Automatic IPv4 to IPv6 Transition D2.2 - Transition Engine Specification and Implementation

Frédéric Beck, Isabelle Chrisment, Olivier Festor

Automatic IPv4 to IPv6 Transition D2.2 - Transition Engine Specification and Implementation

Frederic Beck, Isabelle Chrisment, Olivier Festor

June 1, 2010

Contents

1 Introduction **2**

2 Specifications **3**
2.1 Generic comments and technical choices . . . 3
2.2 UML Modeling . . . 3
2.3 Configuration Files . . . 5
2.3.1 Network Topology . . . 5
2.3.2 XML configuration . . . 6

3 Implementation **8**
3.1 Graph Handling . . . 8
3.2 XML Parsing and Serialization . . . 9
3.3 Remote Configuration . . . 10
3.4 Routing Protocols . . . 10
3.5 Graphical Interface . . . 11
3.6 Tool Distribution . . . 17
3.6.1 VMWare Image . . . 17
3.6.2 GForge . . . 17

4 Conclusion **18**

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 is seen as 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 automate the transition. 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 many 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 tricks like NAT, the exhaustion of available addresses, and other scale issues like routing tables explosion will occur in a near future.

IPv6 [2] was defined with a bigger address space (128 bits) and comes along with new built-in services (address autoconfiguration [5], native IPSec, routes aggregation, simplified header...). Despite its slow start, IPv6 is today more than ever the most mature network protocol for the future Internet. To faster its acceptance and deployment, it has however to offer autonomic capabilities 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 specification and implementation of the Transition Engine based on the algorithms presented in D2.1.

Chapter 2

Specifications

2.1 Generic comments and technical choices

To validate the addressing algorithm proposed in D2.1, we developed a prototype using the Python language. This language has been chosen for being Object Oriented and offering a large set of libraries. The network is represented as a graph derived from a DOT [1] representation.

To perform the numbering of the network, some more information and data are required. The are given in a separate XML file. This couple DOT/XML consists in the two configuration files that are required by the transition engine.

Once the engine has completed its operation, it produces an other DOT/XML couple detailing the newly generated network. This couple respects the same data model as the one used for configuration. It can thus be reused for another run of the engine on the same topology.

Finally, the engine must be capable of remotely configuring the network devices in order to enable IPv6 on the network.

The code architecture is shown in figure 2.1.

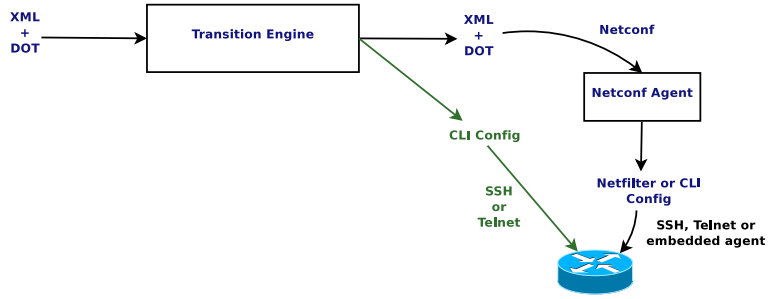

Figure 2.1: Planned Architecture

We were planning to use Netconf for the configuration of the network devices, but since no full framework was available, for all envisioned devices, at the time we started the project, we decided to configure the devices directly via CLI over SSH or Telnet.

2.2 UML Modeling

Figure 2.2 presents an UML modeling of the transition engine's code.

