Université catholique de Louvain

LINGI2142

Computer networks: configuration and management

Multi-homed network

Authors: ***Group 3***
Romain Dizier
Quentin Gusbin
Thibault Jacques
Léonard Julémont
Nicolas Vrielynck

Supervisor:
O. Bonaventure
O. Tilmans
M. Chiesa

May 1, 2017

1 Introduction

The purpose of this project is to configure a multi-homed IPv6 network with a topology similar to the network of the Univeristé catholique de Louvain. A multi-homed network corresponds to a network with different ISPs that provide different prefixes. In this project, the challenge is to manage two prefixes and to be reachable via two prefixes, but also to configure the entire network and to handle failures. Finally, the users of the network should not change anything in their habit in order to use our network.

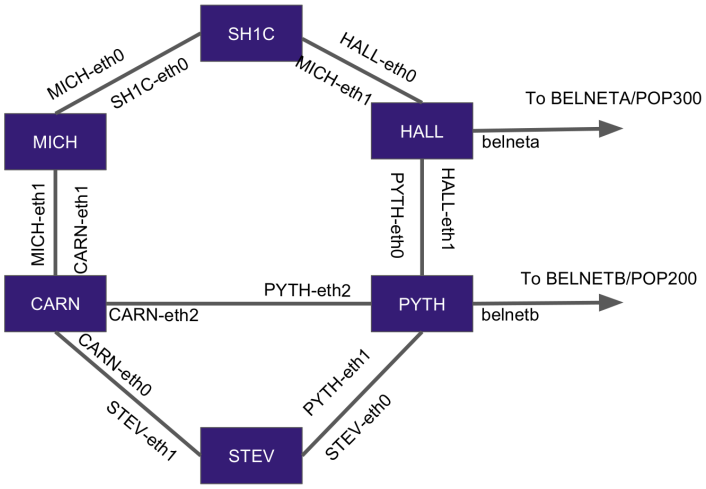
2 Topology of the network and addressing plan

This first section will focus on the topology of our network and the addressing plan that we designed.

2.1 Topology of the network

For this project, we started from the provided network topology, illustrated by the Figure 1.

Figure 1: Schema of the topology of the given network



The first choice that had to be made was to define the positions of our data centers in this network. We decided to have two data centers linked to different routers in the purpose to have redundancy and to have a more robust network. However, we can consider that the data centers are risk areas. Indeed, every services will be placed in it. We decided to place the data centers on the routers linked to the providers : HALL and PYTH. Thanks to that, the traffic between the data centers and the outside of our network will not go through other routers of our network and will not use the bandwidth of the users.

The second choice that has been made was to move the link PYTH-CARN between PYTH and HALL. There are two reasons that justify this change. Firstly, it allows the source routing between PYTH and HALL to work even if the primary link fails (see section 3 for more details). Secondly, it makes it possible to avoid the traffic between the data centers of having to go around the network in case of a link failure between PYTH and HALL. In a real scenario, we can imagine that there is a large amount of data between data centers, in particular to allow the synchronization.

Note that we added a monitoring server to manage our network linked to the router PYTH.

2.2 Addressing plan

Concerning the addressing plan, we receive two IPv6 prefixes of 48 bits : fd00:300:3::/48 and fd00:200:3::/48. We decided to use the stateless addressing auto configuration, so we must keep the last 64 bits for this pur-

pose. Finally we have 16 bits to organize and there are multiple possibilities. As explained in the document "Preparing an IPv6 addressing plan" [9], starting with the site then the usage facilitates the security policies and has only a small impact on the routing performances because nowadays routers are able to support huge routing tables. Thus the first 4 bits are used for the site, then the 4 next for the usage and finally the 8 bits left are used for the location. We use 4 bits for the site and for the usage because there are already 6 sites and 7 uses, and we want to have the possibility to add more in the future.

Addressing plan fd00:X00:3:SULL::/64

- [0, 47]: Prefix
- [48, 52]: Site (S) :
  - 0 : Louvain-la-Neuve
  - 1 : Woluwe
  - 2 : Tournai
  - 3 : Mons
  - 4 : Saint-Gilles
  - 5 : Namur
- [53, 56]: Usage (U) :
  - 0 : Infrastructure
  - 1 : Service
  - 2 : Staff
  - 3 : Student
  - 4 : Guest
  - 5 : Voice IP
  - 6 : Camera video
- [57,64]Location (LL)
- [65,128]: Interface ID

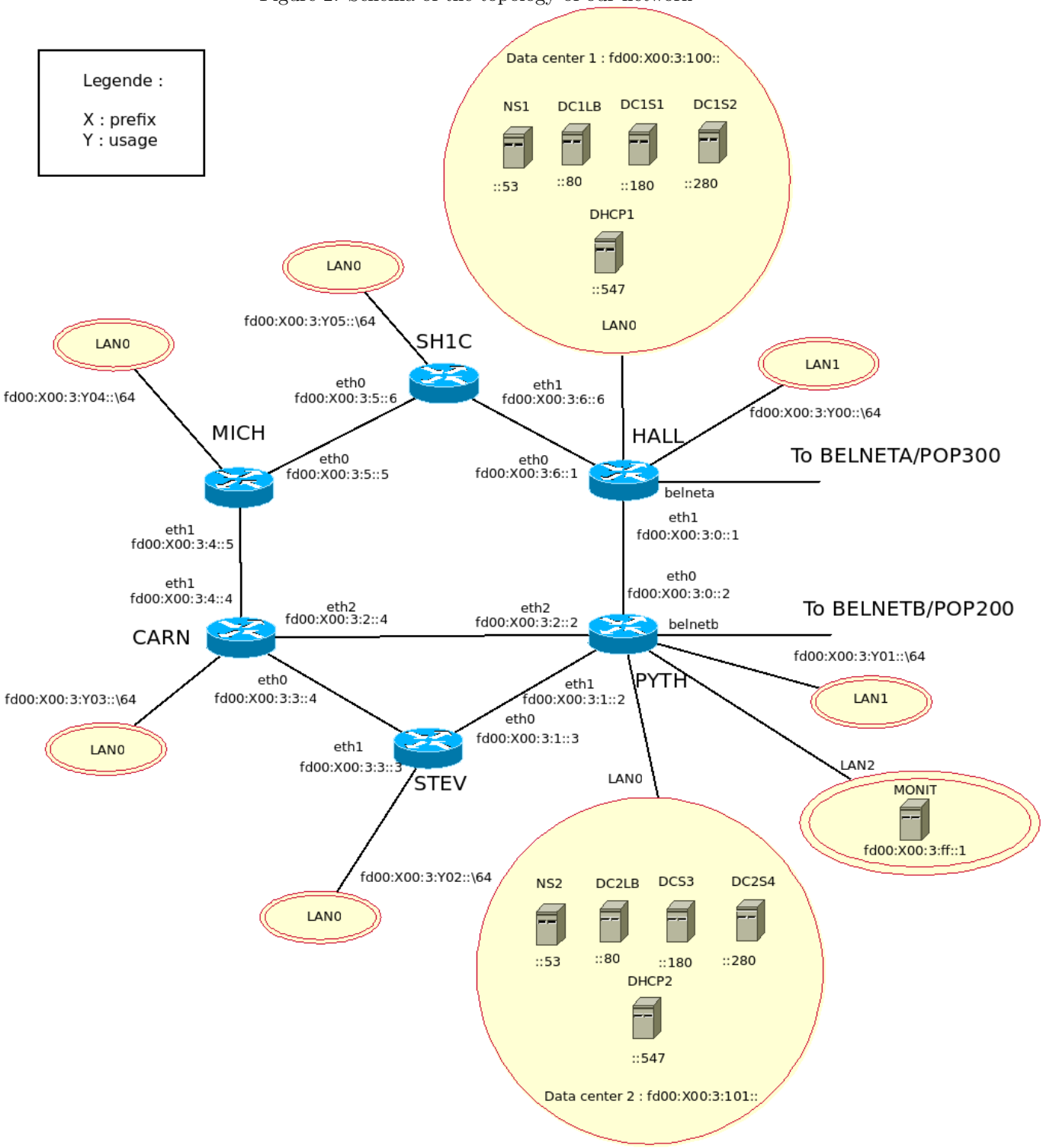
Moreover, to describe specific address :

