Programmable Networks

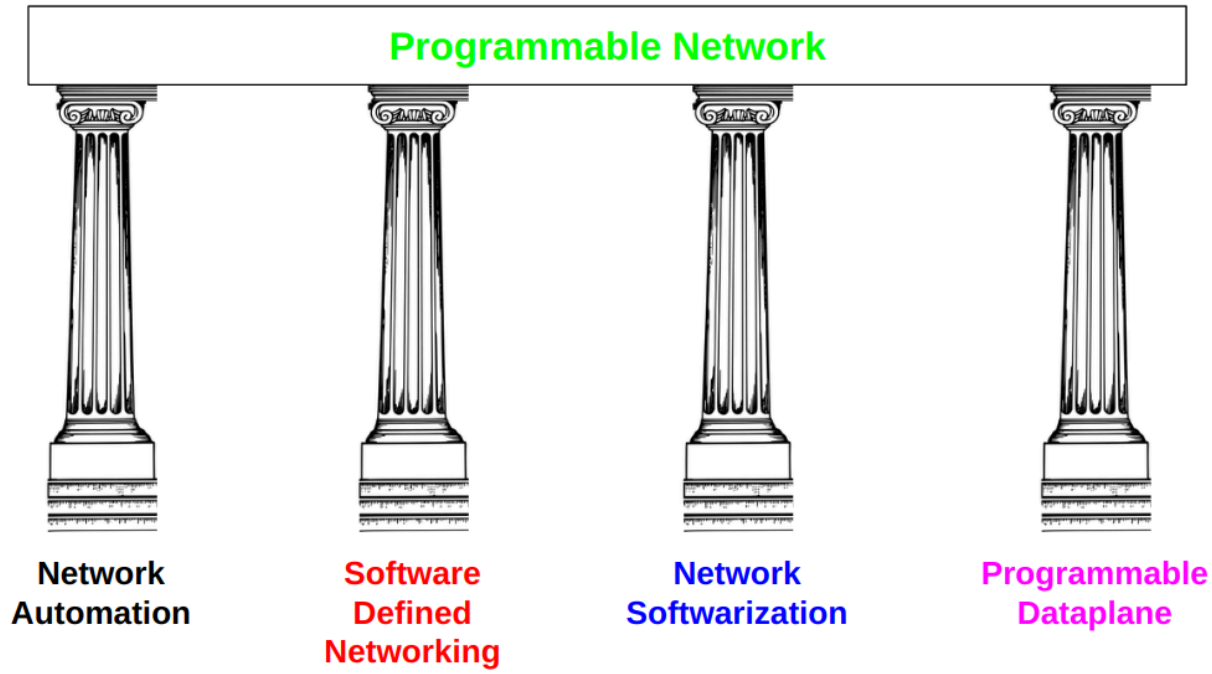
Simone Palumbo's Notes 2023/2024

MSc in Artificial Intelligence and Robotics

Introduction

28/02/2024

Pillars of Programmable Network:



- Network Automation: automate the network management (configure routers, assign IP addresses, set up firewall rules). For doing so we need APIs.
- Software Defined Networking: approach that enables dynamic and programmable networks. Decoupling of Control and Data Plane.
- Network Softwarization: create on demand virtual machines that represent virtual elements of virtual networks. Decoupling HW and SW.
- Programmable Dataplane: decoupling HW and SW as in network softwarization but can't use horizontal scaling since there is the Amdahl Law. So we use DPDK and VPP. Not easy since the data plane is made by the hardware.

A network is composed by 3 layers:

1. Data Plane, for which we have the Programmable Dataplane. A router dataplane is the forwarding element.
2. Control Plane, for which we have the Software Defined Networking. A router control plane is the route processor
3. Management Plane, for which we have network automation. A router management plane can be accessed with the CLI.

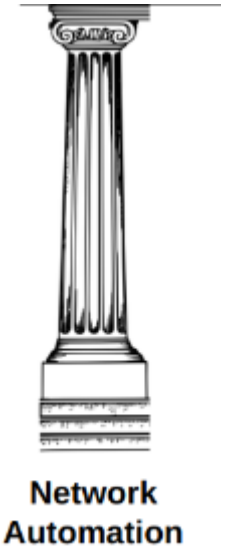
Auto-Learning in switches refers to the capability of a switch to automatically detect and learn the MAC addresses of devices connected to its ports. Here's how it works:

1. Initial State: when a switch is first powered or reset. In this state it doesn't know which devices are connected to which ports

2. Learning Phase: while data packets flow through the switch, the switch analyzes the source MAC addresses of the packet, associating in the MAC Address Table (Forwarding Table) the source MAC address with its port from which the packet came
3. Forwarding: now that the switch has learned, it can forward incoming packets to the correct ports without flooding. If the switch doesn't know the output port, it does flooding, so it forwards the packet to every port
4. Aging Process: if a device does not communicate with the switch, its entry in the Forwarding Table will be removed. In this way the table is always updated and never too large

Network Automation

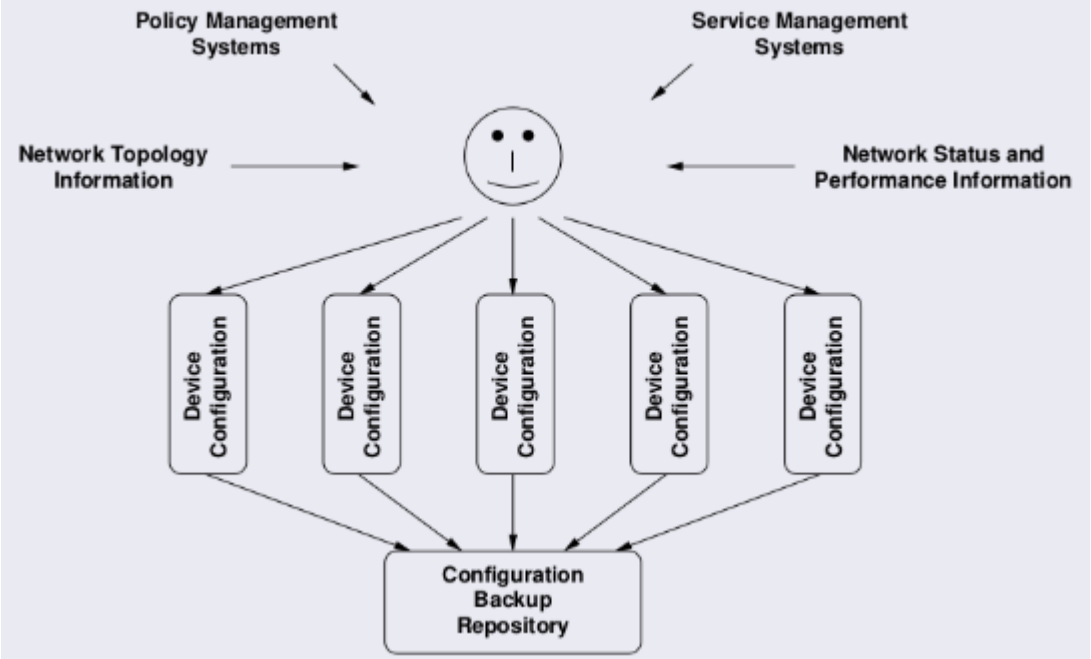
29/02/2024, 6/03/2024, 7/03/2024



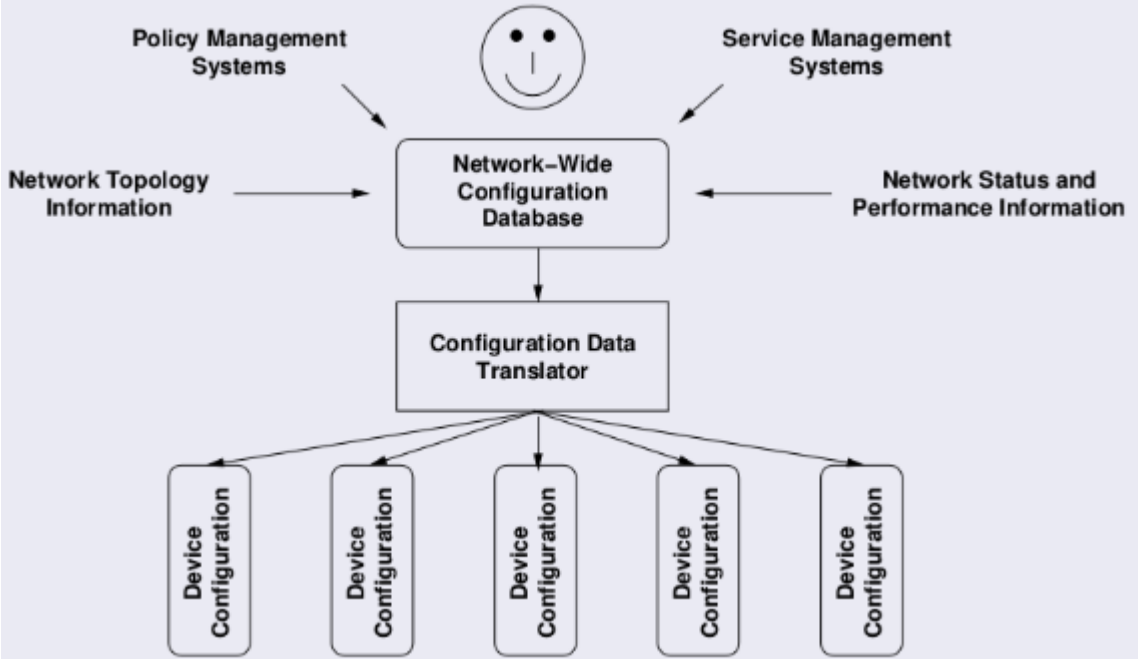
"The Network is the Record" vs "Generate Everything" approaches

You can configure a device manually. For instance, you can open your terminal and type: *ifconfig eth0 10.0.0.1 netmask 255.0.0.0 up*. This is a management task, with which I am assigning a configuration parameter to a network interface. A network operator takes as input a management task (e.g configure OSPF in all devices), and uses its machine to connect to every single device at a time and configure them with CLI. To connect to the devices, the operator can plug a cable (console cable) but that requires physical connection, or have a remote connection (TELNET or SSH for instance). In this approach "The Network is the Record", since the network is a database of configuration files, in which we store our information. The drawbacks of this approach are:

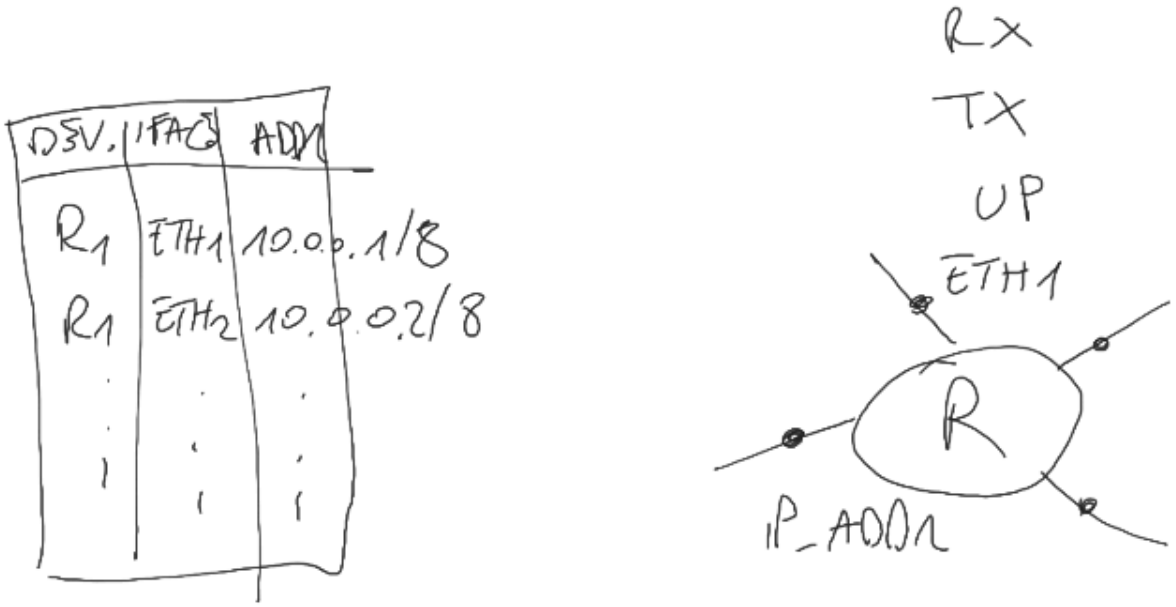
- Labor intensive
- Expensive
- Error prone
- Since each vendor has its own management plane, the operator must be skilled in all of them



The other approach is to “Generate Everything”: instead of interacting with every device, the network operator interacts with a Configuration Data Translator, that translates the received input in a set of configuration commands for the different devices.



For instance:



So the operator writes just a text file instead of a lot of commands in the CLI. The configuration data translator then translates this table in a list of commands. In this case, the devices can be heterogeneous, so from different vendors.

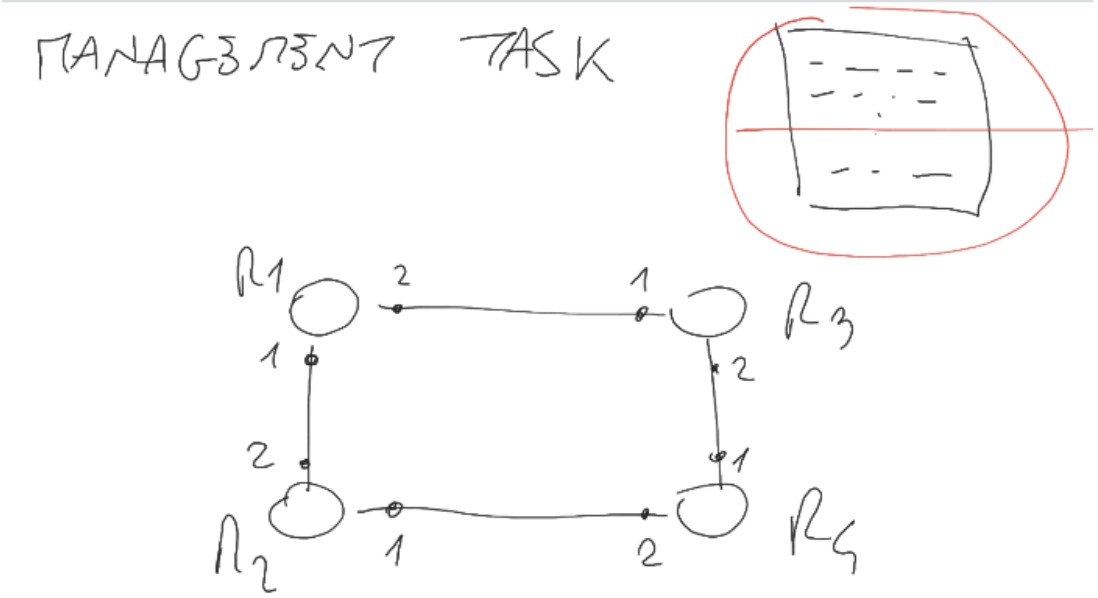
Configuration Management Protocol Requirements

- Requirement 1: a CMP must be able to distinguish between:
    - configuration state: things that are explicitly configured. If you want to assign the IP addresses this is a configuration state. Also the routing protocol is a configuration state, because I have to configure it.
    - operational state: things that specify the state of a device. If a router has 4 interfaces and we name one of them ETH1, ETH1 is an operational state. It's data that represents the state of the interface. Another example is “the interface status is up”.

  There are some elements that can be both configuration and operational. For instance the IP address is configurable but once is set it becomes an operational state for the device.

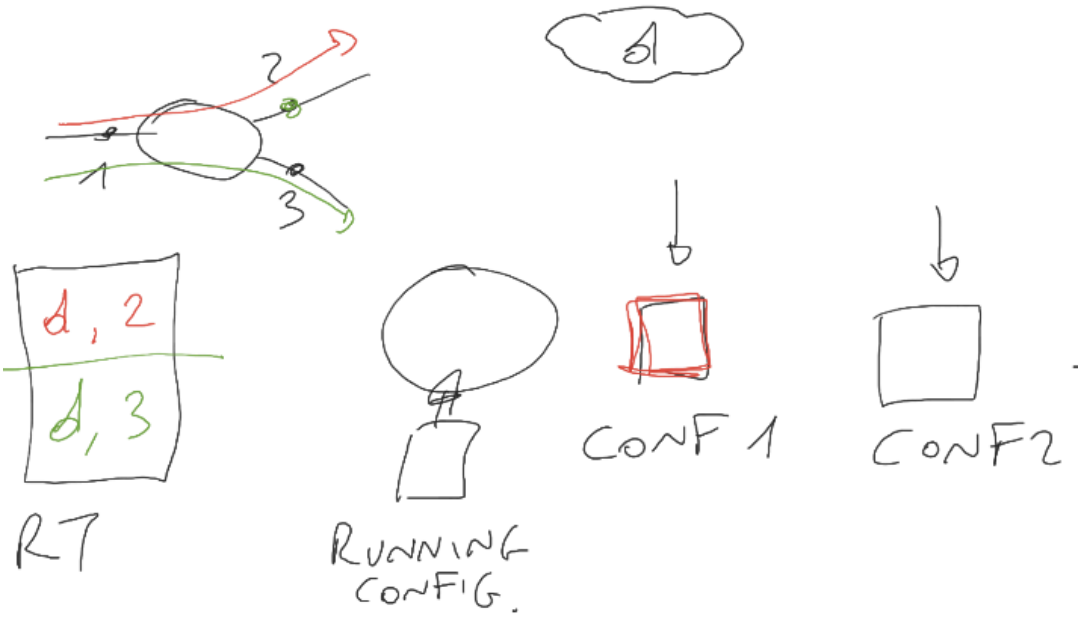
- Requirement 2: a CMP must provide primitives to prevent errors due to concurrent configuration changes. In fact, in a multithreaded system, if one of your threads wants to do something in a cell memory and another thread wants to read the same cell, these two operations can’t be performed at the same time. There must be a locking mechanism.

- Requirement 3: a CMP must provide primitives to apply configuration changes to a set of network elements in a robust and transaction-oriented way.



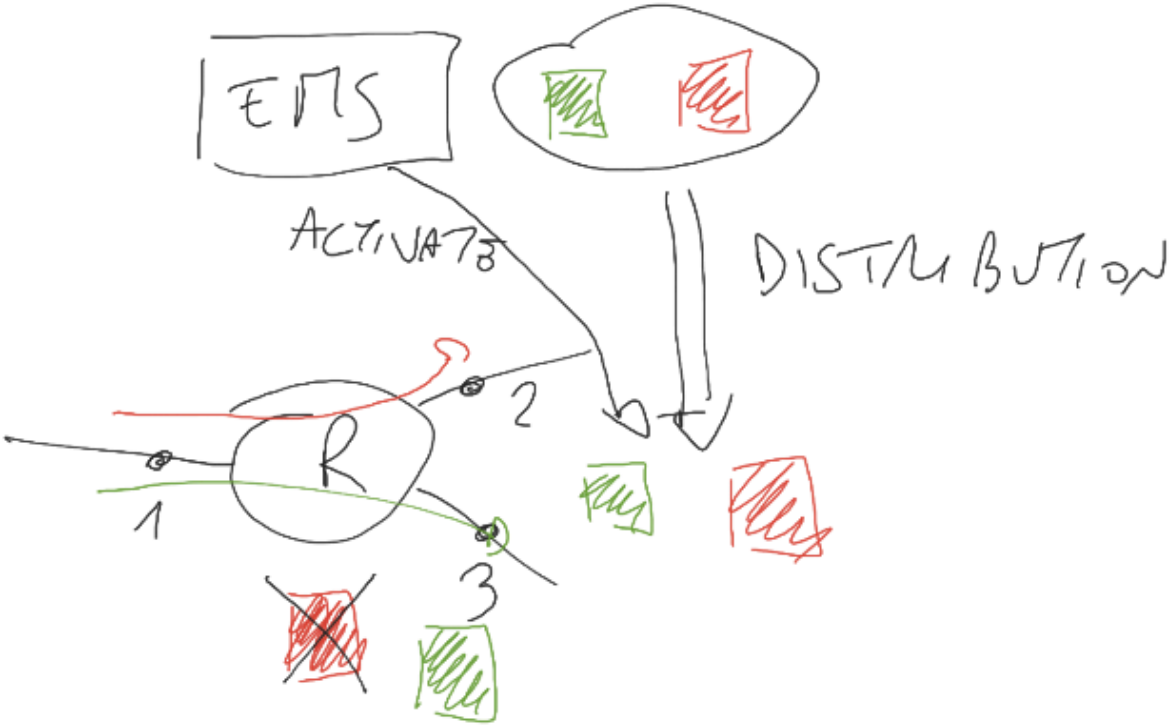
Given the network of four routers above, the management task is to assign IP addresses to all the interfaces of all the routers, having the list of all the operations needed for doing so (assign address on interface 1 of R1, 2 of R1, …). The management task is completed only when all the commands have been successfully issued. If the procedure stops in the middle this is a problem because I have part of the network with the old configuration and part with the new one. For this reason all these instructions represent transactions, I am satisfied only when all the commands have been issued successfully, this is robustness and transaction-oriented required by R3.

- Requirements 4: a CMP must be able to distinguish between several configurations and devices should be able to hold multiple configurations.



A device always has a running configuration, stored in the RAM. The device should be able to switch from CONF 1 to CONF 2. But why? Maybe the amount of traffic from the current used link increases so we may want to move from another path, and maybe while the running configuration says go through interface 2, the new configuration says go through interface 3.

- Requirements 5: distinguish between distribution and activation of configurations. Example:



Assume the Element Management System has two different configurations. When the red one is applied to R the path will be the upper one. The EMS sends the configuration to the router R, this is the distribution of the configuration. Since the router can have only one running configuration, the EMS could perform an activation[1], to specify which of the two configurations is the running one. The distribution is done once for several configurations because it is more efficient to activate a different locally stored configuration than just ship it from the EMS.

- Requirements 6: a CMP must be clear about the persistence of the configuration changes. For instance, if I want that at the next restart the configuration is deleted, or if I want to apply a change immediately and be remembered after reboots. You can modify the running configuration and apply the modification immediately, but if there is a problem, maybe the router stops working and it becomes unreachable. So it's riskful. For this, most of the time you use a "cold" management, so it will be modified and made active in the next reboot. When we are sure there are no mistakes in the modification we just reboot the device. For instance, NETCONF exposes 3 different

[1] Yes, is the EMS that do both the distribution and the activation since the switch is not intelligent

commands so that the client can interact with the configurations: its transaction model will be explained afterwards.

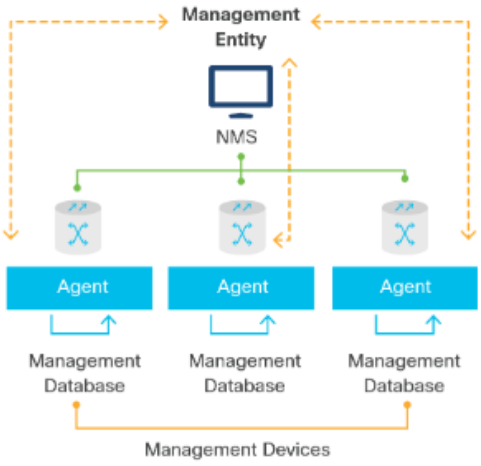
- Requirements 7: a CMP must be able to report configuration change events. Basically, if we change the configuration we want that the event is logged.
- Requirements 8: a full configuration dump and a full configuration restore must be supported.
- Requirements 9: a CMP must support standard tools.

SNMP: Simple Network Management Protocol

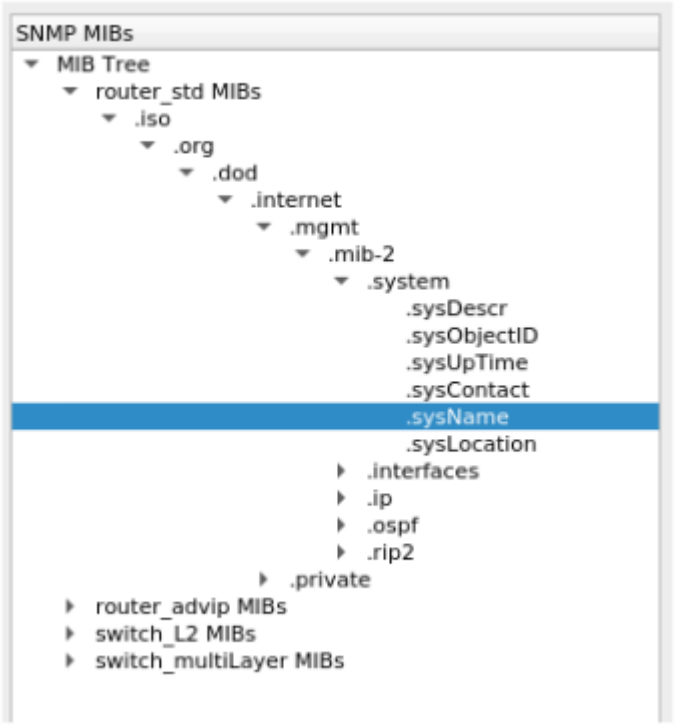
This is the predecessor of NETCONF.

Elements of SNMP

It consist of three elements:



1. SNMP manager: the Network Management System
2. SNMP agents: the managed devices. The agent updates the management database, that hold operational states and configuration states
3. Management Information Base (MIB), is inside the management database. The MIB keeps the relevant data in an ordered manner. MIB structure is a description of the elements of your system. It is organized as a tree:



SNMP has a strange number that specifies what resource the client is asking for. This number, shown above, is not human readable:

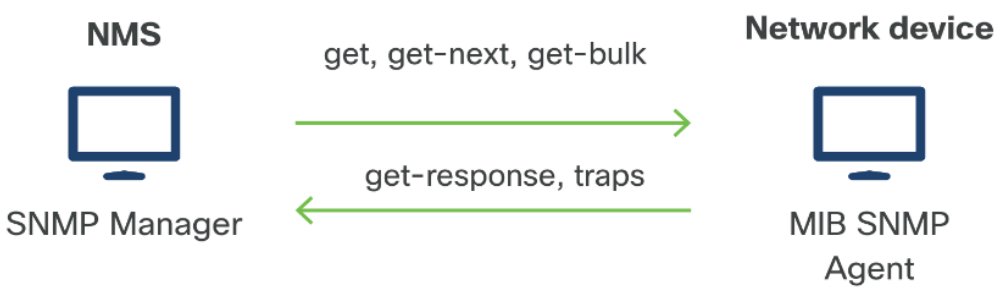
Name/OID
.1.3.6.1.2.1.1.5.0 (.iso.org.dod.internet.mgmt.mib-2.system.sysName.0)

on the other hand NETCONF and YANG have a more readable way of specifying what the client is asking for.

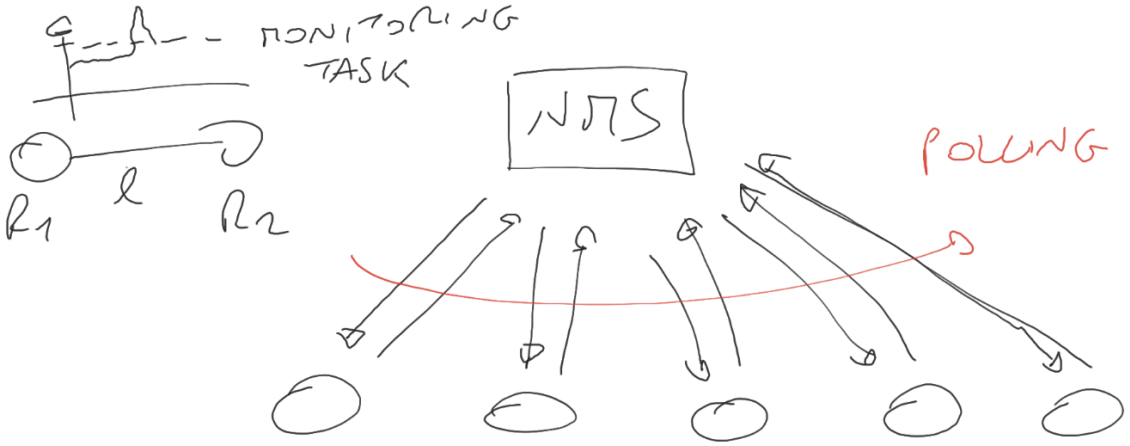
Polling vs Trap

The SNMP protocol responds to the client-server paradigm: the agents are the server and the NMS is the client. The SNMP protocol has two commands to convey MIB info:

1. Request/Response Mode with polling



with a get message the client asks for the configuration or just an aspect of it (e.g. the name of an interface). The answer of the server is a get-response. So the NMS does polling in a round robin fashion to get the configuration data. The problem is that this is time consuming and does not scale.



2. Trap Mode: a trap is a condition that triggers an interaction. For instance, the traffic load is increased over the threshold shown above, and this condition triggers R1 that sends to NMS some data. This allows reduction of overhead w.r.t. to NMS do polling and in this way I also detect earlier bad situations. Other events that can trigger the interaction are, for instance, improper user authentication, restart, closing of a TCP connection…

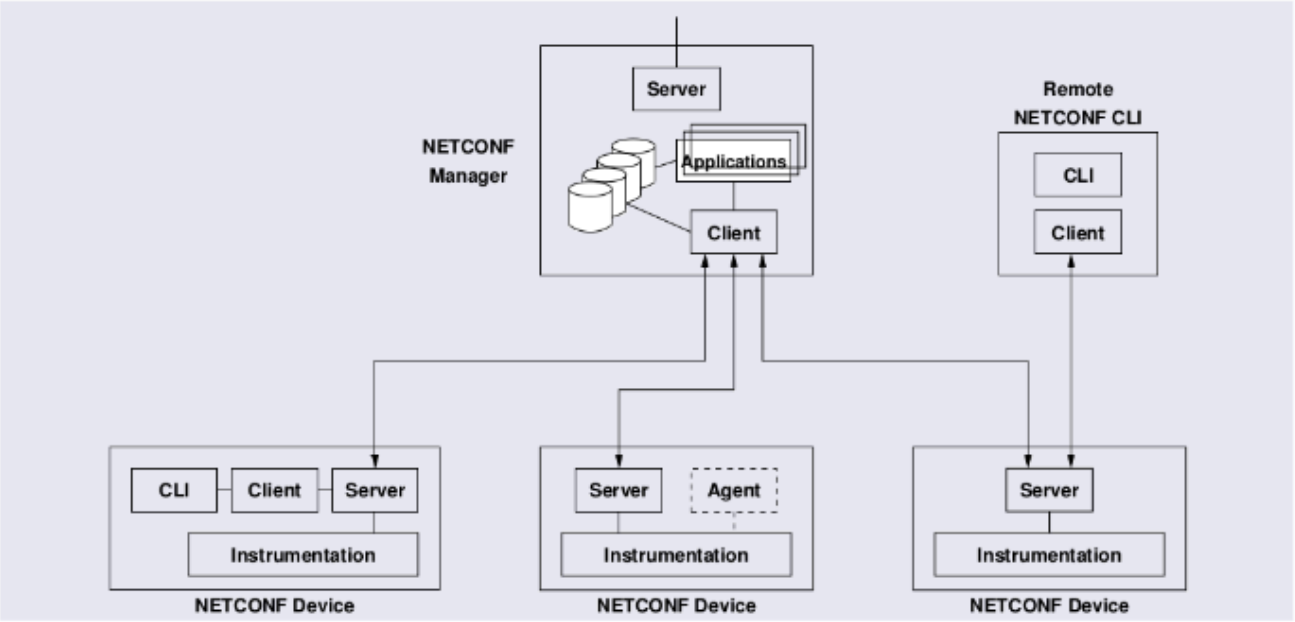
SNMP Shortcomings

1. Does not scale up well especially when doing polling
2. Does not have a discovery process for finding MIBs supported by a device, differently from NETCONF when we have the exchange of capabilities
3. Does not support transaction (Requirement 3)
4. Lacks backup-and-restore of element configuration: no rollback

NETCONF: NETwork CONFiguration protocol

NETCONF Deployment Model

The NETCONF Client is in the manager, while the Servers are in the devices.



We have:

1. NETCONF devices. In all the three cases we have the instrumentation (that is hardware + data plane + control plane), what changes is the management plane:

    - The device has a local NETCONF server, NETCONF client and a CLI. The CLI is used to tell the client the commands to execute. This is the worst approach since it's not automated (we have a CLI) and since the CLI + NETCONF Server and Client are local you have to access and configure the device locally[2].

    - In the second case the management plane has only the NETCONF server, which allows the management remote system to connect and interact with the device. Then applications in the form of scripts are run to automate the configuration of the device. This is the best approach since it's automated and you don't need local access to the device. The agent is needed in this case since we don't have a CLI for manual configuration

    - In the third case we use a remote CLI with a remote NETCONF client, so without application scripts that automate the configuration but we do that manually. The difference between this approach and the first one is that here

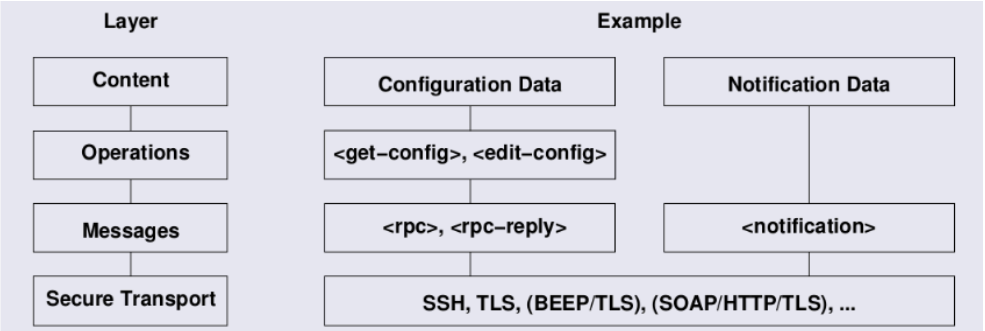
[2] I think that the arrow between the manager and the device is just a connection used for making the manager aware of the presence of the device, but the configuration happens locally

we use NETCONF remotely, while in the first approach we can use it only locally. So it's a better approach but not the best

2. NETCONF manager, that is the client. Inside the manager we have:
   - databases, where we store the configurations
   - applications: scripts in a high level programming language to automate some processes.

   By the scripts we can take as input the data from the databases and then through the client you interact with the network devices

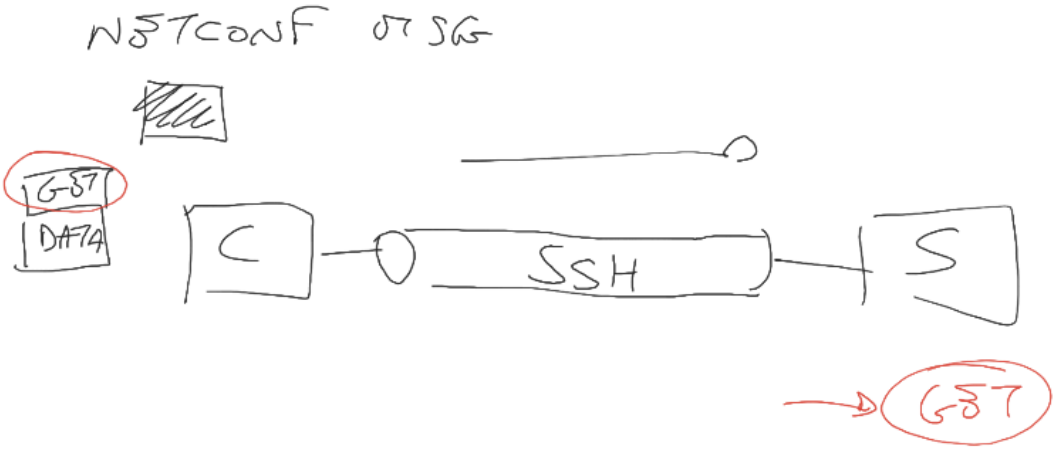
Layering Model of NETCONF



The messages we exchange have the following layers:

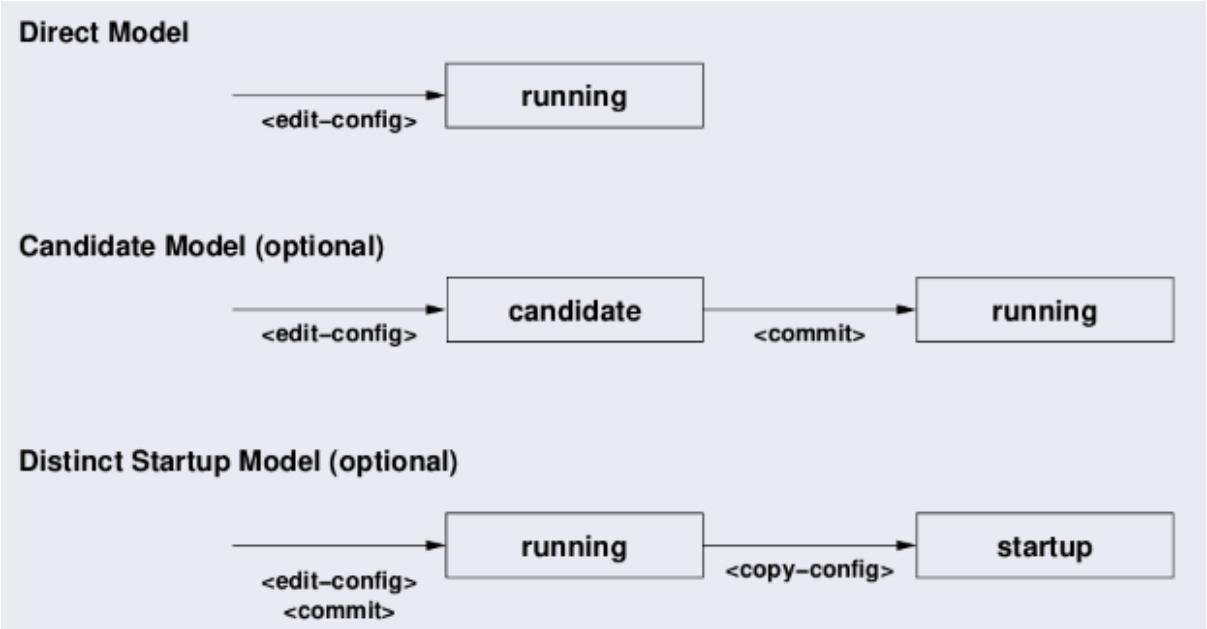
1. Payload: the actual message, the configuration data.
2. Operations: focus on the primitives. The client has to tell the server which type of operation it wants to perform: do you want to get some configuration parameter, to change others or what?
3. Messages: this layer specifies, for instance, that the get command is implemented as a RPC[3]
4. Secure Transport: SSH can be used to deliver these messages

[3] When you do a RPC you are asking the remote computer to execute a local function it has. For instance, if you want to obtain some configuration data in the device, you can ask the device itself to execute a function called get-config e return its result to you



An SSH connection is established between the client and the server. The client then sends a NETCONF message to the server. This message is formatted using the YANG language.

NETCONF Transaction Model



1. Direct Model: you can edit directly with the running configuration with the <edit-config> on the running configuration. “Hot” management. If you make an error here, the rebooting will not save you since the startup coincides with the running.
2. Candidate Model: you can edit the candidate configuration (the one that will be applied at the next startup) and then commit the change. “Cold” management.
3. Distinct Startup Model: you can create a “snapshot” file called startup and edit the running configuration and copy the modification in the startup configuration. So you are “parato” if there are errors in the running configuration, you just have to reboot.

YANG and Capability Exchange

Differently from the SNMP MIB, NETCONF has a format that is like an XML file, more human readable:

```
S: <hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
S:    <capabilities>
S:          <capability>
S:                urn:ietf:params:xml:ns:netconf:base:1.1
S:          </capability>
S:          <capability>
S:                urn:ietf:params:xml:ns:netconf:capability:startup:1.0
S:          </capability>
S:          <capability>
S:                urn:ietf:params:xml:ns:yang:ietf-interfaces?
S:                module=ietf-interfaces&amp;revision=2012-04-29
S:          </capability>
S:    </capabilities>
S:    <session-id>4<session-id>
S: </hello>
```

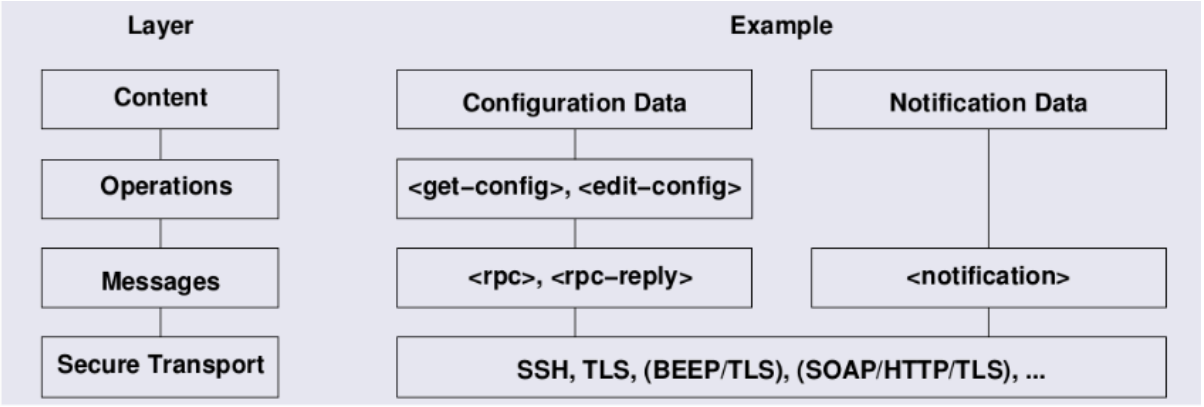
As you can see from this hello message we are specifying:

- The NETCONF version
- The capabilities. A capability is a specific YANG model that the device supports. For instance this model implements the ietf-interfaces
- Session ID

```
C:   <rpc message-id="101"
C:            xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
C:        <edit-config>
C:            <target>
C:                 <running/>
C:            </target>
C:            <config xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0">
C:                 <interfaces xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
C:                      <interface nc:operation="replace">
C:                           <name>Ethernet0/0</name>
C:                           <mtu>1500</mtu>
C:                      </interface>
C:                 </interfaces>
C:            </config>
C:        </edit-config>
C:   </rpc>
```

One of the problems of SNMP was that the client doesn't know the versions I am supporting. With NETCONF/YANG I can tell the client what versions I support as capabilities. With the <edit-config> command, the target specifies what is the configuration we want to interact with, this time is the running one (<running/>). Moreover, it specifies what we want to do: replace, so override, the configuration, assigning a new name that is Ethernet0/0 and a new MTU that is 1500.

With YANG we are focusing on the configuration data:



For instance, we can create a parser that takes as input the text of ifconfig and look for keywords. The problem is that this is “hardcoded” for a specific OS[4]. With YANG, on the other hand, since it’s a standard format, all the vendors that decide to be compliant to it must display the info in its way, and the script will work for each OS.
Let’s see an example of a YANG module:

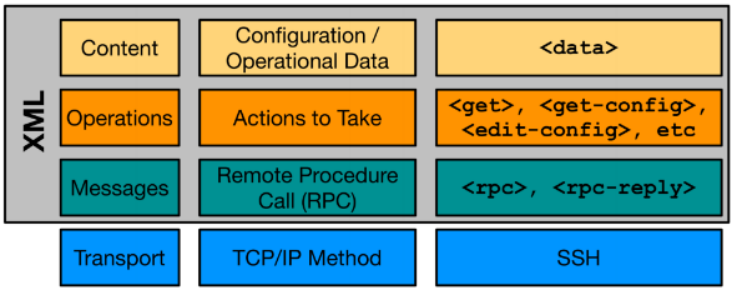
```
module ietf-interfaces {
  import ietf-yang-types {
    prefix yang;
  }
  container interfaces {
    list interface {
      key "name";
      leaf name {
        type string;
      }
      leaf enabled {
        type boolean;
        default "true";
    }
}
```

This module has a single container. A container is a portion of the model. In this case we put all the interfaces in this container, in the form of a list. This container has only a *list*, that is a sequence of *leaves*. A leaf is the element of the configuration file I can interact with. They are attributes of the model that can assume multiple values. In this case the leaves of the interface list are the *name*, that can assume values in the string space, and *enabled*, that is a boolean value.

[4] e.g. the ifconfig of ubuntu

Classical NETCONF Communication Workflow

NETCONF has a multilayer structure, based on the XML format.



The content is a request from the client, formatted with a YANG model. The last layer, the transport with SSH, requires a connection between the client and the server. Connecting from the client to the server can be done in the following way:

```
$ ssh admin@192.168.0.1 -p 830 -s netconf
admin@192.168.0.1's password:
```

SSH Login

Note that 830 is a standard port for NETCONF

```
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.1</capability>
<capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-netconf-monitoring</capability>
<capability>urn:ietf:params:xml:ns:yang:ietf-interfaces</capability>
[output omitted and edited for clarity]
</capabilities>
<session-id>19150</session-id></hello>]]>]]>
```

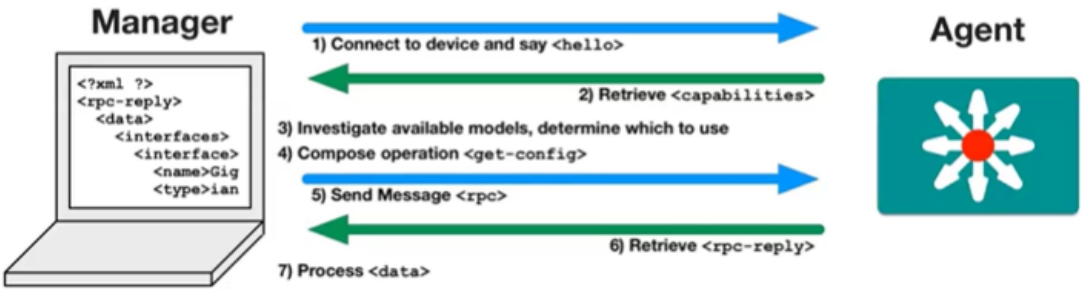
Server (Agent) sends hello

After the login, the server will send you back the hello message. This message contains the capabilities, in the form of yang models, that the server supports. After receiving the hello message, you (the client) can send the hello message to the server:

```
<?xml version="1.0" encoding="UTF-8"?>
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<capabilities>
<capability>urn:ietf:params:netconf:base:1.0</capability>
</capabilities>
</hello>]]>]]>
```

Client (Manager) sends hello

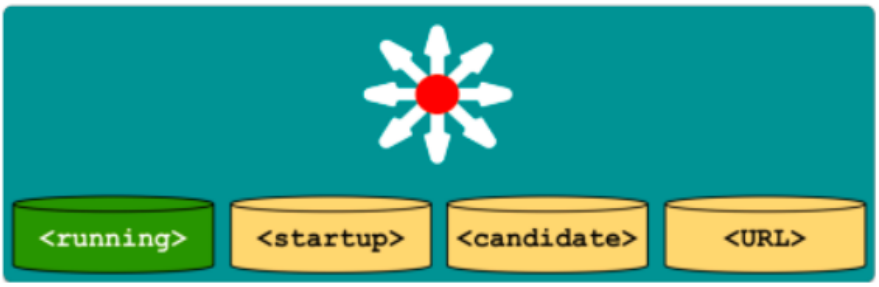
Doing this manually is not the right way to do it! We want to automate it with scripts!