[1] `http://www.graphviz.org`

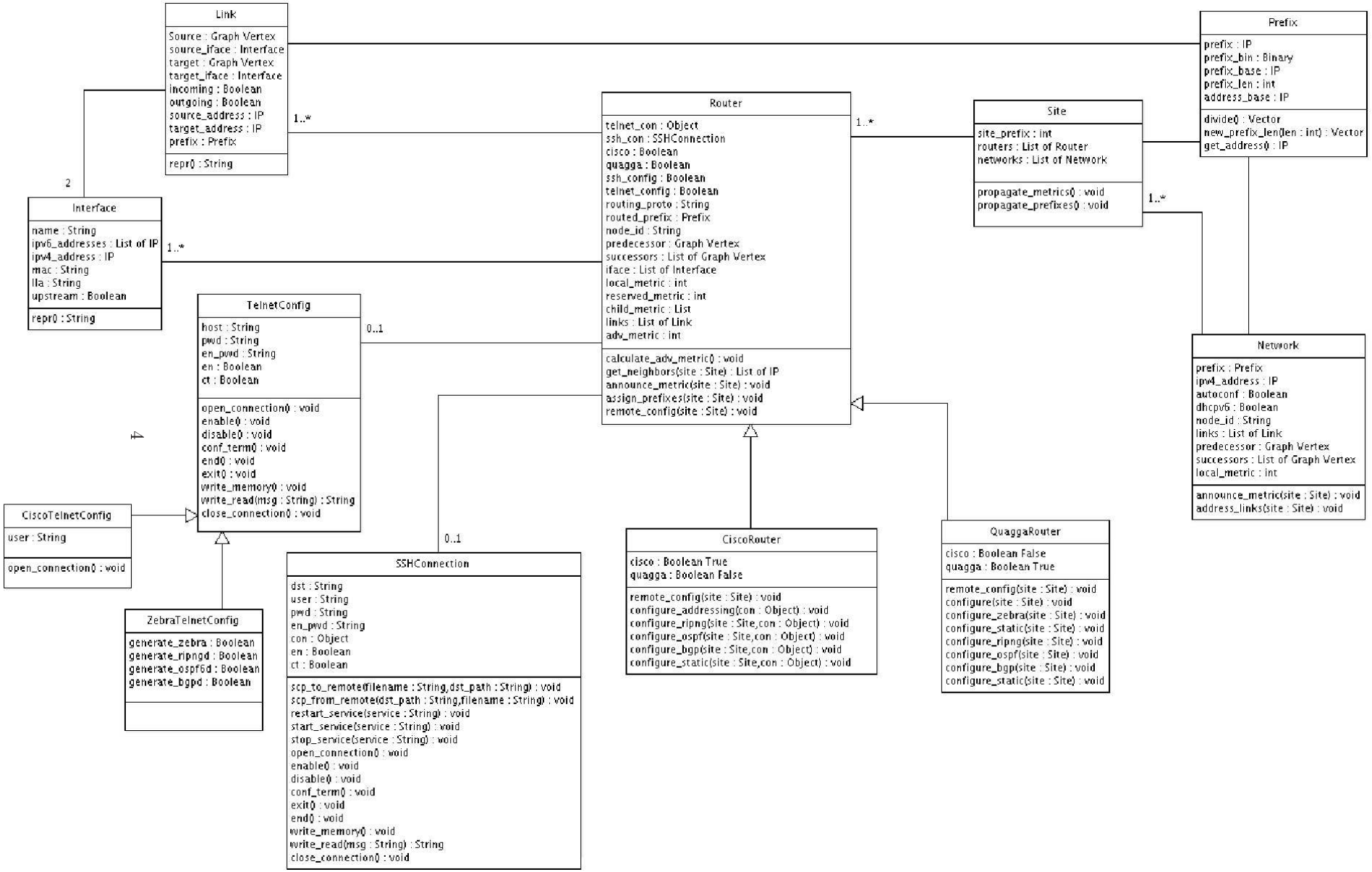

Figure 2.2: Transition Model

This representation is non exhaustive, as only the key objects and attributes/methods are shown. The core of this model is the Site class. It stands for the whole network, and is the main class which has to be instantiated.

2.3 Configuration Files

2.3.1 Network Topology

As we already said, the network is represented as a graph. We decided to use the Dot language from the Graphviz framework [2]. The advantage of this framework, is that it allows to represent the network with many characteristics, easily generate a graphical view exploitable in WEB or GUI interfaces, and the language itself is really easy. A network is defined as follows:

```
graph G {
        node [shape=plaintext];
        edge [arrowhead=none, labeldistance=1];

        chocolat [label="chocolat",comment="border",group="router", image="./logo/router.png", labelloc=b];
        kran [label="kran",group="router", image="./logo/router.png", labelloc=b];
        kunu [label="kunu",group="router", image="./logo/router.png", labelloc=b];
        garou [label="garou",group="router", image="./logo/router.png", labelloc=b];
        luffy [label="luffy",group="router", image="./logo/router.png", labelloc=b];

        lan1 [label="LAN 1", group="network", image="./logo/cloud.png"];
        lan2 [label="LAN 2", group="network", image="./logo/cloud.png"];
        lan3 [label="LAN 3", group="network", image="./logo/cloud.png"];
        lan4 [label="LAN 4", group="network", image="./logo/cloud.png"];
        lan5 [label="LAN 5", group="network", image="./logo/cloud.png"];
        backbone [label="backbone", group="network", image="./logo/cloud.png"];

        chocolat -- lan1 [weight="1.0",taillabel="eth2",headlabel=""];

        chocolat -- backbone  [weight="1.0",taillabel="eth1",headlabel=""];
        backbone -- kran [weight="0.0",taillabel="",headlabel="eth0"];
        kran -- lan2 [weight="1.0",taillabel="eth1",headlabel=""];

        backbone -- kunu  [weight="0.0",taillabel="",headlabel="Gi0/0"];
        kunu -- lan3 [weight="1.0",taillabel="Gi0/1",headlabel=""];

        kran -- garou [weight="1.0",taillabel="eth2",headlabel="Gi0/0"];
        garou -- lan5 [weight="1.0",taillabel="Gi0/1",headlabel=""];

        backbone -- luffy  [weight="0.0",taillabel="",headlabel="eth0"];
        luffy -- lan4 [weight="1.0",taillabel="eth1",headlabel=""];
}
```

All nodes are bound to a group, which is either *router* or *network*. The border router, gateway in this example, has an additional comment *border*, and is the root of the graph. Finally, all edges have weights defined as explained D2.1, and have tail and head labels, corresponding to the physical interfaces on the routers. The edges do not have comments at the moments, but this attribute will be used for defining VLAN, tunnels or trunks.

Figure 2.3 presents a graphical representation of this network.