- the link between the routers : fd00:X00:3:00Z::N where Z is the number of the link and N the number of the router
- service : fd00:X00:3:010D::NP where D is the number of the data center, P the port of the service and N gives the number of the service instance if there are more than one instance of a service type.

2.3 Prefix usage

The machine in our network are using the 300 prefix by default, unless they need to send a packet towards a address having the 200 prefix or if the prefix 300 becomes unavailable. However, each machine is reachable using both prefixes. Based on some business elements, it might be more interesting to use both prefixes at the same time. It depends for example on the way the providers charge us to use their services.

Figure 2: Schema of the topology of our network



3 Routing *Léonard Julémont*

This section presents the important elements about the routing that has been implemented in our network. BIRD[1] is the routing daemon that is used on the routers.

3.1 Inter-domain routing

The Border Gateway Protocol has been used as the inter-domain routing protocol. We have three BGP sessions running, one on HALL connected to *belneta* (POP300) and another one on PYTH connected to *belnetb* (POP200). The third BGP session is also running on PYTH and connects our network to the group 1.

ISPs

The configuration of the BGP sessions with our two ISPs are quiet simple. We accept only accept the default route and we advertise them only one */52* prefix. The prefix depends on the one that the ISP provides us. As an example, *belneta* allocates us the prefix *fd00:300:3::/48* and we advertise *fd00:300:3:0::/52*. The extra *0* corresponds to the number that we chose for the Louvain-la-Neuve site has explained in the section 2 *Topology of the network and addressing plan.*

We do not want to announce the entire routing table over BGP because it would give the ISP or another BGP peer the entire topology of our network. Moreover, the ISP could disable the BGP session as long as we advertise too much routes. Thus, we decided to export only one prefix per BGP peer, which is the smallest one corresponding the Louvain-la-Neuve site.

In order to force a certain route to be exported over BGP with BIRD, one can use the "static" protocol in BIRD, as presented in the code sample 1. As explained in BIRD's wiki [1], the route to be exported is presented as an unreachable route, but it is not a problem for BGP. The static protocol is then used in the BGP protocol configuration as the exported routes, thus the routes that are exported from the global routing table to the BGP tables.

```
protocol static static_default_bgp_out{
    import all;
    route fd00:300:3::/52 reject;
}

protocol bgp pop300{
    export where proto = "static_default_bgp_out";
    ...
}
```

Listing 1: Exporting specific route over BGP with BIRD

BGP peering with Group 1

The BGP session connecting our network to the group 1 is running on PYTH and the configuration is similar to the one used for our ISPs. We advertise them one route which is *fd00:200:3::/52*. We decided to have only one connection with them because the probability of a failure on the physical link is low and the cost of another link is high. But if something were to happen to the link, the traffic towards their network would simply be transported over the internet.

The only difference compared to our ISPs is that we do not want to accept all the routes they could declare. As an example, if they advertise the default route *::/0*, we should not accept it because we do not want our traffic to go through their network in order to reach the internet. It would allow them to scan our traffic. Thus, we use a filter to only accept the routes having the prefix *fd00:200:1:1000::/52*.

[1] `http://bird.network.cz`

3.2 Intra-domain routing

The intra-domain routing protocol used in our network is OSPF. It is activated on all the **eth** interfaces of each router and disabled on **lan** interfaces, by using the *stub 1* command, to make sure that the routers will not accept OSPF messages coming from a host.

On the two border routers, there is one more specific configuration that needs to be done in order to transfer the default route *::/0* received over BGP by the ISPs to OSPF, and thus to all the routers of the network. Once again, one can use a "static" protocol in BIRD to define the route and then use it in the OSPF protocol configuration as the export prototype. Then code sample 2 shows the configuration on HALL.

```
protocol static static_ospf {
    import all;
    route ::/0 via fd00:300::b;
}
protocol ospf {
    export where proto = "static_ospf";
    ...
}
```

Listing 2: Propagating default route to OSPF with BIRD

3.3 Specific rules

The particularity of our network is to be multi-homed. It means that we are provided with two different prefixes but as one can imagine, the providers do not accept packets with a source address that does not correspond to the address that it assigned to our network. But, we mentioned in the previous section that a default route is defined on PYTH and HALL towards the ISP. It means that every packet that does not match any other route in the routing table will be sent to the provider, even if the packet has a source address that does not match the prefix assigned by the provider.

In order to solve this problem, we decided to use *ip rules* on PYTH and HALL. When a packet arrives at the router, it first checks which *ip rule* matches the packet. Each rule gives the action that should be performed with the packet. For example, the rule can mention the routing table that should be used to forward the packet.

