

RÉPUBLIQUE DU BÉNIN

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE  
LA RECHERCHE SCIENTIFIQUE



UNIVERSITÉ D'ABOMEY-CALAVI (UAC)

CENTRE DE FORMATION ET DE RECHERCHE EN  
INFORMATIQUE (CEFRI - UAC)



MÉMOIRE DE FIN DE FORMATION

POUR L'OBTENTION DU

DIPLOME DE LICENCE EN INFORMATIQUE GÉNÉRALE

THEME :

---

# **Solution libre de supervision réseau: initialisation automatique du NAV en utilisant SNMP**

---

Encadrant :

Monsieur Toussaint OKEY

Année Académique : 2011-2012

2<sup>ème</sup> Promotion



RÉPUBLIQUE DU BÉNIN

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE  
LA RECHERCHE SCIENTIFIQUE



UNIVERSITÉ D'ABOMEY-CALAVI (UAC)



CENTRE DE FORMATION ET DE RECHERCHE EN  
INFORMATIQUE (CEFRI - UAC)

MÉMOIRE DE FIN DE FORMATION

POUR L'OBTENTION DU

DIPLOME DE LICENCE EN INFORMATIQUE GÉNÉRALE

THEME :

---

# **Solution libre de supervision réseau: initialisation automatique du NAV en utilisant SNMP**

---

Présenté & soutenu par :  
**AROUNA Hospice Alfred** Le jeudi 20 décembre 2012

Devant le jury composé de :

**Président :** Dr **ADEDJOUMA** Semiyoun A., Enseignant-Chercheur à l'EPAC

**Membres :**

1°) Dr. **Baloitcha** Ezinvi, Coordonnateur des études au CEFRI

2°) Mme **Gnonlonfoun** Miranda, Enseignant au CEFRI

**Année Académique : 2011-2012**

**2<sup>ème</sup> Promotion**

# Résumé

L'assurance de la qualité de service (*Quality of Service (QoS)*) fournie aux utilisateurs est un élément primordial dans la gestion des réseaux. Cette assurance de la qualité de service est possible grâce à la Supervision qui permet d'être actif et surtout d'être pro-actif. **La supervision permet de résoudre les problèmes du réseau avant qu'ils ne surviennent, et même en cas de problème, de pouvoir avoir la bonne information (source du problème) et de prendre la meilleure solution.**

La présente étude porte sur notre solution à une des faiblesses d'une application de supervision : le *Network Administration Visualized (NAV)*. **Notre apport est une implémentation en python des études les majeures relatives à la découverte automatique des équipements et de la Typologie d'un réseau en utilisant le standard *Simple Network Management Protocol (SNMP)*.** Cette implémentation permet un début d'automatisation de l'initialisation de la base de données des équipements du réseau géré par leNAV : l'étape du *seedDB* se trouve essentiellement simplifiée.

**Mots clés :** *SNMP, supervision, NAV, python, réseau, sécurité, logiciel libre.*

# Abstract

Ensuring the Quality of Service (QoS) provided to users is an essential criteria in network management. Monitoring is the main base to have correct QoS. Monitoring allow network administrator to be active and especially to be pro-active. **Monitoring can help to solve network problems before they occur, and even if something goes wrong, to be able to get the right and accurate information (root cause of the problem) and take the best solution.**

The present study focuses on our solution to one of the weakness of one monitoring application: NAV. **We provided an implementation in python of the most important studies on the automatic discovery of network topology using the standard *SNMP*.** This implementation introduce the automation of initializing the database of network devices managed by the NAV: the step of *seedDB* is essentially simplified.

**Key words:** *SNMP, monitoring, NAV, python, network, security, free software.*

# Table des matières

Résumé / Abstract	i
Glossaire	i
Remerciements	vi
Introduction	1
<b>1 Le Network Administration Visualized</b>	<b>3</b>
I Présentation rapide	3
II Limites de NAV	4
1) Domaines d'utilisation	4
2) Fonctionnalités	4
<b>2 Initialisation automatique de la base de données</b>	<b>5</b>
I Initialisation automatique de la base de données grâce à SNMP	5
1) Études relatives à la découverte de la typologie réseau avec SNMP	5
II Les algorithmes	6
1) Starting Point	7
2) Device Discovery	8
3) Device grouping	10
4) Device type discovery	11
5) Create Bulk format	13
6) Store in Db	13
III Implémentation, tests et résultats	13
1) Structuration	13
2) La classe SeebDB	15
3) La classe GetIndirectNextHop	17
4) La classe DeviceGrouping	17
5) La classe GetLocalNetAddress	18
6) Évaluation du code	18
7) Test dans un environnement virtuel	18
<b>Bibliographie</b>	<b>24</b>

# Liste des algorithmes

2.1	Starting Point . . . . .	8
2.2	Device Discovery . . . . .	9
2.3	Device Grouping . . . . .	11
2.4	Device Type . . . . .	12

# Table des figures

0.1	Mode de fonctionnement de SNMP. . . . .	2
2.1	Modèle de réseau (créé avec NetKit) à découvrir par le seedDB . . . . .	19
2.2	Fenêtre de confirmation de l'importation des données du fichier des équipements du réseau . . . . .	22
2.3	Résultat de l'importation du fichier des équipements du réseau . . . . .	23

# Liste des tableaux

2.1	Valeur de la variable sysServices du MIB-II (RFC 1213) . . . . .	12
-----	--	----

# Glossaire

**Agent (SNMP) :**

Composant logiciel ou matériel qui gère les MIBs d'un matériel. Les agents répondent aux requêtes du Manager ou lui envoie un trap en cas d'alerte.

**BER :**

Basic Encoding Rules

**IP :**

Internet Protocol

**Manager :**

Le manager est l'élément actif de la supervision. Il collecte les informations des hôtes en interrogeant les agents. Il reçoit aussi les traps provenant des agents.

**MIB :**

Management Information Base

**NAV :**

Network Administration Visualized

**NetKit :**

Netkit permet de créer plusieurs machines virtuelles sur le même hôte. Chaque machine virtuelle est reliée à un ou des domaines de collisions virtuels permettant ainsi les communications entre-elles. Chaque machine virtuelle peut jouer le rôle de PC, routeur ou switch.

**NMS :**

Network Management System

**OID :**

Object Identifier

**QoS :**

Quality of Service

**RFC :**

Request for Comments

---

**seedDB :**

L'étape du *Seed Database* permet de fournir les informations importantes pour NAV pour pouvoir parcourir et découvrir le réseau. En effet, après installation et démarrage de NAV, NAV ne fait rien si les paramètres n'ont pas été fournis à la base.

**SNMP :**

Simple Network Management Protocol

**Supervision :**

Superviser consiste à indiquer/modifier l'état d'un système informatique ou d'un hôte. La supervision est donc la fonction qui permet de remonter/éditer les informations techniques relatives à un hôte ou à un ou des réseaux. En d'autres termes, la supervision de réseaux peut être définie comme l'utilisation de ressources réseaux (matériel et logiciel) adaptées dans le but d'obtenir des informations (en temps réel ou non) sur l'utilisation ou la condition des hôtes dans le réseau afin d'assurer un niveau de service garanti, une bonne qualité et une répartition optimale et de ceux-ci.

**Typologie :**

La topologie d'un réseau peut se définir comme l'étude de la structure du réseau et des interconnexions entre hôtes. La typologie réseau peut être de type *link layer topology*, *network layer topology* ou *Internet topology* et *overlay topology*.

**UDP :**

User Datagram Protocol



# Remerciements

Nos premiers remerciements vont à l'endroit notre encadreur : Mr. Toussaint OKEY, enseignant au Centre de Formation et de Recherches en Informatique (CeFRI) ; pour sa disponibilité et ses fructueuses orientations.

Nos remerciements s'adresse aussi Mr Musa BALTA, du Computer Engineering Department, Faculty of Computer and Information Science du Sakarya Üniversitesi en Turquie. Mr Musa BALTA est l'auteur de l'article *The Discovery of Enterprise Network Topology Created in a Virtual Environment with SNMPv3* paru dans le *TOJSAT : The Online Journal of Science and Technology* d'Avril 2012. Aussi Mr Musa BALTA ; suite à notre demande, a mis gracieusement à notre disposition l'article *SNMP-based enterprise IP network topology discovery* de Pandey et. al. Cet article est la référence actuelle en matière de découverte de la topologie d'un réseau en utilisant *SNMP*.

Avant de finir, nous tenons à reconnaître la qualité de l'enseignement reçue et à remercier toute l'administration de CeFRI.

Nous ne saurions finir sans remercier les membres du jury pour l'importance à nous accorder pour l'évaluation de notre travail.

