RT2 is received, it is interpreted as an update that informs PE2 that the route 10.1.0.0 should be inserted in the yellow VRF and removed from the grey one (because the message does not contain RT1), which is not what we want.
The RD is an 8-byte identifier that is prepended to an address before the latter is inserted in a BGP message. The MP-BGP extension allows to include in messages not only IPv4 addresses, but different types of address families. In particular, the RD concatenated to the IPv4 address of a network form a new address family called **VPN-IP**, which is supported by MP-BGP. RDs are configured on the PEs by the network administrator, and they have to be chosen so that the resulting VPN-IP addresses are globally unique. For this purpose, we might assign a different RD for each customer, but this implies that all the PEs must agree on the same RD for a given customer. Actually, this is not needed: we can as well use a different RD per PE per customer. In other words, if different PEs use different RDs for the same customer, it is perfectly fine. In fact, we don't need RDs to identify customers, we just need them to distinguish overlapping addresses. Pay attention to this point: RDs do not univocally identify customers. Note that the RD is removed before an address is inserted in the VRFs (although BGP still keeps internally the association between VRF entries and the full VPN-IP address).

We said that multiple RTs can be included in a BGP message, because we may want a PE to add a route to multiple of its VRFs, corresponding to different customers. This is typically done when two customers want to share part of their network infrastructure (*extranet*) and therefore a customer needs to provide access to part of its infrastructure to the other (and vice-versa).
In the following, an example of shared infrastructure is shown:



Grey site 1 and yellow site 1 can communicate because PE1 exports Route Target RT1ex from the grey VRF, which is imported by PE2 yellow VRF, and vice-versa with RT2ex.

Routing at a PE is label switched: a PE (destination PE), upon receiving a packet from another PE (source PE), routes that packet to a given destination network depending on the label applied to it. The source PE labels the packet depending on the destination that has to be reached. The labels to be used to reach specific destinations are chosen by the destination PE, which distributes them to the source PE. Label distribution occurs directly in the BGP message, in which another attribute containing the label to be used to reach the specified destination is included. At the source PE, once the selected label has been applied, another label, which corresponds to a pre-existing LSP between source and destination PE, is stacked on top of the first, so that the packet is tunneled over this LSP through the core network (therefore, there are two distinct labels, one for tunneling and the other for label switching at the destination PE).
The example shown in the following figure shows this procedure:



- A packet coming from CE2.2 and addresses to 10.1.1.1 arrives at PE2
- PE2 knows that the packet comes from the yellow customer and therefore it has to use the yellow VRF to route it
- The packet is forwarded towards PE1, after applying a label L2 that was chosen by PE1 beforehand
- Before forwarding, another label Lx is stacked on top of the packet, which allows to tunnel the packet over the pre-existing LSP towards PE1
- Upon receiving the packet, PE1 recognizes the label and it knows that it has to forward towards CE2.1

Note that label switching at the destination PE is not only good for performance, but it also allows to solve ambiguity due to overlapping destination addresses: in the example above, PE1 can determine the next hop without ambiguity thanks to the label rather than based on the destination address.