In our case, we need one new routing table and three new *ip rules*. The code sample 3 shows the configuration that is used on PYTH (and we do the same on HALL for the 200 prefix). The two first commands create rules matching the traffic having a source address provided by the other ISP (connected to HALL) and having as destination an address in our network. For this traffic, we simply specify that the table "main" should be used in order to route the traffic, because this traffic is not leaving our network and can then be forwarded by PYTH, using the primary link. The third command creates a new routing table "10" and adds to it a default route towards the router HALL. The last command creates a last rule matching any traffic having an address using the 300 prefix and tells that the traffic must be routed using the routing table "10". Note that this last rule will be evaluated after the first two rules because the preference assigned to it is higher. It implies that this third rule will only be applied to traffic using the 300 prefix as source address and having a destination address that does not correspond to our network, which is exactly what we want.

```
> ip -6 rule add from fd00:300:3::/48 to fd00:200:3::/48 pref 1000 table main
> ip -6 rule add from fd00:300:3::/48 to fd00:300:3::/48 pref 1000 table main
> ip -6 route add ::/0 via fd00:200:3:1::1 table 10
> ip -6 rule add from fd00:300:3::/48 pref 2000 table 10
```

Listing 3: IP rule to forward packet based on source address

There are two elements that deserve to be mentioned. Firstly, one might ask : what would happen if a provider is down ? The traffic using the prefix it provides us would not be able to leave the network. Fortunately, the monitoring server would detect the ISP failure and would ensure that the host on the network are no more using the that prefix (for further details see section 5). Thus no more traffic would be redirected from one router to the other via the new *ip rule*.

The second element is about the fact that this solution works because the two routers are directly connected with a physical link. If it was not the case, we could have loops in the network, with routers sending packets back and forth. In that case, we would need to use tunnelling. A new header would then be added on packets to tell the routers on the path that the packets have to be forwarded to a certain router.

Finally, if the primary link between HALL and PYTH fails, the default route defined by the third command would not be usable anymore because the specified gateway address is no more directly connected to the router. In order to handle this case, there is a script (backup_link_switcher.sh) running on PYTH and HALL that checks the state of the primary link. If it is down, it will change the default route of the table 10, to make it use the backup link. It will switch back to the primary link when it is up again.

3.4 VLANs

On each router, we decided to have a LAN to have hosts everywhere in the network. Because we have no access to the layer 2, we have no way to authenticate a host and assign the right address based on its type. In order to use the different use types, we created VLANs. There are two VLANs on the LAN of each router, one with use type 2 (staff) and one with use type 3 (student). In order to make it easy for maintenance, the VLANs on a LAN use the same "location".

3.5 Monitoring server

The monitoring server is placed on a LAN on the router PYTH, has the use type "0" (infrastructure) and is on the special location "ff", which gives the address fd00:200:3:ff::1. This server is constantly running two scripts. The first one checks the status of BGP sessions on PYTH and HALL with the ISPs. The second one checks the availability of the different DNSs and load balancers. For further details, see section 5 *Routing and End-user reliability*.

3.6 Configuration and Extensibility

The BIRD configuration files for routers and ip configuration (start files) for routers, host and services are created automatically based on .json files containing all the information about them. It allows to easily change the configuration on all the routers but also the change the "location" of a specific LAN for example. Moreover, the extension of the network is quiet easy. One can simply add the machine in the *project_topo* file and then add the necessary information in the corresponding .json configuration file. When relaunching the network, all the files will be created and the machine will be part of the network.

The files are named router_config_creation.py, services_config_creation.py and host_config_creation.py, and the .json files are router_configuration.json, services_configuration.json and host_configuration.json.

3.7 Monitoring

In order to monitor the routing in the network, BIRD logs are activated on each router and is stored in */etc/log/bird_log*. The monitoring server also logs information about the availability of prefixes, DNSs and load balancers (/etc/log/isp_status_log and /etc/log/services_status_log).

3.8 Testing

In order to test the routing of our network, one must connect to the monitoring server (*sudo ./connect_to MONIT*). The different test are located in the folder *routing_test*. There are two types of test.

The first type corresponds to the tests that simply check the network. For example, host_connectivity.sh which checks the connectivity from one host per router to all the routers, DNSs, webservers, ISPs and Google. The same test exists for routers but is not trying to reach the internet because it does not make sense for a router to ping a machine outside the network (except BGP peers). There are also tests to check the reachability of ISPs and the prefixes that we export (and the same test for the BGP peering with group 1).

The second category regroups the tests that simulate a failure on the network and then check that everything is still working as excepted. For example, tests are made to simulate an ISP failure (and thus a prefix becomes unusable), link failure and router failure.

4 End-user *Thibault Jacques*

In this section, we will speak about the end-user management. The purpose of this part is that a new user on the network will have a complete configuration ready without configuring manually. It involves receiving multiple IPv6 addresses, information about the DNS servers and the default routes.

4.1 Choice of infrastructure

We had to choose between three possible infrastructures:

- Stateless Address Autoconfiguration (SLAAC) thanks to the router advertisements.
- DHCPv6 Stateless to advertise the DNS and SLAAC to manage the addressing.
- DHCPv6 Stateful to manage all the information thanks to DHCPv6.

We decided to combine two infrastructures: the stateless address autoconfiguration and the DHCPv6 Stateless to advertise the DNS and SLAAC to manage the addressing. Indeed, it doesn't exist an infrastructure supported by every user. Indeed, Windows doesn't support RDNSS extension of router advertisement that allows announcing the DNS of the network. On the other hand, Android doesn't support DHCPv6. The purpose is to deploy a system supported by the greatest number of users. This is why we decide to use the two infrastructures together and not the DHCPv6 Stateful not supported by Android. Moreover, the management of the address and the prefix with DHCPv6 only can cause problem because most of the DHCPv6 clients only ask for one non-temporary address but in our case, we want that the users have one address with each prefix.[8] So, in this case, we are dependent on the DHCPv6 client implementation.

