LINGI2142 : Networks Project
Configuration and deployment of an ISP network

Group 1
van de Walle Nicolas - 2790 1600
Gobeaux Alexandre - 4219 1600
Ortegat Guillaume - 0854 1500

Abstract

People across the world are now more connected than ever before and with the expansion of the Internet of things (IOT), estimations approximate to 75 billion the amount of connected devices in 2025[1]*. Such connectivity between endpoints has been and will still remain a big challenge. They need a network to communicate and this network is composed of thousands of interconnected sub-networks. Deploying and managing such sub-networks can be very challenging as it consists in multiple small components and protocols and relies on others' connectivity. We will thus explain our approach to solve these problems and manage a network and its complexity.*

Introduction

In the following report, we will discuss the case of an Internet service provider (ISP) which has for aim to provide connectivity to customers and exchange information with other ISPs and autonomous systems in general.

The objective is to be able to easily deploy and maintain the network. It must be flexible enough to handle the adding of new routers/components, external peers, clients, providers etc. The configuration must be an easy, centralized and non-repetitive task.

To simulate this Internet service provider, we had at our disposal a virtual machine running Linux and each component (mostly routers) are namespaces in this VM. Some scripts that will be explained later in this report have been coded in bash scripting or python. *Note: some scripts require Python 3.6+ which was not initially installed on the system.*

The following report will detail the general architecture of the backbone network and the addressing plan allowing us to keep the devices organized inside it. We will also discuss the different BGP (Border Gateway Protocol) connections that have been established with other autonomous systems (i.e. groups) and then, explain the software configuration of the network and how to deploy and start it.

The next 3 sections explain how we performed the testing of the network in general (internal and external), what kind of security improvements we made and finally, which advanced BGP communities we implemented to go deeper in the usage of BGP and get more optimized external connections.

[1]Source: `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`

1 Overview of the network

The following figure (Figure 1) gives you an overview of the currently implemented network.

