LINGI2142 - Computer networks: configuration and management

Nikita Tyunyayev - Soukéina Bojabza - Arnaud Lahousse

I. Introduction

As a part of the course ”LINGI2142 - Computer networks: configuration and management”, we have been asked to create a complete network. We used as a template the OVH network and we implemented it on IPMininet. In order to design the network, we had to implement all the routers and configure the protocols: the border gateway protocol and anycast. We then added some BGP communities and different techniques to ensure the security. Below, you will find the details of the network implementation.

We will first describe the network architecture. Then, we will present our IP addressing plan, in IPv6 and IPv4. We will then show you an example of a router configuration on IPMininet. Next, we will discuss our BGP configuration choices and the justification of the IGP costs we set in the network.

After that, we will present you what we did in order to set BGP communities in our network, to add an implementation of anycast servers and to address the security question.

II. Presentation of the network

The network implemented is the Asia part of OVH, with some of the Europe and North America parts to buckle the loop. An illustration can be found on the figure 1. On this schema, the boxes represent the routers while the clouds are the clients of our network, which are Vodafone, Equinix and NTT. For the sake of realism we have connected each peer at the same router (same location) than in the OVH network.

In practice, the clients are modeled by routers outside the network. So, in the schema, each cloud figure is in fact a router to which a host is connected. The routers of a same client are in the same autonomous system (AS). We did not connect these routers between them because the AS configuration must be managed by the client (we are only in charge of OVH).

In order to have redundancy, we have put two routers by city. We also connected them in a way that we can always find another route to reach the destination. More precisely, there is always a way to reach the Asia part since this is where Equinix and NTT are located and there will therefore be an important traffic towards them. Also, the Asia part is the principal part of our network.

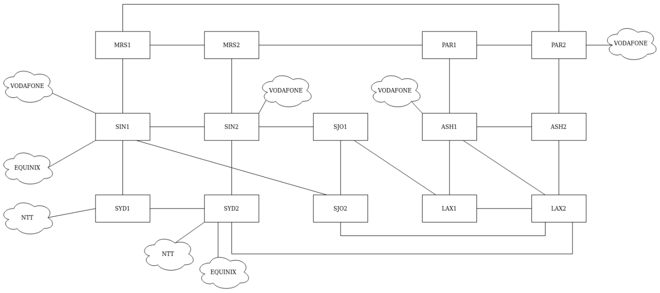


Fig. 1: Implemented network

III. Addressing

A. IPv6

For the IPv6 addressing we have set the first 48 bits common to the whole network : `1627:6000:0000::`. The next 4 bits are used to differentiate the continents: `0` for Europe, `1` for North America and `2` for Asia. The network mask of each router's loop-back address is `/64`.

For the addresses of the links, they have a `/64` mask.

The figure 2 shows an illustration of our IPv6 addressing plan.

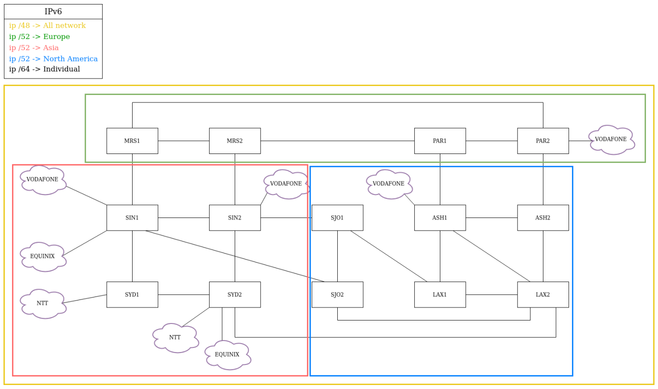


Fig. 2: IPv6 addressing plan

B. IPv4

For the IPv4 addressing, the first 20 bits are the same in the whole network: `162.76.240..`, with a network mask of `/20`. Instead of taking the next 8 bits by continent like we did in IPv6, we decided to assign the next 4 bits to the cities: 1 for Marseille, 2 for Paris, 3 for Singapore, 4 for Sydney, 5 for Los Angeles, 6 for San José and 7 for Ashburn. Finally, each router has a network mask of `/32` for it's loop-back address.

About the addresses of the links, we had the choice between /31 and /30 masks. We have chosen /30 because it's generally better supported by protocols such as OSPF, it is the most used in point to point links and because we have enough IPv4 addresses to do that.

For the peers attached to the network, we assigned the address 160.76./24 and the next 8 bits are used to differentiate the clients: 7, 8 and 9 for Vodafone, Equinix and NTT respectively.

The figure 3 shows an illustration of our IPv4 addressing plan.

