Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche
researchportal.unamur.be
University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Spécification et développement d'une chaîne automatique de test des agents SNMP

Ndaye Mukuna, Maurice

Award date:
1999

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

FACULTES UNIVERSITAIRES NOTRE–DAME DE LA PAIX
INSTITUT D'INFORMATIQUE
Rue Grandgagnage, 21
NAMUR
BELGIQUE

SPECIFICATION ET DEVELOPPEMENT D'UNE CHAINE AUTOMATIQUE DE TEST DES AGENTS SNMP

Maurice NDAYE MUKUNA

Mémoire présenté en vue de l'obtention du grade de
Maître en Informatique

Promoteur : Pr. Jean RAMAEKERS

Année Académique
1998–1999

FACULTES UNIVERSITAIRES NOTRE–DAME DE LA APAIX DE NAMUR
INSTITUT D'INFORMATIQUE
Rue Grandgagnage, 21
5000 NAMUR
BELGIQUE (BELGIUM)

SPECIFICATION ET DEVELOPPEMENT D'UNE CHAINE AUTOMATIQUE DE TEST DES AGENTS SNMP

Maurice NDAYE MUKUNA

Résumé

Ce mémoire s'intéresse à un des problèmes d'administration des réseaux informatiques : la création des agents d'administration du protocole SNMP. Il s'agit de concevoir un outil permettant de vérifier et de valider le bon fonctionnement de différents agents d'administration construits suivant le protocole SNMP.

Ce travail a la prétention de servir de guide de l'utilisateur de l'outil de test développé pour faciliter la tâche des développeurs des agents snmp.

Abstract

This work is interested in one of the problems of network management : the network management SNMP protocol agent's creation. We have to design a tool whitch can check and validate the operation of agents built according to protocol SNMP.

This work claims to be used by développers as user guide and to make them sure about the right building of the snmp agents

Mémoire présenté en vue
de l'obtention du grade de
maître maître en informatique
Septembre 1999

Promoteur : Jean RAMAEKERS

TABLE DES MATIERES

TABLE DES MATIÈRES 1

REMERCIEMENTS
5

I. INTRODUCTION. 5

I ÈRE PARTIE : CONCEPTS FONDAMENTAUX. 9

CHAPITRE 1 : LES CONCEPTS FONDAMENTAUX. 11

1.1 Introduction. 11
1.2 Principes et concepts de réseau. 11
1.3 Système informatique distribué. 12
1.4 Le modèle TCP/IP 12

CHAPITRE 2 : LE PROTOCOLE SNMP (Simple Network Management Protocol). 15
2.1 Introduction. 15
2.2 Présentation du protocole snmp 15
2.3 Présentation de la MIB snmp. 19
2.4 Spécification du protocole SNMP. 21
2.4.1 Le format SNMP. 21

III. 2 ÈME PARTIE : DEVELOPPEMENT D'UN AGENT SNMP ET DE LA CHAINE AUTOMATIQUE DE TEST DES AGENTS SNMP. 25

CHAPITRE 1 : DEVELOPPMENT D'UN AGENT SNMP : l'AGENT PING. 27
1.1 Introduction. 27
1.2 Présentation de l'agent SNMP. 27
1.3 Présentation d'un agent SNMP construit avec GAM-OAT. 28
1.4 Construction de l'agent PING. 30
1.5 Fonctionnement de l'agent PING. 35
1.6 Simulation du fonctionnement de l'agent PING. 49

CHAPITRE 2 : SPECIFICATION ET DEVELOPPEMENT DE LA CHAINE AUTOMATIQUE DE TEST **53**
2.1 Introduction. 53
2.2 Spécification. 53
2.3 Développement de la chaîne automatique. 56
2.4 Simulation des tests sur l'agent PING. 63
2.5 Validation. 68

IV. CONCLUSION **69**

V. ANNEXE **71**

Annexe 1.1 : syntaxe des objets de la MIB. **71**

Annexe 1.2 : Code source de la méthode à associer au noyau de l'agent PING **72**

Annexe 1.3 : Code source de l'automate **81**

Annexe 1.4 : Fichier rc.Pingmib **: contenu du script de lancement du noyau de l'agent PING** **96**

Annexe 1.5 : fichier rc.methode1 : contenu du script de lancement de la méthode de l'agent PING **97**

Annexe 1.6 : fichier Ping.Ksh **: programme de lancement de la commande Ping** **98**

Annexe 2 : présentation de l'API de GAM **99**

Annexe 2.5 Types utilisés. **107**

VI. REFERENCES BIBLIOGRAPGIQUES. **109**

REMERCIEMENTS

La réalisation de ce mémoire s'est fait en deux étapes : une étape pratique et une phase de rédaction. Je commence tout d'abord par remercier le professeur Jean RAMAEKERS pour m'avoir donné l'occasion, afin de bien préparer mon mémoire, de vivre une expérience professionnelle très enrichissante dans le domaine du développement chez BULL et de m'avoir guidé durant toute la période de rédaction de mon mémoire.

Durant mon séjour chez BULL en France, j'ai eu l'occasion de travailler dans une équipe très sympathique et très professionnelle, qui m'a facilement intégré dans leur groupe, bien encadré et aidé. Ils m'ont permis de me rendre compte de mes capacités et m'ont appris l'importance de la rigueur et de l'efficacité dans une démarche de production industrielle d'un produit informatique. Je remercie particulièrement Jean BRUNET, Gérald SEDRATI, Florence LAMBERET et Patrick NOLET pour leur disponibilité et conseils.

Je remercie toutes les personnes qui ont cru en mes capacités et m'ont donné l'occasion de mener mes études à leurs termes. Je pense profondément à toutes les familles MUKUNA et GILLAIN, et ainsi qu'à toutes les autres personnes qui sont tellement nombreuses que je ne saurais pas les citer tous ici.

Il est des êtres qui nous sont tellement chers qu'on ne saurait oublier leur absence. Je dédie du fond de mon coeur ce mémoire à mon jeune frère Jean Philippe MUKUNA TSHISUMPA décédé durant mon séjour d'études aux Facultés Universitaires Notre-Dame de la Paix de NAMUR.

I. Introduction.

Dès l'apparition du premier ordinateur commercial en 1950, l'informatique s'est imposée dans tous les domaines d'activités, tant professionnels que privés. L'ordinateur se caractérise par son aptitude à traiter des volumes de données considérables à très grande vitesse. Les données peuvent être de différentes natures, pour autant que le système ait été conçu, programmé pour le traitement. Pour leur part, les organisations ne sont pas restées à l'écart de cette nouvelle technologie, au contraire, elles constituent l'un des champs qui a vu sa structure complètement et radicalement modifiée par l'informatique.

Le monde des organisations peut être considéré comme une résultante complexe de flux qui se mêlent les uns des autres (parallèles, circulaires, réciproques). Pour savoir comment fonctionnent les organisations, il faut connaître, outre les parties dont elles sont faites, les fonctions que remplissent ces parties et la façon dont elles sont reliées les unes aux autres, plus précisément, comment les flux de travail, d'autorité, d'information et de décision irriguent les organisations. L'informatique apporte un support aux organisations.

A partir du moment où, dans une organisation, plusieurs postes, plusieurs services travaillent avec les même données, il faut pouvoir découpler les applications des fichiers et rassembler ces données dans une ou plusieurs bases de données auxquelles peuvent se connecter les postes ou services, via leurs applications. Il s'agit, en d'autres termes, de construire un ensemble cohérent des fichiers de référence auxquels se rapporte l'ensemble des applications informatiques, la saisie d'informations primaires n'étant plus effectuée qu'une seule fois et transférée d'une application à l'autre, les résultats des unes devenant les données d'entrée des autres. Pour cela, il va falloir doter l'organisation d'un ensemble de techniques et de moyens matériels pour relier entre eux les utilisateurs. Donc, construire un réseau.

L'inexorable augmentation des besoins des organisations en système d'information s'accompagne d'un développement rapide au niveau des matériels et des technologies de gestion distribuée des données. Actuellement, une organisation type a un grand réseau constitué d'une variété des services (applications) et équipements distribués, incluant des PC, des stations de travail et des serveurs. Il devient donc déterminant de disposer d'un moyen d'administration de tout cet ensemble.

L'administration de réseau concerne les politiques, les procédures et les outils nécessaires pour administrer les réseaux. En fait, c'est comme si on gérait des ressources comme le capital et les informations. L'administration des réseau est importante pour le système distribué car les applications distribuées auxquelles on accède à travers le réseau sont en augmentation. En plus, les croissantes utilisations des stations de travail, des réseaux locaux et des applications fonctionnant selon le modèle client-serveur soulignent le rôle des réseaux dans les organisations.

Les fonctions suivantes sont utilisées pour définir les caractéristiques fonctionnelles de l'administration de réseau :
– la configuration de l'administration : concerne la définition, le changement, le pilotage et le contrôle des ressources et données du réseau.
– le compte : concerne l'enregistrement des utilisations des ressources et l'établissement des notes d'informations.
– erreur d'administration : concerne la détection, le diagnostic et la correction des erreurs.
– l'analyse des performances : concerne le pilotage des performances du réseau
– sécurité : en vue de s'assurer que seuls les accès autorisés accèdent aux ressources.

