# Automatic IPv4 to IPv6 Transition - D2.1 Metric and Addressing Algorithm

Frédéric Beck, Isabelle Chrisment, Olivier Festor

# Automatic IPv4 to IPv6 Transition D2.1 - Metric and Addressing Algorithm

**Frederic Beck, Isabelle Chrisment, Olivier Festor**

June 1, 2010

# Contents

**Abstract**

Over the last decade, IPv6 has established itself as the most mature network protocol for the future Internet. While its acceptance and deployment remained so far often limited to academic networks, its recent deployment in both core networks of operators (often for management purposes) and its availability to end customers of large ISPs demonstrates its deployment from the inside of the network leading to the edges.

For many enterprises, the transition remains an issue today. This remains a tedious and error prone task for network administrators.

In the context of the Cisco CCRI project, we aim at providing the necessary algorithms and tools to enable this transition to become automatic. In this report, we present the first outcome of this work, namely an analysis of the transition procedure and a model of target networks on which our automatic approach will be experimented. We also present a first version of a set of transition algorithms that will be refined through the study.

# Chapter 1

# Introduction

IP networks are widely spread and used in a multitude of different applications and domains. Their growth continues at an amazing rate sustained by its high penetration in both the Home networks and the mobile markets. Although often postponed thanks to hacks like NAT, the exhaustion of available addresses, and other scale issues like routing tables explosion will occur in a near future.

IPv6 [1] was defined with a bigger address space (128 bits) and comes along with new built-in services (address autoconfiguration [4], native IPSec, routes aggregation, simplified header...). It is a fact that IPv6 deployment is slower than foreseen. Many reasons are valid to explain this: economical, political, technological, and human. Despite this slow start, IPv6 is today more than ever the most mature network protocol for the future Internet. To faster its acceptance and deployment however, it has to offer autonomic capacities that emerge in several recent protocols in terms of self-x functions reducing and often eliminating the man in the loop. We are convinced that such features are also required for the evolutionary aspects of an IP network, the transition from IPv4 to IPv6 being an essential one.

In this project, we are interested in the scientific part of the technological problems that highly impact human acceptance. Many network administrators are indeed reluctant to deploys IPv6 because, first, they do not know well the protocol itself, and they do not have sufficiently rich algorithmic support to seamlessly manage the transition from their IPv4 networks to IPv6. To address this issue, we investigate, design and aim at implementing a transition framework with the objective of making it self-managed.

As the IPv4 to IPv6 transition is a very complex operation, and can literally lead to the death of the network, there is a real need for a transition engine to ease and secure the network administrator's task; the ideal being a "one click" transition.

This report presents the metrics and addressing algorithm that we proposed to perform the initial numbering of an IPv4 network. In chapter 5.2, we present the metric and its propagation. Then, in chapter 4, we present the addressing algorithm. An illustration of how it works is given through an example in chapter 5. Finally, in chapter 6, we show how the constraints specified in D1.2 have been integrated in these algorithms.

# Chapter 2

# Glossary

In the following chapters, the terminology we are using is as follows:

**backbone** inter-connection network between more than 2 routers

**border** the border is the router that is in charge of the interconnection with the ISP

**end-network** a network at distance of 1,for which we are the predecessor, and which does not have outgoing links

**leave** a router or network at distance of 1,for which we are the predecessor, and which does not have outgoing links

**need** (id,metric,interface) tuple, where id is the child or interface id, metric the metric announced by the child or calculated on an interface, and interface is the local interface on the router on which the child is connected or null if the tuple stands for an interface need

**needs** list of need tuples

**next** all the vertices in the graph which are destination of a link which source is the current vertex.

**previous** all the vertices in the graph which are source of a link which destination is the current vertex.

**predecessor** the router at distance 1 in direction of the root.

**root** in the graph, the root is the border router

**successors or children** all the routers at a distance of 1 which are not closer to the root

**vertex** a node in the graph, can be a router, an end-network or a backbone

# Chapter 3

# Metric

One of the main requirement during this study is the aggregation of IPv6 prefixes. To do so, we added one new step in the algorithm and metric initially defined, which can be considered as a summarizing step, where the metric is summarized per interface. Then, the prefixes are first assigned to an interface, ensuring aggregation is respected, and then only assigned to the children in the graph.

## 3.1 Definition

The metric is for a vertex the number of networks it has under is authority. It is thus the number of /64 networks itself or its children have to assign. As we summarize the metric at the interface level, the components of the global node metric appear at the node level itself, but also at the interface level:

**reserved_metric** enables provisioning of network prefixes at the router level: e.g. a new interface is added in the router

**reserved_metric_per_interface** enables provisioning of network prefixes at the interface level: e.g. a new child router is added on an existing interface

**local_metric_per_interface** the number of outgoing links or out degree for a given interface, i.e. the number of links issued by this interface that we have to address

**child_metrics** the metrics announced by the successors, it is a tuple (successor_id, nb_/64_required, interface_to_successor)

We calculate the metric for each interface i as follows:

$$M_i = \sum_{N=0}^{Nbsuccessors} M_n[i] + L_v[i] + R_v[i]$$

where $L_v[i]$ is the local metric of the interface i of vertex v, $R_v[i]$ its reserved metric, and $M_n[i]$ the metric announced by the child n of vertex v, which is connected to v via the interface i.

The metric for the vertex v is thus calculated as follows:

$$M_v = \sum_{i}^{Interfaces} M_i + R_v$$

where $M_i$ is the metric of the interface i of the vertex v, and $R_v$ the reserved metric of the vertex v.

## 3.2 Propagation

This metric is propagated form the leaves (router or network) to the root. If we advertise the metric strictly as defined in the earlier section, a problem appears. Figure 3.1 illustrates this problem.