[2] http://www.graphviz.org

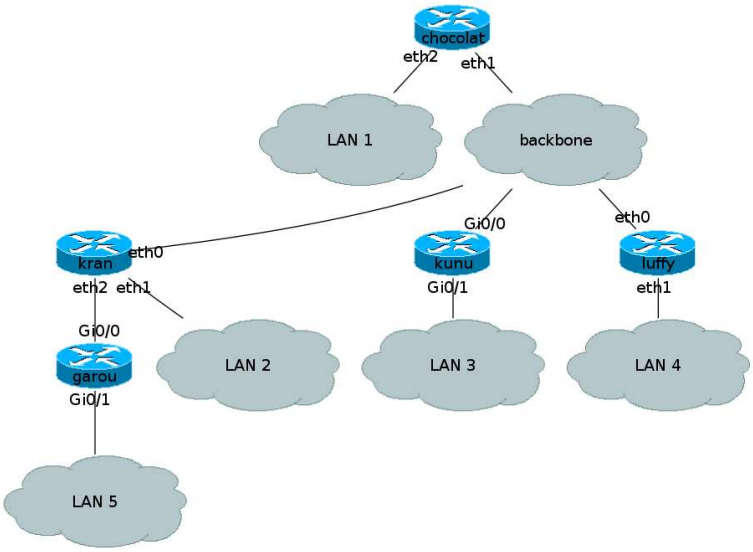

Figure 2.3: Network Graph Topology

2.3.2 XML configuration

Alongside the dot representation, the program uses an XML file as input, to gather more information about the network. This file can be validated with the following DTD:

```
<!ELEMENT config (site_prefix?, routed_prefixes?, available_prefixes?, bgp?, write_memory?,
    addressing?, exclude_prefixes?, routers,networks)>

<!ELEMENT site_prefix (#PCDATA)>

<!ELEMENT routed_prefixes (prefix*)>
<!ELEMENT available_prefixes (prefix*)>

<!ELEMENT bgp (as,neighbor*)>
<!ELEMENT as (#PCDATA)>
<!ELEMENT neighbor (#PCDATA)>
<!ATTLIST neighbor as CDATA #REQUIRED>

<!ELEMENT exclude_prefixes (prefix*)>
<!ELEMENT prefix (#PCDATA)>

<!ELEMENT addressing (sequential?, random?, eui64?)>
<!ELEMENT sequential (start, inc)>
<!ELEMENT start (#PCDATA)>
<!ELEMENT inc (#PCDATA)>
<!ELEMENT random (#PCDATA)>
<!ELEMENT eui64 (#PCDATA)>


<!ELEMENT routers (router*)>
<!ELEMENT router (id, routed_prefixes?, available_prefixes?, border?, quagga?, cisco?, telnet?,
    ssh?, predecessor?, interfaces, force_prefix?)>
<!ELEMENT id (#PCDATA)>
```

```
<!ATTLIST id config (telnet|ssh) "telnet" routing (static|ripng|ospf|bgp) "ripng">
<!ELEMENT border (#PCDATA)>
<!ELEMENT quagga (#PCDATA)>
<!ELEMENT cisco (#PCDATA)>
<!ELEMENT telnet (addr, user?, password, enable_password,generate_zebra?,generate_ripngd?,
    generate_ospf6d?,generate_bgpd?)>
<!ELEMENT ssh (addr, user, password, enable_password?)>
<!ELEMENT addr (#PCDATA)>
<!ELEMENT password (#PCDATA)>
<!ELEMENT enable_password (#PCDATA)>
<!ELEMENT generate_zebra (#PCDATA)>
<!ELEMENT generate_ripngd (#PCDATA)>
<!ELEMENT generate_ospf6d (#PCDATA)>
<!ELEMENT generate_bgpd (#PCDATA)>
<!ELEMENT user (#PCDATA)>

<!ELEMENT predecessor (#PCDATA)>
<!ELEMENT force_prefix (#PCDATA)>

<!ELEMENT interfaces (interface*)>
<!ELEMENT interface (id, aggregated_prefixes?, available_prefixes?, low_priority_prefixes?,
    upstream?, mac?, lla?, ipv4_address?, ipv6_addresses?, reserved_metric?, force_prefix?)>
<!ELEMENT upstream (#PCDATA)>
<!ELEMENT mac (#PCDATA)>
<!ELEMENT lla (#PCDATA)>
<!ELEMENT ipv4_address (#PCDATA)>
<!ELEMENT ipv6_addresses (ipv6_address*)>
<!ELEMENT ipv6_address (#PCDATA)>
<!ELEMENT reserved_metric (#PCDATA)>
<!ELEMENT aggregated_prefixes (prefix*)>
<!ELEMENT low_priority_prefixes (prefix*)>

<!ELEMENT networks (network*)>
<!ELEMENT network (id, autoconf?, dhcpv6?, predecessor?, ipv4_address?, ipv6_prefixes?,
    force_prefix?)>
<!ELEMENT autoconf (#PCDATA)>
<!ELEMENT dhcpv6 (#PCDATA)>
<!ELEMENT ipv6_prefixes (ipv6_prefix*)>
<!ELEMENT ipv6_prefix (#PCDATA)>
```

This file contains all the information we need on the network devices or subnets, and how we want the new IPv6 network to behave in terms of addressing.

This file allows to tune the execution of the security engine by configuring various constraints (exclude and force prefixes, links end points addressing...), and give the information missing in the DOT file (site BGP AS and neighbors, do we write the configuration in the devices memory ?).

It also details the information about the routers interfaces (ID, IPv4 and MAC addresses, existing IPv6 addresses or prefixes assigned in an earlier execution of the engine...), the routing protocol they use, their type (Cisco or Quagga), and permits to set constraints at the router level (force prefixes, force predecessor).

