LINGI2142: Computer Networks - Group 5
Transit Network Simulation

LANGLOIS QUENTIN
19281700
quentin.langlois@student.uclouvain.be

DARDENNE FLORENT
02601700
florent.dardenne@student.uclouvain.be

IAVARONE SIMON
29911700
simon.iavarone@student.uclouvain.be

***Abstract*—This report describes our analysis of the European OVH network topology. It details different aspects such as OSPF, BGP, security and anycast. In the end, we present our simulation techniques, tests and results.**

I. INTRODUCTION

As part of the computer networks course [2], we were asked to choose a part of the OVH network, describe and emulate it with the IPMininet [1] library.

The objective of this project is therefore to discover how BGP works in a network by configuring it in the best way, to discover and implement BGP communities, to support the anycast protocol and to improve the network security. In order to get the most out of it, we will no longer consider OVH as a service provider but as a transit network.

This report will first describe the area of the OVH network we have chosen as well as our strategy in this network for the implementation of the BGP and OSPF protocols.

Then we will present our strategy and implementation of 3 additional aspects such as the security, the BGP communities and the anycast protocol.

Finally, we will show our simulation methodology, tests and results.

II. DESCRIPTION OF THE TOPOLOGY

A. Zone description

We chose the European zone of the OVH [3] topology. This choice was made for 2 reasons :

- There are many redundancies in this part of the network.
- Many external ASes are present at many places which allows us to explore in more details the eBGP mechanism.

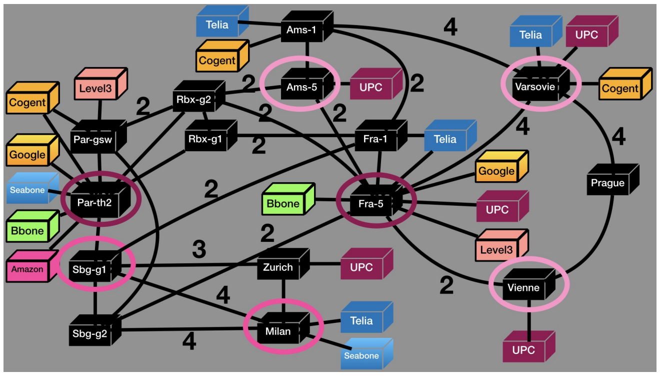


Fig. 1. Topology representation

Legend :

- Boxes represent routers and their color represents their AS (one color per AS).
- Links represent physical links with their associated IGP cost[1] (1 by default).
- Circles represent a group of Route-Reflectors.

B. OSPF and addressing plan

In order to save a maximum of IP addresses, we made an hierarchical addressing plan.

For this purpose, we first assigned a specific subnet for each AS (in /24 for IPv4 and /48 for IPv6).

Next, we divided the OVH network by region (cities) and gave them a more specific subnet (in /28 for IPv4 and /56 for IPv6).

The last step was to determine the loopback-address of routers and the address of their interfaces for link addresses. As described in the section VI, we made an incremental address generator for loopback-address and used the same technique for link-addressing. This means we took the first available address (in /32 or /128 for loopback) and assigned it to the router. For the links, we chose the subnet of one of the two routers and generated 2 addresses (in /31 or /127) in order to put the 2 routers in the same subnet.

C. iBGP configuration

In order to have a minimal number of iBGP connections, we made a multi-level Route-Reflector (RR) architecture. In this type of architecture, low-level RR are clients of high-level RR.

Furthermore, as shown in Fig. 2, the iBGP connections follow, in most cases, physical links connections.

In order to create redundancy, all routers are connected to at least two Route-Reflectors.

IGP costs allow to avoid diflection also in case of crashes.

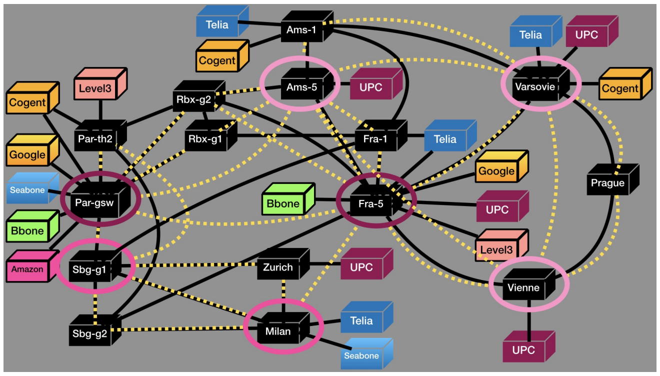


Fig. 2. iBGP connections

Description of the RR-levels :

- **First level** : main level which will dispatch routes to second-level-RR.

[1] See section II-C for more details

We chose *fra-5* and *par-gsw* because they are connected to many other ASes and therefore have a central position and receive many external routes.

- **Second level** : we splitted the second-level RR in 2 sub-groups:
  - At the left of the topology, *milan* and *sbg-g1*, will dispatch routes to this part.
  - *ams-5*, *vienne* and *varsovie* which will dispatch routes to the left part of the topology.

  A fullmesh is done for each subgroup.
- **Third Level** : This level of RR is specific to the Anycast part of the simulation. We will therefore consider it later in the anycast simulation section.