L'administration de réseau peut être menée dans un réseau local comme un modèle ayant un seul niveau et dans lequel, l'administrateur communique directement avec les éléments du réseau. Ce modèle fonctionne bien pour de petits réseaux homogènes. Dans ces types des réseaux, les différents éléments communiquent avec l'administrateur en utilisant le même protocole. La situation la plus réaliste pour des larges réseaux constitués des éléments de différents vendeurs et supportant différentes architectures réseau est celle des réseaux dans lesquels l'administrateur communique avec les différents domaines à administrer. Dans ce cas, le réseau est décomposé en plusieurs sous-réseaux (domaine) et dans chaque domaine, un administrateur local supervise son propre sous-réseau.

Les standards sont utiles dans l'administration de réseau. Les standards les plus connus sont le Common Management Information Protocol (CMIP) de l'ISO et le Simple Network Management Protocol (SNMP) qui a été développé pour les réseaux basés sur le modèle TCP/IP. Un administrateur est un module logiciel qui réside dans le système administrant.

Dans le cadre de l'administration de réseau par SNMP, le réseau est constitué d'un certain nombre de stations d'administration et d'éléments ou noeuds (ressources). Les stations d'administration supportent les fonctions d'administration du réseau, afin de contrôler les différentes ressources du réseau. Les ressources du réseau sont modélisées par des agents. Ces agents doivent être capables de s'acquitter de la réalisation d'un certain nombre d'opérations nécessaires à l'administration du réseau, et ce, à la demande des stations d'administration dont ils dépendent.

Les opérations qui doivent être supportées par les agents sont principalement liées à la consultation et la modification d'informations existantes au sein de l'agent, aussi bien que le signalement de la survenance d'un certain nombre d'événements prédéterminés. De manière plus précise, le protocole SNMP se borne à offrir à la station d'administration des fonctionnalités de consultation et de modification d'une base de données d'informations au sein de chacun de ses agents, ainsi qu'un mécanisme permettant à chaque agent de signaler la survenance de certains événements particuliers.

Les équipes de développement construisent des agents SNMP réalisant toutes les fonctionnalités ci-dessus décrites. C'est afin de s'assurer du bon fonctionnement de ces agents suivant la norme snmp qu'a été exprimé le besoin de disposer d'un outil indépendant de test de ces agents snmp. L'outil visé devait consister en une chaîne automatique des tests. Le mémoire que nous avons réalisé a pour objectif de réaliser cette chaîne automatique qui permet de tester des agents fonctionnant suivant le protocole SNMP.

Pour faciliter la compréhension du fonctionnement de cette chaîne, nous commençons par la présentation des concepts que nous avons mis en oeuvre durant notre travail. Il s'agit principalement des concepts liés au réseau et au protocole SNMP. Avant de décrire la réalisation de la chaîne automatique de test, nous tenterons de créer un agent snmp implémentant toutes les fonctionnalités ci-dessus décrites. Cet agent nous servira de cobaye pour simuler les tests sur la chaîne qui sera construite. La réalisation de la chaîne automatique de test est construite pour tester les agents d'administration construits suivant le protocole SNMP. Après la construction d'un agent snmp virtuel, nous expliquerons le fonctionnement de la chaîne que avons réalisée.

Ce travail est structuré en deux parties :

1 ère Partie : Les concepts fondamentaux.

Cette partie est consacrée à la présentation des concepts qui seront utilisés dans ce travail. Le protocole SNMP sur lequel porte ce travail ne peut se concevoir que dans un environnement distribué. Il opère au niveau de la couche application du modèle TCP/IP. Ainsi, nous présenterons dans cette partie les concepts de réseau, de système distribué et le modèle TCP/IP.

Pour ferons également une présentation du protocole SNMP afin de bien comprendre les services offerts par ce protocole, sa structure de fonctionnement et les différents échanges et messages qu'il met en oeuvre.

2 ème Partie : Technologie de développement des agents SNMP et Spécification et développement d'un automate de test des agents SNMP

.

Cette partie sera considérée comme la plus pratique de ce travail. Nous commencerons par construire un agent snmp virtuel. Cette démarche nous permettra essentiellement de maîtriser les différentes techniques qui sont utilisées pour développer des agents snmp. L'agent sera construit suivant la technologie mis en oeuvre par le toolkit GAM–OAT. Puis, nous passerons au développement de la chaîne automatique de test des agents snmp. Cette chaîne sera développée sous l'environnement UNIX. Cette chaîne simule le fonctionnement d'un agent snmp et teste si le comportement de cet agent permet de réaliser toutes les fonctionnalités dédiées à un agent snmp normal. Pour valider cette chaîne de test, nous simulerons son fonctionnement sur l'agent snmp virtuel que nous avons construit auparavant.

I ère Partie : CONCEPTS FONDAMENTAUX.

Le protocole simple d'administration de réseaux ne peut se concevoir que dans un environnement distribué. Cette partie sera consacrée à la présentation des principaux concepts qui seront utilisés dans la suite.

CHAPITRE 1 : LES CONCEPTS FONDAMENTAUX.

1.1 Introduction.

La plupart d'organisations utilisent le réseau pour optimiser la réalisation de leurs activités. Un réseau interconnecte les différentes ressources de l'organisation. Il s'en suit une sorte de coopération des applications distribuées. Les applications distribuées interagissent au travers d'un système distribué. Etant donné la variété des origines de différents équipements constituant un réseau, un besoin de standardisation s'est fait sentir. C'est dans ce contexte que LE DoD (Department of Defense) américain a proposé un modèle de définition de réseau en couches : le modèle TCP/IP.

Le protocole d'administration de réseau snmp sur lequel a porter notre travail opère dans un réseau et utilise la notion de système distribué. Il sollicite les services de bas niveau qu'offre le modèle TCO/IP. Dans ce chapitre, nous allons présenter les différents concepts relatifs au réseau, au système distribué et au modèle TCP/IP.

1.2 Principes et concepts de réseau.

Un réseau est une collection d équipements vu comme un tout autonome, qui interconnecte deux ou plusieurs stations.
Une station est un point final dans un réseau de communication et elle peut être un terminal ou un ordinateur. Les réseaux fournissent des services d'échange d'informations dans un système informatique distribué.

Ils sont principalement responsables de trois services suivants : livraison, compréhension et accord. La livraison porte sur le transport physique des données entre stations. Dans ce contexte, les données consistent en toute chose qui a une signification pour l'utilisateur. La livraison implique la recherche du chemin et l'envoi correct des données à travers ce chemin. La compréhension garantit que les données envoyées sont bien dans le format compréhensible par le récepteur ou le destinataire. Les données pourraient nécessiter une transformation entre l'expéditeur et le destinataire. L'accord garantit que les données sont envoyées lorsque le destinataire est prêt à les recevoir. Cela signifie que les règles d'échange (protocole) doivent être établies entre les deux.

Un réseau peut se présenter sous trois configurations :
– il peut être un large réseau (WAN : Wide Area Network) utilisant l'infrastructure commune de communication,
– il peut être un réseau local (LAN : Local Area Ntework) qui exploite les moyens de communication privés pour connecter ses terminaux sur une superficie de taille limitée,
– il peut aussi être réseau métropolitain (MAN : Metropolitan Area Network) et installé sur une étendue de la taille d'une région.

1.3 Système informatique distribué.

1.3.1 Définition

Le terme de système informatique distribué est utilisé couramment pour désigner des ensembles informatiques constitués d'unités de traitement ou de stockage interconnectés par un système de communication. Sur ces unités s'exécutent des programmes qui coopèrent à la réalisation d'une tâche commune ou partagent des ressources communes. Le progrès des techniques de transmission de données et la baisse du coût des processeurs et mémoires ont permis de concevoir des systèmes informatiques dont la structure s'adapte à celle des applications traitées actuellement. Ces applications couvrent des domaines variés : réseaux de communication, systèmes de commande de processus industriels, systèmes de gestion administrative ou documentaire, bases de données distribuées.

1.3.2 Pourquoi des systèmes informatiques distribués.

Des applications géographiquement distribuées ont été traitées dès les débuts de l'informatique : il s'agissait alors essentiellement de relier à un site central des terminaux d'accès ou des organes de mesure munis ou non des possibilités de traitement local. Dans une étape ultérieure, des soucis d'économie (mise en commun de ressources ou d'informations) ont conduit à la connexion, par des réseaux de communication, des installations existantes capables de fournir des services généraux ou spécialisés. Dans un premier temps, ces services différaient peu de ceux accessibles par un terminal connecté à un ordinateur éloigné; puis se mirent en place des applications mettant en jeu un ensemble de services fournis par les différents systèmes connectés. Enfin, il est maintenant courant de concevoir comme un tout un système informatique distribué, c'est-à-dire l'ensemble constitué par le matériel, le logiciel et le système de communication.

Il est maintenant économiquement et techniquement possible de décentraliser les éléments d'un système informatique pour les rapprocher du lieu de production ou d'utilisation des informations traitées. L'évolution technologique permet d'envisager la construction d'ensembles de traitement de grande puissance par assemblage d'un grand nombre de composants élémentaires de conception simple et normalisée. Cette construction pose des problèmes matériels d'interconnexion et des problèmes de mise en oeuvre et d'utilisation (coopération de processeurs multiples, décomposition et répartition des programmes et des données), qui ne sont encore qu'imparfaitement résolus. D'où l'intérêt d'avoir un outil d'administration de tous ces éléments.