Finally, as for the routers, it gives more detailed information about the networks.

Chapter 3

Implementation

We implemented the specified Transition Engine using the Python language. The implementation has been done with Python version 2.5.

We will not detail the algorithms implementation, as it is a python expression of the algorithms detailed in D2.1 and in figure 2.2.

The tool has been named *6Te@*, The IPv6 Transition Engine.

3.1 Graph Handling

The DOT file is parsed using the Boost Graph Lib [1] and its python bindings [2], and Python objects are generated accordingly. This library includes several algorithms, especially for shortest path calculation. In our implementation, we chose to use the Bellman-Ford algorithm [3].

We detail here the installation procedure we followed to install version 1.39.0 of the library and its bindings, and all the dependencies required by the engine, on a freshly installed Ubuntu 9.04, which has been then updated safely to Ubuntu 9.10.

First, set environment variables:

```
export BOOST_ROOT=path/to/boost_boost_1_39_0
export EXPAT_INCLUDE=/usr/include
export EXPAT_LIBPATH=/usr/lib
```

Install the dependencies

```
apt-get install libexpat-dev libbz2-dev bjam zlib1g-dev python2.5 \
    build-essential python2.5-dev python2.5-ipy python2.5-4suite-xml \
    python2.5-pexpect graphviz
```

Make sure python2.5 is the default python version and set it as default python version in Boost Graph Library

```
echo "using python : 2.5 ;" >> $BOOST_ROOT/tools/build/v2/user-config.jam
ln -sf /usr/bin/python2.5 /usr/bin/python
```

Build and install Boost Graph Library

```
cd $BOOST_ROOT
bjam --toolset=gcc -a --without-mpi install
```

[1] http://www.boost.org/libs/graph/doc/index.html
[2] http://www.osl.iu.edu/~dgregor/bgl-python/
[3] http://en.wikipedia.org/wiki/Bellman-Ford_algorithm

Build and test python module

```
cd $BOOST_ROOT/libs/python
bjam --toolset=gcc -a --without-mpi test
```

Build and install BGL-Python

```
cd bgl-python
patch -p1 . <./sparse_ordering_redefinition.patch
bjam --debug-configuration --toolset=gcc --without-mpi
bjam -a --python-libdir=/usr/local/python2.5/site-packages/ \
    --exec-prefix=/usr --without-mpi --toolset=gcc \
    --libdir=/usr/local/lib install
```

Where *sparse_ordering_redefinition.patch* is:

```
--- bgl-python.orig/boost/graph/detail/sparse_ordering.hpp      2009-08-07 14:46:33.000000000 +0200
+++ bgl-python/boost/graph/detail/sparse_ordering.hpp   2009-07-16 11:41:05.000000000 +0200
@@ -122,7 +122,7 @@
     public:
       typedef typename Sequence::iterator iterator;
       typedef typename Sequence::reverse_iterator reverse_iterator;
-      typedef queue<Tp,Sequence> queue;
+      //typedef queue<Tp,Sequence> queue;
       typedef typename Sequence::size_type size_type;

       inline iterator begin() { return this->c.begin(); }
```

There is also a problem with the python pexpect module used to remotely configure the routers. The prototype crashes because of the usage of the function *flush* of an *OutputStream* object which supposely does not exist, whereas it does. When executing the same code as in pexpect directly in the interpreter, it works fine. As the python documentation states that the *flush* method of an *OutputStream* object does not do anything, we simply commented it:

```
--- pexpect.py  2009-08-07 15:04:41.000000000 +0200
+++ /usr/lib/python2.5/site-packages/pexpect.py 2009-08-05 11:55:27.000000000 +0200
@@ -1484,7 +1484,7 @@

         # Flush the buffer.
         self.stdout.write (self.buffer)
-        self.stdout.flush()
+        #self.stdout.flush()
         self.buffer = ''
         mode = tty.tcgetattr(self.STDIN_FILENO)
         tty.setraw(self.STDIN_FILENO)
```

3.2 XML Parsing and Serialization

As we can see in the dependencies installed, XML parsing and serialization is performed via the 4Suite [4] framework. Particularly, the parsing of the XML configuration file relies on the XPath module.

[4] http://www.4suite.org/

3.3 Remote Configuration

Once all the prefixes and addresses are assigned, the program remotely configures the devices to make IPv6 available on the network. This is done by using Telnet and SSH remote configuration interfaces, and is perform on Quagga [5] and Cisco routers. The routing protocol used at the moment is RIPng [3].

SSH

SSH connection is used at the moment only for copying the initial RIPng configuration to the GNU/Linux node running Quagga, and then restart the service. This feature will be improved to support remote configuration on Cisco routers.

SSH connection are implemented by using the pexpect library [6]. The primitives available are:

- scp_to_remote
- scp_from_remote
- start_service
- stop_service
- restart_service

Telnet

Telnet is used for remote configuration of Quagga and Cisco routers. It is implemented by using the Python telnetlib [7]. All basic commands have been implemented: enable, configure terminal, write terminal/memory, end, exit... The primitive *write_read* sends a command to the router and returns the result. It is used for the configuration of the interfaces or the routing protocol.