# Introduction

Une entreprise moderne dépend d'un réseau informatique dont la taille est dans une croissance permanente. Il est primordial pour cette entreprise de garantir une qualité de service à travers son réseau informatique. C'est dans cette perspective qu'est apparu, il y a maintenant une vingtaine d'années, le concept de supervision de réseau.

Que ce soit le grand réseau d'un opérateur ou d'une université ou bien le petit réseau local d'une petite entreprise, la supervision doit pouvoir apporter des outils performants, adaptables aussi bien à la taille des réseaux qu'à leur grande diversité technologique.

D'un point de vue pragmatique, la supervision réseau a pour objectif :

- La **Pro-Activité** pour anticiper sur les problèmes.
- La **Réactivité** suite à un incident.
- La **Localisation d'un problème** afin de prendre la meilleure décision.

Un des éléments clé de la supervision réseau est la possibilité d'avoir un œil sur le réseau. Quel que soit le type d'administrateur (expérimenté ou non), la vue graphique ou topologie du réseau donne un outil clair et facile pour un début d'analyse du réseau.

Vu l'importance de la mise à disposition des administrateurs de la typologie du réseau, de nombreuses études ont porté sur la question ; certaines en utilisant le *standard*<sup>1</sup> *SNMP*. Nous porterons notre attention sur les études portant sur la détection de la typologie au sein d'une organisation.

*SNMP* est de nos jours le *standard* de supervision. Créé dans le but de superviser les premiers routeurs lors de l'explosion d'Internet, *SNMP* a connu diverses versions pour en être depuis 1999, à la version 3. *SNMP* présente les caractéristiques suivantes :

---

<sup>1</sup>Cycle des statuts des RFC : *proposed*, *draft* puis *standard*. Si un RCF a été mis à jour par un autre son statut devient *historical*. De même si un RFC n'est pas encore accepté comme standard, il reste à l'état de *experimental*.

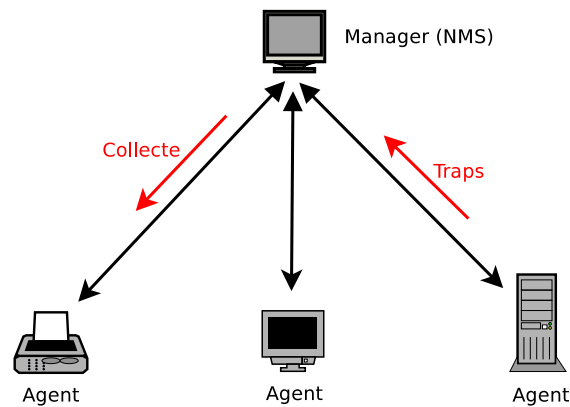


FIGURE 0.1 – Mode de fonctionnement de SNMP.

- Fonctionne de manière asymétrique : *Agent (SNMP)* et *Manager* (Request for Comments (RFC) 1157).
- Utilise des opérations en écriture (*set*) et en lecture (*get*).
- Utilise UDP et le *Basic Encoding Rules (BER)* pour le transport.
- Introduit le concept de *Management Information Base (MIB)* (RFC 1213).
- Identifie chaque MIB par son *Object Identifier (OID)*.

Le présent mémoire porte de manière générale sur l'implémentation en python de la découverte des équipements d'un réseau d'une organisation en utilisant *SNMP*. Cette découverte automatique des équipements du réseau servira de source d'importation des informations sur les équipements du réseau dans une solution libre de supervision : le NAV.

Le présent document se résume à notre apport pour l'*initialisation automatique de la base de données* du NAV. Il est composé de deux chapitres :

### Présentation théorique de la solution

Le premier chapitre présente brièvement le NAV. NAV intègre bien d'autres solutions libres dont les plus importants sont Cricket, PostgreSQL, OpenStreetMap. Ce chapitre se termine sur la présentation de quelques limites du NAV disponible sur la feuille de route du projet. En particulier, l'accent est mis sur la fonctionnalité de l'*autodiscovery-wizard*. Cette fonctionnalité permettra de fournir de manière automatique au NAV la base de données des équipements du réseau. Un début d'implémentation de cette fonctionnalité est donnée dans le cinquième chapitre.

### Initialisation automatique de la base de données

Le dernier chapitre constitue notre apport pour l'implémentation de l'*autodiscovery-wizard*. Cette implémentation permet l'*initialisation automatique de la base de données* du NAV. Plus besoin d'ajouter un à un les équipements du réseau ou bien de les lister soi-même dans un fichier. Notre solution en s'inspirant des études en matière de découverte de réseau sur la base de *SNMP* permet de découvrir tous les équipements du réseau et de créer un fichier prêt pour importation dans la base de données du NAV. Après une présentation des algorithmes, l'implémentation en code python est expliquée, suivie des tests et résultats dans un environnement virtuel grâce à *NetKit*.

# Le Network Administration Visualized

## I Présentation rapide

---

Le **Network Administration Visualized** est une suite logicielle de pointe pour superviser les réseaux informatiques de grande taille.

NAV présente les caractéristiques suivantes :

- Originaire de la Norvège : *NTNU (Norwegian University of Science and Technology)*.
- Licence GPL v2.
- Sous la responsabilité du *UNINETT (the Norwegian research network)*.
- Utilisation par de nombreuses universités dans le monde.
- Programmation en Python.

### Intérêt du NAV

NAV est une collection intégrée de solutions libres ayant déjà fait leurs preuves. Ce qui fait de NAV, la seule solution distribuée offrant de nombreuses fonctionnalités.

Comme fonctionnalités majeures du NAV, nous avons :

- *Geographical Map* : typologie sur une carte *OpenStreetMap*.
- *Network Weather Map* : état d'utilisation des liens (topologie).
- *Machine Tracker* : historique du couple adresse IP et adresse MAC pour chaque équipement.
- *Mac Watch* : surveillance des adresses MAC du réseau.
- *Layer 2 trace* : trace du parcours d'un paquet d'une source à une destination.
- *Arnold* : blocage des ports des routeurs et switches en les mettant en quarantaine.
- *Maintenance Tasks* : Exclusion d'un équipement du mécanisme de supervision sans enlever l'équipement du réseau.
- *Radius Accounting* : Suivi à la trace des utilisateurs nomades.

Plus d'informations sont disponibles sur le wiki<sup>1</sup> du projet.

## II Limites de NAV

---

### 1) Domaines d'utilisation

NAV est certes une suite logicielle de pointe pour superviser les réseaux informatiques de grande taille mais NAV ne peut *évidemment* pas tout faire.

En effet, NAV est une solution de supervision réseau et non de *configuration* réseau ; et ceci à deux exceptions près :

- L'utilisation de SNMP pour bloquer les ports avec l'outil *Arnlod*.
- L'utilisation de SNMP pour configurer les ports des switchs et les Vlans avec l'outil *PortAdmin*

NAV n'est pas non plus une solution magique qui va découvrir *tous* les problèmes dans le réseau. NAV essaie de découvrir tous les problèmes *importants* sur le réseau et non tous les problèmes. Il s'en va dire que d'autres outils doivent être utilisé pour découvrir ces problèmes. L'exemple classique est celui du statut des équipements. La minute ou la seconde après la vérification du statut de l'équipement par NAV, ce dernier peut changer de statut. Avant la prochaine vérification par NAV, l'équipement peut revenir à son état originel. Ce changement d'état ne sera pas détecter par le NAV.

Aussi NAV ne donne pas des solutions toutes faites aux problèmes découverts. NAV ne donne que des alertes et des pistes de recherche de solution. D'autres outils d'investigation doivent être utilisés pour trouver la cause fondamentale du problème et ainsi apporter la meilleure solution.

### 2) Fonctionnalités

NAV donne des informations sur la charge du trafic, pas sur le sens du trafic, ni les détails sur le trafic (http, ftp, etc). En d'autres termes, NAV ne permet pas de connaître la source, ni la destination d'un paquet comme les sites les plus visités par exemple. D'autres outils libres tels *Netflow Sensor (NfSen)*<sup>2</sup> le font déjà correctement. Une documentation sur l'usage de Netflow est donné par *The Swiss Education & Research Network*<sup>3</sup>

NAV à certes de nombreuses fonctionnalités mais de l'avis même de ses concepteurs, il reste encore de nombreuse fonctionnalités potentielles à ajouter au NAV. La liste complète de ses fonctionnalités est disponible sur le *launchpad*<sup>4</sup>.

En autre fonctionnalité, NAV ne fait pas au premier abord de la découverte du réseau. Après installation de NAV, il faut initialiser la base de données en y ajoutant les équipements du réseau : c'est l'étape du *seedDB*.

Une automatisation de cette étape permettrait de simplifier l'utilisation du NAV : Il suffirait d'installer NAV ; de lancer le script de découverte puis deux minutes après avoir l'état du réseau.

Nous nous proposons dans le chapitre suivant de donner un début de simplification de l'étape du seedDb.