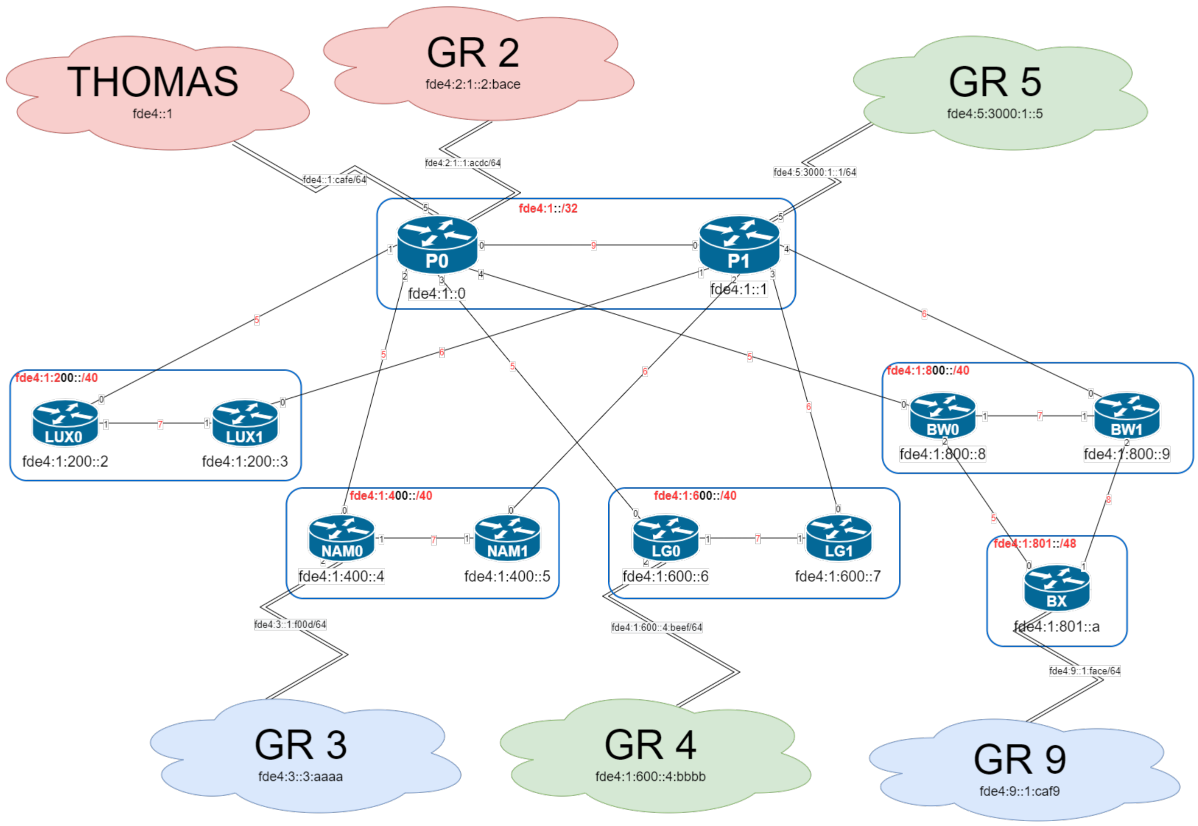

Figure 1: Network map

Our first design choice was to replicate every router of the core network. It allows us to handle failures as every router has its backup which handles the traffic if something happens to the other one.
We decided to subdivide the architecture according to (some of) the Belgian provinces.

- Luxembourg (`LUX0` and `LUX1`)
- Namur (`NAM0` and `NAM1`)
- Liège (`LG0` and `LG1`)
- Brabant-Wallon (`BW0` and `BW1`)

But also added some extra routers such as `P0` and `P1` – which are the main routers of the architecture as depicted on figure 1 – or `BX` (Brussels) which is not replicated as it is considered to be a small one.

2 Addressing plan

Our ISP is using IPv6 packets and we have been given the `fde4:1::/32` prefix. It means that we have $2^{96} = 80 \times 10^{27}$ IP addresses. In order to keep the network well organized, we subdivided the `/32` IP prefix in more precise `/40` prefixes which represent the different regions which have been listed before. Those regions now have their own prefix and we can have up to 256 of them:

- LUX: `fde4:1:200::/40`
- NAM: `fde4:1:400::/40`
- LG: `fde4:1:600::/40`
- BW: `fde4:1:800::/40`

Now that we have the different regions, we can talk about the different kinds of customers. Private customers do not need the same amount of IP addresses as enterprises. That is why we decided to give /48 prefixes to our business customers whereas our private customers will get a /56 prefix allowing them to create 256 subnetworks inside their home[2].

3 Communicating inside the network

All the routers of the backbone network are running `OSPF` and `BGP`.

`OSPF` allows them to know how to reach the other routers and to keep up with failures as it computes the best path to a specific router.

`BGP` can be found in two forms: internal `BGP` (`iBGP`) to communicate the external prefixes inside the network and external `BGP` (`eBGP`) to send and receive the different prefixes that can be reached from/via our ISP.

The infrastructure uses 2 levels of route reflectors (`RR`) to announce the prefixes in the network.

`P0` and `P1` are level 1 `RR` and `BW0` and `BW1` are level 2 `RR`. Each router establishes an `iBGP` session with the 2 higher-level route reflectors (e.g. `LUX0` has an active `iBGP` session with `P0` and `P1`. `BX` has an active `iBGP` session with `BW0` and `BW1`).

The `iBGP` sessions are depicted using green lines on figure 2 and are, as usual, quite messy.