If the configured router is a Cisco, a single Telnet connection is sufficient. But if the remote node is a Quagga router, the program connects first to the zebra daemon for configuring the addresses, and then reconnects to the ripng (or any other routing protocol) daemon to configure the routes.

3.4 Routing Protocols

In order to ensure routing on an IPv6 network, different routing protocols are available. In this section, we present the ones the tool implements. In order to choose the routing protocol used on a router, the attribute *routing* is used in the configuration file, on the tag *id* of the router description.

First of all, a very common solution for small networks with few routers and subnets is the usage of static routes. For each router, we get the neighbors, and set a route to their *routed_prefix*. the default route is set with the predecessor as next hop.

Then, we implemented several common routing protocols. The protocols implemented are RIPng [3], OSPFv3 [1] and BGP [4]. We took into account the specificities and possibilities of each protocol and implementation in the routers. The border router always initiates the default route for the site. Some parameters have been arbitrarily chosen, such as the process ID for RIPng and BGP, while some other ones can be configured in the configuration file, like the BGP Autonomous system (AS) or external neighbors.

During the implementation, we encountered several issues, and only one could not be solved. Following an update in the Quagga distribution (version 0.99.7), the statement *default-information originate* which is originating the default route in the AS is not working [8]. The bug is solved in the CVS since the 14th

[5] http://www.quagga.net/
[6] http://pexpect.sourceforge.net/
[7] http://docs.python.org/lib/module-telnetlib.html
[8] http://bugzilla.quagga.net/show_bug.cgi?id=370

June 2007, but this correction has not yet been integrated in official releases. Older releases of Quagga do not have this problem, 0.99.7 is the only one.

Finally, we ensured than the redistribution between all these protocols is working. By default, all protocols are redistributing static routes and connected networks. If a neighbor is using a different routing protocol, this routing protocol is enabled automatically on the router, and the redistribution commands are added. We also make sure that only one router (the one closer to the border) redistributes the protocol.

3.5 Graphical Interface

We implemented a graphical interface for the tool by using GTK 2.6.

The GUI is composed of a window containing a Menu bar, a tools bar and several tabs.

The toolbar contains a button that permits to show/hide the console, which shows the logs of the engine (namely stdout and stderr).

The tabs are composed of:

Initial Configuration Load and apply the configuration files, see figure 3.1

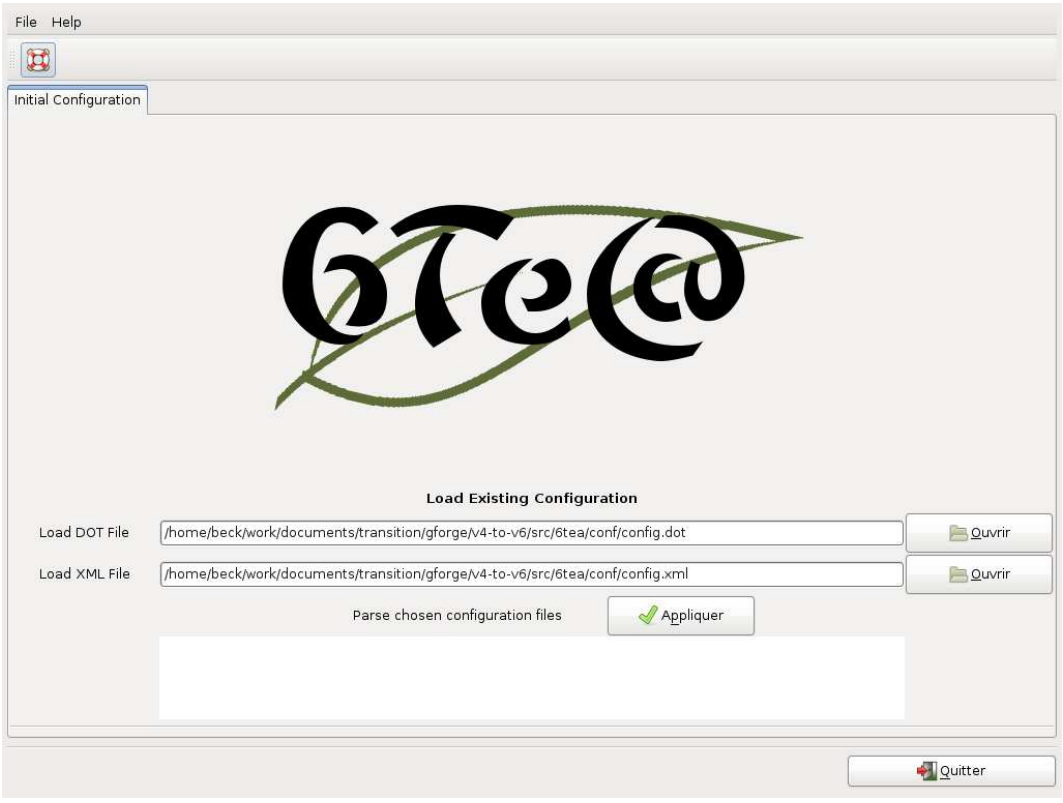

Figure 3.1: 6Te@ Welcome Screen

Initial Network A graphical representation of the initial network, see figure 3.2