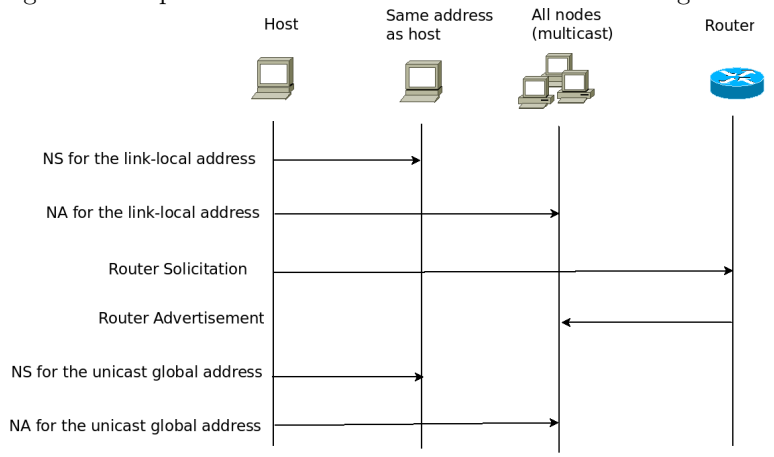
4.2 Stateless Address autoconfiguration (SLAAC) thanks to the router advertisement

For the stateless address configuration, we deployed on each router the daemon radvd that implements the Neighbor Discover Protocol[2]. We can summarize how it works when a new host is added to a LAN of our network.

When a host is connected to the network, it will generate a link-local address and will send a unicast Neighbor Solicitation to now if it is available. If there is no answer, the host will send a multicast Neighbor advertisement to advertise its address because it is the only host on the LAN with this address. After that, it sends a multicast Router Solicitation to locate the router of the network. This router will answer with a multicast Router Advertisement with the information about the network. From this information, the host can create two global unicast addresses from the two received prefix. It will send a unicast Neighbor Solicitation to know if the address created is not used by someone. If it has no answer, it will send a multicast Neighbor advertisement of its own address on the LAN.

[2] `https://tools.ietf.org/html/rfc2461`

Figure 3: Simple schema of the stateless address autoconfiguration

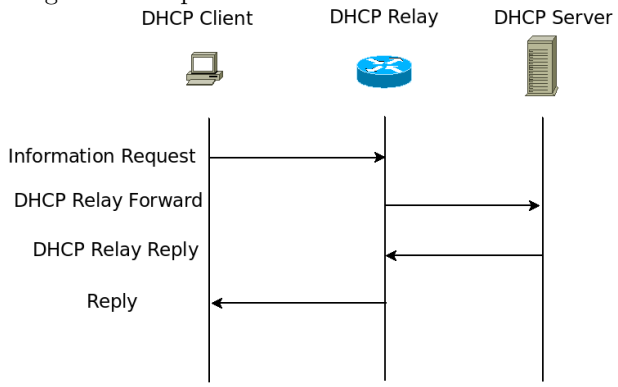


In the Router Advertisement, the flag M (Managed) is put to 0 to advertise the host to create its own address and the flag O (Other) is put to 1 to to advertise the host to use DHCPv6 to have the other information as the DNS server but the host can always use the RDNSS extension to have this information. Indeed, we use the RDNSS extension to advertise the DNS servers of our network. However, the radvd daemon allows to advertise only 3 DNS addresses but we have two DNS with two addresses. So, we advertise the two DNS thanks to the RDNSS extension but with only one prefix, in our case, the prefix is 200. From the point of view of the hosts, we decide to simulate some hosts using the stateless address autoconfiguration only thanks to the daemon rdnssd that allows managing the RDNSS extension of the router advertisement.

4.3 DHCPv6 Stateless

We installed two DHCPv6 servers, one on each data center. This DHCPv6 servers will only share the DNS servers information to the users. To be reachable for the users, we installed on each router a DHCPv6 relay agent to relay the DHCP messages between the users and the DHCPv6 servers. The used daemons come from the Internet Systems Consortium[5] (ISC). The DHCPv6 servers advertise the two DNS servers with the two prefixes: a total of 4 addresses.

Figure 4: Simple schema of the DHCPv6 stateless



The DHCP client will send a Information Request to the DHCP server. The relay agent on the router will encapsulates the DHCP messages in a DHCP Relay Forward and forwards to the other routers to finish at the DHCP server. The server will answer to the DHCP relay with a DHCP Relay Reply. Finally, the DHCP relay will decapsulate the message to send the Reply to the DHCP client.

4.4 DNS

We deployed two DNS servers[4] in our data centers. These two DNS servers are built thanks to the bind9 daemon. In our case, our DNS must be the master of the zone group3.ingi. We decided to use two independent DNS that will be updated by our monitor server when the topology of our network change. For example, if one provider is down, we don't have to advertise addresses with the prefix of this provider.

Firstly, we will describe the zone managed by our DNS :

- group3.ingi : this zone contains the information about the DNS server and the website accesses.
- router.group3.ingi : this zone contains the information about the routers in our network
- service.group3.ingi : this zone contains every services available on the network

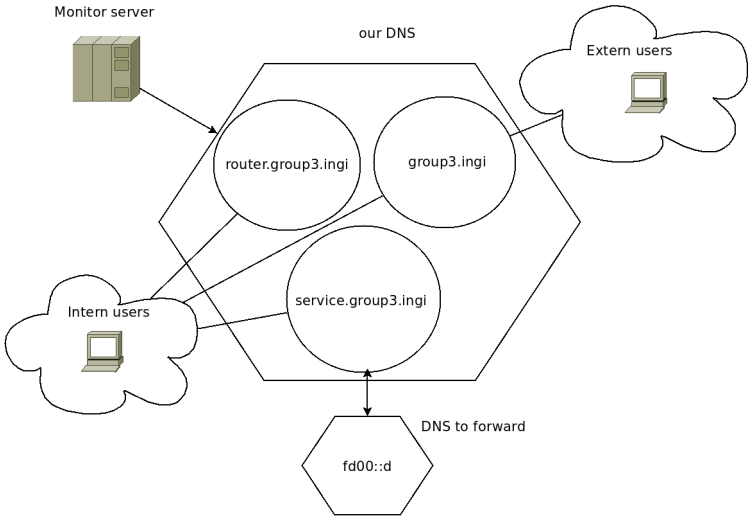
From this information, we must decide the configuration of our DNS. Indeed, every user can't access to every zones of our DNS. Moreover, the DNS can't be used by every user in the same way. To do that, we define two types of user: the internal users, the users inside our network and the external users, the users outside our network.

Now, we can describe the abilities of the two kinds of users:

- For the internal users, the DNS servers
  - Answer to the query about every zones managed by the DNS servers
  - Do the forward recursion to the DNS server fd00::d
- For the external users, the DNS
  - Answer to the query about the "public" zone group3.ingi.
  - Don't answer to other queries.
  - Don't do the forward recursion.