---

<sup>1</sup><http://nav.uninett.no/navtechdoc>

<sup>2</sup><http://nfsen.sourceforge.net/>

<sup>3</sup><http://meetings.ripe.net/ripe-50/presentations/ripe50-plenary-tue-nfsen-nfdump.pdf>

<sup>4</sup><https://blueprints.launchpad.net/nav>

# Initialisation automatique de la base de données

## I Initialisation automatique de la base de données grâce à SNMP

*L'initialisation automatique de la base de données grâce à SNMP constitue notre apport. En effet, avec le modèle actuel proposé par NAV, il faudra actuellement soit ajouter un à un les équipements du réseau, soit remplir un formulaire où chaque ligne correspond à chaque équipement au niveau de l'interface d'administration. Cette tâche s'avère très fastidieuse d'où la nécessité de l'automatiser.*

C'est dans cette perspective que la fonctionnalité *Autodiscovery wizard for NAV* a été prévu sur la feuille de route des futurs développements<sup>1</sup>

Nous proposons d'écrire un script permettant de découvrir tous les hôtes du réseau supportant SNMP à partir d'un point de départ. Une fois ces hôtes découverts, il faudra les mettre sous un format acceptable par NAV et les insérer dans la base de données.

### 1) Études relatives à la découverte de la typologie réseau avec SNMP

De nombreux études ont porté sur la découverte de la topologie physique et logique des réseaux informatiques. Nous ne citerons ici que les quatre dernières études en la matière :

1. Pandey S, Choi M, Lee S, Hong J. <sup>2</sup>. *IP Network Topology Discovery Using SNMP*. International Conference on Information Networking (ICOIN'09). 23 33-37 2009
2. Yang Xiao. *Physical Path Tracing for IP Traffic using SNMP*. BBC Research White Paper WHP 188. 2010
3. Pandey S, Choi M, Won Y, Hong J. *SNMP-based enterprise IP network topology discovery*. *International Journal of Network Management* 21 (3) 169-184. 2011

<sup>1</sup><https://blueprints.launchpad.net/nav/+spec/autodiscovery-wizard>

<sup>2</sup> Suman Pandey, Mi-Jung Choi, Sung-Joo Lee, James W. Hong Dept. of Computer Science and Engineering, POSTECH, Korea

4. Musa Balta, Ibrahim Özçelik. *The Discovery of Enterprise Network Topology Created in a Virtual Environment with SNMPv3*. The Online Journal of Science and Technology (TOJSAT) Volume 2, Issue 2, 64-70, Avril 2012.

Toutes les trois dernières études se basent sur les résultats de la première étude de la liste ci-dessus : l'étude sur *IP Network Topology Discovery Using SNMP*.

De ces études, la typologie réseau découverte est de type physique et logique soit le *link layer topology* et le *router level* du *Internet topology* d'une organisation ou d'un AS. C'est donc une découverte de réseau interne à une organisation ou un AS.

La typologie d'un réseau désigne l'organisation des éléments (équipements et liens) du réseau et les interactions physiques et logiques entre ces divers éléments.

La typologie logique se réfère à la structuration ou division logique du réseau en sous réseaux.

La typologie physique désigne les communications ou connexions visibles entre hôtes sur des ports grâce à des liens (de transmission).

Dans le cadre de l'automatisation de l'initialisation de la base de NAV, nous n'implémenterons que certains aspects des résultats de ces études. En effet, la fonctionnalité offerte par le `ipdevpoll` permettent déjà de faire de la mise à jour des informations dans la base. Notre apport permettra de faire des insertions en masse des informations minimales sur chaque équipement découvert dans le réseau.

## II Les algorithmes

---

Se basant sur les études citées dans la section précédente, nous proposons un algorithme pour initialiser la base de données conformément au *minimum requirement* c'est-à-dire le *Level 1 : Minimum requirements*<sup>3</sup> du *seed database*.

L'algorithme proposé par Pandey and et. al. (2011) se déroule en étapes successives :

1. Take network information inputs
2. Device discovery
  - a. Device discovery using next hop mechanism
  - b. Device discovery using ARP cache entries
3. Device type discovery
4. Device grouping based on IP address
5. Connectivity discovery
  - a. L2 to L2 connectivity
  - b. L2 to L3 connectivity
  - c. L3 to L3 connectivity
  - d. L2 and L3 to end host connectivity
6. Subnet discovery and connectivity discovery in subnet
7. VLAN discovery and connectivity discovery in VLAN

Un projet d'algorithme est aussi précisé sur la page du wiki<sup>4</sup>.

---

<sup>3</sup><http://nav.uninett.no/seedessentials>

<sup>4</sup><http://metanav.uninett.no/devel:blueprints:autodiscovery-wizard>

La première étape est la récupération des paramètres nécessaires pour la découverte des hôtes du réseau. Comme paramètres proposés nous avons :

- La *communauté SNMP* pour l'accès en lecture des MIBs.
- Un préfixe ou une liste des préfixes (plage d'adresses IP de la découverte réseau).
- L'adresse IP du routeur de départ.
- Le nom de l'organisation auquel appartient les adresses IP (les équipements).

Une fois les paramètres donnés au script, les étapes suivantes sont proposées :

1. Retrieve all active arp records
2. For each of these IP addresses
  - a. If the IP address exists in NAV - skip
  - b. Tf the address answers to SNMP : new device found!
  - c. Get system.syslocation
  - d. Set orgid to the supplied orgid
  - e. Decide category
  - f. Make new row to bulk file (format: roomid:ip:orgid:catid:ro )

**Notre algorithme est une mise en commun des quatre premières étapes proposées par Pandey and et. al. (2011) et de l'algorithme proposé pour la fonctionnalité de découverte des équipements du réseau. Nous ne prenons pas en compte la découverte de la connectivité (topologie) et des VLAN. `ipdevpoll` s'occupe déjà de ces deux dernières fonctionnalités dans NAV.**

Notre algorithme de découverte du réseau se présente en six étapes suivantes :

1. Starting Point (Take network information inputs)
2. Device discovery
  - a. Device discovery using next hop mechanism
  - b. Device discovery using ARP cache entries
3. Device grouping based on IP address
4. Device type discovery
5. Create Bulk format
  - a. Set default roomid.
  - b. Set default orgid
  - c. Set category (type discovery)
  - d. Make new row to bulk file (format: roomid:ip:orgid:catid:ro )
6. Store in Db

## 1) Starting Point

Avec le mode de fonctionnement asymétrique de *SNMP*, *Agent (SNMP)* et *Manager* utilisent comme protocole de transport l' *User Datagram Protocol (UDP)*.

Or *UDP* (couche 4) se base sur le protocole *Internet Protocol (IP)* (couche 3). Le critère de base de fonctionnement de *SNMP* est donc la présence dans le réseau d'équipement à même de supporter *UDP* et par conséquent *IP*. De manière plus simple, il faut que les équipements membres du *Network*



---

**Algorithme 2.1** Starting Point

---

**Require:**  $R\_IP \leftarrow$  Starting router IP for network discovery**Require:**  $V\_SNMP \leftarrow$  SNMP version**Require:**  $C\_SNMP$  community or security parameters**Ensure:**  $boolean \leftarrow$  Router SNMP availability

```

1: function STARTING_POINT( $R\_IP, V\_SNMP, C\_SNMP$ )
2:   if  $R\_IP$  is manageable then
3:     return True
4:   else
5:     return False
6:   end if
7: end function

```

---

Management System (NMS) possèdent une adresse IP valide.

La découverte d'un équipement du réseau repose donc sur ce critère de base : les équipements en communication doivent avoir chacun une adresse IP valide. L'adresse IP du **routeur** spécifié est considérée comme adresse source (algorithme 2.1, page 8). Si le routeur ne supporte pas SNMP ou ne répond pas correctement à la requête SNMP (Community invalide), le processus de découverte s'arrête. Dans le cas contraire, le processus passe à l'étape suivante : la découverte des équipements connecté à ce routeur.

## 2) Device Discovery

Si l'étape du *Starting Point* est concluant, le processus de découverte des hôtes du réseau peut commencer (algorithme 2.2, page 9).

La découverte des hôtes du réseau se base sur deux groupes de MIB du MIB-II : `ipRouteTable` et `ipNetToMediaTable`.

`ipRouteTable` contient les variables relatives aux règles de routage du protocole IP. Il permet donc de découvrir les adresses IP des **routeurs** connectés avec le routeur courant. Dans cette table deux variables nous intéressent : `ipRouteNextHop` et `ipRouteType`. Le `ipRouteNextHop` indique l'adresse IP du *Next Hop* (un routeur) si son `ipRouteType` est *indirect*. `ipRouteTable` permet de découvrir **tous les routeurs** (hôtes de niveau 3) du réseau. En effet, en cas de découverte d'un routeur, son *Next Hop* permet de découvrir d'autres routeurs du réseau et ainsi de suite. **Il est très improbable qu'un routeur ne supporte pas SNMP** (SNMP s'appelait SGMP).