D. eBGP configuration

In order to have a more realistic topology, each external AS router for each eBGP connection with OVH are different. Actually, one unique router connected to Milan, Warsow, Amsterdam, Paris and Frankfurt for the UPC AS seemed not realistic at all.

IPMininet creates several import/export route-maps depending on the type of eBGP link. Unfortunately these different route-maps interfere with our BGP communities policy. In order to implement our communities correctly, we therefore decided to consider all external AS as customers (technically, we set the eBGP session to None in IPMininet). However, we are aware that a transit network does not only include customers but also suppliers and peers.

E. Topology summary

As described in section VI, we also added methods to print statistics on the topology. Here is a screenshot of the result.

III. Security

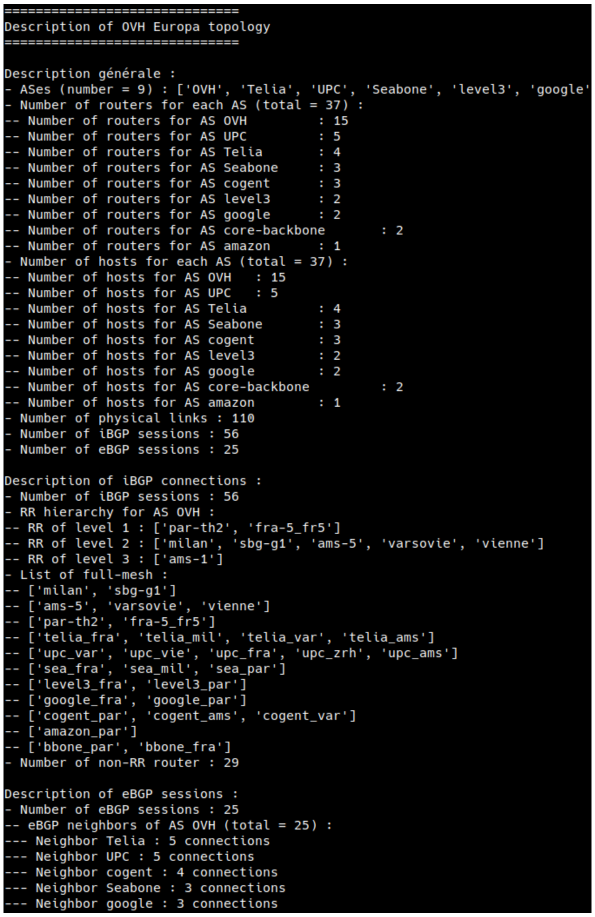
A. Security techniques

In a first part we will discuss all the security aspects we thought of implementing. All these aspects are covered in lots of detail in Ref. 16. We will shortly describe all the techniques of the article. We have not tried to implement them all, but we believe it might be interesting to present them here.

The first security technique we will cover is the GTSM (Generalized ttl security mechanism). To implement this technique we have to ensure three things. If one of our routers that has an eBGP connection receives a packet with ttl < 254 then it must drop it. All the packets that are sent by our routers on the bgp port (port 179) should have their ttl set to 255. Furthermore, we must force our eBGP neighbors to follow the aforementioned rule.

Another security guideline is to use MD5 to ensure that only peers with whom we have an agreement will transmit data to our network. To enforce this policy we have to establish a secret and we have to communicate it to the peer. We tried to implement both these techniques but were not successful. We will describe what was tried later.

Another technique that is recommended is prefix filtering. The article mentions two sources that contain all the special prefixes and instructions on how to route them in ipv4 and ipv6. Furthermore, we must avoid forwarding packets to addresses that were not allocated by the IANA. This idea does not make sense in ipv4 anymore but can be useful in ipv6. It is important to note that this aspect must be updated when the IANA allocates new prefixes.



```
==============================
Description of OVH Europa topology
==============================

Description générale :
- ASes (number = 9) : ['OVH', 'Telia', 'UPC', 'Seabone', 'level3', 'google'
- Number of routers for each AS (total = 37) :
-- Number of routers for AS OVH         : 15
-- Number of routers for AS UPC         : 5
-- Number of routers for AS Telia       : 4
-- Number of routers for AS Seabone     : 3
-- Number of routers for AS cogent      : 3
-- Number of routers for AS level3      : 2
-- Number of routers for AS google      : 2
-- Number of routers for AS core-backbone       : 2
-- Number of routers for AS amazon      : 1
- Number of hosts for each AS (total = 37) :
-- Number of hosts for AS OVH   : 15
-- Number of hosts for AS UPC   : 5
-- Number of hosts for AS Telia         : 4
-- Number of hosts for AS Seabone       : 3
-- Number of hosts for AS cogent        : 3
-- Number of hosts for AS level3        : 2
-- Number of hosts for AS google        : 2
-- Number of hosts for AS core-backbone         : 2
-- Number of hosts for AS amazon        : 1
- Number of physical links : 110
- Number of iBGP sessions : 56
- Number of eBGP sessions : 25

Description of iBGP connections :
- Number of iBGP sessions : 56
- RR hierarchy for AS OVH :
-- RR of level 1 : ['par-th2', 'fra-5_fr5']
-- RR of level 2 : ['milan', 'sbg-g1', 'ams-5', 'varsovie', 'vienne']
-- RR of level 3 : ['ams-1']
- List of full-mesh :
-- ['milan', 'sbg-g1']
-- ['ams-5', 'varsovie', 'vienne']
-- ['par-th2', 'fra-5_fr5']
-- ['telia_fra', 'telia_mil', 'telia_var', 'telia_ams']
-- ['upc_var', 'upc_vie', 'upc_fra', 'upc_zrh', 'upc_ams']
-- ['sea_fra', 'sea_mil', 'sea_par']
-- ['level3_fra', 'level3_par']
-- ['google_fra', 'google_par']
-- ['cogent_par', 'cogent_ams', 'cogent_var']
-- ['amazon_par']
-- ['bbone_par', 'bbone_fra']
- Number of non-RR router : 29

Description of eBGP sessions :
- Number of eBGP sessions : 25
-- eBGP neighbors of AS OVH (total = 25) :
--- Neighbor Telia : 5 connections
--- Neighbor UPC : 5 connections
--- Neighbor cogent : 4 connections
--- Neighbor Seabone : 3 connections
--- Neighbor google : 3 connections
```