We can also notice that the query will be forwarded to the DNS fd00::d as asked. Moreover, the transfer of the zone group3.ingi is available for the address fd00::/16. Finally, every zones have the reverse translation accessible with the same rules that the corresponding zone.

Figure 5: DNS configuration illustration



4.4.1 Values of the SOA resource record.

- The Time-To-Live (TTL) defines the duration that the record may be cached. This value is a tradeoff between the stability of our network and flexibility. Indeed, if we use a little value, we can modify often the addresses advertised by the DNS but on the other hand, we will increase the traffic to our DNS server. In effect, the zones will be cached during less time and the users will send queries more often. In our case, we put a value of 3 hours because our infrastructure must no change a lot over time.
- The SOA Refresh and Retry are values that affect the interval at witch a secondary name server checks for new zone at the primary zone. In our case, we use two different DNS server but we transfer our zones. So, it can exist some slave servers link to us. We decide to use standard value, SOA Refresh : 2 hours and SOA Retry : 15 minutes.
- The SOA expire is the value of the timer to tell to the secondary secondary server how long it should answer queries about our zones when it cannot contact our servers. As previous, we don't have linked servers. We decide to use standard value of 2 weeks.
- The SOA minimum specifies the time of cache a negative response of a DNS answer. We uses a little value to ensure because the status of the answer can change quickly. Thus, the value is 30 minutes.

4.5 Configurations scripts

Every files about the end user management is in the directory `end_user_management` at the root of the project.

4.5.1 Configuration radvd

The configuration of the daemon radvd for each router is written during the launching of network thanks to the `router_config_creation.py`. Each configuration file are created in the folder `radvd` of the router corresponding. Moreover, the script to update the radvd configuration file is in the folder `radvd_update` is distributed to the same folder that the radvd configuration file.

4.5.2 Configuration Bind9

The configuration of the Bind9 are also written during the launching of the network thanks to the the file `deploy_bind_conf.sh` in the folder `bind`. This script will create the different files of the configuration and will copy them to folder `bind` of the DNS server corresponding.

4.5.3 Configuration DHCPv6 server

The configuration files of the DHCPv6 client are distributed to the folder `dhcp` of the two DHCP servers.

4.5.4 Configuration DHCPv6 client

The configuration files of the DHCPv6 client are distributed to the folder `dhcp` of the different hosts.

4.5.5 Configuration of rdnssd

The configuration of the rdnssd daemon are distributed to the folder `rdnssd` of the different hosts.

4.6 Tests

One test `end_user_test.sh` is present in the folder `end_user_test/`. Important remark, this test must be launched from this folder. This test will test :

- Ping every service from every host
- dig request to our DNS servers for every router from every host
- dig request to our DNS servers for every service from every host
- dig reverse request to our DNS servers for every service from every host
- dig request to our DNS servers for google.com from every host
- DHCPv6 requests from every host

4.7 Robustness of the end user management

Firstly, about the stateless address autoconfiguration, the connection between the host and the router is direct. So if the link is down, nothing can work. From the point of view of the DHCPv6, the network is robust. Indeed, the request will be sent to the two DHCPv6 servers. So, it's no a problem if a link is down or a router is down thanks our cyclic architecture. Regarding the DNS, every host have the addresses of the two DNS. If one DNS doesn't work, the host can use the other. Unfortunately, a delay will be visible but the network will always work.

5 Routing and End-user reliability *Thibault Jacques and Léonard Julémont*

Some configurations ensuring the reliability of the network involve both the routing and the end-user management, which is the purpose of this section.

5.1 Loss of a prefix

The loss of a prefix can happen in multiple ways : link failure, ISP failure or the router connected to the provider could fail. The monitoring server is constantly checking the status of the BGP session with our ISPs. If a BGP session is no more established, the prefix that is assigned to us by the corresponding provider can not be used anymore. The monitoring server is then performing two different actions.

Firstly, it launches a script on each router that will update the RADVD configuration, and then sends a SIGHUP signal to restart RADVD on each router. The new configuration makes DHCP to advertise the unavailable prefix with a preferred lifetime of 0. The hosts will then choose the other prefix which is still available, when sending a packet.

The second action by the monitoring server is to launch an update script on DNSs, which makes them stop advertising addresses with the prefix that is no more usable. The hosts outside the network will then only use the available prefix, thus all the services remain available.

5.2 Failure of a service

The services that are running in our network could also fail. If a DNS fails, we want the other DNS to stop advertising the address of the failed one and to continue deliver the service to all DNS requests. If a load balancer fails, the DNSs need to be alerted and to stop advertising its addresses. The website remains accessible because the DNSs will provide the addresses of the second load balancer.

The monitoring server runs a script to check the status of the services present in our network. When detecting a failure, it runs an update script on the DNSs and send SIGHUP signals to restart the DNS daemons. The DNSs will then advertise reachable addresses.

6 Security *Romain Dizier*

The security part of the network essentially consist of several firewalls (one for each router), that contains the different rules to apply for a particular traffic. While some rules are more generic and thus active on several routers, others were specifically created for some distinctive cases.

6.1 Choice of tools

Since we don't have to take into account neither IPv4 nor IPv4 tunneling, we will only use ip6tables to create the different rules for our network. The different rules are stored in scripts, that needs to be executed for each router once the network has been launched and initialized. To make these rules persistent (i.e still active when the network reboots), we decided to use the package *ip6tables-persistent* which takes over the automatic loading of the saved ip6tables rules. This method is rather efficient and better suited for our case than a big single script that is loaded on every reboot of the network.

In addition to this package, we also use a log system to register all the dropped packets and several information about them. We first tried to use the daemon *ulogd2* [7] but due to its numerous dependencies and extensive configuration we switched to the already implemented log rule filters of ip6tables. These rules are known as "non-terminating target", meaning that the rule traversal continues at the next target (thus needing two separate rules with the same matching criteria, one using the target LOG and then the other using the target DROP). These log files can then be accessed using the *syslogd* tool.

6.2 Configuration scripts

Each firewall rules are stored in a script corresponding to the router to protect, in addition to a file that stores the entire set of rules which can then be loaded when the network reboots. These files are :

- *CARN.sh* and *carn.fw*, the script and storage file corresponding to the router CARN.
- *HALL.sh* and *hall.fw*, the script and storage file corresponding to the edge router HALL.
- *MICH.sh* and *mich.fw*, the script and storage file corresponding to the router MICH.
- *PYTH.sh* and *pyth.fw*, the script and storage file corresponding to the edge router PYTH.
- *SH1C.sh* and *sh1c.fw*, the script and storage file corresponding to the router SH1C.
- *STEV.sh* and *stev.fw*, the script and storage file corresponding to the router STEV.