1.4 Le modèle TCP/IP

Dans le cadre de ses travaux de définition de standards, l'ISO (International Organization for Standardization) a proposé un modèle de définition de réseau en couches : le modèle OSI (Open Systems Interconnection), souvent connu comme modèle OSI de l'ISO. Pour sa part, le ministère américain de la défense (DoD) patronnait un réseau de recherche dénommé ARPANET. Depuis le démarrage de ce réseau, on cherchait à relier des réseau très divers. Cette architecture finit par être connus sous l'appellation de modèle de référence TCP/IP, du nom de ces deux principaux protocoles. Ce modèle fait une découpe en couches ayant chacune une fonction bien définie et telles que chacune des couches ne soit en relation qu'avec la couche immédiatement supérieure et la couche immédiatement inférieure.

Le modèle de référence TCP/IP propose les couches suivantes : haute-réseau, Internet, transport et application.

1.4.1 La couche haute-réseau

Le modèle de référence TCP/IP ne dit pas grand chose sur ce qui se passe à ce niveau. L'ordinateur hôte doit se connecter au réseau en utilisant un protocole qui lui permette d'envoyer des paquets IP.

Comparé au modèle OSI, cette couche offre les même services que ceux proposés par les couches liaison physique et liaison de données du modèle OSI. La couche liaison physique concerne le système de transmission au niveau le plus bas. Il définit les types de câbles, de connecteurs, le niveau des signaux électriques, etc. La couche liaison de données prend les impulsions et les constitue en trames qui seront fournies à la couche supérieure. Elle va gérer tous les problèmes de réémission, de collision, de duplication, d'erreurs des trames physiques...

1.4.2 La couche Internet

Cette couche permet l'injection de paquets dans n'importe quel réseau et l'acheminement de ces paquets indépendamment les uns des autres jusqu'à destination. Si les paquets arrivent dans un ordre différent de celui d'émission, c'est aux couches supérieure de les réordonner.

Cette couche définit le format de paquet et un protocole IP. Le rôle de la couche Internet est de remettre les paquets IP à qui de droit.

1.4.3 La couche transport.

Elle accepte les données de la couche supérieure et après les avoir redécoupées en trames, elle assure leur acheminement d'une manière fiable.

Cette couche définit deux protocoles de bout en bout : les protocoles TCP (Transmission Control Protocol) et UDP (User Data Protocol).

Le protocole TCP est un protocole fiable et orienté connexion. Il permet la remise sans erreur à une machine appartenant à un réseau un flux d'octets en provenance d'une autre machine. Il fragmente le flux en message (segments) qu'il passe à la couche Internet.

Le protocole UDP permet des liaisons en mode non connecté. Il ne garanti pas la réception et l'arrivée dans un ordre identique à celui d'émission. Il n'y a pas non plus d'accusé de réception. C'est au récepteur de gérer les problèmes de séquencement, de duplication et d'expiration de délais. Ce protocole garantit que s'il y a réception correcte, elle sera d'un coup (un envoi de 2000 octets provoque la réception de 2000 octets et non de 1600 puis 400 octet, par exemple).

Le protocole SNMP utilise les services offerts par le protocole UDP. Le protocole SNMP utilise les ports 161 et 162 sous UDP. Le premier port est utilisé pour les échanges des requêtes et le deuxième port est quant à lui utilisé pour l'échange des trappes.

1.4.4 La couche application.

Elle permet d'offrir des services comme le traitement de fichiers d'une manière transparente, à travers le réseau ou la messagerie électronique.

Dans le modèle OSI de l'ISO, il n'y a pas de communication directe entre deux couches de même niveau. A l'émission, chacune des couches de même niveau va transmettre les informations, encapsulées dans une trame avec en-têtes (headers) qui lui sont propre, à la couche de niveau immédiatement inférieure. Cela jusqu'au niveau physique qui transporte réellement l'information. A la réception, l'opération inverse est effectuée, chaque couche enlève ce qui la concerne avant de transmettre la trame ainsi dépouillée au niveau supérieur.

CHAPITRE 2 : LE PROTOCOLE SNMP (Simple Network Management Protocol).

2.1 Introduction.

Un réseau est constitué de plusieurs ressources. Les différentes ressources du réseau coopèrent pour la réalisation d'une tâche précise. Le réseau doit se doter d'une politique et des moyens nécessaires pour réaliser efficacement ses objectifs. Le protocole snmp (Simple Network Management Protocol) permet de satisfaire cette préoccupation fondamentale d'un réseau. Le rôle de snmp est de gérer un réseau. Il indique la nécessité de supervision, de pouvoir connaître l'état des différents éléments du réseau et une fonction de contrôle par laquelle il est possible de modifier l'état des ressources.

Il existe deux versions du protocole snmp. La première version est définit par la RFC 1155. C'est encore la version la plus utilisée. La version SNMPv2 s'est longtemps fait attendre. Elle n'est apparue qu'en 1996 et elle est définit par la RFC 1905.

Ce chapitre a pour objectif de présenter les concepts de base et la structure de fonctionnement du protocole snmp.

2.2 Présentation du protocole snmp

2.2.1 Configuration de protocole snmp.

Le modèle de l'administration de réseau proposé pour l'administration des réseaux construits selon la suite TCP/IP comprend les éléments suivants :

- la station d'administration
- l'agent d'administration
- la base d'informations d'administration
- le protocole d'administration du réseau

La station d'administration est souvent une station isolée dans un réseau. La station d'administration fournit une interface d'administration à l'administrateur du système.
Elle comporte au minimum :
– un ensemble d'applications d'administration pour l'analyse des données, la récupération des erreurs, etc..
– une interface par laquelle l'administrateur du réseau pourrait piloter et contrôler le réseau
– la capacité de traduire les besoins de l'administrateur et de contrôler les éléments éloignés du réseau
– une base d'informations de toute entité administrée dans le réseau

Un autre élément actif dans le système d'administration de réseau est l'agent d'administration. Les plates-formes principales telles que les terminaux, les ponts, les routeurs et les noeuds, peuvent être dotés du protocole SNMP. Cela permettra leur administration par la station d'administration. L'administration d'un agent concerne sur la possibilité pour la station d'administration de soumettre à cet agent des requêtes pour information et des requêtes pour action. Elle concerne aussi la possibilité pour l'agent de fournir à la station d'administration des informations importantes.

Le moyen par lequel les ressources du réseau peuvent être administrées est de les représenter comme des objets. Un objet est une donnée variable qui représente un aspect de la ressource administrée. La collection des objets est référencée sous le terme de Management Information Base (MIB). La MIB fonctionne comme une collection des points d'accès à la ressource du point de vue de la station d'administration. La station d'administration réalise sa fonction de pilotage en retrouvant la valeur des objets de la MIB. Une station d'administration peut manipuler les valeurs d'une variable spécifique.

La station d'administration et ses agents sont liées par le protocole d'administration de réseau. Le protocole utilisé pour l'administration de réseau et bâtit selon la suite TCP/IP est le SNMP. Ce protocole permet de mettre en oeuvre les requêtes suivantes :

- Get : permet à la station d'administration de retrouver la valeur des objets de l'agent
- Set : permet à la station d'administration d'affecter une valeur à un objet de l'agent
- Trap : permet à un agent de signaler à la station d'administration la survenance d'un événement significatif.

2.2.2 Services offerts par le protocole snmp.

Le protocole snmp tourne au niveau application du modèle TCP/IP. Il utilise les services offerts par le protocole UDP(User Datagram Protocol). La station d'administration et les agents qui animent les différentes ressources du réseau doivent tous implémenter le protocole snmp. Ils sont reliés par le protocole d'administration snmp. Le protocole snmp permet de réaliser les actions suivantes :
- GET : permet à la station d'administration de retrouver les instances des objets d'un agent dans la MIB
- SET : permet à la station d'administration de modifier la valeur de l'instance d'un objet de l'agent dans la MIB
- TRAP : permet à l'agent de signaler la survenance d'un événement inattendu à la station d'administration.

Le protocole snmp est sans connexion. Lors du transfert d'un message snmp, il n'y a pas établissement d'une connexion permanente entre la station d'administration et l'agent. Chaque échange est réalisé d'une manière séparée.

La figure 1 ci-dessous fournit un aperçu sur le contexte du protocole SNMP. A partir de la station d'administration, trois types de messages SNMP sont utilisés par l'application d'administration : GetRequest,GetNextRequest et SetRequest. L'agent répond à ces messages par des messages réponses GetResponse. En plus, un agent peut renvoyer un message Trap pour signaler la survenance d'un événement qui a affecté sa MIB (donc la ressource administrée).