NETCONF Data Stores

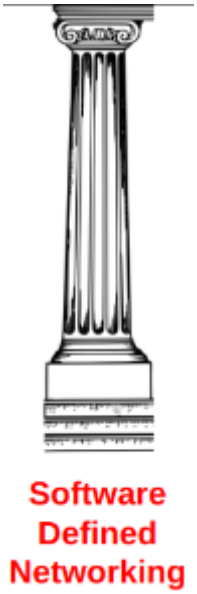
Only the first 2 are mandatory.

- Running configuration: the one currently loaded in the RAM
- Startup configuration: is loaded inside the RAM at next reboot.
- Candidate configuration: optional
- URL configuration: optional



Software Defined Networking

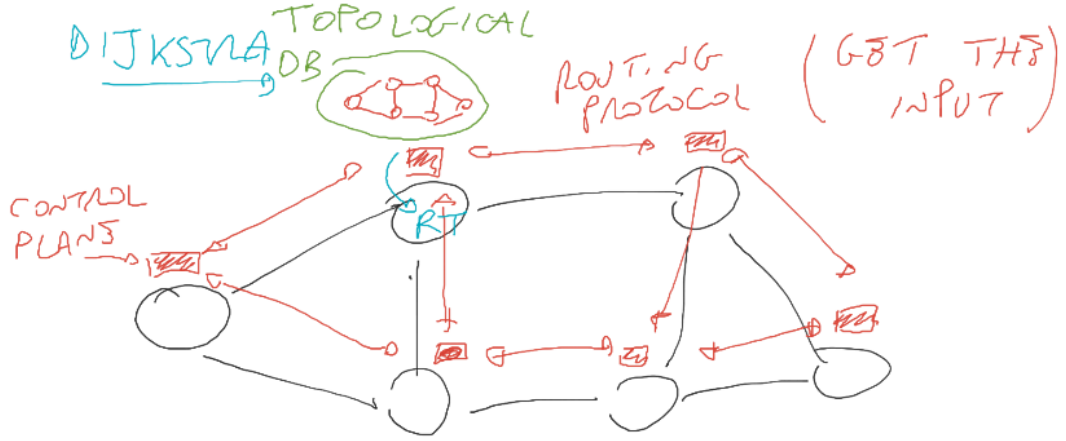
7/03/2024, 13/03/2024, 14/03/2024, 20/03/2024



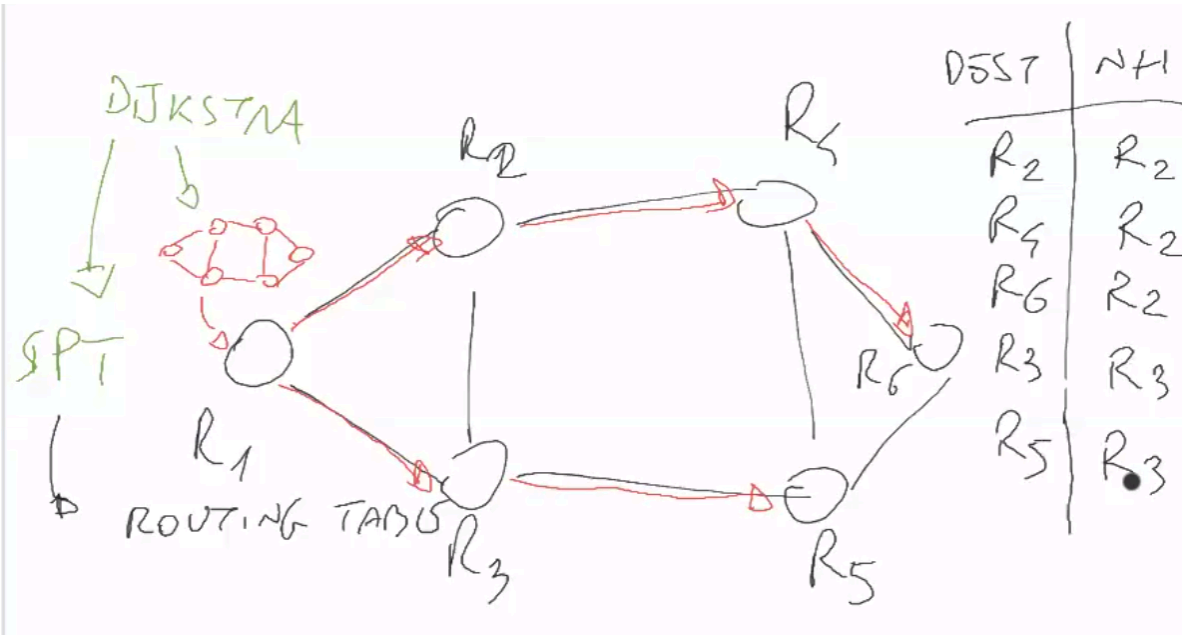
We are now the programmability of the control plane. The control plane's most common functionality is routing. We overcome the problem of Internet Ossification with decoupling. In the case of SDN the decoupling consists of separating the data plane from the control plane: while in a regular router we integrate all the three planes, with decoupling we have devices that are able to deal only with data plane operation while the control functionalities is bring away and put on a remote element. Concretely the control plane is a process running on the machine, so we are running the process not anymore on the device but on the SDN controller

Per-Router Control Plane vs SDN approaches

Let's consider this example, that is a per-router control plane approach:



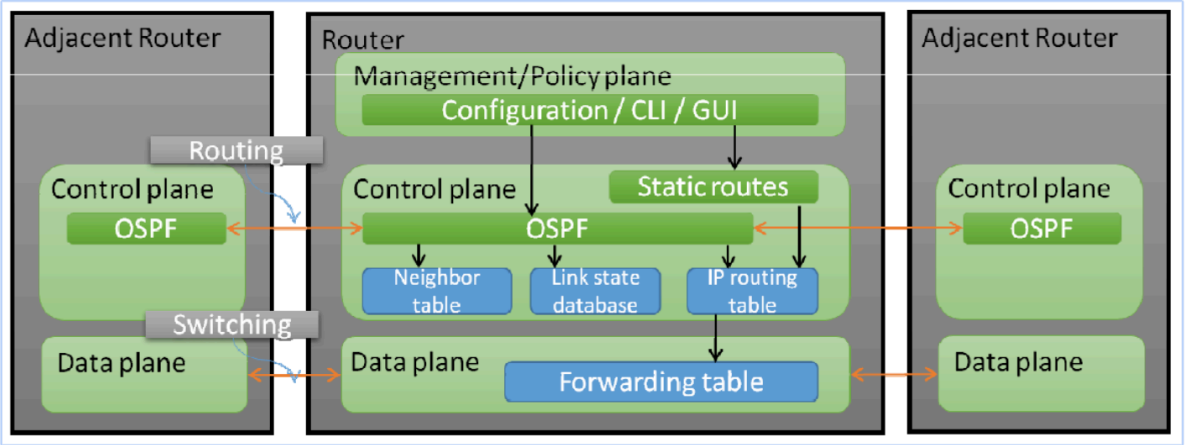
Each router of this network has its own control plane. After a message exchange phase every node will know the topology of the network. Once each router knows the topology, it stores it in the topological DB and it applies Dijkstra. Dijkstra returns the shortest path tree SPT. Now can create the routing table:



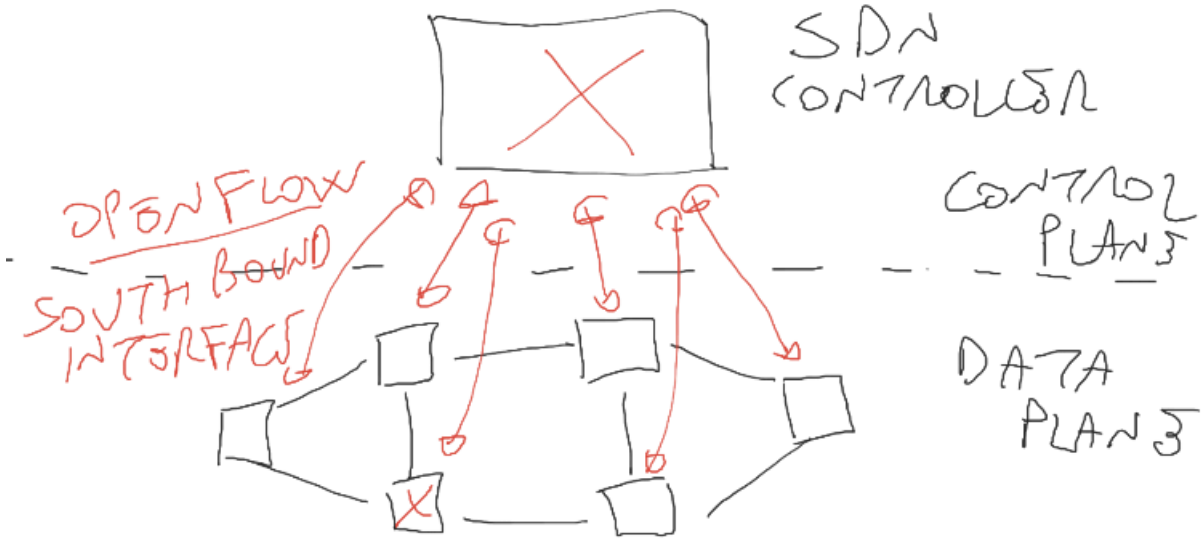
The routing forwarding is a data plane responsibility.

In this traditional networking approach the router (the box at the center) implements all the 3 planes:

1. The Management Plane configures the OSPF daemon or insert static routes (with CLI or Yang), that are both part of the control plane.
2. The control plane interacts with the adjacent router's control plane with routing protocol messages or signaling messages in order to compute the shortest path. The IP Routing Table[5] is part of the control plane since it is created with OSPF or static routes.
3. The dataplane, thanks to IP routing tables, creates while the Forwarding Table[6]



Another approach, the SDN one, is to decouple data plane and control plane:



Since we took the routers and removed the routing capability, these are not routers anymore, just L3 switches. The SDN controller must know the topology, so we need a communication interface between the control and data plane, called south bound interface. For this interface we can use the Openflow protocol. Openflow it's a point to point protocol.

---

[5] also called RIB: Routing Information Base
[6] also called FIB: Forwarding Information Base

Planes time scales, task and location

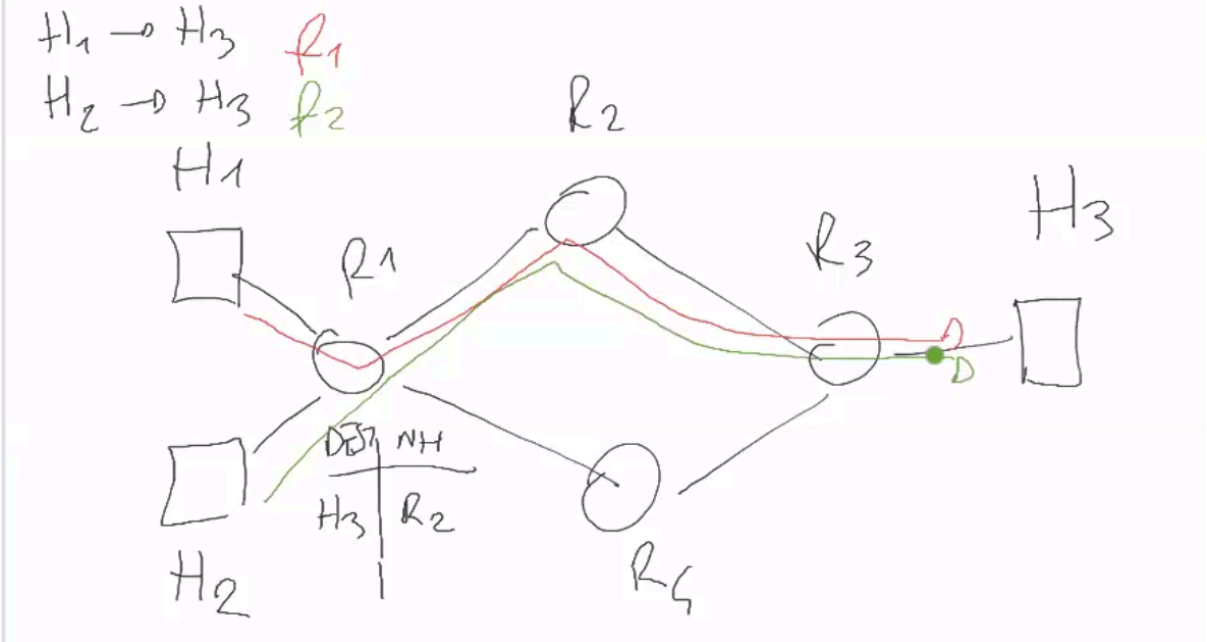
To summarize:

	Data	Control	Management
Time scales	Packets	Events	Humans
Task	Forwarding/ buffering/ filtering/ scheduling	Routing, circuit set-up	Analysis, configuration
Location	Hardware • Specialized hardware • Processes at line rate • Every packet • Very fast	Router software • Uses CPU • Can only process a small number of packets • Very slow	Human or perl scripts

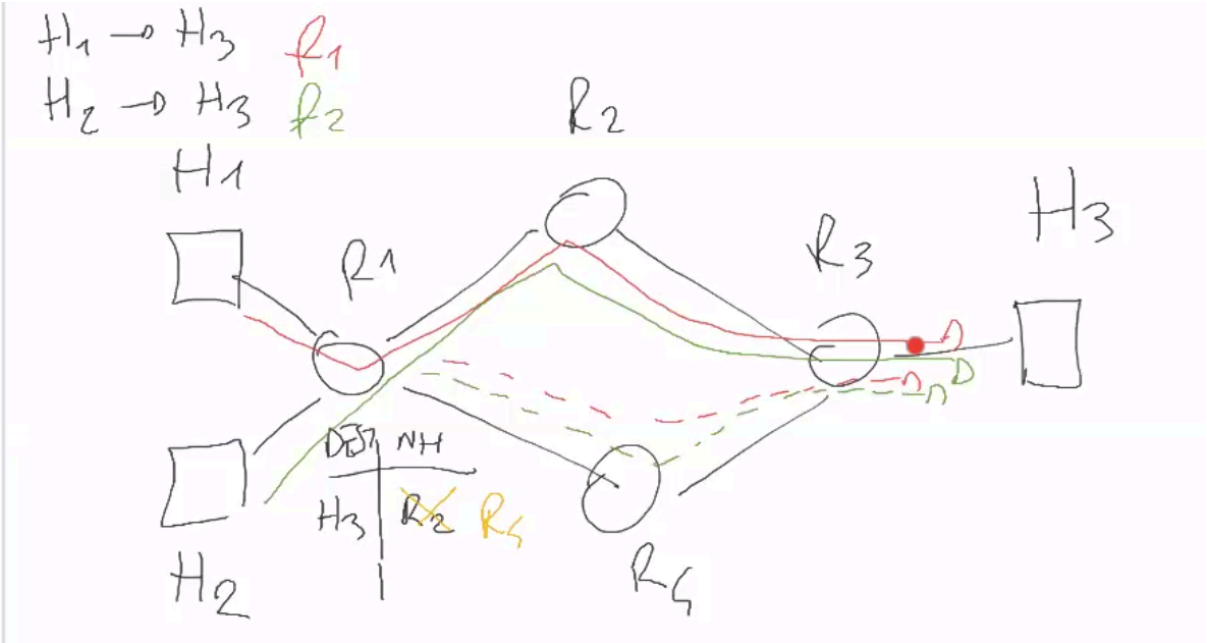
- The dataplane works at nanoseconds level (packet level). The dataplane is implemented in the hardware of every single device.
- The control plane works at events level (milliseconds), so it reacts when there is a change in the network topology (e.g. a node is rebooting). Those events don't happen so often, not on the scale of nanoseconds. Problem: what about events that happen at nanoseconds time scale, like congestion? We can't solve the problem on the control plane since it's a lower granularity: the moment the control plane reacts the situation is already changed. The control plane is implemented as software, using the CPU.
- Management Plane: human time scale, e.g. half an hour to configure a router. The management plane uses scripts.

Generalized Forwarding

Let's consider this case, we have two flows, f1 is in red and f2 is in green:

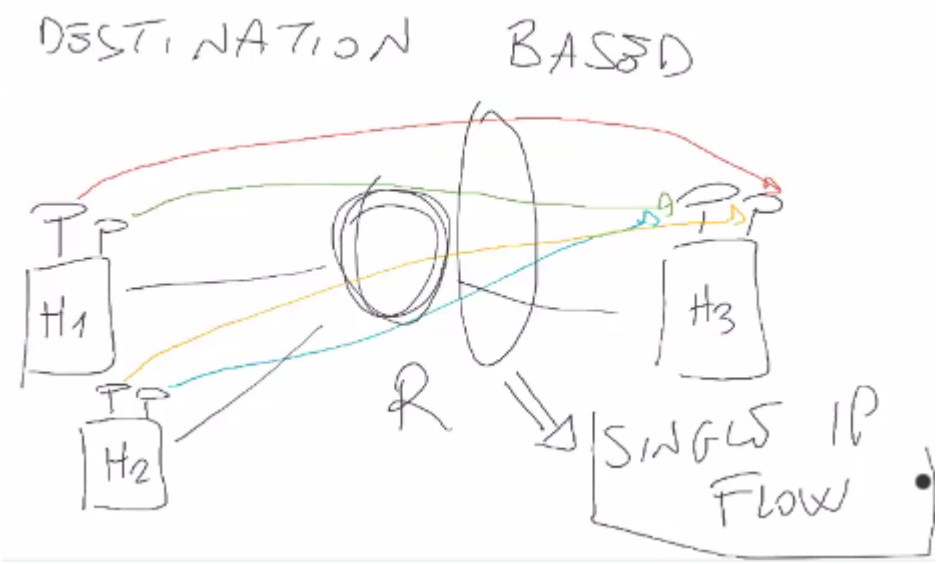


the problem is that we are not using the lower part of the network, so if the two flows become huge there is congestion that could be avoidable. A possible solution is to change the routing table:

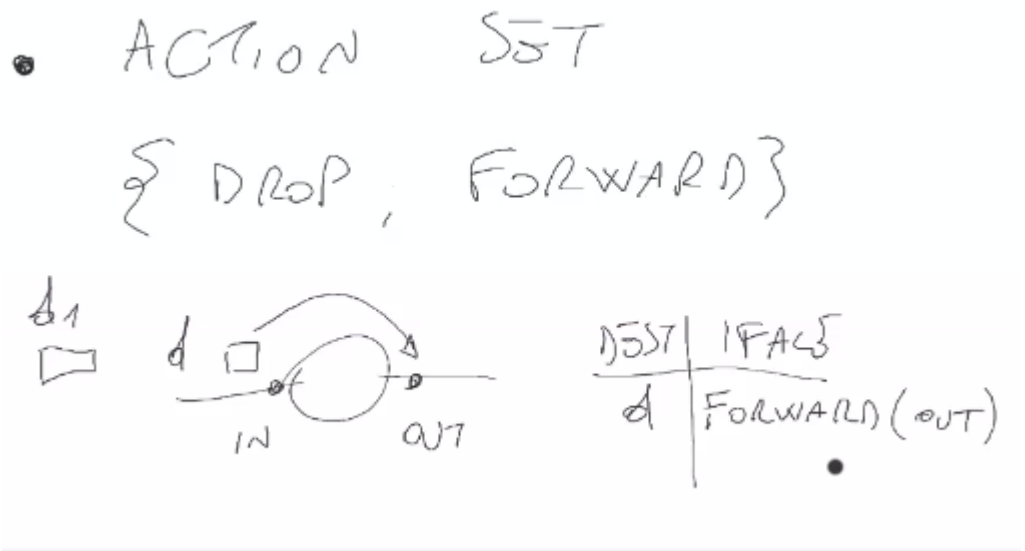


But that could just move the problem on the lower part if the capacity is the same. We are not able to solve the load balancing problem in this way because:

- ip networks work according to the destination based forwarding: we can't distinguish two flows going to the same destination



These are the actions of the ip network:



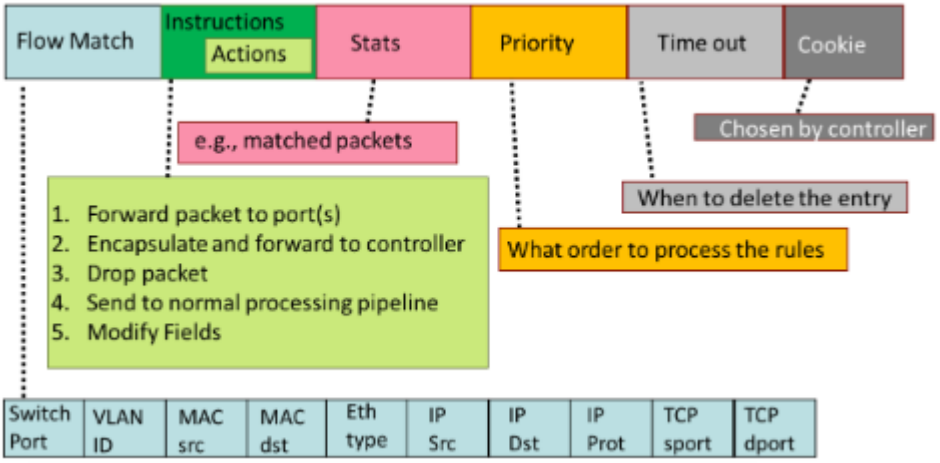
In SDN we will increase this action set.

- we always select the path that has the lowest cost
- single path routing: I can't split the traffic flow over multiple path

How to remove these constraints in order to get more flexibility? With the SDN paradigm.
The three constraints that we want to relax are:

- Destination Based Forwarding
- Multiple actions for a single packet
- Action set not composed only by drop and forward
- bonus: lpm is the only matching paradigm

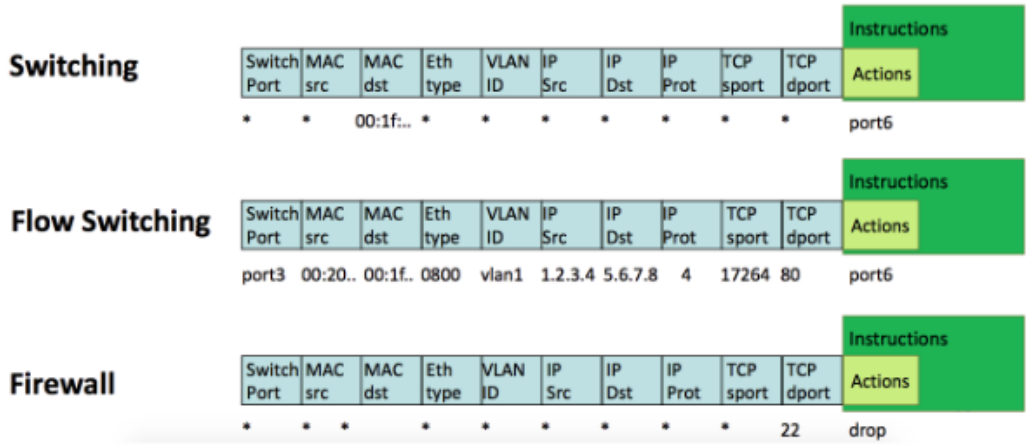
A forwarding device able to match over a wider range of things and can apply a wider set of actions and possible actions over the same packet is a forwarding device that supports Generalized Forwarding. A new action can be for instance "send to the controller".



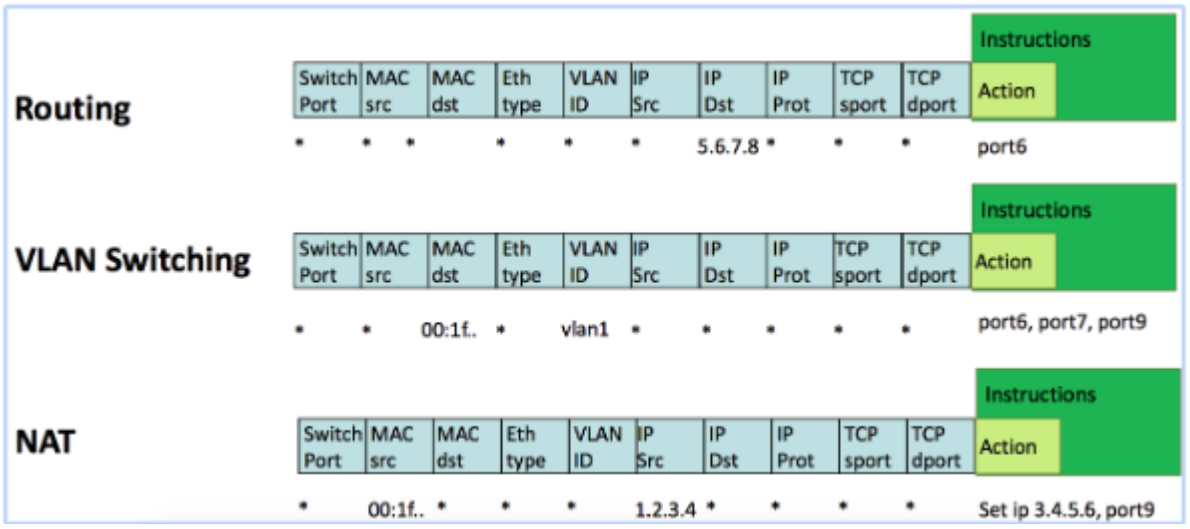
This is the routing table of a device that supports the generalized forwarding. This is called flow table (not routing table anymore). Each entry of the flow table is a flow rule. We call it flow rule since the stream of packets that matches with that rule is a flow. The components are:

- Flow match: physical fields like input ports (L1), MAC addresses (L2), IP source (L3), TCP source port (L4). So it's a multilayer table.
- Actions: set of possible actions is bigger than just DROP and forward. But you can't use your own customized actions.
- Statistics from interfaces and traffic flows.
- Priority: the higher is the number the better is. Equivalent of the LPM rule in IP. In a generalized forwarding we can't rely on LPM since we use data that are not prefix-type data, e.g. the ingress port of the switch. Priority is needed when the packet matches two different rules in the flow table, since you can match with only one rule.
- Time out: remove rule from the flow table
- Cookie 🍪

Asterisk is a wild card: valid values whatever it is. So for the first row we are matching using only the MAC destination address. This is the behavior of an Ethernet switch



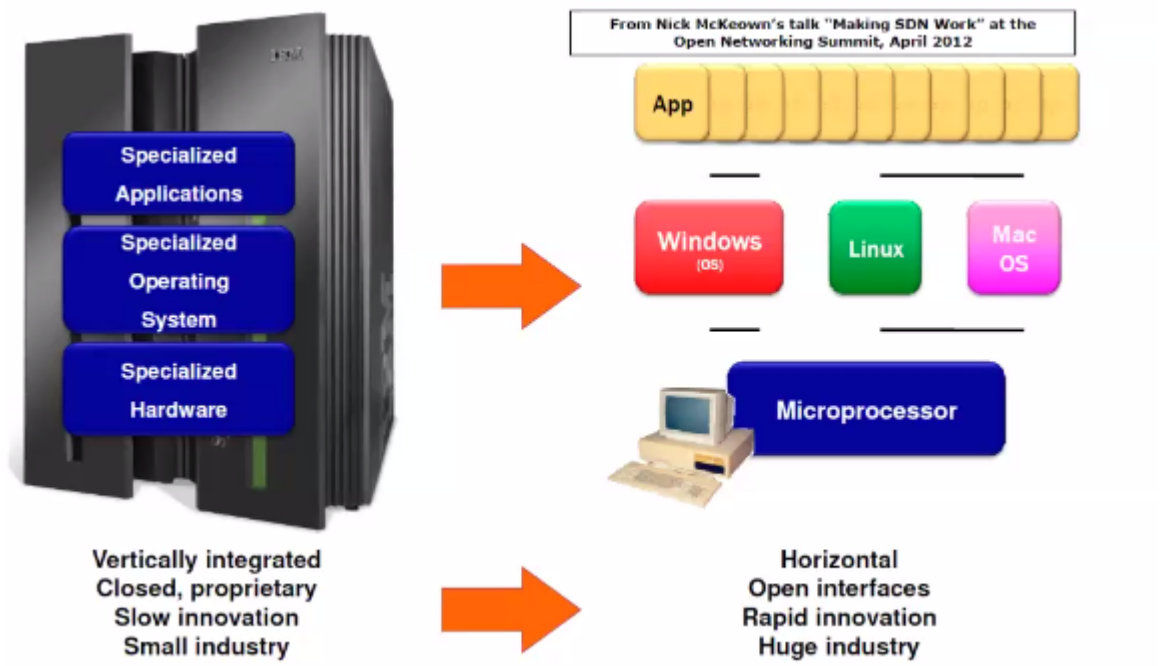
Second row: flow switching, we are describing exactly each field. Third row: behavior of a firewall if the action is DROP.



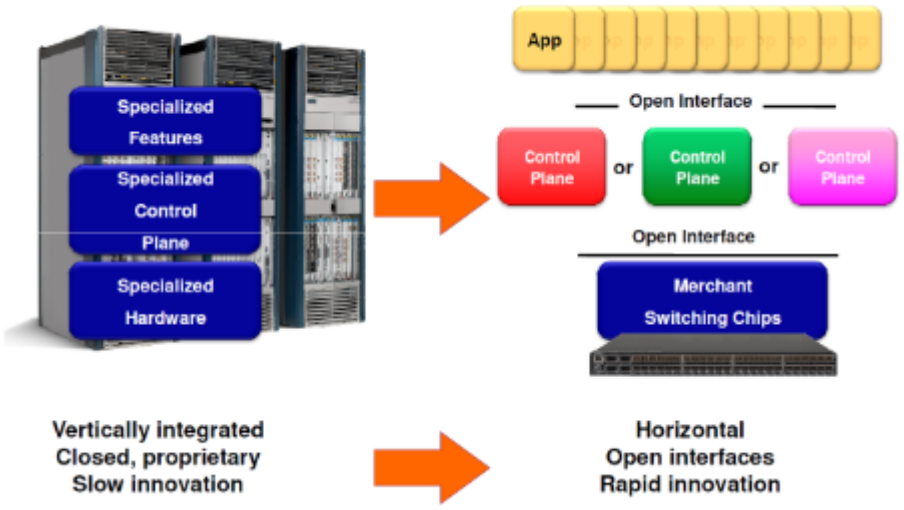
Routers just look at the IP destination address. VLAN Switching is like Ethernet Switch but also looks to VLAN ID. NAT has the SET action. So the device is a programmable device, since we can turn it into a firewall, Ethernet switch, NAT and so on.

Vertical Integration vs Horizontal Approach

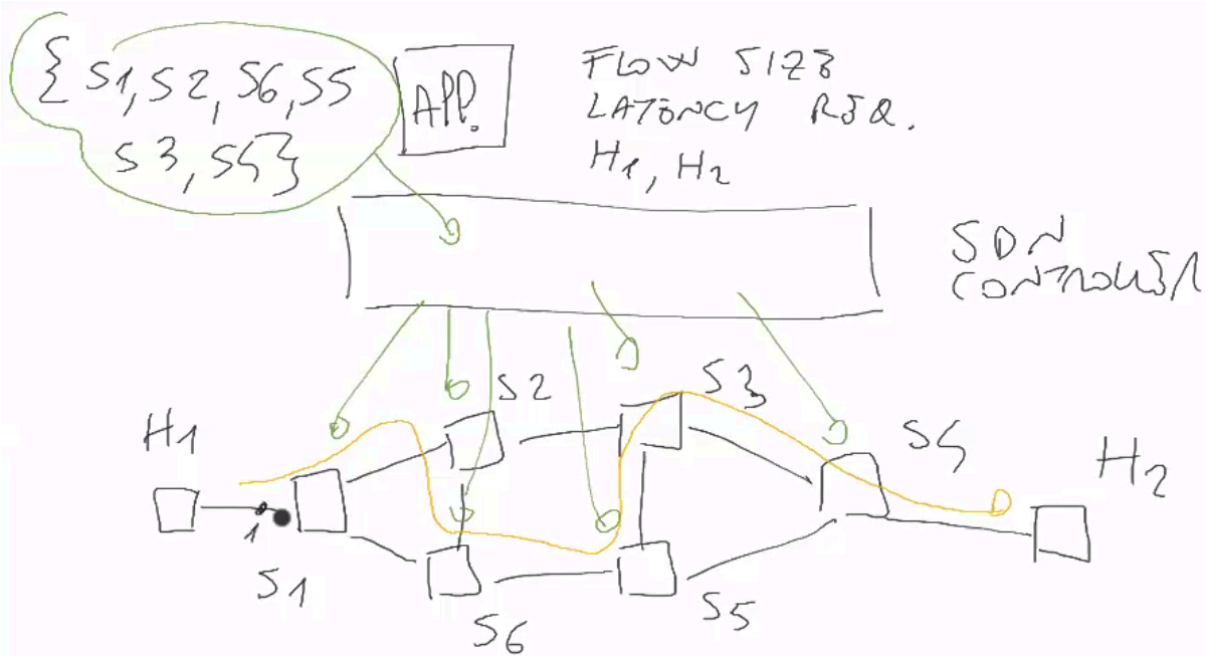
Vertical integration: a big computer that can do a very well a specific task. This is costful and not widespread since the common user doesn't need that. With the horizontal approach we have general-purpose PCs that run a lot of applications. There is an intermediate level that exposes interfaces to the applications in order to abstract the complexity of the hardware: the OS. The OS translates high level instructions to low level instructions.



In the networking field, a Cisco router is a device belonging to the vertical integrated approach: e.g. the control plane, written by Cisco engineer, is specialized and efficient for the device specific task. In the horizontal approach, we have different applications that we can create and customize. Between the app and the hardware we need the corresponding version of the OS in the network realm, that is the control plane, represented by the SDN controller. In this way we have abstracted the complexity of the hardware. The control plane has the task of translating the high level instructions (e.g. python scripts) in flow rules, the low-level instructions.



Let’s do an example:



The SDN controller, having input flow size, latency requirements and source and destination as input, performs the routing, finding that the best path is the yellow one (it didn’t use Dijkstra clearly but some other algorithm). This path information is represented by a sequence of nodes (S1, S2, …). The applications, written in an high-language, are translated in flow rules applied to the nodes of the path S1, S2, …, S4. For example the flow rule applied to S1 can be:

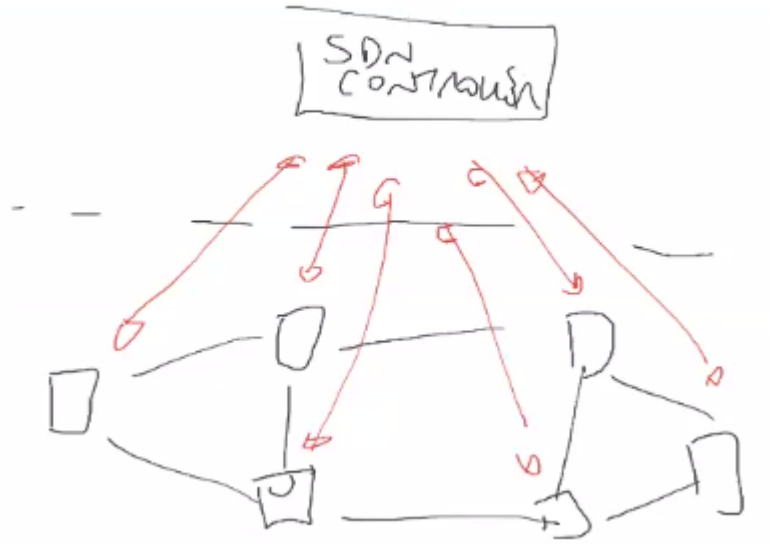
MATCH: SRC_IP == H1_IP
DST_IP == H2_IP

ACTION: OUT -> 2
MODIFY_SRC_IP -> A.B.C.D

so S1 is behaving like a NAT.

Multiple ways of creating a SDN Controller

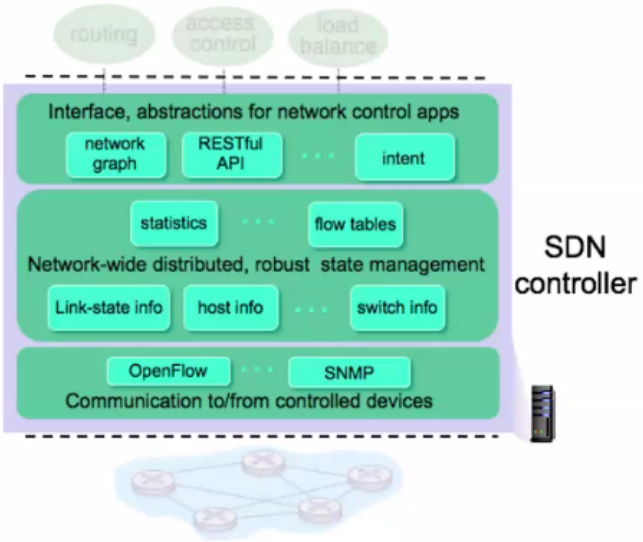
There are multiple ways of creating an SDN controller:



- 1:1 mapping: 1 physical machine that runs the SDN controller
- multiple physical machine and one distributed instance of the SDN controller, which runs on top of these machines
- multiple physical machines and each one is running an instance of the controller, maybe one is the main and the other are backup

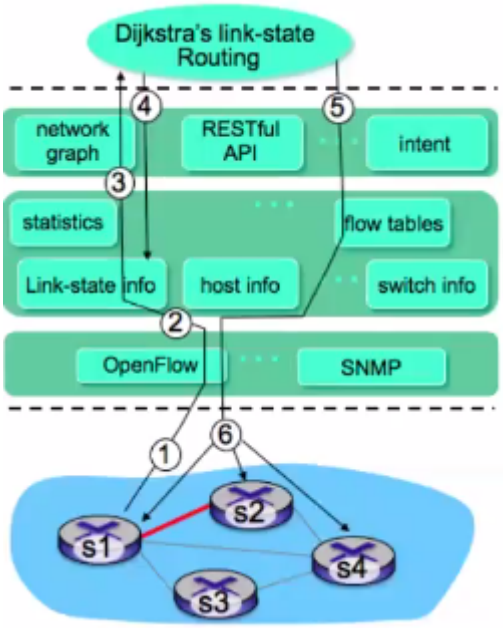
Components of a SDN Controller: NB, Core, SB

Components of SDN controller. This is just an example:



- Communication layer: the southbound interface
- The core part is the network wide state management layers. Is a set of databases with the information about the network status. E.g. we have a table for the network links.
- Interface layer of network control applications (north-bound interface), the API we expose to the applications. For instance if you want to create a routing application with this layer you can abstract the whole network information with a network graph. Intent is a high-level objective expressed in natural language (e.g. I want to connect H1 with H2 with at least 1 Mbps bandwidth), the controller translates this intent in a set of flow rules to install in the devices.

Example of a possible interaction between SB, Core and NB:



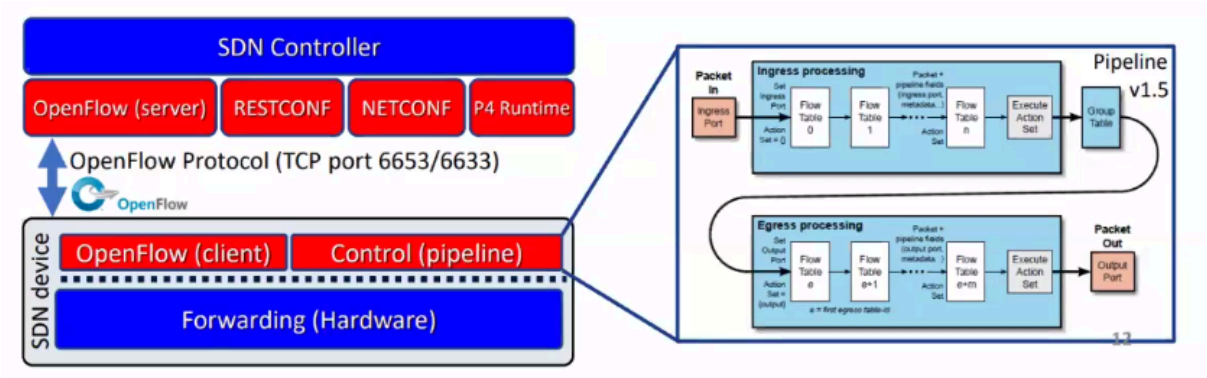
The application is “Dijkstra’s link-state Routing” and performs failure bypass. If there is a failure, the switch notifies the controller through the SB interface. The controller will update the link-state info table in the Core part of the controller. This will expose through the NB interface a different network graph to the application, the app will see that the net graph has changed and will recompute some network parts. After that the new parts are flow rules installed in the devices (point 6). It takes time to react but the traffic is now redirected.

OpenFlow:

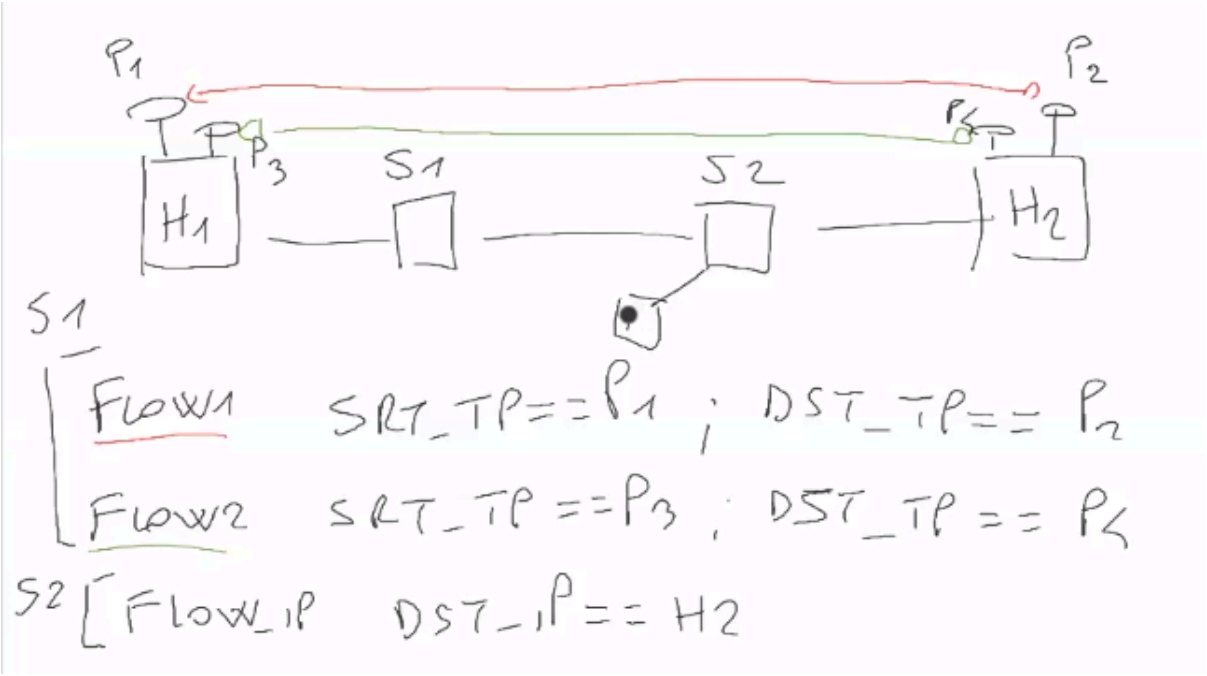
OpenFlow basics:

OpenFlow proposes 2 elements:

- Communication Protocol: southbound interface that connects the switches to the controller. The messages are exchanged inside TCP connections with common port numbers 6653 and 6633.
- Pipeline



How to merge two flows? This could be a solution. S2 merges the two flows that were different in S1



The flow based is really flexible, but the number of flow rules is huge. So for this sometimes it is useful to aggregate flows in a single one. Trade-off between resources and flexibility, the more I aggregate the more I lose performances but I free space in the TCAMs, that are very powerful for the lookup but have just few MBs of memory

In OpenFlow we allow multiple flow tables that operate in a pipeline fashion. We also have:

- Group Tables. Instead of specifying the action for each flow rule, we specify a group_id. Then we have a group table that associates to each group_id the corresponding actions.

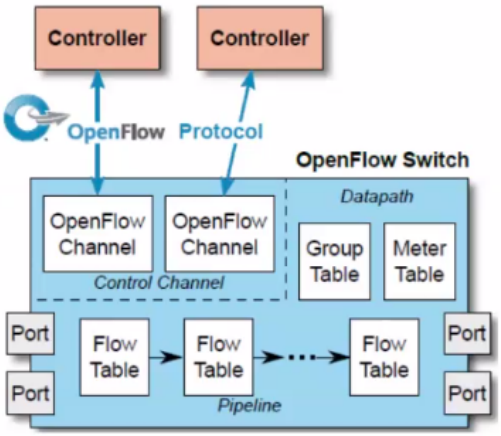
Why is this useful? Because if the link fails:

to find an alternative path we have to change the output interface from 1 to 2. However, we can have tens or thousands of rules to update, and that's costful in terms of time. If we just modify the entry in the group table (so we use the same group_id but we change the corresponding action in the group table).