Fig. 3. Information about the topology

Maximum prefixes on a peering. The idea of this technique is to set a threshold on the number of prefixes we receive from a peer. By doing this, we ensure that nobody will send us the whole internet's routing table as it would be quite annoying.

Another way to secure your network from the outer world is by filtering the AS-Path. One of these rules is to ensure that the first AS number in the AS-Path of the packet is the AS number of the AS the packet comes from. We should then ensure that the packets sent by our AS contain an AS-Path that is not empty.

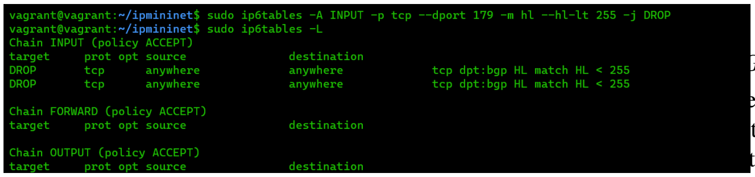
The last discussed technique is the filtering of the bgp nexthop. The way to implement it is to check that the bgp nexthop of a router is his bgp neighbor.

B. Security simulation

This section will just describe the security aspect of the simulation. Other aspects will be discussed in Sec. VI.

We first tried to implement the GTSM. To implement this technique there are few changes to make to two files : `ipmininet/src/router/config/bgp.py` and `ipmininet/src/router/config/templates/bgpd.mako`. We inserted the `neighbor $n.peer ttl-security hops 1` command. We set the hops parameter to 1 so as to drop packets with a ttl < 254.

The second thing we tried is using IPTables in python. At first we tried to insert the rule manually. This worked fine.



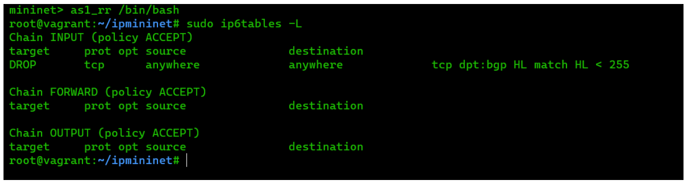
```
vagrant@vagrant:~/ipmininet$ sudo ip6tables -A INPUT -p tcp --dport 179 -m hl --hl-lt 255 -j DROP
vagrant@vagrant:~/ipmininet$ sudo ip6tables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       tcp      anywhere             anywhere             tcp dpt:bgp HL match HL < 255
DROP       tcp      anywhere             anywhere             tcp dpt:bgp HL match HL < 255

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

Fig. 4.

But then we had problems figuring out how to insert these kind of rules in python. When we simply created the rule and then proceeded to add the IP(6)Tables daemon the code would not work. We have just upgraded the kernel on the machine and it seems to work right now :



```
mininet> as1_rr /bin/bash
root@vagrant:~/ipmininet# sudo ip6tables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
DROP       tcp      anywhere             anywhere             tcp dpt:bgp HL match HL < 255

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@vagrant:~/ipmininet# |
```

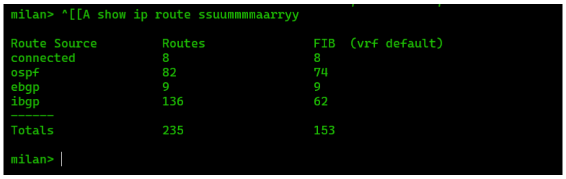
Fig. 5.

Now after having upgraded the system, IPTables seem to be working fine. From now on it should be fairly easy to set more IPTables rules. You can find the topology that was used in the file `project_tests/security_topo.py`.

After the reviews we tried more things with IPTables and we were able to add a better firewall to the routers of `project_tests/security_topo.py`. The ipv4 rule work fine but we had more trouble with the ipv6 rules. Indeed we didn't find a way to enable tcp traffic based on the sources of the routers in our network. Moreover when trying to add these rules to json_topo.py we noticed the ping would drop dramatically which is a big problem.

We also implemented the md5 security mechanism. To do this we added the following lines in `ipmininet/src/router/config/template/ospfd.mako` `neighbor ip ospf message-digest-key 1 md5 PASSWRDSIMQUEFLO` and `ip ospf authentication message-digest`.

Furthermore we added a maximum prefix rule. We set it to 500. Indeed when looking at our routers we notice that most of them hold around 400 routes. We thought it would be a good idea to set a maximum prefix limit higher than the number of routes each router holds. The command we added is the following `neighbor $n.peer maximum-prefix 500 60` [16]. We decided to set a warning when 60% of this amount of prefix is reached. When the number of prefixes reaches 500, the peering is dropped.