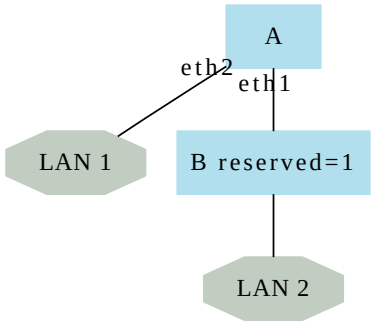


Figure 3.1: Propagation problem

In this example, if $A$ calculates its needs per interface, it gets ( (eth1,3), (eth2,1) ). When calculating the metric it should advertise, it gets 4, which stands for a /62. When that /62 is assigned to A, it begins to assign prefixes to its interfaces. The interface *eth1* needs 3 /64, and is thus assigned a /62, leaving nothing for *eth2*.

Thus when calculating the metric to advertise, the sum between the metrics of each interface should be done in terms of prefix length, and not directly in terms of numeric values. In this example, the metric to advertise for A would be:

$$M_A = M_{eth1} + M_{eth2} = /62 + /64 = /61$$

Therefore, A will demand a /61, and will be able to assign a /62 on eth1, and a /64 on eth2.

In our algorithms, we introduced a new function called *get_metric_to_adv(integer)*, that takes an integer as parameter (the metric calculated), and returns the following power of 2. In our example, this function would return 4 for eth1 (it needs a /62 which permits to have 4 /64) and 1 for eth2, and finally, $A$ will advertise a metric of 8.

We also added another function, *get_predecessor_interface()*, that returns the interface of the predecessor on which we are connected.

The propagation algorithm is thus:

$N = []$
**for** $i$ in $self.interfaces$ **do**
  $N[i] = local\_metric[i] + reserved\_metric[i]$
  **for** $M_c$ in $self.child\_metrics[i]$ **do**
    $N[i] = N[i] + M_c$
  **end**
**end**
return N

**Function** `calculate_metric_per_interface`

$N = calculate\_metric\_per\_interface()$
$M = reserved\_metric$
**for** $i$ in $self.interfaces$ **do**
| $M = M + get\_metric\_to\_adv(N[i])$
**end**
return get_metric_to_adv(M)

**Function** `calculate_metric`

**if** $self$ is border **then**
| stop
**end**
**if** $len(self.child\_metrics) \neq nb(self.successors)$ **then**
| wait until $len(self.child\_metrics) == nb(self.successors)$
**end**
$M_i = calculate\_metric()$
$Pred_i = self.get\_predecessor()$
$Pred_{iface} = self.get\_predecessor\_interface()$
$Pred_i.child\_metric.append((M_i, Pred_{iface}))$
$Pred_i.announce\_metric()$

**Function** `announce_metric`

**foreach** leave $i$ **do**
| $i$.announce_metric()
**end**

**Algorithm 4**: Metric Propagation Algorithm

## 3.3 Conclusion

The metric defined and its propagation can be summarized in figure 3.2.

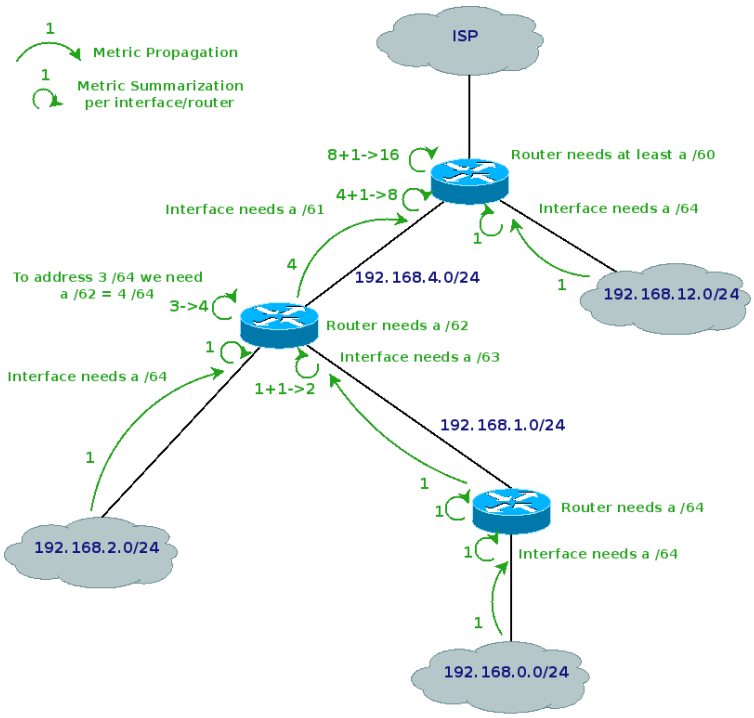


Figure 3.2: Metric Propagation Summary

# Chapter 4

# Addressing Algorithm

## 4.1 Algorithm

Contrarily to the metric that was propagated from the leaves to the root, the addressing algorithm is executed from the root to the leaves. In input, we give the prefix of the site which is delegated to the root.

On each router, beginning from the root, we determine the per interface needs. The need of an interface or a child is a *(id,metric,interface)* tuple, where id is the child or interface id, metric the metric announced by the child or calculated on an interface, and interface is the local interface on the router on which the child is connected or null if the tuple stands for an interface need. The metrics have already been propagated and calculated, we just need to build a list of needs tuples.

As the router has already been assigned a prefix matching its needs (either the site prefix if we are running the algorithm at the border, or a prefix assigned by the parent), it assigns an aggregated prefix to each interface. All prefixes that have not been assigned are kept in a list of prefixes globally available on the router level. Namely, if we use the reserved metric at the router level, the reserved prefixes will be kept in that list.

Then, on each interface, it determines the needs of its children (router, network or link interconnecting 2 routers), and assigns them prefixes. As it was done at the router level, all non-assigned prefixes are stored in a list of available prefixes.

Finally, the router configures addresses on all links end points and passes th torch to its successors.