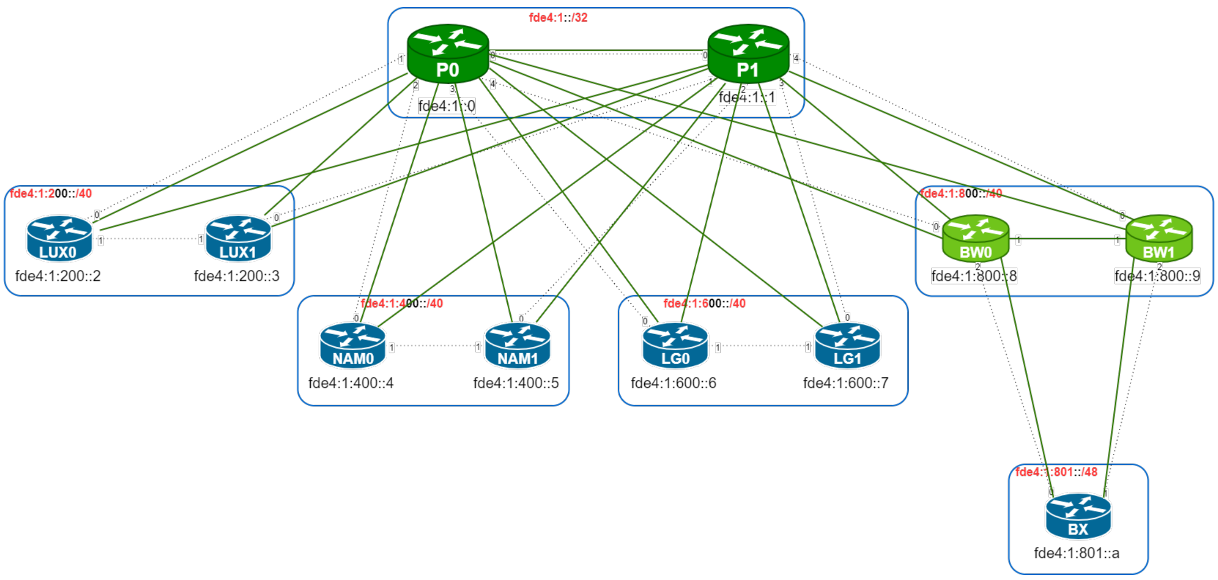

Figure 2: `iBGP` sessions between routers

4 `BGP` neighbors

As depicted on figure 1 and explained before, our ISP interacts with other autonomous systems. We have different relationships such as provider, client and shared-cost which implies different prefix advertisement policies. Those policies will be explained later in this report in the advanced `BGP` section.

To establish the `eBGP` session with our peers, we use a /64 subnet owned by either one of us. If the used subnet is part of our prefix (`fde4:1::/32`), it follows the addressing plan and uses an `IP` corresponding to the region and containing information about the peering `AS` (e.g. `fde4:1:600::4:beef/64`: `600` for Liège and `4` for the `AS65004`).

Our providers (red clouds in figure 1) are connected to the northen routers, clients (blue clouds in figure 1) are connected to the southern routers and finally, shared-cost peers (green clouds in figure 1) are connected to « any » router of the network. Here is the list of the `BGP` peers, the relationship and the router they are connected to.

- `AS64512` : Provider connected to `P0`
- `AS65002` : Provider connected to `P0`
- `AS65005` : Shared-cost connected to `P1`

[2]good practices assume that we do not create subnets smaller than /64 subnets

- AS65003 : Client connected to NAM0
- AS65004 : Shared-cost connected to LG0
- AS65009 : Provider connected to BX

5 Network configuration

As you now have a good understanding of our network, let's get into the details of the configuration.

To build and configure the network, we require 2 files.

- The network topology file (`project_topo`) which is used by the `create_network.sh` script to generate the different routers (i.e. namespaces).
- The configuration file (`config.json`) which is a json file containing all the information to recreate the network.

5.1 The `config.json` file

An example of `config.json` is given in the Appendix A at the end of this report.

This file contains, for each router, all the interfaces it has and all the relevant information depending on the type of session established on it (OSPF of BGP).

As explained before, thanks to this file, we can recreate the whole topology, OSPF/BGP sessions and IP plan of the ISP. It allows the network to be versatile as adding a router is as easy as adding a JSON object to the routers list.

The way it works is that it generates the configuration files (`bgpd.conf` and `ospf.conf`) and the scripts that are used and executed when to network boots and starts (set links up, run OSPF daemon etc.). This generation is enabled by the usage of mako templates.

5.2 Starting the network

To start the network, move to the CampusNetwork folder and run:

```
vagrant@group1:~/CampusNetwork# sudo ./rebuild.sh
```

That's it, the network has been started. You might need to wait up to 1 minute before OSPF converges.

6 Testing the network **[Nicolas van de Walle]**

When we are managing a network, it is very important to be able to verify that everything works as expected. Therefore, we needed to develop some tests allowing us to verify the differents parts of the network.

6.1 Keeping track of the changes

The main decision we made was to log everything that happens during the tests. In fact, being able to know when a certain functionality was working and when it started to malfunction is one of the most important information to debug. Therefore, when the tests are run, a log file (`ROUTER_NAME.log`) is generated for each router in the `tests/logs/` folder.

The logs stored in those files can have different levels.

- INFO is used to log success messages (e.g. if a router can reach another one without any issues or following the right path)
- WARNING is used when something unexpected happens but it won't affect the availability of the network (e.g. Router LUX0 reached P1 via P0 and not via LUX1)
- ERROR is used to inform about an unexpected behavior which affects the availability of the routers/network (e.g. Router P0 was unreachable from BX)

An important point to notice is that, everything is automatic. If tomorrow, you decide to add a new router inside the network, it will be tested as all the others because it will be listed in the configuration file and this configuration file is the starting point for all the tests that are detailed below.