Once the network has been launched, the scripts can be started by first connecting to the related router (using the command *sudo./connect_to.sh X*, with X being the name of the router), then opening the folder */vagrant/ip6tables* and loading the last stored rules (using the command *ip6tables-restore < /vagrant/ip6tables/X* with X being a storage file with the extension *.fw*).
To modify the rules of the firewalls, the new commands can first be added to the related script of the router, then following the last paragraph connect to the given router, moving to the folder *ip6tables*, executing the newly modified script, and finally store these new rules in the corresponding *.fw* file using the command *ip6tables-save > /vagrant/ip6tables/X* with X being a storage file with the extension *.fw*.

6.3 Types of rule

Several types of rules are used in the firewalls [2], each of them covering specific cases of possible misuse or functionality that we want to prevent or allow. Since we use *ip6tables*, all the rule we created follow the syntax *ip6tables C P*, with C corresponding to one or several commands (such as appending a rule, inserting a rule, ...) and P corresponding to parameters (such as the protocol, the source, the destination, and so on). The basis of the firewalls we decided to use is a *whitelist*, where everything is prohibited by default and only the allowed traffics can occur.Furthermore, we chose to implement *stateful* firewalls, meaning that they keep in memory the states of the different connections going through them. This allows to enable these *ESTABLISHED* or *RELATED* connections to last, and only packets matching one of these active connections is allowed to pass.
We also added the possibility to accept new connections coming from the trusted routers of our networks, based on their IP addresses. It is important to notice at this point that since this network is dual-homed (i.e two providers), several of these rules are duplicated based on the prefix of the addresses.

Another important rule is adding the possibility of *loopback* for a given router. This allows the routers to send traffic back to themselves, and is fairly easy to set up using the incoming interface and outgoing interface (*-i* and *-o* respectively). Accepting to receive and send back *Router Solicitation*, *Router Advertisement* messages as well as *Neighbor Solicitation* and *Neighbor Advertisement* is important to ensure the proper functioning of the network. This is done by accepting icmpv6-type packets corresponding to the correct type (133/134 for RS/RA and 135/136 for NS/NA). But routers should only accept theses messages from other routers inside of the network ! So we have to *DROP* all RS packets coming from students connected to the network, based on their source where the bits allocated to the usage is equal to "student" (in our case, 3).

To allow several tests to be made on our network, the echo requests (or ping) are enabled on every six routers. This is done by again checking the type of icmpv6 packets, and accepting the ones that matches 128. One small exception is made for the edge routers *HALL* and *PYTH*, a rate limit of 30 ping per minute has been set in order to prevent abusive behaviours. Several services have also been authorized, such as incoming SSH (using port 22) connections or outgoing DNS (port 53) and HTTP (port 80) connections. Finally, in order to protect sensitive information, the traffic related to the security cameras has been prohibited from leaving the network (all packets with an IP address in which the usage bits is set to 6 as source is able to leave the network by the routers *HALL* and *PYTH*).

Only a part of the firewall rules are resumed in this section of the report, the scripts and ip6tables-save files contain the exhaustive list.

6.4 Tests

The tests for the security of the network relied on the previously stated cases (ping the different services from the hosts, send sensitive packets such as the one related to the cameras and verify that they are not forwarded out of the network, ...) to ensure that no functionality was blocked by an over-restrictive rule while the prime goal of security is achieved.

6.5 Possible improvements

Some possible improvements would be to implement the log daemon *ulogd2* to replace the basic log function from ip6tables, which offers a lot more modules and options (record the logs in a database on top of the local files, numerous data to store, ...). To push things even further, the package *nulog2* offers a graphical interface as well as several tools that can be applied to the data collected from ulogd2 to perform deeper data analysis.

7 Services *Nicolas Vrielynck*

For the service part of the network, We implemented two data center. Both of the data center are composed of a load balancer and 3 web servers. We also implemented ssh server on all routers and on the load balancer and web servers of the data center.

7.1 Data center

The two data center are each of them composed of 3 web servers and one load balancer. In this section we will discuss the localisation, the configuration and the implementation of the data centers.

7.1.1 Localisation

The first data center is located in HALL and the second in PYTH. Those two end points are the to endpoint of BelnetA and BelnetB. We decided to put our data center the closer to the provider to reduce the end to end latency. With this configuration we don't need a redundancy link between SETV and PYTH like in the UCL topology because our two data centers are already linked by the link existing between PYTH and HALL. What we could do instead is adding a second link between PYTH and HALL to be sure that the two data centers are always connected through the shortest path even if this link fails.

7.1.2 Web Server

We decided to install Apache2 to handle the http requests. In order to do so on different node of the network, we had to specify for each a proper and unique configuration file in /etc/apache2. We also had to create folder (/etc/var/run/apache2, /etc/var/lock/apache2 and etc/var/log/apache2) which is unique for each of the webserver. This is were the lock, log, etc will be located and this will allow to have multiple instance running on the same VM (one on each of the servers).

In a typical static web balancing problem, each servers would return the same page. Here, in order to see that the load balancer is properly working, we decided to put a different html static page on each of the servers. Those static page specify which server (1,2,3) on which data center (1,2) is being called. This html page is located in /etc/www/html/index.html and is unique for each web server.

This html page is like so (X = 1-2) (Y = 1-2-3):

$$< h1 > DataCenterX < /h1 >$$

$$< h2 > ServerY < /h2 >$$

All those unique folder and file required for the web server are located in lingi2142/project_cfg/Webserver_name in our project folder. This will allow the web server to have unique folder and file when the project is build using create_network.sh script.

7.1.3 Load Balancer

We decided to isntall HAProxy for the load balancing between the servers. This will allow us to change the balancing technique as we whish. For now we are using the classical Round Robin. This is the simplest balancing technique and will allow us to see clearly that the load balancing is working as expected. For the future HAProxy handle also static round robin, least connection, etc...

In order to have 2 instance of HAProxy running on the same VM, we have to specify for each of the load balancer a unique folder where to store the haproxy pid. This folder is located at /etc/var/run/ and the pid is haproxy-private.pid.