```
milan> ^[[A show ip route ssuummmmaarryy

Route Source         Routes               FIB  (vrf default)
connected            8                    8
ospf                 82                   74
ebgp                 9                    9
ibgp                 136                  62
------
Totals               235                  153

milan> |
```

Fig. 6.

IV. BGP COMMUNITIES

GP Communities techniques

e BGP protocol defines attributes to influence the routing of ts in the network. Among these attributes we find the local- the AS-Path, the MED, etc. In this section we are interested in the communities attributes. BGP communities are attributes allowing to group several routes in a single entity where the same routing decisions can be applied [4].

We have defined in the transit network several communities with each time a different strategy. To describe these different types of communities we will use the classification defined by Quoitin and Bonaventure [5]:

- **Inbound communities**:
  - Set local-pref: We have implemented communities allowing customers to modify the local-pref of their routes in the transit network. These allow them to define priority and backup routes. The MED also allows them to do this, but since the local-pref is located before the AS-Path in the BGP decision, it is important in certain situations.
  - Route-tagging: We decided not to include route-tagging communities because we do not consider them useful in the network. Furthermore we avoid to create too many communities as much as possible because they increase the size of our BGP messages and can induce unnecessary updates [6].
- **Outbound communities**: These communities allow to filter BGP announcements and allow modifying other attributes in the route when redistributing routes in order to do traffic engineering.
  - Announcement: We provide customers with communities to filter their route announcements from the transit network to other customers in certain cities.
  - AS Prepending: These BGP communities allow customers to request the transit network to prepend the transit network AS in the AS-path x times when a router announces it at the eBGP level to a customer. This community allows client AS to indicate their preferences to other AS in relation to the different transit network of which they have a BGP session. e.g. if Google prefers to be contacted via AS3-transit rather than AS4-transit then it will request an AS-path prepending to AS4-transit to have a higher AS-Path than AS3-transit.

    Not having been able to do it in time we could not implement it in our topology. However, it would be an interesting community to know how to implement it even if in our topology its usefulness would be quite limited.
- **Blackhole**: Blackhole communities allow AS to ask their ISPs to block certain packets. It is a community of interest for security and especially for dealing with DoS attacks. We have not implemented it in our transit network.

B. BGP Communities in our topology

In the table below we can see the communities we have created in our transit network and their action.

C. BGP Communities in our simulation

In this part we discuss how the implementation of this part was done in our code.

In order to create route-maps to implement the set local-pref we had to modify the IPMininet library. Indeed, this one unfortunately

Community	action
16276:120	Set local-pref to 120
16276:90	Set local-pref to 90
16276:80	Set local-pref to 80
16276:401	Do not announce to customer in Paris
16276:402	Do not announce to customer in Amsterdam
16276:403	Do not announce to customer in Vienne
16276:404	Do not announce to customer in Frankfurt
16276:405	Do not announce to customer in Varsovie
16276:406	Do not announce to customer in Roubaix
16276:407	Do not announce to customer in Prague
16276:408	Do not announce to customer in Milan
16276:409	Do not announce to customer in Strasbourg
16276:410	Do not announce to customer in Zurich
16276:20x	Prepend x times the AS in the AS-Path when announcing to customer (not implemented)

TABLE I
BGP COMMUNITIES IMPLEMENTATION

didn't give us enough flexibility to do it. [7] By adding some parameter to the function, we managed to implement the local-pref and provide it to the transit network customer. We will see in the sub-section "Test" and "Result" the customers using this functionality.

For the outboud announcement functionnality it was easier, the technique here was to use the deny function of the IPMininet library with the name of a route-maps already created by IPMininet and an order smaller than this one. We will also see in the sub-section "Test" and "Result" the customers using this functionality.

Finally the implementation of these features required a lot of route-maps debugging in the router configuration. This allowed us to understand in depth how route maps work.

V. ANYCAST

A. Anycast techniques

Anycast is the ability to have multiple **servers** with the same IP address in a network and join the nearest one from its position.

There are many possible utilization for this feature such as DNS servers, data-centers, ...

In order to configure anycast, there are 3 main methods :

- **DNS zones** : grouping some servers under a unique domain name.
- **OSPF** : advertise multiple times the same IP address over OSPF.
- **BGP** : advertise multiple times the same IP address over BGP.

B. Anycast in our topology

For this project, we chose the third solution : anycast over BGP. Here are some reasons :

- The implementation is pretty easy.
- The BGP decision process allows to advertise the route to the nearest server.
- The convergence in case of failure is relatively fast.
- It allows to use BGP properties such as communities to make traffic engineering.
- It allows to better secure the server's behaviour compared to the OSPF technique.

C. Anycast in our simulation

More details about the simulation are given in its appropriate section. We will just show here the anycast aspect of it.

As we chose the BGP method to implement anycast, the simulation was easy to implement because BGP is well supported by the ipmininet library.

However, the BGP daemon is not supported by the **Host** class and we therefore chose to use the **Router** class to represent anycast servers.

To specify their common anycast IP address, we just specify it as their loopback-address.

In order to advertise this anycast address to other AS, we had to connect these anycast servers to a Route-Reflector. To simplify the simulation, we added a new level of Route-Reflector responsible to advertise anycast server's routes[2].

VI. SIMULATION

A. Build, run and configure

In order to run our topology, you just have to run "sudo python3 json_topo.py" in the vagrant terminal. This command will build the topology, show general information (see Fig 1.) and run the Mininet Command Line.

The README.md file on our github (also present the .zip file of the code) describes in more detail the different topologies configuration we created (with / without communities / security) and how to run a specific topology.

B. JSON representation

For the project, we chose to create an automatized construction of the topology based on a structured representation of it. One of the easiest and most common way to structure data is the **JSON** format.

Here is a brief overview of our formatted OVH topology and its different configuration fields :