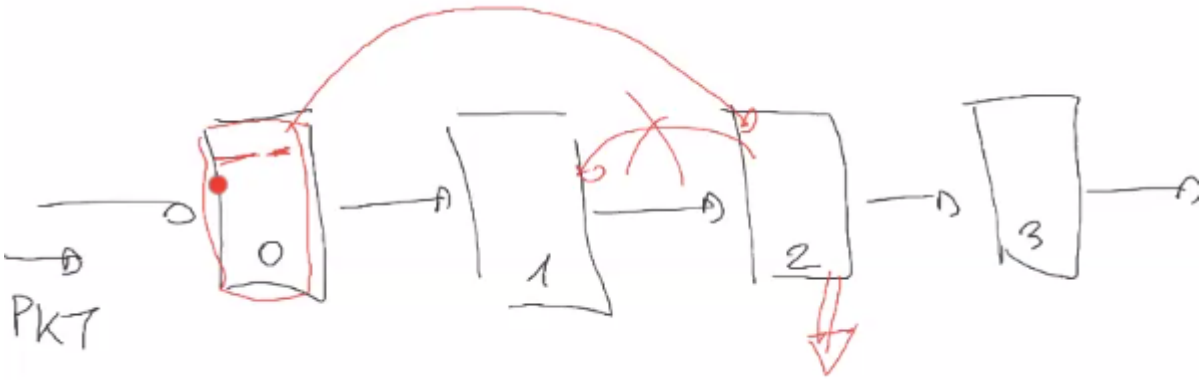
- Meter Table: like traps, these tables trigger a variety of performance-related actions on a flow of packets. Meters enable OF to implement QoS operations.
    1. Meter identifier for the Meter
    2. Meter Band specifies the (lowest) packet rate at which the band applies and the way packets should be processed. If the current rate of packets exceeds the rate of multiple bands, the highest rate is used.
    3. Counter is updated when the packet are processed by the meter

Meter Identifier	Meter Bands	Counters

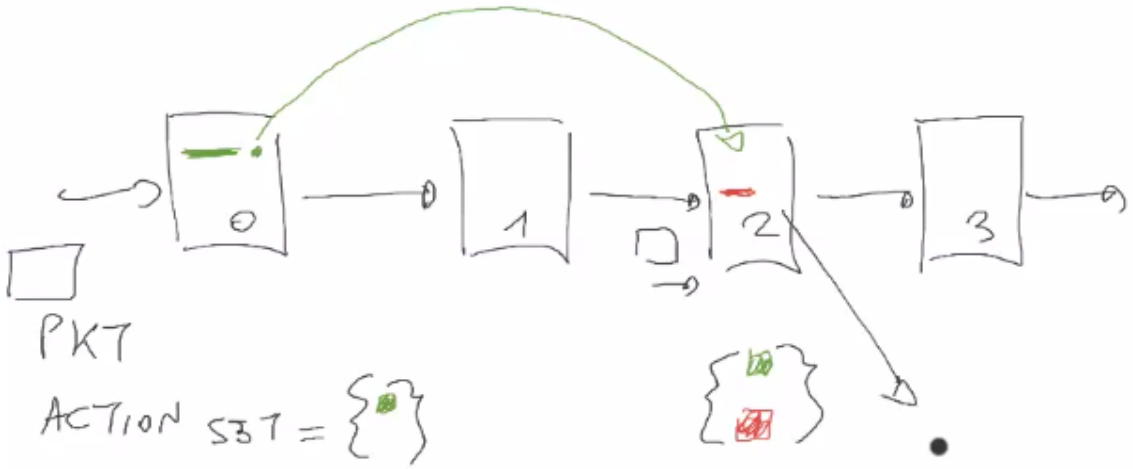
This is the structure of a OpenFlow switch:



There is a pipeline of flow tables that uses the match-action paradigm. The table 0 is always visited, then we can jump to table 2 for instance (no need to pass to all the tables), but of course we can't go back. We can leave the pipeline at any point.



First, the action set is empty, after entering the table 0, the first action (in green) is put in the action set, but is not applied immediately. Why? If there is a rule that modifies a field, it will be modified only at the end, where the pipeline is left. This makes sense: first the next state can change the action to apply if you modify the field in the previous one and moreover imagine the action is "send the packet out to port 3 and jump to table 2", how is that possible? If I have to go forward I can't jump to table 2 too. Let's assume we go to table 2 and another action (in red) is added:

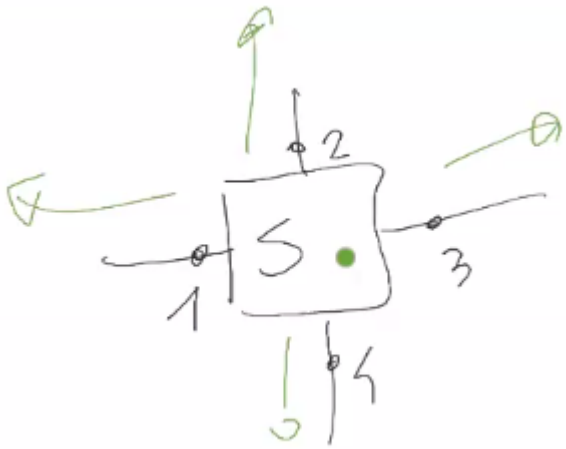


Now let's assume the packet leaves the device. The actions are now applied.

OF Ports

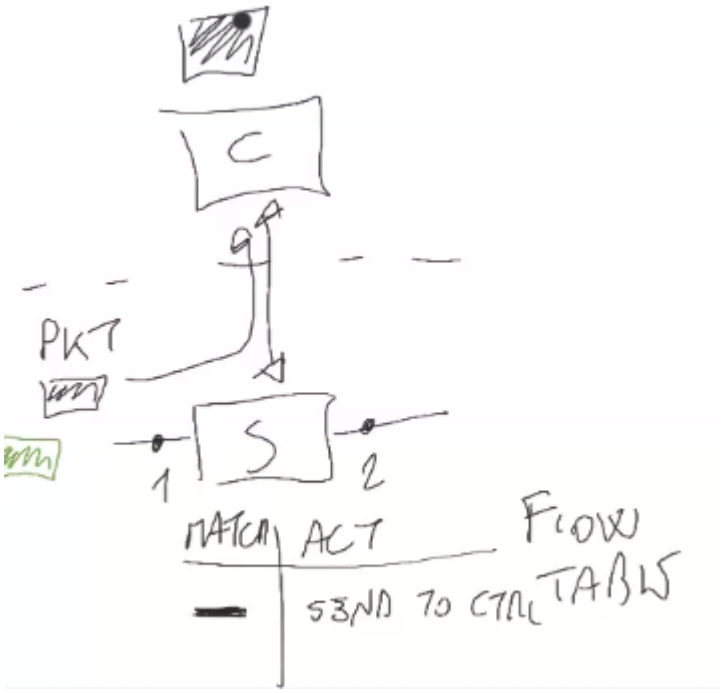
A switch has multiple ports. All the ports that we see are physical ports. Actually there are more than the physical:

- Logical: if you have to configure a VPN there is a logical interface (e.g. ifconfig after configuring a VPN will show a new port)
- Reserved: needed to accomplish tasks of the OF protocol. Examples of reserved ports:
    a. ALL: if you want to broadcast a message:



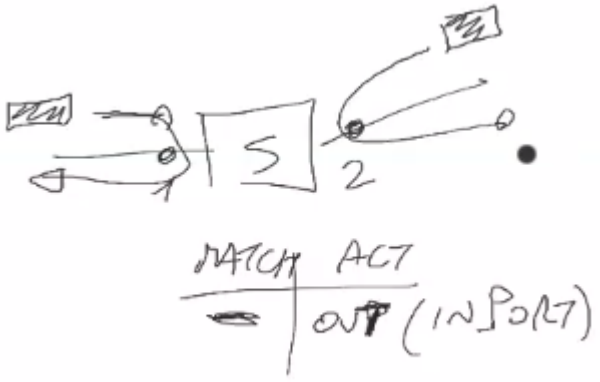
    b. CONTROLLER: For the black packet there is a rule for the green not. We may want to send the packet to the controller with the action SEND TO THE CONTROLLER. It is useful because the controller may have some useful functionalities. e.g. app responder: the app response to arp request. So

whenever you see an arp request SEND TO THE CONTROLLER.



Second possibility: further inspection, security reason. Sometimes the switch receives the packet and doesn't know how to process it because there is no rule for it. An IP packet will drop it. A OF switch can redirect the packet to the controller, this is the second reason. The default rule has as action SEND TO THE CONTROLLER and has, of course the lowest priority.

c. TABLE: represents the start of the OF Pipeline, so it submits the packet to the first flow table
d. IN_PORT: if the packet is black send out to inport



This is a reflector

e. ANY: perform anycast. It is useful when the flow rules applied don't specify the out port.
f. LOCAL: loop interface

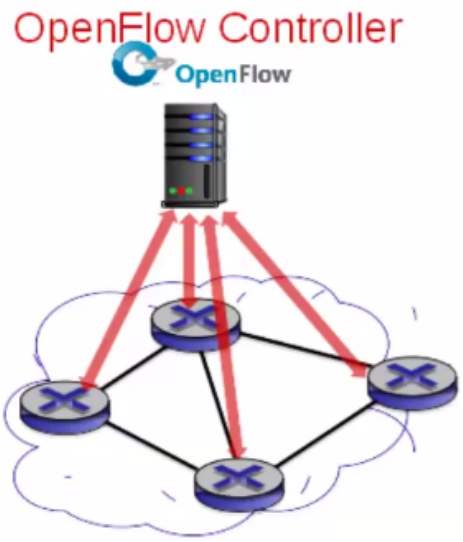
g. NORMAL: a switch generally implements more than a pipeline. OF pipeline and NORMAL pipeline. The NORMAL pipeline is the IP pipeline for a router. Imagine to have a router: surely it has an IP pipeline with LPM + it can support OF pipeline. The first pipeline that is visited is the OF. Since it's a heavier process, maybe there is no need for that and we can use the normal pipeline. So a device can perform multiple types of processing.
h. FLOOD: send the packet out of all the ports except the one from which we have received the packet.

OF Messages

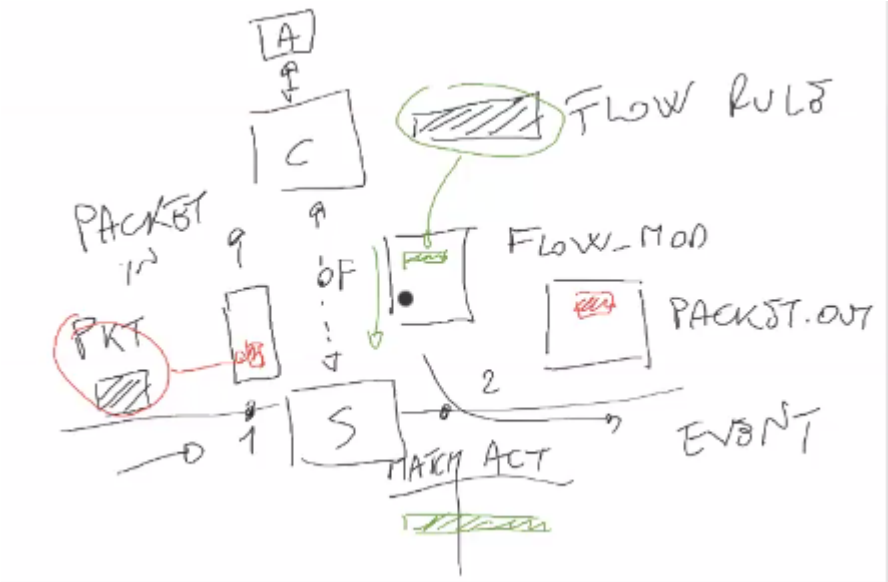
Messages used for the OF communication protocol. The message are:

- Controller to Switch (asymmetric, synchronous):
    - packet_out: used to inject packets in the data plane.
    - flow mode (or modify-state), used to modify, install or delete rules in the flow rules
    - feature: the controller queries the features (the basic capabilities) of the switch.
    - configure: modifica la configurazione (guarda slides)
- Switch to Controller (asymmetric, asynchronous). They are not sent every time period but when an event occurs, e.g. flow-rule missing event.
    - packet_in is an asymmetric message.
    - flow-removed: whenever a time expires the rule is removed and the switch notifies the controller.
    - port-status, then client notify the controller that there has been a change on a port status
- Symmetric: hello messages exchanged to keep alive the connection between data and control plane

First thing is to establish a TCP connection with the controller.



Once the connection is established. The controller will ask the features, configuration. Sometimes it will install rules in the switches with modify-state
Since the table is empty the default action (SEND TO THE CONTROLLER) is performed. The fact that the table was empty is an event, and the SDN architecture reacts to it, by redirecting the packet to the controller. Specifically we create an OF message that encapsulates the packet. This message is called packet in. Is asynchronous from data to control. The controller delivers the packet to the app that performs its job and comes out with an action (e.g. the packet must be forwarded to port 2) so the controller creates the flow rule and installs it in the switch. For doing so the controller creates a flow_mod packet that encapsulates the flow rule. Another OF message, called packet out, will contain the packet that has to return in the data plane.



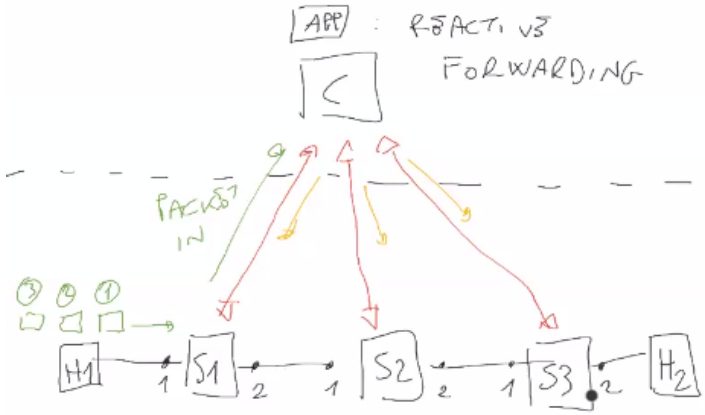
Another packet of the same flow will be now handled directly in the data plane, so we have to pay this overhead latency just one time.

Flow Routing vs Aggregation:

Flow-Based and Aggregated Forwarding: tradeoff between flexibility and cost, measured in terms of flow rules needed to complete the task. This cost is due to the small memory of TCAMs and also the latency of interactions between data and control plane. Using more group tables rules means complying more to an aggregation approach.

Reactive vs Proactive:

With a reactive approach the first packet of the flow triggers the controller. Let's consider the following case: there is an app called "Reactive Forwarding" and the flow tables are empty for now. At a given time H1 sends messages to S1 and S1 redirects the packet to the controller, since the event flow-rule missing happens. The app reactive forwarding will find an end-to-end path from the source to the destination. As a consequence the controller will send flow rules to all the switches (S1, S2, S3)



Practically, the flow rule can be something like this:
SRC_IP = 10.0.0.1
DST_IP = 10.0.0.2
ACTION = OUT(2)

Since it's a reactive application it will set a timeout (e.g. 10 seconds). All the switches will install this rule. In this way the packets will go to H2. Another examples of the application of the reactive approach is the Dijkstra's link-state Routing with bypass.

Advantages of reactive approach:

- we install rules when they are needed, so TCAMs will have only "useful" rules, since the timeout will delete the not used rules.

- It is useful also when the network environment changes frequently, for example:

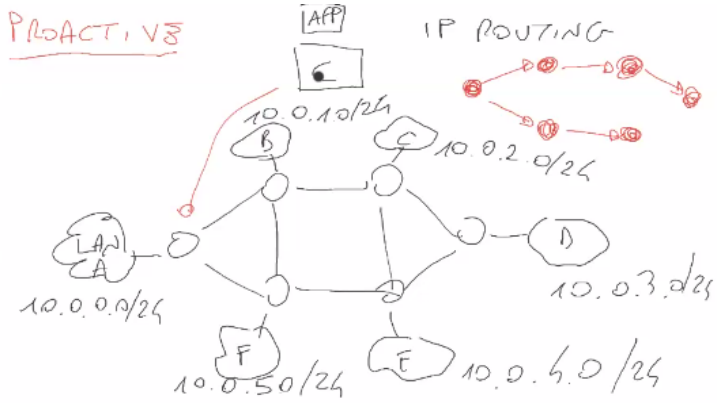
The app “minimize latency” wants to minimize the latency of the end-to-end path H1-H2. Using a proactive approach we create the path before the interactions between the H1 and H2, so the network conditions can change, leading to congestion. Using a reactive approach on the other hand, the controller is queried and then the path is computed in the moment of the interaction between H1 and H2

The blue path in the figure is the least congested in the moment of the interaction.

Shortcomings of reactive approach:

- we continuously have data and control plane interactions, and this is time consuming. For instance, if we have a really big network geographically speaking, the controller and the devices can be really distant, and so there is a lot to pay in terms of latency every time the data plane and the control plane interact.

Let’s do an example of the proactive approach:



We have an app “IP ROUTING” that computes the shortest path tree and install some flow rules in the switches. For the first router the flow table will be something like this:

MATCH | ACTION
DST_IP = 10.0.0.0/25 | out (3)
DST_IP = 10.0.1.0/25 | out (1)
DST_IP = 10.0.2.0/25 | out (1)
…

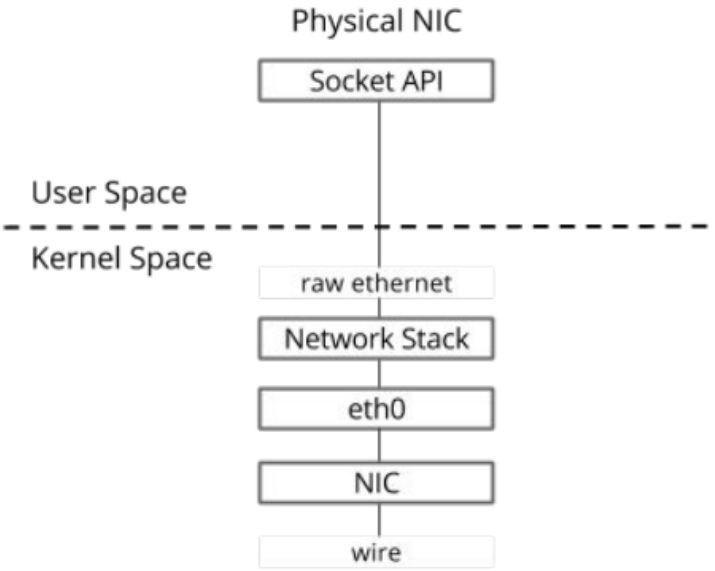
If I do that in all devices there is no way that flow rule missing events happen. So we reduce the number of interactions between control and data plane but the rules are not flexible.

A good trade-off can be to have some of the rules in a proactive approach and some in a reactive approach. For instance, traffic that has specific requirements on latency, bandwidth and so on. For them we want a reactive approach to always have the best path. Best effort traffic can be handled with a proactive approach.

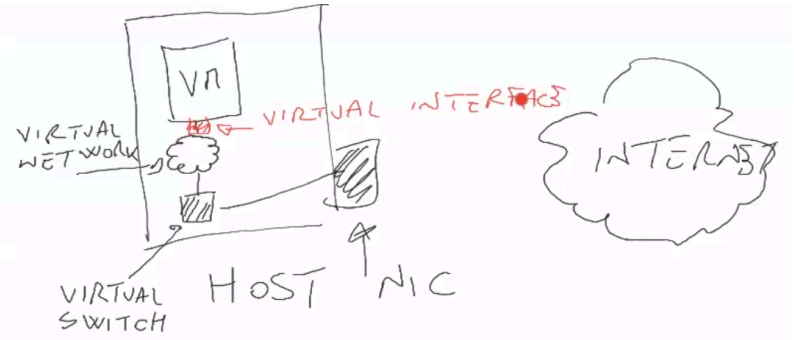
Link, Host, Switch and Controller Virtualization (SDN Lab 1)

Links

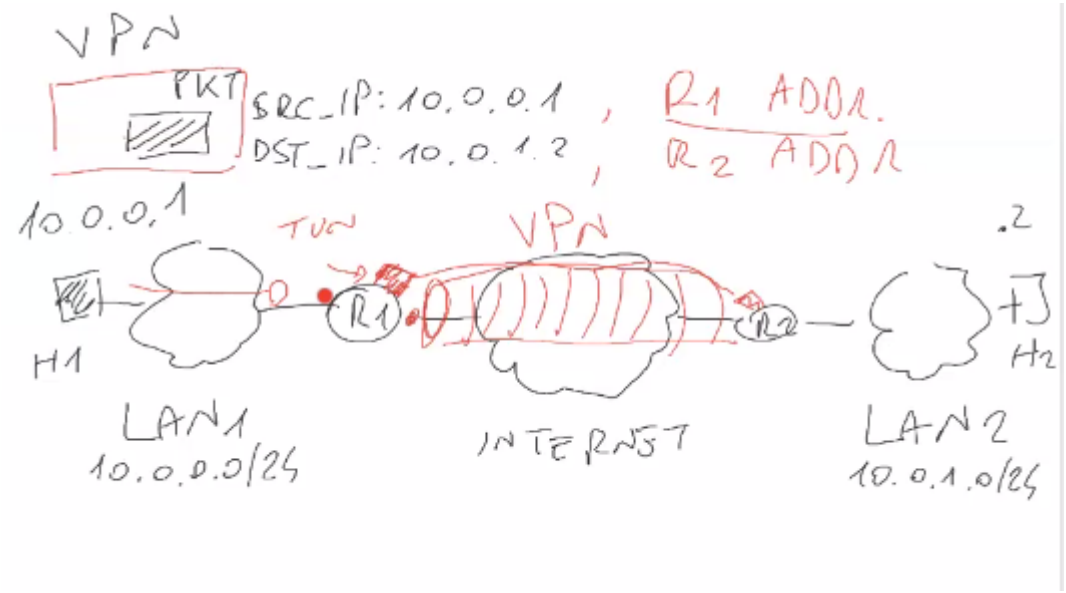
We have an app that is running in the user space and has to send messages outside the physical machine. The OS delivers the message to the network stack in the kernel space and finally the packet can be sent over the channel (wire or wireless)



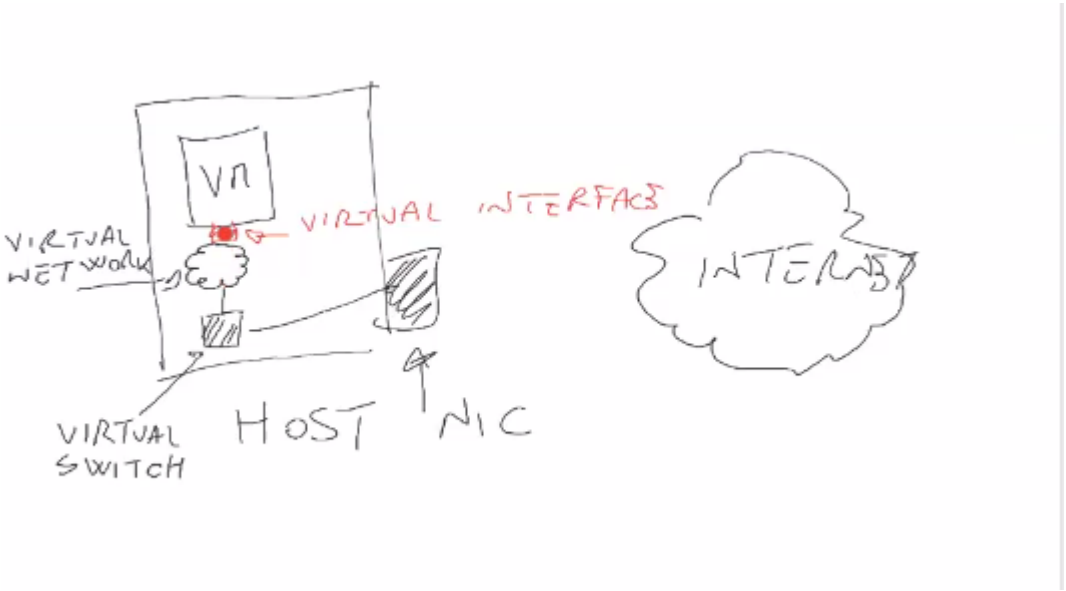
What about the communication inside the host? Let's consider the following case: a VM is run inside an host. The VM wants to send a message to the outside (Internet). For doing so it needs to use the NIC, but the NIC is of the host, so the VM doesn't have direct access to the NIC. We create a virtual network with a virtual interface for the VM and a virtual switch that is connected to the physical NIC.



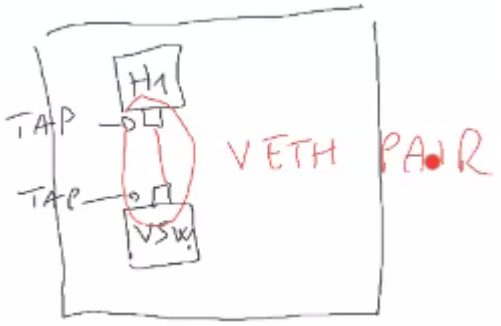
There are three types of virtual interface: TUN, TAP and VETH. TUN/TAP provides packet reception and transmission but at different points of the TCP/IP stack. TUN allows to create IP packets, TAP creates Ethernet frames. A VPN is basically the setup of a set of TUN interfaces:



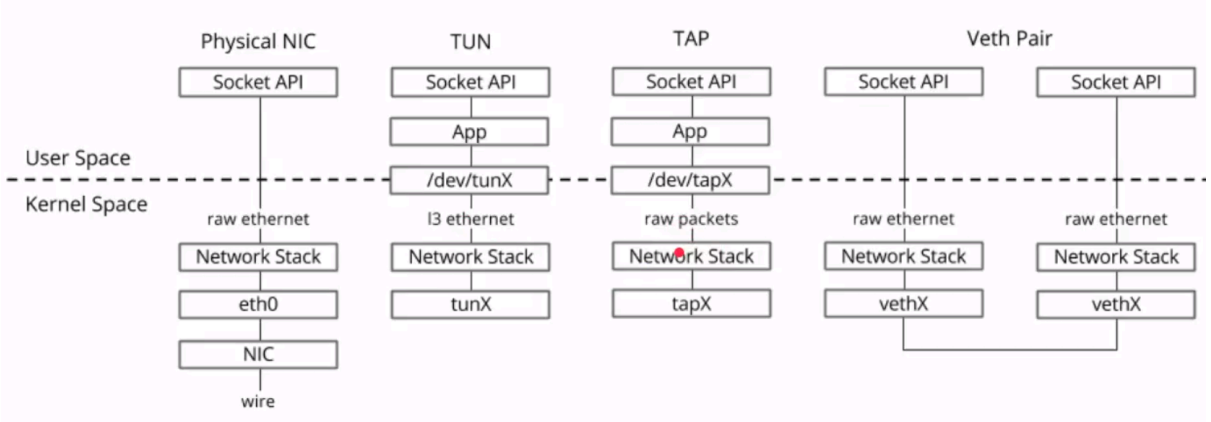
If H1 creates a packet for H2 the source IP will be 10.0.0.1 and DST_IP = 10.0.1.2. We need an encapsulation technique that takes the original packet and encapsulates it in a new packet with R1 address as source and R2 address as destination. The R1 address is assigned to the TUN interface, which creates this abstraction. But in our case:



we are not performing encapsulation, so we use a TAP interface, since we can only send Ethernet frames. The concept of a transmission for TUN/TAP, since no electromagnetic transmission is happening but we are in the same device, is just moving a packet from a location to memory to another location. So TX is writing in a file and RX is reading from the file. Often we will create point-to-point communication between an host and a virtual switch:



Host H1 and Virtual Switch VSw are point-to-point connected with a VEth pair (virtual ethernet cable). To summarize:

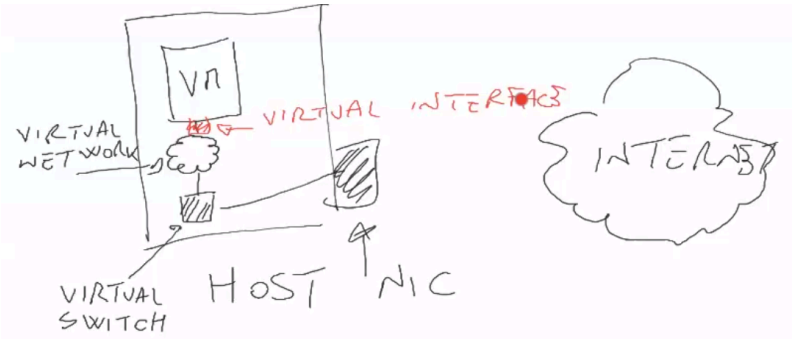


the first is the classical approach, where the host machine communicates with its NIC. With TUN we create an IP packet, with TAP we create a raw (Ethernet) packet. Veth pairs are just

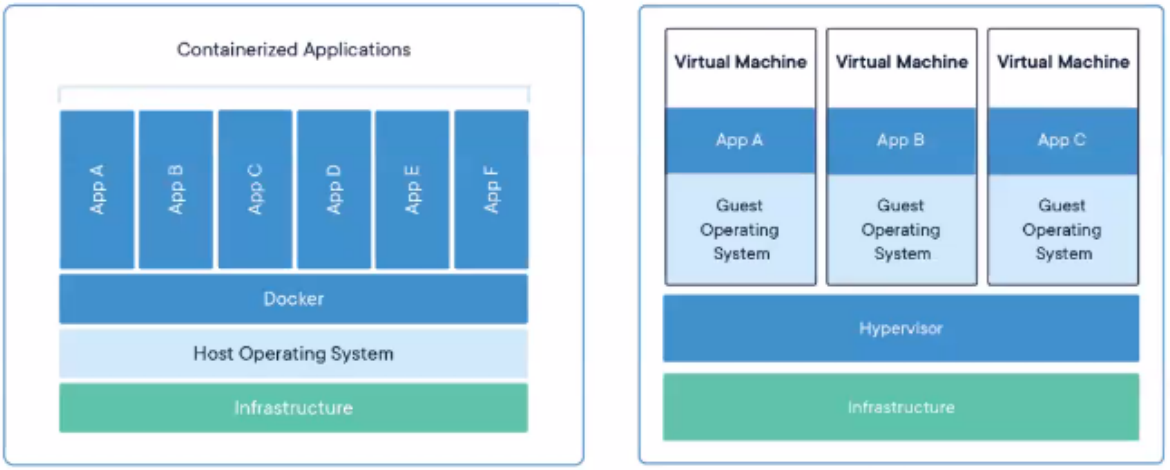
two TAP interfaces interconnected. What is transmitted from an end is directly sent in output to the other side. This makes veth pairs ideal for connecting virtual network components.

Host

Until now we have understood how to emulate links for a virtual network that is present inside a physical host.

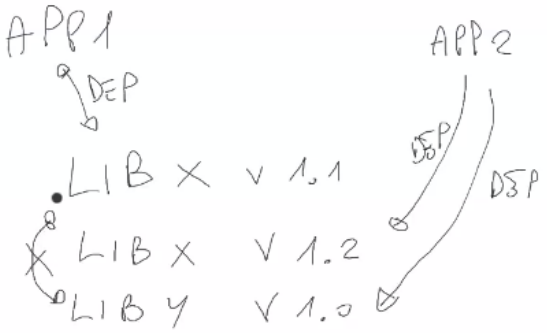


Now let's see how to emulate the hosts. Two approaches:

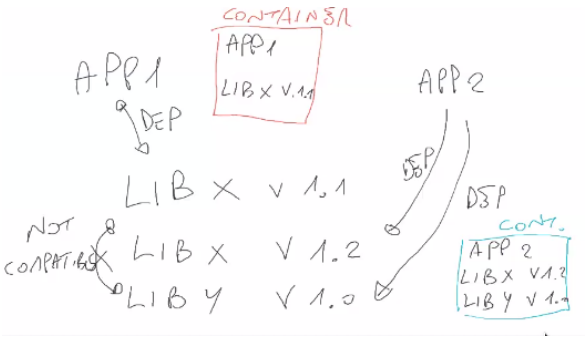


- Virtual Machines: we create a complete guest machine inside the host. The guest has its own applications and operating system. There is a software called Hypervisor that creates an abstraction of the physical resources of the host machine (CPU, RAM, storage, network interfaces and so on). We need to expose to the VM a VCPU (Virtual CPU), because we don't want the VM to use all the Physical CPU but just a portion of it; VRAM; Virtual interfaces and Virtual storage. This virtualization is done by a Hypervisor, such as VirtualBox. The VM are completely isolated environments: VM1 can't see processes running on VM2, unless you send a packet with this information over a virtual network. Quite expensive in terms of memory and resources.
- Containers: lightweight virtualization. An application needs libraries and dependencies to run, a container is a package that contains the applications and all

its dependencies. The applications are isolated if they are in different containers, so they can't see each other, unless we create a virtual network. This is useful if you have two applications that require two different versions of the same library or two libraries that are not compatible.

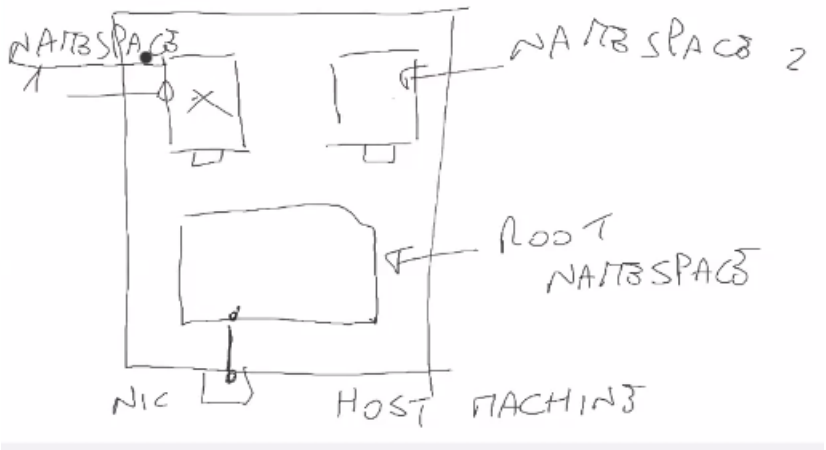


In the host you can't run both applications if you don't use two containers to isolate the two applications



Also for containers we need Hypervisors. A famous hypervisor it's Docker. This approach is lightweight since we are using the host OS but the level of isolation is smaller, so for security reasons the VM approach is better

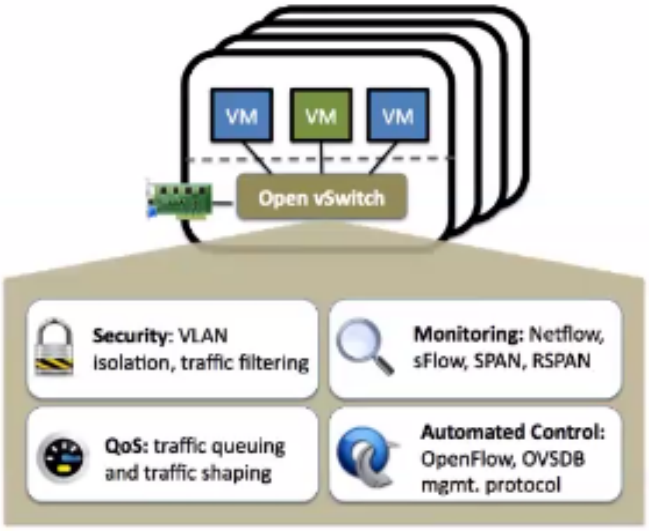
Namespace is a form of containerization. Let's consider our host machine, that has a Linux kernel with Root Namespace:



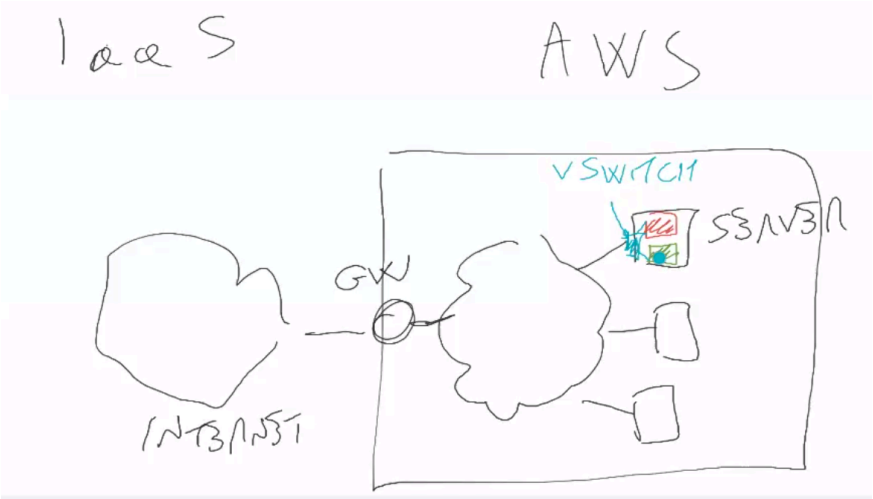
The Root Namespace is the only network namespace that can interact directly with the NIC. Then we can create isolated replicas of the root namespace (namespace 1 and 2 in the figure). If we enter the namespace 1 and type ifconfig we don't see the physical NIC since it's associated with the Root Namespace.

Switch: Open vSwitch

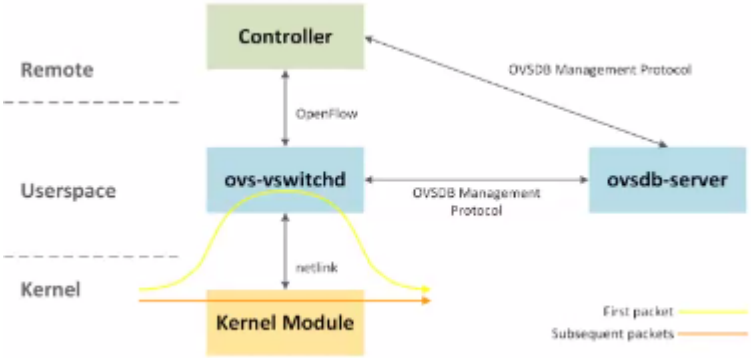
Now we know how to create links and hosts. How to create switches? We want them to support generalized forwarding. For this purpose we use Open vSwitch.



IaaS (Infrastructure as a Service): you pay, for instance AWS, to use resources. I ask Amazon to create a VM and the way Amazon answers is by creating virtual resources to one of its servers. Later on, another customer asks for resources, and they are allocated on the same server. We want to isolate the two machines since they are from different users and we want them to be reachable from the outside so we create a VSwitch to interconnect the VMs with the rest of the world.



Open vSwitch is quite used in datacenters. The Open vSwitch has a daemon in the user space (ovs-vswitchd) and the user has a CLI to interact with it. The controller has an OF interface with the daemon and there is also a kernel module in the kernel space to process the packets.

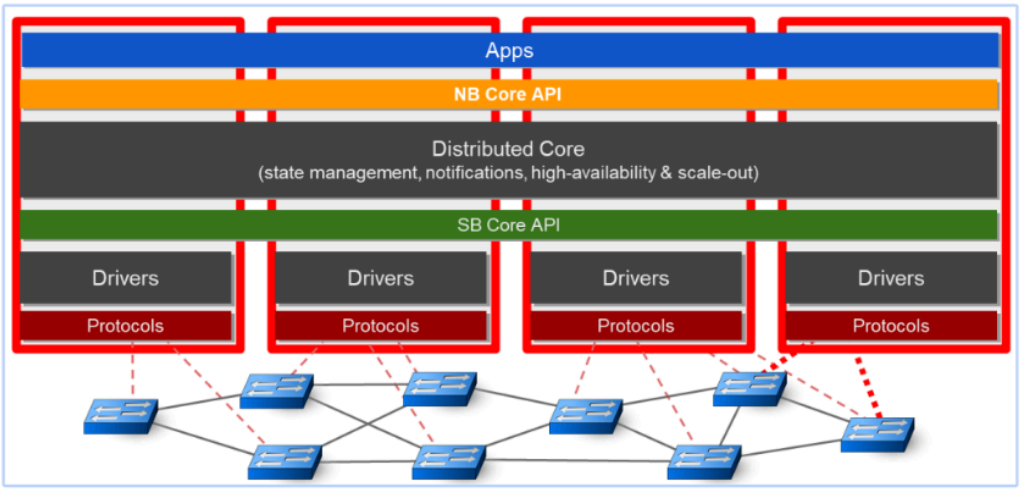


An important difference between user and kernel space is that if we process things in the kernel we are fast, while in the user space is much slower (slow path). The first packet of a flow passes on the slow path. We stay in the kernel space when the flow table knows the rule to process the packet..

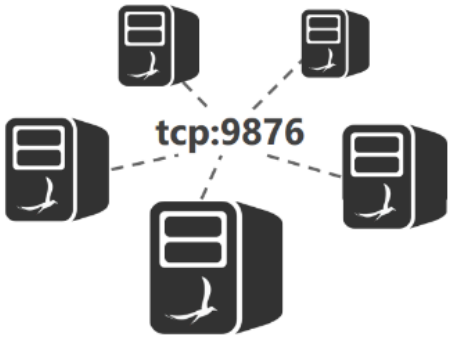
The ONOS Controller

We will consider the ONOS controller only for the beginning of our practical part. After that we will pass to a simpler controller with Kathará: POX. Open Network Operating System (ONOS) is an open source SDN network operating system. ONOS has a distributed architecture, so we can implement multiple instances of the SDN controller in different devices. In this way there is not just one single point of failure, and we have high availability

and high scalability.

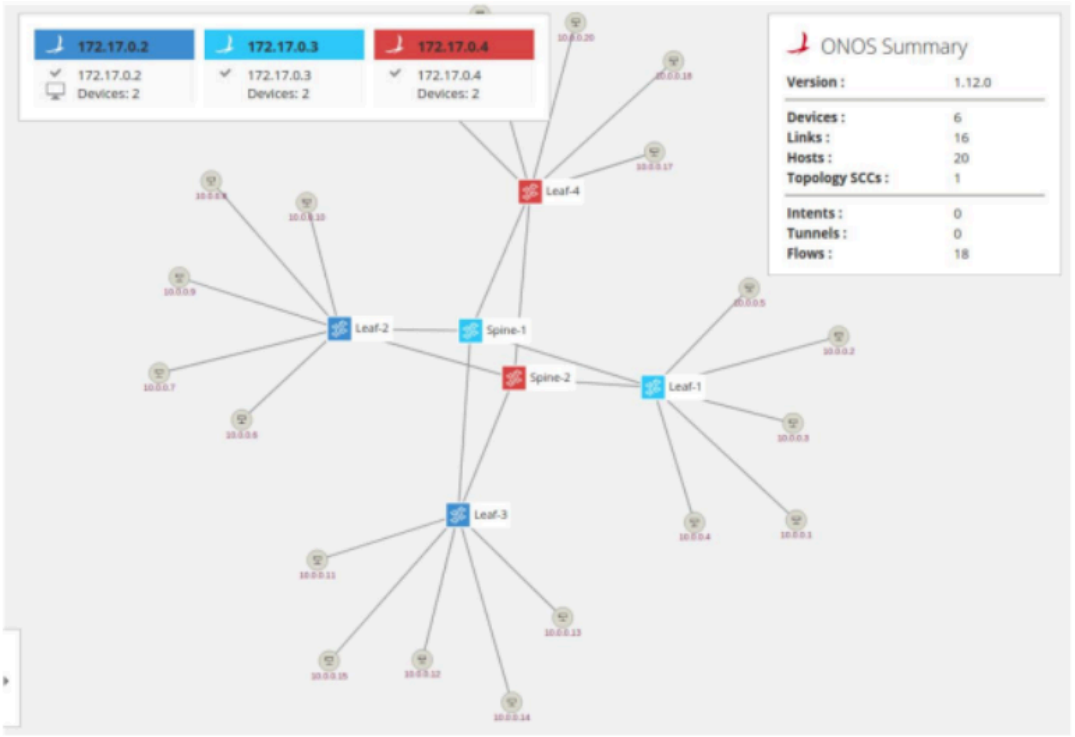


The red squares in the picture above represent different VM in which ONOS is running. These devices need to be synchronized with each other, so we establish TCP connections.

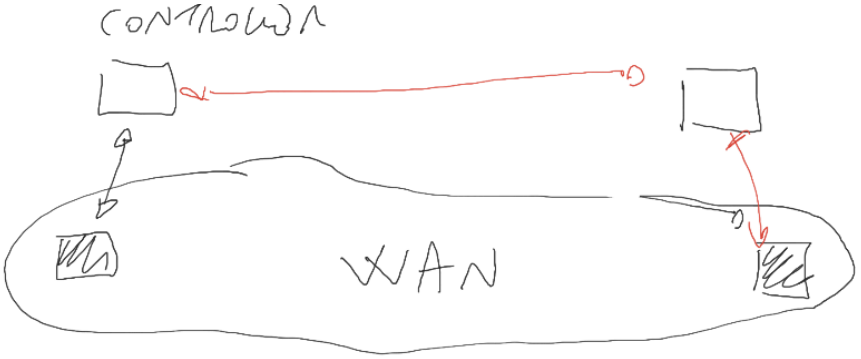


Distribution of the Workload

As you can see from the ONOS GUI below, ONOS distributed architecture supports the distribution of the workload**:** each of the three ONOS devices is controlling a different part of the network. So 0.2, 0.3 and 0.4 are getting assigned to different switches in order to split the workload:

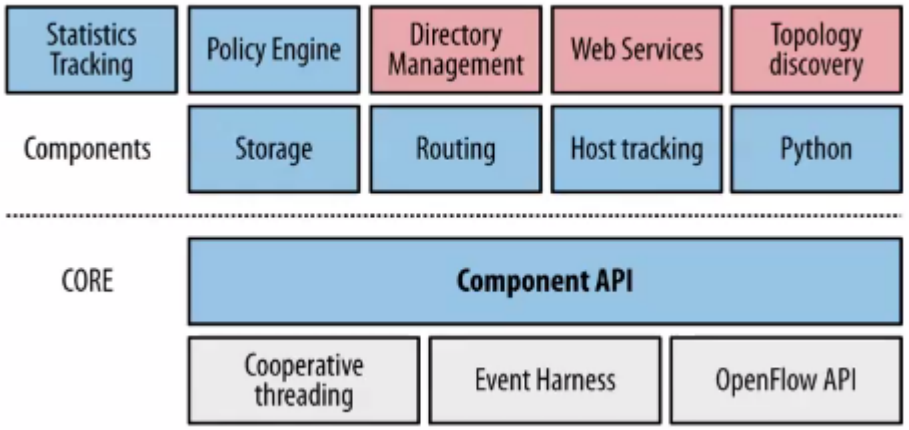


Note that hosts (the gray nodes) are not controlled by the controller of course, since the controllers control only network elements (switches). However they are visible in the GUI since ONOS implements host discovery. This distribution of the workload is particularly useful in a WAN for instance, where two switches can be really distant, and the best thing is to control them using two different controllers that are close to them. The controller will then communicate with each other on the control plane



The POX Controller (SDN Lab 2)

This is the architecture of the POX controller:



Components are POX jargon for applications. Differently from ONOS, where the core part contains databases, if you want to keep track of the status in POX you have to create a specific application. The core exposes APIs to the applications (component API). The APIs are event-based and this is important in a SDN because in a reactive approach we want to catch events, e.g. arrival of traffic flow, and react to them. For this, the core of the POX controller has to raise events that are listened to by components that react to them. The POX controller allows communication with the dataplane using OpenFlow API.