The prefix assignment algorithm is described below:
where:

$candidate_{prefix}$ the current candidate for assignment

**divide** divides a prefix of length X in two prefixes of length X+1

**get_prefix_len** takes a metric as entry, and returns the prefix length required to fulfill the metric needs

**get_matching_prefix** gets in the list of available prefixes the one with the smallest mask equal or bigger to the requirements.

**max** in the list of needs, returns the need tuple that has the biggest metric

**min** in the list of needs, returns the need tuple that has the smallest metric

**required_length** the prefix length returned by get_prefix_len, it is the length of the prefix we need for assignment

**Input**: $site\_prefix = P$
$self.available = [P]$
$interface_{needs} = []$
**for** $i$ in $self.interfaces$ **do**
  $interface_{needs}[i] = local\_metric\_per\_interface[i] + reserved\_metric\_per\_interface[i]$
  **for** $c$ in $self.child\_metrics[i]$ **do**
    $interface_{needs}[i] = interface_{needs}[i] + M_c$
  **end**
**end**
**for** $max(interface_needs)$ to $min(interface_needs)$ **do**
  $(I, metric_I, interface_I) = need$
  $required\_length = get\_prefix\_len(metric_I)$
  $candidate_{prefix} = get\_matching\_prefix(self.available, metric_I)$
  $self.available.remove(candidate_{prefix})$
  **while** $len(candidate_{prefix}) \neq required\_length$ **do**
    $(P_1, P_2) = candidate_{prefix}.divide()$
    $candidate_{prefix} = P_1$
    $self.available.append(P_2)$
  **end**
  $I.prefix = candidate_{prefix}$
**end**
$needs = self.child\_metrics$
**for** $max(needs)$ to $min(needs)$ **do**
  $(R, metric_R, interface_R) = need$
  $iface = R.get_iface(interface_R)$
  $required\_length = get_prefix_len(metric_R)$
  $candidate_{prefix} = get\_matching\_prefix(iface.available, metric_R)$
  $iface.available.remove(candidate_{prefix})$
  **while** $len(candidate_{prefix}) \neq required\_length$ **do**
    $(P_1, P_2) = candidate_{prefix}.divide()$
    $candidate_{prefix} = P_1$
    $iface.available.append(P_2)$
  **end**
  $R.prefix = candidate_{prefix}$
**end**
**foreach** successor $R$ which is a router **do**
  $R.assign\_prefixes()$
**end**

**Algorithm 5**: Prefix Assignment Algorithm

## 4.2 Conclusion

The proposed addressing algorithm is illustrated in figure 4.1.

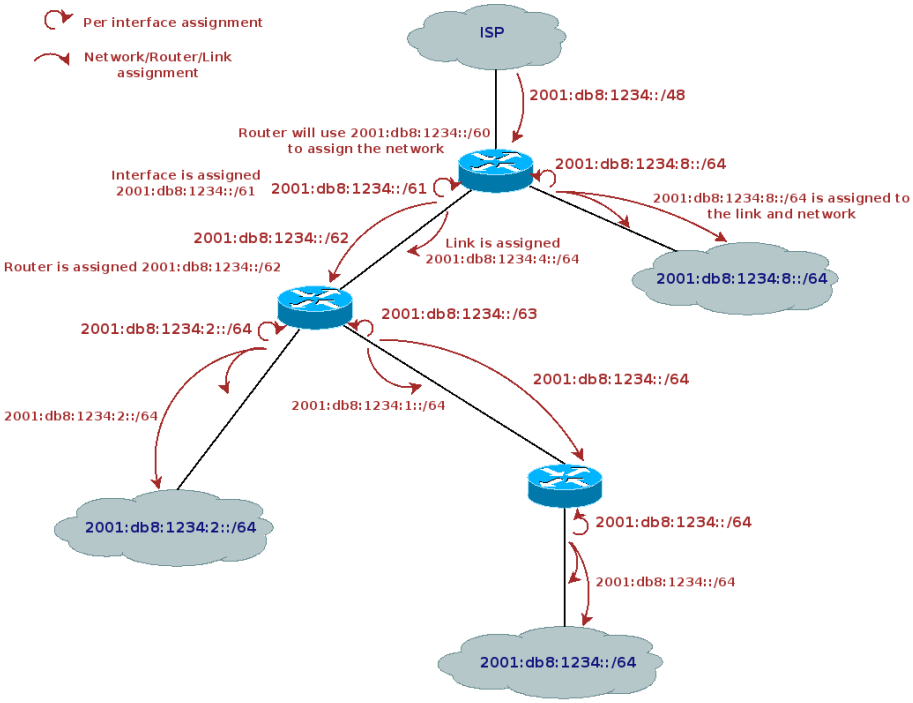


Figure 4.1: Addressing Algorithm Summary

# Chapter 5

# Illustration

If we consider the following IPv4 network, with its set of basic constraints:

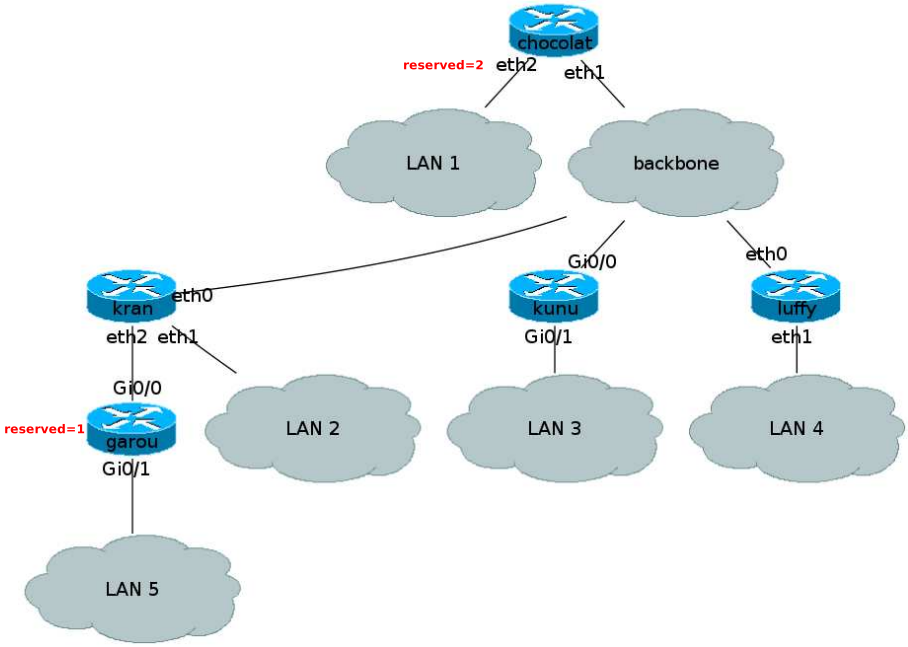


Figure 5.1: Initial IPv4 network

In this example, the router *chocolat* has a reserved metric of 2 at the interface *eth2*, for example because other routers will be connected to *LAN1*. The router *garou* has a global reserved metric of 1 at the router level, be cause a new interface will be added with one new subnet behind it.

If we apply the metric and its propagation, we obtain the figure 5.2.

Then by running the addressing algorithm, we obtain the figure 5.3.

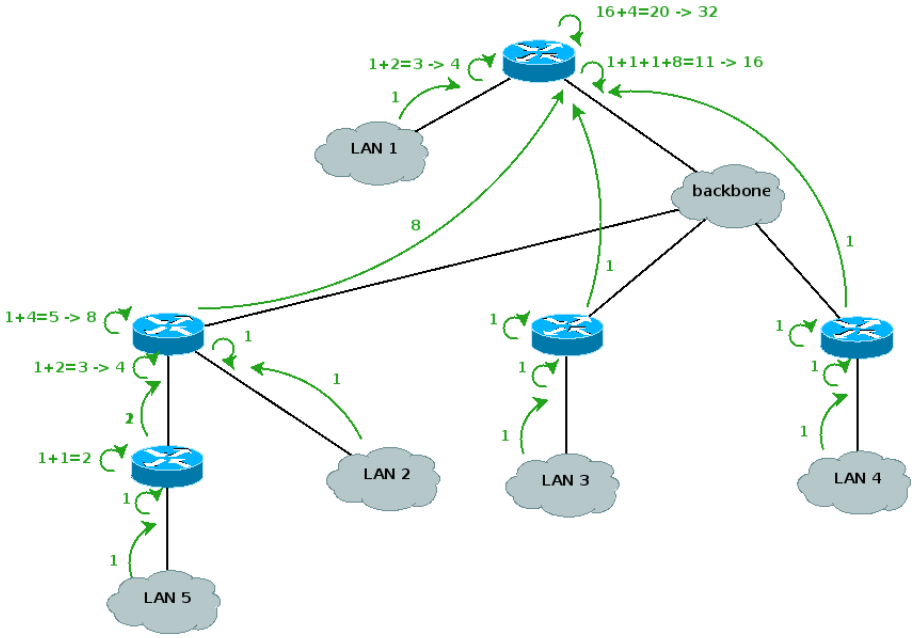


Figure 5.2: After metrics propagation

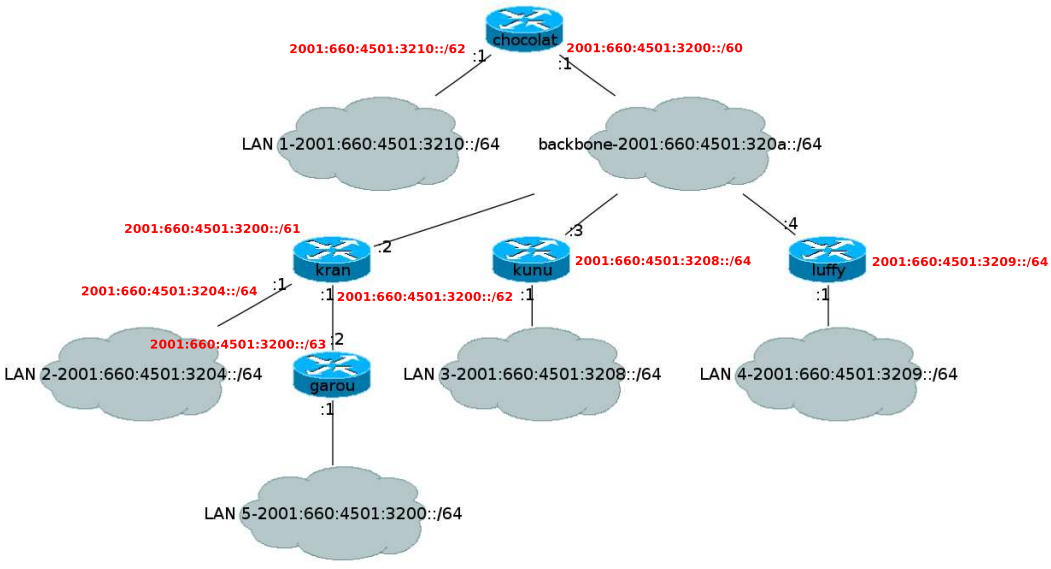


Figure 5.3: After metrics propagation

*chocolat* has a reserved metric of 2 at the interface *eth2*. Thus, this interface requires a /62 to address the existing subnet, and the 2 reserved, which is why it is assigned 2001:660:4501:3210::/62. *LAN1* is assigned 2001:660:4501:3210::/64, while 2001:660:4501:3212::/63 and 2001:660:4501:3211::/64 are still available for assignment on that interface. In the end, we have the possibility to address 3 subnets even if we reserved only 2, due to aggregation issues.