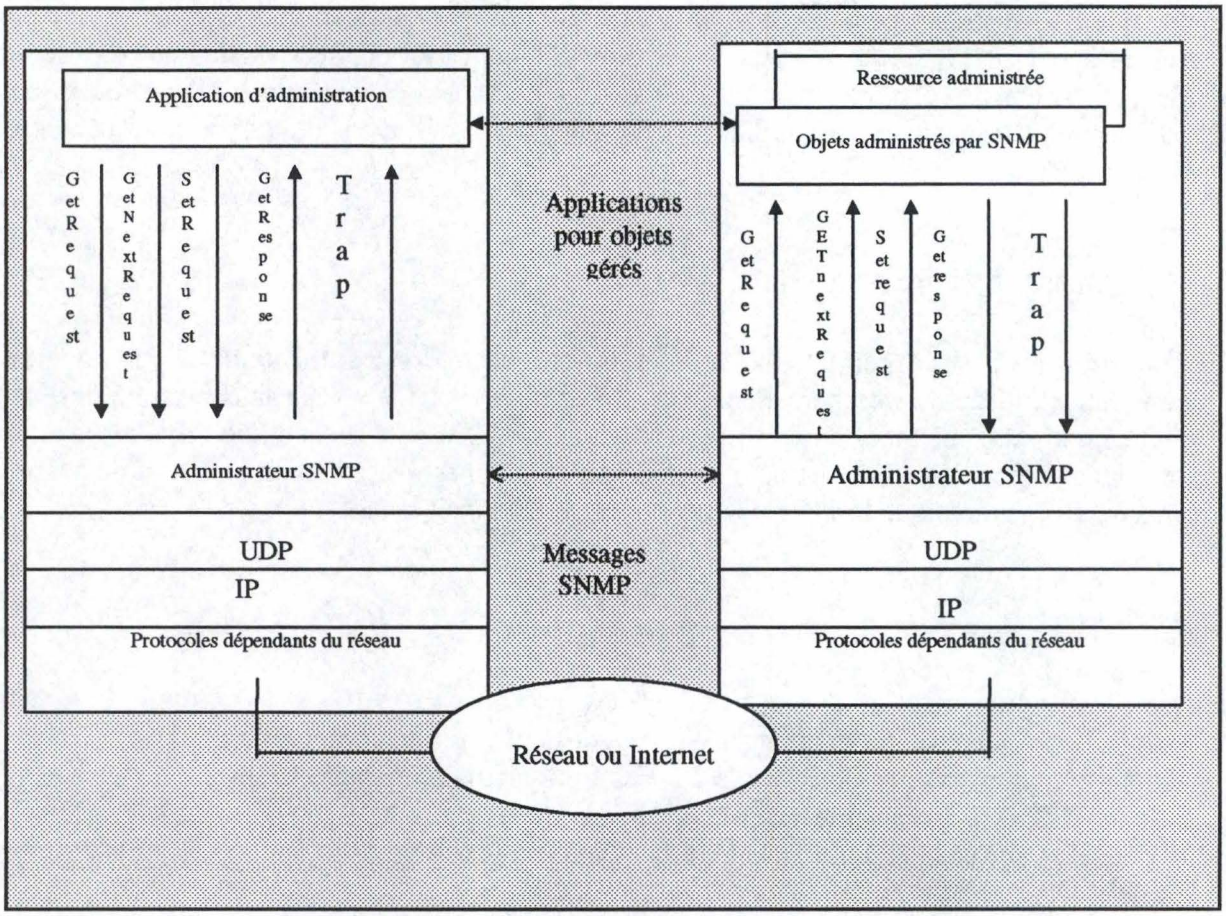


Figure I.2.1 : L'architecture du protocole SNMP

2.2.3 Notion de communauté entre l'agent et les stations d'administration.

L'administration de réseau implique une interaction entre un certain nombre d'entités-applications supportées par un protocole application. Dans le protocole SNMP, les entités applications sont les applications des stations d'administration et les applications des stations administrées (agents) qui utilisent le protocole SNMP.

Une station d'administration établit des relations du type « un à plusieurs » avec un groupe d'agents. La station d'administration est capable de solliciter et de modifier les valeurs des instances des objets des agents et est capable de recevoir des trappes en provenance des agents. D'un point de vu opérationnel et de contrôle, la station administre un certain nombre des agents.

A l'inverse, le protocole snmp permet d'établir une relation du genre « un à plusieurs » entre un agent et un groupe des stations d'administration. Chaque agent contrôle sa propre MIB et contrôle l'utilisation de cette MIB par un certain nombre des stations d'administration.
Le contrôle qu'instaure un agent sur sa MIB porte sur deux aspects :
– L'authentification de service : permet à l'agent de limiter l'accès à sa MIB uniquement aux stations d'administration qui en ont le droit.
– La politique d'accès : permet à l'agent de donner différents privilèges d'accès à différentes stations d'administration.

Une communauté SNMP consiste en une relation définissant une authentification de service et un contrôle d'accès entre un agent SNMP et un groupe des stations d'administration SNMP. Il s'agit d'un concept défini sur l'agent. A chaque communauté établie par l'agent est associé un nom unique et les stations d'administration pour lesquelles cette communauté est déterminées peuvent utiliser ce nom dans chaque opération GET et SET.

2.2.3.1 L'authentification de service.

Une authentification de service permet de s'assurer qu'un échange est authentique. Pour un message SNMP, l'authentification de service permet de s'assurer que le message reçu par une entité destinatrice vient de l'entité émettrice souhaitée. Tout message (SET ou GET) venant de la station d'administration et à destination d'un agent contient un nom de communauté. Ce nom est utilisé comme un mot de passe et le message est sensé être authentique lorsque l'entité émettrice reconnaît le mot de passe.

2.2.3.2 La politique d'accès.

En définissant une communauté, un agent limite les accès à sa MIB à un groupe des stations d'administration. Avec plus d'une communauté, un agent peut fournir différentes catégories d'accès à la MIB aux stations d'administration. Un contrôle d'accès comporte deux caractéristiques :
– la vue de la MIB SNMP : concerne un sous-groupe d'objets de la MIB. Différentes vues de la MIB peuvent être définies pour chaque communauté. Les objets d'une vue peuvent appartenir à plusieurs sous-arbres de la MIB.
– le mode d'accès à SNMP : doit être un élément du groupe suivant {READ-ONLY, READ-WRITE}. Un mode d'accès est défini pour chaque communauté.

2.2.4 Notion de l'ordre lexicographique.

Un identifiant d'objet est une séquence d'entiers reflétant une structure hiérarchique et en arbre des objets de la MIB. Etant donné la structure de l'arbre d'une MIB, l'identifiant d'un objet particulier correspond au chemin représentant un parcours depuis la racine jusqu'à cet objet.

Comme les identifiants d'objets sont des séquences des entiers, ils représentent un ordre lexicographique. Cet ordre peut être généré en traversant l'arbre des identifiants d'objets de la MIB. La raison pour laquelle un ordre entre les identifiants des instances des objets est important est que la station d'administration ne doit pas connaître la vue totale de la MIB qu'un agent lui présente. La station d'administration peut juste parcourir la structure de la MIB pour accéder et rechercher les instances d'un objet dans la MIB sans préciser le nom de cet objet. A partir d'un noeud quelconque de l'arbre de la structure de la MIB, la station d'administration peut atteindre un objet et en atteindre l'instance de l'objet suivant.

L'ordre lexicographique entre les objets et leurs instances dans la table peut être obtenu par un simple parcours de l'arbre.

2.3 Présentation de la MIB snmp.

Pour administrer les différentes ressources du réseau, la station d'administration a besoin d'avoir accès aux informations d'administrations liées à ces ressources. Ces informations représentent les différentes caractéristiques de la ressource administrée. Chaque caractéristique de la ressource représente un objet de la MIB. Une MIB est donc une collection structurer de ces objets. La MIB reflète le statut de la ressource administrée à moment donné. Pour administrer le réseau, la station d'administration doit avoir la possibilité de lire et de modifier les instances des objets de la MIB.

2.3.1. structure de la MIB snmp.

La MIB est construite suivant une structure précise. Chaque objet est identifié par un identifiant d'objet. Un identifiant d'objet permet de donner un nom à la MIB. Il doit donc être unique. Sa valeur est constituée d'une suite d'entier. Ces différents entiers représentent les différents noeuds parcourus à partir de la racine de l'arbre de tous les objets de la MIB. Le sous-arbre constitué des objets se rapportant à une ressource particulière du réseau doit se rattacher à un noeud de l'arbre de tous les objets de la MIB. En partant de la racine de l' arbre de tous les objets de la MIB, on rencontre les noeuds suivants : iso(1), org(3), dod(6), internet(1), private(4) et enterprise(1). Lorsqu'une entreprise veut partager ses informations avec les autres entreprises dans au travers d'un grand réseau, elle doit se rattacher au noeud *enterprise(1.3.6.1.4.1)*. Cette valeur correspond à l'identifiant d'objet du noeud enterprise. Cette valeur servira comme préfixe aux prochains noeuds de bas niveaux de l'arbre. Dans le cas de l'entreprise BULL, les prochains noeuds de bas niveau sont bull(146) et gam(107). C'est à ce dernier noeud que doivent se rattacher les différents sous-arbres des objets se rapportant aux différentes ressources du réseau de cette entreprise.

2.3.2 rôle des objets de la MIB.

Une MIB est constituée d'un groupe d'objets. Un type d'objet défini une caractéristique particulière d'une ressource du réseau. Un type d'objet a une valeur. Une instance d'un objet représente une instanciation d'un type d'objet auquel on a affecté une valeur.

Le RFC 1155 donne la syntaxe des objets que doit contenir une MIB. La définition d'un objet doit contenir les composantes suivantes :
1. Syntax : indique le type de l'objet. Le type peut être universel, d'application ou composé(construit à partir d'un type universel ou d'un type d'application).

2. Access : définit la manière dont on peut accéder à une instance d'un objet. Les options permises sont : read-only, read-write et not-accessible. Dans le dernier cas, la valeur de l'objet ne peut qu'être lue, dans le deuxième cas, elle peut être lue et modifiée, et dans le troisième cas, elle ne peut être lue et modifiée.

3. Status : indique le support d'implémentation exigé pour cet objet. Ce support doit être mandatory ou optional.

4. Description : il s'agit d'une description textuelle de la sémantique du type de l'objet. Cette clause est optionnelle.