```
{
# subnets to infer loopback and links addresses
  "subnets" : {
    "net_ovh" : { # Main OVH subnet in /24 and / 48
      "ipv6" : "beaf:cafe:babe:/48",
      "ipv4" : "204.32.46./24"
    },
    "net_mil" : { # Specific subnet in /28 and /56.
      # Adresses are structured with the
      # 'net_ovh' field. It will be replaced
      # by the corresponding address of this subnet
      "ipv6" : "{net_ovh}0500::/56",
      "ipv4" : "{net_ovh}112/28",
      "nodes" : ["milan"]
    },
    # ... other subnets for the cities
    "net_upc" : { # Example of subnet for another AS.
      "ipv6" : "beaf:cafe:bab1::/48",
      "ipv4" : "204.32.48.0/24",
      "nodes" : ["upc_var", "upc_vie", ...]
    },
    # ... other subnets for ASes
  },
  "AS" : {     # ASes in the topology
    "OVH" : { # Our main AS.
```

[2] if their neighbor router was already a RR, we just added the anycast server to its clients.

```
      # Routers in the AS with their configuration.
      "routers" : {
        #As milan has a 'clients' field,
        #it is considered as a RR.
        "milan" : { # Name of the router : milan
          #Peers, clients and level for the RR
          #Note: if level is not specified,default=1
          #peers should only be specified for level>1
          "clients" : ["zurich", "sbg-g2"],
          "peers" : ["sbg-g1"],
          "niveau" : 2
        },
         #Simple router without config:
        "zurich": {},
        # ... other routers
      },
      # Default config for all routers of
      # this AS
      "rconfig" : {
        "daemons" : {   # Default daemons
          # Daemons without config:
          "ospf": {}, "ospf6":{},
          # Config for the BGP daemon:
          "bgp": {
            "communities": {
              # Communities to add
              "set_local_pref": {
              # Conditionned Local-pref
                "16276:120": 120
              }
            }
          }
        }
      },
      "anycast" : [ # Anycast servers in this AS
        {
          "addresses" : { # Anycast addresses
            "ipv4": "10.10.10.10/32",
            "ipv6": "10::10/128"
          },
          # Nodes to which connect an
          # anycast server
          "nodes" : ["milan", "varsovie", "ams-1"]
        }
      ]
      # the field 'hosts' can also be added
      # to specify hosts in the AS
      # In our case, we did not have any hosts
      # so this field is not present.
      # However, for our tests, we added many
      # fictitious hosts automatically
    },
    "Telia" : {
      "routers" : { ... },
      "rconfig" : {
        # Daemons are specified
        # as a list (no config)
        "daemons" : ["ospf", "ospf6", "bgp"]
      },
      # Automatically create a fullmesh of links
      # between all routers of the AS
      "linked" : true
      # Automatically create a iBGP fullmesh
      "bgp_fullmesh" : true
    },
    # ... other AS configuration
  },
  "links" : {    # Physical links
    "milan"     : [   # Node and its neighbors
      "zurich",    # Link without config
      # Link with specified IGP metric
      ["sbg-g1", {"igp_metric": 4}],
      ["sbg-g2", {"igp_metric": 4}]
    ],
    # ... Other links
    # Example of a link between routers from
    # 2 different AS.
    # The code will automatically create an
    # eBGP connection between them
    "amazon_par"   : ["par-th2"]
  }
}
```

C. *Topology generation*

In this section, we will briefly describe the main aspects of the topology generation that are useful to understand our tests.

A more detailed description of each part of the construction (AS, routers, links and subnets) can be found on our code.

1) IP address generator: In order to easily generate IP addresses, we created an IP address generator based on a specific subnet. This generator is used in 2 features : loopback-address and link-address generation.

For the loopback-address part, the target is to create a /32 (or /128) address based on the router's subnet. As shown in the section VI-B, we can easily get the subnet of a specific router. After the identification, we compute the number of already created addresses for this subnet, add 1 to the last byte and obtain the new address.

For the links-addresses, the difficulty was to create a /31 (or /127) address so we had to determine the right subnet (and not add 1 like for loopback addresses).