Run the POX controller

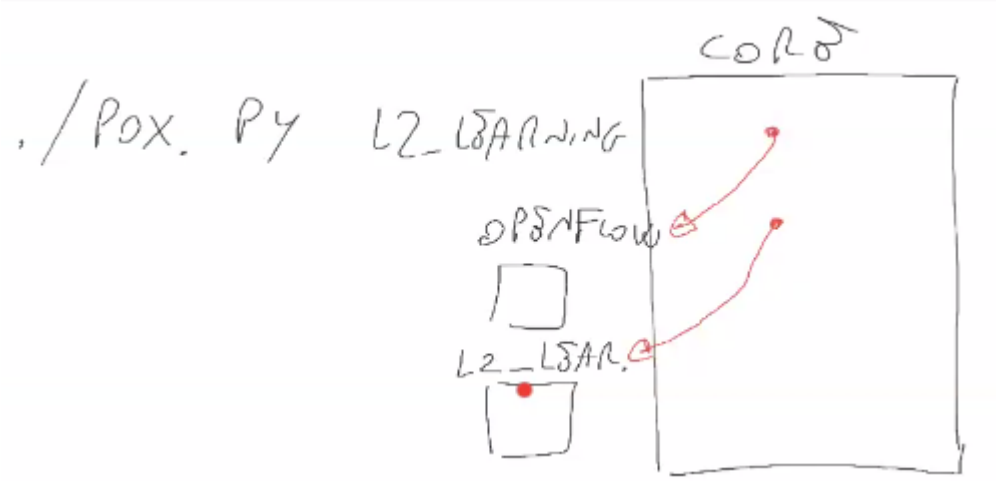
In the device that has it installed:

- in pox/pox/ you can find all the services that are natively available. For instance in pox/pox/lib you can find packets that contain scripts to create packets. For instance arp.py for creating arp messages.
- in pox/ext we deploy applications. For instance link_discovery_solution.py is a link discovery application to detect interconnection between elements.

If we want to start the controller and run an application we can run ./pox.py forwarding.l2_learning. This will run the controller and execute the Layer 2 learning application.

POX Core Object

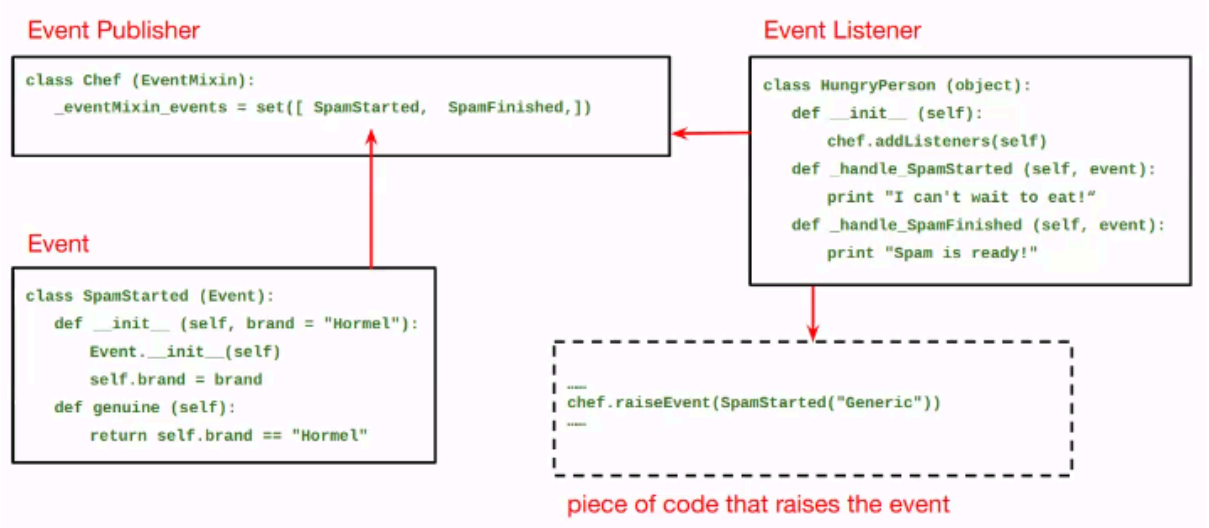
from pox.core import core is needed to use the core object in your applications. The core object is the first element executed when you run the pox controller and contains the reference to all the other objects instantiated after. For instance, when you run ./pox.py L2_learning you create the OpenFlow object, the L2_learning and so on and so forth. In the core object we have a pointer for each of these objects.



How can we register a new component on the core? This linking above is not automatic for new objects. So if I create a new object x in an app, if we want to make it accessible from the core we need to register x with core.register() or core.registerNew()

Events in POX

In our controller we can define custom events. Event handling in POX works with a publisher/subscriber paradigm. Metaphor: Polverini, the publisher, will tell us, the subscriber, where the grades from the exam are available on the website, without having us to poll on his website continuously. A publisher has to import and extend the EventMixin object.



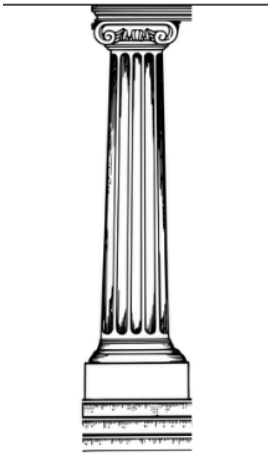
OpenFlow events have three main attributes:

attribute	type	description
`connection`	Connection	Connection to the relevant switch (e.g., which sent the message this event corresponds to)
`dpid`	long	Datapath ID of relevant switch (use `dpid_to_str( )` to format it for display)
`ofp`	ofp_header subclass	OpenFlow message object that caused this event

1. connection provides a pointer to the connection object from which the OF packet_in message was received
2. dpid (Data Packet IDentifier) that identifies the switch
3. ofp changing depending on the specific event we are dealing with (e.g. if the event is packet_in we can find the message itself in ofp). So this attribute changes if a new event is raised.

Network Softwarization

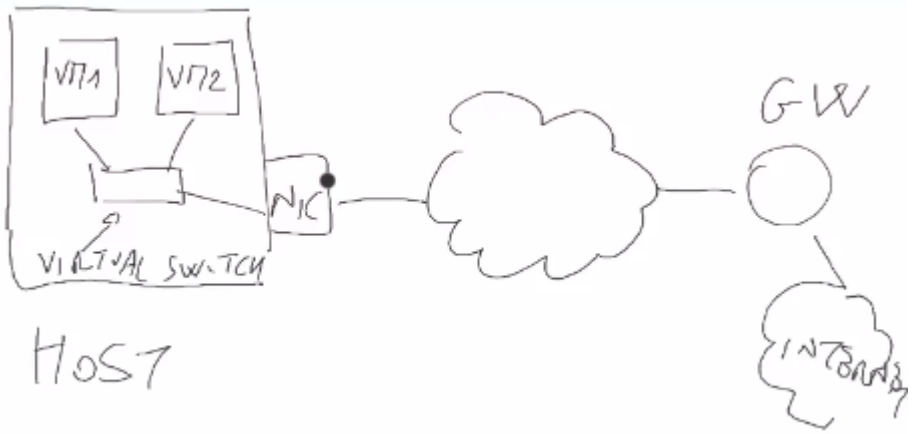
21/03/2024, 27/03/2024, 4/04/2024, 17/04/2024, 24/04/2024



Network Softwarization

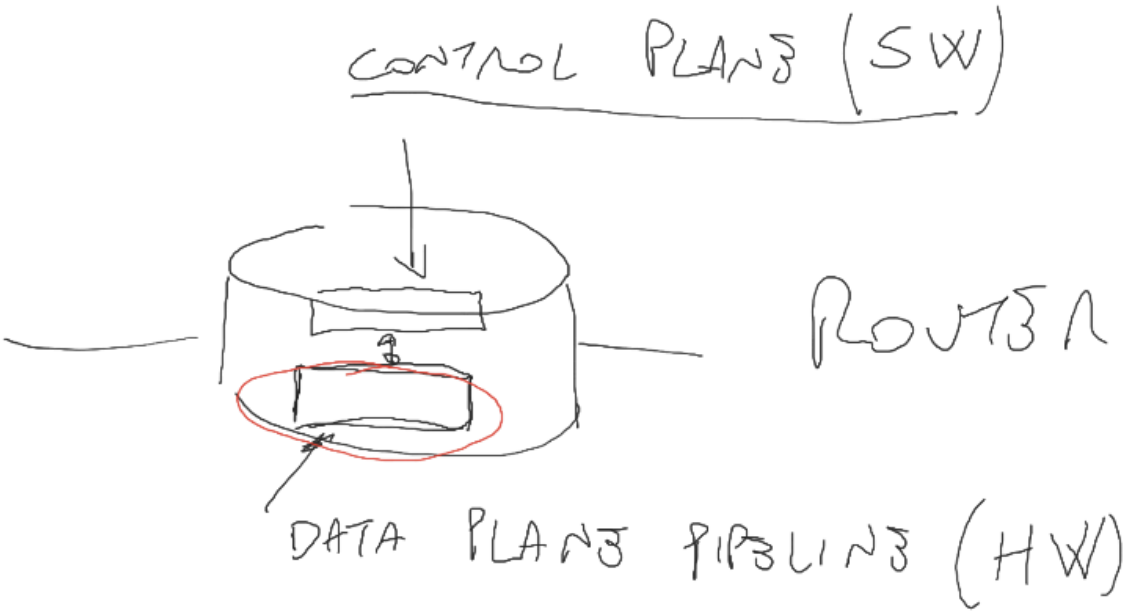
Networks nowadays are hardware based devices when there is software running. In the context of Network Softwarization we want an hybrid architecture: some functions will be hardware based and some software based.

A host has a connection with the Internet through a NIC. Inside the host there are two VMs and they are connected through TAP interfaces and virtual switch to the external world. This is a hybrid approach. Actually this is not exactly a softwarized network since the VMs are the end users: when we virtualize the end systems we are not talking about network softwarization, since the network functionalities are the target of the network softwarization.



A target can be, for instance, the virtual switch, that is a middle box that is in between the communication between the end users. For this reason we will talk about Virtual Network Function (VNF). We will refer to devices like the GW, that is physical, as Physical Network Function (PNF).

How can we create a VNF? Consider a router, an example of a middle box. We know that it processes packets and has a router engine. The part that processes the packets is the dataplane pipeline, while the other part is the control plane. The dataplane is realized in HW while the control plane is in SW. By decoupling we have defined the SDN paradigm.

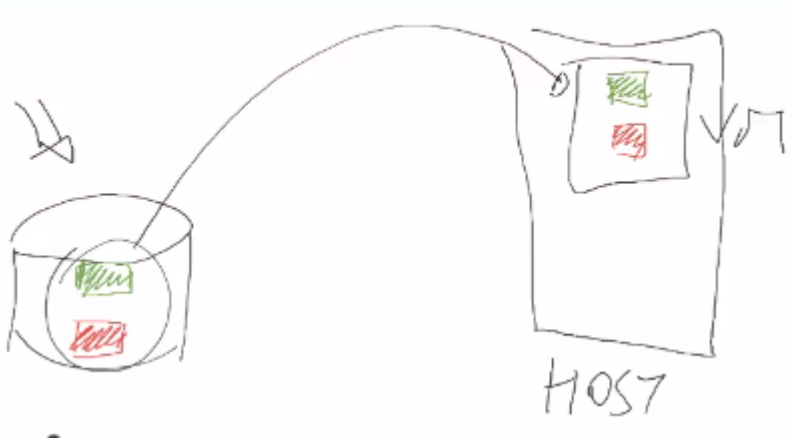


In the VNF we want to decouple as in SDN, but it's another type of decoupling: decoupling of HW from SW. Let's consider the case of the router: we already know how to run its control plane virtually in a general-purpose machine (e.g. CISCO csr1000 that we run on a VM), and it's really easy since it's already software. When it comes to the data plane pipeline, that is

HW, we know that if we consider the IP router processing pipeline we have to perform the following tasks:

- Decrement TTL
- Look up the routing table
- Compute the checksum
- Send the packet out

Can we define a program in a general-purpose machine that does these things? Yes, it's not difficult. We understood that we can run both the data plane and the control plane in a general-purpose machine, so we can move those functionalities in a server.

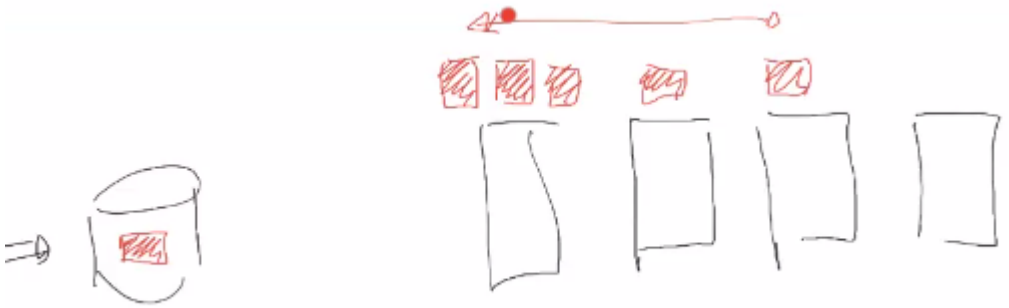


We have successfully decoupled the SW from the HW since we abstract what the router does, executing its functionalities in a general purpose machine. It's important to note that the router is based on an ASIC (Application Specific Integrated Circuit), which is a dedicated chip to process very rapidly; while the host has a classical CPU. ASIC can't do anything else outside of what it was programmed for. To a CPU you can ask whatever you want but you are losing in performance.

Benefits of Virtualizing the Middle Boxes

What is the advantage of moving the functionalities in a general purpose machine?

1. Flexibility (horizontal scaling) for matching the demand: imagine having many servers and a router whose functionalities we want to abstract. At a given point the user demand increases, in the physical domain we can't give more power to the machine, since it has its own physical resources, while in a general purpose machine we can do horizontal scaling: we launch other instances of the same functionality. If a machine is full, we can create instances in the other servers.

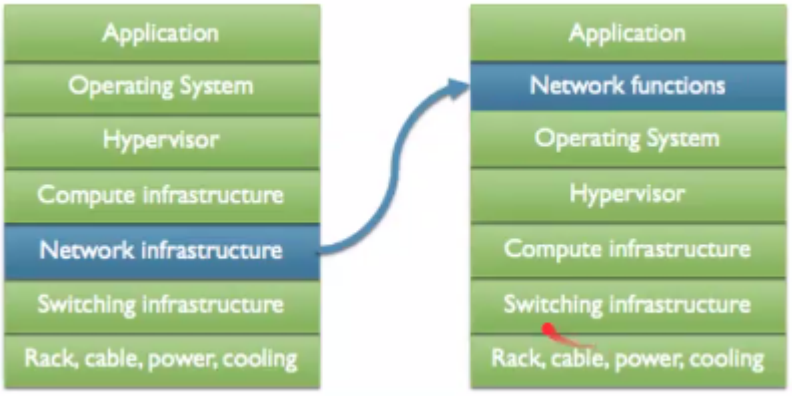


The cost of deploying the infrastructure can be adapted: few users? Just a single machine and I am not wasting a lot of money to keep all the servers running. If my business grows I can execute more VNF.

2. In case we have failures we can restore easily by running new instances, while if the router fails we have to fix or buy a new router hardware.
3. The router can cost quite a lot of money (tens of thousands of euro). On the contrary an virtual image of a router costs less (although an image can cost thousands of euro), or even if it costs the same you can run multiple instances. Moreover, you have to hire technicians, rent a building and pay for electricity. So the costs are dramatically reduced.
4. To create a new network infrastructure is as fast as running a new virtual machine, while it's not so easy with a physical infrastructure of routers.

A Huge Paradigm Shift

We are moving the network infrastructure in servers in datacenters.



As we go from bottom to up we move toward the core of the network:

The connection between the lower and the upper layers are PoP (or central office), buildings in which the ISP keep all their devices, and also servers. We can consider this PoP as datacenters.

NFVIaaS (Use Case #1)

Infrastructure as a Service is borrowed from the Cloud Computing field. We have:

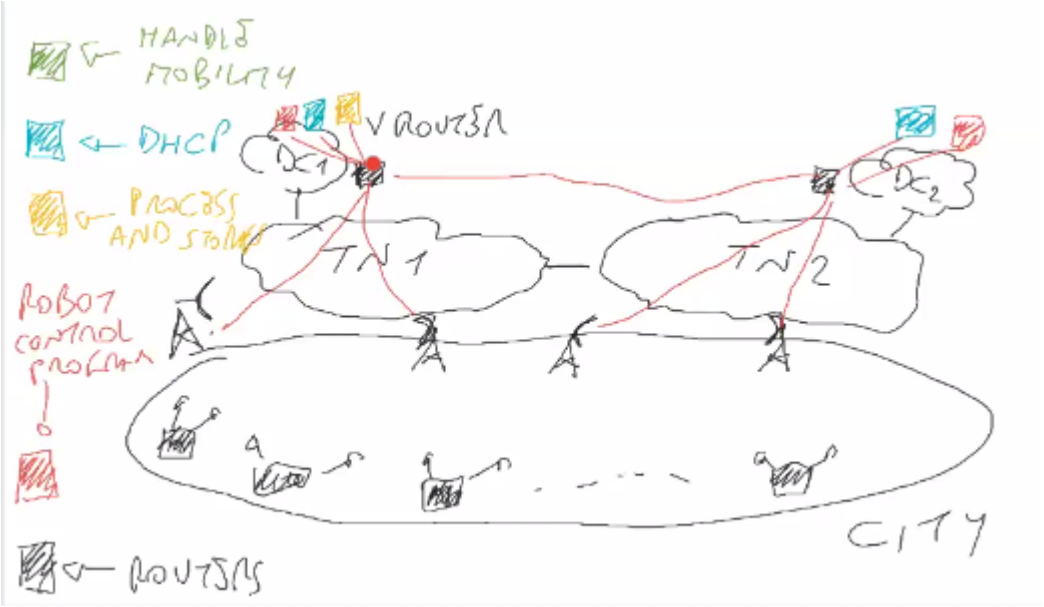
1. On-site: in a local implementation you have to take care of everything.
2. IaaS: I need computational power without having to buy servers because I want to use this power only for a limited time. Think about AWS: you can ask them for a VM with this type of CPU, this amount of memory, storage and so on. You pay a rent to have this VM dedicated to you. The provider gives to you the physical infrastructure, (in red) and then on top of this you manage your business, starting from the OS up to your applications (blue part).
3. PaaS: maybe you just want to create a service without having to deal with OS and other aspects not related to the service. E.g. you are building a social network, a website, an e-commerce, you want the platform to instantiate your service with. Kubernetes is a way of orchestrating your services, so it gives you the platform.
4. SaaS: the cloud provider gives to you the software, e.g. I am using google drive to see the slides. So I am a user that is using the software.

The difference is in what we manage.

In the networking area, imagine the following scenario: your organization has a branch in Rome and another in Milan. In the middle there is a transport network that interconnects the two branches. The customer (Rome and Milan) is buying the virtual connection between the two end points. If you want you can see that as NaaS (Network as a Service). Merging IaaS and NaaS: NFVIaaS. A new form of business: TIM has its own transport network, so it can sell us NaaS + it has a set of datacenter, so computational resources it can sell us (IaaS). So if we want to create our softwarized network we ask TIM for both things to create our services. Let's make an example: consider a city in which we have two ISPs, with their TNs and their DCs. I have a lot of money and I want to create a new business. E.g. I bought robots that move around the city and gather data. Let's say they are piloted by an AI algorithm remotely, deployed in a software, the Robot Control Program. The next step is to install antennas in the city to provide wireless connectivity to these robots. I also need routers. It could be useful to have:

- a function that handle mobility
- a DHCP server function
- a storage and data processing function

All these elements can be virtualized. For instance, I can have some virtual routers in DC1 and DC2. These virtual routers will be connected to the physical antennas. The red connection between the virtual routers and the antennas can be bought from the ISP too. Then we can create the services instances in the ISP DCs



Why are the same services running in two different DCs? To reduce the latency: I place the functionality as close as possible to the robot. This is an example of a softwarized network.

Multi-domain environments: we are buying services from multiple providers. Maybe a provider doesn't have the right or enough infrastructure to sustain my needs.

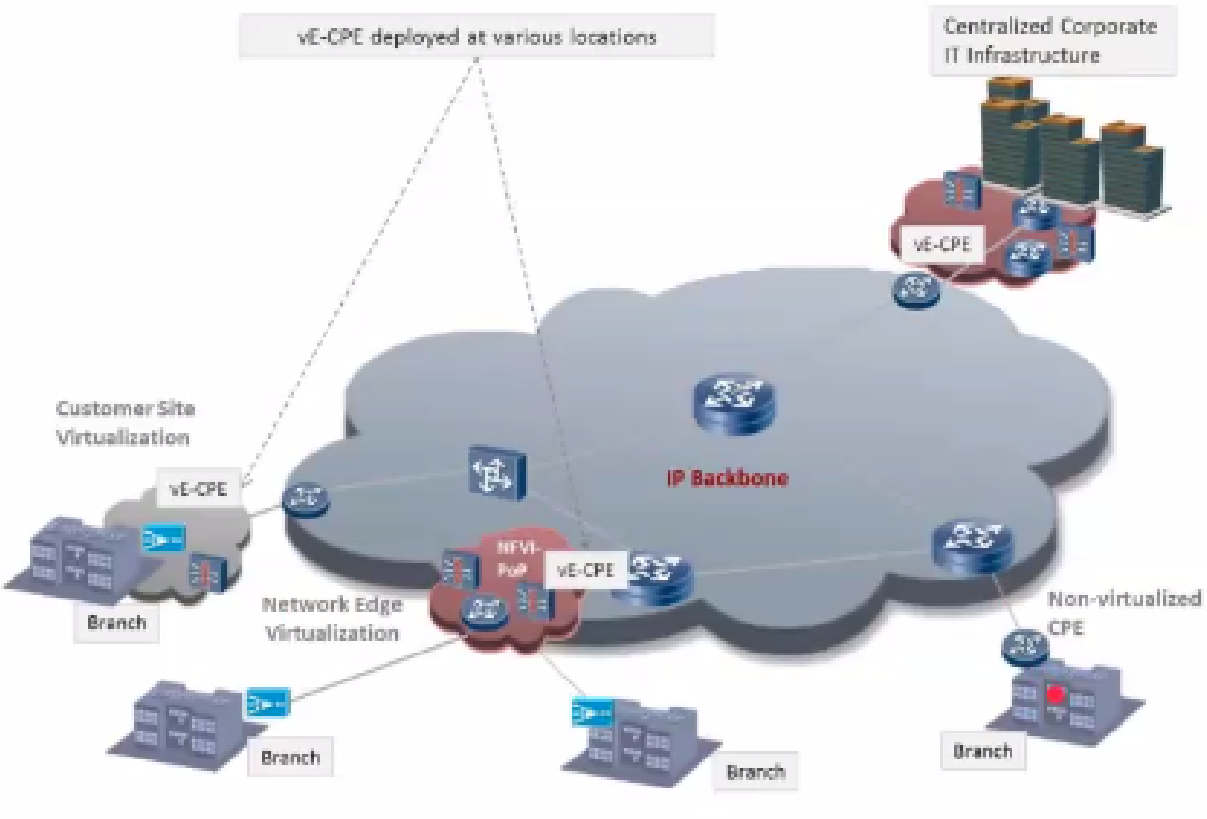
VNFaas (Use Case #2)

You can see this as an extension of the Software as a Service**.** Now with VNFaaS we are providing a specific function to a customer

Figure 6.2.2-1: Service Provider without virtualisation of the enterprise

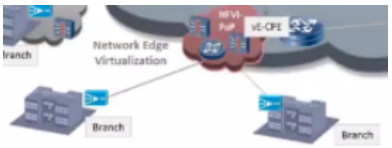
The service provider has its own infrastructure, and the customers need to buy, install, manage and operate the customer equipment (CE): the router installed in the company branches. From a company perspective, if I have a device, I need to buy it and pay operators. If we want to start services the company has to configure it by themself. So what the provider can do is provide a virtual image of the CPE (Customer Premise Equipment)

that runs not necessarily in the CE (Customer Equipment) but in cloud provided by the service provider.

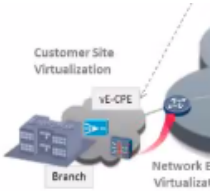


sometimes we don’t have virtualization at all, as in this case:

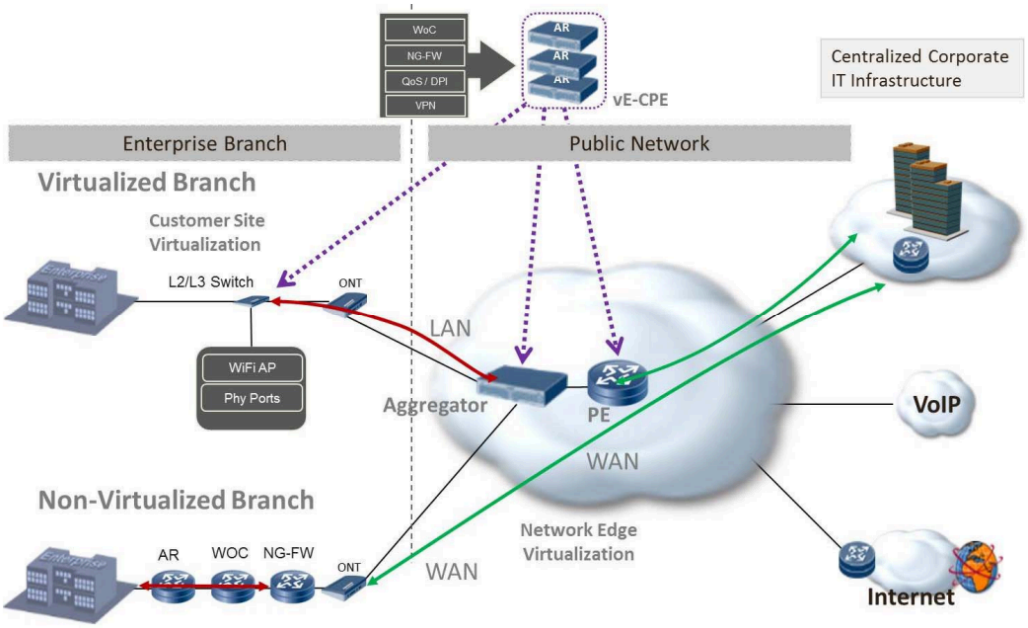
sometimes we run the CPE as a virtual image in a NFVI-PoP



some other time the virtualization can happen in servers that belong to the client

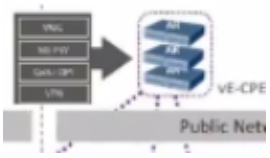


The vE-CPE performs routing (forward the packet from local to Internet and vice versa), VPN termination (VPN from a branch to, let’s say, the HQ of the company), QoS support (real time traffic), DPI, NG-FW (firewall), WOC (TCP accelerator, to boost performance).

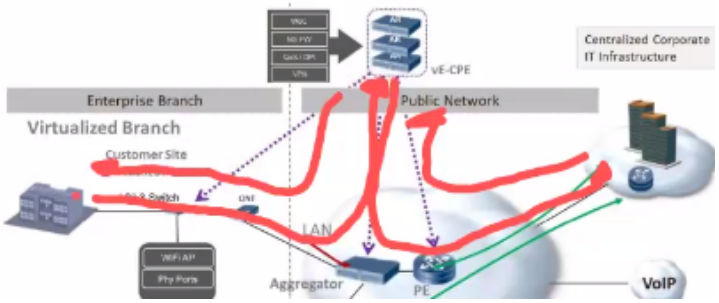


If those functions are deployed physically in the branch they are positioned in the perimeter between the branch (the local network) and the rest of the infrastructure. Downstream and Upstream traffic has to pass through these functionalities.

If we virtualize the CPE the functions are executed in a VM in the remote cloud. So, the problem is that if we don't know nothing about the traffic, the traffic won't pass through the functionalities

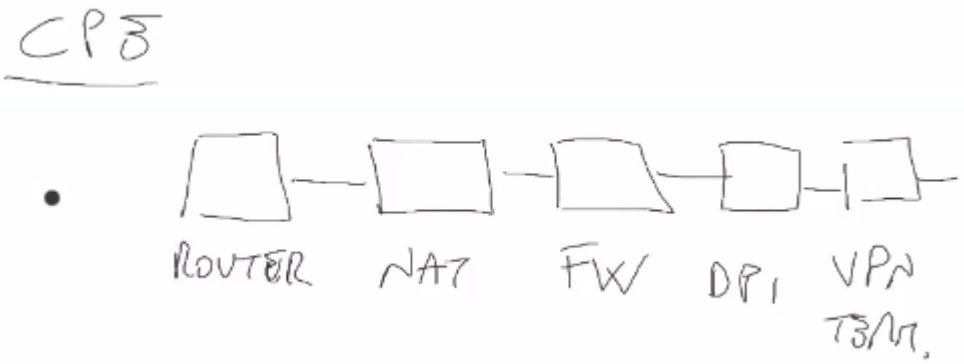


So we must steer the traffic in the following way:

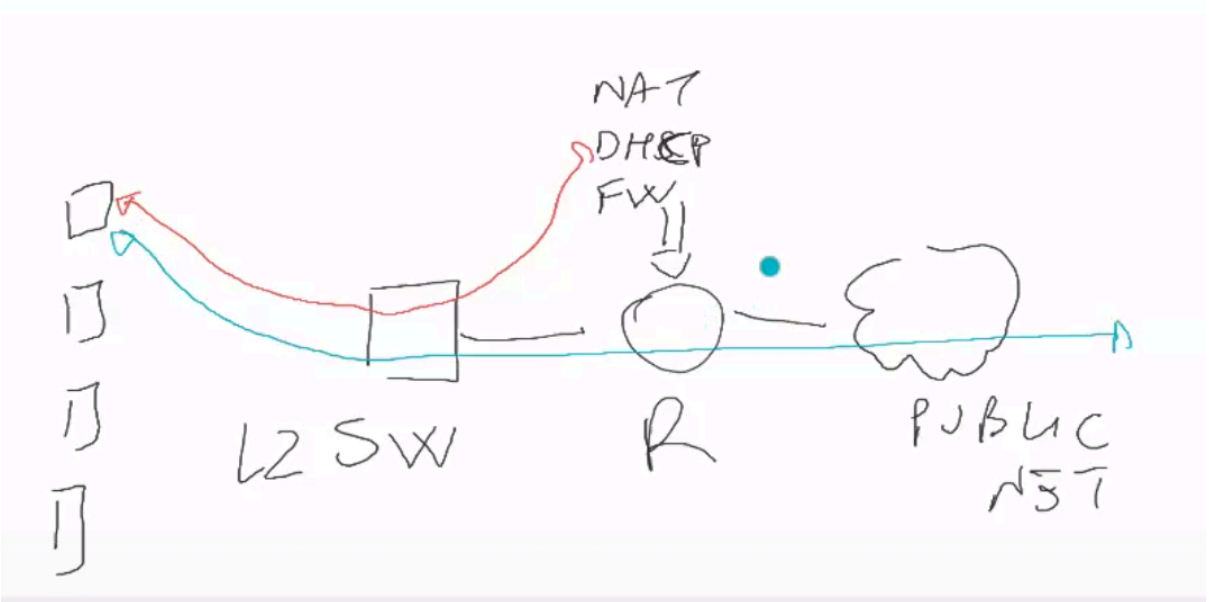


VNF Forwarding Graphs (Use Case #4)

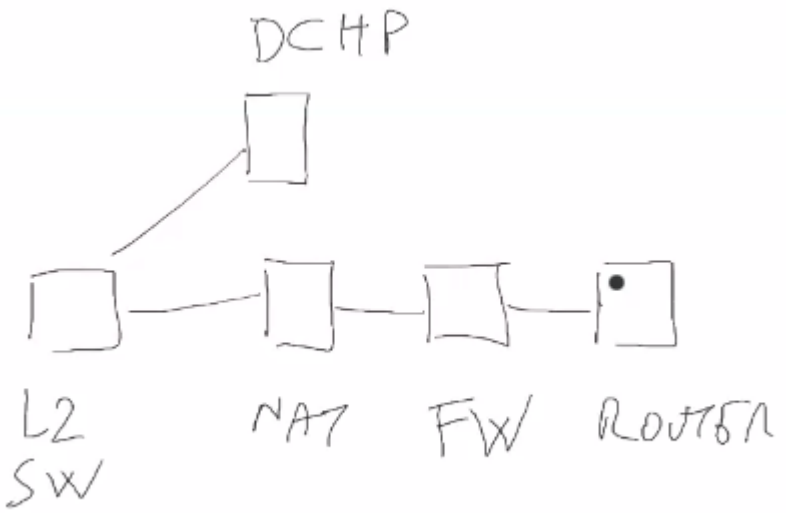
A CPE requires the execution of some functionalities:



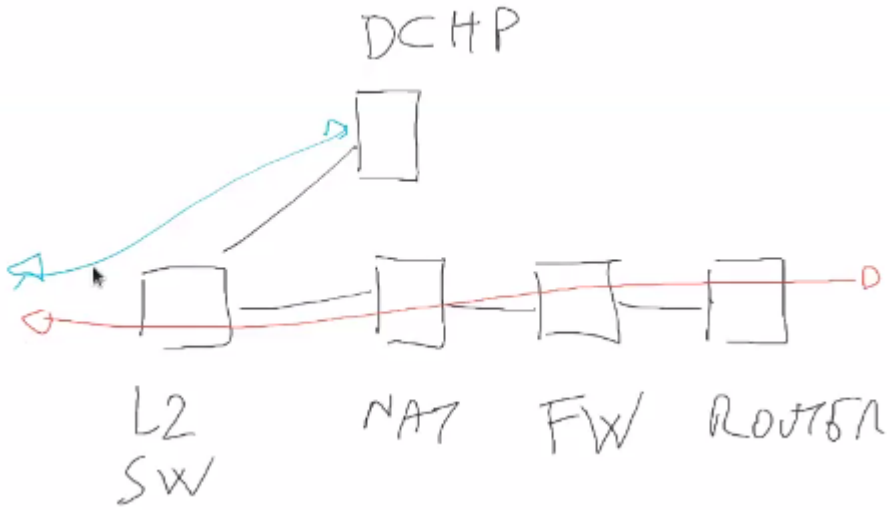
The CPE is a network service. Its logical representation is the one shown above, with 5 nodes interconnected, and is called forwarding graph. So a forwarding graph is a logical representation, an abstraction, of a network service. This case is a linear graph, called *chain*. There are cases modeled with more complex graphs. For instance, the home network:



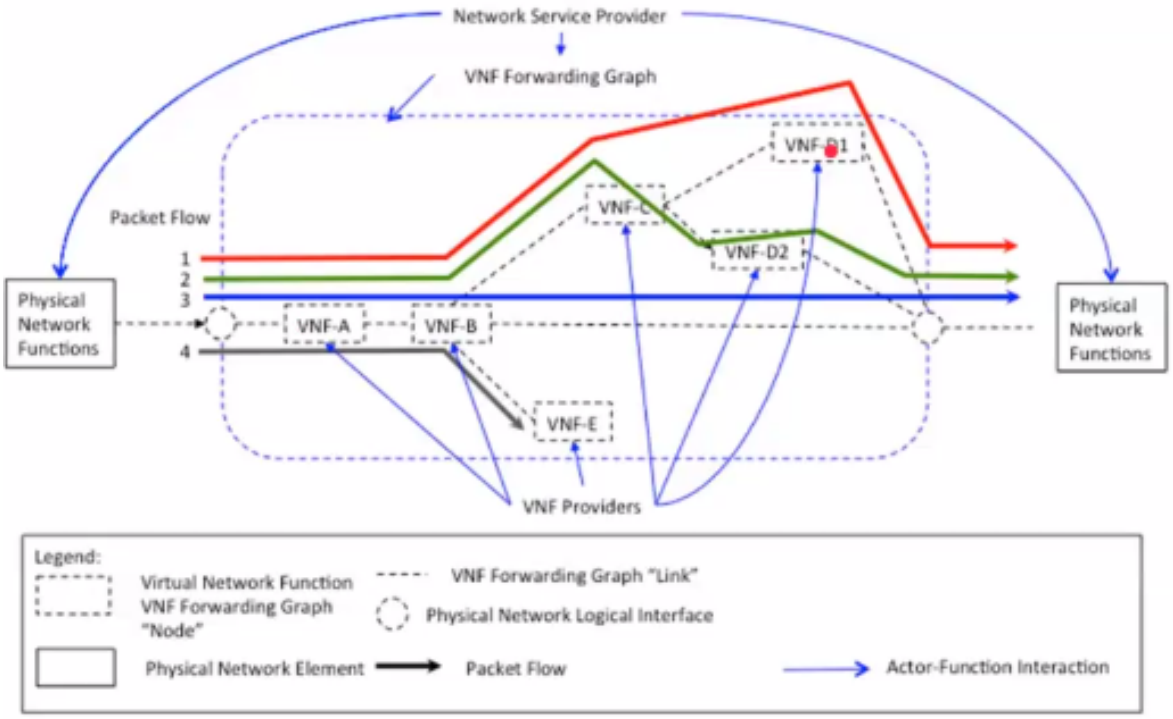
end users that want to connect to the home network require IP addresses, so they ask the DHCP server. Once they get the IP addresses, if they want to send packets to the internet the traffic will need the routing, the NAT and the FW services. The forwarding graph that models this network service can be something like this:



Note that end-users are not reported in the forwarding graph, only middle boxes. We have flows from L2 SW to the DHCP and flows from L2 SW to Router, to exit the home network.



Another, abstract case:

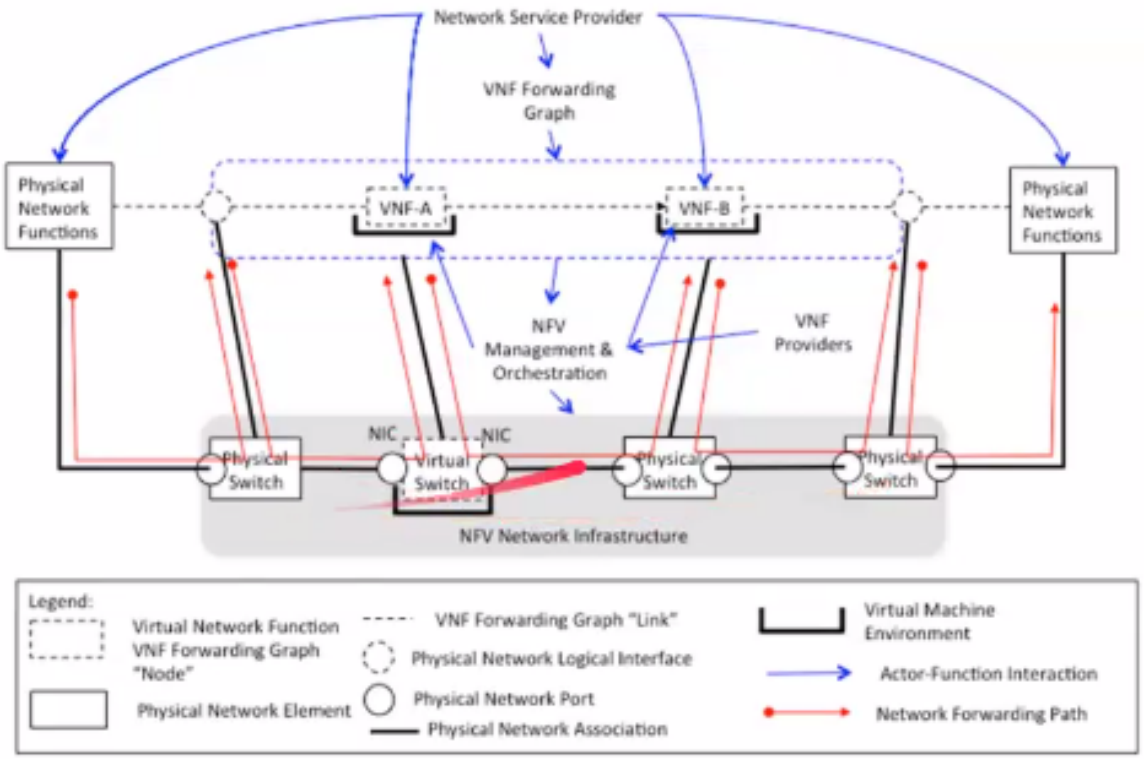


Logical View of Virtual Network Function Forwarding Graph (VNFFG)

This forwarding graph, depending on the specific functionalities, allows to steer the traffic according to these four paths. We see that there must be interconnection between the VNF nodes: this connection is logical, so maybe VNF-A and VNF-B belong to different data centers, but they must be connected logically since there is a path that passes through these two nodes.

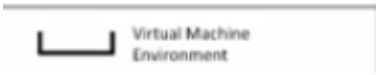
This below is a linear graph and an overlay model[7]. Physically the VNF are hosted in the machines above. The logical interconnections is not realized with a direct physical link:

[7] logical abstraction mapped on a physical infrastructure

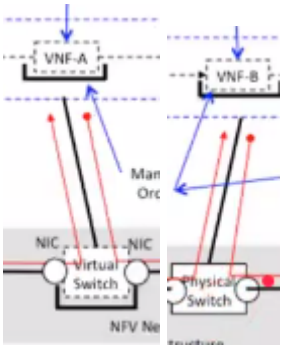


Physical View of Virtual Network Function Forwarding Graph (VNFFG)

This below means that there is hardware support for the creation of virtual entities (like VM):

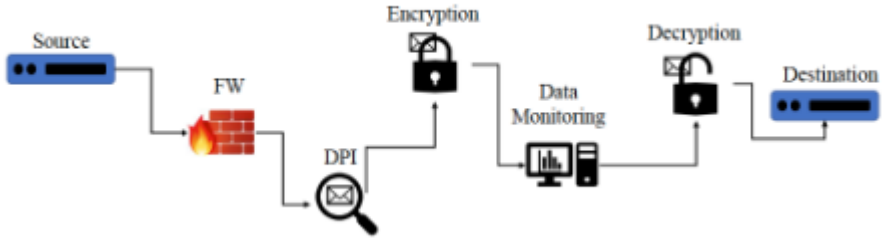


What's the difference between these two:



The first one has a physical machine (server) that is hosting either a virtual switch that interconnects to another virtual machine that is hosted in the same physical machine (the server). In the second case you are connected directly to a physical machine that hosts a single virtual machine. In the first case there are multiple virtual machines running inside the physical machine since you have a virtual switch to switch.

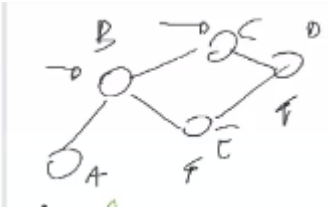
Another example, let's see a security service: there is a firewall, IDS, Gateway, Encryption and Decryption functions



Network Slicing (Use Case #17)

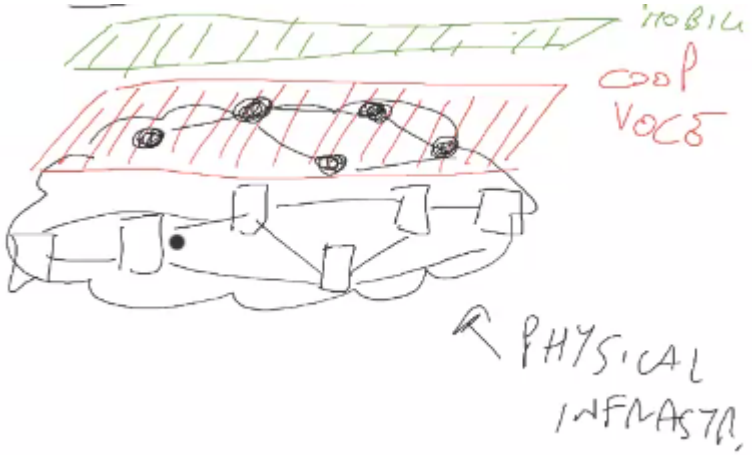
Is a specific instantiation of a forwarding graph. We can abstract a network service as a forwarding graph then I can run instances of the forwarding graph. Let's consider this examples: I am a big mobile network operator, such as TIM, and this is my physical infrastructure:

This infrastructure is composed of base stations, fiber and so on… and it's used to serve the users. Let's assume this is the forwarding graph of the mobile network:



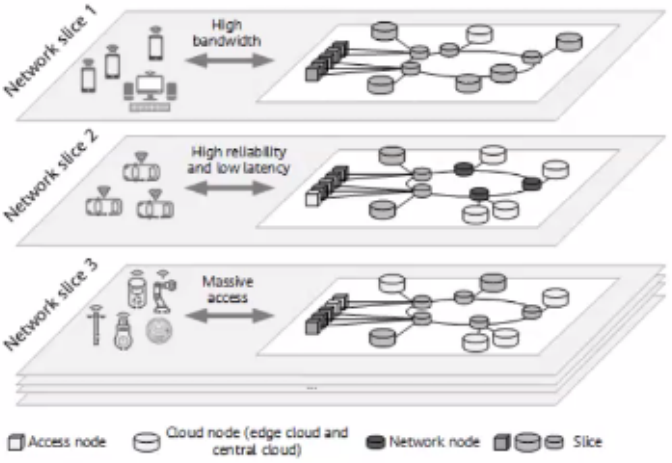
I instantiate the mobile network forwarding graph by instantiating the single network functions (nodes in the graph) and logically connect them as in the graph. It can happen that

I have an over provisioned infrastructure: my users are using my infrastructure but I have a lot of resources that are unused. So what I can do is to provide the opportunity to create virtual mobile operators on top of my physical infrastructure. These operators are not infrastructured, they rent the physical infrastructure. This is done by instantiating multiple forwarding graphs for the mobile network service: each instance is a network slice for a virtual network operator. Inside a network slice we instantiate A B C D E

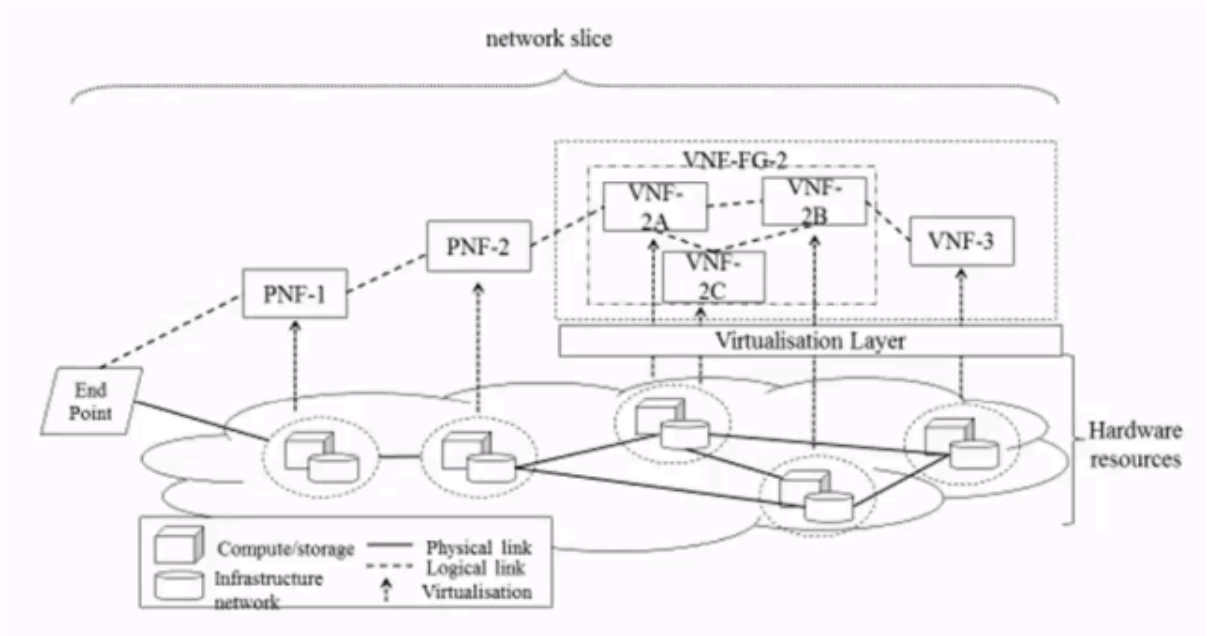


E.g. CoopVoce or PosteMobile are virtual operators that rent the infrastructure, they don't own it.