La table `ipNetToMediaTable` contient les informations relatives à la couche 2. Elle permet donc de découvrir le couple adresse IP / adresse MAC des hôtes du même réseau (réseau local).

La première étape de la découverte est constituée par la création du tableau vide  $D[]$ . Ce tableau contiendra les adresses IP de tous les équipements retrouvés dans le réseau. L'intérêt de ce tableau est double : avoir la liste des équipements du réseau et éviter à parcourir le même équipement plusieurs fois. Un routeur ayant par définition plusieurs interfaces.

Le premier routeur connu est le *Starting Point*. Son adresse est alors ajoutée au tableau  $D[]$ .

La découverte du réseau commence par la recherche des routeurs connecté au *Starting Point*.

---

**Algorithme 2.2** Device Discovery

---

**Ensure:**  $D[]$  all network device IP array

```

1: function DEVICE_DISCOVERY
2:    $D[] \leftarrow$  all network visited routers and device array, initially empty
3:    $D[].push(Starting\_Point\_IP) \leftarrow$  add Starting router IP to  $R[]$  stack
4:   for all  $D[n]$  do
5:     function GETINDIRECTNEXTHOP
6:       if ( $getRequest(D[n]) == True$ ) then  $\leftarrow ipRouteNextHop$ 
7:          $N\_H[] \leftarrow$  next hop set for  $R[n] \leftarrow ipRouteNextHop$  if  $ipRouteType$  is indirect
8:         for all  $N\_H[m]$  do
9:           if ( $D[]$  contains  $N\_H[m]$ ) then
10:            continue
11:          else
12:             $D[].add(N\_H[m])$ 
13:             $Device\_Grouping(D[])$ 
14:          end if
15:        end for
16:      end if
17:    end function
18:  end for
19:  function GETLOCALNETADDRESS
20:    for all  $D[i]$  do
21:      if ( $getRequest(D[i]) == True$ ) then  $\leftarrow ipNetToMediaNetAddress$ 
22:         $N\_D[] \leftarrow$  nettomediatable for  $D[i]$ 
23:        for all  $N\_D[j]$  do
24:          if  $D[]$  contains  $N\_D[j]$  then
25:            continue
26:          else
27:             $D[].add(N\_D[j])$ 
28:             $Device\_Grouping(D[])$ 
29:          end if
30:        end for
31:      end if
32:    end for
33:  end function
34: end function

```

---

### 1) Device discovery using next hop mechanism

Pour chaque routeur dans le tableau `D[]` (à la première itération, nous n'avons que le *Starting Point*), nous cherchons les valeurs du MIBs `ipRouteNextHop` de type *indirect*. Si on en trouve, on crée localement le tableau `N_H[]` contenant les adresses IP de tous les routeurs connectés au routeur courant. Ensuite, nous vérifions pour tous les routeurs contenus dans `N_H[]`, la présence de leurs adresses IP dans `D[]`. Si le routeur n'est pas encore connu, son adresse est ajouté à `D[]`. `D[]` contient donc de plus en plus d'éléments permettant à la boucle de tourner.

Au cas où le routeur n'aurait pas de table `ipRouteNextHop`, alors ce n'est pas un routeur mais un tout autre équipement qui sera découvert par l'étape suivante.

### 2) Device discovery using ARP cache entries

Lorsqu'il n'y aura plus de routeur non parcourus, `D[]` contiendra la liste de tous les routeurs visités du réseau. Pour chacun de ces routeurs, nous cherchons dans le cache ARP (`ipNetToMediaTable`), les adresses IP des hôtes de son réseau local que nous stockons dans le tableau `N_D[]`. Pour chaque adresse trouvée, nous vérifions l'unicité dans `D[]` pour éviter les doublons : le routeur est d'une part connecté à un autre routeur, d'autre part connecté à un autre réseau local. Dans le cache de ce routeur, nous aurons les adresses IP non seulement des hôtes du réseau local mais aussi celui des autres routeurs auquel il est connecté. La vérification dans le tableau `D[]` permet d'éliminer les adresses des autres routeurs connectés à celui-ci. Si nous avons une nouvelle adresse, elle est ajoutée à `D[]`.

#### Le problème de l'ARP

L' *Address Resolution Protocol* (ARP) assure l'établissement dans le cache de l'interface réseau, d'une table de correspondance entre adresse MAC et adresse IP des autres hôtes du même réseau local. Ce protocole n'est pas sécurisé puisqu'il est sujet au *cache poisoning* : il est possible de faire aussi bien de l'usurpation d'adresse IP que d'adresse MAC. Dans un réseau commuté, il suffit de faire croire au switch que nous sommes l'hôte possédant l'adresse que nous souhaitons écouter, et renvoyer après le trafic au véritable détenteur de l'adresse usurpée. De ce fait, tout le trafic passe par notre hôte.

Aussi la découverte d'hôtes grâce à l'ARP peut échouer à cause du TTL du caching au niveau des interfaces. Après un certain temps d'inactivité d'un hôte sur le réseau local, les informations relatives à cet hôte sont effacées du cache des autres interfaces du même réseau. Un hôte peut bien être dans le réseau local mais parce qu'il est éteint par exemple, la découverte par ARP va échouer.

Une solution est d'utiliser ICMP (*Internet Control Message Protocol*) à la place de l'ARP dans le réseau local. Au cas où le contenu de la table `ipNetToMediaTable` serait vide, nous pourrions envoyer des requêtes ICMP *type 8 (Echo)* c'est-à-dire des *pings* aux adresses IP du réseau local. Ces *pings* réguliers vont permettre de rafraichir le cache des interfaces des hôtes du réseau local. À noter que même si un hôte ne répond pas avec une réponse de *type 0 (Echo response)*, les informations relatives au couple adresse IP / MAC sont renvoyées. Mais dans le cas de cette étude, nous nous limiterons au contenu du `ipNetToMediaTable`.

### 3) Device grouping

Cette étape permet d'éviter les *synonymes* dans le réseau. Un même hôte pouvant avoir plusieurs adresses IP (sur plusieurs interfaces). Si par exemple un hôte était doublement connecté sur le réseau

**Algorithme 2.3** Device Grouping**Require:**  $D[]$  all network device IP array**Ensure:**  $D\_G[]$  : all network grouping device IP array

```

1: function DEVICE_GROUPING
2:   for all  $D[i]$  do
3:      $IP\_T[] \leftarrow$  all IP in  $ipAddrTable$  related to curent device  $\leftarrow ipAdEntAddr$ 
4:     for all  $IP\_T[i]$  do
5:       if  $IP\_T[i]$  equal  $D[i]$  then
6:         continue
7:       else
8:         if  $D[]$  contains  $IP\_T[i]$  then
9:            $D[].pop(IP\_T[i])$ 
10:        end if
11:      end if
12:    end for
13:  end for
14: end function

```

(physique et sans fil), nous aurions dans  $D[]$  deux adresses IP différentes signifiant deux hôtes différents au lieu d'un seul. De même vu le modèle de réseau virtuel de test (figure 2.1, page 19), on aurait dans  $D[]$  un nombre supérieur à six (nous n'avons que six routeurs dans ce réseau) d'adresses IP.

Une solution est de vérifier pour chaque adresse IP d'équipement ajouté dans  $D[]$ , si ce dernier ne possède pas en plus de son adresse connue, d'autres adresse enregistrées dans  $D[]$ . La table  $ipAddrTable$  contient pour chaque interface les informations sur l'adresse IP et l'adresse MAC. Pour chaque équipement du réseau dans le tableau  $D[]$ , nous récupérons sa table  $ipAddrTable$ . Pour chaque adresse IP de la table hormis l'adresse avec laquelle l'équipement est parcouru, nous vérifions la présence dans  $D[]$ . Si l'adresse IP est présent, nous l'enlevons du tableau des équipements  $D[]$ .

**L'étape du device grouping s'opère à chaque ajout d'un nouvel équipement dans  $D[]$ . C'est une étape exécutée aussi bien au niveau du *Device discovery using next hop mechanism* que du *Device discovery using ARP cache entries*.**

**4) Device type discovery**

Avec le tableau  $D[]$ , il devient possible de découvrir les *types* d'hôte. La variable  $sysService$  de *SNMP* nous permet d'identifier certains types d'hôtes en fonction des services offerts.

$sysService$  est une variable qui indique le type de service offert par l'équipement. Elle est représentée par la somme des bits correspondant aux couches OSI auxquelles l'équipement fournis des services. Le tableau 2.1 ([Valeur de la variable sysServices du MIB-II \(RFC 1213\)](#)) montre les valeurs en termes de position de bit pour les couches du modèle TCP/IP (RFC 1213). Le résultat du  $sysService$  détermine les autres *MIBs* à prendre en compte pour déterminer les types des hôtes. Le *RCF 1242* donne les *définitions* de quelques hôtes utilisées dans un réseau informatique.