To solve this, we computed the next multiple of 2 (2^{32-31}) for the last byte and it gives us the first address of the subnet. We then compute the second address by adding 1 to the last byte.
Note : to select the subnet for the link, we took the subnet of the link's "main"[3] node.

2) Hosts generator: In order to make tests on our topology (ping, traceroute) we also provided a tool to add fictitious hosts to AS[4].

As a basic usage, we simply added 1 host for each router of each AS. With this technique, we were able to test connections between all parts of our topology.

D. *Tests*

For our tests, we used the **Mininet command-line** to execute commands on our routers. Here is a brief overview of commands used to test different aspects :

- **OSPF and BGP** : as shown in the previous section, we added many fictitious hosts in our topology. This allowed us

[3] The main note is the key in our json-format of links. The values are its neighbors.

[4] A more detailed description of the different features of this tool can be found in the documentation of our *JSONTopo class*

to test the connectivity between all pairs of routers with the **ping6all** command. We also verified through the **traceroute** command that the path taken by each host to other hosts was in accordance with our expectations.

- **Security** : To test our security topology we used the pingall, ping6all, iptables and ip6tables commands
- **BGP communities** : In order to test our communities, we have had external AS attach communities to their roads.

  - The Set Local-pref test :
    Level3 router in Frankfurt attaches community 16276:120 to its routes towards our network. This community allows it to define the local-pref of its routes at 120 : this will be its main link.
    Then it is the level3 router in Frankfurt which attaches the community 16276:80 to set the local-pref of its routes to 80 : this will be the backup link of the AS level3.

    In order to test this, we first look if the route-maps are correctly defined in the routers. Then we perform traceroutes from routers to level3 to make sure that only the link to Frankfurt is used. Finally we cut the link between Level3 Frankfurt and our network to see if the backup link Level3 Paris is used correctly. We will see the results of this test in the next section.

  - The Outbound Announcement test:
    UPC does not wish to announce its routes to the transit networks customers located in Paris. For this purpose it will attach the 16276:401 community to its routes.

    In order to test this, we will make a ping between Amazon AS which is only attached to Paris in our network and UPC. If our community works well, Amazon being isolated in Paris, it will not be able to contact UPC and vice versa. UPC will know the route to Amazon but Amazon don't know the route to UPC so it can't answer to it. To be sure that it is not a firewall blocking the pings, we will check in Amazon's routing table that the routes to UPC are not there.
- **Anycast** : in order to test Anycast, we added 3 anycast-servers in different places (cf section VI-B, anycast entry). Next, we used the traceroute command from different hosts (inside / outside our AS) to the anycast address and see which server has been reached.

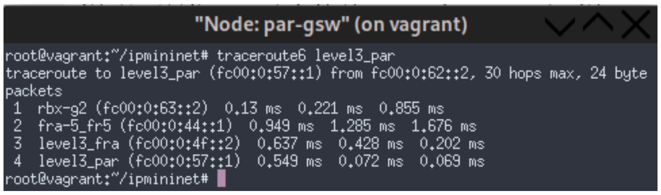
E. Results

a) OSPF and BGP results: with the **ping6all** command, all hosts tried to contact each other hosts so that we could check if all parts of our network is reachable by all other parts. The result of the command showed a 0% packet drop : it means that all routes are well distributed and reachable in the whole topology.

We then traced the paths taken by some hosts to other hosts and then checked with **traceroute** if they were in agreement with our calculations.

b) Security: When adding the security rules, pingall and ping6all still work fine. We didn't implement much features so far in the matter of security. However, now that the major issue with IPTables seems to be solved, it should be fairly easy to implement more security techniques.

c) BGP communities: The route maps being well checked on the OVH side and on the Level3 side, we can proceed to the verification of the **local-pref set**. We connect to OVH Paris-Gsw and then do a traceroute to level3 Paris.

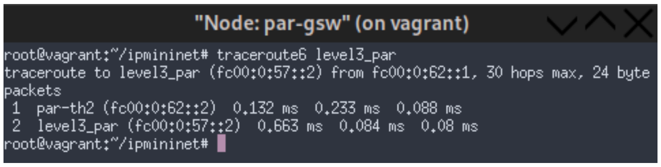