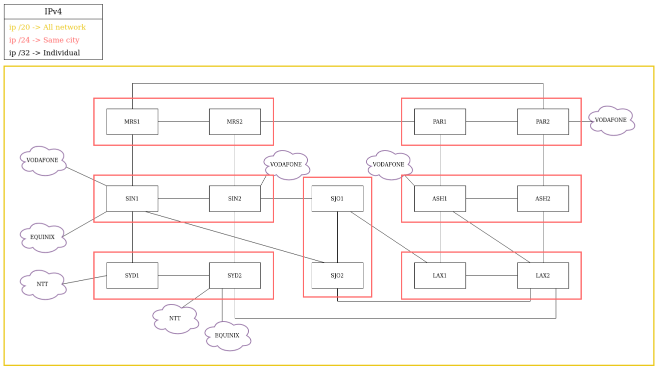
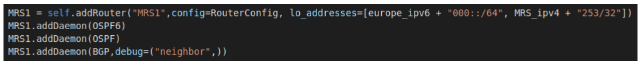


Fig. 3: IPv4 addressing plan

IV. Configuration of the routers and peers

On each router of the network and on the routers representing the peers, we have set the configuration with RouterConfig. It allows to cancel the OSPF and OSPF6 on each router and so to manage how we want to configure the routers properly. After that, we have added daemons OSPF6, OSPF and BGP. We simply use the OSPF6 and OSPF configuration provided by IPMininet because this configuration suits well the the needs of our network. You will find an example of implementation on the router MRS1 on the figure 4 bellow.



```
MRS1 = self.addRouter("MRS1",config=RouterConfig, lo_addresses=[europe_ipv6 + "000::/64", MRS_ipv4 + "253/32"])
MRS1.addDaemon(OSPF6)
MRS1.addDaemon(OSPF)
MRS1.addDaemon(BGP,debug=("neighbor",))
```

Fig. 4: Implementation of the router MRS1

A. *OSPF area*

We thought about implementing OSPF areas to quickly resolve links failure and reduce the load on the network. But since our network is pretty small we didn't see the need in our case.

V. iBGP, eBGP and route reflectors

The BGP configuration is shown on figure 5.

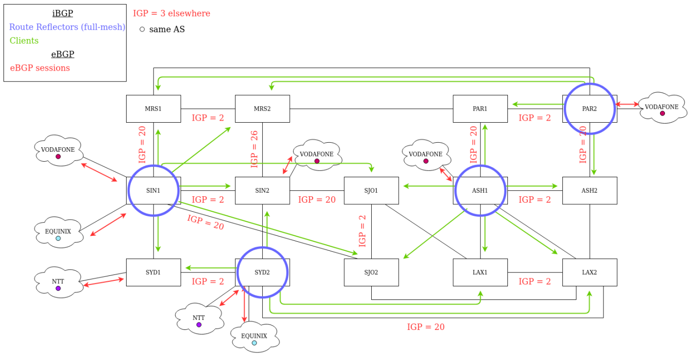


Fig. 5: BGP configuration

A. *iBGP*

There are four route reflectors (RR) in our network (circled in blue), all connected in full-mesh. The clients of these RRs are indicated by the green arrows. All the clients are connected to two different RRs in order to have redundancy. Furthermore, we always connected them to the closest RRs to avoid any issue such as loops in the network. Of course, we are aware that if the network grows, one RR (two in the case of Asia) per continent is not enough but we considered that is not difficult to implement more RR so it's not necessary for a network like this.

B. *Choice of the route reflectors*

The choice of the route reflectors was made in the following way : since we have 3 continents, we wanted at least a route reflector in each of them in order to orient the internal traffic. Indeed, since Vodafone is located in the three continents, we wanted a way to tell the routers to send the packets destined to a prefix belonging to Vodafone to the one located in their continent; because in practice, we would not want to reach it in North America if we are in Europe for example.

For the route reflectors themselves, we tried to put them the closest to the continents borders. As said before, each BGP client is connected to two RRs for redundancy. To be more precise, the first route reflector is one located in the same continent as him and the other is on another one. This is only true for the routers in Europe and America because there is only Vodafone so we found it necessary to have a link with other continents (again to ensure redundancy).

For the RRs of Asia, we decided to put two RRs because the three clients are present there and on several routers. Therefore, with two RRs, the traffic would be more distributed between the routers. The RRs are put on the routers that are connected to two clients, to make sure that the packets will go through them.

C. *Hierarchical route reflectors ?*

We chose to not use hierarchical route reflectors because we found it useless in our case. Indeed, we only have four route reflectors so the number of sessions ($\frac{n \times (n-1)}{2} = 6$) between them is acceptable. Using hierarchy would complexify something that was not too complicated from the beginning.

Furthermore, the choice of using full-mesh was more obvious given the configuration of our network : as said previously, we want at least one route reflector in each continent, which already makes 3 route reflectors. We just added an another one to distribute the asian traffic more efficiently. Also, we want the three continents to communicate. Hierarchical route reflectors would not have met these requirements.

D. eBGP

The eBGP sessions are represented by the red arrows. There is a session between the clients (Vodafone, Equinix and NTT) and the router to which they are directly linked. Indeed, the clients are located in other ASes so the only place where the eBGP sessions can be placed is on the paths between those ASes and OVH.