**Router**

Le premier type d'équipement à identifier est le routeur.

Un routeur est un équipement qui transfère les paquets. La variable  $ipForwarding$  (RFC 4292) sera à 1. Aussi la variable  $sysService$  sera 1001110. Le second bit (de poids faible), le troisième, le

Couche	Service (matériel)
1	physique (repeteur)
2	liaison (switch non manageable)
3	réseau (switch manageable, routeur)
4	transport (hôtes)
7	application (mail, http, etc)

TABLE 2.1 – Valeur de la variable sysServices du MIB-II (RFC 1213)

**Algorithme 2.4** Device Type**Require:**  $D[]$  all network device IP array**Ensure:**  $D\_Type$  : local network devices type

```

1: function DEVICE_TYPE
2:   for all  $D[i]$  do
3:     if  $sysServices(D[i])$  then
4:       if  $sysServices(D[i])$  equal 78 then
5:          $D[i].append(GW)$ 
6:       else if  $sysServices(D[i])$  equal 72 then
7:         if  $ipForwarding(D[i])$  equal 1 then
8:            $D[i].append(GW)$ 
9:         else
10:           $D[i].append(SRV)$ 
11:        end if
12:      else
13:         $D[i].append(OTHER)$ 
14:      end if
15:    else if  $dot1dBridge(D[i])$  then
16:       $D[i].append(SW)$ 
17:    else
18:       $D[i].append(OTHER)$ 
19:    end if
20:  end for
21: end function

```

quatrième et le septième sont à 1 ; ce qui donne en base 10, 78. Un routeur offre les services des couches 2 (liaison), couche 3 (adressage), couche 4 (transport) et couche 7 (application). Un hôte qui sert de passerelle dans un réseau sera considéré comme un routeur. Il en est de même pour un routeur-switch.

**Host**

Les Hosts sont des équipements qui ne transfère pas de paquet mais qui dispose de table de routage. Ces hôtes peuvent être des PC, des imprimantes, des serveurs, etc. Pour plus de précision sur les types d'hôtes, nous pouvons utiliser les *Product specific MIB* tels *Printer MIB*, *WWW-MIB*, *APACHE-MIB*, etc. Mais dans le cadre de notre étude, nous nous limiterons à la valeur de `sysService` qui est 72<sup>5</sup> (1001000) pour les Hosts. Les Hosts offrent donc des services de niveau 4 (transport) et niveau 7 (application)

<sup>5</sup>RCF 1213, p 15

## Bridges

Si l'équipement n'est ni un routeur, ni un simple hôte, il pourrait être un switch.

Un pont ou switch de niveau 2, est un équipement transparent (au niveau 3) dans le réseau. Il ne transfère pas de paquet au niveau IP et ne dispose pas de table de routage. Le transfert de trame se fait au niveau liaison grâce au maintien d'une table d'adresses des cartes réseaux qu'il voit sur ses ports : l'*Address Forwarding Table (AFT)* (RFC 4188).

La détection des switches pourrait se baser sur les *Bridge-MIB*. Si un équipement dispose du *MIB dot1dBridge* alors nous pouvons le considérer comme un switch.

Dans le contexte de NAV, un équipement est soit un GW (routeur), GSW (routeur-switch), SW (switch), WLAN (AP), EDGE (switch en contact direct avec les ordinateurs. NAV ne collecte pas de données depuis ces switches), SRV (serveurs) et OTHER (autres type d'équipements). Dans le cas de cette étude, nous ne pouvons que identifier grossièrement les GW, les SW et les OTHER. *SNMP* ne permet pas directement de faire la distinction entre un PC (OTHER) et un serveur (SRV). Les deux pouvant offrir les mêmes services. Tout Host sera considéré comme OTHER. Il s'en va dire que même après l'identification de certains hôtes avec *SNMP*, la probabilité d'erreur est très proche de un et une identification manuelle est obligatoire : un routeur-switch pouvant être considéré comme un routeur (table de routage).

### 5) Create Bulk format

L'initialisation de la base se fera conformément au *Level 1 : Minimum requirements*<sup>6</sup> du *seed database*. Ainsi des valeurs par défaut existe déjà pour le *roomid* : **myroom** et l'*orgid* : **myorg**. Nous avons les adresses IP et leur type dans le tableau `D[]`. Pour chaque couple (adresse IP, type), il ne nous reste plus qu'à ajouter les valeurs communes telles le *roomid* et l'*orgid*, puis ajouter le *community* pour tous les équipements sauf les serveurs.

Pour chaque équipement, une ligne est ajoutée dans un fichier texte, contenant tous les équipements du réseau.

### 6) Store in Db

Pour le stockage dans la base de données, nous allons utiliser la méthode d'import en masse déjà disponible dans le NAV. En effet l'interface graphique du NAV permet de faire des importations à partir d'un fichier respectant le format adéquat. Cette importation en masse se fait à partir du menu Home > Seed DB > IP Devices puis Bulk import.

L'étape précédente, celle du **Create Bulk format** nous a déjà permis d'obtenir un fichier suivant le format adéquat. Il ne nous reste plus qu'à l'importer dans la base du NAV.

## III Implémentation, tests et résultats

---

### 1) Structuration

Le code du *SeedDB* est composé de quatre classes qui implémentent en python les algorithmes présentés dans le chapitre précédent.

La classe *SeedDB* est composée de fonctions telles :

---

<sup>6</sup><http://nav.uninett.no/seedessentials>

1. `__init__`
2. `request_snmp_parameters`
3. `check_snmp`
4. `device_discovery`
5. `get_device_type`
6. `create_bulk_format`
7. `catch_exit`

Cette classe `SeedDB` dépend de trois classes qui lui offrent des fonctionnalités spécifiques :

1. `DeviceGrouping` pour le filtrage des synonymes dans le réseau.
2. `GetIndirectNextHop` pour la découverte de tous les routeurs du réseau.
3. `GetLocalNetAddress` pour la découverte des équipements actifs dans un réseau local.

Aussi les modules suivants sont importés :

- `nav.Snmp` : utilisation du module *SNMP* du NAV.
- `buildconf` : utilisation du module `buildconf` de NAV pour retrouver automatiquement le répertoire des fichiers de logs.
- `signal` : interception action de l'utilisateur comme l'arrêt prématuré du script avec la commande `Control-C`.
- `datetime` : personnalisation du nom du fichier de bulk créé à la seconde près.
- `os` : utilisation des ressources du système, comme la création d'un répertoire pour la sauvegarde des fichiers de bulk.
- `IPy` : vérification de la validité IPv4 et IPv6 de l'adresse donnée pour le `seedRouter`.
- `logging` : création de logs suivant le modèle de Python.
- `threading` : utilisation du module *threading* de la classe *thread*.

Des variables de portée globale sont créées pour contenir les OIDs des MIBs nécessaires pour la découverte des types hôtes dans le réseau.

- `sys_service="1.3.6.1.2.1.1.7"` pour identifier les types d'équipements dans le réseau par la classe mère `SeedDb`.
- `ip_route_next_hop="1.3.6.1.2.1.4.21.1.7"` pour récupérer tous les autres routeurs auxquels est connecté un équipement (classe `GetIndirectNextHop`).
- `ip_route_type="1.3.6.1.2.1.4.21.1.8"` pour ne récupérer que les *NextHop* de type *indirect* (classe `GetIndirectNextHop`).
- `ip_forwarding="1.3.6.1.2.1.4.1"` pour filtrer les PC routeurs des simples PC (classe `SeedDb`).



- `ip_ad_ent_addr="1.3.6.1.2.1.4.20.1.1"` pour récupérer tous les adresses IP de tous les interfaces d'un équipement (classe `DeviceGrouping`).
- `ip_net_to_media_net_address="1.3.6.1.2.1.4.22.1.3"` pour récupérer les adresses IP des équipements appartenant au même réseau local (classe `GetLocalNetAddress`).
- `dot1d_bridge="1.3.6.1.2.1.17"` pour identifier les switches (classe `SeedDb`).

Aussi les variables globales `my_room` et `my_org` initialisées avec respectivement les valeurs *myroom* et *myorg*.

## 2) La classe `SeebDB`

C'est la classe principale qui importe tous les modules nécessaires et appelle ses méthodes et autres classes en fonction du déroulement du processus. Comme méthodes de cette classe nous avons :

### 1) `__init__`

La méthode `__init__` est le constructeur de la classe `SeedBD`. À l'appel de la classe, le constructeur demande deux informations à l'utilisateur : l'adresse IP du *Starting point* et la version de SNMP à utiliser. Si une de ses informations est incorrecte, le script s'arrête. De même, si les paramètres de sécurité ne sont pas correctes pour le *Starting point*, le script s'arrête.