```
root@vagrant:~/ipmininet# traceroute6 level3_par
traceroute to level3_par (fc00:0:57::1) from fc00:0:62::2, 30 hops max, 24 byte
packets
 1  rbx-g2 (fc00:0:63::2)  0.13 ms  0.221 ms  0.855 ms
 2  fra-5_fr5 (fc00:0:44::1)  0.949 ms  1.285 ms  1.676 ms
 3  level3_fra (fc00:0:4f::2)  0.637 ms  0.428 ms  0.202 ms
 4  level3_par (fc00:0:57::1)  0.549 ms  0.072 ms  0.069 ms
root@vagrant:~/ipmininet#
```

Fig. 7. traceroute6 from par-gsw to level3_par with set-local communities

We can see in the screenshot that Paris-gsw join level3 Paris by taking the route with the local-pref of 120 (level3 Frankfurt) and not by taking the one with the local of 80.

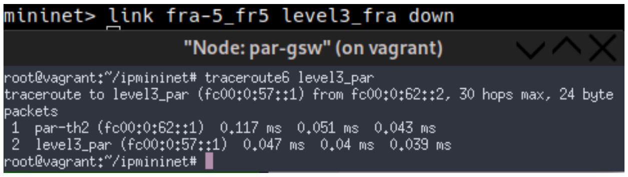
We also tested a traceroute before putting the local-pref communities and here are the results:



```
root@vagrant:~/ipmininet# traceroute6 level3_par
traceroute to level3_par (fc00:0:57::2) from fc00:0:62::1, 30 hops max, 24 byte
packets
 1  par-th2 (fc00:0:62::2)  0.132 ms  0.233 ms  0.088 ms
 2  level3_par (fc00:0:57::2)  0.663 ms  0.084 ms  0.08 ms
root@vagrant:~/ipmininet#
```

Fig. 8. traceroute6 from par-gsw to level3_par without set-local communities

Now let's simulate a failure between Frankfurt and Level3 by cutting the link via IPMininet and let's retrace the route to Level3 Paris from Par-Gsw.



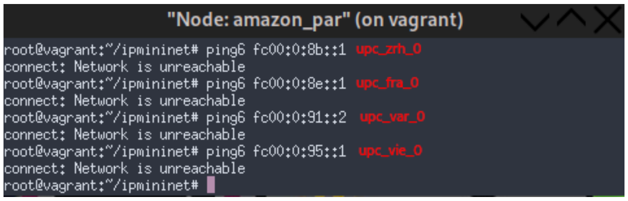
```
mininet> link fra-5_fr5 level3_fra down
```

```
root@vagrant:~/ipmininet# traceroute6 level3_par
traceroute to level3_par (fc00:0:57::1) from fc00:0:62::2, 30 hops max, 24 byte
packets
 1  par-th2 (fc00:0:62::1)  0.117 ms  0.051 ms  0.043 ms
 2  level3_par (fc00:0:57::1)  0.047 ms  0.04 ms  0.039 ms
root@vagrant:~/ipmininet#
```

Fig. 9. traceroute6 from par-gsw to level3_par with set-local communities and link failure between Frankfurt and Level3

In case of link failure, we see that the backup route is taken.

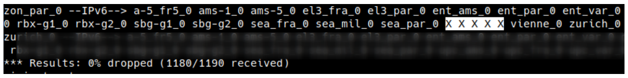
Now we will proceed to the verification of the **Outbound Announcement community**. UPC sending community 16276:401, Amazon (Paris) must not contain any route to UPC hosts:



```
root@vagrant:~/ipmininet# ping6 fc00:0:8b::1 upc_zrh_0
connect: Network is unreachable
root@vagrant:~/ipmininet# ping6 fc00:0:8e::1 upc_fra_0
connect: Network is unreachable
root@vagrant:~/ipmininet# ping6 fc00:0:91::2  upc_var_0
connect: Network is unreachable
root@vagrant:~/ipmininet# ping6 fc00:0:95::1  upc_vie_0
connect: Network is unreachable
root@vagrant:~/ipmininet#
```

Fig. 10. ping6 between amazon AS and UPC AS

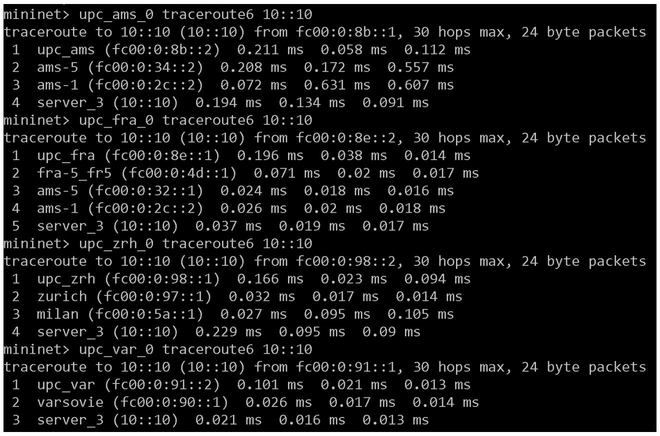
We can also see in IPMininet that the Ping6all contains 10 losses representing the 5 packet from Amazon to the 5 UPC hosts and 5 other losses from the 5 UPC hosts to Amazon.