VI. IGP COST

As you can see in the figure 5, we have set IGP costs between all the routers inside the network. At the beginning, we had set the IGP cost between the routers of the same city (ex: between the two `SIN` routers) to 1, of the same continent but not the same city to 3 and between continents to 11. These choices were made in order to keep the traffic local: since the cost is high to go in another continent, the routers are less likely to choose these routes. With the same logic, same cities are more proned to communicate with each other since the cost is minimal. However, there is a problem with IPMininet: if we do that IPMininet crash. The problem is on the side of IPMininet rather than ours.

Therefore, we have set IGP cost values to 2 between the same city, 3 for the cities in the same continent and 20 between the continents. There is one exception: we set a cost of 26 between MRS2 and SIN2 in order to avoid a route deflection.

VII. COMMUNITIES

We were asked to define some BGP communities in our network. By looking at existing examples, like [3] for NTT, we found interesting to use the following ones:

Action	community
Set local-pref to 80	80:80
Set local-pref to 200	200:200
Set AS prepending to 1	1:100
Set AS prepending to 2	2:100
Set AS prepending to 3	3:100
Set no-export to North America	10:15
Set no-export to Europe	20:15
Set no-export to Asia	30:15

Let's describe each of them.

A. Set local-pref

This is a pretty basic feature that we provide in our network. Our peers can set community `80:80` or `200:200` on their prefix to set the local-pref to 80 or 200. If no community is set, the local-pref value is set at this initial value 100.

B. AS prepending

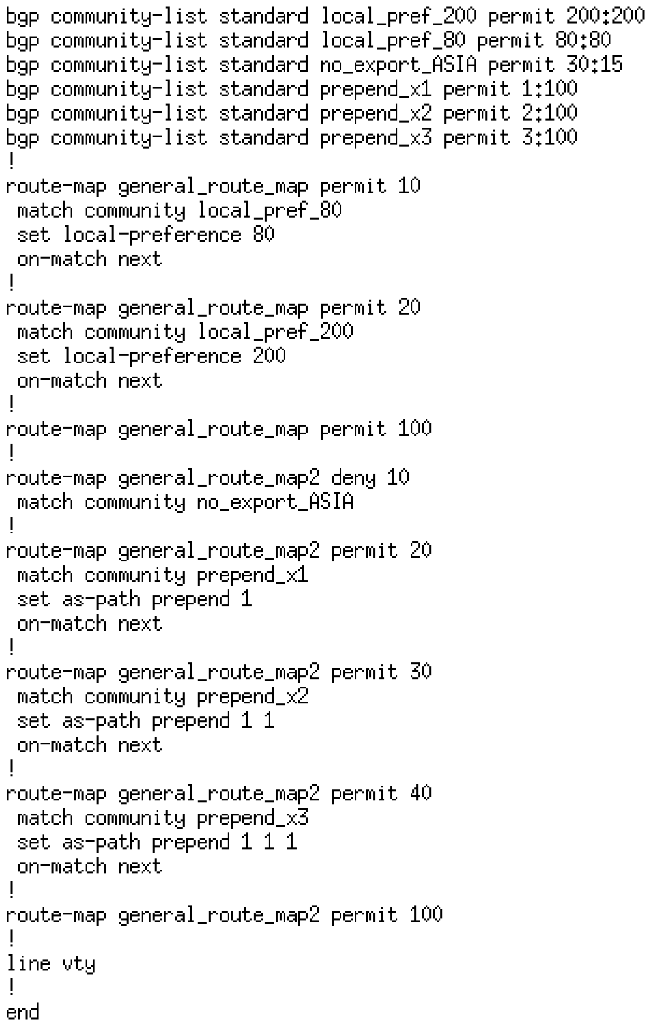
AS-prepending is useful in many cases. For instance, it enables our peers to set up back-up routes. Inside our network we provide ”to prepend one, two or three times”.

C. No export

The border routers are configured to handle the no-export communities. For instance, if one of these routers from Europe receives a BGP packet with the ”no-export to Europe” attached to it, it won't forward this packet to his peers.

D. Implementation

In practice, the communities are implemented by using Route Maps. You can see their definition in the following figure:



```
bgp community-list standard local_pref_200 permit 200:200
bgp community-list standard local_pref_80 permit 80:80
bgp community-list standard no_export_ASIA permit 30:15
bgp community-list standard prepend_x1 permit 1:100
bgp community-list standard prepend_x2 permit 2:100
bgp community-list standard prepend_x3 permit 3:100
!
route-map general_route_map permit 10
 match community local_pref_80
 set local-preference 80
 on-match next
!
route-map general_route_map permit 20
 match community local_pref_200
 set local-preference 200
 on-match next
!
route-map general_route_map permit 100
!
route-map general_route_map2 deny 10
 match community no_export_ASIA
!
route-map general_route_map2 permit 20
 match community prepend_x1
 set as-path prepend 1
 on-match next
!
route-map general_route_map2 permit 30
 match community prepend_x2
 set as-path prepend 1 1
 on-match next
!
route-map general_route_map2 permit 40
 match community prepend_x3
 set as-path prepend 1 1 1
 on-match next
!
route-map general_route_map2 permit 100
!
line vty
!
end
```

Fig. 6: Route-map configuration

As you can see two route-maps are used: one on input and the other one on output.