En fonction de la version valide de SNMP choisie, la méthode `request_snmp_parameters` est appelée.

### 2) `request_snmp_parameters`

La fonction `request_snmp_parameters` récupère les informations de sécurité en fonction de la version de SNMP choisie précédemment. Pour les versions 1 et 2, seule le `community` est nécessaire. Pour la version 3, sont nécessaires les informations suivantes :

- `username` : identifiant d'authentification,
- `security_level` : niveau de sécurité entre *noAuthnoPriv*, *authnoPriv* et *authPriv*,
- `authentication_protocol` : protocole d'authentification entre *MD5* et *SHA*,
- `password` : mot de passe d'authentification,
- `privacy_protocol` : protocole de cryptage, entre *DES* et *AES*,
- `passphrase` : clé de cryptage.

Vu que NAV ne supporte pas encore la version 3 de SNMP, nous n'implémenterons que les versions 1 et 2 du protocole.

En fonction de la version choisie (1 ou 2), la méthode `check_snmp` intervient.



### 3) `check_snmp`

L'objectif de la méthode `check_snmp` est de s'assurer que les paramètres de sécurité fournis sont bien valide pour le *seedRouter* donné. Dans le cas des versions 1 et 2, seul le `community` est vérifié. La vérification passe par l'envoi d'une requête SNMP sur le MIB `sysDescr` (OID: 1.3.6.1.2.1.1.1). Cet OID est utilisé de manière native par NAV pour s'assurer de la validité des paramètres de sécurité SNMP au niveau des équipements. Dans une dynamique de compatibilité avec le NAV, nous utilisons également cet OID comme critère d'évaluation.

Si l'équipement répond à la requête SNMP `getRequest`, alors la fonction `device_discovery` prend le relais.

### 4) `device_discovery`

C'est la fonction principale (la plus importante) de notre implémentation. Elle inclue les classes `GetIndirectNextHop`, `DeviceGrouping`, `GetLocalNetAddress` et les méthodes `get_device_type` et `create_bulk_format`. La méthode `device_discovery` permet d'avoir dans un fichier *txt*, la liste des équipements du réseau au format de NAV.

Pour gérer les successions d'accès à la ressource commune qu'est le tableau des équipements du réseau, un sémaphore est mis en place à travers la variable `lock = threading.Lock()`

Grâce aux résultats qui lui sont fournis par les classes `GetIndirectNextHop`, `DeviceGrouping`, `GetLocalNetAddress`, la méthode `device_discovery` dispose dans un tableau la liste complète de tous les équipements uniques dans le réseau. Cet tableau est alors utilisé par les méthodes `get_device_type` et `create_bulk_format`.

### 5) `get_device_type`

L'identification des hôtes est assurée par la méthode `get_device_type` en utilisant les MIBs `sysService`, `ipForwarding` et `dot1dBridge`. En fonction des résultats obtenus par `getRequest` pour chaque MIB, un type d'équipement conforme aux types d'équipements<sup>7</sup> du NAV est détecté.

### 6) `create_bulk_format`

Cette dernière méthode du `device_discovery` permet de générer un fichier texte contenant les informations conforme au *Level 1 : Minimum requirements*<sup>8</sup> dans le répertoire `/tmp/nav`. Le fichier est identifié à la seconde près par son nom composé en partie par l'année, le mois, le jour, l'heure, la minute et la seconde courante : `ip_device_bulk_import_%Y%m%d%H%M%S`. Ce fichier est généré dans le répertoire temporaire puisque son usage est temporaire : importer les équipements du réseau dans la base de données du NAV.

### 7) `catch_exit`

Cette méthode n'a absolument rien à voir avec la découverte des équipements du réseau. Elle permet de terminer *correctement* le programme en interceptant une interruption *brutale* par l'utilisation du `Control-C`.

---

<sup>7</sup><http://nav.uninett.no/categoriesandtypes>

<sup>8</sup><http://nav.uninett.no/seedessentials>

### 3) La classe `GetIndirectNextHop`

La classe `GetIndirectNextHop` en utilisant l’OID `ipRouteNextHop` permet de retrouver tous les routeurs du réseau en partant du *Starting point* et en ne prenant en compte que les *Next Hop* de type *indirect*. Chaque nouveau routeur est ajouté au tableau des routeurs. Mais vu que par définition un routeur à plusieurs interfaces, la classe `GetIndirectNextHop` renverrait tous les adresses IP de tous les interfaces de tous les routeurs si la classe `DeviceGrouping` était absente.

La classe `GetIndirectNextHop` est composée de deux méthodes : `__ini__` et `run`.

#### 1) `__ini__`

La méthode `__ini__` est le constructeur de la classe. Vu que la classe `GetIndirectNextHop` étend la classe `threading.Thread`, le constructeur de la classe `threading.Thread` est initialisé aussi bien que d’autres variables nécessaires pour la découverte des routeurs du réseau. Une variable particulière est le `lock` provenant de la classe mère `SeedDb`. Cette variable est en fait un *sémaphore* permettant de gérer les accès concurrentiels entre threads à la ressource commune qu’est le tableau des équipements du réseau. Sans ce verrou, les divers threads pourraient écrire de manière anarchique dans le tableau, ce qui ne permettrait pas à la classe `DeviceGrouping` de pouvoir bien filtrer les synonymes dans le réseau.

#### 2) `run`

La méthode `run` est celle qui est exécutée lorsque la méthode `start` de l’objet de l’initialisation de la classe `GetIndirectNextHop` est appelée. Au début, un verrou est mis sur la ressource commune : le tableau des équipements du réseau. A la fin de la découverte des routeurs connectés au routeur courant, le verrou est enlevé.

Notons que la classe `GetIndirectNextHop` est appelée dans une boucle. D’où l’intérêt aussi d’un verrou et de la méthode `join` qui oblige le thread suivant à attendre la fin d’exécution du thread courant. Ainsi l’appel de la classe `DeviceGrouping` par chaque thread de type `GetIndirectNextHop` ne se fait qu’une seule fois à la fois.

### 4) La classe `DeviceGrouping`

Cette classe contient juste des attributs et une méthode. Le choix d’une classe au lieu d’une méthode dans la classe `SeedDb` se justifie par le fait qu’une classe peut facilement être implémenter dans une autre. Une méthode disponible dans la classe mère `SeedDb` n’est pas forcément disponible pour les autres classes qui sont appelées par cette même classe mère.

La classe `DeviceGrouping` permet d’éliminer les doublons dans le réseau en se basant sur le MIB `ipAdEntAddr`. Pour chaque adresse IP, nous vérifions si les autres interfaces portent des adresses déjà présentes dans le tableau des hôtes. Si oui, nous gardons cette adresse et supprimons celles des autres interfaces. A la fin de cette fonction, nous avons exactement le même nombre d’adresses IP que d’équipements physiques dans le réseau.

Ici pas besoin de thread, donc de verrou. La classe `DeviceGrouping` étant déjà appelé dans chaque thread crée. Mettre un thread au niveau de cette classe avec acquisition du verrou entrainerait un inter-blocage entre threads. Le thread parent ayant déjà la main sur la ressource et ne la libérant qu’après sa fin d’exécution. Or le thread fils aura aussi besoin de prendre la main sur la ressource pendant l’exécution du thread parent. Ce qui induit un blocage entre ces deux threads : le parent

attendant la fin d'exécution du fils pour relâcher la ressource, le fils en attendant le fin d'exécution du parent pour prendre la main sur la ressource.

## 5) La classe `GetLocalNetAddress`

Tout comme la classe `GetIndirectNextHop`, la classe `GetLocalNetAddress` est un thread.

En utilisant le cache ARP grâce au *MIB*, `ipNetToMediaNetAddress`, il devient possible pour chaque routeur, de détecter les hôtes actifs de son réseau local. Ici aussi pour chaque nouvel équipement découvert, il nous faudra s'assurer de l'unicité de l'adresse IP grâce à la classe `DeviceGrouping`. A la fin de cette fonction, nous avons tous les adresses IP uniques des équipements du réseau. Reste à identifier les types d'équipements.

La classe `GetLocalNetAddress` est aussi composé de deux méthodes : `__ini__` et `run` conformément à la structure du module `threading` de python. Le modèle de fonctionnement est identique à celui de la classe `GetIndirectNextHop`.

La méthode `start` (de l'objet) exécute la méthode `run` (de la classe). Un `acquire` permet au thread courant de prendre la main sur le tableau des équipements du réseau. A la fin de l'exécution de ce thread, un `release` permet de libérer la ressource. Le verrou permet de gérer les successions d'accès en écriture et lecture à la ressources partagée : le tableau des équipements du réseau. La méthode `join` oblige chaque thread à attendre la fin d'exécution du thread courant.