We can do even more. Let's consider the case of the 5G: it promises eMBB (very high bandwidth), URLCC (low latency and ultra-reliability) and MIoT (massive IoT). 5G relies on network slices to provide all these things that are contradictory all together. In this way, for instance, the applications that require low-latency, but no requirements on bandwidth, we can provide it. Each slice represents a different network service.

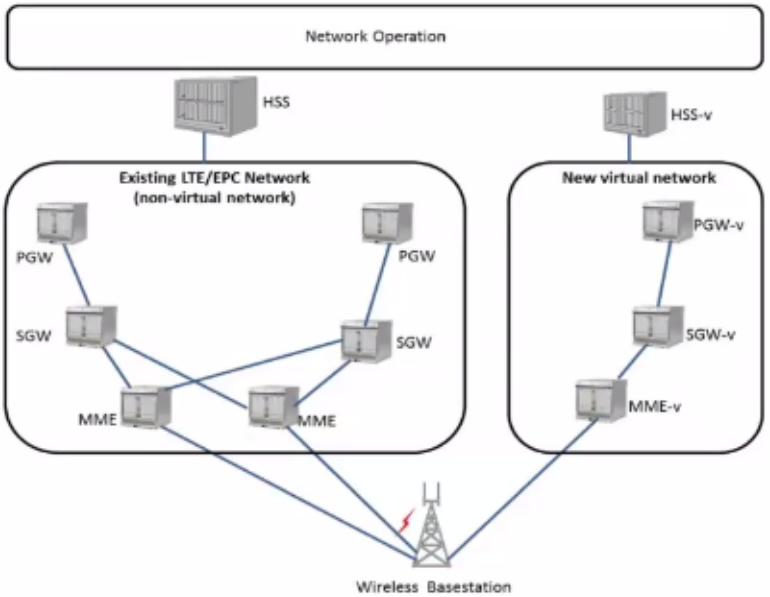


Note that we can have some physical network functions. In this case above we have 2 physical functions and some that are virtualized:



Virtualization of Mobile Core Network and IMS (Use Case #5)

The core part of a mobile network is done like this:



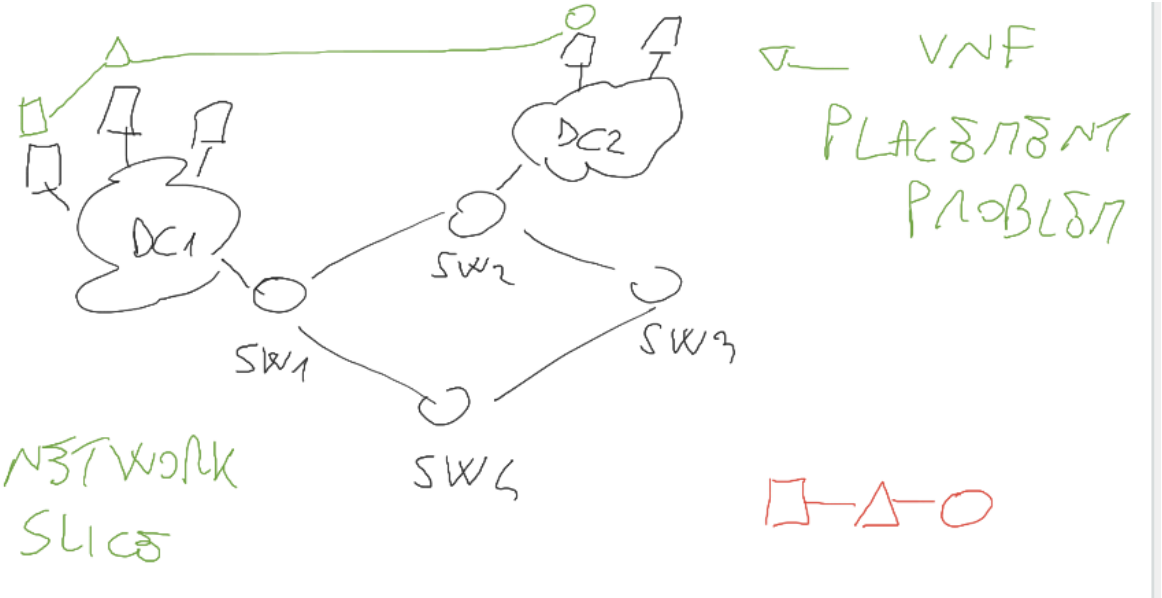
These are the main building blocks of a mobile network. Also the base station can be virtualized since it performs some functions that can be virtualized, but we don't cover this part. From the figure, you see that network slicing was performed: the box on the left is a

non-virtualized core network (e.g. it can be TIM infrastructured network), while the box on the right is a network slide, so a virtual network (e.g. it can be CoopVoce network).

Need for NFV Orchestration, Renting and Traffic Steering

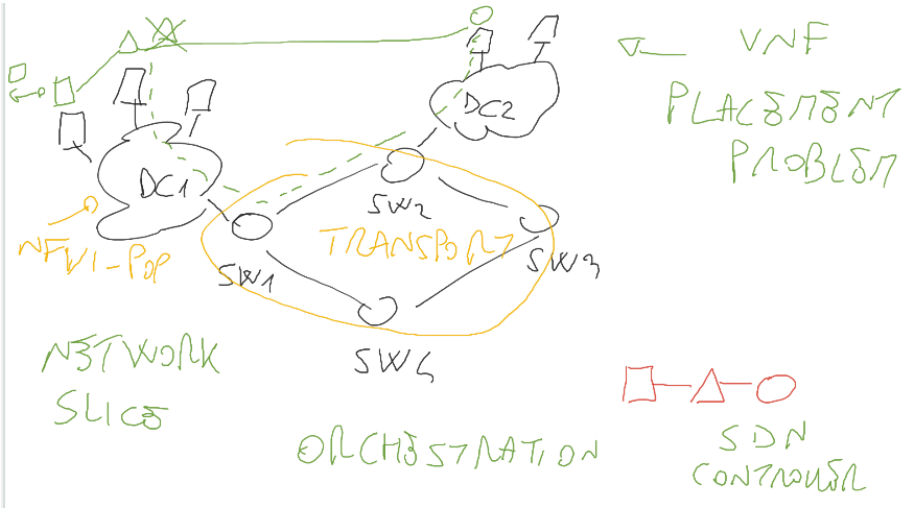
Consider this infrastructure, with two Data Centers, their compute nodes, and the switches. Assume the network service is represented by the following forwarding graph:

Note that different shapes in the graph are associated with different functions in the service. Now we instantiate this forwarding graph in the infrastructure:



And we get a network slice. Question, how can we do this creation of a network slice automatically, dynamically and on the fly? We need something that manages the orchestration[8]. The NFVI-PoP provides storage while the network provides bandwidth and connectivity on demand. So, the logical interconnection, for example, between the triangle and the circle is physically done with the dotted line in the transport

[8] In the cloud computing jargon, the orchestration is the life cycle of virtual resources.



So now we have seen how we can get the cloud resources (with NFVI-PoP) and the Transport. Finally, talking about the SDN controller, it allows us the creation of physical paths representing the logical interconnection between the shapes.

It is possible to rent the middle boxes from a NFVI provider, without having to deal with the device and all the corollary costs. Basically "*Making Middleboxes Someone Else's Problem*". The problem with these middleboxes is cost, as you can see from here:

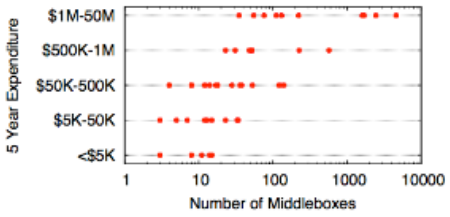


Figure 2: Administrator-estimated spending on middlebox hardware per network.

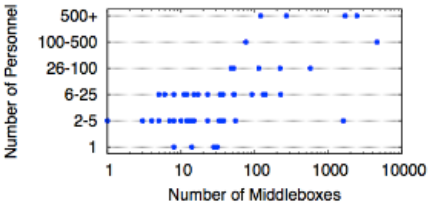
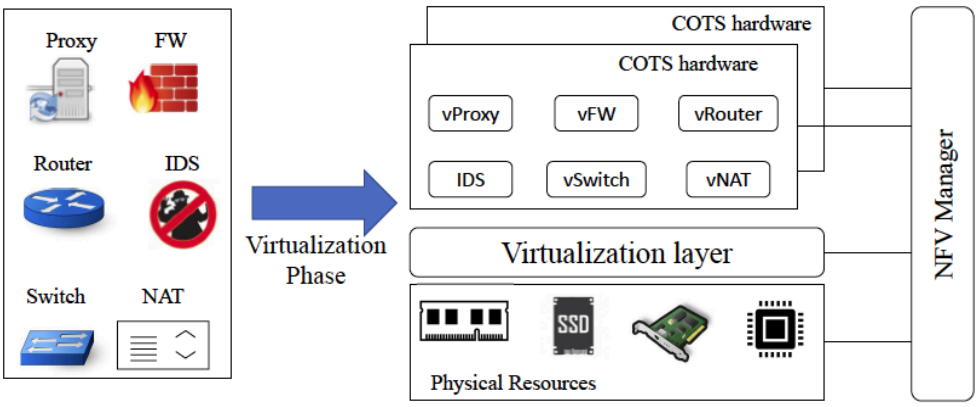


Figure 3: Administrator-estimated number of personnel per network.

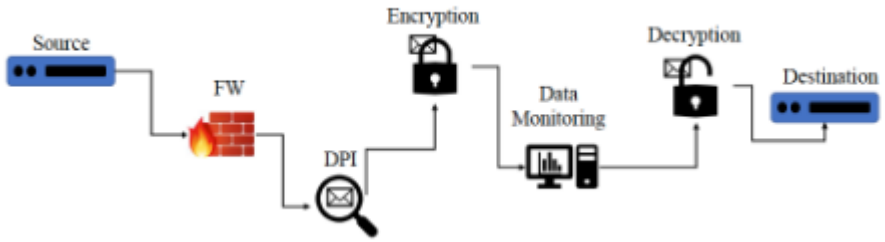
So instead of owning and deploying them, due to their cost, enterprises may want to rent them. In the NFV paradigm, Network Functions (NFs) are executed on Commodity Off-The-Shelf (COTS[9]) hosts



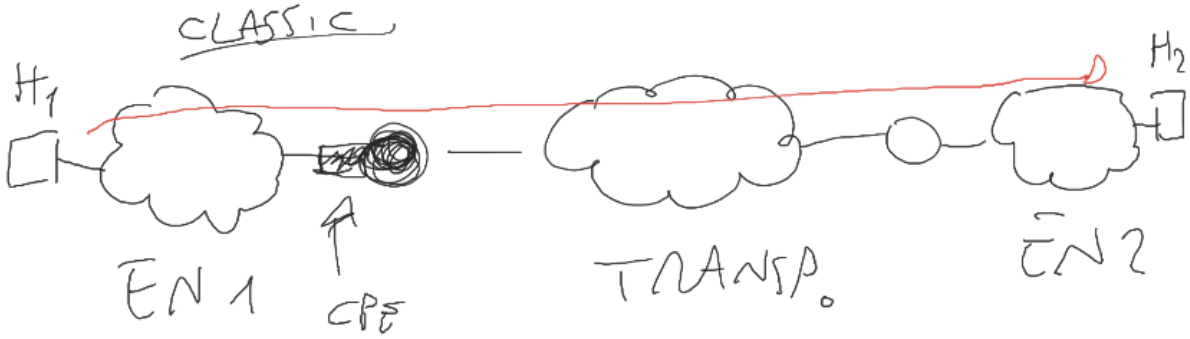
The layer of physical resources (RAM, memory, storage, NIC, CPU…) has on top a virtualization layer that abstracts the physical resources. Above, we have virtual functions that are running inside of Commodity Of The Shelf. On the side we have a management

[9] COTS refers to hardware available on the market for purchase

framework that does the orchestration and interaction with the SDN controller. Problem: let's consider the security network service previously explained:



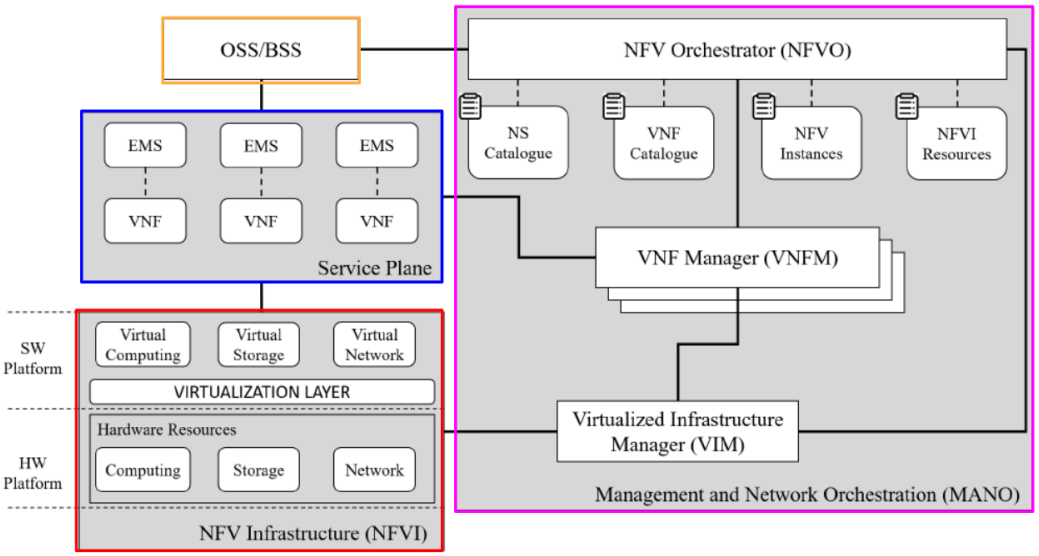
How can we force the user traffic to pass through these functions? In the classical approach all the services that we need are at the edge, in the CPE. So, if H1 sends something to H2 the traffic pass naturally to the functions



In the sofwarized approach, since we virtualize the functions, and so they are in the datacenter, the user traffic doesn't naturally pass through the function.

So we can't use the classical IP paradigm, but we need a way to make the traffic go to the DC. This problem is traffic steering, and we solve it with SFC.

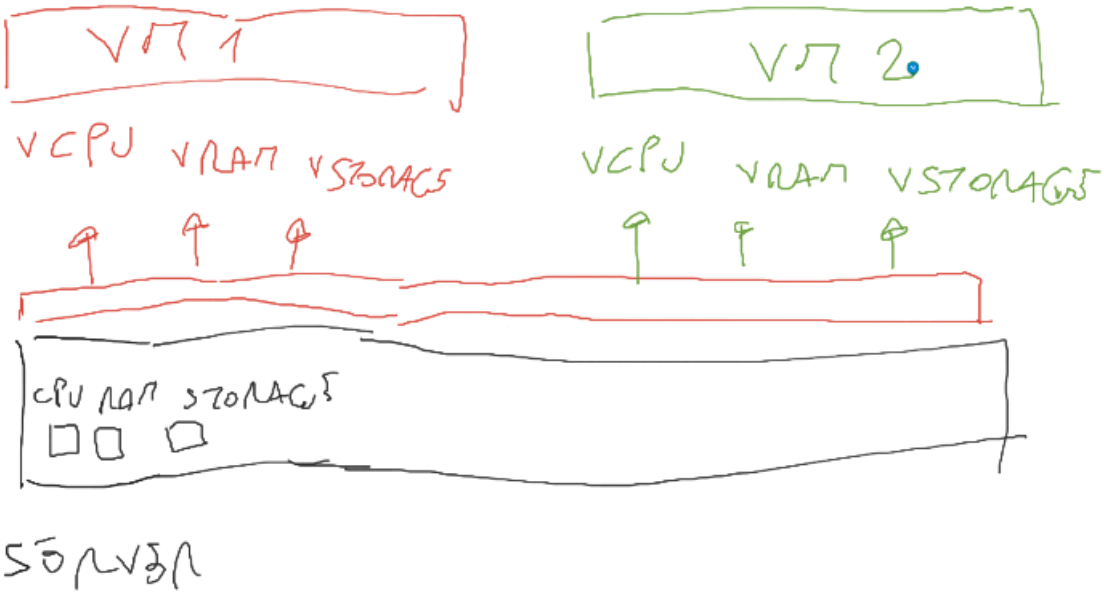
NFV Architecture



Management and Network Orchestration (MANO):

Virtualized Infrastructure Manager (VIM):

Let's explain it with an example:

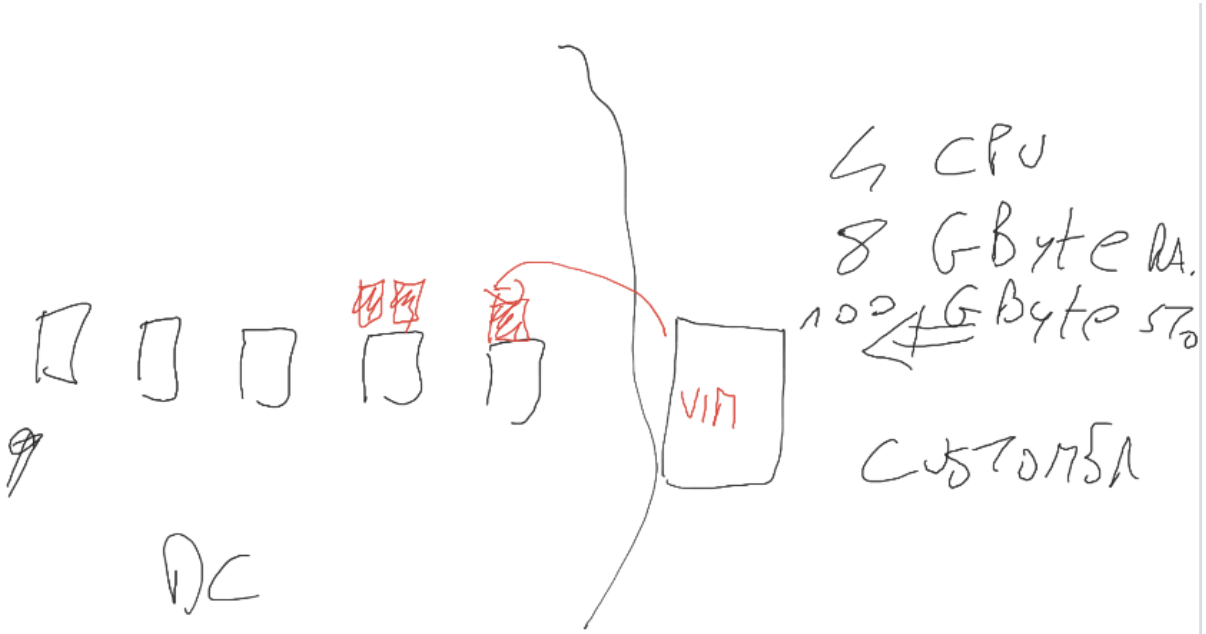


As you can see, the server has a hypervisor that virtualizes the hardware component for the VMs. The hardware is the same but the virtualized instances are different.

When, instead, we have a set of servers, we have the following situation:

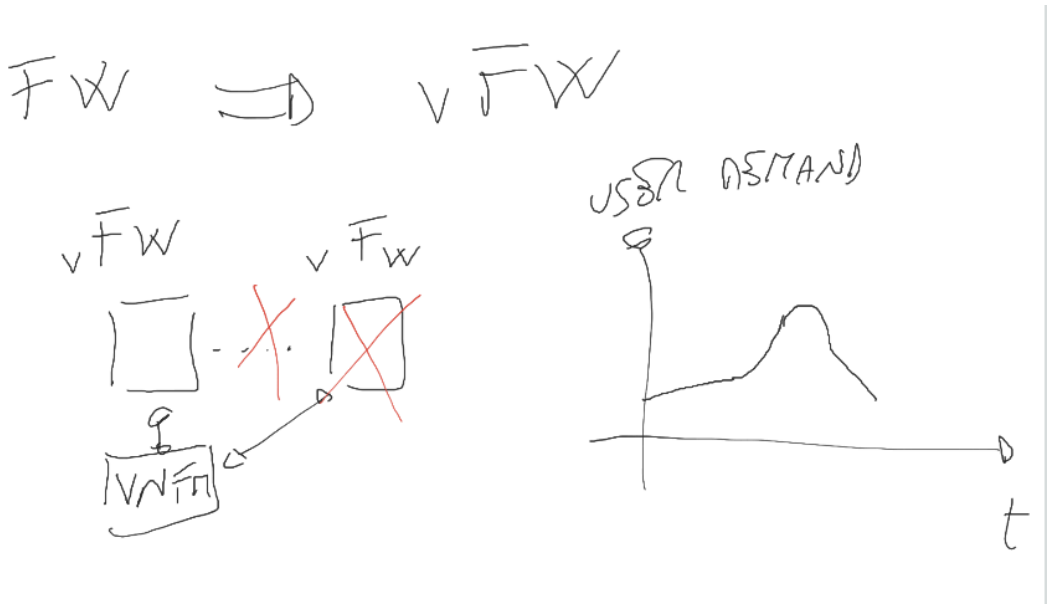
Since this is a cluster of machines, we need a further layer of abstraction that allow the outside to see the servers as a single big machine

The CPU of the single big server is the total CPU, and the same holds for RAM and storage. From outside, a customer can ask for resources, for instance 4 CPU, 8 GByte of RAM, and 100 Gigabyte of storage. The customer is not interested in, say. the specific storage of a machine, they just want to know if the resource is available. The goal of assigning the customer this resources is the VIM, so the VIM choose which specific RAM from a specific machine will be assigned to the user in the external world, that just asks for resources.

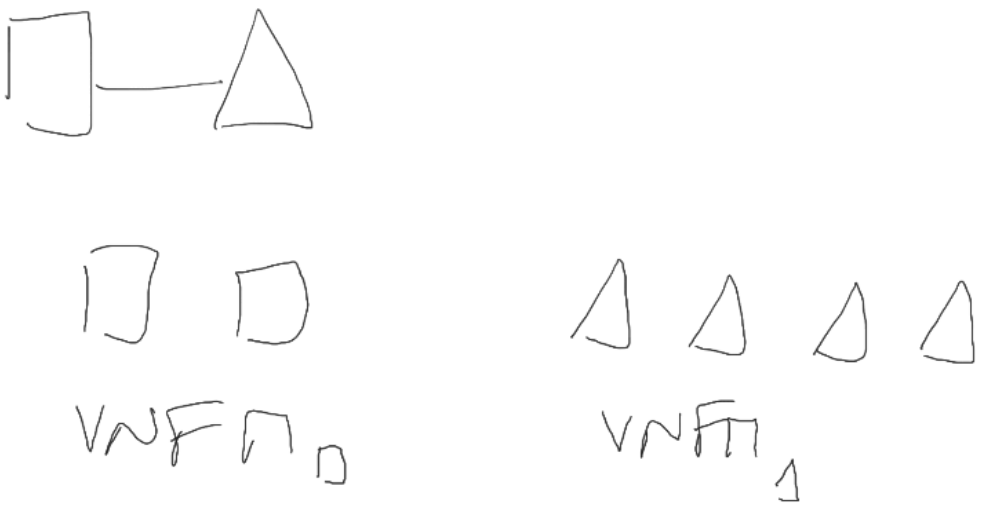


VNF Manager (VNFM)

Let's explain this with an example: we can virtualize the Firewall Service (vFW). Imagine that initially the user demand is low but is growing. So initially we just need 1 vFW, then multiple instances will be required. Who is in charge of creating multiple instances? The VNF Manager: it creates new instances, kills them and manages the load balancing.



This considered just one function. In case we have a network service, with multiple functions. say square and triangle, we have two VNFM one for the square and one of the triangle



NFV orchestrator (NFVO)

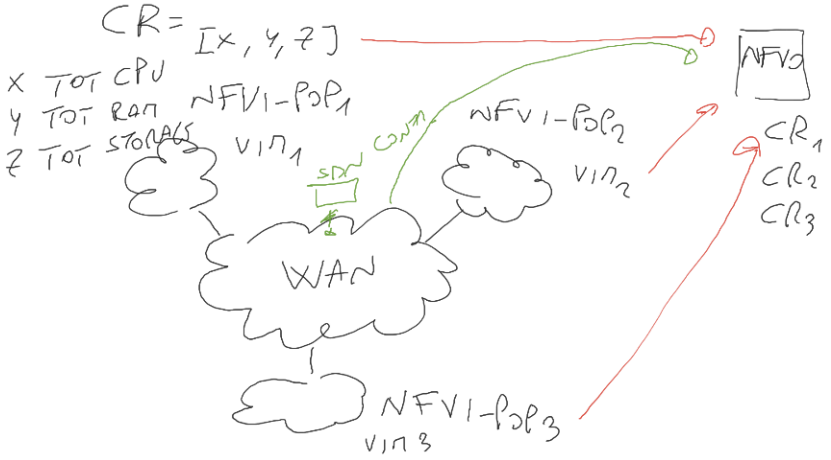
It orchestrates the NFVI resources across multiple VIMs and instantiates all the VNFMs. Basically, it exposes a set of catalogs to the customer.

- The NS catalog: the collection of all the services available in the NFV service provider.

- The VNF catalog: the collection of all the figure we have in the NS catalogs

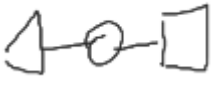
- NFV Instances repository: keeps track of all the VNF and NS instances during their lifecycles
- NFVI Resources repository: keeps track of all the available resources in the NFVI.

We can abstract the NFVI-POP$_i$ with its total amount of resources (CPU, RAM and storages). so for each NFVI-POP we have a vector [VCPU, VRAM, VSTORAGE]. Who is keeping track of these NFVI-PoP quantities? In each PoP there is a VIM. Each of them is keeping track of the available resources. The VIMs from the different NFVI PoPs must communicate their local status to the NFVO.



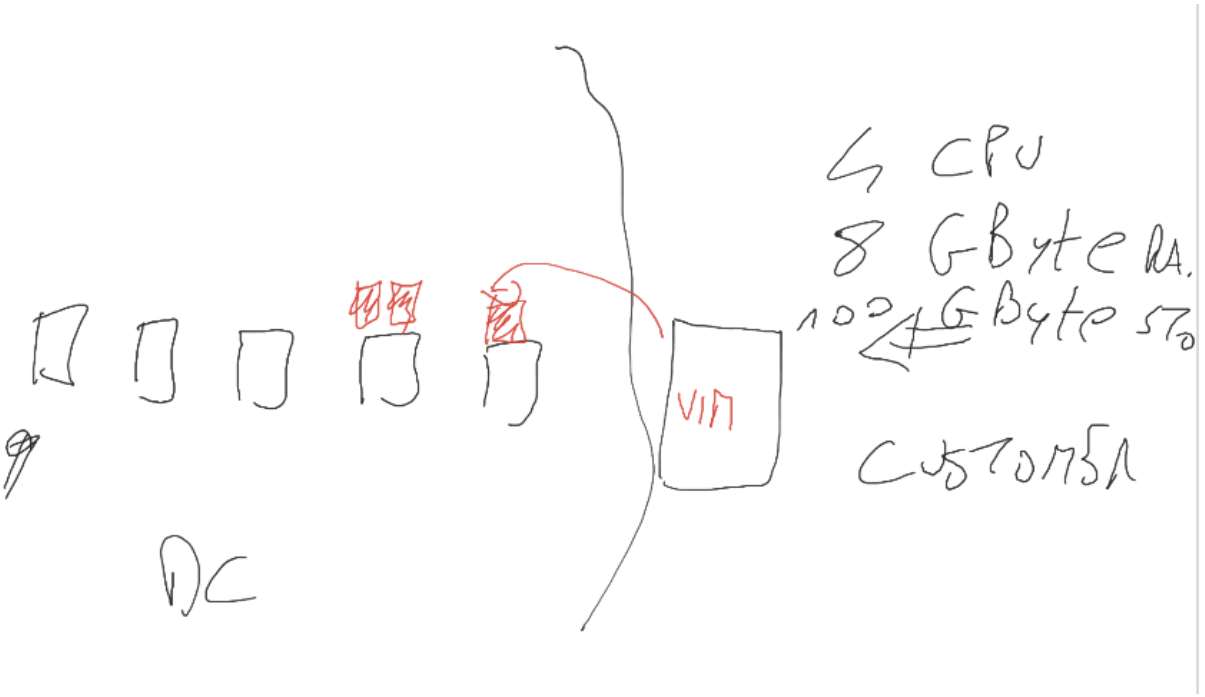
Moreover the NFVO also receives the information of the available resources in the WAN. This information is given by the SDN controller.

Let's see that the NFVO has a network service like this one:



A user asks, through the OSS/BSS, for a network slice for the NS above. The NFVO has to decide where to place these functions (these shapes). The problem is that we know that in a NFVI-PoP we have hundreds of servers and the NFVO can't decide on a specific server that should host a specific function (e.g. the triangle).

The NFVO can decide only in which NFVI-PoP it instantiates function, and then the VIM has the task to decide in which specific server of the PoP function will be implemented



NFV Infrastructure (NFVI)

The transport network and all the data centers. DC are not necessarily like the Google ones: sometimes we can rely on cloud resources that are closer to the user, for instance mobile edge computing. In the NFVI we have the hardware resources, and on top of it the software platform, that is able to abstract the hardware and expose virtualized hardware elements to the VM. For instance, Virtualbox is the virtualization layer that abstract hardware to the PC.

Service Plane

Is the logical place where the virtual middle boxes are running. Each middle box (VNF) has a northbound interface with its own EMS (Element Management System). The EMS provides VNF users with the same interface as if it were a PNF.

Operation Support System/Business Support System (OSS/BSS)

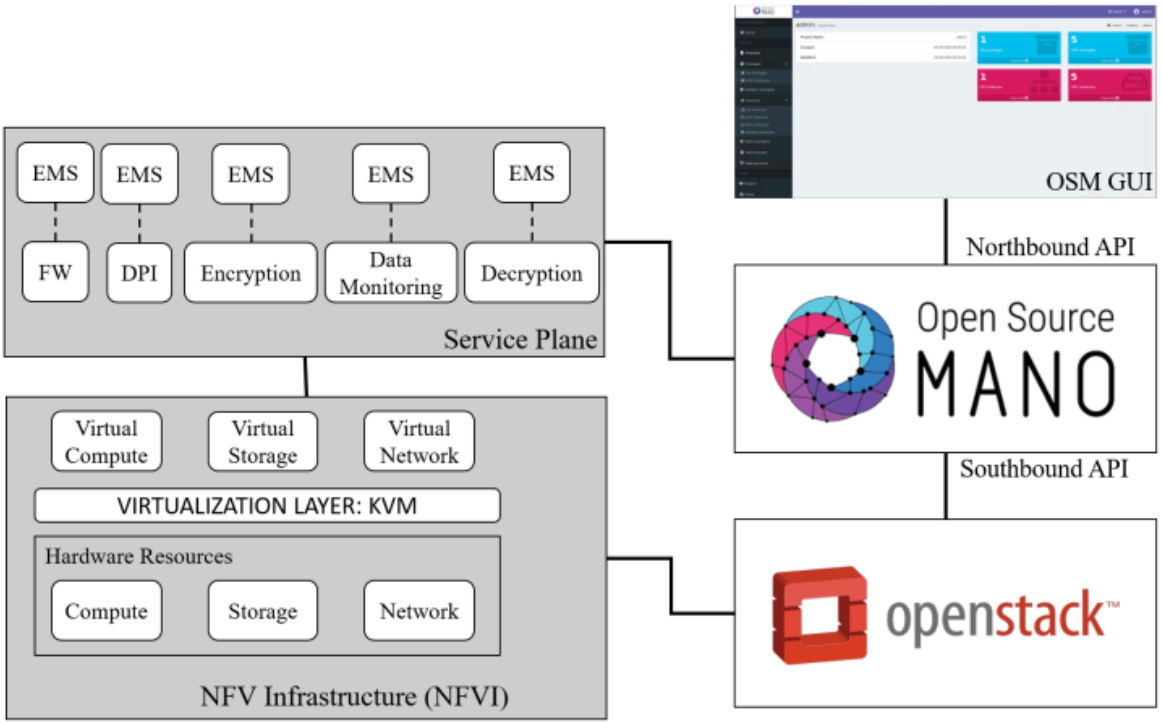
Let's consider this example to understand: let's consider a NFV service provider that has its catalog of the services it can provide. For instance: of them is represented by a forwarding graph. The OSS/BSS is the element by which our customer interacts with the NFV service provider. So it's the interface from which the customer asks: "please create a network slice of the CPE", for instance, or another NS. Precisely:

- The OSS does service provisioning, network configuration, fault management…
- The BSS does ordering, billing and revenue management aspects

Creation of a NS deploying the NFV Architecture

You can use:

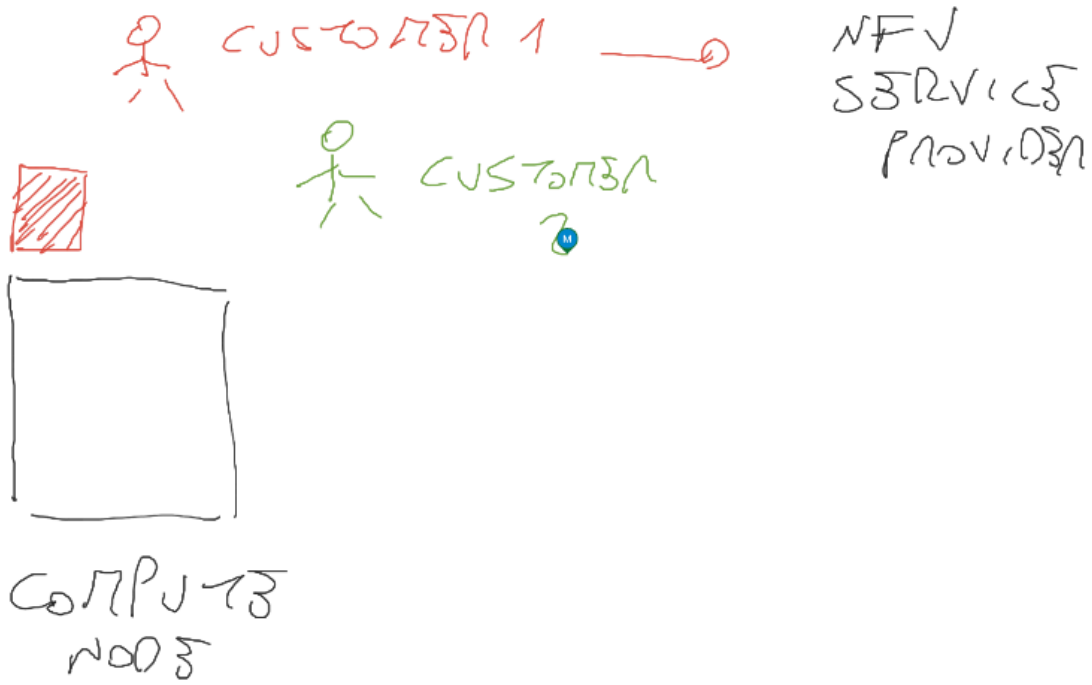
- Open Source MANO as NFVO and NFVM. OSM has a NB interface with the GUI and a SB interface with OpenStack
- OpenStack as VIM. It has
    - Controller: manages the services needed in the OpenStack environment
    - Compute: run the VNF instances and connect them



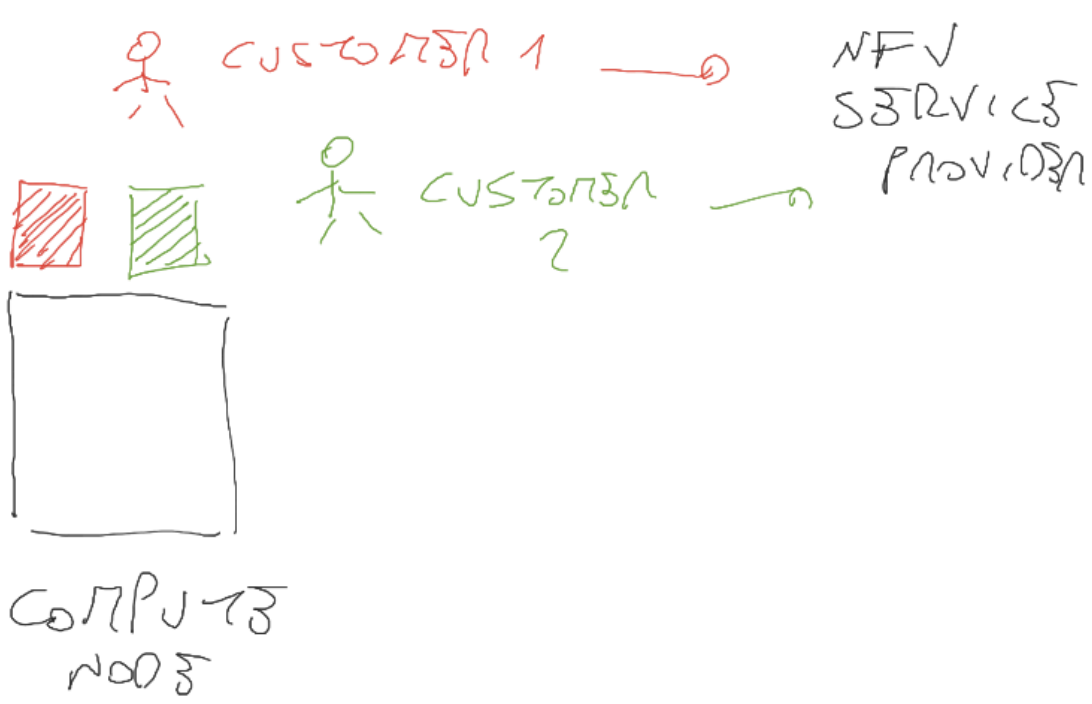
In this case the network service is implementing a FW, DPI, Encryption, Data Monitoring and Decryption.

Isolation for Securing the Virtualization of Middle Boxes

Thanks to virtualization we can run multiple VNF inside the same physical machine. An important feature of virtualization is that it creates isolation between different VNF running simultaneously. Let's say that Customer 1 asks the NFV Service Provider for a network slice. The VIM in the MANO framework of the NFV Service Provider decides that it will put the virtual function in this compute node above:



Then another customer (the green) arrives and asks for another Network Slice. Let's assume the VIM instantiates it in the same compute node.

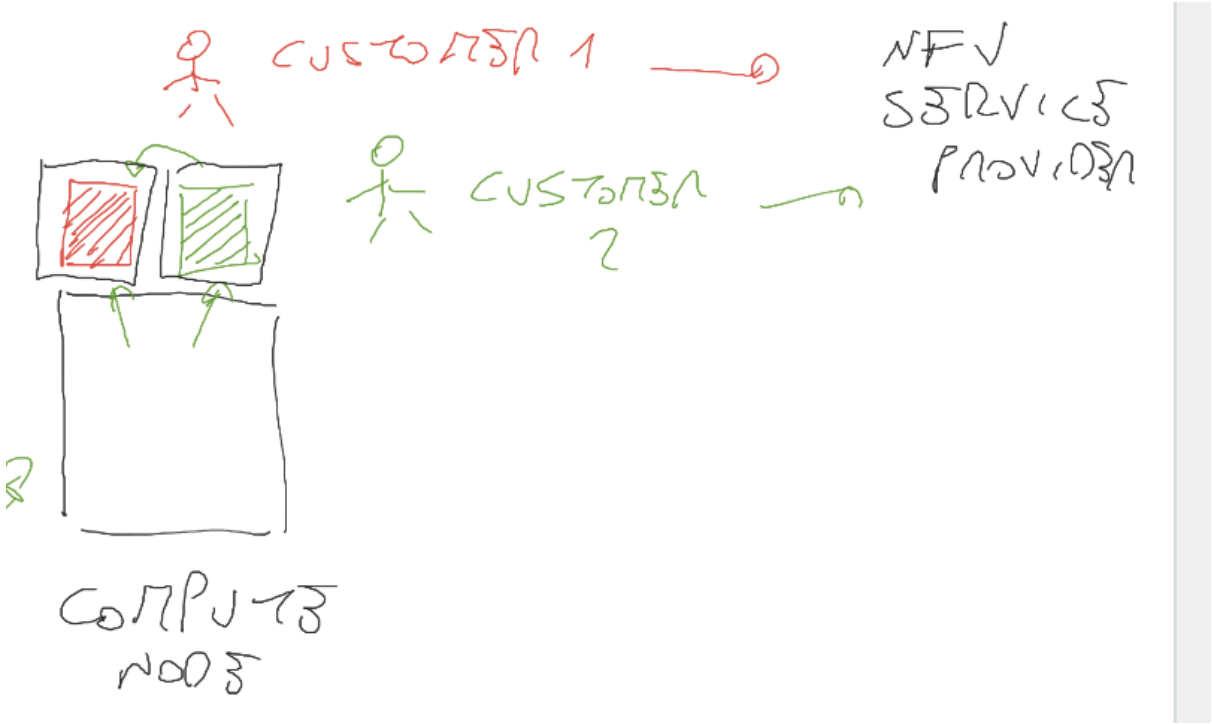


This is multitenancy: several customers share the resource and use them as they were the sole users of those resources, not interfering with the usage of the other users.

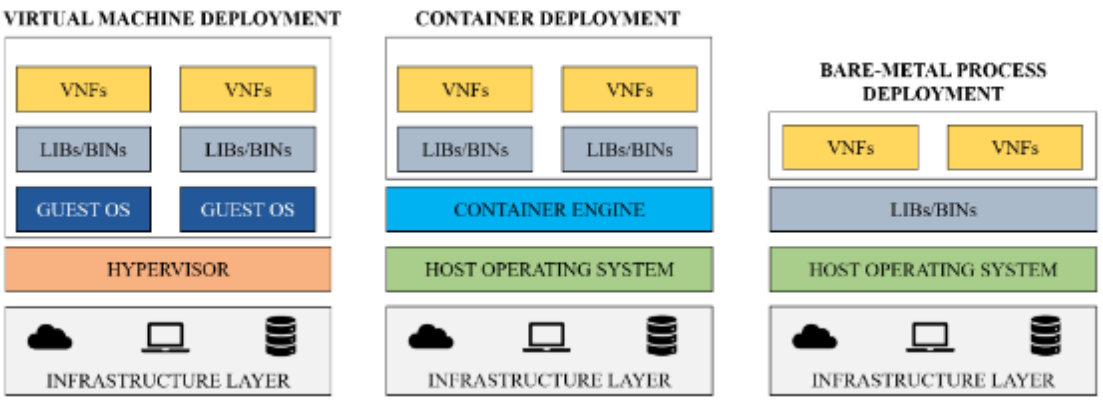
With this approach, we have the following two problems of security:

1. If one of the two VNF get compromised it may get a backdoor to access to the other VNF.
2. If the physical machine gets compromised, you have access to both VNF.

Isolation prevents these from happening.



Different type of virtualization provide different level of isolation, and so, of security:



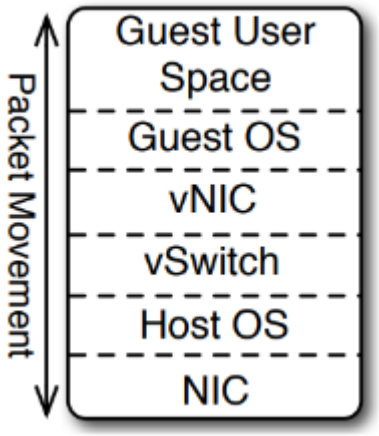
1. VM Deployment: each VM has its own functions, libraries and OS. VirtualBox is a Hypervisor. Hypervisors are classified as:
    a. Native Hypervisors: installed on the host machine hardware directly.
    b. Hosted Hypervisors: executed inside a traditional OS installed on the host machine. Virtualbox is a hosted hypervisor.

2. Container Deployment: containers share the same OS, the one of the local machine, but they have isolated libraries and processes in the user space. A container engine is Docker. In this paradigm we are sharing the OS, so we are losing isolation.
3. Bare-Metal Process Deployment: the virtualization is at the application level.

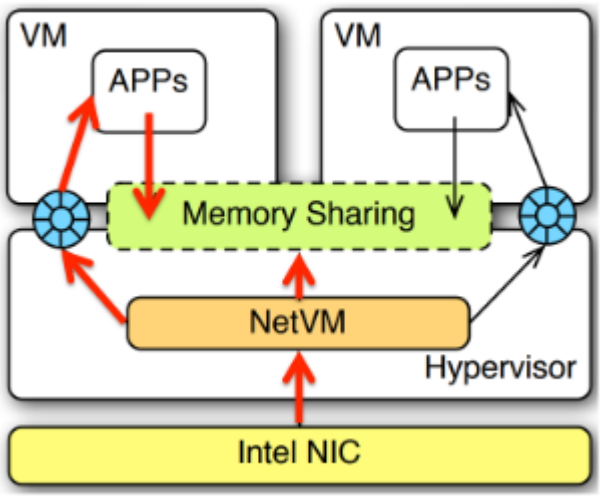
As you can see, the more we go to the right in the figure above, the lighter we go, but we lose isolation.

NetVM

Using a CPU, this is the traditional packet movement when a VM has to use the host NIC.



We can do better by improving performances. NetVM is one of these attempts. Let's assume we have a compute node with two VNFs. There is the physical NIC with the user traffic that enters, and should be redirected to the VNF. Each VM has its own virtual NIC, connected to the virtual bridge. Through the virtual bridge we redirect the traffic to the VNFs. These virtual elements perform read and write operations in the memory in order to pass packets from a VNF to another, and these read/write operations are time-consuming. NetVM task optimizes this read/write mechanic with a shared memory inside the hypervisor that is exposed to the VMs.