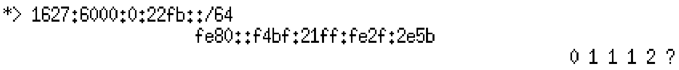
`general_route_map` handles the input. It thus handles the set local-pref communities. On the other hand `general_route_map_2` handles the output and is used for AS-prepending. The last permits are used to counter the implicit deny at the end of every route-maps. We use `on-match next` in our conditions to enable the use of multiple communities on the same prefix.

Since the figure 6 comes from the SYD2 router, the no_export_ASIA is present. If we had chosen PAR2 the no_export_EU would have been present instead.

We decided to implement our communities using FRROUTING commands directly. We then used the pexpect library to automate the deployement of the commands. The pexpect scripts are located in the /scripts folder.

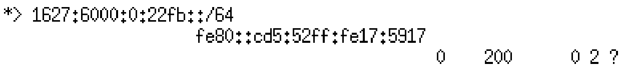
- `BGP_ccl_COMM_NAME.py` to create the community-lists.
- `BGP_PX1_COMML_RMNAME_SEQ.py` to add ”append x1” action to a route-map.
- `BGP_PX2_COMML_RMNAME_SEQ.py` to add ”append x2” action to a route-map.
- `BGP_PX3_COMML_RMNAME_SEQ.py` to add ”append x3” action to a route-map.
- `BGP_COMML_LPREF_RMNAME_SEQ.py` to add set local-pref action to a route-map.
- `BGP_deny_COMM_RM_SEQ.py` to deny a particular community. This scripts is used to handle the no-export’s communities.
- `BGP_NEIGHBOR_RMAP_INOUT.py` to apply a route-map to a peer.

A test is provided at the end of the script which sets local-pref_200, prepend X2 and no-export to EU communities for routes from the ”EQXSIN1” router. The `applyCommunities` function is very useful to test our communities.



```
*> 1627:6000:0:22fb::/64
                    fe80::f4bf:21ff:fe2f:2e5b
                                                            0 1 1 1 2 ?
```

Fig. 7: AS prepending

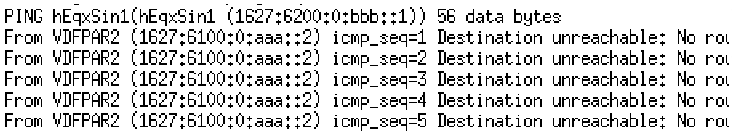


```
*> 1627:6000:0:22fb::/64
                    fe80::cd5:52ff:fe17:5917
                                                0    200      0 2 ?
```

Fig. 8: Local-pref

As you can see on the figures 7 and 8, these communities are working well.

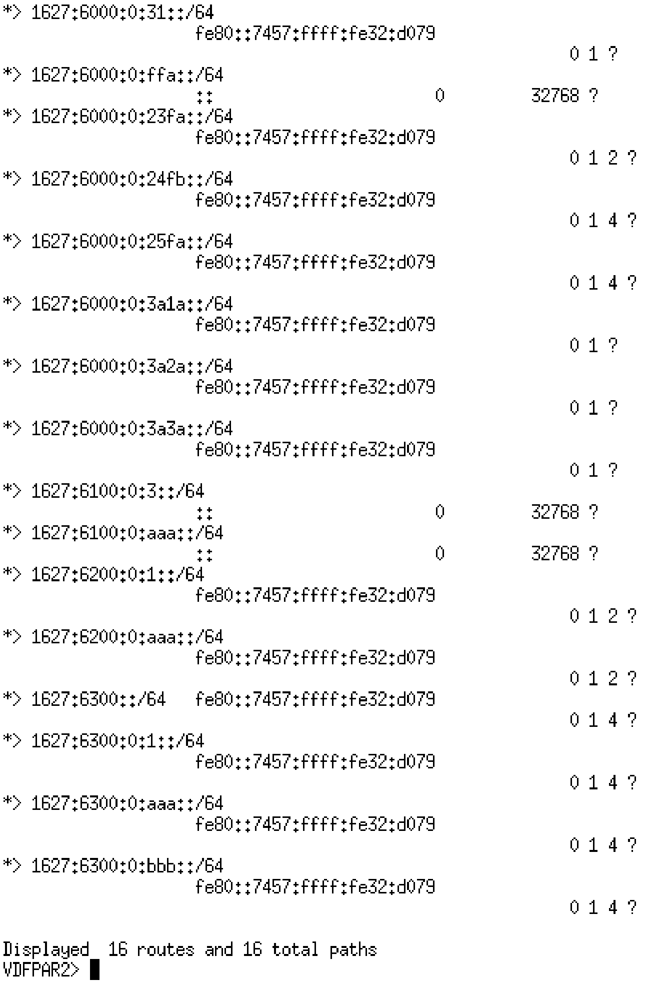
To test the no-export communities we can simply ping the hEqxSin1 host from another host in Europe, for instance hVdfPar2, and see that no routes are available (as shown on figure 9).



```
PING hEqxSin1(hEqxSin1 (1627:6200:0:bbb::1)) 56 data bytes
From VDFPAR2 (1627:6100:0:aaa::2) icmp_seq=1 Destination unreachable: No rou
From VDFPAR2 (1627:6100:0:aaa::2) icmp_seq=2 Destination unreachable: No rou
From VDFPAR2 (1627:6100:0:aaa::2) icmp_seq=3 Destination unreachable: No rou
From VDFPAR2 (1627:6100:0:aaa::2) icmp_seq=4 Destination unreachable: No rou
From VDFPAR2 (1627:6100:0:aaa::2) icmp_seq=5 Destination unreachable: No rou
```