## 6) Évaluation du code

Un code *pythonique*<sup>9</sup> doit suivre les normes définies dans le *PEP8* (*Python Enhancement Proposal*)<sup>10</sup>. Après installation du *PEP8*, une évaluation du code donne :

```
1 $ pep8 --show-source --show-pep8 --benchmark seeddb.py
2 0.06      seconds elapsed
3 16       files per second (1 total)
4 5116     logical lines per second (316 total)
5 8127     physical lines per second (502 total)
```

Cette réponse prouve bien que notre code est conforme au *PEP8*. Ce code est donc valide selon les recommandations du *PEP8* et peut être facilement compréhensible par tout développeur python.

## 7) Test dans un environnement virtuel

### 1) Réseau virtuel de test des implémentations des algorithmes

Pour la mise en pratique, nous avons mis en place un réseau virtuel grâce à *NetKit*<sup>11</sup>. Selon le mode de fonctionnement de *NetKit*, notre réseau virtuel est un laboratoire.

<sup>9</sup><http://chrisarndt.de/talks/rupy/2008/output/slides.html>

<sup>10</sup><http://www.python.org/dev/peps/pep-0008/>

<sup>11</sup><http://wiki.netkit.org>

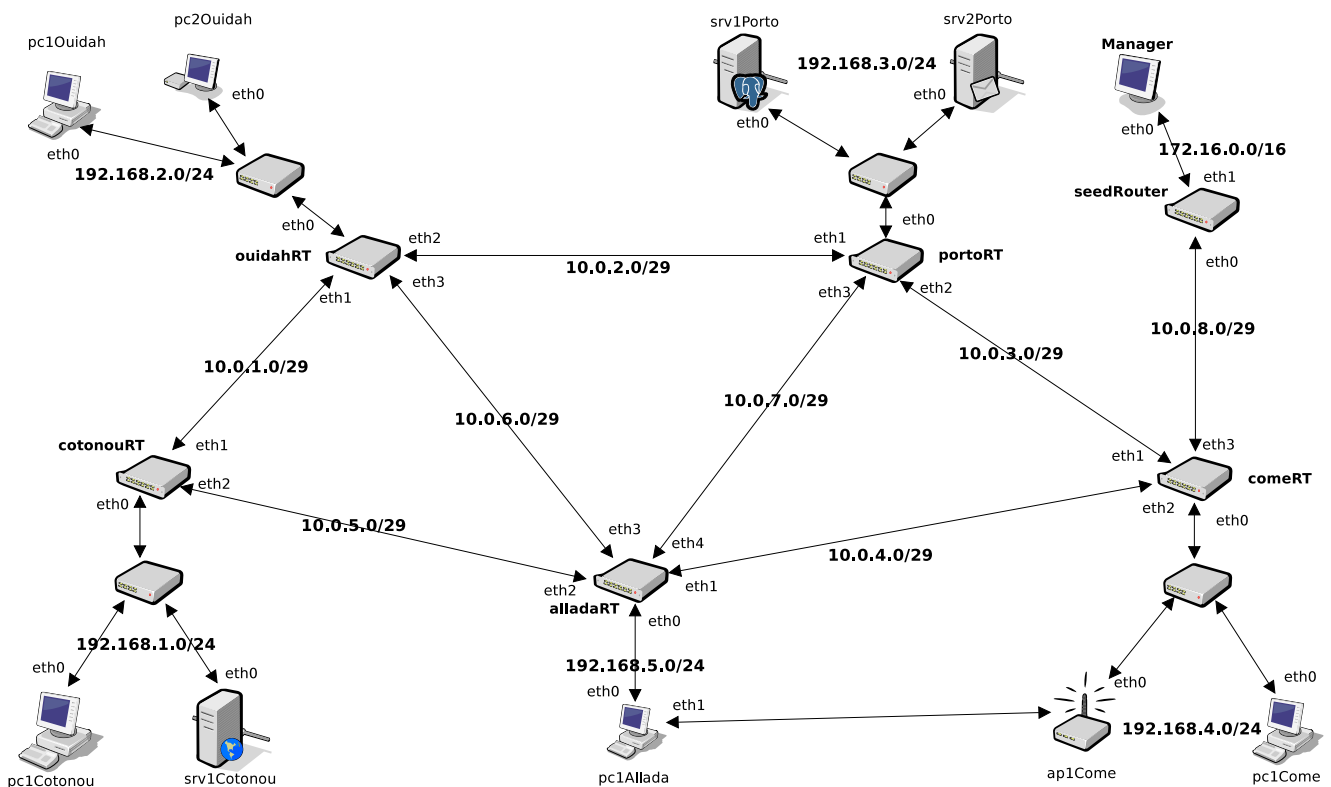


FIGURE 2.1 – Modèle de réseau (créé avec NetKit) à découvrir par le seedDB

La figure 2.1 (page 19) montre le modèle de réseau créée. Ce laboratoire est constitué des équipements ayant les caractéristiques réseaux suivantes :

- PCs
  - pc1Cotonou (192.168.1.1 sur eth0)
  - pc1Ouidah (192.168.2.1 sur eth0)
  - pc2Ouidah (192.168.2.2 sur eth0)
  - pc1Allada (192.168.5.1 sur eth0 et 192.168.4.100 sur eth1)
  - pc1Come (192.168.4.1 sur eth0)
- Serveurs
  - srv1Cotonou (192.168.1.2 sur eth0)
  - srv1Porto (192.168.3.1 sur eth0)
  - srv2Porto (192.168.3.2 sur eth0)
- Access Point : ap1Come (192.168.4.2 sur eth0)
- Routeur
  - rtCotonou :
    - \* eth0 : 192.168.1.100
    - \* eth1 : 10.0.1.1
    - \* eth2 : 10.0.5.1

```
- rtOuidah :  
    * eth0:192.168.2.100  
    * eth1:10.0.1.2  
    * eth2:10.0.2.2  
    * eth3:10.0.6.2  
  
- rtPorto :  
    * eth0:192.168.3.100  
    * eth1:10.0.2.3  
    * eth2:10.0.3.3  
    * eth3:10.0.7.3  
  
- rtCome :  
    * eth0:192.168.4.100  
    * eth1:10.0.3.4  
    * eth2:10.0.4.4  
    * eth3:10.0.8.4  
  
- rtAllada :  
    * eth0:192.168.5.100  
    * eth1:10.0.4.5  
    * eth2:10.0.5.5  
    * eth3:10.0.6.5  
    * eth4:10.0.7.5  
  
- seedRouter :  
    * eth0:10.0.8.6  
    * eth1:172.16.0.6
```

Le *Manager* ne fait pas partie du réseau virtuel. L'adresse IP de son interface en communication avec le réseau virtuel est : 172.16.0.7.

L'adresse `eth1` du `seedRouter` est totalement différent de celui du réseau virtuel (nous passons de la classe A à la classe B) pour éviter un conflit au niveau des adresses des interfaces `tap` et de celui du réseau virtuel de *Netkit*.

Nous donnerons comme point de départ à notre algorithme, le `seedRouter`. A noter que *SNMP* est déjà configuré par nos soins sur tous les équipements. Pour que depuis notre PC, nous puissions accéder au réseau virtuel à travers le `seedRouter`, il faut activer au niveau du `seedRouter`, l'interface `tap`.

Vu que *NAV* ne supporte pas encore *SNMPv3*, et que la typologie va être découverte par `ipdevpoll` du *NAV*, notre code pour plus de cohésion, implémente la classe *SNMP* du *NAV*, qui en fait est une extension de la classe *pysnmp* (module *SNMP* de python).

Pour finir sur le réseau virtuel, notons que les hôtes du réseau peuvent être tout type d'équipement. Dans le cadre de cette étude nous nous sommes limités aux PCs, Switchs L2, serveurs et routeurs. Le plus important étant que l'équipement ajouté dans le réseau supporte *SNMP* et soit correctement configuré.