In the same way, *kran* who needs to assign 5 subnets needs a /61 and is assigned 2001:660:4501:3200::/61. Behind *eth1*, we have only *LAN2*, and thus we only need to assign 2001:660:4501:3204::/64. Behind *eth2*, we have 4 prefixes and thus assign 2001:660:4501:3200::/62. *kran* still has the prefixes 2001:660:4501:3206::/63 and 2001:660:4501:3205::/64 available on the router level, and 2001:660:4501:3203::/64 at *eth2*, even if nothing was reserved, because otherwise we would not be able to respect the aggregation.

Finally, *garou* has been assigned 2001:660:4501:3200::/63. 2001:660:4501:3200::/64 has been assigned to *LAN5* while 2001:660:4501:3201::/64 is still available, as required by the constraint.

# Chapter 6

# Constraints Integration

In this chapter, we will present the integration of the constraints defined in D1.2. the constraints will not be detailed, as they have been presented in D1.2, we will only describe their integration.

Some of the constraints have been integrated since the beginning of the study. These are the *reserved* and *backbone* constraints.

## 6.1 Exclude Prefix

This constraint enables to exclude some prefixes from the addressing plan. As the prefixes are assigned recursively from the root to the leaves, the decision and actions must be taken at the border router, when the prefixes are assigned to the interfaces of this router:

**Input**: $exclude\_prefixes = [P_1, P_2 ... P_N]$: list of excluded prefixes
$P_{candidate}$: candidate prefix for assignment to an interface
**if** $P_{candidate}$ in $exclude\_prefixes$ **then**
  return True
**end**
**else**
  **for** $P$ in $exclude\_prefixes$ **do**
    **if** ($P$ contains $P_{candidate}$) or ($P_{candidate}$ contains $P$) **then**
      return True
    **end**
  **end**
**end**
return False

**Function** `is_prefix_excluded`

This function checks if a prefix is excluded. A prefix is consider as excluded if it matches exactly a prefix in the list, if it contains at least one excluded prefix or if it is contained at least in one excluded prefix.

**Input**: $exclude\_prefixes = [P_1, P_2 ... P_N]$: list of excluded prefixes
$P_{candidate}$: candidate prefix for assignment to an interface
**while** not *stop* **do**
  **if** $is\_prefix\_excluded(P_{candidate}, exclude\_prefixes)$ **then**
    $available\_prefixes.append(P_{candidate})$
    $P_{candidate} = get\_new\_candidate()$
  **end**
  **else**
    $stop = True$
  **end**
**end**
Assign $P_{candidate}$ to the interface

**Algorithm 7**: Exclude Prefix Algorithm

A prefix considered as excluded is added in the list of available prefixes, so that it can be reused with a longer prefix, and thus limit the impact of the constraint to the longest prefix possible, which will be the closest one to the prefix excluded itself. The impact of this constraint could be even less if we distribute the decision among the routers, and do not limit ourselves at the border, but considering our scope of SME networks, we can accept to waste a few /64 prefixes and make the decision at the border.

If no new candidate matching the needs can be used, the constraint is ignored and the addressing plan is proposed without it.

## 6.2 Force Prefix

This constraints permits to force a prefix on a link, an interface, or a router itself. To do so, we modified slightly the metric propagation and addressing algorithm.

When calculating and propagating its metric, a node also checks if a prefix is forced locally on an interface or at the router, of if a child has such a constraint set. If no constraint is set, pursue with the regular algorithm, otherwise generate a *requested prefix* with matches the metric calculated and the constraint and propagate it alongside the metric to the parent in the graph, which repeats the same steps.

When the root is reached, all subtrees in the graph that have to force a prefix at any place have a set of requested prefixes at each level, including the root. When addressing the network, the algorithm will try first to assign the requested prefix. This requested prefix is available and can be assigned directly without any verification, as all these steps have been validated by the metric propagation.

If at any step of the algorithm the force prefix does not match the needs, or if the prefix is not available for any reason, the constraint is ignored. To avoid this last condition, when addressing the network, we address first the children which have this constraint set. As the requested prefixes are propagated alongside the metric, another option is to force the modification of the metric in order to match the requested prefix length, if this prefix is longer than the metric calculated. We chose to minimize the prefixes used by the algorithm, but this decision could be left to the administrator.

If we take the example shown in figure 6.1, we have chocolat that forces the prefix 2001:660:4501:3234::/62 on eth2, kran who requests the prefix 2001:660:4501:3220::/61, the backbone that forces 2001:660:4501:3229::/64, LAN5 2001:660:4501:3220::/64 and the link between garou and kran that must have the prefix 2001:660:4501:3222::/64. The reserved metrics are the same than in figure 5.1.

The metric and requested prefix propagation goes as shown in figure 6.2.