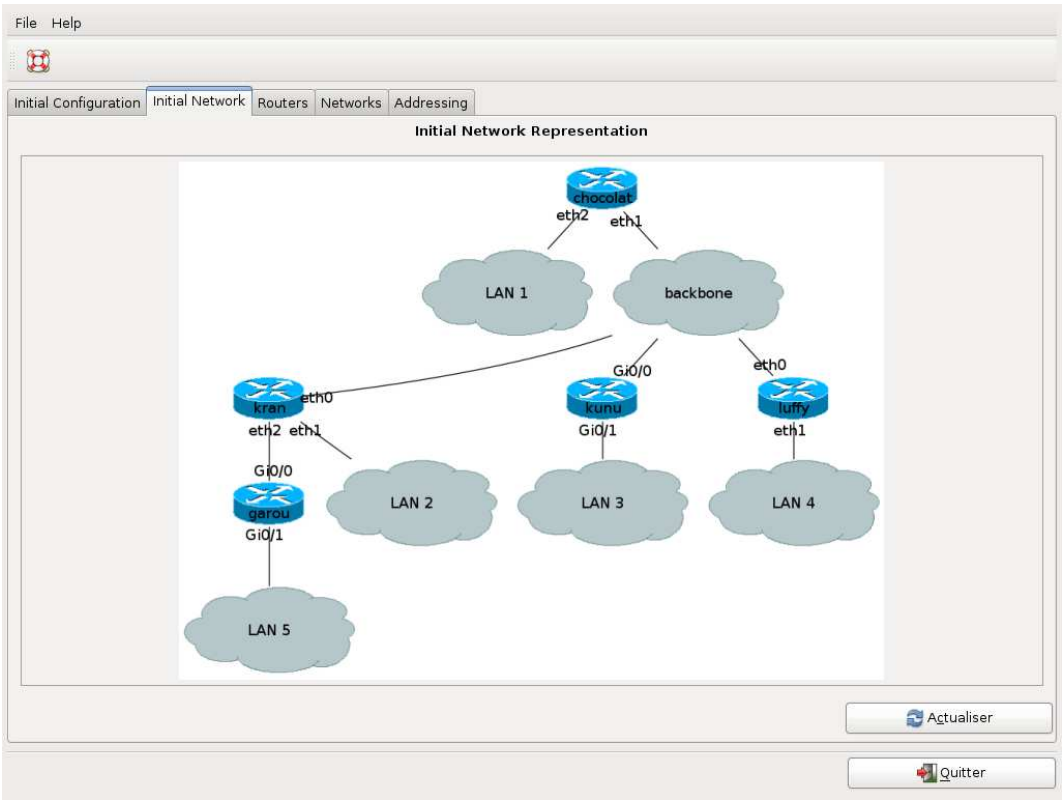

Figure 3.2: Initial network representation

Routers Details about the loaded routers, see figure 3.3

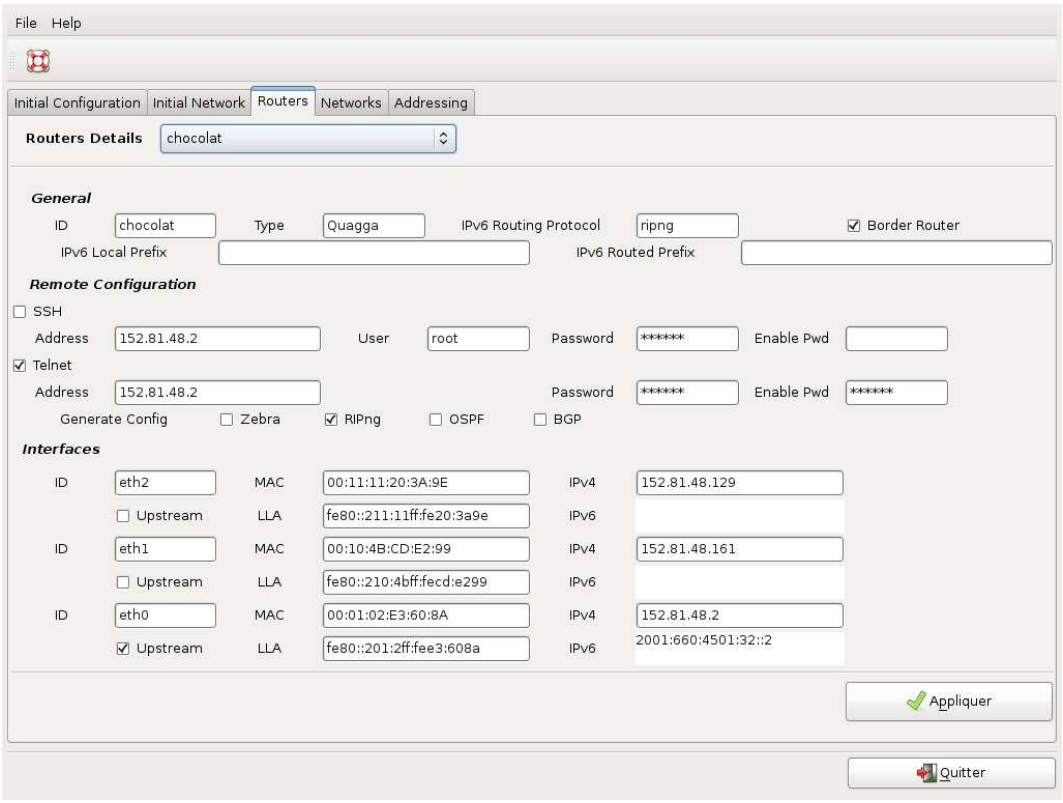

Figure 3.3: Router details

Networks Details about the loaded networks, see figure **??**

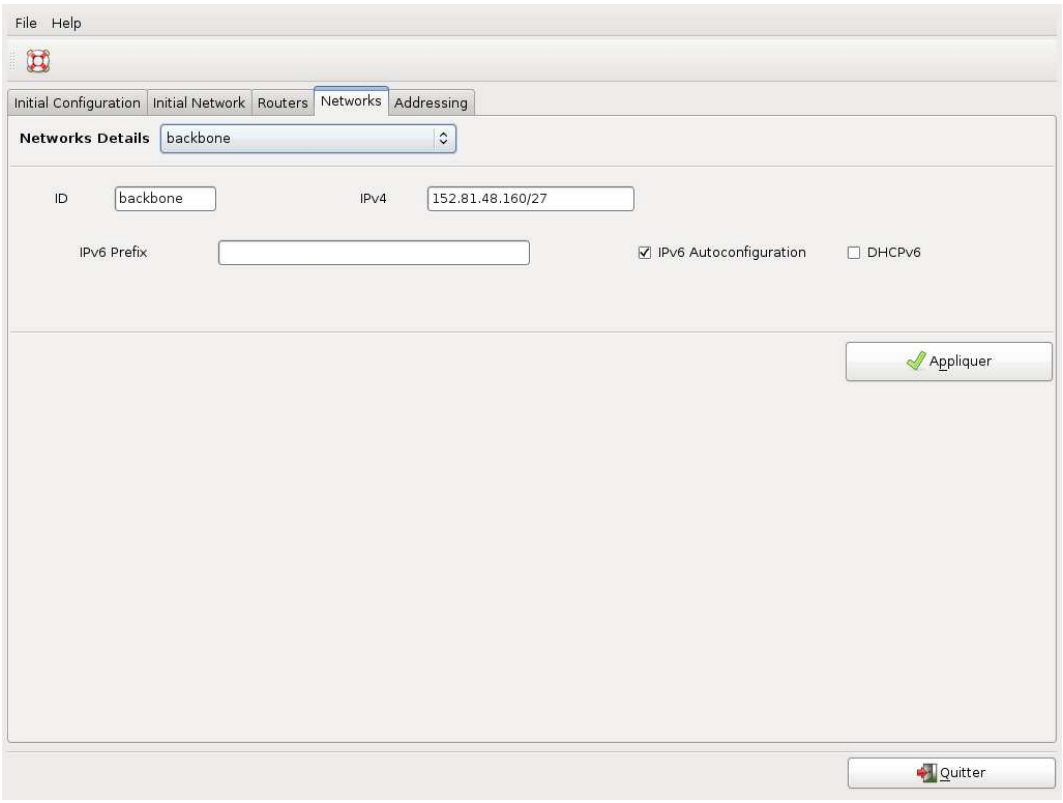