## 2) Résultats

La classe SeedDB est ajouté dans le répertoire `/opt/nav-3.11.5/python/nav/seeddb`. L'appel de la classe avec les paramètres correctes donne :

```

1 $ pwd
2 /opt/nav-3.11.5/python/nav/seeddb
3 $ sudo python seeddb.py
4
5 =====
6 === Welcome to NAV SNMP based Network Host Discovery ===
7 =====
8
9 Enter starting router IP: 172.16.0.6
10 [OK] seed_router...
11 =====
12 ==== Select SNMP version for your Network ====
13 ==== 1 for SNMP v1 ====
14 ==== 2 for all versions of SNMP v2 ====
15 ==== 3 for SNMP v3 ====
16 =====
17 2
18 Enter community for version 2: public
19
20 Checking SNMP parameters...
21
22 [OK] SNMP parameters...
23 Starting device discovery
24
25 Getting all routers on the network
26
27 [OK] Routers list...
28
29 Getting all Hosts on the network
30
31 I'am afraid , looks like something went wrong with 10.0.8.6 !
32 ===== get_local_net_address error =====
33 Timed out waiting for SNMP response: Manager: no response arrived before timeout
34
35
36 [OK] All devices list...
37
38 Setting all network Host type
39
40 [OK] Device type...
41
42 Creating bulk import file
43
44 [OK] Create bulk import file: /tmp/nav/ip_device_bulk_import_20120829013534.txt
45 Exiting...
```

La classe SeedDB ajoute à chacun de son appel, plus de détails sur le déroulement du processus de découverte des équipements dans le fichier `/var/log/nav/autoseeddb.log` spécifié dans la classe SeedDB. Ce fichier journal contient beaucoup de ligne pour chaque opération : ajout dans le tableau des routeurs, regroupement des routeurs, ajout des équipements du réseau local, regroupement des équipements du réseau local, identification des types d'hôte, création du fichier d'importation. Les dernières lignes de ce fichiers peuvent donner au cas où tout c'est bien déroulé :

```

1 $ tail -f /var/log/nav/autoseeddb.log
2 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Adding line myroom:192.168.5.1:myorg:OTHER:public in file
3 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Adding line myroom:10.0.5.1:myorg:GW:public in file
4 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Adding line myroom:10.0.7.3:myorg:GW:public in file
5 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Adding line myroom:10.0.2.2:myorg:GW:public in file
6 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Adding line myroom:10.0.3.4:myorg:GW:public in file
7 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Closing file /tmp/nav/ip_device_bulk_import_20120829013534.txt
8 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Create bulk import file: /tmp/nav/ip_device_bulk_import_20120829013534.txt
9 [ 2012-08-29 01:35:34,900 ][ INFO ] root - Exiting at the end of the script...
```

Le fichier obtenu se présente comme suit :

```

1 $ cat /tmp/nav/ip_device_bulk_import_20120829013534.txt
2 myroom:10.0.8.6:myorg:GW:public
3 myroom:172.16.0.7:myorg:OTHER:public
4 myroom:192.168.1.1:myorg:OTHER:public
5 myroom:192.168.1.2:myorg:OTHER:public
6 myroom:192.168.3.1:myorg:OTHER:public
7 myroom:192.168.3.2:myorg:OTHER:public
8 myroom:192.168.2.1:myorg:OTHER:public
```

```

9 myroom:192.168.2.2:myorg:OTHER:public
10 myroom:192.168.4.1:myorg:OTHER:public
11 myroom:192.168.4.2:myorg:OTHER:public
12 myroom:10.0.7.5:myorg:GW:public
13 myroom:192.168.5.1:myorg:OTHER:public
14 myroom:10.0.5.1:myorg:GW:public
15 myroom:10.0.7.3:myorg:GW:public
16 myroom:10.0.2.2:myorg:GW:public
17 myroom:10.0.3.4:myorg:GW:public

```

Les figures 2.2 (page 22) et 2.3 (page 23) montrent les résultats d'importation du fichier généré par SeedDB dans NAV.

Seed DB
IP device
Service
Room
Location
Organization

List
Add new IP device
Bulk import

Status	Line	Input	Remark
●	1	myroom:10.0.8.6:myorg:GW:public	
●	2	myroom:172.16.0.7:myorg:OTHER:public	
●	3	myroom:192.168.1.1:myorg:OTHER:public	
●	4	myroom:192.168.1.2:myorg:OTHER:public	
●	5	myroom:192.168.3.1:myorg:OTHER:public	
●	6	myroom:192.168.3.2:myorg:OTHER:public	
●	7	myroom:192.168.2.1:myorg:OTHER:public	
●	8	myroom:192.168.2.2:myorg:OTHER:public	
●	9	myroom:192.168.4.1:myorg:OTHER:public	
●	10	myroom:192.168.4.2:myorg:OTHER:public	
●	11	myroom:10.0.7.5:myorg:GW:public	
●	12	myroom:192.168.5.1:myorg:OTHER:public	
●	13	myroom:10.0.5.1:myorg:GW:public	
●	14	myroom:10.0.7.3:myorg:GW:public	
●	15	myroom:10.0.2.2:myorg:GW:public	
●	16	myroom:10.0.3.4:myorg:GW:public	


Import

FIGURE 2.2 – Fenêtre de confirmation de l'importation des données du fichier des équipements du réseau

« Previous 1 Next » 100 View

Move selected Delete selected

**IP Devices**  
16 found in database, showing 1 to 16.

<input type="checkbox"/>	 Sysname	Room	Ip	Category	Organization	Read only	Read write	SNMP version	Type name	Serial
<input type="checkbox"/>	10.0.2.2	myroom	10.0.2.2	GW	myorg	public		1		
<input type="checkbox"/>	10.0.3.4	myroom	10.0.3.4	GW	myorg	public		1		
<input type="checkbox"/>	10.0.5.1	myroom	10.0.5.1	GW	myorg	public		1		
<input type="checkbox"/>	10.0.7.3	myroom	10.0.7.3	GW	myorg	public		1		
<input type="checkbox"/>	10.0.7.5	myroom	10.0.7.5	GW	myorg	public		1		
<input type="checkbox"/>	10.0.8.6	myroom	10.0.8.6	GW	myorg	public		1		
<input type="checkbox"/>	172.16.0.7	myroom	172.16.0.7	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.1.1	myroom	192.168.1.1	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.1.2	myroom	192.168.1.2	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.2.1	myroom	192.168.2.1	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.2.2	myroom	192.168.2.2	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.3.1	myroom	192.168.3.1	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.3.2	myroom	192.168.3.2	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.4.1	myroom	192.168.4.1	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.4.2	myroom	192.168.4.2	OTHER	myorg	public		1		
<input type="checkbox"/>	192.168.5.1	myroom	192.168.5.1	OTHER	myorg	public		1		

Move selected Delete selected

« Previous 1 Next » 100 View

FIGURE 2.3 – Résultat de l'importation du fichier des équipements du réseau



# Bibliographie

- [1] Douglas R. Mauro, Kevin J, 2005 : *Essential SNMP, Second Edition*, O'Reilly Media, 462 p, <http://shop.oreilly.com/product/9780596008406.do>
- [2] NGUYEN Manh Tuong. *Les protocoles pour la gestion des réseaux Informatiques* . 2005. [http://www2.ifi.auf.org/rapports/tpe-promo10/tipe-nguyen\\_manh\\_tuong.pdf](http://www2.ifi.auf.org/rapports/tpe-promo10/tipe-nguyen_manh_tuong.pdf)
- [3] Yves Bertsch, Frederic Stmarcel. *Administration Réseau-système SNMPv1, SNMPv2, SNMPv3 et HTTP* . 2001. <http://2001.jres.org/actes/snmpvhttp.pdf>
- [4] Simple Web (Aiko Pras, Ramin Sadre), Design and Analysis of Communications Systems (DACS) group, University of Twente, The Netherlands, *Internet Management Tutorials* . 2012. [http://www.simpleweb.org/wiki/Internet\\_Management\\_Tutorials](http://www.simpleweb.org/wiki/Internet_Management_Tutorials)
- [5] Pandey, Choi, Lee and Hong. *IP Network Topology Discovery Using SNMP*. International Conference on Information Networking (ICOIN'09). 23 33-37 2009 [http://dpm.postech.ac.kr/papers/ICOIN/09/TopologyDiscovery\\_November\\_18.pdf](http://dpm.postech.ac.kr/papers/ICOIN/09/TopologyDiscovery_November_18.pdf)
- [6] Yang Xiao. *Physical Path Tracing for IP Traffic using SNMP*. BBC Research White Paper WHP 188 2010
- [7] Pandey, Choi, Won and Hong. *SNMP-based enterprise IP network topology discovery*. International Journal of Network Management 21 (3) 169-184. 2011
- [8] Musa Balta, Ibrahim Özçelik *The Discovery of Enterprise Network Topology Created in a Virtual Environment with SNMPv3*. The Online Journal of Science and Technology (TOJSAT) Volume 2, Issue 2, 64-70, April 2012.
- [9] R. Siamwalla, R. Sharma, S. Keshav *Discovering Internet Topologym*. Cornell Network Research Group, Department of Computer Science, Cornell University, Ithaca, 1988.
- [10] Bierman and Jones *Physical topology MIB*. IETF web page, 2000 <http://www.ietf.org/rfc/rfc2922.txt>
- [11] Mark Lutz. *Learning Python, Fourth Edition*, O'Reilly Media, 1214 p, 2009.
- [12] Mark Lutz. *Programming.Python, Fourth Edition*, O'Reilly Media, 1628 p, 2010.