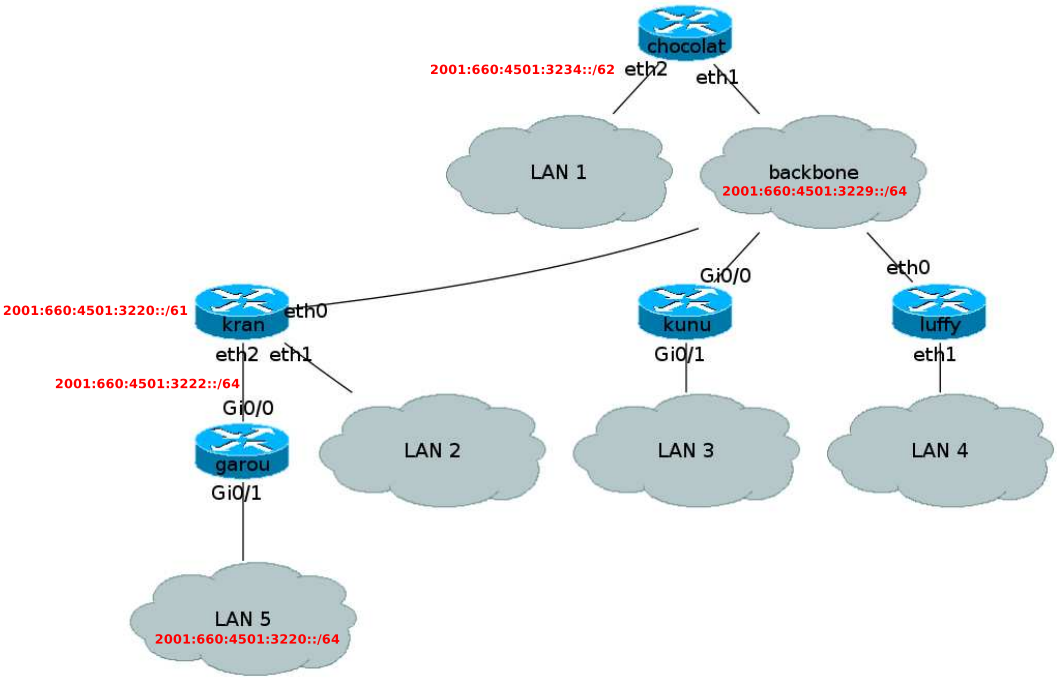


Figure 6.1: Initial network with forced prefixes

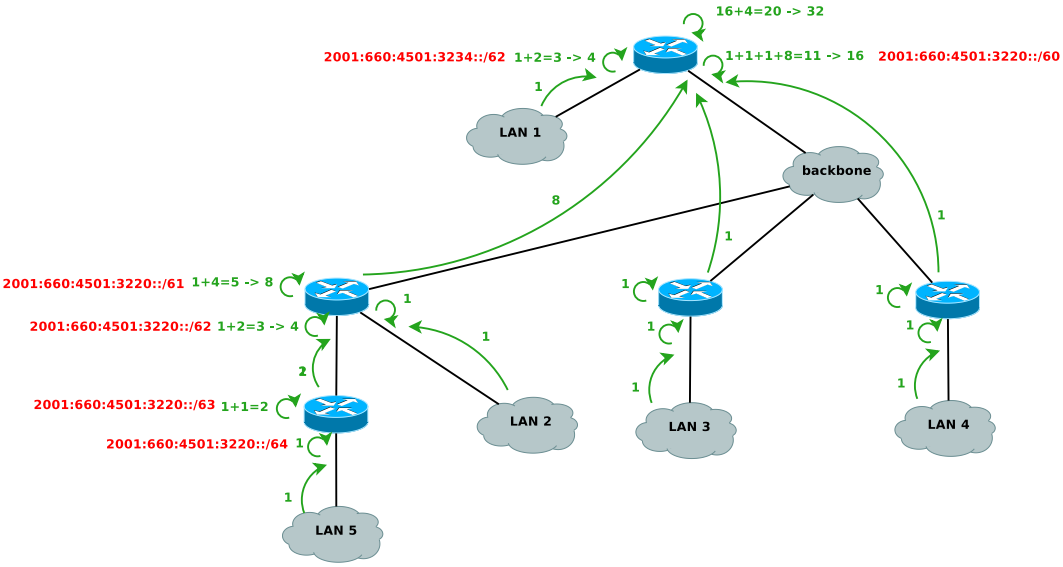


Figure 6.2: Metrics and requested prefixes propagation

when assigning the prefixes with the addressing algorithm, as all requested prefixes match the constraint, we obtain the new addressing plan shown in figure 6.3.

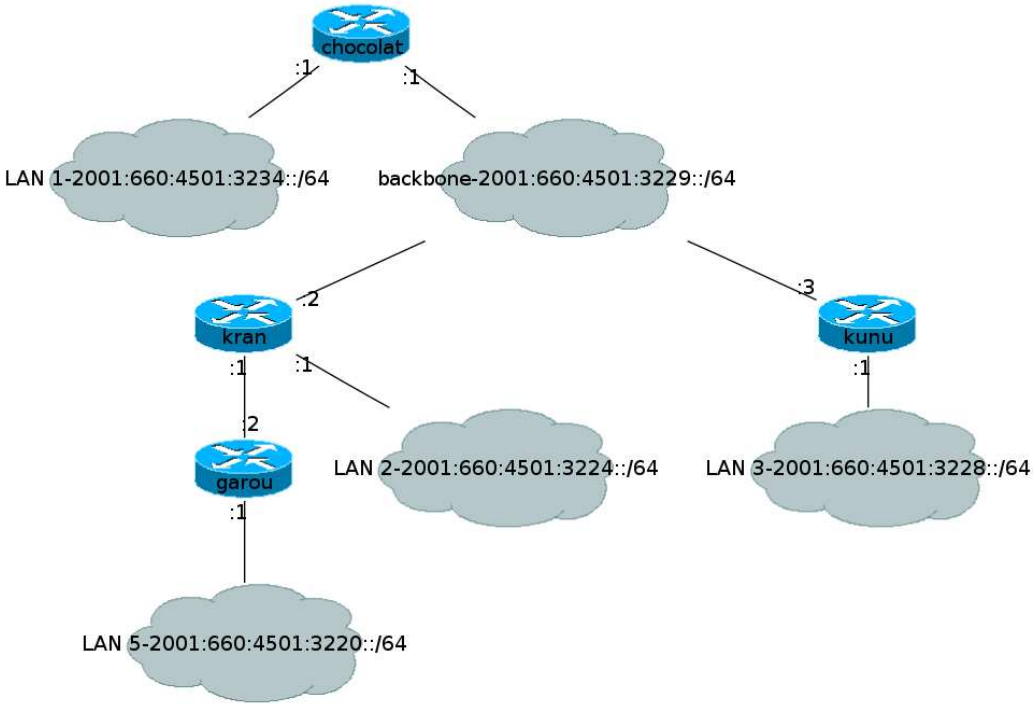


Figure 6.3: New addressing plan with forced prefixes

## 6.3 End Points Addressing

For security issues, it is not recommended to use sequential addressing such as ::1, ::2... as it eases the scanning of the network. We have proposed and implemented 3 options for the addressing of the links end points:

**sequential** The default behavior, where the addressing is sequential, but with the possibility to set the starting point and increment. By default, we start at ::0 and increment by one.