Fig. 9: Ping test

We can also look at the BGP table of VDFPAR2 and see that hEqxSin1 (`1627:0000:0:22fb::/64`) is missing:



```
*> 1627:6000:0:31::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 ?
*> 1627:6000:0:ffa::/64
                    ::                       0         32768 ?
*> 1627:6000:0:23fa::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 2 ?
*> 1627:6000:0:24fb::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?
*> 1627:6000:0:25fa::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?
*> 1627:6000:0:3a1a::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 ?
*> 1627:6000:0:3a2a::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 ?
*> 1627:6000:0:3a3a::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 ?
*> 1627:6100:0:3::/64
                    ::                       0         32768 ?
*> 1627:6100:0:aaa::/64
                    ::                       0         32768 ?
*> 1627:6200:0:1::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 2 ?
*> 1627:6200:0:aaa::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 2 ?
*> 1627:6300::/64   fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?
*> 1627:6300:0:1::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?
*> 1627:6300:0:aaa::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?
*> 1627:6300:0:bbb::/64
                    fe80::7457:ffff:fe32:d079
                                                            0 1 4 ?

Displayed  16 routes and 16 total paths
VDFPAR2> █
```

Fig. 10: BGP table

VIII. Anycast

Inside the network, we have implemented anycast. We have modeled it by adding three servers: one connected to the router of SJO2, another one to the router of PAR2 and the last one to SIN2. An illustration can be found on the figure 11, where the anycast servers are in orange.

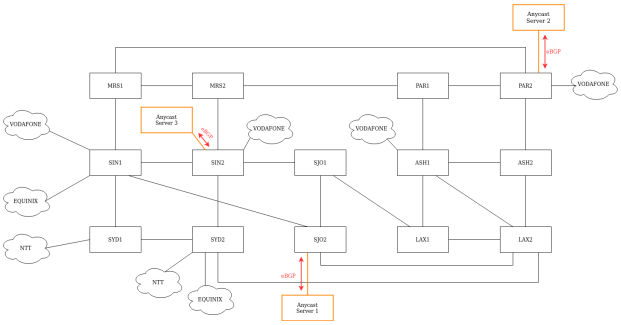


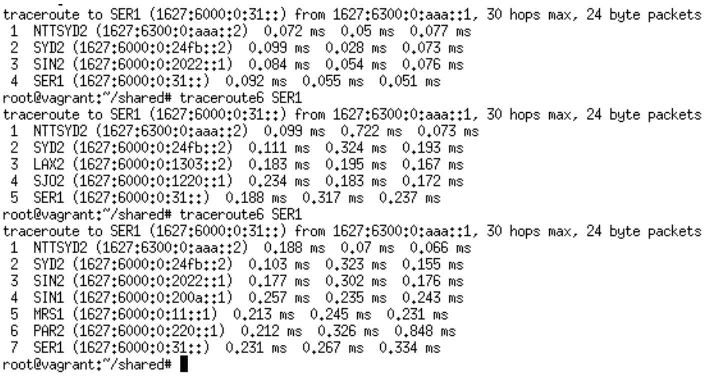
Fig. 11: Anycast servers

We wanted to implement a DNS server since it's a good use case for anycast servers, anycast working well with UDP. Unfortunately, we are limited by IPMininet since the Named daemon only works for hosts.

For the implementation, we had to choose between two pertinent daemons to put on the router: OSFP or BGP. At first, we wanted to use OSPF because the "hello request" which allows to verify if a router is still alive is more frequently sent than with BGP, every 1 second against 180 seconds for BGP. However, in terms of security, OSPF is not secure. In fact, if the router which represents the server has a problem, it could send OSPF packets in the network and therefore break the configuration, which is unacceptable. With BGP, we can use route-maps to filter imports so this problem would not happen.

Therefore, we decided to implement BGP on the router and to decrease the time of fault detection to address the timing problem: we have set the rate of "keep alive packet" to 3 seconds and the hold time to 10 seconds. Now, if a client wants to reach the server, he chooses the one with the "shortest" path, but if the link between this server fails he will be redirected to an other server.