Syntaxe d'un objet porte sur la définition du type de l'objet. Un objet peut avoir un type prédéfini ou un type dérivé. Un type dérivé est construit à partir d'un type prédéfini. Les types dérivés permettent de définir certains objets à l'aide des types plus complexes.
Le RFC 1155 détermine les types prédéfinis qui peuvent être utilisés. Il distingue les types d'application et les types et universels.

2.3.2.1 Les types universels.

Seuls les types suivants peuvent être utilisés pour définir les objets de la MIB :
- INTEGER,
- OCTET STRING
- NULL
- OBJECT IDENTIFIER
- SEQUENCE, SEQUENCE OF

2.3.2.2 Les types d'application.

Les types suivants sont disponibles pour la définition des objets d'une MIB :
- IpAddress : il s'agit d'une adresse de 32 bits utilisant le format spécifié dans le protocole IP
- Counter : il s'agit d'un compteur incrémenté de 0 à 2exp32-1 et qui se remet à zéro chaque fois que le maximum est atteint
- TimeTicks : il s'agit d'un compteur de temps. L'unité de base utilisée est le centième de seconde.
- Gauge : il s'agit d'un compteur qu'on peut incrémenter ou decrémenter.

2.3.3 signification des instances d'objets de la MIB.

Les objets de la MIB peuvent être rangés dans une structure sous forme de table à deux dimensions. La définition des tables utilise les types SEQUENCE et SEQUENCE OF. Le constructeur SEQUENCE et SEQUENCE OF permettent respectivement de créer des listes et des tables. Une table contient des colonnes et des lignes. Les colonnes correspondent aux objets de la MIB et les lignes d'une colonne représentent les différentes instances d'un objet quelconque de la MIB.

Pour différencier les différentes colonnes d'une table, on défini un INDEX. Un index détermine la valeur de l'objet qui sera utilisé pour distinguer les colonnes d'une table.

2.4 Spécification du protocole SNMP.

2.4.1 Le format SNMP.

Avec SNMP, l'information est échangée entre une station d'administration et un agent sous forme d'un message SNMP. Chaque message contient un numéro de version indiquant la version du protocole SNMP utilisé, un nom de communauté à utiliser pour cet échange et un des PDU (Protocol Data Units) ci-dessous. Les formats de ces PDU sont donnés dans la figure I.2.3 suivante :

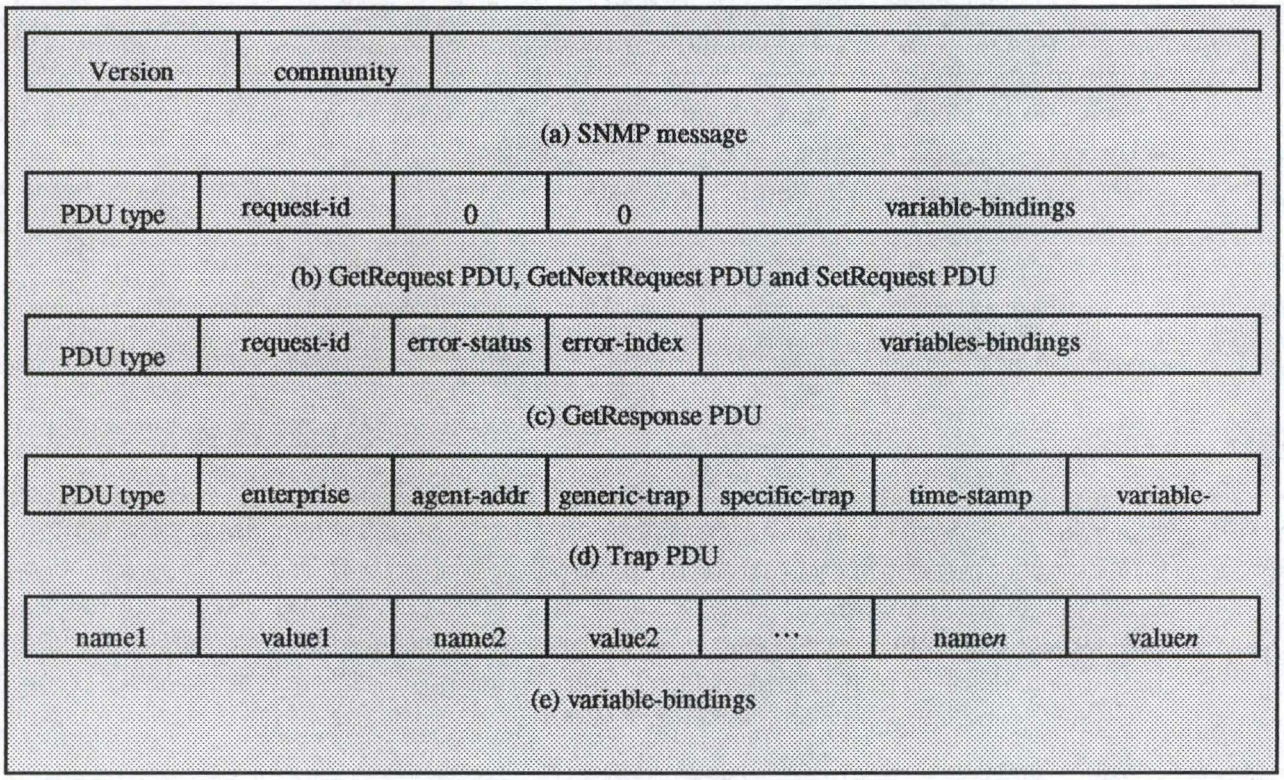


Figure I.2.3 : formats SNMP

2.4.2 La transmission d'un message SNMP.

La trsnsmission d'un message va de l'expédition du message jusqu'à sa réception par l'entité snmp destinatrice. En plus, dans un échange snmp, on a la possibilité de demander le renvoi d'une liste des valeurs dans un seul message grâce à l'uitlisation des variables obligatoires.

2.4.2.1 Expédition du message snmp

En principe, une entité snmp réalise les actions suivantes quand elle veut transmettre un de cinq types des PDU décrits ci-dessus, à une autre entité snmp :

- elle construit le PDU en respectant la structure ASN.1 définit dans le RFC 1155.
- le PDU (auquel on a ajouté les adresses des entités émettrices et destinatrice et un nom de communauté) est ensuite envoyé à un service d'authentification. Le service d'authentification réalise certaines

transformations nécessaires pour cet échange (le codage ou l'ajout de code lié à l'authentification) et renvoi le PDU à l'entité qui l'a émis.
- après la réception du PDU renvoyé par le service d'authentification, l'entité émettrice construit ensuite un message comportant un champ pour la version, un nom de communauté et le PDU construit à l'étape précédente.
- ce message est alors codé et envoyé au service de bas niveau (transport) pour sa transmission.

2.4.2.2 La réception du message snmp.

A la réception d'un message snmp, l'entité snmp destinatrice réalise les actions suivantes :
- elle commence par vérifier la syntaxe du message.
- elle vérifie ensuite que le numéro de version correspond bien à la version du protocole snmp qu'il utilise.
- ensuite seulement elle envoi au service d'authentification le nom de l'utilisateur, le PDU contenu dans le message et les adresses de l'entité émettrice et de l'entité destinatrice.
* si l'authentification échoue, le service d'authentification signale l'échec à l'entité SNMP et celle-ci génère une trappe.
* si l'authentification réussit, le service d'authentification renvoi le PDU sous forme d'un objet.
- elle doit enfin vérifier la syntaxe du PDU. Lorsque la syntaxe est correcte, elle vérifie le nom de communauté et va voir le type de politique d'accès utilisé avant de traiter le PDU .

2.4.2.3 Les variables obligatoires.

Toutes les actions snmp nécessitent un accès à une instance d'un objet. Il est impossible dans le protocole snmp de regrouper un nombre d'opérations de même type (get, set, trap) dans un seul message. Par exemple, lorsque la station d'administration veut obtenir les instances de tous les objets d'un même type, elle peut utiliser les variables obligatoires pour envoyer un seul message sollicitant toutes les valeurs et obtenir en retour une seule réponse listant toutes les valeurs. Cette possibilité peut sensiblement réduire la charge des échanges nécessaires pour l'administration de réseau.

Tous les PDU contiennent des champs pour variables obligatoires afin d'impléménter ces types d'échanges. Ces champs servent à contenir une séquence des références aux instances des objets et ainsi que leurs valeurs. Certains PDU ne contiennent qu'un nom de l'instance de l'objet. Dans ce cas, les valeurs d'entrée dans ce champ sont ignorées par l'entité protocole recevante. Le RFC 1155 recommande dans ce cas que l'entité expéditrice utilise la valeur NULL pour ce champ.

2.4.3 Descriptions des PDU

2.4.3.1 Le PDU Get Request.

Le PDU GetRequest est envoyé vers une entité SNMP par une application de la station d'administration de réseau. L'entité expéditrice remplit les champs suivants dans le PDU :

- le type du PDU : indique qu'il s'agit d'un PDU Get Request
- un request-id : l'entité expéditrice assigne un nombre tel que chaque request en suspens venant d'un même agent soit uniquement identifiée. Le request-id permet à l'application SNMP de corréler la réponse arrivante avec le request en suspens. Il permet aussi à une entité SNMP de gérer les PDU en double, générés par un service de transport non relié.

variables-bindings : une liste des instances des objets dont les valeurs sont sollicitées.