6.2 The core network

Testing the core network consists in verifying if every part of it (i.e. routers) is reachable from everyone inside the network.

6.2.1 Reachability tests

The way it works, in our case, is that we connect to each router and try to ping all the routers of the network. This easy task gives us a quick overview of what is connected and thus, if OSPF worked correctly.

To run this ping test, just run the following command:

```
vagrant@group1:~/CampusNetwork# ./test-ping.sh
```

6.2.2 OSPF path tests

Now that we know that all the routers are reachable, we need to verify that the packets follow the right path. First, we decided to dynamically create a weighted graph representing our architecture using the python `networkx` package[3]. Thanks to this graph which was created using the `config.json` file, we can compute the theoretical best path between two routers.

The next step is to analyze which path is really used. We use the `traceroute6` command to get the different hops of the packets between every pair of routers.

The final step is to compare both path to see if they match and thus, verify that OSPF made a good job and informed the routers correctly.

To go even further, we decided to randomly break some links between the routers to see if they react as expected. As we have the graph, we are also able to remove an edge between two nodes to simulate the failure. We run the `traceroute` tests again and can see if this OSPF reacted correctly to this « failure ». *Note: after reactivating the links, OSPF can sometimes take a long time to detect and propagate the update.*

To run these OSPF tests (without and then with link failures), just run the following command:

```
vagrant@group1:~/CampusNetwork# ./test-ospf.sh
```

Note: these tests require Python 3.6 to work. This version of python was not initially installed on the VMs[4]

6.3 The outside world

To test our connection with the outside world (eBGP) and the propagation of the information inside our network with iBGP, we verify that every AS we are supposed to be connected with is well connected and that the session is active.

Therefore, for each router that communicates with BGP, we first verify that the session is started on it and then, make sure that the peering router has also started a BGP session on his side. We can see, thanks to the `show bgp summary` command, a specific state saying if the connection is well established and if not, what the state is.

If the session is established between the ASes, we try to ping an IP address of the peer network to ensure that the BGP session is used to send the right information and that the filters work correctly.

To run these BGP tests, just run the following command:

```
vagrant@group1:~/CampusNetwork# ./test-bgp.sh
```

To run all the previously detailed tests sequentially (`ping -> OSPF -> BGP`), run the following command:

```
vagrant@group1:~/CampusNetwork# ./test.sh
```

Note: running those test can quite quite a lot of time (approximately 15 minutes in total due to the OSPF test)

[3]http://networkx.github.io/

[4]You can follow this tutorial to install it. It takes quite a long time: https://www.rosehosting.com/blog/how-to-install-python-3-6-4-on-debian-9/

7 Securing the network [Alexandre Gobeaux]

To secure the network, several things were done. First, we configured MD5 passwords for the iBGP and eBGP sessions. We chose to have different passwords logic for the different eBGP-peers we had, so that nobody could easily find another password we used with another group. We configured MD5 passwords with groups 3, 4, 5 and 9 and those were tested : we tried to use a wrong password and could not connect, we had to choose the right password. We did not manage to configure an MD5 password with the group 2 since we have not been connected for a long time. A better way to secure the iBGP and eBGP sessions would have been to use the Generalized TTL Security Mechanism (GTSM) but it would have required our peers to also use it (for the eBGP sessions).
After that, we created ip6tables rules to filter OSPF and BGP packets : OSPF packets from outside the network should not get to/through our network and BGP packets should always get to our external routers but not through them (they should not be forwarded by our external routers). After updating our configuration and mako files to implement this, the network was fully functional. Tests were made to verify the configuration, in addition to running Nicolas' tests, ip6tables lines were also tested by disallowing one of our router of reaching the other routers with OSPF and the result was that no router could reach it, as expected.
Finally, we created other ip6tables rules to filter packets using private addresses or not allowed ones. In fact, we should not receive private addresses from other groups. Although, since Link-Local Addresses and Unique Local Addresses (ULAs) are used by BGP, we could not really filter them with my ip6tables rules without breaking our network. Those addresses were thus not filtered. The filtered addresses were :

- The unspecified address which should not be used as destination (`::/128`);
- The Benchmarking addresses (`2001:0002::/48`);
- The Orchid addresses (`2001:0010::/28`);
- The Documentation addresses (`2001:db8::/32`);
- and the Multicast addresses (because those should not be used as source (`ff00::/8`).