Let's take an example: we are the NTT client connected to the router SYD2. If the client tries to send a request to the server, it'll go to the server connected to the SIN2 router (closest server). Now imagine this server fail, the client will automatically be redirected to the server attached to the SJO2 router. If this server fail too, the client will be redirected to the server attached to PAR2. The figure 12 below shows three `traceroute6` outputs. The first represents the normal situation, the second if the first server fails and the last if the second server fails.



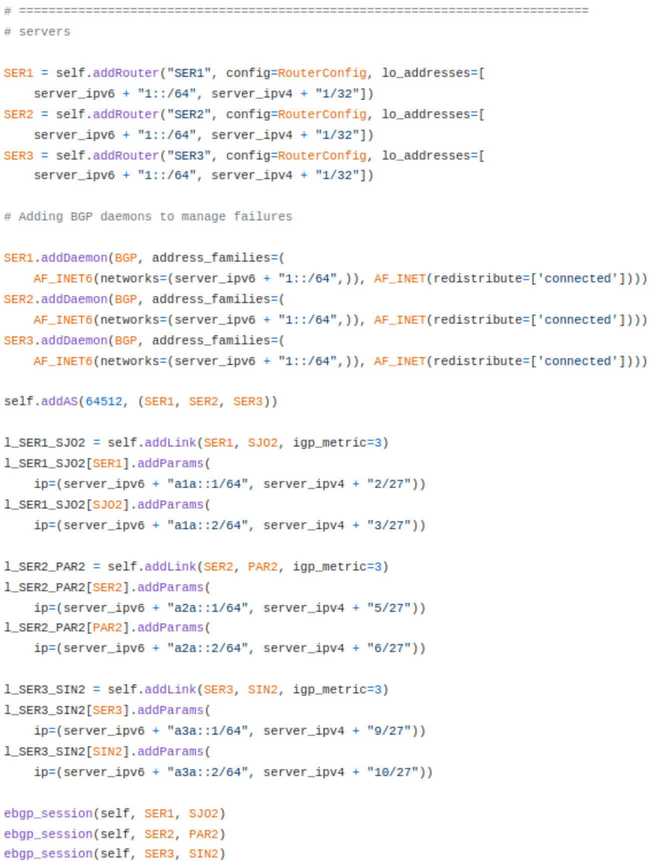
```
traceroute to SER1 (1627:6000:0:31::) from 1627:6300:0:aaa::1, 30 hops max, 24 byte packets
 1  NTTSYD2 (1627:6300:0:aaa::2)  0.072 ms  0.05 ms  0.077 ms
 2  SYD2 (1627:6000:0:24fb::2)  0.099 ms  0.028 ms  0.073 ms
 3  SIN2 (1627:6000:0:2022::1)  0.084 ms  0.054 ms  0.076 ms
 4  SER1 (1627:6000:0:31::)  0.092 ms  0.055 ms  0.051 ms
root@vagrant:~/shared# traceroute6 SER1
traceroute to SER1 (1627:6000:0:31::) from 1627:6300:0:aaa::1, 30 hops max, 24 byte packets
 1  NTTSYD2 (1627:6300:0:aaa::2)  0.099 ms  0.722 ms  0.073 ms
 2  SYD2 (1627:6000:0:24fb::2)  0.111 ms  0.324 ms  0.193 ms
 3  LAX2 (1627:6000:0:1303::2)  0.183 ms  0.195 ms  0.167 ms
 4  SJO2 (1627:6000:0:1220::1)  0.234 ms  0.183 ms  0.172 ms
 5  SER1 (1627:6000:0:31::)  0.188 ms  0.317 ms  0.237 ms
root@vagrant:~/shared# traceroute6 SER1
traceroute to SER1 (1627:6000:0:31::) from 1627:6300:0:aaa::1, 30 hops max, 24 byte packets
 1  NTTSYD2 (1627:6300:0:aaa::2)  0.188 ms  0.07 ms  0.066 ms
 2  SYD2 (1627:6000:0:24fb::2)  0.103 ms  0.323 ms  0.155 ms
 3  SIN2 (1627:6000:0:2022::1)  0.177 ms  0.302 ms  0.176 ms
 4  SIN1 (1627:6000:0:200a::1)  0.257 ms  0.235 ms  0.243 ms
 5  MRS1 (1627:6000:0:11::1)  0.213 ms  0.245 ms  0.231 ms
 6  PAR2 (1627:6000:0:220::1)  0.212 ms  0.326 ms  0.848 ms
 7  SER1 (1627:6000:0:31::)  0.231 ms  0.267 ms  0.334 ms
root@vagrant:~/shared# █
```

Fig. 12: Traceroute6 with different link failures

The servers S1, S2 and S3 have the same loopback address. This is why both addresses have the same destination S1.

To implement this service we have created three routers with the configuration `RouterConfig` and added a daemon BGP. The three routers have the same loopback address in IPv4 and IPv6 and they are all in the same autonomous system. As we have chosen to use BGP on the router, we have initialized an eBGP session between the routers simulating the servers and the routers of SJO2, PAR2 and SIN2.