**eui64** If we know the MAC address of the router interface, we generate and use the EUI-64 interface identifier.

**random** If we know the MAC address of the router interface, we generate a random interface identifier as defined in RFC4941 [3].

This constraint is meant to be set at the site level in the XML configuration file.

## 6.4 Router-to-Router Link

We identified 3 options that we think are suitable to perform the addressing of point-to-point links:

- /64 prefixes on all links

- /126 or /127 prefixes
- /112 prefixes

Setting up point-to-point links could also be considered, and could be set up from a technical point of view, but is not an option that we considered, because of the complexity and implication it would have on the addressing plan (we would need to remember the link local addresses of the routers to identify them, debugging...).

As we are mainly targeting SME networks, we can expect to get a /48 to address the site as the default configuration. Moreover, considering the size of the network, it would contain a reasonable number of links and subnets. We decided to use /64 prefixes on all subnets and links, because it means less complexity and a better aggregation within the site. Moreover, it is easier to build, read, maintain and update the resulting addressing plan, making both the maintainance and debugging of the network easier. Of course, we are well aware that this implies waisting some address space (as we address point-to-point links with a full /64 prefix), but this is not considered as harmful regarding the scope of this study and the amount of prefixes that we have at our disposal.

The 2 others options are better suited for a use in a provider environment with a huge backbone. Considering the number of links, avoiding the waist of address space makes more sense. Using these mechanisms makes possible to aggregate the backbone in a single /64 (or shorter) prefix and filter all access to it using this single prefix. Moreover, we can also consider that the network administrators in charge of such a network will feel more at ease dealing with the added complexity of using this types of prefixes.

## 6.5 Multihoming

We consider 2 scenarios of multihoming: at subnet level, and at the site level.

### 6.5.1 At subnet level

#### One router advertises 2 prefixes on a subnet

No modifications have been necessary for this scenario. In the logical representation, the multihomed network is represented as 2 logical networks issued on the same interface at the router. We called this scenario *multihomed_1*, and the output if shown in figure 6.4

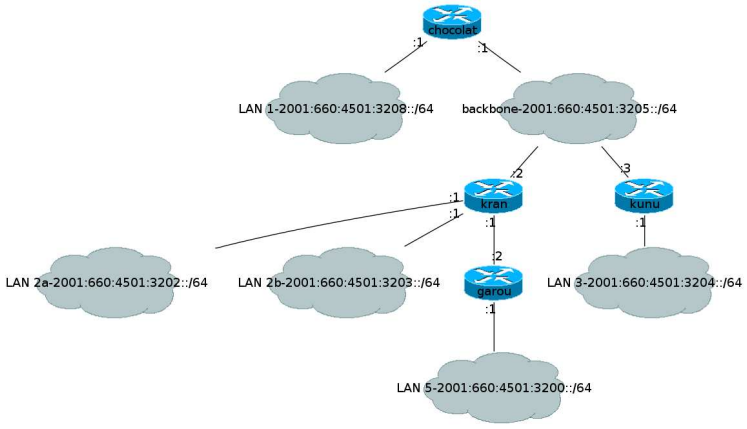


Figure 6.4: One router advertises 2 prefixes on a subnet

### Two routers advertise 2 prefixes on a subnet

No modifications have been necessary for this scenario neither. In the logical representation, the multi-homed network is represented as 2 logical networks issued at each of the routers. We called this scenario *multihomed_2*, and the output if shown in figure 6.5

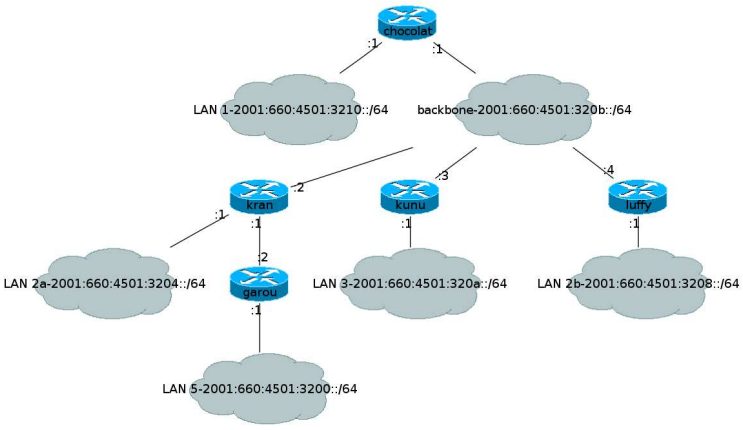


Figure 6.5: Two router advertise 2 prefixes on a subnet

### Two routers advertise the same prefix on a subnet

This scenario stands as a loop in the routing infrastructure. This may be set for redundancy. The important step here is to define with of the predecessors will be the upstream router. It can be done by 2 different ways:

- By default, the shortest path algorithm is used on the graph, and the predecessor on the shortest path to the root is chosen as upstream router
- By using the *Force Upstream Router* constraint, we can force another router to be the upstream. We will not detail the integration of this constraint, as it is quite straightforward, consisting only in parsing a tag in the configuration and setting the value in the corresponding python object, no modification of the algorithm is necessary.

As it was done with the backbone, the router chosen as upstream will have a weight of 1 on the link to the subnet and will calculate its metric while taking this subnet into account, whereas the backup router will put a weight of 0 on the link to the subnet and will thus not consider it in its metric calculation.

When configuring the routers, we want both of them to advertise the prefix, and we use RFC4191 [2] to set routers preferences to High for the upstream and Low for the backup, Medium behind the default otherwise. By doing so, the upstream router is always the default router chosen for outgoing traffic.