```
zon_par_0 --IPv6--> a-5_fr5_0 ams-1_0 ams-5_0 el3_fra_0 el3_par_0 ent_ams_0 ent_par_0 ent_var_0
0 rbx-g1_0 rbx-g2_0 sbg-g1_0 sbg-g2_0 sea_fra_0 sea_mil_0 sea_par_0 X X X X X vienne_0 zurich_0
zu
 r
*** Results: 0% dropped (1180/1190 received)
```

Fig. 11. ping6all with 10 loss UPC<->Amazon

d) Anycast: as mentionned in section VI-B, we created 3 anycast servers in Milan, Warsow and Amsterdam. We then used our fictitious hosts in different regions to see which router is reached and here are the results :



```
mininet> upc_ams_0 traceroute6 10::10
traceroute to 10::10 (10::10) from fc00:0:8b::1, 30 hops max, 24 byte packets
 1  upc_ams (fc00:0:8b::2)  0.211 ms  0.058 ms  0.112 ms
 2  ams-5 (fc00:0:34::2)  0.208 ms  0.172 ms  0.557 ms
 3  ams-1 (fc00:0:2c::2)  0.072 ms  0.631 ms  0.607 ms
 4  server_3 (10::10)  0.194 ms  0.134 ms  0.091 ms
mininet> upc_fra_0 traceroute6 10::10
traceroute to 10::10 (10::10) from fc00:0:8e::2, 30 hops max, 24 byte packets
 1  upc_fra (fc00:0:8e::1)  0.196 ms  0.038 ms  0.014 ms
 2  fra-5_fr5 (fc00:0:4d::1)  0.071 ms  0.02 ms  0.017 ms
 3  ams-5 (fc00:0:32::1)  0.024 ms  0.018 ms  0.016 ms
 4  ams-1 (fc00:0:2c::2)  0.026 ms  0.02 ms  0.018 ms
 5  server_3 (10::10)  0.037 ms  0.019 ms  0.017 ms
mininet> upc_zrh_0 traceroute6 10::10
traceroute to 10::10 (10::10) from fc00:0:98::2, 30 hops max, 24 byte packets
 1  upc_zrh (fc00:0:98::1)  0.166 ms  0.023 ms  0.094 ms
 2  zurich (fc00:0:97::1)  0.032 ms  0.017 ms  0.014 ms
 3  milan (fc00:0:5a::1)  0.027 ms  0.095 ms  0.105 ms
 4  server_3 (10::10)  0.229 ms  0.095 ms  0.09 ms
mininet> upc_var_0 traceroute6 10::10
traceroute to 10::10 (10::10) from fc00:0:91::1, 30 hops max, 24 byte packets
 1  upc_var (fc00:0:91::2)  0.101 ms  0.021 ms  0.013 ms
 2  varsovie (fc00:0:90::1)  0.026 ms  0.017 ms  0.014 ms
 3  server_3 (10::10)  0.021 ms  0.016 ms  0.013 ms
```

Fig. 12. Anycast testing with traceroute

As shown in the figure, all hosts contacted the nearest anycast server[5] as expected.

VII. Conclusion

As shown in this report, a good automatized simulation allows to modify and easily test different aspects of a topology configuration.

For the OSPF part, our IP address generator helped to just define subnets instead of defining IP addresses for all routers and links.

For the BGP aspect, the automatization was helpful to avoid duplication of codes (adding daemons and BGP configuration such as communities).

For the security aspect, it was more difficult to add features because it was not well supported by the ipmininet library. However, lot's of research was made to understand good practices.

For the Anycast part, the automatization allowed to automatically add servers at different places without adding links, subnets, etc.

As shown in Sec. VI-D, our test were really conclusive except for the security aspect. However, now that some features are working well, it should not be too difficult to add them in our main topology.

References

[1] IPMininet, https://github.com/cnp3/ipmininet

[2] O. Bonaventure computer course, https://uclouvain.be/cours-2020-LINGI2142.html

[3] OVH Weathermap, http://weathermap.ovh.net/

[4] O. Bonaventure and B. Donnet, "On BGP Communities"

[5] B. Quoitin and O.Bonaventure, "A survey of the utilization of the BGP community attribute"

[6] T. Krenc, R. Beverly, G. Smaragdakis, "Keep your Communities Clean: Exploring the Routing Message Impact of BGP Communities"

[7] IPMininet set_local_pref issue, https://github.com/cnp3/ipmininet/issues/99

[8] Introduction to route maps, https://networklessons.com/cisco/ccnp-route/introduction-to-route-maps

[9] https://docs.umbrella.com/deployment-umbrella/docs/configure-anycast

[10] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.6367&rep=rep1&type=pdf

[11] http://routedo.com/posts/frr-ospf

[12] BGP Anycast configuration guide, https://documentation.nokia.com/html/0_add-h-f/93-0267-HTML/7X50_Advanced_Configuration_Guide/BGP_anycast.pdf

[13] BGP anycast on 2 servers. https://serverius.net/bgp-anycast-dual-datacenter-using-1-ip-multiple-locations/

[14] IPMininet Documentation, https://ipmininet.readthedocs.io/en/latest/daemons.html#named

[15] Good security practices for BGP, https://conf-ng.jres.org/2015/document_revision_2476.html?download

[16] 'Configuring the BGP Maximum-Prefix Feature - Cisco'. https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/25160-bgp-maximum-prefix.html (accessed Dec. 15, 2020).

[5] The server names are not always correct due to the ipmininet reverse DNS but we can identify the right server by the last router reached