These choices were based on a RIPE document[5].
For the ip6tables rules (both OSPF, BGP and filtering some addresses), only the tables INPUT and FORWARD have been used. We had a choice to make : we could either use the OUTPUT table also, to filter the traffic going out, or we could assume that the filtering our own packets would not be useful enough. Both options have advantages and drawbacks but we chose the second one so that the delay in our network would not be impacted by a process checking all the lines of the OUTPUT table. Another reason is that we knew that we did not use the type of addresses we filtered, thus we did not need to put those rules in this table.
After that, we decided to protect our network against DDoS attacks. To do so, we decided to put a special rule on the interfaces that are linked to other ASes : accept only 30 packets per second over those interfaces. If this amount is exceeded, drop those files and log some of them (the logging mechanism is explained in the next paragraph). The rate of 30 packets per second was chosen with this strategy : we used tcpdump to check how many packets were transmitted over the external interfaces and got a rate of approximately 2.7 packets/second. Thus, we decided that 30 packets/second would be a great maximum rate value.
Finally, we decided to implement a logging system. Our logging system logs the packets we chose to drop. ip6tables provides an option to log information. We decided to limit the rate of logging to 2 logs/minute because it is not useful to log every request and it is needed to limit the logging rate in case of DDoS attacks. All those features were implemented using the `boot.mako`, bgpd.mako and `config.json` files, so everything will continue to work if you decide to add a router for example.
We also wanted to implement a protection against a customer/client trying to spoof an IP address which does not belong to its AS but, unfortunately, this was not really possible. In fact, during the project, clients did not use the addresses we gave to them, so it was not possible to implement this.
Concerning the BGP filters, they will be discussed in the next section.

8 Advanced BGP configuration [Guillaume Ortegat]

In this section, we will see what we set up to manage the different relations with the other ASes and minimize the size of the BGP routing tables. We use two main tools to achieve this goal: filters and BGP communities.
To implement filters and communities, we implement one by one each filter and community. Between each implementation, we test the network configuration, checking if all the BGP routing tables are as expected.
Notice that we made a connection with the group 4 to test all the things we installed.

[5]Source: `https://www.ripe.net/manage-ips-and-asns/ipv6/ipv6-address-types/ipv6addresstypes.pdf`

8.1 Filters

To avoid announcing wrong prefixes learned by other ASes, we apply different filters on the external BGP sessions. We use 3 different patterns for the filters according to the relation we have with them. On the external BGP relation, we apply two filters : one for the prefixes we allow to announce to the AS, and a second one for the prefixes we allow to be announced by the AS.

- For our clients, we announce them all the prefixes we have learned. We allow them to announce their prefix and theoretically their client prefix. But we are in a circle configuration between ASes, we can not allow them to announce their clients.
- For our providers, we announce them our own prefix and our client prefixes. But we accept all the prefixes they announce to us.
- For our shared-cost peers, we announce them our own prefix and our client prefixes. But we accept all the prefixes they announce.

By applying these filters, we reduce the size of the BGP routing table.
For example, for our provider AS65002, we apply these filters:

```
Filters-in : All prefixes.
Filters-out : fde4:1::/32
neighbor fde4:2:1::2:bace prefix-list provider-65002-out out
[...]
ip prefix-list provider-65002-out permit fde4:1::/32
ip prefix-list provider-65002-out permit fde4:3::/32
ip prefix-list provider-65002-out permit fde4:9::/32
```

fde4:1::/32 is our own prefix. fde4:3::/32 and fde4:9::/32 are the prefixes of our two clients. Of course, the configuration files are automatically generated by the bgpd.mako and `config.json` files.

8.2 BGP Communities

A community can be applied on a defined route. A community is represented by 32 bits. We use communities on each direct route to external ASes. We distinguish 4 different communities in our AS:

- garbage for AS64512
- customer for AS65009 and AS65003
- provider for AS65002
- sharedCost for AS65005 and AS65004

The customer, provider and sharedCost communities are used to set the local-preferences. The highest local-preference is set for the customer (used for our client). Obviously we prefer the routes which go through our clients. We set a little bit lower local preference for the shared-cost relation. And the lowest local-preference is for our provider path. This way we only have to pay when this path is the only one to reach our destination. The garbage community is set to keep all the packets coming from AS64512 at router P0. This AS does not allow us to ping through AS64512 so we redirect all the packets from AS64512 to the localhost of P0. Redirecting those packets destroys them. But the P0 router can always reach AS64512. The default route goes through P0 so all the network can reach this AS and don't need his path. We create this special community for AS64512 after a discussion with Thomas who said that we can block AS64512.
After applying a command on the community, we have to declare an empty command for the packets which does not match the conditions. We apply a lowest priority level on the empty command, so it will check the condition before.