Because we are using two ip prefixes (200 and 300), We decided to have 2 different listener. The loadbalancer will listen on each of the prefixes and will connect to the server using the same one. (see lingi2142/project$_cfg/DC1LB/haproxy/haproxy.cfg$)

7.1.4 Set up

First, Apache2 and HAProxy need to be installed on the VM. Either by adding it in the build script for the VM or by launching the script webservers.sh on the root of the VM. This will download and install Apache2, HAProxy and Curl (for testing purpose)

Then, when the create_network.sh script is launched, each web servers (DC1S1, DC1S2, DC1S3, DC2S1, DC2S2 and DC2S3) will automaticly launch the apache2 service with the command "apachectl start". This will launch apache2 with the configuration file located on /etc/apache2/apache2.conf.

The load balancer need to be start manualy after the topolgy was created. We need to wait that the topology is done before launching the HAProxy load balancer because they will check that the servers are up and will thus need a link up and running between the LB and the web servers. Once the topolgy is done we just need to launch the script $start_haproxy.sh that will launch the command 'haproxy -f /etc/haproxy/haproxy.cfg -D' on each of the loadbalancer. This launch the haproxy loadbalancer using the configuration file specified.

7.1.5 Test

In order to test that the loadbalancer is working properly on each of the datacenter, you just need to run the sciprt 'test_datacenters.sh'. This will call 3 times each datacenter on each prefixes. The expected response is 2 time 6 unique HTML pages (like described above in the section web servers). Because we contact each data center on both prefixes, each datacenters will response 2 times with the 3 pages. Making a total of 12 pages.

7.2 Server SSH

We decide to use Open-ssh has a ssh server for all the routers, the load balancer and the webservers. All thoses services and routers launch their ssh server at the start of the network with the cmd '/usr/sbin/sshd'. We need then to launch the script 'configure_ssh_monitor.sh' that will generate a id key on the monitor server and add it to the authorized key file for the services and routers. Then, once logged in the monitoring server (MONIT), one just need to launch the script 'ssh_connect_to' with the ipv6 addr of the service or router he wants to connect as argument. This will connect to host to the ip address using ssh with the id created during the 'configure_ssh_monitor.sh' script.

8 Quality of service *Quentin Gusbin*

8.1 Choice of software

The quality of service was setup with two software: tc and ip6tables. Tc (Traffic control) allows us to create the whole qos tree and ip6table to mark each packet depending on the different rules. Qos can also be done only using tc without ip6tables. In this case the filtering of the packet is done with tc filter. As explained in [6] this last technique is not optimal because the packet has to be analyzed twice : once in ip6table and once tc. This should not change the performances too much in our case because the qos tree is quite small and the rules are not very complicated. But in case the tree gets larger later on or the rules more complex to handle we chose to use ip6table to improve future scaling.

For testing we used iperf3. It creates traffic on the lines/routers so that we can verify that qos rules were respected in case of congestion.

The qos tree is installed on every interface of every router.

8.2 Qos tree

The root qdisc used is the classfull disc htb. Base on [3] we chose to use htb because it is an improved version of cbq. "HTB works just like CBQ but does not resort to idle time calculations to shape." It provides a way to shape the traffic based on packet characteristics.

The tree was inspired from [6]. However in this article they created a qos tree for a home network so we adapted it for a enterprise/ university network.

The first class(1:1) under the root htb qdisc is a class to simulated that the link has a maximum bandwidth of 80Mbits/seconds has requests in the project characteristics.

Under the first class(1:1) we have 5 different classes. Since each one of those classes is under the same unique class(1:1), each class can borrow bandwidth from each other if there is no congestion. In case of congestion the classes with priority will be able to use more bandwidth while classes with least priority will be limited.

The first priority class is (1:10). This class will contain all the network related packets (hello from ospf, dns, dhcp). The traffic has the highest priority to ensure that the network works well(if a hello package is not deliver the network think that a link is broken). In case of congestion this class is guaranteed 8Mbit/second bandwidth. This type of traffic should never need more that 1/10 of the overall bandwidth because it is not very heavy and it is processed first since it has the first priority). Since this class should experience congestion we used a pfifo as qdisc for this class in the tree as it is much faster to process packets than sfq.

The second priority class is VoIP. It is very sensitive to latency so it has a higher priority than most of the traffic(except network traffic). We allocated a max bandwidth of 8MBits/second for voice over ip. Voice is not too heavy and this class has the second priority so we should never need more than that. Since this class should experience congestion we used a pfifo as qdisc for this class in the tree as it is much faster to process packets than sfq.

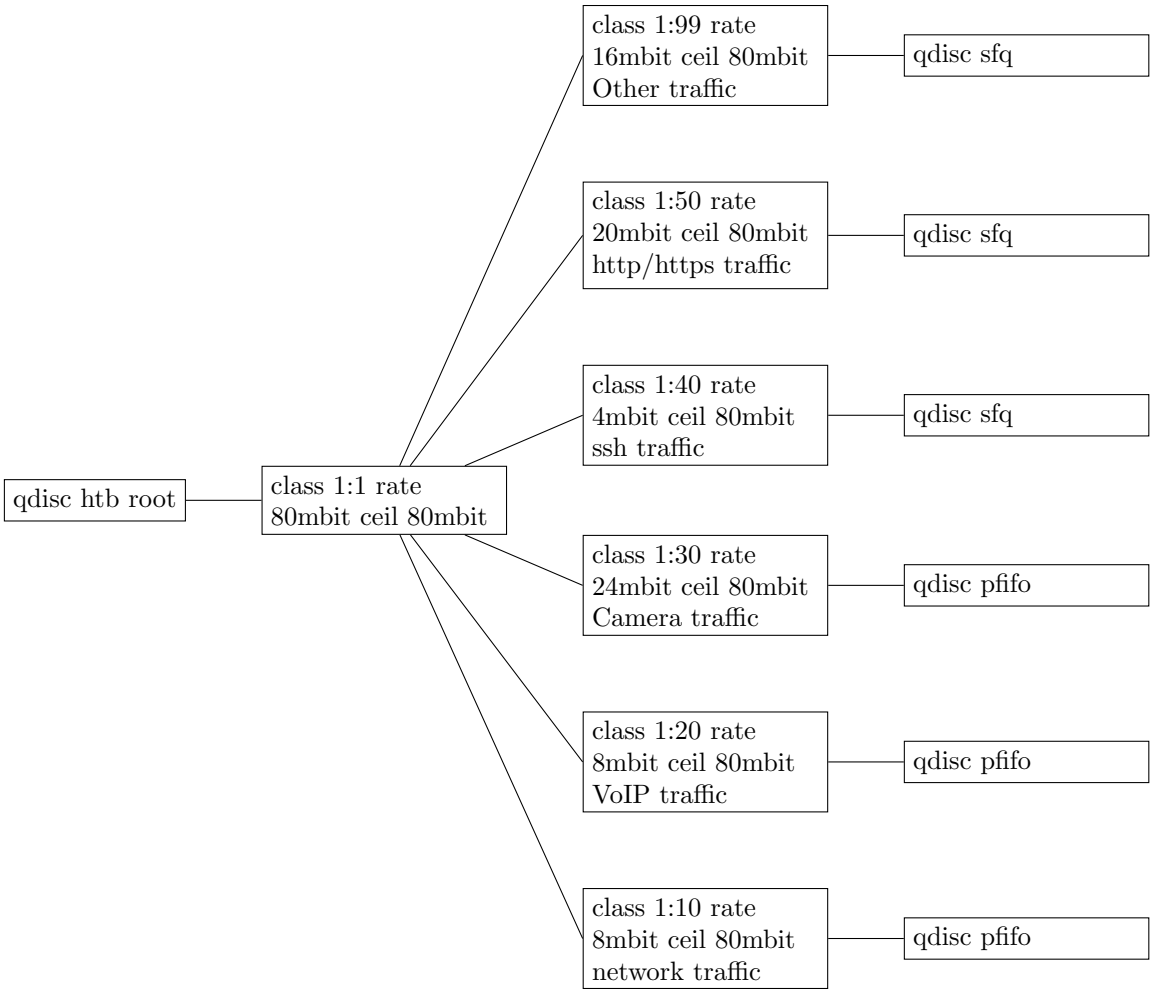
The third priority class is for camera. Video camera is also critical for surveillance. We allocated 3 times more bandwidth than class 1 and 2 because video is heavy. Once again we should never need more than 24Mbits/seconds for camera. This minimum bandwidth ensure that video traffic will never be dropped. Since this class should experience congestion we used a pfifo as qdisc for this class in the tree as it is much faster to process packets than sfq.