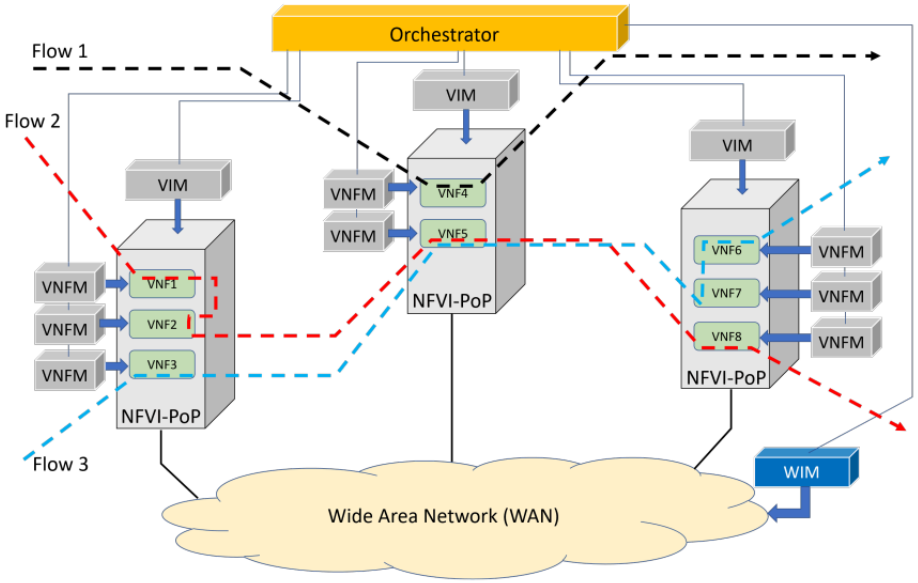


The packets coming from the outside through the NIC are inserted in the shared memory and the first VNFs is notified that there is a packet in the shared memory. The VM directly accesses the packet in the memory and potentially writes in the packet directly in the shared memory, then is the turn of the second VNFs and so on.

VNF Placement and Interconnection

We want to determine:

1. VNF optimal placement: instantiating functions in NFVI-PoP
2. The optimal migration strategies to migrate VNF from a server to another
3. How to interconnect the VNF

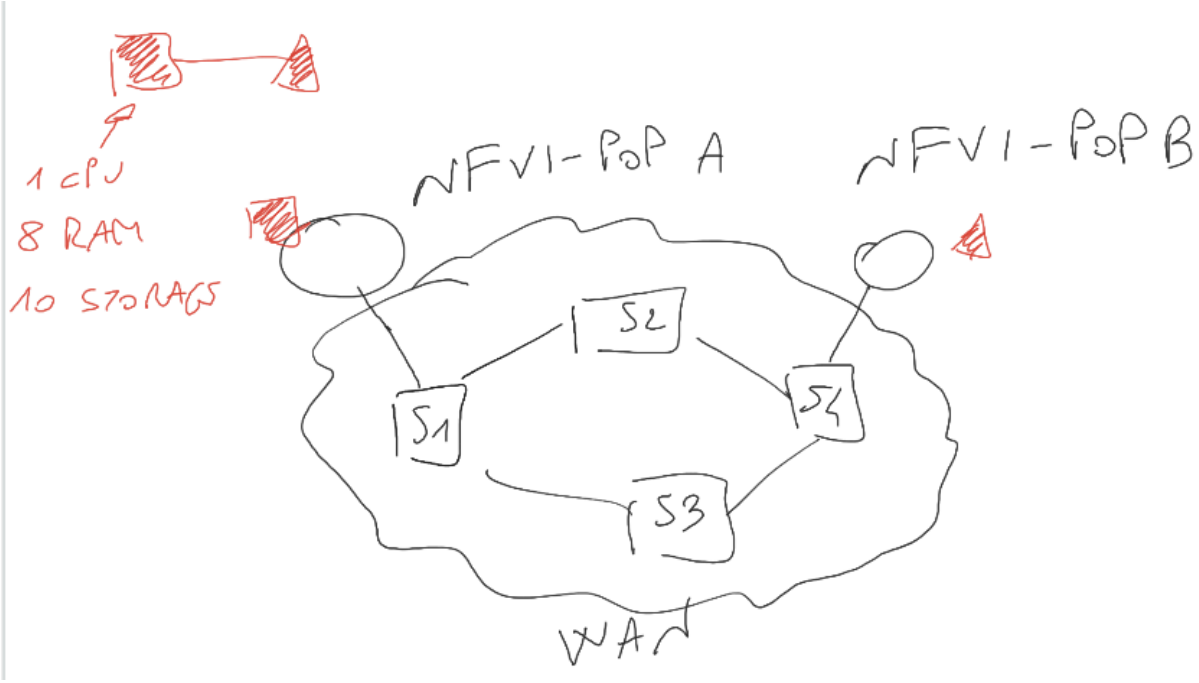


In the figure above we have:

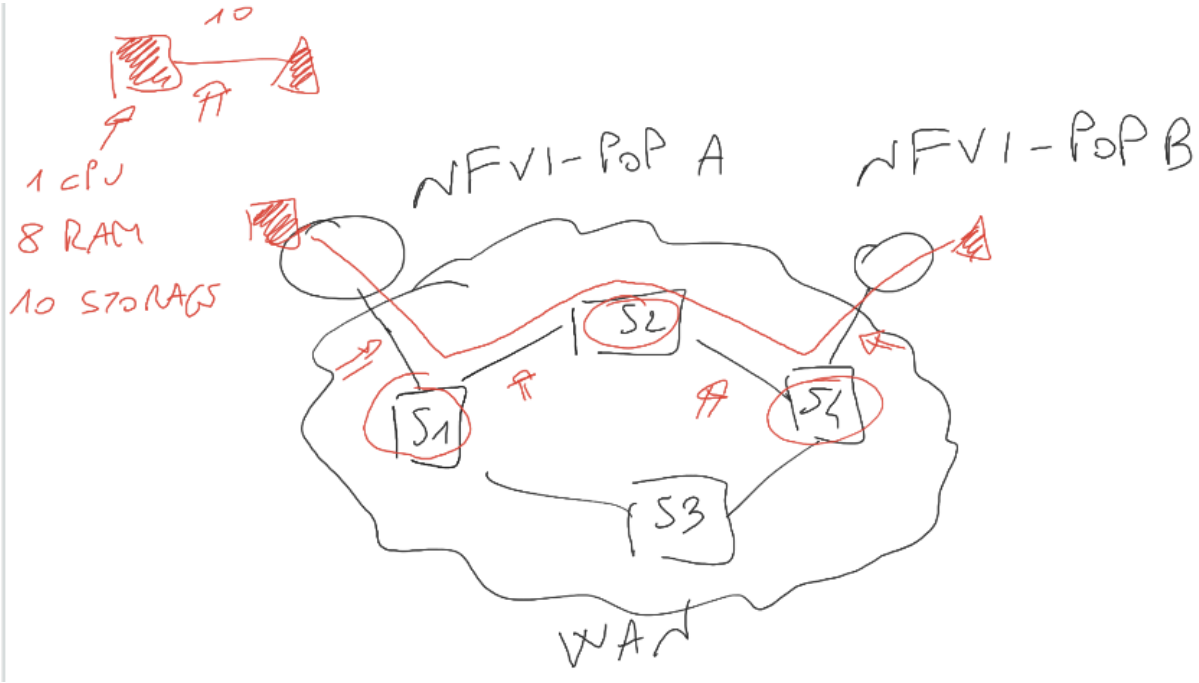
1. The WAN, the transport that interconnects the different NFVI PoP. The WIM (Wan Infrastructure Manager) is the SDN controller.
2. The NFVI PoPs, that are the data centers. In a PoP, all the compute nodes are abstracted as a big server, thanks to the VIM. Inside each PoP we have different instances of running VNFs, with their own managers. The managers communicate with the orchestrator.
3. The orchestrator, that is communicating with the different VIMs using the specific APIs.
4. There are three user traffic flows that are steered through the VFNs in the various PoPs.

The orchestrator decides in which NFVI-PoP (DC) instantiates a VNF, the level of granularity of a specific machine in the DC is related instead to the VIM. For the VNF we need cloud resources, available in DCs. If I don't need specs on latency, I can use cloud (AWM, Azure etc.) where I can assume I have infinite capacity. In Mobile edge computing the resources are placed closer to the end user. Network resources: bandwidth of the links + the buffer.

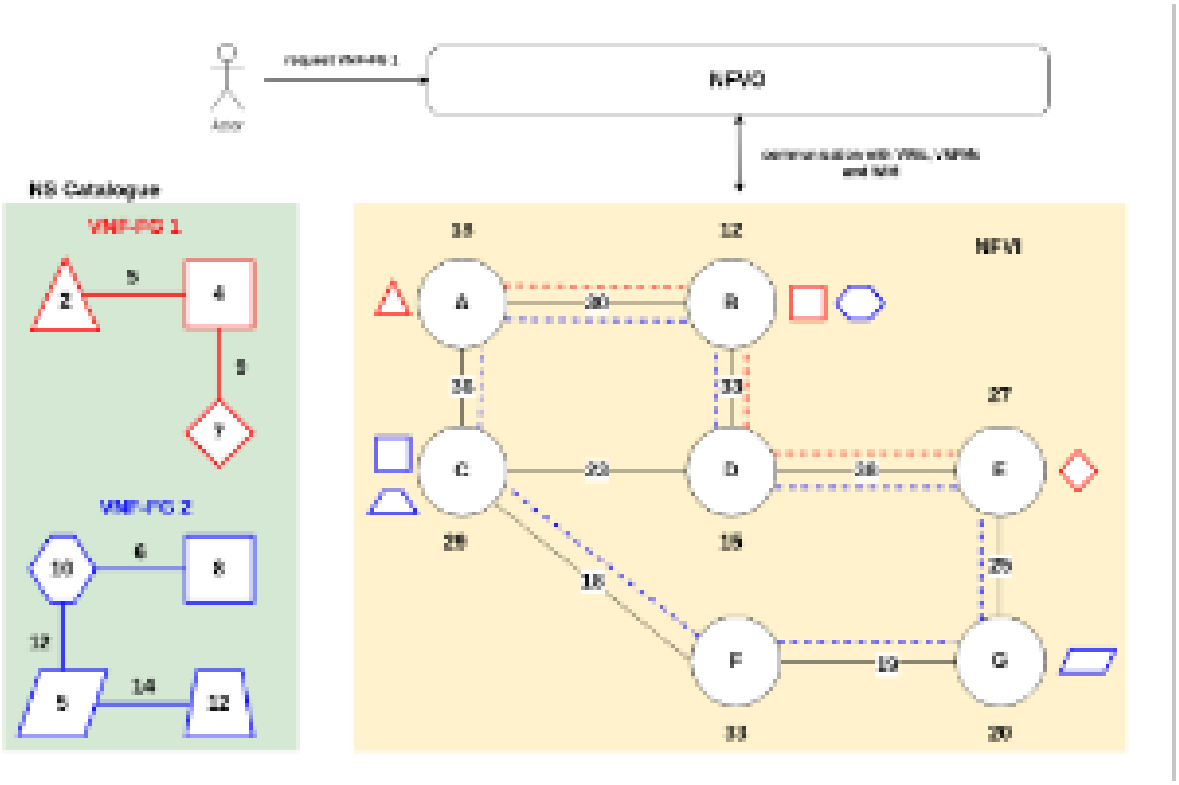
Then we have to configure the network switches such that they know how to interconnect the VNF:



The logical connection has to be mapped through physical links. I have to go over all the links and perform resource (bandwidth) reservation.



I have to install flow rules in the switches. So these are the three steps.

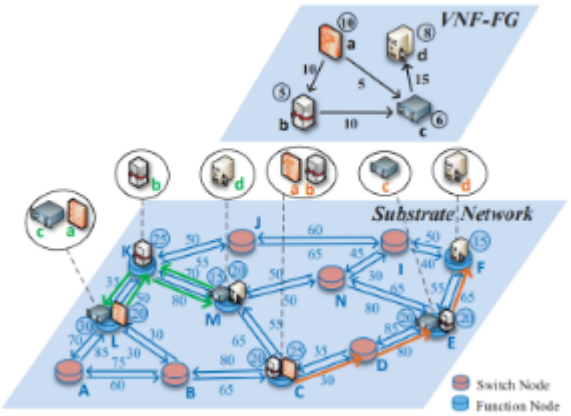


We have the network service catalog and the VNF catalog (catalog of all the virtual network functions). If in the NFVO we have three different VNF available we have three different images. The VIM abstracts the entire DC (the circle in the NFVI) and gives the NFVO a number that represents the resources available. We have two network slices since we have two colors.

Mathematical Definition of the Problem

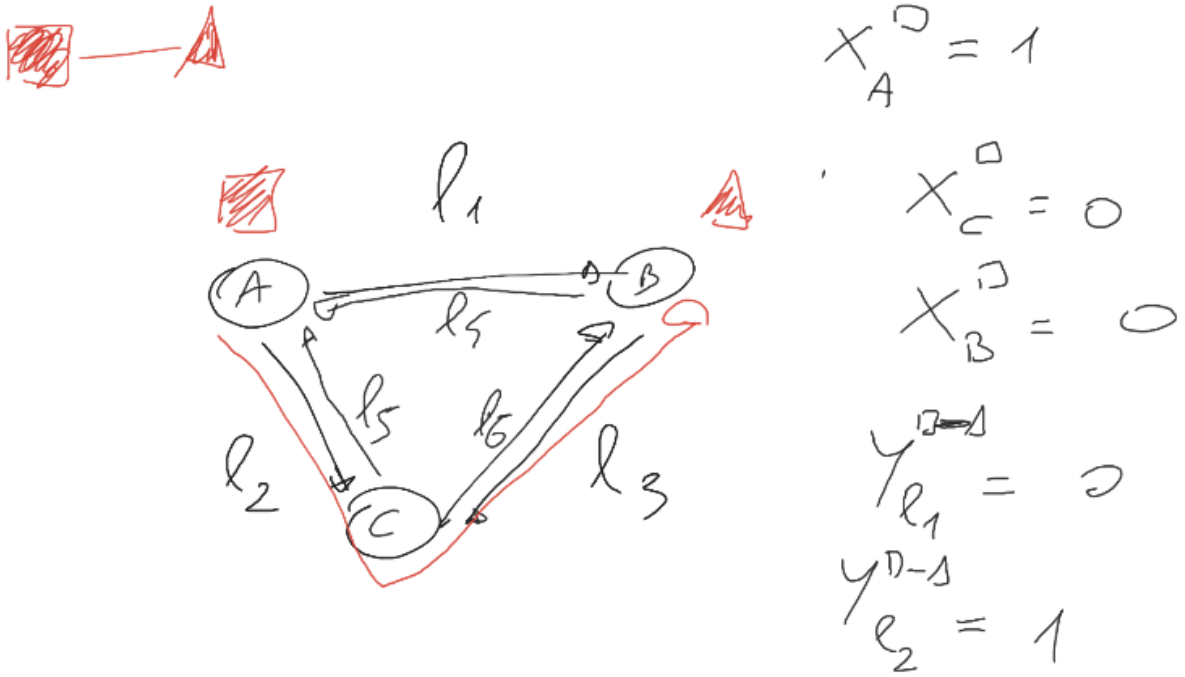
In the dynamic problem we consider the fluctuation of the user demand and move the placement of the VNF according to the user demand, that can vary in terms of bit per second and number of users. But here we assume the static scenario, so the number and the set of requested VNF-FGs[10] is known in advance and doesn't change over time.

[10] VNF Forwarding Graphs



In the figure above the DCs (function node) are shown in blue, while the two slices of the FG are in green and red. To define the problem we need to define the decision variables:

- x_i^m is whether a virtual node m is allocated at function node i
- $y_{i,j}^{mn}$ is whether a virtual link mn (connection between virtual node m and n) use physical link ij



After the decision variables, we need to define also the constraint of the problem:

- Physical resources of a node
- Physical resources of a link
- To place all the VNF

The problem is NP-Hard, so a heuristic approach is required to allow the NFVO to find a

solution in polynomial time.

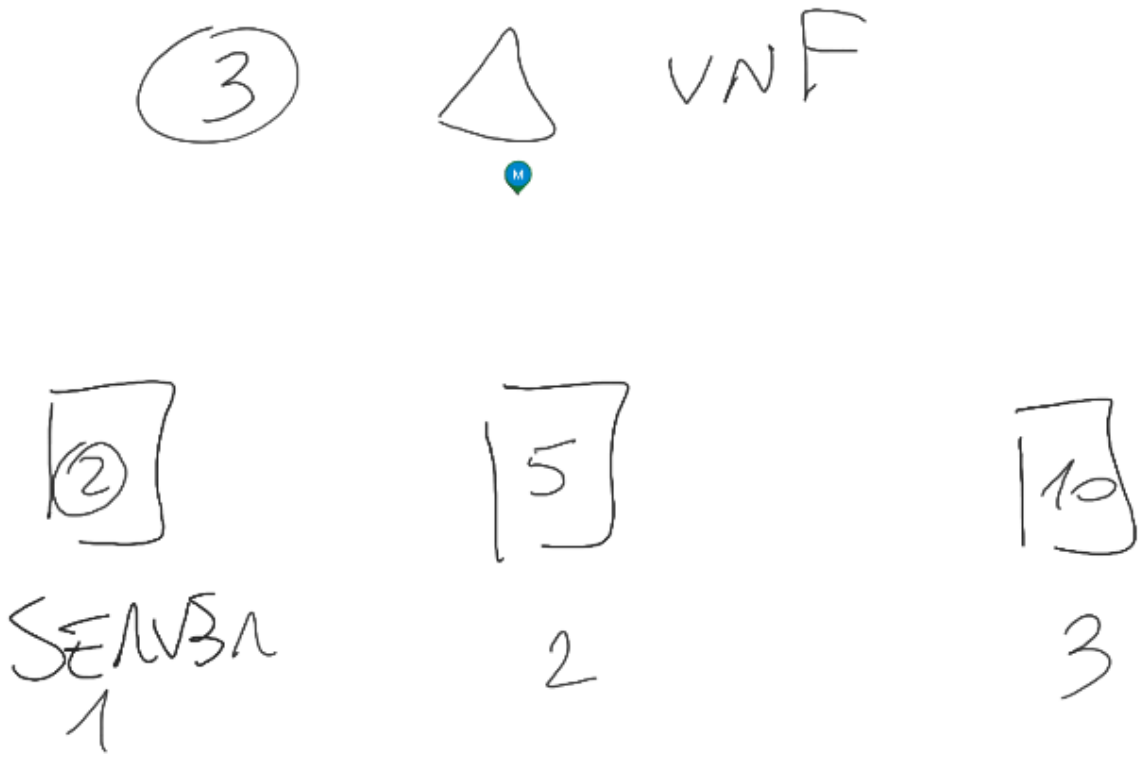
AVMVPH Greedy Heuristic

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$

- c^i_{VNF} is the processing capacity required by the i-th VNF
- $c^i_{NFVI\text{-}PoP}$ is the available processing capacity of the j-th NVFI-PoP

Consolidation: try to put as many VNF as possible in a node.

Let's do this simple example:



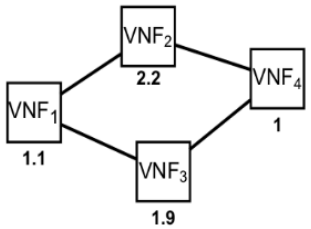
$c^{triangolo}_{VNF}$ = 3, $c^1_{NFVI\text{-}PoP}$ = 2, $c^2_{NFVI\text{-}PoP}$ = 5, $c^3_{NFVI\text{-}PoP}$ = 10. So for the first server we have an INF value, for the second we have 2 units that remain unused, while for the third server 7. So we prefer the second.

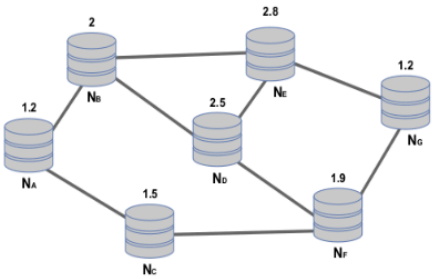
A more complex example:

	N_A	N_B	N_C	N_D	N_E	N_F	N_G
VNF_1	0.01	0.81	0.16	1.96	2.89	0.64	0.01
VNF_2	Inf	Inf	Inf	0.09	0.36	Inf	Inf
VNF_3	Inf	0.01	Inf	0.36	0.81	**0**	Inf
VNF_4	0.04	1	0.25	2.25	3.24	0.81	0.04

Distance between the VNFs and compute nodes

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$

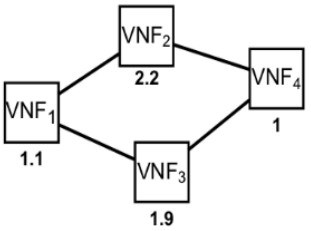


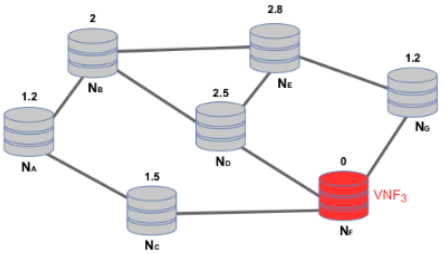


	N_A	N_B	N_C	N_D	N_E	N_F	N_G
VNF_1	**0.01**	0.81	0.16	1.96	2.89	Inf	0.01
VNF_2	Inf	Inf	Inf	0.09	0.36	Inf	Inf
VNF_3							
VNF_4	0.04	1	0.25	2.25	3.24	Inf	0.04

Distance between the VNFs and compute nodes

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$

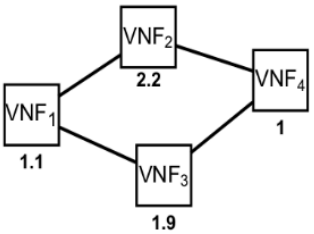


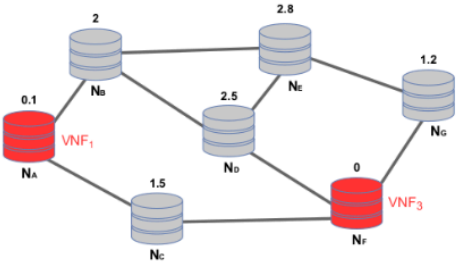


	N_A	N_B	N_C	N_D	N_E	N_F	N_G
VNF_1							
VNF_2	Inf	Inf	Inf	0.09	0.36	Inf	Inf
VNF_3							
VNF_4	Inf	1	0.25	2.25	3.24	Inf	**0.04**

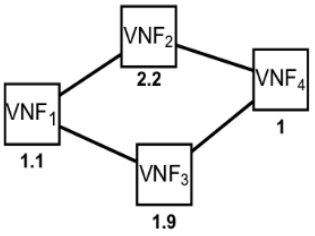
Distance between the VNFs and compute nodes

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$



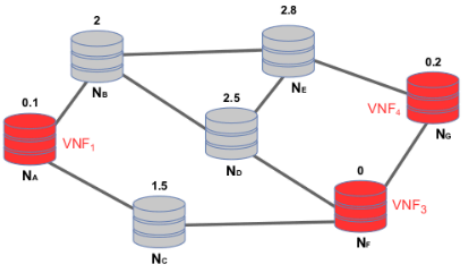


	N_A	N_B	N_C	N_D	N_E	N_F	N_G
VNF_1							
VNF_2	Inf	Inf	Inf	**0.09**	0.36	Inf	Inf
VNF_3							
VNF_4							

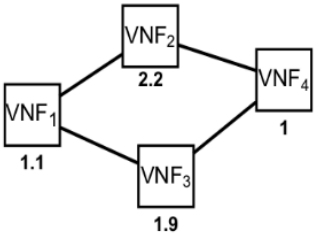


Distance between the VNFs and compute nodes

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$

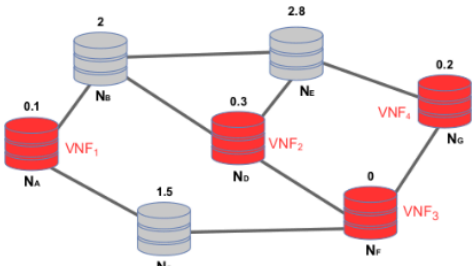


	N_A	N_B	N_C	N_D	N_E	N_F	N_G
VNF_1							
VNF_2							
VNF_3							
VNF_4							

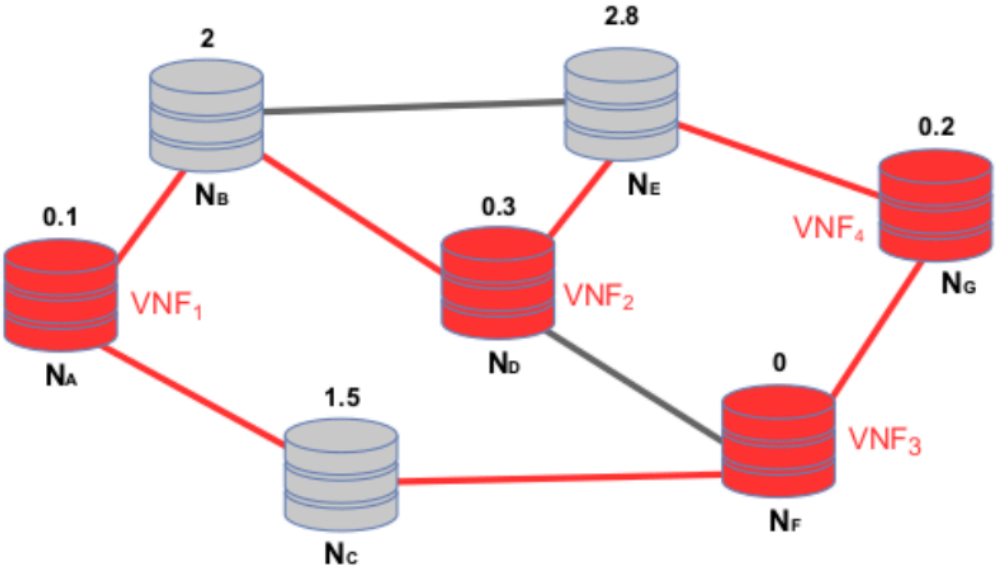


Distance between the VNFs and compute nodes

$$distance(VNF_i, N_j) = \begin{cases} (c^i_{VNF} - c^j_{NFVI-PoP})^2 & \text{if } c^i_{VNF} \leq c^j_{NFVI-PoP} \\ Inf & otherwise \end{cases}$$

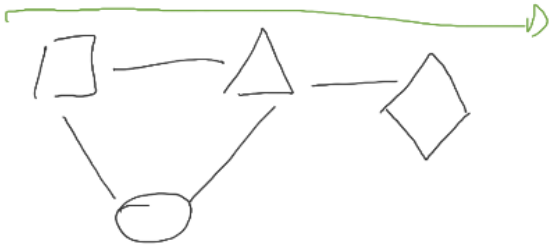


So now we have placed the VNF, and we have to connect them. For doing so each virtual link connecting two VNF is mapped over the shortest path in the substrate graph

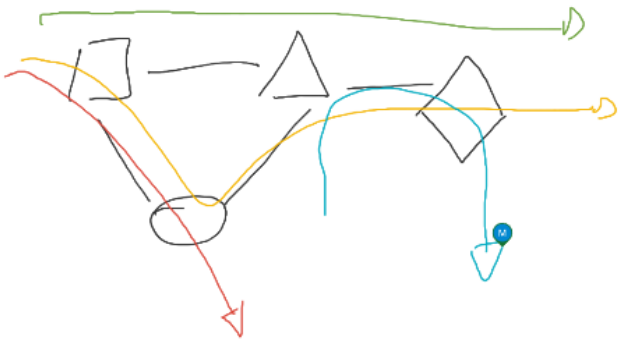


Service Function Chaining

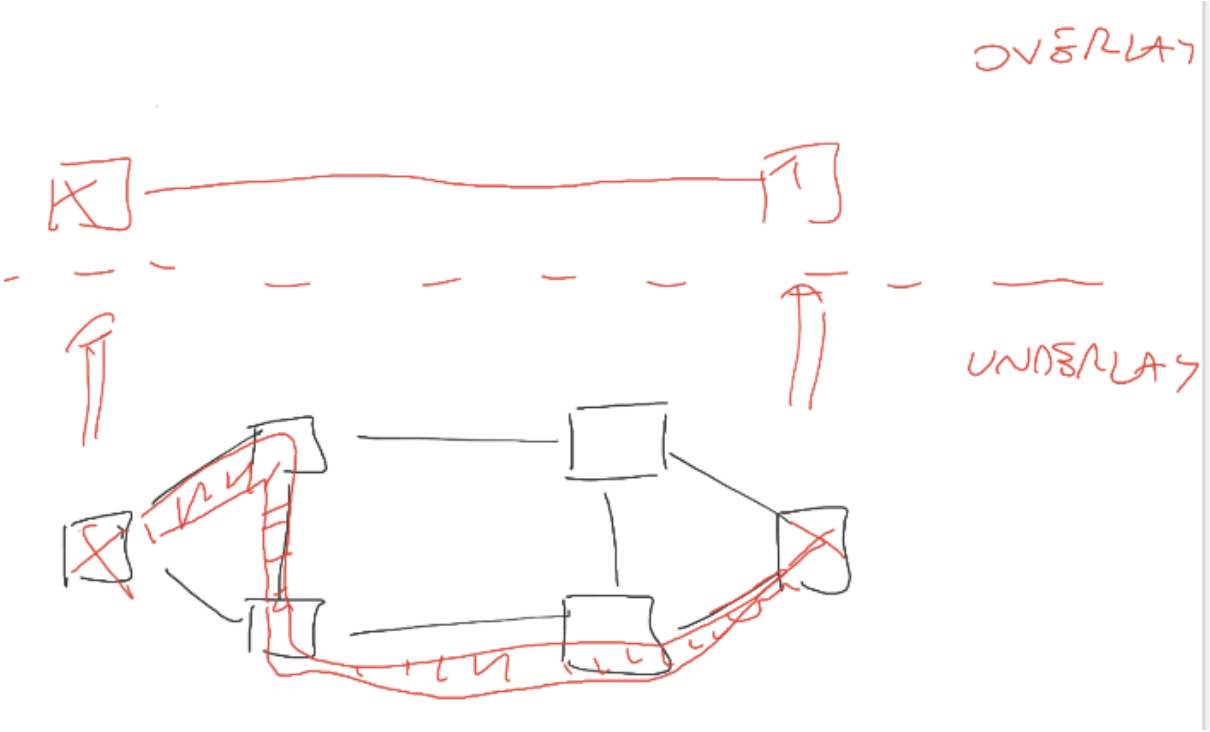
Now that we have placed the VNF, we have to connect them. Let's consider this FG:



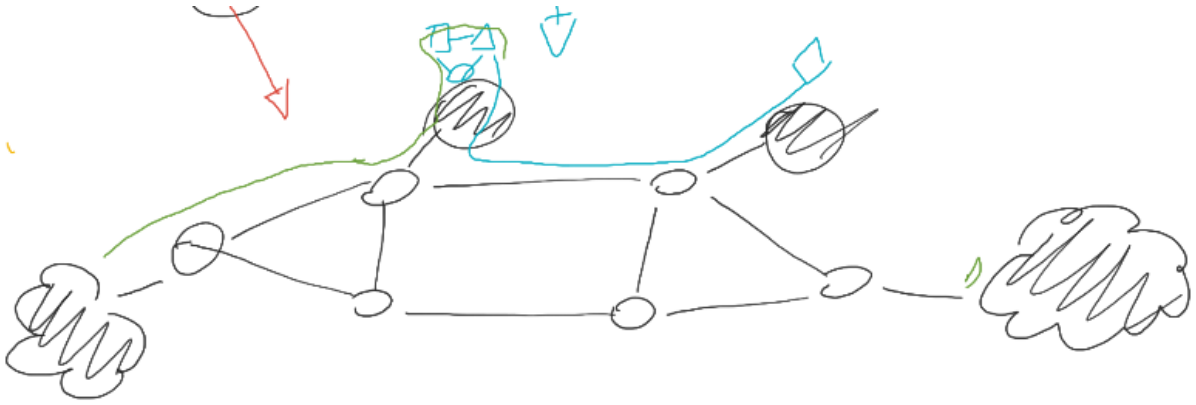
SFC is an architecture that steers the traffic flow in order to pass through an ordered sequence of VNF.



Overlay vs Underlay



The overlay network, the logical one, is built on top of a physical substrate. The physical substrate is this one, for instance:



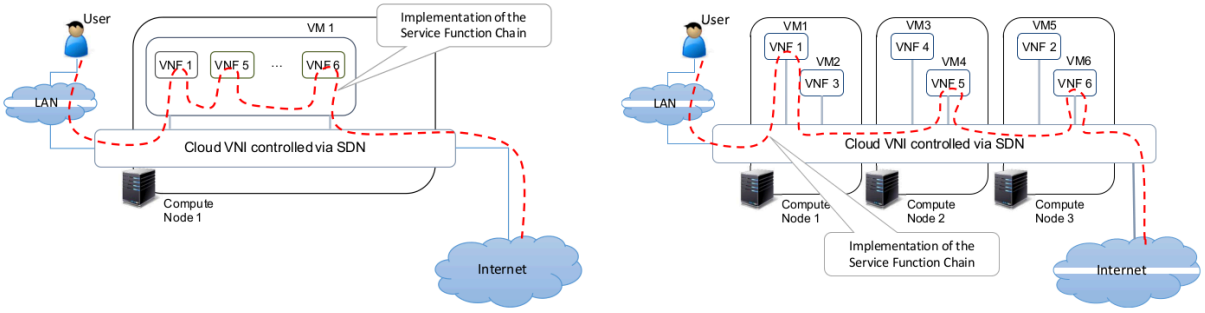
Something easy to see logically in the overlay can be difficult in the underlay.

Goals of SFC

1. How to interconnect two VNF? Namely, how to create this link:

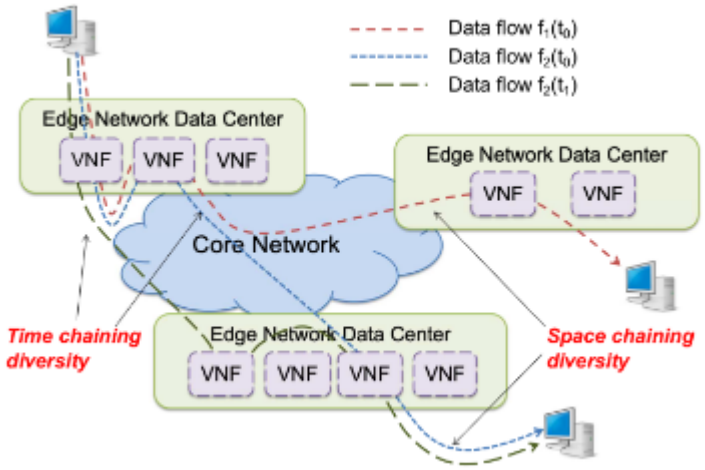
2. How can I force my traffic to pass through the ordered sequence?
3. How can I migrate a function to another DC, reshape the network based on the situation and reconfigure everything? We do that with Automation and SDN.

Let's consider these two different implementations of a SFC:



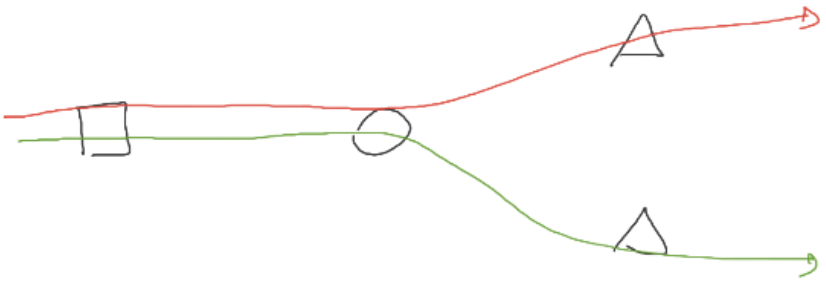
In the image above we have two different implementations of the SFC. In the second one I have to pass through different physical machines and they are not even in the LAN, so I need traffic steering to break IP classic forwarding because if I make source and destination node to communicate directly, the traffic flow won't pass through the local DC, where the VNF are. We will break IP classical forwarding using NSH or SR.

Space and Time Diversity

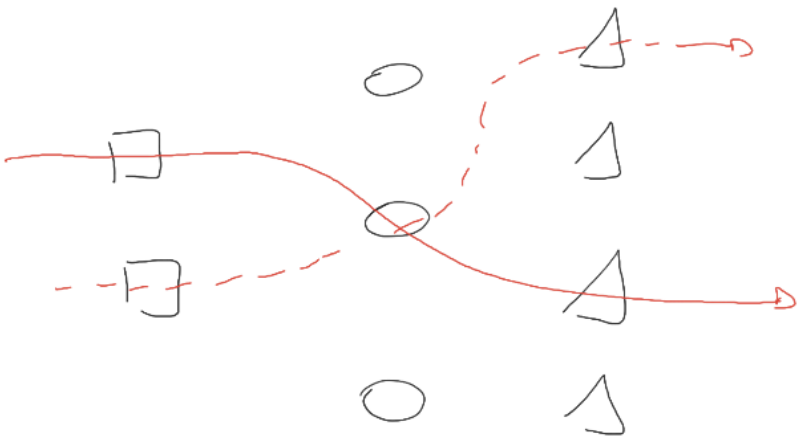


We have:

- space diversity: different flows in the same time period are in different instances of the VNFs. For instance, we can duplicate a VNF creating two instances in order to balance the load. In this way two flows can pass through the same VNF but in two different instances diverse in space



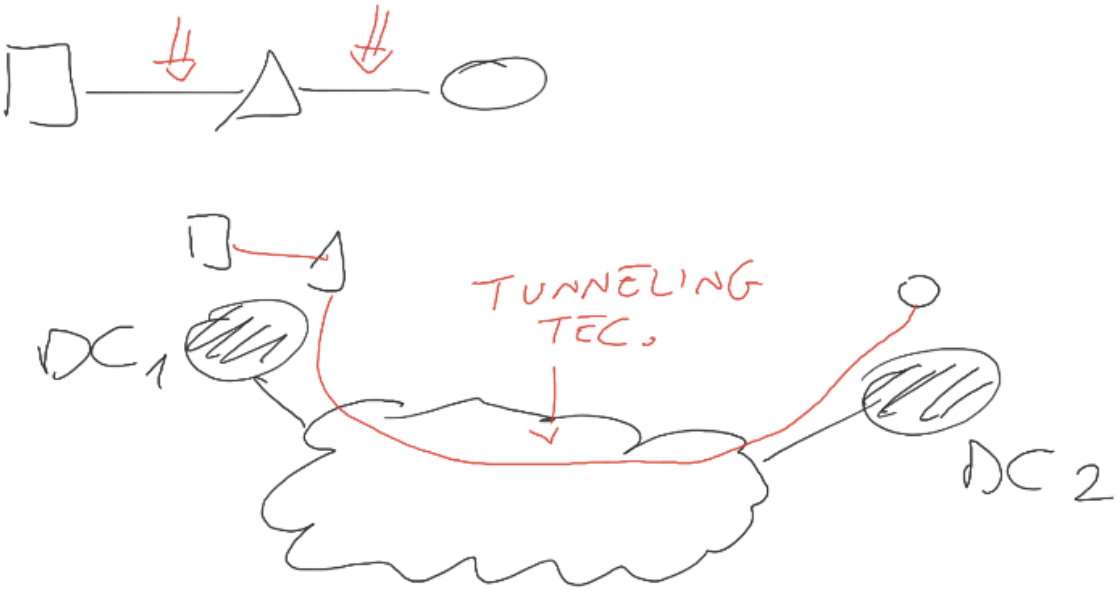
- time diversity: the same flow in different instants is in different instances of the VNFs For instance in two different times the same flow can pass through different paths, so it can follow two different SFC.



-

SFF, Classifier and Proxy for traffic through the Tunnels

To create the interconnection between two VNFs, we can use tunneling technique in the underlay:

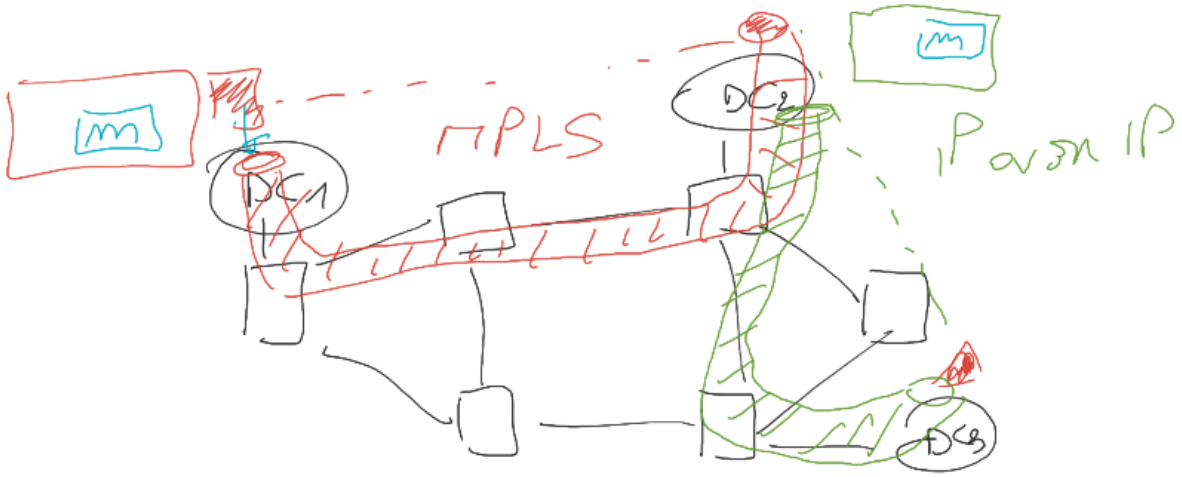


So we encapsulate the message in the packet that is suitable for transmission in the underlay network that is intermediate between the two VNFs. Valid encapsulation protocol:

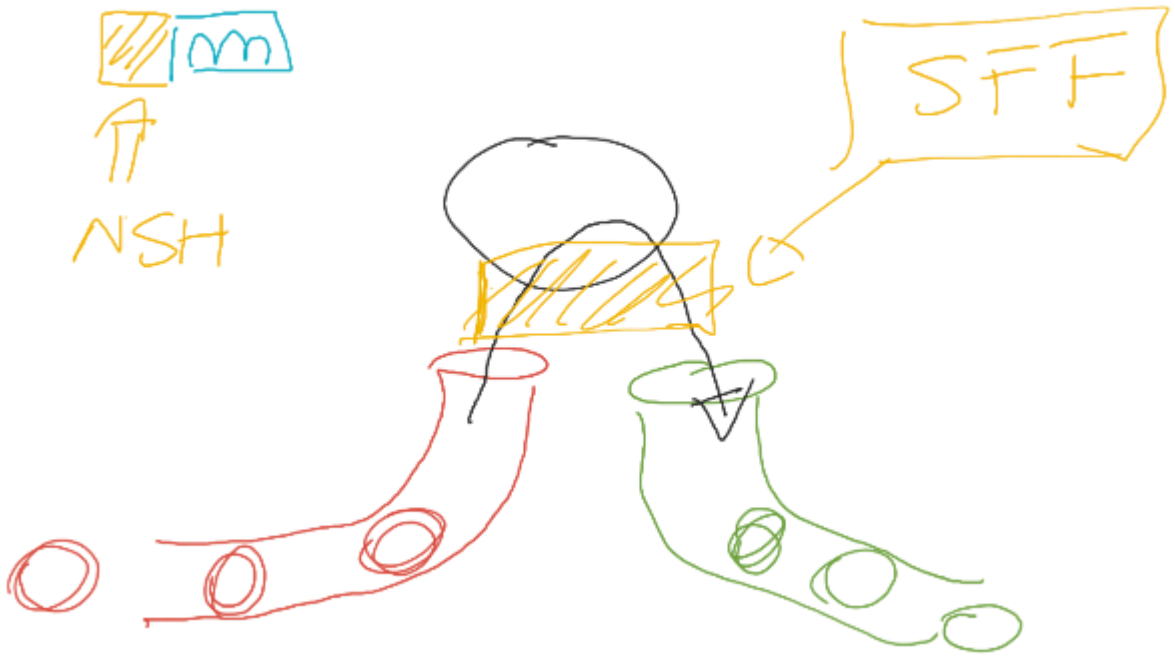
- IPSec
- MPLS
- VXLAN
- GRE

The good thing about this is that we can rely on different techniques simultaneously. In fact, let’s consider this example:

We encapsulate using MPLS at the start and decapsulate at DC2. Then we will encapsulate again but with a different protocol and go over the green path to reach DC3:

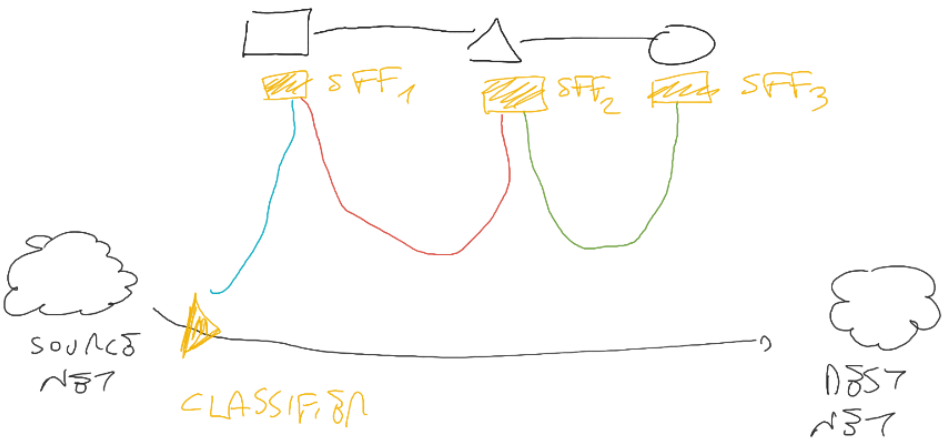


NSH (or SR) is used to forward from one tunnel to the other. This is called SFF forwarding, done by a switching element in the overlay that decides based on the NSH (or SR) header.



Precisely, we have:

- The SFF, the switching element in the overlay:
- Service Function (synonym of VNF)
- SFC classifier: it redirects the packet from the source network to the first SFF. Basically it matches over the fields of the packet and decides which SFC to use, namely its action is NSH encapsulation.



A good candidate for implementing the SFC classifier is an Open vSwitch.

- SFC proxy: a SF may not be able to process the NSH. In this case we use a proxy that stores the header. The NSH-unaware SF does whatever it has to do with the message and after the proxy reapplies the NSH.