However, from the routing point of view, the returning traffic will follow the shortest path, which means that if we force the router that is not on the shortest path to the root as upstream, the outgoing traffic will go through him, but the returning traffic will not, making the routing asymmetric. This could be corrected by using the bandwidth command in the routers to virtually modify the bandwidth of the links and force the return path through the upstream router. As it was not considered vital in this study, we decided not to address it at this stage.

We called this scenario *multihomed_2*, and the output if shown in figure 6.6

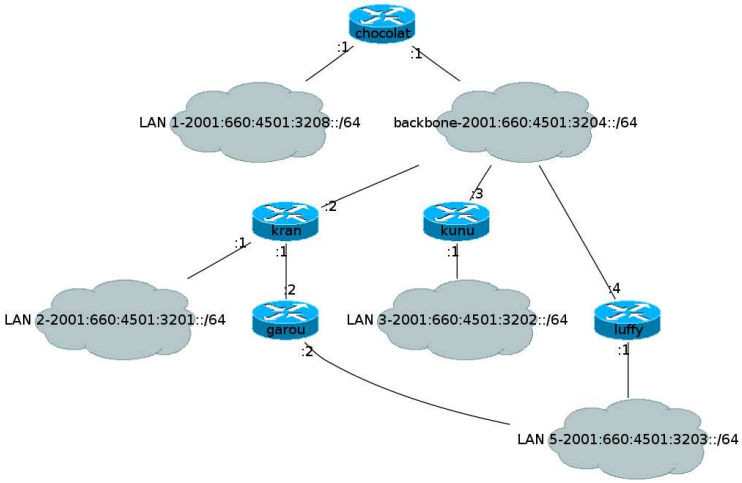


Figure 6.6: Two router advertise the same prefix on a subnet

### Two routers advertise 2 prefixes on a subnet with redundancy

In this case, we use two logical networks and we have 2 loops in the routing infrastructure. There is no difference for the addressing algorithm, but it raises a problem concerning RFC4191.

The router preference option is set at the interface level in the router, which leads to the problem shown in figure 6.7.

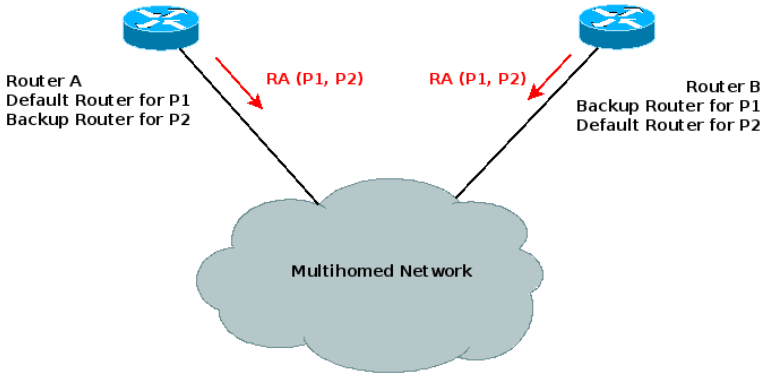


Figure 6.7: Router Preference Issue

We have 2 routers A and B and 1 network. The network is multihomed with 2 prefixes P1 and P2. We want both routers to advertise both prefixes, each one being the default router for one prefix, and the backup for the other one:

- A is the default router for P1, and is used as a backup for P2
- B is the default router for P2, and is used as a backup for P1

Each router is thus the backup for the other one. To do so, we would want to use the router preferences in RAs as defined in RFC 4941, i.e.

- A with a preference High on P1 and Low on P2

- B with a preference High on P2 and Low on P1

The problem is that the router preference option is not present at the prefix level, but at the interface level. We can give a high preference on A, but we cannot link it to P1, if we do it, the router will have a preference of High for P2 as well, as the information is not set in the prefix information option of the RA.

We submitted the problem to the authors to see if they considered this scenario and are planning upon their response to write a small IETF draft on this issue depending on the feedback we obtain.

### 6.5.2 At site level

All the scenarios concerned by this type of multihoming depend on the logical representation and do not impact the addressing algorithm. If the logical representation is correctly defined, running several times the algorithms with the parameters corresponding to each site prefix is sufficient.

The only modifications we made were implementation issues, to make sure the tool was able to take into account an existing IPv6 address plan and generate a new one without corrupting the existing one, especially when configuring the routers. No information is reused from the existing addressing plan (such as the subnets IDs), as the topology and constraints could modify it, and if we have the exact same topology and constraints, the algorithm will always generate the same IDs and addressing plan. The only parameter reused is the random seed if this type of end-points addressing is used.

## 6.6 Unique Local Addresses and Provider Independent Addresses

ULA and PI prefixes are just other Global prefixes. They are handled the exact same way than regular global prefixes, and no modifications were required. We performed some experimental studies of the deployment of such prefixes and their interaction with address selection, and the results we obtained are the ones we were expecting.

Using ULA has some impact on the filtering, as we must make sure that they do not spread out of the network. This will be discussed in D3.1.

## 6.7 Filtering Related Constraints

Additional constraints (NAT, DMZ...) have only impacts on the filtering and will be discussed in D3.1.

# Chapter 7

# Conclusion

In this deliverable, we presented the metrics and addressing algorithm we proposed to generate an addressing plan for the initial numbering of a network during its transition to IPv6. We also showed how the constraints defined in D1.2 have been integrated in these algorithms.

D2.2 will present the implementation of these algorithms in order to implement a prototype of the aimed one click transition tool.

# Bibliography

[1] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFC 5095.

[2] R. Draves and D. Thaler. Default Router Preferences and More-Specific Routes. RFC 4191 (Proposed Standard), November 2005.

[3] T. Narten, R. Draves, and S. Krishnan. Privacy Extensions for Stateless Address Autoconfiguration in IPv6. RFC 4941 (Draft Standard), September 2007.

[4] S. Thomson, T. Narten, and T. Jinmei. IPv6 Stateless Address Autoconfiguration. RFC 4862 (Draft Standard), September 2007.