Conclusion

Setting up a secured and optimized network has now been made easier. Thanks to the configuration file, we do not have to worry about the different syntax of all the protocols, we just need to specify the parameters and the generator takes care of understanding it.

Testing is also intuitive and exhaustive. We are able to check the reachability and the routing between routers. It tests the reaction of the production network to failures and verifies the reachability of our BGP peers.

Concerning the security of our network we implemented different features : we used MD5 passwords over iBGP and eBGP sessions, we filtered OSPF and BGP packets that should not be received by peers, we filtered packets with private addresses following the RIPE document we found, we added protection against DDoS attacks and finally, we added log options for two types of dropped packets (the ones that might be DDoS packets and the ones using the types of private addresses described before).

The setting up advanced BGP policies has been enabled by the precise definition of the different relationships we have with our peers. We first had to set up filters to prevent our peers to advertise any prefix they want and thus, prevent us to broadcast unexpected prefixes. We also implemented route maps on the external connections to manage local-preferences on different links to match to real configuration.

The network configuration system we developed can now be reused by anyone to easily deploy its own internet service provider.

Appendix A. config.json example

The following config.json file example represents an architecture consisting in 2 routers (P0 and P1) connected to each-another on interface 0 and running OSPF and BGP. Router P1 has an active external BGP (eBGP) session with AS65005 on his interface P1-eth5.

The corresponding architecture is depicted on figure 3

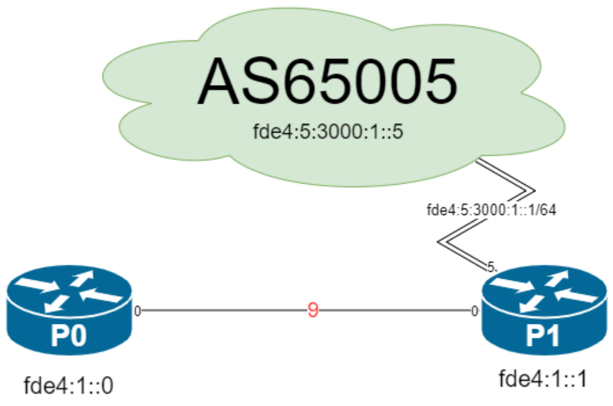

Figure 3: config.json example architecture

```
{
    "config_folder": "project_config",
    "as_number": 65001,
    "routers": [
        {
            "name": "P0",
            "id": 0,
            "ip": "fde4:1::",
            "subnet": "128",
            "interfaces": [
                {
                    "number": 0,
                    "connected-router": 1,
                    "cost": 9,
                    "hello-time": 10,
                    "dead-time": 40,
                    "instance-id": 0,
                    "area": "0.0.0.0",
                    "ospf_active": true
                }
            ],
            "bgp": {
                "active": true,
                "neighbors": [
                    {
                        "type": "internal",
                        "as_number": 65001,
                        "ip": "fde4:1::1",
                        "prefixes": [],
                        "MD5_password": "P0P1"
                    }
                ]
            }
        },
        {
            "name": "P1",
            "id": 1,
            "ip": "fde4:1::1",
            "subnet": "128",
            "interfaces": [
                {
                    "number": 0,
                    "connected-router": 0,
                    "cost": 9,
                    "hello-time": 10,
                    "dead-time": 40,
                    "instance-id": 0,
                    "area": "0.0.0.0",
                    "ospf_active": true
                }
                {
                    "number": 5,
                    "cost": 1,
                    "bgp_active": true,
                    "bgp_address": "fde4:5:3000:1::1/64",
                    "instance-id": 0
                }
            ],
            "bgp": {
                "active": true,
                "neighbors": [
                    {
                        "type": "internal",
                        "as_number": 65001,
                        "ip": "fde4:1::",
                        "prefixes": [],
                        "MD5_password": "P0P1"
                    }
                    {
                        "type": "external",
                        "as_number": 65005,
                        "ip": "fde4:5:3000:1::5",
                        "prefixes": ["fde4:1::/32"],
                        "interface-number": 5,
                        "relationship":"peer",
                        "community":"shareCost",
                        "input-whitelist": [],
                        "output-whitelist":["fde4:1::/32", "fde4:9::/32", "fde4:3::"],
                        "MD5_password": "ASes6500165005",
                        "ping-ips": ["fde4:5:3000:1::5"]
                    }
                ]
            }
        }
    ]
}
```