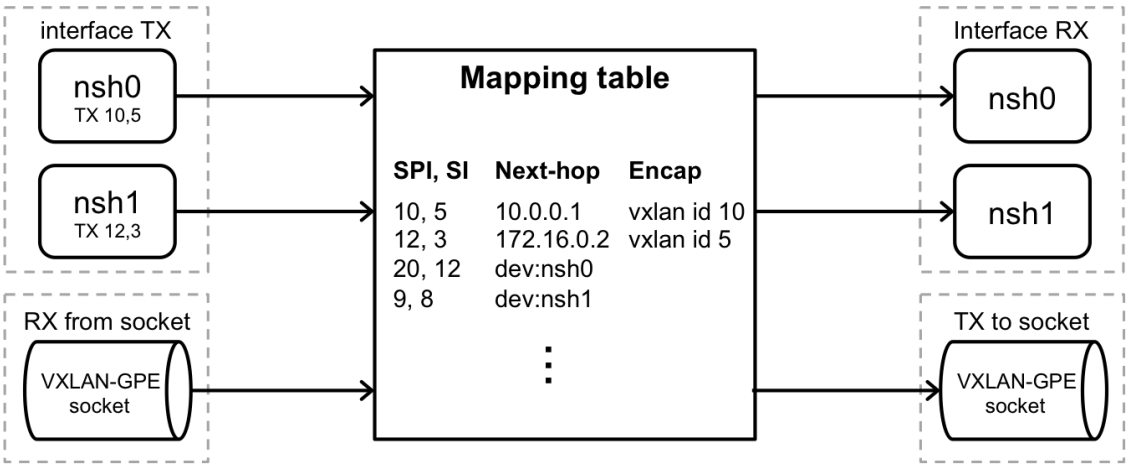
Network Service Header (NSH)

We want to make the history of the packet relevant. For doing so we add the NSH. The NSH-encapsulated packet go through a Mapping Table having as entries:

- The SPI: identifier of a SFC
- The SI: sort of a time to live in the overlay. Whenever a SFF forwards the message the SI is decremented by one.

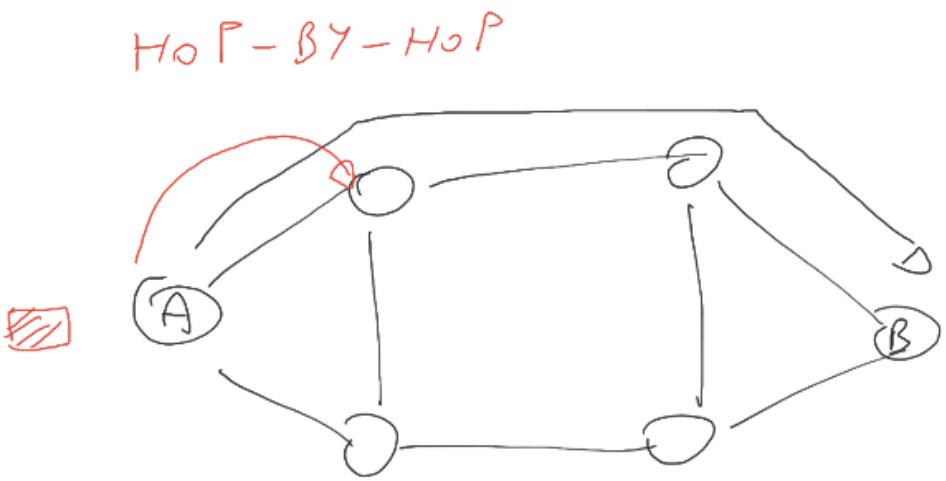
A SPI/SI pair corresponds to:

- a next-hop IP address with an encapsulation type for the tunnel
- or a new nsh interface

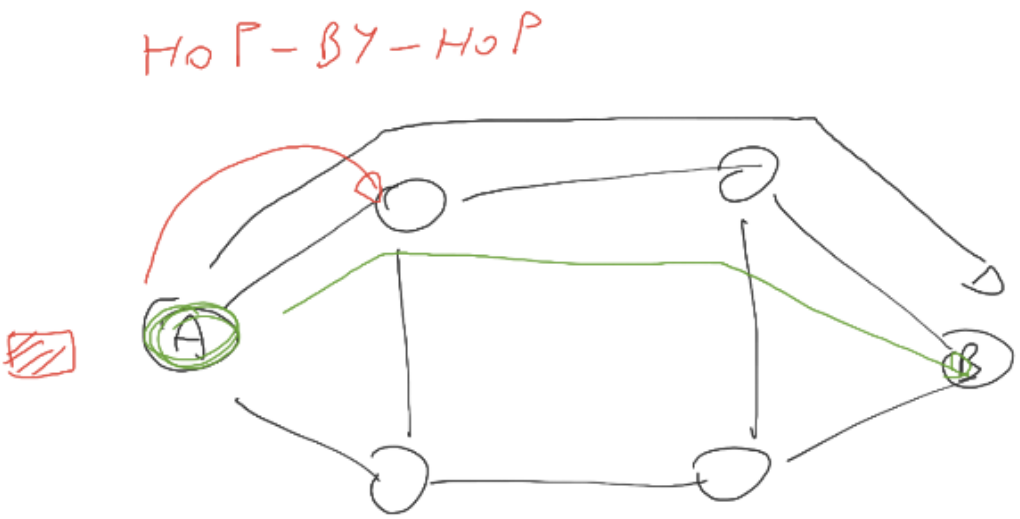


Segment Routing

Segment Routing is used as an alternative to NSH for SFC. Is an architecture that implements the source routing paradigm. In a simple IP network a path from A to B is the upper one, one of the two that are the shortest:



The control plane establishes the path, I am node A (a router) I receive a packet, what can I decide? It's just the forwarding, since it can choose only the next hop. The node A doesn't know which is gonna be the path, just the next hop. On the contrary, in the source routing, the node A, that is the source node, will decide what will be the path to follow. Enforcing the packet to pass through that path.



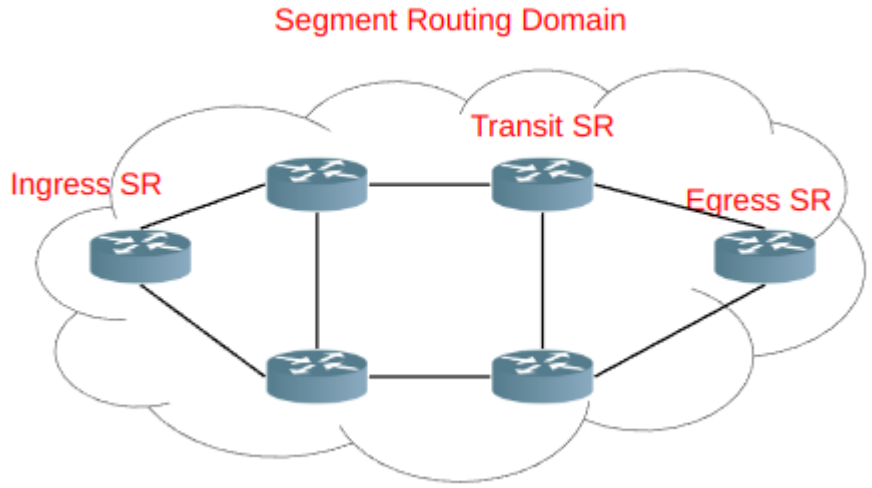
Topological, Service-Based, Local and Global Instructions

In SR we apply a sequence of instructions to a packet: we can have two type of instructions (also called segment):

1. topological instruction: action related to forwarding (take a packet and send it out using this link)
2. service-based instruction: modify a field, encrypt and so on…

Every instruction is assigned a label. The label is called segment identifier or SID, just a number. Furthermore segments can be:

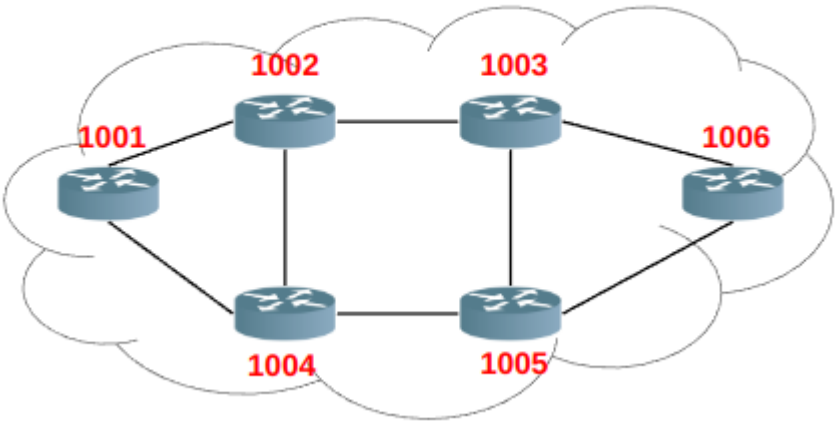
1. Local
2. Global



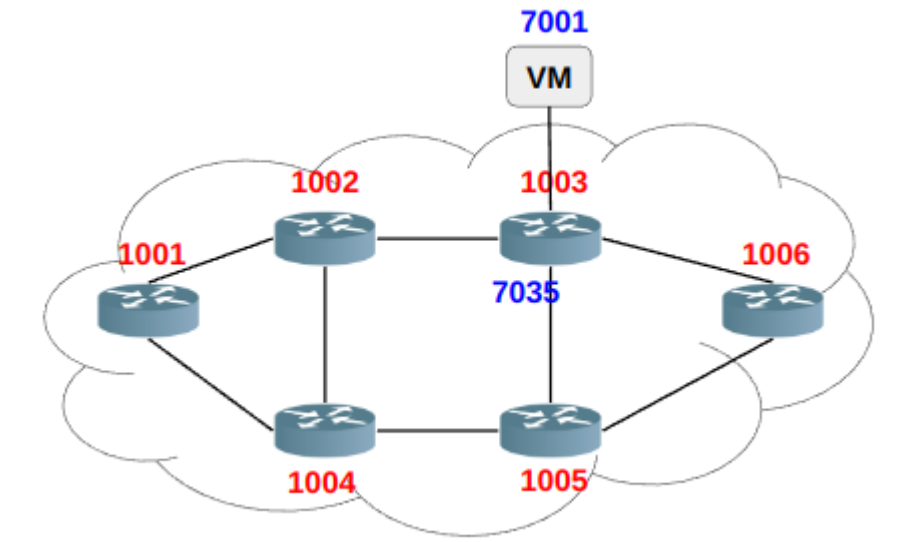
An SR-capable router has a SR processing pipeline to process SR header. We distinguish:

1. Ingress/Egress node
2. Transit Node

In SR domain we can label each node with a SID



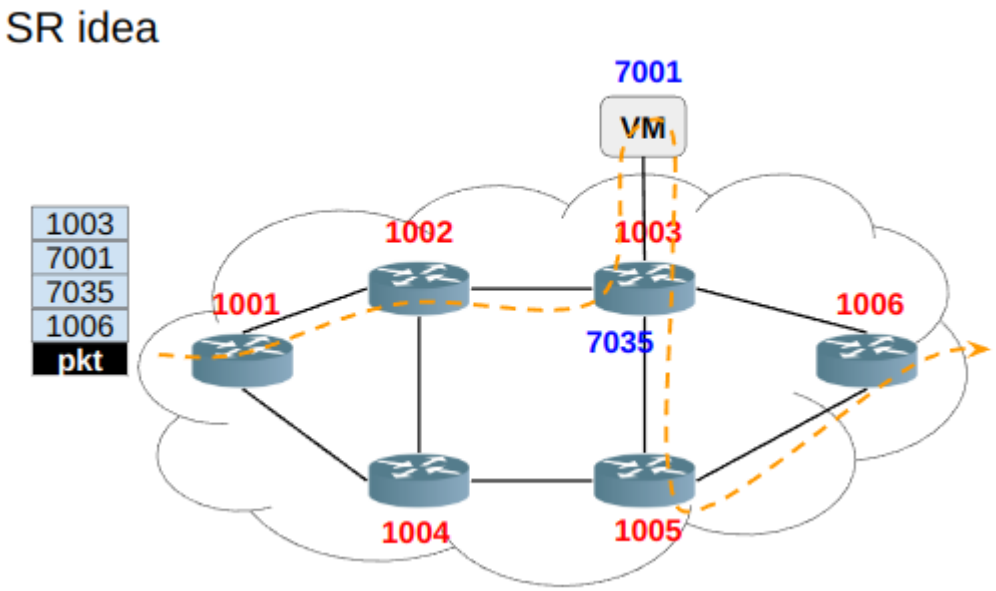
The instruction 1004 is “reach the router with SID 1004”, and it’s topological and global



7035 is a topological instruction. 7001 is a service based instruction.

SID List

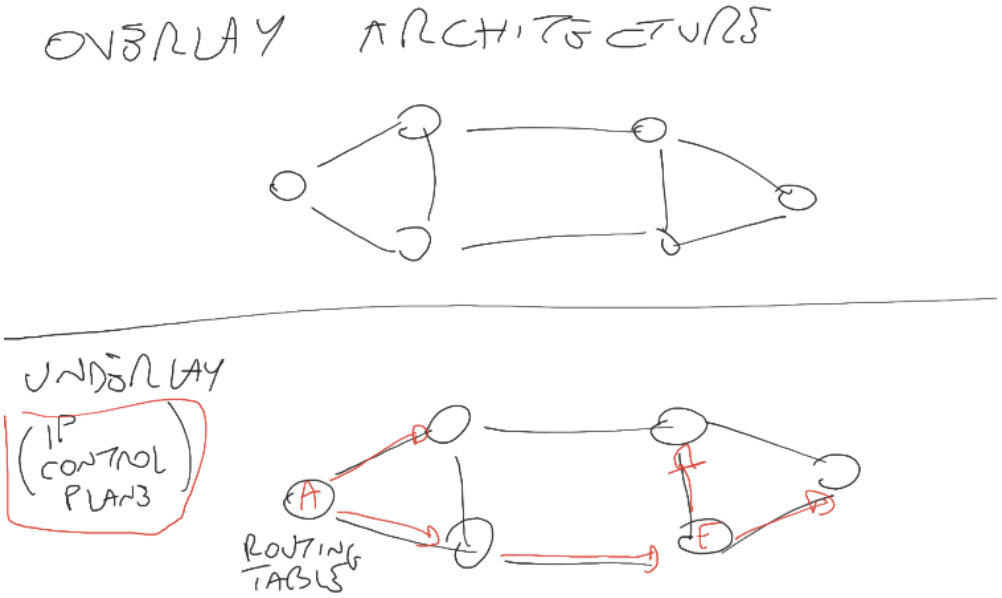
Consider this example below. Our goal is to enforce the routing path in yellow. So the one on the left is the corresponding segment list needed to achieve our goal. The first thing 1001 does is to perform 1003.



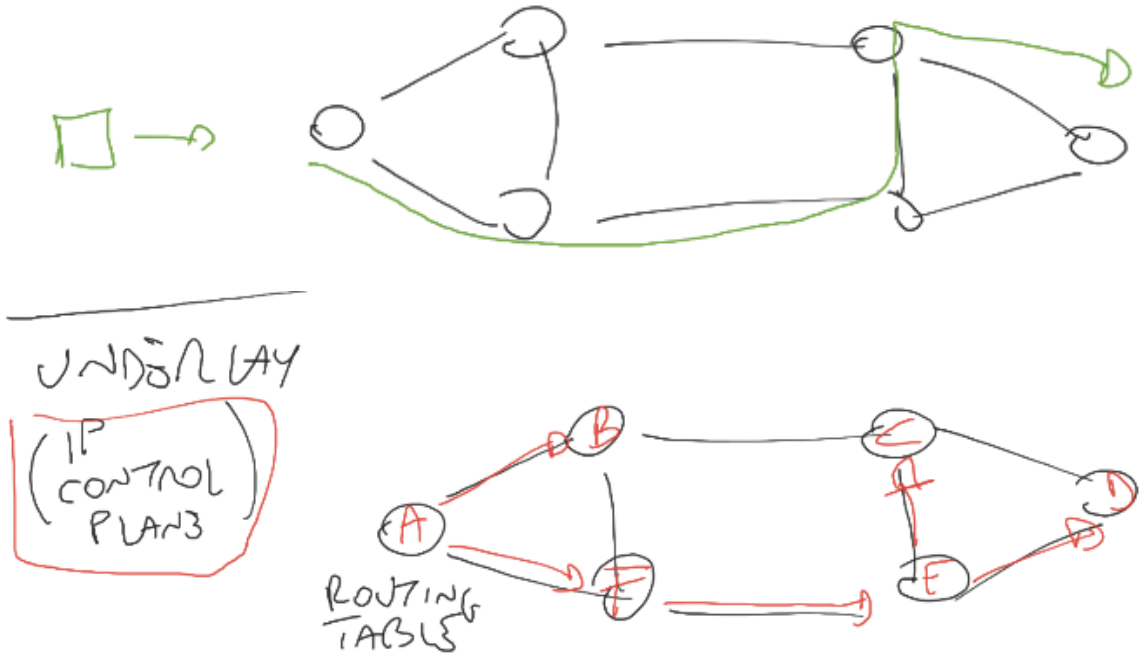
You can imagine that in the segment list there is a pointer that points to the active segment. The instruction that is currently under execution. When we reach 1003 the pointer is updated, but before, when the instruction to be executed is 1003 and we are in 1002 the pointer is not updated, so the instruction is not yet completed.

Overlay and Underlay

How does the node 1002 decide to go to 1003 if we don't have a precise indication? We are assuming that 1002 knows the path toward 1003 thanks to the underlay base on the IP control plane: basically 1002 knows the shortest path to 1003. This is an example of loose routing, in which we declare only intermediate nodes.



Imagine we want to realize this path in the overlay:

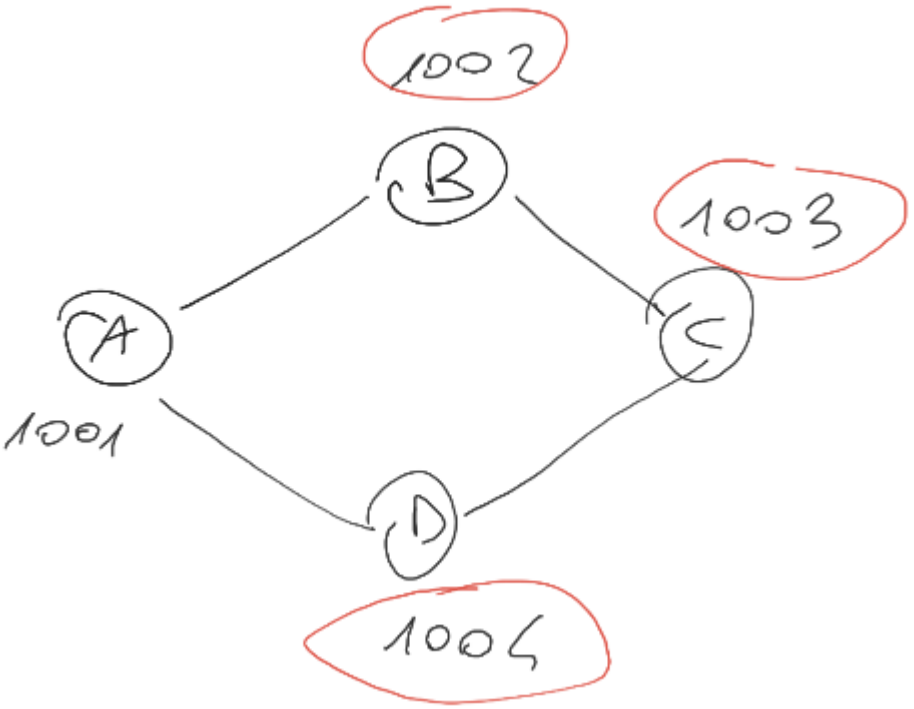


A possible (one of the possible) SID list is ECD. A better is CD[11], since it's shorter. A strict explicit path is FECD, but is longer so the drawback is the overhead. When we insert the SID list CD, we let the underlay process the instruction, and A will forward the packet to F, E and finally C, then we go back to the overlay where there is the node C, that updates the pointer and changes the active segments. Then it delivers the packet to the underlay that uses its path to complete the instruction. How can the overlay pass the packet to underlay and vice versa? With encapsulation and decapsulation.

SR Control Plane

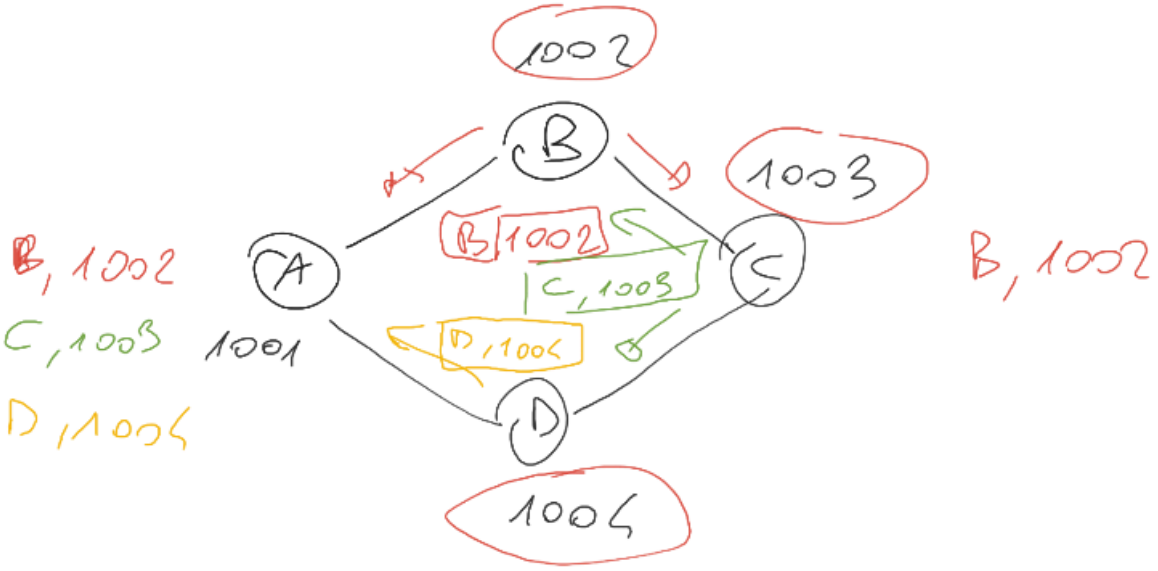
SR has to define some functionalities for the control plane. We have three different type of architecture:

1. Distributed Control Plane: in which segments are allocated and signaled by OSPF (or BGP) in order to make each node aware of the other node SID and each node individually computes the source-routed policy



Node A is currently unaware of the other SIDs. We can rely on OSPF protocol, through flooding of Link State Packet. An LSA contains the prefixes of the node connected to the node. So B will send an LSA containing 1002 to A. So A will know now that B has the SID 1002, and so on for the other nodes.

[11] Since A knows the shortest path to C, it will not send the packet to B that is a dead end, but it knows that sending it to F then it will reach C



In the routing table of A I will now have:

2. Centralized Control Plane: segments are allocated and signaled by an SR Controller that also computes the source-routed policies for the traffic. The centralized node has the task of computing the segment list, finding good paths and installing in nodes rules called SR policy that allow the packet to follow specific paths. With the distributed approach we are limited in the possibility of choosing weird and complex paths, so this is useful for flexibility.
3. Hybrid Control Plane: complements a base distributed control-plane with a centralized controller. Es. when the destination is outside the IGP domain the SR controller computes a source-routed policy on behalf of an IGP node.

SR Data Plane

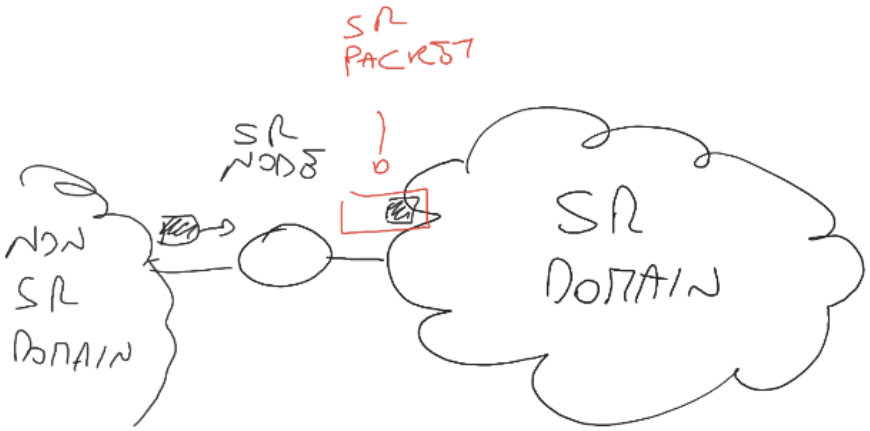
We can have an SR-capable device by reprogramming its data plane pipeline. There are already existing and widespread data planes in which we can run SR.

1. MPLS, so we obtain SR-MPLS.
2. IPv6, so we obtain SRv6. In this case we need a new header called SRH. We will see that later.

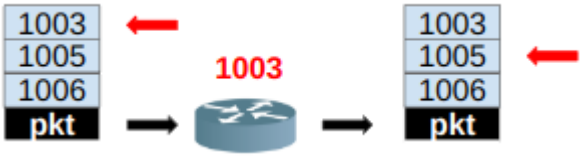
SR Forwarding Actions

What are the actions implemented in the SR dataplane?

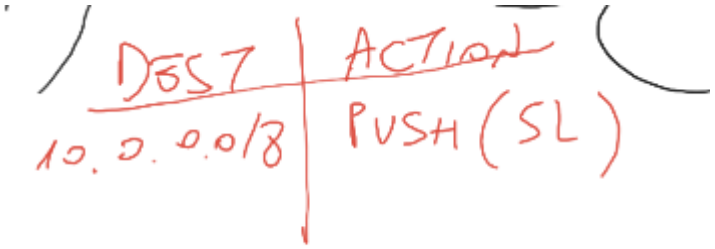
1. PUSH: let's see this example



When a device of the non-SR domain wants to inject the packet in the SR domain generates a normal packet and the SR node will push the SID list in the packet, in order to travel in the SR domain.



The router has two tables: one is used inside the domain and the other one is used to associate the packet to a SID list in order to enter the SR domain.



2. NEXT: we move the position of the pointer.
3. CONTINUE: a node, identified with a SID that is not the active one, once it receives the packet will perform CONTINUE

The SR table has a column for the matching, and it's populated with SID. PUSH, NEXT and CONTINUE are the three possibilities.

MyLocalSID Table

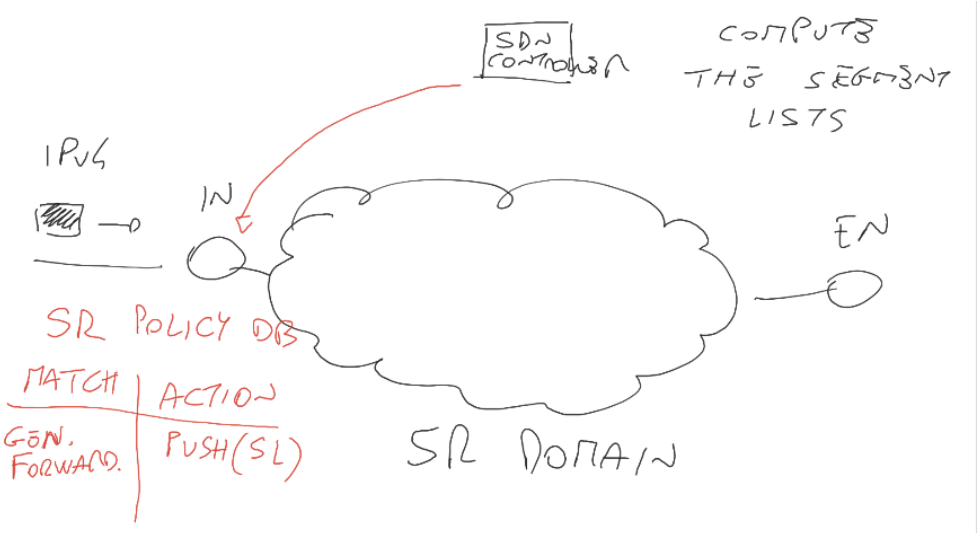
MyLocalSID Table

SID	instruction
...	...

Is present in every SR-capable node. It associates a SID with a specific instruction (or set of instructions). The instructions are not necessarily forwarding (topological) but they can also be service-based. We can also have complex functions like this one: assuming you want to get high performance, guaranteeing the smallest latency possible but you don't know which of the two possible paths is the best one. As soon as the packet arrives, you can perform a special action stored in the MyLocalSID Table. called LIVE-LIVE, that duplicates the packet and makes them pass through the two links. The packet following the best path will reach the egress node first, and the egress node, having a sliding window that makes us understand if a packet is a duplicate, will understand which path is the best. Another synonym of SR function is behavior.

SR Routing Policy

Let's understand it with an example

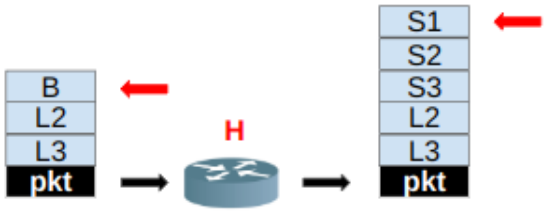


We have border nodes (ingress and egress) that receive packets from non-SR domains. We can imagine that there is a SR controller which computes the segment lists. The SR controller configures, with OF or whatever, an entry inside a specific table in the IN node that instructs on how to act for packets coming in the net. The table in which it is installed is the

SR Policy DB. The matching part of the rule is flexible, we can specify for instance source and destination of the packet, or protocol, or whatever: it's generalized forwarding.

Binding SID:

A policy it's an instruction, with its SID called Binding SID. When the node reads the binding SID it pushes another segment list in the packet[12]

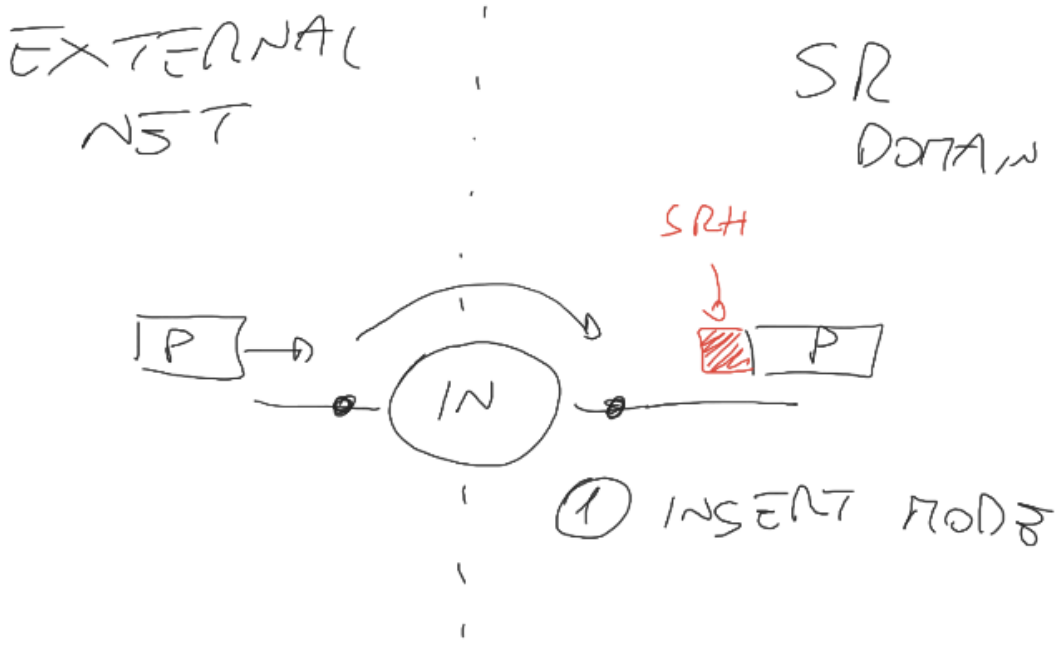


SRv6: SR over IPv6

Insert and Encap Mode

IPv6 is extensible by default, with an extensible header. The SHR is an extension header that can be included into the packet by the node that originates the packet or the ingress node of a SR domain. There are two ways of including a SR header:

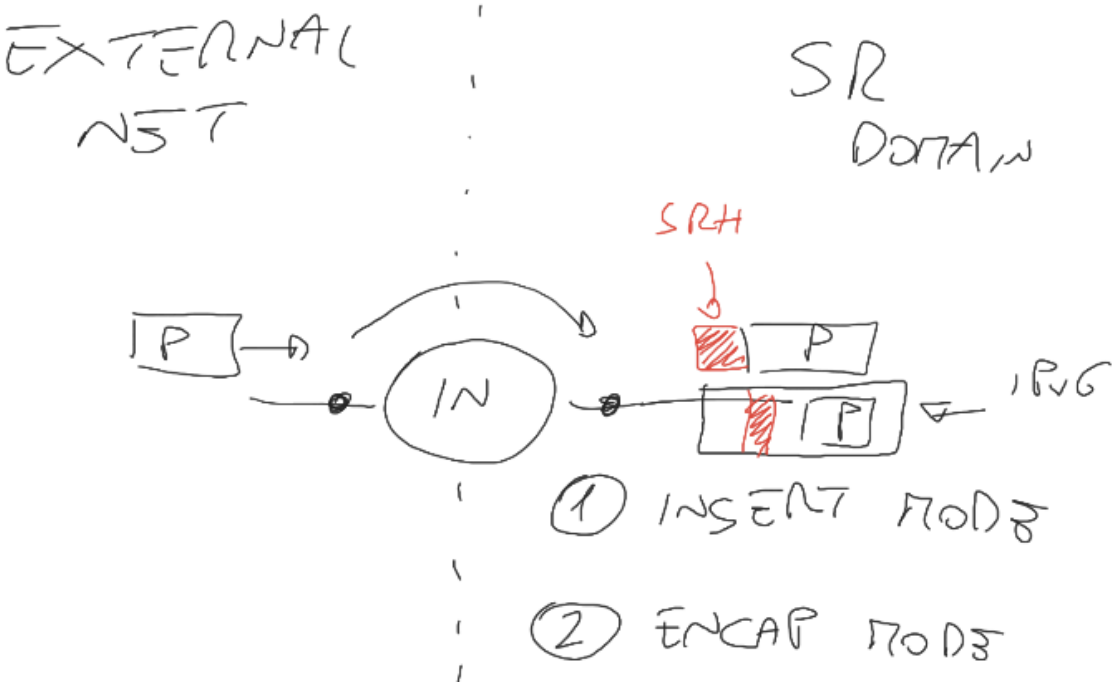
1. Insert mode: the SRH is directly attached to the incoming packet



There is an assumption: the incoming packet must be already IPv6. Here we are reducing the overhead w.r.t to the encap mode.

[12] This is useful for fast rerouting in case of link failure, for instance

2. Encap mode: the original packet is encapsulated inside an IPv6 packet that has also the SR header. So we are wrapping the packet in an IPv6 packet and the outer packet has the SRH.



Encap mode is the most common one, since it leaves the original packet untouched. It's mandatory when the packet is IPv4.

Segment Routing Header (SRH)

This is the SRH:

next header	hdr ext len	routing type	segments left
last entry	flags	tag	
segments list [0] (128 bit IPv6 address)			
.....			
segments list [n] (128 bit IPv6 address)			
optional type length value objects (variable)			

Adding a segment is adding 128 bits, so we want to have the path encoded with the shortest possible SID lists. Then we have the pointer, which specifies how many segments are left.

Mapping PUSH, NEXT and CONTINUE to IPv6 operations

The idea is that we want to replicate the behavior of the pure ideal SR but relying on an IPv6 dataplane. We map the concept of SID with an IPv6 address. How can we map the three actions with what we have in IPv6?

1. For PUSH we use insert or encap mode, things that are already available in IPv6

2. For NEXT, we have the END or END.x behavior. Let's see this example:

In red there is the active segment. The active segment is also the destination address of the IPv6 packet. As soon as the router receives the packet it understands that is for itself. Whenever we get to the destination node in a layer we have to decapsulate and perform END or END.X. This is END:

```
IF SegmentsLeft > 0 THEN
        decrement SL
        update IPv6 Destination Address with SRH[SL]
        FIB[13] lookup on updated Destination Address
        forward the packet accordingly
ELSE
        drop the SRH
        now we have a simple IPv6 packet
```

3. For CONTINUE: non SR-capable routers can't process the SRH, so they see the SRH header as payload (transparency), and they just do IPv6 forwarding (they don't read the blue part, just the green one).

Network Programmability with IPv6 Address

In SRv6, a SID is encoded through an IPv6 address, which is very long. We give a structure at this string getting high flexibility. By definition, sequence of instruction is a program, so the SID list is a network program.

locator *function* *argument*

1111:2222:3333:4444:5555:6666:7777:8888

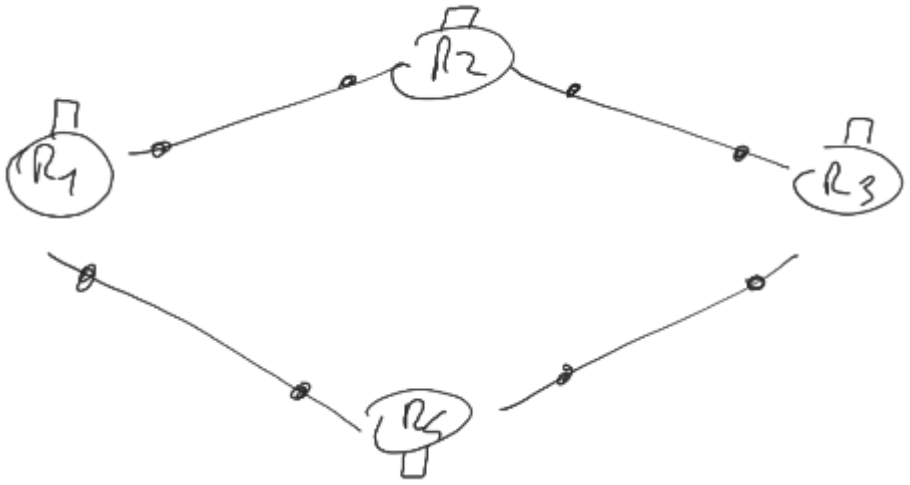
Each IPv6 address has:

1. Locator: tells which node has to perform the processing. The locator is topological and global.

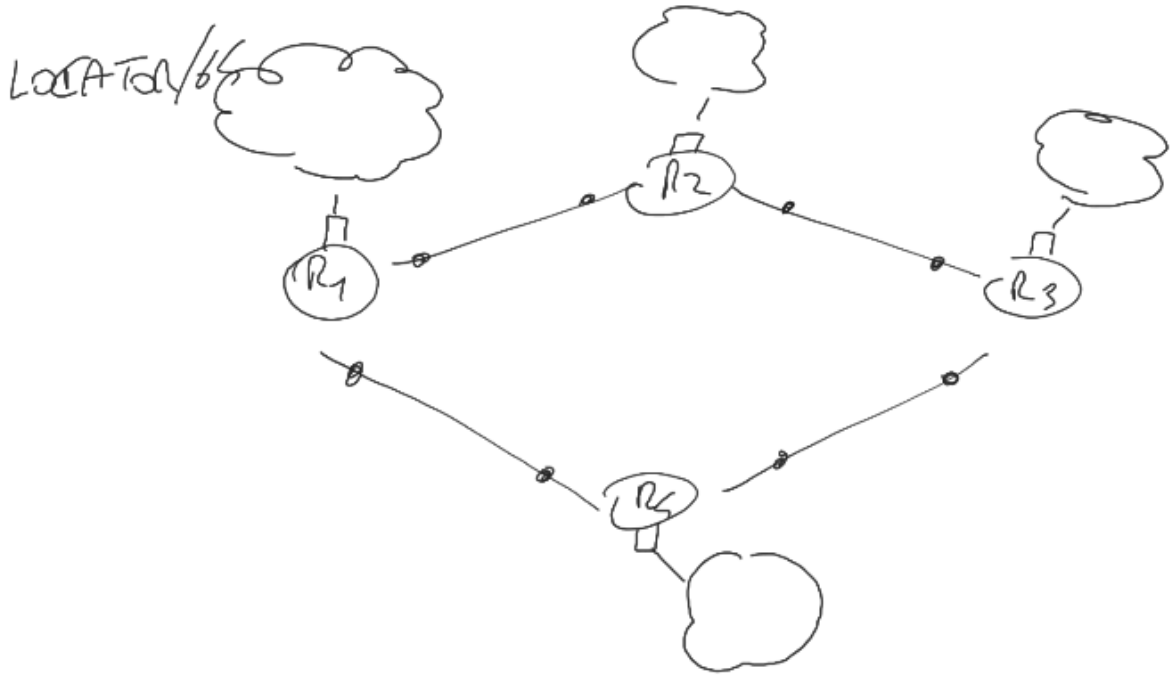
[13] the Forwarding Information Base, also known as Forwarding table or MAC Table.

2. Function: identifies the function to be performed. This is local and can be topological or service based.
3. Argument: specifies the function argument

This address will match on the MyLocalSID tables encountered in order to reach the locator and perform the requested function with the specified argument. Let's consider this example:



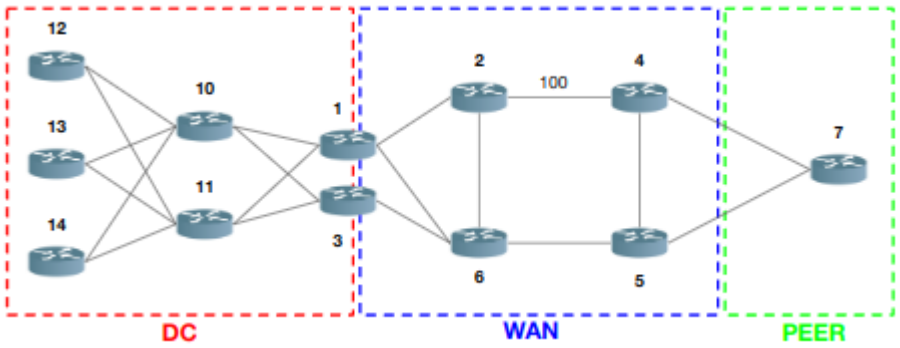
Each interface has its own IPv6 address. No one of this interface will be the locator, the locator is associated with the loopback interface. The loopback interface will have an IPv6 associated to a virtual network connected to the corresponding node.



We use OSPF, that is commonly used to share prefixes, to share this locator. If R1 wants to use a function in R3, I am not interested in the specific function, but just to know how to reach the locator R3. The My Local SID table will be populated with IPv6 addresses with prefixes coming from the locator. With this paradigm you can write programs such as a traffic steering program and TI-LFA.

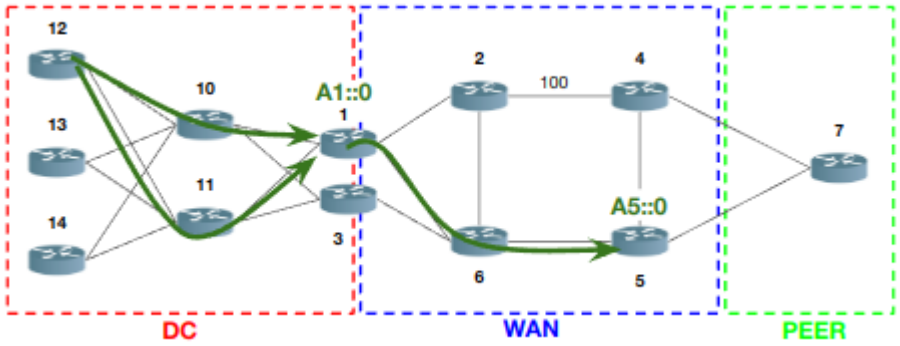
Use Case: Traffic Steering

Let's see this application:

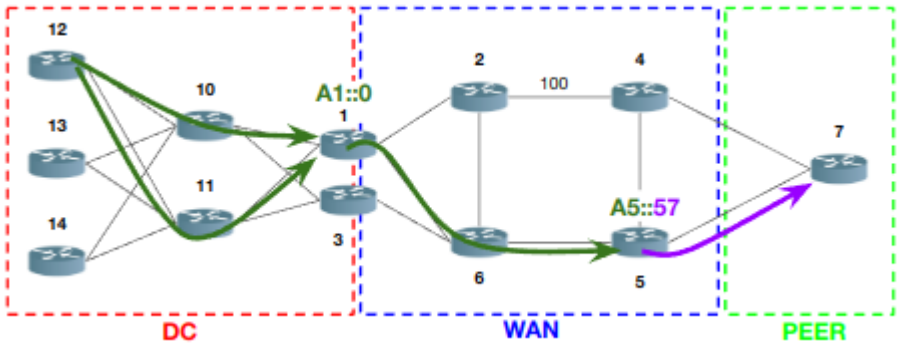


There is a high cost link, that is a parameter of the underlay. So the OSPF cost to pass to that link is very high, so Dijkstra won't use it. The locator will be AK::/64. Two behavior:

1. END
2. Cross Connect: END.X, send the packet out through a specific interface. Identified with CJ, C specifies that it's the cross connect and J is the next node we want to visit.

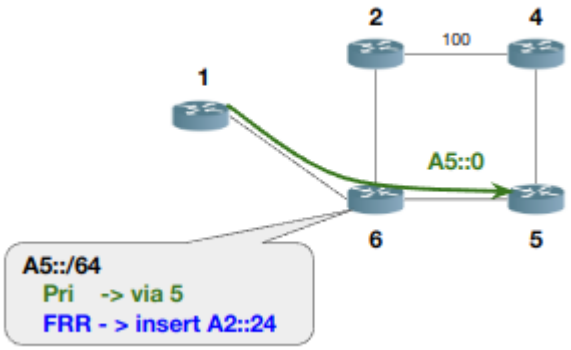


The underlay provides the connectivity between 12 and A1::0. If in the underlay there are multiple paths to go from a node to another, SR allows us to balance the load if they have the same cost. Once we arrive at A1::0 the active segments become A5::0. Finally we want to use the link between 5 and 7, so we use the cross connect A5::57



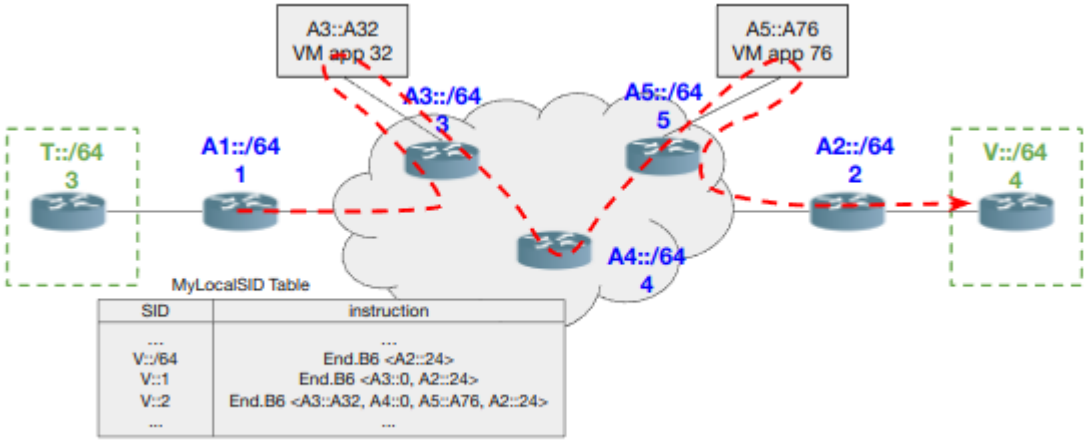
Use Case: Topology Independent Loop Free Alternate (TILFA)

By means of a binding SID we define a policy that says “in case the link fails insert inside the incoming packet the network program A2::24, that means go to router 2 and inspect the My Local SID table to search for ::24 that is a function that forces the packet to pass through the costful link”.