2.4.3.2 Le PDU GetNextRequest.

Le PDU GetNextRequest est presque identique au PDU GetRequest. Ils ont le même format. La seule différence est la suivante : dans le PDU GetRequest, chaque variable de la liste des variable-bindings fait référence à un objet dont on retourne la valeur. Dans le PDU GetNextRequqet, pour chaque variable, on retourne la valeur de l'objet suivant d'après l'ordre lexicographique.

2.4.3.3 Le PDU SetRequest.

Le PDU SetRequest est envoyé par une application de l'entité SNMP représentant la station d'administration. Il a le même format d'échange que le PDU GetRequest. La différence est que le SetRequest est utilisé pour écrire la valeur d'un objet en plus de la lire. Ainsi, la liste des variable-bindings dans le PDU SetRequest contient les identifiants des instances d'objet et une valeur à assigner à chaque instance de la liste.

2.4.3.4 Le PDU Trap.

Le PDU Trap est envoyé à une application de l'entité SNMP représentant la station d'administration. Il est utilisé pour signaler à la station d'administration des notifications asynchrones des événements significatifs. Son format est très différent des celui des autres PDU du protocole SNMP.

III. 2 ème Partie : DEVELOPPEMENT D'UN AGENT SNMP ET DE LA CHAINE AUTOMATIQUE DE TEST DES AGENTS SNMP.

CHAPITRE 1 : DEVELOPPMENT D'UN AGENT SNMP : l'AGENT PING.

1.1 Introduction.

Administrer un système distribué, c'est manager l'ensemble de ses ressources. Pour ce faire, on a recours à une plate-forme d'administration (station d'administration) qui doit avoir une vision aussi complète, fine et détaillée que possible des ressources qu'elle doit gérer.

Cette vision d'une ressource est rendue possible grâce à un modèle de la ressource. La modélisation d'une ressource s'appuie sur une approche et une structuration en objets des informations de management. Ce modèle de la ressource est géré par un agent capable d'instancier des objets à l'aide d'information en provenance de la ressource ou de la plate-forme d'administration. L'agent joue donc un rôle important puisque c'est lui qui est chargé d'entretenir et d'animer le modèle de la ressource.

Dans ce chapitre, nous allons créer un agent fonctionnel gérant une MIB en utilisant le toolkit GAM–OAT (Generic Agent of Management–Open Agent Toolkit) développé par Bull*Soft*. Ceci devrait nous permettre de maîtriser les concepts et techniques liés au développement et au fonctionnement des agents SNMP.

Dans ce travail nous adoptons le formalisme suivant :
- la notation <NomAgent>.RFC, qui désigne le nom d'un fichier, signifiera par exemple qu'on devra remplacer la chaîne <NomAgent> par le nom d'un agent snmp.
- La notation PING désignera le nom de l'agent qu'on va créer et la notation ping désignera la commande UNIX ping.

1.2 Présentation de l'agent SNMP.

De plus en plus d'entreprises travaillent dans un système distribué. Un système distribué est composé de plusieurs ressources qui coopèrent dans la réalisation d'une tâche précise. Ces ressources peuvent être des imprimantes, des stations de travail, des routeurs ou des applications. Il s'agit en fait de toute entité du système d'information. L'ensemble de ces ressources constitue un réseau.

Le rôle du protocole SNMP est de gérer ce réseau. Il nécessite des rôles de supervision, la possibilité de connaître l'état des différents éléments du réseau, et une fonction de contrôle par laquelle, il soit possible de modifier leur état, par exemple relancer un routeur.

Pour cela, on définit deux entités :
– une plate-forme ou station d'administration : elle permet de demander des informations, de les interpréter et, d'une manière générale, d'être une interface avec l'opérateur humain.
– la ressource à superviser dans le réseau : elle est gérée par un agent qui permet de répondre aux sollicitations de la station d'administration et le cas échéant, effectue sur l'élément les opérations demandées.

Notons qu'un agent peut de plus jouer le rôle de proxy pour gérer un élément trop bête pour disposer de son propre agent SNMP, ou tout simplement parce qu'il ne supporte pas ce protocole ou un protocole sous-jacent.

Un agent contient une base de données qui est appelée MIB (Management Information Base). Sa structure et les caractéristiques qui président à sa conception sont appelées SMI (Structure and

Identification of Management Informations). Une MIB contient les instances des objets. Un objet correspond à une information d'administration. La description des objets, leurs types et leurs caractéristiques sont définis à l'aide d'un langage de description appelé ASN.1 (Abstract Syntax Notation One). Un objet est représenté par ses instances. Une instance d'un objet en représente la valeur. Le rôle de SNMP est donc d'obtenir et de positionner des instances des objets se trouvant dans la MIB et de notifier des événements s'y rapportant.

Les ressources qui composent un système sont administrées par une station d'administration. Pour rendre cela possible, un modèle de chaque ressource doit être construit. Un modèle de ressource permet de définir et de déterminer les différents objets qui vont représenter les informations de gestion qu'une ressource doit fournir. Ce modèle est géré par un agent qui doit être en mesure d'instancier les informations se rapportant à la ressource qu'il modélise.

Un agent joue donc un rôle important dans le processus d'administration d'un système. C'est lui qui doit animer la ressource à travers son modèle. Cela souligne l'importance déterminante dans la conception d'un bon modèle.

L'agent s'occupe essentiellement de deux activités suivantes :
– il gère les communications entre la station d'administration et la ressource qu'il modélise
– il manipule les objets dont les instances sont contenues dans la MIB.

Ce qui suppose qu'un agent contient une base de données et des algorithmes lui permettant de manipuler les instances des objets de sa base de données.

1.3 Présentation d'un agent SNMP construit avec GAM-OAT.

L'agent SNMP sur lequel nous allons simuler nos tests sera construit à l'aide du toolkit GAM-OAT développée par BULL*SOFT*. Generic Agent of Management-Open Agent Toolkit (GAM-OAT) permet de construire très simplement des agents fonctionnels SNMP gérant n'importe quelle MIB.

Un agent fonctionnel SNMP construit avec GAM–OAT contient une MIB, un noyau et une ou plusieurs méthodes (voir Figure 1 ci-dessous).

1.3.1 Notion de noyau.

La MIB étant déjà définie, nous ne parlerons que du noyau et des méthodes. Le noyau assume l'émission et la réception des trames SNMP, l'encodage et/ou le décodage en ASN.1 et ainsi que l'accès aux objets de la MIB.

1.3.2 Notion de méthode.

Une méthode est un algorithme qui permet au noyau de manipuler les objets de la MIB. Chaque objet de la MIB est géré par une méthode. Dans le cas où l'agent contient plusieurs méthodes, il faut préciser la méthode qui gère chaque objet (voir la description ci-dessous sur le développement d'un agent à l'aide de GAM–OAT). Lors de la construction de l'agent, son noyau (généré automatiquement) reçoit des méthodes minimales qui lui permettent de rendre l'agent opérationnel dès sa construction. Plutôt que de gérer les valeurs des instances en allant chercher directement les informations dans la ressource physique, ces méthodes implantées par GAM–OAT vont extraire ces informations des bases de données en mémoire. Ces bases de données sont initialisées au lancement du noyau en lisant un fichier d'instances. Les méthodes réelles manipulant de manière significative les instances d'objets de la MIB gérée doivent être développées ensuite. Ces méthodes peuvent être de natures différentes : elles peuvent être des exécutables , des fonctions du système d'exploitation, des scripts shell, des procédures interprétées ou compilées... Elles tourneront en continu ou seront lancées périodiquement en interrogeant la ressource physique afin d'alimenter la base de données du noyau.

1.3.3 développement de l'agent avec GAM–OAT.

Pour développer un agent pouvant gérer une MIB, on doit réaliser les étapes suivantes :

1. fournir au toolkit dans un fichier <agent>.rfc une déclaration de la MIB décrivant les objets modélisant la ressource indiquée dans la syntaxe ASN.1 et un fichier <agent>.confget contenant les indications pour chaque objet de la MIB gérée, de la politique utilisée pour réaliser un GET ou un NEXTGET sur cet objet d'une part et d'autre part, l'identifiant de la méthode qui lui est associé. Ces fichiers doivent être présents sur le répertoire courant.
2. Exécuter la commande de création de l'agent « buildagt [-d|-s] <agent> ». Les options -d ou -s permettent de construire un agent dont l'interface entre le noyau et la méthode est de type direct ou socket. Le toolkit va créer un noyau de l'agent, auquel les méthodes minimales sont déjà implantées.

3. L'agent ainsi créé n'est pas encore opérationnel car le toolkit ne dispose en entrée que des objets définis dans la MIB. Il est impossible de construire de manière générique des méthodes réelles qui peuvent dès le départ initialiser les instances de la MIB. Or il est important de disposer des agents opérationnels dès leurs générations. C'est pour cela que GAM-OAT implante des méthodes minimales au noyau afin de permettre l'initialisation de sa MIB. Cela se réalise par la lecture dans un fichier d'instances <agent>.inst des valeurs nécessaires pour initialiser la MIB et rendre l'agent opérationnel dès le départ. Pour disposer d'un agent réel accédant aux informations de management reflétant la signification sémantique des instances, il faut construire et doter cet agent des méthodes réelles. L'architecture de l'agent GAM-OAT permet d'ajouter des méthodes véritables à son noyau. Comme le noyau de l'agent est structuré autour de la base de données en mémoire, il suffit pour les méthodes de venir réaliser des écritures dans ces bases.