The implementation is shown in the figure 13.

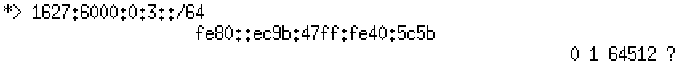


```
# ==========================================================================
# servers

SER1 = self.addRouter("SER1", config=RouterConfig, lo_addresses=[
    server_ipv6 + "1::/64", server_ipv4 + "1/32"])
SER2 = self.addRouter("SER2", config=RouterConfig, lo_addresses=[
    server_ipv6 + "1::/64", server_ipv4 + "1/32"])
SER3 = self.addRouter("SER3", config=RouterConfig, lo_addresses=[
    server_ipv6 + "1::/64", server_ipv4 + "1/32"])

# Adding BGP daemons to manage failures

SER1.addDaemon(BGP, address_families=(
    AF_INET6(networks=(server_ipv6 + "1::/64",)), AF_INET(redistribute=['connected'])))
SER2.addDaemon(BGP, address_families=(
    AF_INET6(networks=(server_ipv6 + "1::/64",)), AF_INET(redistribute=['connected'])))
SER3.addDaemon(BGP, address_families=(
    AF_INET6(networks=(server_ipv6 + "1::/64",)), AF_INET(redistribute=['connected'])))

self.addAS(64512, (SER1, SER2, SER3))

l_SER1_SJO2 = self.addLink(SER1, SJO2, igp_metric=3)
l_SER1_SJO2[SER1].addParams(
    ip=(server_ipv6 + "a1a::1/64", server_ipv4 + "2/27"))
l_SER1_SJO2[SJO2].addParams(
    ip=(server_ipv6 + "a1a::2/64", server_ipv4 + "3/27"))

l_SER2_PAR2 = self.addLink(SER2, PAR2, igp_metric=3)
l_SER2_PAR2[SER2].addParams(
    ip=(server_ipv6 + "a2a::1/64", server_ipv4 + "5/27"))
l_SER2_PAR2[PAR2].addParams(
    ip=(server_ipv6 + "a2a::2/64", server_ipv4 + "6/27"))

l_SER3_SIN2 = self.addLink(SER3, SIN2, igp_metric=3)
l_SER3_SIN2[SER3].addParams(
    ip=(server_ipv6 + "a3a::1/64", server_ipv4 + "9/27"))
l_SER3_SIN2[SIN2].addParams(
    ip=(server_ipv6 + "a3a::2/64", server_ipv4 + "10/27"))

ebgp_session(self, SER1, SJO2)
ebgp_session(self, SER2, PAR2)
ebgp_session(self, SER3, SIN2)
```

Fig. 13: Implementation of anycast

To secure the link between the routers and the servers, we have implemented a password on the link. We would use a SHA256 but IPMininet doesn't allow that, so we used MD5. Of course, the AS containing the servers is a private AS and is therefore never announced outside OVH, we have hidden the AS number to the peers. To do this, we have implemented route map on the routers connected to the server (this part is not in the code above because to do that we use "pexpect" to launch the command directly in FRRouting with the script `BGP_rm_aspath_ASN_RM_SEQ.py`).

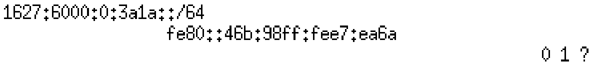
The figure 14 below shows the situation where we don't hide the AS number:



```
*> 1627:6000:0:3::/64
                          fe80::ec9b:47ff:fe40:5c5b
                                                                            0 1 64512 ?
```

Fig. 14: Not hidden AS number

And our situation is on the figure 15:



```
 1627:6000:0:3a1a::/64
                          fe80::46b:98ff:fee7:ea6a
                                                                     0 1 ?
```

Fig. 15: Hidden AS number

IX. Security

When designing our network, we had to think about some ways to make it secure, in order to avoid attacks coming from outside. In this section, we will present you the techniques we used to achieve this goal.

A. *Password on BGP*

For the security, we have implemented a password between the peers connected to the network and the routers at the edge of it. For the moment, we have used MD5 hash because FRRouting can only support that. We are aware that this hash is not the most secure, which is why we wanted to implement a SHA256 hash at the beginning.

There are two interesting things to note about this password:

- To resolve the problem of how to store the password and how to attribute a password for each peer, the hash of a peer is a hash of a mix of a secret password and the name of the peer. Like this, the only thing we are storing is the secret password.
- Why would we use a password on BGP since it is in a higher layer than TCP ? Answer: because it can prevent the TCP *reset attack*. *Reset attack* is an attack deployed by a ”hacker” or a firewall who sends reset packets to the router to interrupt the connection. With the password, we are protected against the attackers.

B. *Password on OSPF*

Similarly to BGP, we have implemented MD5 hash over the OSPF packets. Like BGP, we wanted to implement a more secure hash like SHA256 but IPMinet doesn't support this encryption.