Figure 3.4: Network details

Addressing The Transition Engine, set the site prefix and run the algorithms, validate the output and configure the devices, see figure 3.5

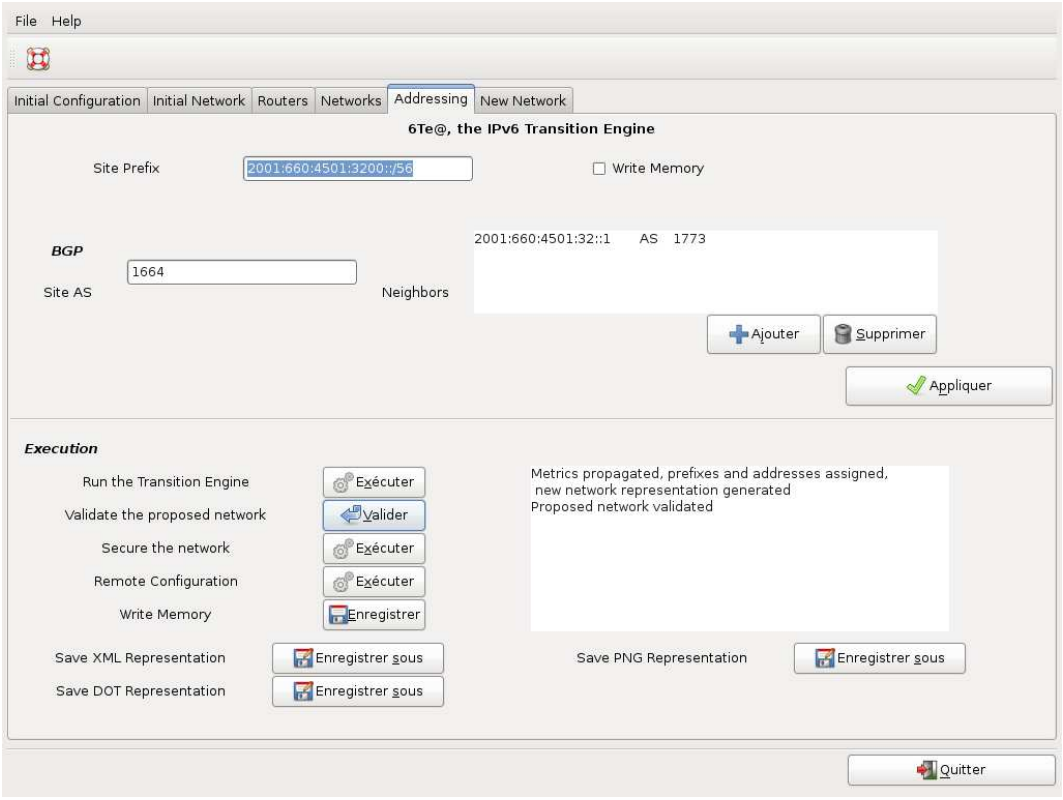

Figure 3.5: Transition Engine

New Network A graphical representation of the proposed addressing plan, see figure 3.6