Those 3 classes are the most critical ones and this is why we tried to allocated more bandwidth than necessary in case congestion. Those 3 services should always have more than enough bandwidth to work 100% without ever dropping a packet.

The fourth priority class is ssh traffic. We wanted to create a specific class for ssh so that in case of congestion we still have a minimum amount of bandwidth to connect to our servers/ routers. In case of a denial of service attack this would allow us to connec to to our servers to detect where the attacks comes from and neutralize it. We also used a sfq under this class in the tree so that we the ssh bandwidth is shared equally between all the ssh connections.

The fifth priority class is for http/https traffic. For most network this is the most used type of traffic. We allocated 24Mbits/second in case of congestion for this class. In case of congestion http/https traffic could need more than 3 times the maximum bandwidth. This is why we installed a SFQ qdisc under this class in the tree. This will make sure that all users will get some bandwidth. This makes sure that one user can't use all the allocated bandwidth in case of congestion. SFQ tries to be fair and allocate some bandwidth to all users.

Finally the fifth class is the default class. All the traffic that did not match the previous classes will go to that class. We allocated the rest of the bandwidth for that class (16Mbits/seconds). The class also has a SFQ qdisc under it in the qos tree to make sure that all users get a minimum of bandwidth in case of congestion.



8.3 Testing the qos

We used iperf3 because it is the latest version of iperf and has more options than iperf and iperf2.

We used it in udp mode(-u) for the test. It is easier to understand what happens when we don't have to deal with some type of ACK. With udp if there is congestion and a packet is dropped it doesn't have any consequences on the rest of the test.

Also we used iperf3 in reverse mode(-R). This means that the traffic is send from the server to the client. It makes it easier to test the results as we don't have to connect on the server afterwards to check the logs and see how much of the data was well received.

We also had to change the -l parameter to a higher value because iperf3 wasn't able to create traffic fast enough in some scenario with the default value. We usually set -l to 20Kb. We tried different value and the test seemed to always work fine with a buffer length of 20Kb

The qos tests in the project are setup between the SH1C lan and HALL lan. The SH1C hosts contain the iperf3 servers and the HALL hosts are the clients. HA1 and HA2 are default host using default traffic(they represent the default class of the qos tree). HA3 is a voice over IP device, HA4 a camera and NS1 a DNS server to test network packets. You can test the priorities between classes of the qos by using bash /qos/test_qos.sh arg1 arg2. For instance bash /qos/test_qos.sh HA1 HA3 will flood the links from SH1C lan to HALL lan(because we are in reverse mode). The log files HA1.log and HA3.log show the iperf output of the test in order to make sure that the priority and classes minimum bandwidth is well respected.

Further improvements can be brought to the tests in future works:

1. Add tests for more classes of the tree as we can currently only run automatic tests on 4 of the 6 classes.
2. Add tests between different routers and not only SH1C and HALL.
3. Add test with more than 2 concurrent connections.

Those tests cannot be done with our test script (qos/test_qos.sh) but can be launched manually.

9 Conclusion

The configuration that we proposed for this multi-homed network follows all the required elements. Moreover, our network seems to handle failures correctly. There is still place for improvement. We could add a complete monitoring system, develop more tests, use the two prefixes at the same time as source addresses or use anycast addresses.

References

[1] Bgp_example_1. `https://gitlab.labs.nic.cz/labs/bird/wikis/BGP_example_1`. Accessed: 2017-03-19.

[2] Chapiter 18. firewalling - linux ipv6 howto (en). `http://mirrors.deepspace6.net/Linux+IPv6-HOWTO/x2416.html`. Accessed: 2017-02-23.

[3] Chapter 9. queueing disciplines for bandwidth management. `http://lartc.org/howto/lartc.qdisc.classful.html`. Accessed: 2017-02-25.

[4] Dns for rocket scientists. `http://www.zytrax.com/books/dns/`.

[5] Internet consortium system. `https://www.isc.org/`.

[6] Journey to the center of the linux kernel: Traffic control, shaping and qos. `http://wiki.linuxwall.info/doku.php/en:ressources:dossiers:networking:traffic_control`. Accessed: 2017-02-25.

[7] Netfilter et le filtrage du protocole ipv6. `http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-041/Netfilter-et-le-filtrage-du-protocole-IPv6`. Accessed: 2017-03-11.

[8] L. Colitti, V. Cerf, S. Cheshire, and D. Schinazi. Rfc7934 : Host address availability recommendations. 2016.

[9] SURFnet. *Preparing an IPv6 addressing plan*, 2013.