L'architecture complète de l'agent GAM-OAT produit est alors la suivante :

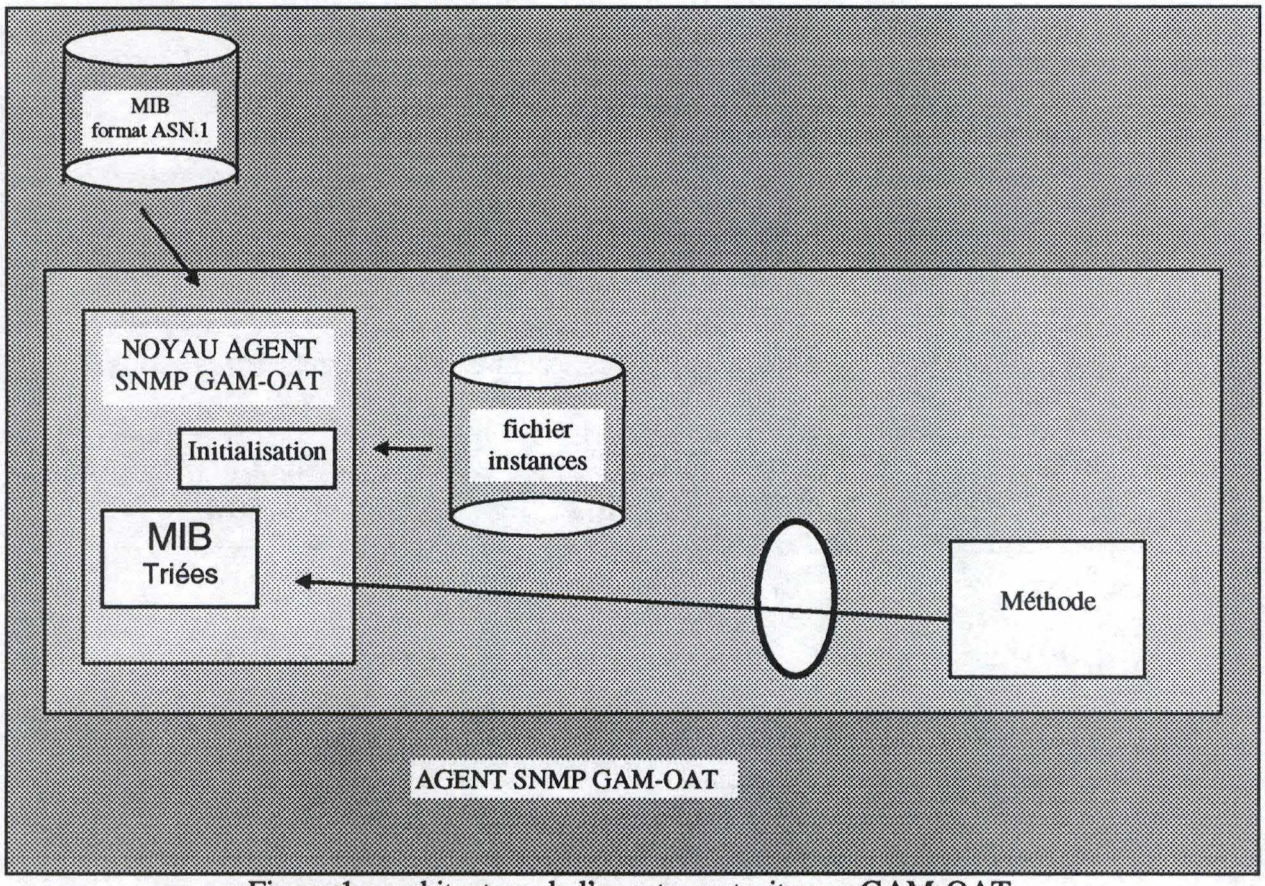


Figure 1 : architecture de l'agent construit avec GAM-OAT

1.4 Construction de l'agent PING.

En vue de disposer d'un agent sur lequel nous allons simuler notre chaîne automatique des tests, nous allons procéder ici à la réalisation d'un agent SNMP en utilisant la technologie GAM-OAT. Cela nous permettra de maîtriser le développement des agents SNMP et des concepts fondamentaux y relatifs.

1.4.1 Présentation.

Supposons qu'on est dans un réseau composé de plusieurs ressources de natures différentes et ayant des fonctions différentes. Ces ressources et leurs adresses IP sont les suivantes :

NOMS	ADRESSES
Belle	129.182.59.94
Wire	129.182.59.155
Virenque	129.182.165.6
Tupa	129.182.165.27
Remus	129.182.165.44
Cloe110	129.182.59.126

Si nous souhaitons ponctuellement avoir la possibilité de vérifier l'état des machines (actives ou pas actives) ou plutôt visualiser l'état des connexions, on peut construire un agent virtuel qui réalise cette fonction et que la station d'administration pourra lancer périodiquement. Le développement d'un tel agent doit se baser sur l'utilisation de la commande UNIX "ping".

La commande "ping" émet un paquet de type ECHO REQUEST en direction d'une ressource et attend en retour de celle-ci un autre paquet marqué ECHO RESPONSE. Cette commande permet de tester le bon fonctionnement d'appareils ou du réseau, dans la mesure où ceux-ci sont capables de répondre au niveau IP.

1.4.1.1 Syntaxe

La syntaxe de la commande PING est la suivante (voir manuel UNIX : commande ***man ping***):

ping [-d] [-n] [-q] [-r] [-v] [-R] [-c Count] [-f | -i Wait] [-l Preload] [-p Pattern] [-s PacketSize] [-L] [-I a.b.c.d] [-T ttl] Host [PacketSize [Count]]

1.4.1.2 Description

Cette commande est utile pour :
- déterminer le statut du réseau et des ressources
- détecter et isoler les problèmes de matériels et logiciels intervenant dans le réseau
tester, mesurer et administrer les réseaux

1.4.1.3 Exemple.

Si dans notre réseau, on veut envoyer trois paquets de 56 octets à la machine Belle à partir de Wire, on lance la commande suivante :

<root>% ping Belle 64 3

La commande ping ci-dessus envoie un paquet de 64 octets en direction de la ressource Virenque toutes les secondes.
Elle affiche au fur et à mesure de la réception des réponses :
– la taille du paquet émis et l'adresse Internet du site distant (respectivement 64 et 129.182.59.94);
– un numéro de séquence, débutant à zéro, qui permet de voir si des paquets sont perdus (icmp_seq=0, 1et 2). On se sert ici du champ séquence, inclus dans l'en-tête des messages ICMP de type ECHO.
– le temps en, millisecondes, de l'aller-retour (Time=40, 20 et 20 ms). C'est-à-dire le temps entre l'envoi du paquet REQUEST et la réception de RESPONSE.
– la durée de vie d'un paquet envoyé dans le réseau (ttl=255 secondes).

Lors de l'arrêt de l'exécution, le nombre de paquets transmis et reçus, ainsi que le pourcentage de paquets perdus, sont affichés (3 packets transmitted, 3 packets received and 0% packets loss). La commande donne les temps minimaux, moyens et maximaux d'émission-réception pour l'ensemble de l'envoi (min /avg/max=20/26/40).
On obtient la réponse suivante (sous UNIX) :

```
Ping Belle.bull.fr : 56 data bytes

64  bytes  from  129.182.59.94  : icmp_seq  =  0.  ttl=255  Time  = 40.  ms
64  bytes  from  129.182.59.94  : icmp_seq  =  1.  ttl=255  Time  = 20.  ms
64  bytes  from  129.182.59.94  : icmp_seq  =  2.  ttl=255  Time  = 20.  ms

        - - -  Belle.bull.fr     PING     Statistics  - - - -

    3 packets   transmitted,  3 packets received,   0%  packets  loss

            round-trip  (ms)     min /avg/max  =  20/26/40
```

Figure 2 : Format de réponse de la commande ping

1.4.2 Création de l'agent PING.

1.4.2.1 déclaration de la MIB.

Ponctuellement, la station d'administration va lancer l'agent pour être imprégné de l'état du réseau. On doit pouvoir lui renvoyer à cet effet les renseignements suivants qui seront contenus dans sa base de données (sa MIB) :

- les adresses Internet de toutes les machines présentes sur le réseau
- les états accessibles ou pas de ces différentes machines
- les nombres des paquets envoyés et perdus sur chaque machine du réseau
- les temps moyens d'envoi des paquets et de retour de l'accusé de réception.

Pour plus d'efficacité et de pertinence, on enverra chaque fois 3 paquets sur chaque machine.

Une analyse objective de ce qui précède nous amène à identifier cinq objets que doit contenir la MIB et dont les différentes valeurs (pour les différentes machines du réseau) en constitueront les instances. Ces objets seront groupés dans une structure sous forme de table. Ces objets et leurs types respectifs sont les suivants :

1. Address : de type IpAddress.
2. State : de type INTEGER. Il prendra la valeur 0 lorsque la machine est inaccessible et la valeur 1 lorsque la machine est accessible.
3. Ptransmis : de type INTEGER qui indique le nombre de paquets envoyés (toujours 3).
4. Pperdu : de type INTEGER qui indique le pourcentage des paquets perdus par rapport au nombre des paquets transmis.
5. Time : de type INTEGER qui indique le temps moyen mis pour envoyer un paquet et en recevoir l'accusé de réception.