Using SR for SFC as an alternative for NSH

Interconnecting different elements of a forwarding graph.



Imagine we have a SR domain with two VNF deployed in two VM. We have two customers (in green) that need to communicate through the WAN, in a VPN. We want the traffic to be steered to the two VMs. We have a policy that says that if a packet is destined to V::2, a machine of the other customer / branch of the enterprise. We should perform:

1. Go to A3 and apply the function A32
2. Go to A4 and do END behavior.
3. Go to A5 and perform A76
4. Leave the network through the egress node and use the connection from 2 to 4.

Programmable Data Plane

18/04/2024, 15/05/2024, 16/05/2024, 22/05/2024

Programmable Dataplane

SDN and OF for Programmable Data Plane

In the data plane we perform packet forwarding and processing in the time of nanosecond, so we must be efficient. The SDN approach is not optimal since:

- OF switches are not stateful
- With OF the number of tables is fixed (around 250)
- You can't create new protocol/header[14] and actions, unless doing it in the controller, but that would mean sending each packet in the controller creating too much overhead

So the generalized forwarding is not flexible enough.

Network Softwarization for Programmable Data Plane: Flexible but Low Performance

The Network Softwarization approach is not optimal in Dataplane since it leads to loss of performances that are fundamental in the dataplane.

Horizontal Scaling

When the user demand increases we create a new instance for the service. There is an upper bound that we reach in performances when we scale horizontally

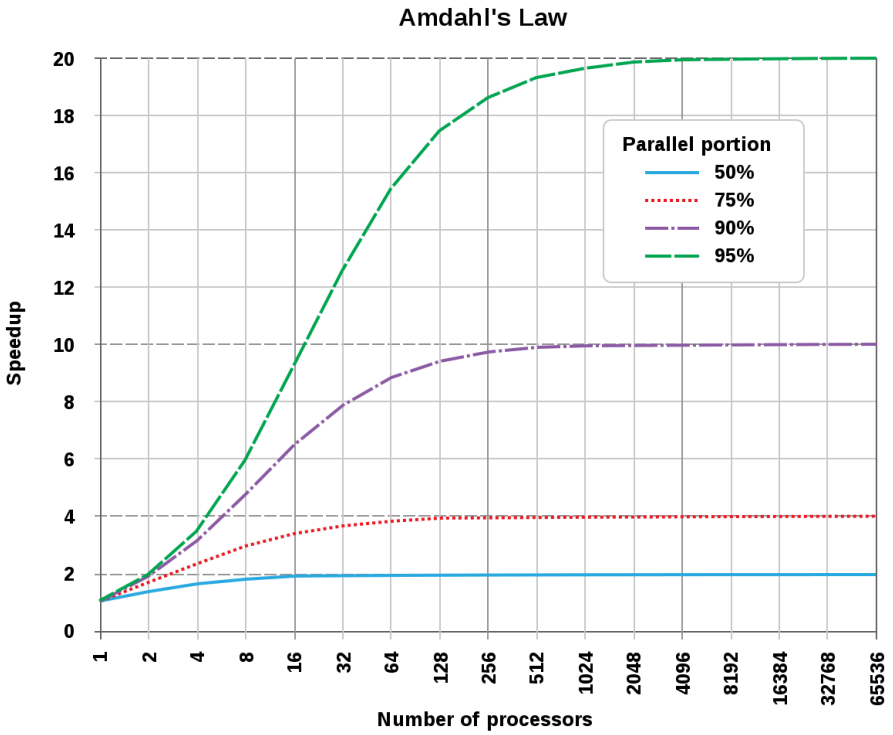
[14] For instance NSH is not supported, at least in the 1.0 version of OF. 1.5 OF supports NSH, so you need to release a new version of OF every time you want to include a new protocol

So we can use it to increase performance but there is a limit. This limit is given by the Amdahl Law.

Amdahl Law

When we perform a parallelizable task, there is an upper bound. Specifically, depending on the number of CPU and how parallelizable is our task we have a sped up factor.

$$\text{speedup} = \frac{1}{(1-F) + F/N}$$

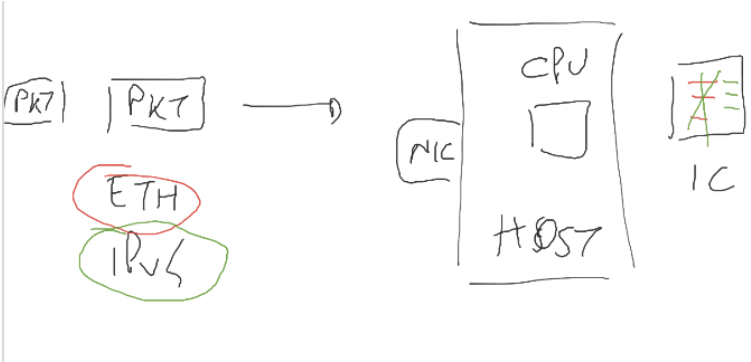


Software Acceleration Technique (DPDK and VPP)

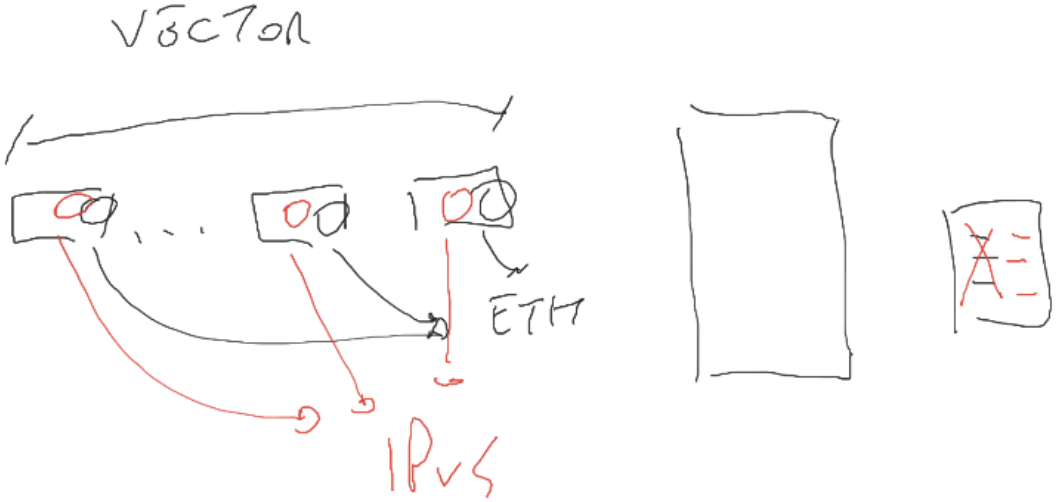
A better network softwarization technique than horizontal scaling is the software acceleration technique. For this second approach we have:

- DPDK: intel library that performs the kernel bypass. All the networking things in the linux machine are handled by the kernel. It is inconvenient to pass through the kernel so I allow the app to directly access to the NIC, bypassing the kernel. I can assign one interface to one application, when the kernel I am able to share the same interface for multiple applications, so for DPDK there must be a lot of interfaces.

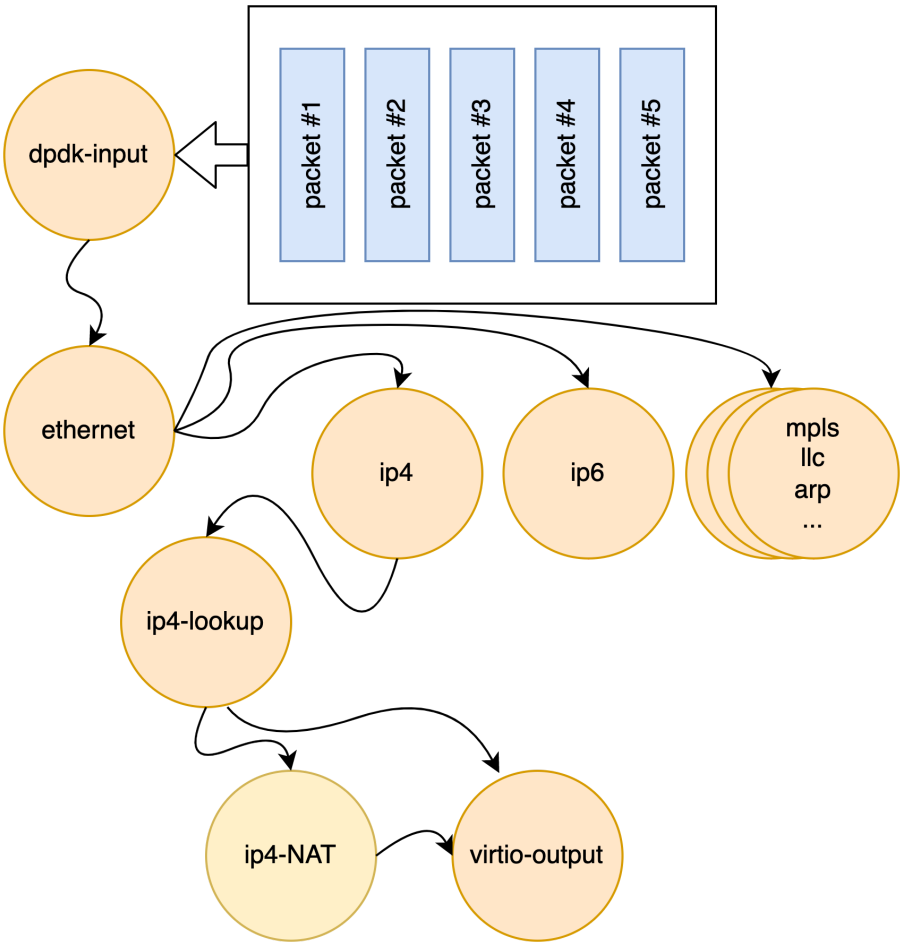
- VPP: Vector Packet Processing. Example: when the packet arrives, with its header, say Ethernet and IPv4, the first thing done is to inspect the Eth header, loading the corresponding instruction in the Instruction Cache (IC). Then I process the IPv4, removing and adding the new instruction in the IC. This leads to more CPU miss and so it wastes time, since I have to load new instructions every time in the cache.



Since after the packet there is another one that will likely have the same structure (Eth and IPv4 header), I can use a vector of different packets and extract the Eth Header from all the packets in the vector. I perform the processing for the Eth header for all the packets once and then I do the same for the next header, in this case IPv4.



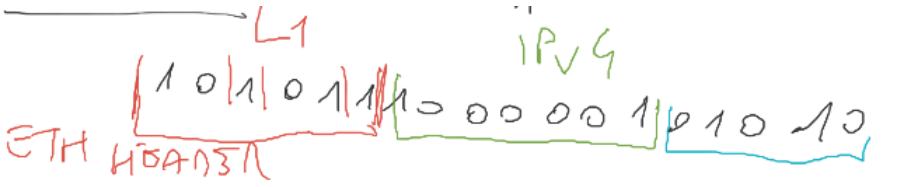
The diagram below shows the processing of the vector of packets: being in one of the nodes means that you are processing that header at that moment.



Conventional Data Plane: High Performance but not Flexible

Each device has:

- The control plane, in charge of establishing the packet processing policies
- The data plane, that just executes the packet processing policy. To enforce the decision of the control plane, the data plane does the following:
    1. Parsing of the packet bytes. For instance, when an Ethernet packet arrives, the router knows that the first L1 bits of the packet are for the Eth Header.

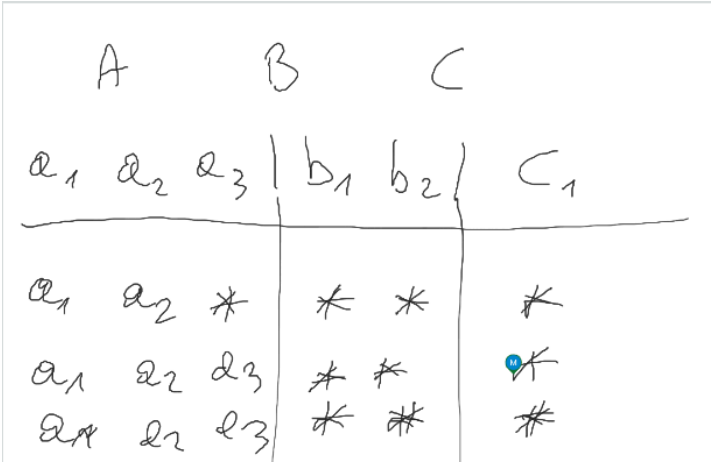


2. Processing operations, such as checksum calculation, counter and also inspection of the match-action tables.
3. Forwarding of the modified packet based on the result of the performed operation

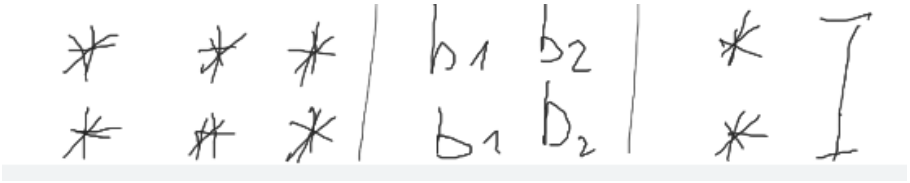
SMT vs MMT

After parsing we have a match and action in the routing table. A match table can be:

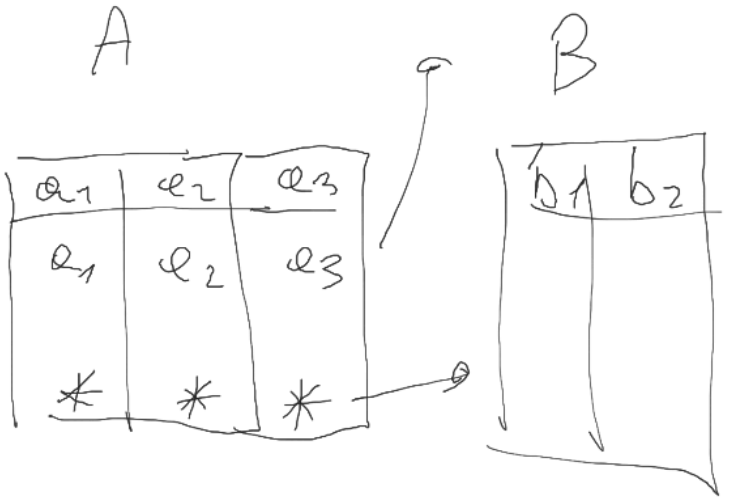
- Single Match Table (SMT): all fields of the header are in one table. For instance, the L2 table is SMT.
- Multiple Match Table (MMT): each table has a subset of headers in a pipeline of stages. For instance, OpenFlow is MMT. In general having multiple match tables is more convenient, since they allow us to use the resources more efficiently. For instance, imagine having three headers.



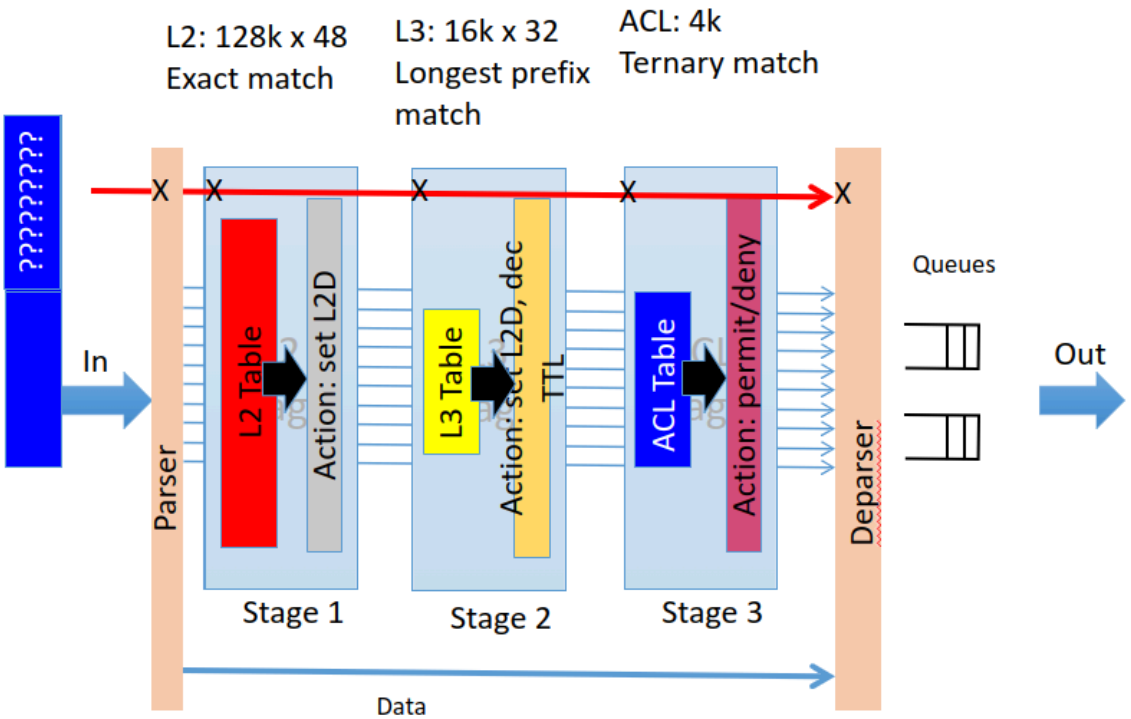
Note that * are wildcards. So for these first three packets I just want to know the header. Assume we have some packets for which we want to know just the header



Look how much space we are wasting in our memory to store wildcards. We solve this with MMT:



In a MMT pipeline we can leave the pipeline and just go to another table without having to pass through each table. The last row is saying “look that if you don’t care about the header A you can pass to table B”. An example of MMT fixed function switch is this one:



We have three blocks of match and actions, the red arrow models the so-called metadata: while the header propagates through the pipeline I can associate some metadata to the header. The vector of headers passing through the pipeline encounters tables that can have different sizes (e.g. L2 is bigger than L3, so it has more fields). For L2 there are 128k rules, so it’s quite long. The last is an access list that allows us up to 4k rules. In the L2 table we may want to do exact matching: the header and the rule are equal or I declare that the rule doesn’t match. On the other hand in L3 we rely on LPM. In the ACL we are using ternary match, so we allow the presence of “don’t care” information. So it’s like LPM but in the ternary we don’t decide based on the length of the prefix but something else: priority, as in firewalls.

Note that this is a fixed function switch, so we are not reprogramming the data plane. So even if we have high performance, in contrast with the softwarization approach, we are not flexible enough.

High Performance and Flexibility: Tofino and P4

The tradeoff between the flexibility of software and high performance is solve using:

- ASIC for high performance: Intel Barefoot Tofino chip, based on the match-action paradigm.
- Degree of Programmability with a domain-specific language: Programming Protocol-Independent Packet Processor (P4)

Programmable Data Plane Definition and Architecture

> *programmability refers to the capability of a switch to expose the packet processing logic to the control plane to be systematically, rapidly, and comprehensively reconfigured*

It means that our device is not saying to the control plane “look this is the way I process the packet and that’s it” is more “this is the process pipeline and these are the points you can change”.

The three degrees of programmability that we want are:

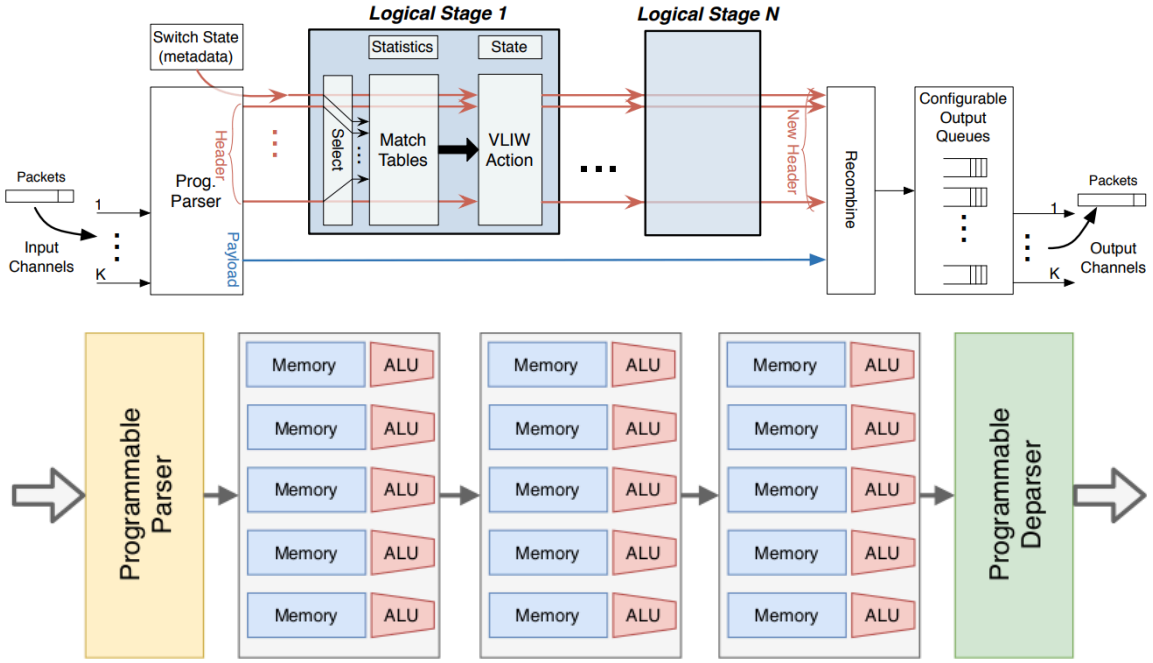
- The parser: useful to add new protocols, like NSH.
- Stages of our process pipeline. We might want to increase the number of stages, change the size of the table w.r.t. to the rules and fields. We can change the type of matching (LPM changed in exact matching)
- Add new actions

Dataplane architecture that supports the dataplane programmability:

- Protocol Independent Switching Architecture (PISA)
- Open State: OF into a programmable pipeline, trying to include statefulness in the processing

Both PISA and Open State use CAM and TCAM.

PISA Architecture



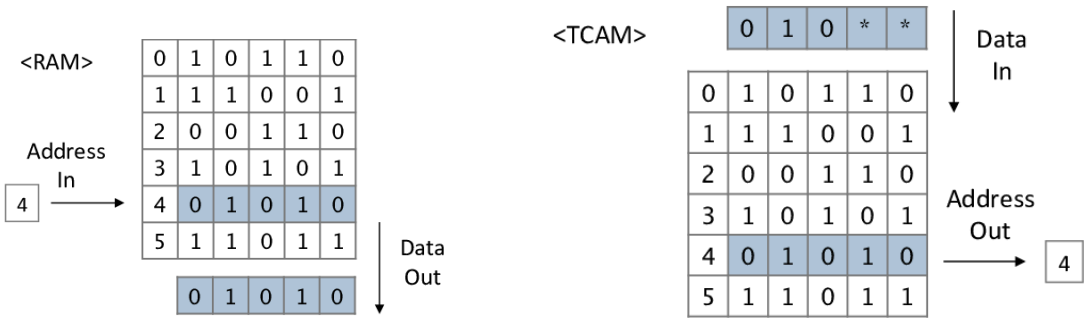
Each of these logical stage is an RMT, containing:

- Memory: a TCAM
- ALU for calculations

You can modify the number of stages. In each stage we do matching with the RMT and we parallelize the action thanks to the VLIW CPU. Moreover, we can add metadata. The payload goes directly to the deparser.

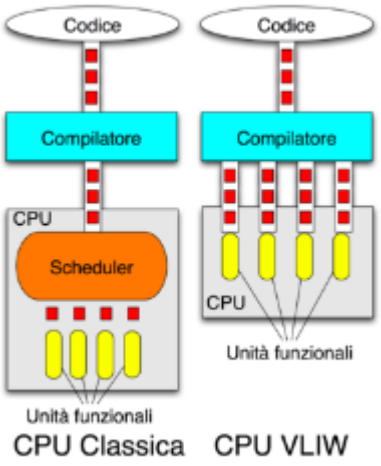
RAM, CAM and TCAM

We have to rely on the match-action abstraction: this is the most efficient way of processing packets. We want to push in this direction since we have CAM (Content Addressable Memory) devices, the opposite paradigm w.r.t. to classical RAM. RAMs are based on this paradigm: you give me the address you want from the memory and I provide you with the output. RAMs are bad in the networking since most of the time, especially with prefix based objects, we must go through all the elements of the RAM, so the search time grows as the size of the table increases. On the contrary, for CAMs, I am not giving in input the address but I am sending the content in input and asking where the content is. Wildcard bits make the CAM a Ternary CAM (TCAM). Ethernet switches rely on CAM, while routers on TCAM. This is because we don't want LPM in ethernet addresses, but exact matching. TCAMs consume a lot of power: the circuit is performing in parallel a logical end of the content provided and all the memory it has. So the table lookup time is O(1) and in 2 clocks it always provides you the output.



Assuming we also have RAM memory in a device, besides the TCAM. We can decide to put some rules in the RAM and some others in the TCAM.

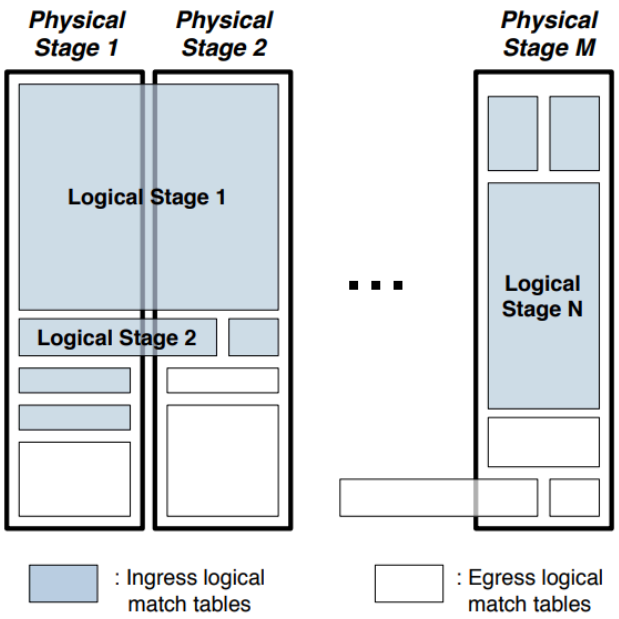
Very Long Instruction Word (VLIW)



In VLIW we can do parallel processing. In general, in regular architecture when we have a code we pass through a compiler that translates that in a sequence of instructions that are passed to a scheduler. The scheduler is the one able to parallelize the task. In the VLIW the programs explicitly specify instructions to execute in parallel, so the complexity of determining what to parallelize is offloaded from the scheduler to the compiler. This type of architecture is suited when we have a very parallelizable task. Modifying a field is parallelizable: modifying an Eth field and an IPv4 field is something we can do in parallel.

Reconfigurable Match Tables (RMT)

While CAM and TCAM are used both in PISA and Open State, RMT are exclusive of PISA

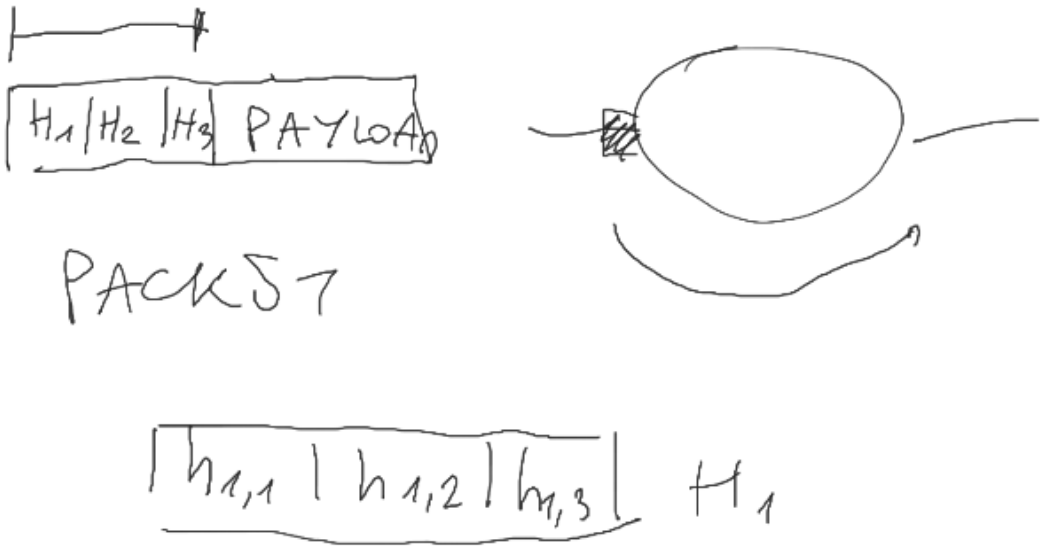


In this way we can have tables with more rules and less fields or vice versa.

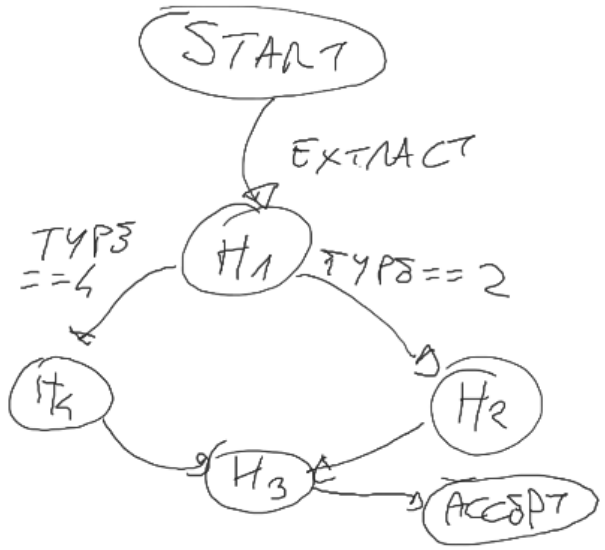
Parser, Stages and Deparser in PISA

Being independent we can process any type of packet. We have three main building blocks:

1. Parser: extract the headers, then going deeper inside each single header distinguishing each field

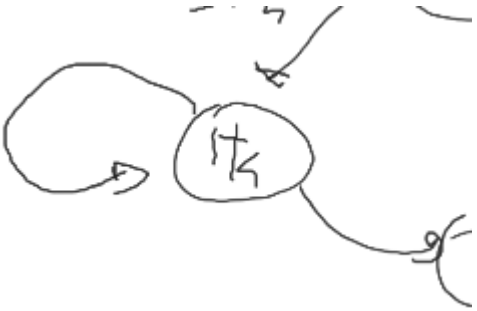


Remember that we are talking about binary strings. By knowing the length of H_1 we can extract from the incoming string of the header which part is related to H_1 and then knowing the length of each field the single field. Inside the header struct you have to define the length of each field, so that the parser can operate. The parser can be programmed: a new protocol with a new header I can reprogram the parser. A parse graph is associated with the parser. For instance:

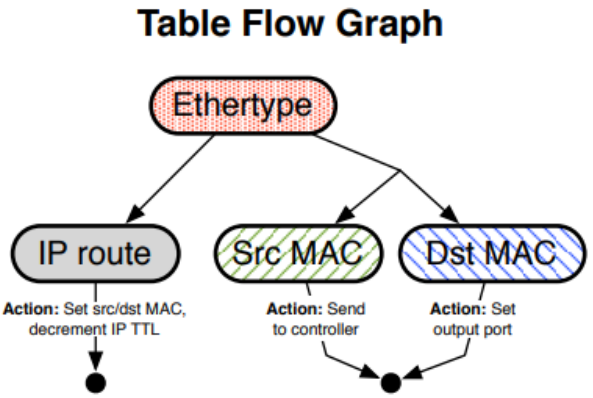


- First we extract H1
  - If H1 is of type 4 we parse H4
  - If H1 is of type 2 we parse H2
- Then we parse H3

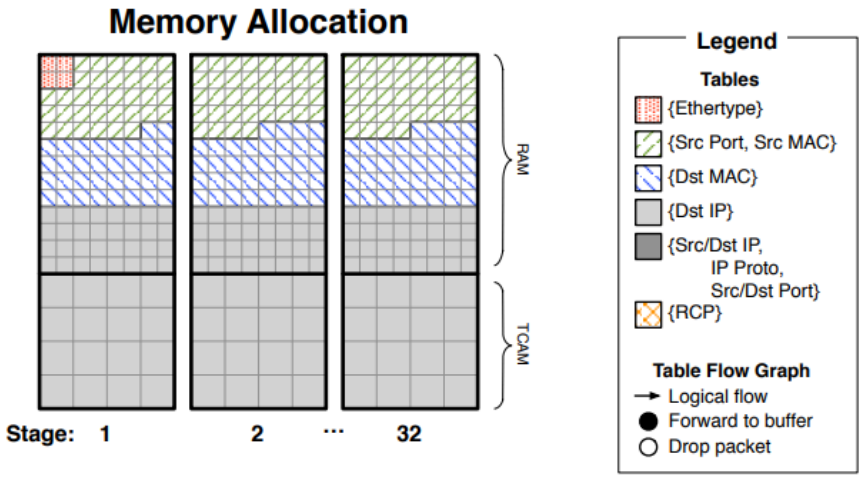
Sometimes there can be a loop, if there are more headers of the same type: for instance in MPLS where we create a stack of labels, so we need to iterate a number of times equal to the number of labels.



2. Pipeline: the set of stages of type match and action. Each stage is a match-action table. I can add new stages and reconfigure their size. We can move from left to right, eventually jump through tables, but never go backwards. For instance, I can change the ip add, or mac address and so on and so forth. Also the number of header can change, maybe because I am encapsulating. Associated to a pipeline, there is a **flow graph**, that tells us which fields are used in the tables and the action applied:



Moreover we have to consider the **memory allocation**: how the logical tables are mapped in the physical memory (RAM and CAM/TCAM) of the device

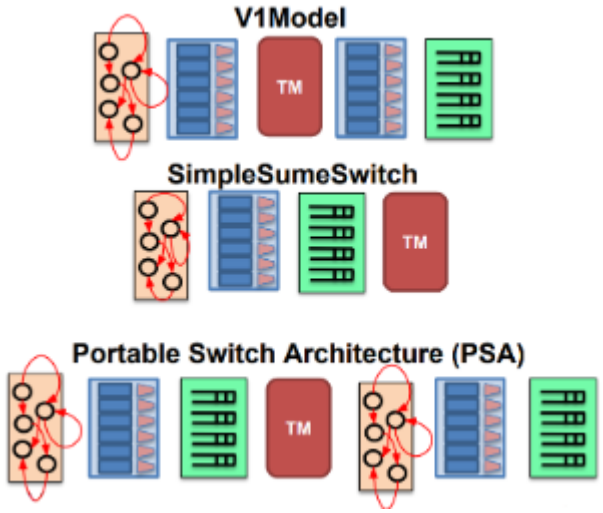


3. Deparsing: we have to serialize the field back, after parallelizing it in the pipeline. We have to specify the order and the deparses will order them in the list.

Metadata is added at the input of the pipeline and passes through the match and action stages. For instance, we can measure the number of seconds that the packet passes inside the pipeline, or the port, or a flag that determines if the packet should be dropped or not, the egress port and so on and so forth. You can also create your own metadata.

Evolution of PISA

The PISA architecture was the first being proposed but there are new ones. PISA, PSA, V1Model are all examples of logical view of the physical target:



For instance, the V1Model architecture is composed of 5 blocks:

1. The parser
2. Ingress pipeline: stage of match and action. For instance IP table lookup is done at the ingress since it's useful to determine the output
3. Traffic Manager: scheduling, queuing and replicating packets
4. Egress pipeline: stage of match and action. All the process before sending it to the next hop. Computing the checksum is done in the egress
5. The deparser: that serializes the headers

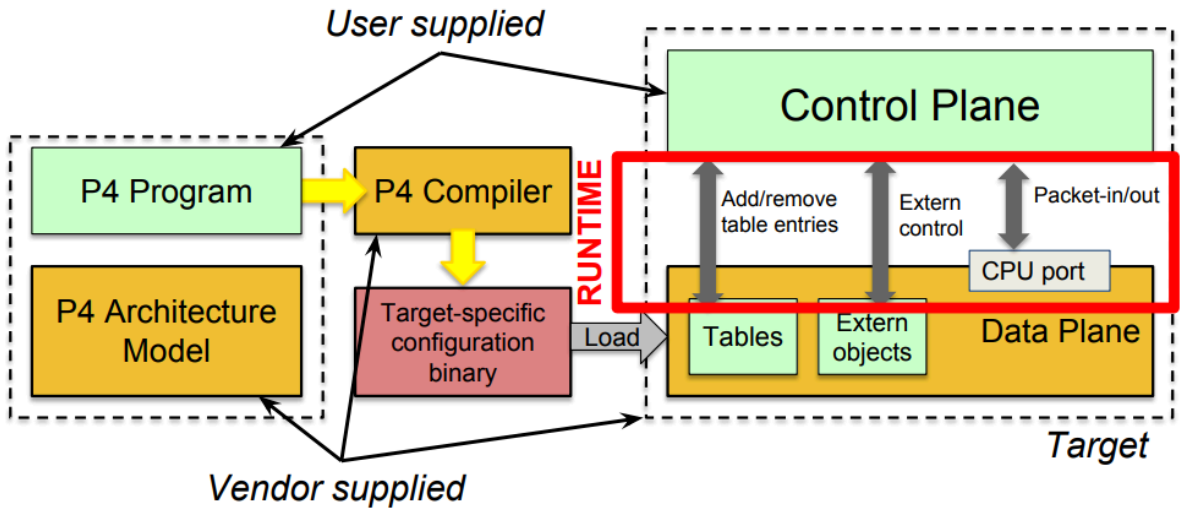
How can we define our own architecture? Writing a piece of code in P4 language.

1. Interface between the blocks of the pipeline
2. Extern functions: library if you want
3. Extensions to the core P4 language types

P4 Compiler

P4 is target-independent, like Python for instance[15]: the language is the same if you use it in a linux or windows machine, regardless of the hardware. It provides a high level abstraction. Once we have a P4 program we have to compile it with a P4 compiler. All the burden of mapping P4 instruction in hardware is in the compiler. The compiler could change based on the HW. The P4 programmer has to know the logical view of the target, called architecture, for instance the PISA. Don't care which hardware is implemented, which is a responsibility of the compiler. Assume a vendor provides me the hardware, the target in this figure:

[15] Actually P4, unlike Python, C etc., does not support loops, pointers or dynamic memory allocation. These functionalities are not useful for a match and action pipeline.



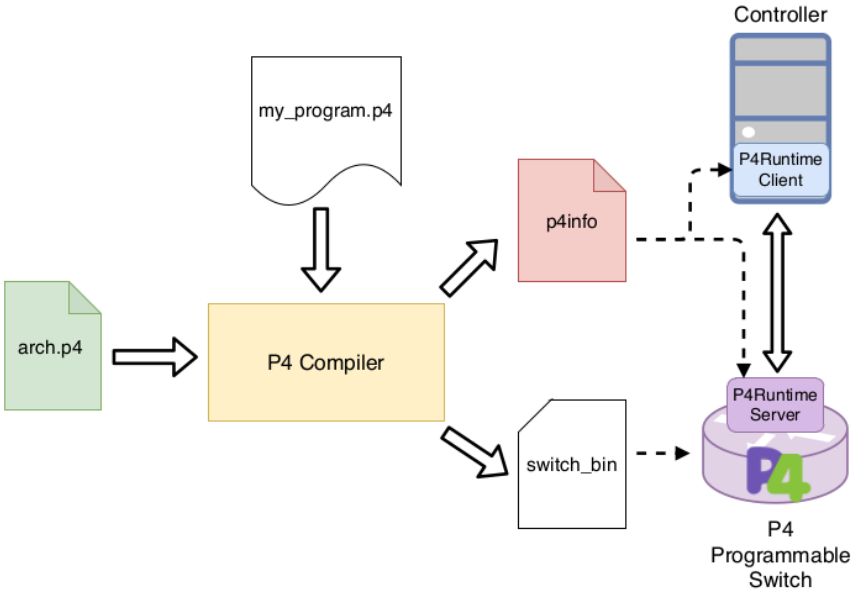
The hardware implements the dataplane. The vendor is going to give me also:

1. P4 compiler, since each target has its own compiler.
2. P4 architecture model

These three elements are vendor supplied. The user writes its own P4 program, for instance the program for a router, the compiler creates a binary file for the target. Once the program is loaded my programmable switch will behave according to my program. I can't change the program in real time, I have to turn it off and restart.

Connecting to an SDN controller with P4 Runtime

We also have to implement the control plane functionalities. In our activities we will manually type the rules to populate the tables, so we are going to be the control plane. However, you can control this device with a SDN controller, using a south bound protocol, called P4 runtime, it's a protocol similar to Open Flow. The rule installation is at runtime, so meanwhile the switch is operating, while the loading of the program is offline. This is the whole story:

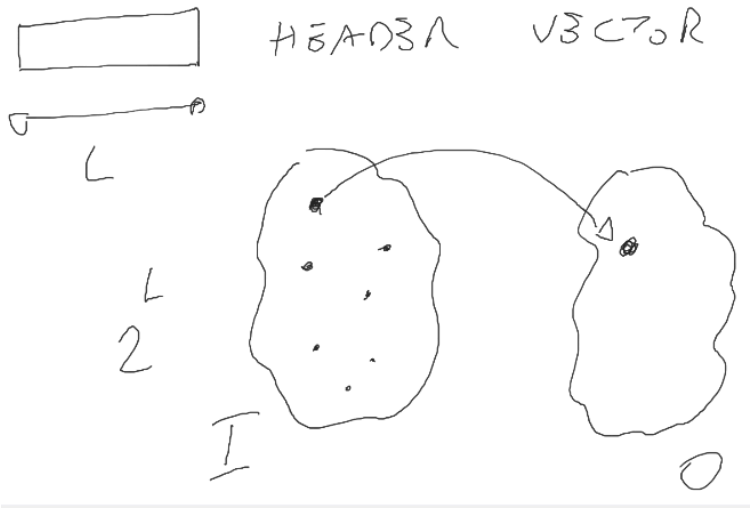


Starting from p4 scripts, the P4 compiler outputs:

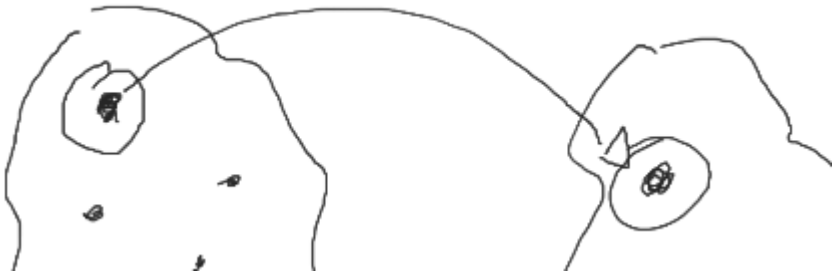
- p4info: a sort of contract that specifies the logical structure of the tables and actions supported by the switch. The controller is interested in this file, since it can understand how the switch should be instructed with rules to add in the tables[16]
- The binary that makes the switch behave as we have programmed

Open State

Add statefulness in the OF switches. Note that this will not remove the other constraints of OF, such as the impossibility of defining new actions and new protocol. This is just adding statefulness. What does it mean to be stateful? Think of this header vector. It has a specific length in bits, let's say L bits. There are at most 2^L of possible instances of this vector. This is the input space.



The match-action paradigm is just a function that maps an input point with an output. The output is just the action you apply. In this way you have stateless processing: there is no way that you apply an action that is different from the one decided.



The match-action, with TCAM as an engine, is:

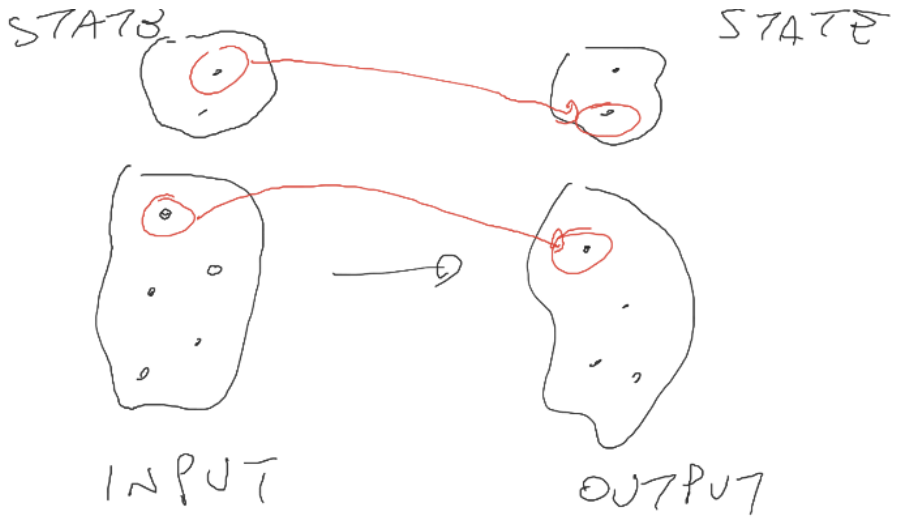
[16] In OF we don't need the SDN to receive an info file similar to this, since the OF pipeline is fixed (in a certain version of OF) and is not programmable, so the header knows is advance how to instruct the switch

$$T : I \longrightarrow O$$

- $I = \{i_1, \ldots, i_M\}$ is a finite set of **Input Symbols**
  - all the possible matches supported by an OpenFlow specification
- $O = \{o_1, \ldots, o_K\}$ is a finite set of **Output Symbols**
  - all the possible actions supported by an OpenFlow switch

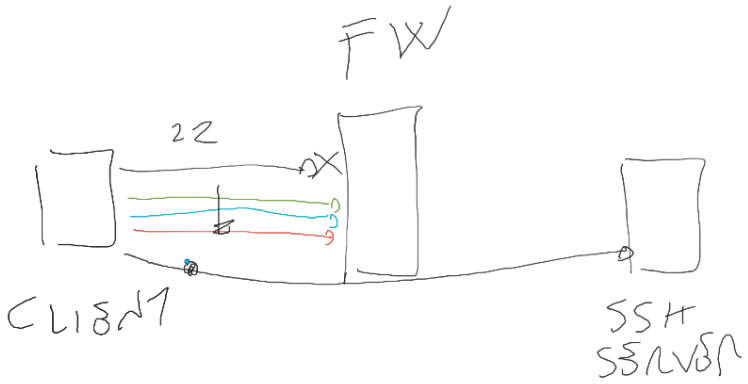
To have something stateful, we need to have for the same input, under different conditions, a different action. Based on the combination of input and state we return an output and a state transition, so I can change state.

$$T : S \times I \longrightarrow S \times O$$



Example: Port Knocking Procedure

You have firewalls to connect to specific services. If you knock to the door with a secret sequence of knocking you will get in.



If the client directly tries to connect to port 22 it gets blocked. But if it completes a sequence, maybe three attempts to connect to three different sequences, the firewall will let it in. As we

move to the different requests the state of the switch is changing. The FW behaves in different ways depending on the state we are in.