To use OSPF password, we simply add the option `password = "yourPassword"` to a link the same way as you will set an `igp_metric` for instance.

C. *TTL check*

To prevent BGP connections from unwanted peers, we decided to use the `neighbor PEER ttl-security hops NUMBER` FRROUTING command. It prevents connections from peers which are more than `NUMBER` hops away. The base TTL for IPv6 packets is 64, and we increased it to 255. By doing this we are sure that we effectively don't connect to peers further than `NUMBER` hops. `NUMBER` is set to 2 in our network.

D. *Limitation of the number of prefixes*

We had thought about an implementation of a limitation of the number of prefixes announced. In theory it is a good idea. However, according to Pr. Bonaventure, it is difficult to implement realistically because we don't have any information about the traffic of each peer, so the limit number will be randomly attributed. So we decided to not implement that in our network.

X. Limitations of our design

A. *IPv4 and eBGP on ipmininet*

At the beginning of the project, when BGP wasn't implemented, the IPv4 worked, but now that we have implemented BGP, the IPv4 only works partially. For instance, `ping4all` doesn't work totally and we don't understand why.

B. *Convergence time*

The network takes a little while to converge totally (maximum one minute). So when we want to use commands or test our network, we need to wait for the network to converge. Given the number of routers, protocols and scripts to launch, it seems to take a reasonable time for us.

C. *Ping6all*

If we launch a `ping6all`, there are only 40 pings that match and 16 drops, which is normal since the hosts from the same peer are not connected in their own AS in our configuration. To fix this problem, we can link all the routers of the same peer together, but since we are not in charge of the peer configuration, we let this task to network engineers of the peer.

XI. How to launch the code

To respond to all the specifications, we have made some modifications to IPMininet. So in the file of the project, we have added our version of IPMininet. Furthermore, in order to configure the routers using FFRouting commands, we use the Python library `pexpect`.

Before launching our code, you need to install our IPMininet version like the following:

```
sudo pip install −e MyIpmininet/
```

You also need to install `pexpect` :

```
sudo pip install pexpect
```

XII. Conclusion

With this project, we are now able to design a network, from the routers organisation to the BGP configuration, passing by the addressing plan. We also took some measures to ensure the security of the network. We then upgraded the network to use the BGP communities. And we finally made it support anycast and designed it to detect failures efficiently.

All of that was implemented on IPMininet in order to see if our design choices were pertinent.

References

[1] Computer network. (2020). Retrieved December 10, 2020, from https://en.wikipedia.org/wiki/Computer_network.

[2] FRRouting User Guide. (2017). Retrieved December 10, 2020, from http://docs.frrouting.org/en/latest/.

[3] Routing Policies. (2020). Retrieved December 10, 2020, from https://www.gin.ntt.net/support-center/policies-procedures/routing/.

[4] Introduction to Route-maps. (2020). Retrieved December 10, 2020, from https://networklessons.com/cisco/ccnp-route/introduction-to-route-maps

[5] Technology & Support. (2020). Retrieved December 10, 2020, from https://community.cisco.com/t5/technology-and-support/ct-p/technology-support.

[6] IPMininet's documentation!. (2019). Retrieved December 10, 2020, from https://ipmininet.readthedocs.io/en/latest/.

[7] Telnet to login with username and password to mail Server. (2020). Retrieved December 10, 2020, from https://stackoverflow.com/questions/27941036/telnet-to-login-with-username-and-password-to-mail-server?fbclid=IwAR1bq8O-2AUTw7naOGe0UXn_GNh5mGxEDE7sOR_I93oO4RB_8PI6Hl1AV4Q.

[8] Pexpect version 4.8. (2013). Retrieved December 10, 2020, from https://pexpect.readthedocs.io/en/stable/.

[9] Implementing Anycast in IPv4 Networks. (2004). Retrieved December 10, 2020, from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.6367&rep=rep1&type=pdf.

[10] RIPE NCC. (2020). Retrieved December 10, 2020, from https://stat.ripe.net/0.

[11] PeeringDB. (2020). Retrieved December 10, 2020, from https://www.peeringdb.com.

[12] BGPview. Retrieved December 10, 2020, from https://bgpview.io/.

[13] Guides OVHcloud. (2020). Retrieved December 10, 2020, from https://docs.ovh.com.

[14] OVHcloud Network Weathermap. (2020). Retrieved December 10, 2020, from http://weathermap.ovh.net/.

[15] The network layer. (2014). Retrieved December 10, 2020, http://cnp3book.info.ucl.ac.be/1st/html/network/network.html#the-border-gateway-protocol.

[16] BGP Route Reflection Revisited. (2012). Retrieved December 10, 2020, http://irl.cs.ucla.edu/~j13park/rr-commag.pdf.

[17] RFC 4271 - A Border Gateway Protocol 4 (BGP-4). (2006). Retrieved December 10, 2020, https://tools.ietf.org/html/rfc4271.html.