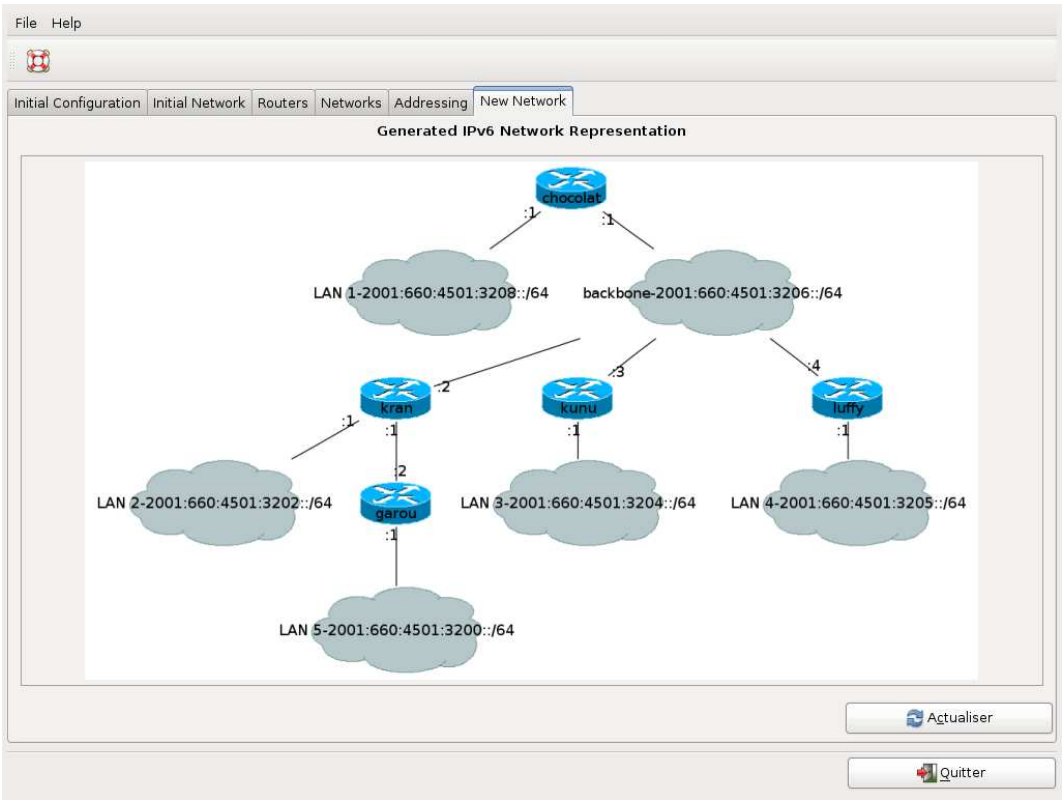

Figure 3.6: New network representation

3.6 Tool Distribution

3.6.1 VMWare Image

We retrieved from `http://vmplanet.net/node/95` an Ubuntu 9.04 VMWare image that we modified by following the installation procedure, in order to have a binary distribution of the tool. This virtual machine has been updating to the latest version of Ubuntu.

The latest version of the transition engine has been installed in it, and a script on the desktop permits to run the tool. Several configuration files have been created, and allow to run many different scenarios in order to test the tool.

To connect, use the user *ipv6* and the password *transition*. The image is quite big (2GB) and can be uploaded on put on a share web location for download upon request.

3.6.2 GForge

The tool is hosted in the INRIA forge [9] under the project *v4-to-v6*. At the moment, the project is private, and we only use the subversion repository for the code and documentation.

When the licence and dissemination mechanisms have been discussed, the project may become public and/or be migrated to SourceForge [10].

[9] `https://gforge.inria.fr/projects/v4-to-v6/`
[10] `http://www.sf.net`

Chapter 4

Conclusion

In this report, we presented the specification and implementation of the Transition Engine, prototype implementation and a first step towards a "one-click" transition tool.

The resulting prototype, containing the code for task 3 on security is composed of 15 875 Python lines (counting empty lines and comments). The tool has been validated and tested on all scenarios defined in D1.1. All constraints defined in D1.2 and algorithms defined in D2.1 have been implemented successfully in it.

Bibliography

[1] R. Coltun, D. Ferguson, and J. Moy. OSPF for IPv6. RFC 2740 (Proposed Standard), December 1999. Obsoleted by RFC 5340.

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

[3] G. Malkin and R. Minnear. RIPng for IPv6. RFC 2080 (Proposed Standard), January 1997.

[4] P. Marques and F. Dupont. Use of BGP-4 Multiprotocol Extensions for IPv6 Inter-Domain Routing. RFC 2545 (Proposed Standard), March 1999.

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