La définition d'une table, qui est unique pour toutes les définitions, consiste à utiliser des constructeurs SEQUENCE et SEQUENCE OF comme suite :

- la table **PingmibTable** consiste en une SEQUENCE OF **PingmibEntry**. Le constructeur SEQUENCE OF de la notation ASN.1 comprend 0 ou plusieurs éléments, tout de même type. Dans la table PingmibTable, chaque élément PingmibEntry correspond à une ligne de la table. Ainsi, une table a 0 ou plusieurs lignes.

- Chaque ligne représente une SEQUENCE comprenant 5 valeurs. Aussi, le constructeur SEQUENCE de la notation ASN.1 comprend un nombre fixe des éléments, de types éventuellement différents.

Le composant INDEX de la définition d'une entrée détermine la valeur de l'objet qui permettra de distinguer une colonne des autres colonnes dans la table. Le dernier élément dans la colonne est nécessaire et suffisant pour distinguer sans ambiguïté, une colonne de la table des autres colonnes.

La Figure 3 ci-dessous correspond à la table qui représente les instances des objets de la table **PingmibTable**. Chaque colonne contient les instances d'un objet de la table PingmibTable.

PingmibTable (1.3.6.1.4.1.107.146.999.1)

Address (1.3.6.1.4.1.107.146.999.1.1)	State (1.3.6.1.4.1.107.146.999.1.2)	Ptransmis (1.3.6.1.4.1.107.146.999.1.3)	Pperdu (1.3.6.1.4.1.107.146.999.1.4)	Time (1.3.6.1.4.1.107.146.999.1.5)
129.182.59.94	1	3	0	2
129.182.59.155	1	3	0	2
129.182.165.6	1	3	0	2
129.182.165.27	1	3	0	2
129.182.165.44	1	3	0	2
129.182.175.127	1	3	0	2

Figure 3 : Instances de la table pingmibTable

La table pingmibTable comprend les types d'objets suivants : address (de type IpAddress), state (de type dérivé StateSyntax), ptransmis (de type INTEGER), perdu (de type INTEGER) et time (de type TimeTicks) qui correspondent aux attributs ou colonnes de la table pingmibTable. Les différentes instances des objets correspondent aux valeurs que prennent les attributs de la table.

Nous obtenons la déclaration de la MIB suivante :

```
PINGMIB       DEFINITIONS    ::= BEGIN

Bull          OBJECT IDENTIFIER  ::=  { enterprises 107}
gam           OBJECT IDENTIFIER  ::=  { bull 146        }
pingmib       OBJECT IDENTIFIER  ::=  { gam  999        }
- - -
- - - The SNMP MIB ROOT
- - -
- - - DEFINITIONS DE LA TABLE

StateSyntax    ::= INTEGER {
                        accessible (1),
                          inaccessible (2)
                       }

pingmibTable       OBJECT-TYPE
                        SYNTAX    SEQUENCE OF PingmibEntry
                        ACCESS    not-accessible
                        STATUS    mandatory
                        DESCRIPTION   "Table des instances"
                        ::= { pingmib 1 }

pingmibEntry       OBJECT-TYPE
                        SYNTAX    PingmibEntry
                        ACCESS    not-accessible
                        STATUS    mandatory
                        DESCRIPTION   " Elément d'une entrée"
                        INDEX       { Adress }
                    ::= { pingmibTable 1 }

PingmibEntry       ::= SEQUENCE  {
                              Address       IpAddress,
                              State         StateSyntax,
                              Ptransmis     IINTEGER,
                              Pperdu        INTEGER,
                              Time          Timelicks
                            }

Address                OBJECT-TYPE
                         SYNTAX    IpAddress
                         ACCESS    read-only
                         STATUS    mandatory
                         DESCRIPTION   "Adresse Internet de la machine"
                        ::= { pingmibEntry 1 }

State                  OBJECT-TYPE
                         SYNTAX    IStateSyntax
                         ACCESS    read-only
                         STATUS    mandatory
                         DESCRIPTION   " indique si la machine est accessible ou pas "
                        ::= { pingmibEntry 2 }

Ptransmis              OBJECT-TYPE
                         SYNTAX    INTEGER
                         ACCESS    read-only
                         STATUS    mandatory
                         DESCRIPTION   " indique le nombre des paquets transmis avec  succès "
                        ::= { pingmibEntry 3 }
```

```
pperdu                    OBJECT-TYPE
                          SYNTAX    INITEGER
                          ACCESS    read-only
                          STATUS    mandatory
                          DESCRIPTION   "indique le nombre des paquets perdus durant une transaction"
                          ::= { pingmibEntry 4 }

time                      OBJECT-TYPE
                          SYNTAX    TimeTicks
                          ACCESS    read-only
                          STATUS    mandatory
                          DESCRIPTION   " indique le temps moyen d'une transaction "
                          ::= { pingmibEntry 5 }
END
```

Fichier 1 "pingmib.RFC" : Déclaration de la MIB de l'agent

1.4.2.2 création.

Pour créer l'agent **"PING"**, nous lancerons la commande « buildagt -s pingmib ».
Le toolkit va recherche dans le répertoire courant les fichiers suivants :
– «**pingmib.RFC**» dans lequel on a décrit les objets de la MIB
– «**pingmib.confget**» dans lequel on a indiqué, pour chaque attribut de la MIB gérée, de la politique utilisée pour réaliser un GET ou un SNMP-NEXT sur cet attribut.

Après exécution de la commande, le toolkit va créer :
– **le noyau de l'agent** PING,
– le fichier **rc.Pingmib** qui contient le script de lancement de ce noyau,
– le fichier « **pingmibd.Comm**» qui contient la configuration des communautés (voir chapitre 1 de la première partie) SNMP de l'agent PING et de leurs droits (lecture,écriture)
– le fichier «**pingmib.snmpdef** » définissant les correspondances entre l'OID d'un objet et son nom en toutes lettres.

1.5.2.3 développement de la méthode.

Etant donné l'importance que nous consacrons au développement de la méthode, nous parlerons longuement de cette partie du travail dans la section suivante consacrée au fonctionnement de l'agent PING.

1.5 Fonctionnement de l'agent PING.

Afin de disposer d'un agent réel pouvant accéder aux informations de management, nous devons ajouter une méthode véritable à son noyau. Comme ce noyau est structuré autour des bases de données en mémoire, cette méthode pourra venir réaliser des écritures et lectures dans ces bases. Ainsi, outre la définition d'un modèle de la ressource et la construction automatique du noyau de cet agent PING, on va développer une méthode réelle qu'on va associer aux objets du modèle géré par l'agent.

Notons que cette méthode sera constituée des algorithmes interprétés ou compilés Elles tourneront en continu ou seront lancées périodiquement en interrogeant la ressource physique afin d'alimenter les bases de données du noyau.

Le fonctionnement d'un agent commence par une phase d'initialisation.

1.5.1 Initialisation.

Concernant l'initialisation de la MIB, nous distinguons deux niveaux d'initialisation :

– l'initialisation de la MIB lors du lancement du noyau de l'agent
– l'initialisation de la MIB par la méthode

1.5.1.1 L'initialisation de la MIB lors du lancement du noyau de l'agent

L'initialisation de la MIB lors du lancement du noyau de l'agent se fait lorsqu'on exécute le script de lancement **rc.ping** que le toolkit a crée automatiquement lors de la création de l'agent.
Cette étape commence par l'initialisation de la méthode avant l'initialisation de la MIB proprement dite. L'initialisation de la méthode se fait lors de l'appel à la fonction de l'API **connectSock** dans laquelle la méthode déclare au noyau son identifiant. Comme ça, le noyau saura à quelle méthode correspond les sockets de service SNMP.

1.5.1.2 l'initialisation de la MIB par la méthode

Une fois que cette étape est passée, le toolkit procède à l'initialisation de la base de données du noyau. On fait appel à la procédure **Init_meth**. Au fait, on va lire un fichier d'instances Pingmib.inst pour initialiser les instances de la MIB.

Le fichier pingmib.inst correspond à ceci :

```
address.129.182.59.94     129.182.59.94
address.129.182.59.155    129.182.59.155
address.129.182.165.6     129.182.165.6
address.129.182.165.27    129.182.165.27
address.129.182.165.44    129.182.165.44
address.129.182.175.127   129.182.59.126

state.129.182.59.94       0
state.129.182.59.155      0
state.129.182.165.6       0
state.129.182.165.27      0
state.129.182.165.44      0
state.129.182.59.126      0

Ptransmis.129.182.59.94       0
Ptransmis.129.182.59.155      0
Ptransmis.129.182.165.6       0
Ptransmis.129.182.165.27      0
Ptransmis.129.182.165.44      0
Ptransmis.129.182.59.126      0

Pperdu.129.182.59.94          0
Pperdu.129.182.59.155         0
Pperdu.129.182.165.6          0
Pperdu.129.182.165.27         0
Pperdu.129.182.165.44         0
Pperdu.129.182.59.126         0

Time.129.182.59.94            0
Time.129.182.59.155           0
Time.129.182.165.6            0
Time.129.182.165.27           0
Time.129.182.165.44           0
Time.129.182.59.126           0
```

IB

Dans l'initialisation de la MIB par la méthode, on part d'un fichier « MachineReseau » (voir fichier 3 ci-dessous) dans lequel on stocke le nom de toutes les machines du réseau.

```
Wire
Virenque
Tupa
Remus
Cloe110
```

Fichier 3 MachineReseau : la liste des noms des ressources présente dans le réseau

Puis, sur chacune de ces machines, on lance la commande ping et le résultat est redirigé dans un tube nommé dans lequel on va lire.

Deux cas sont possibles :
– soit la ressource répond
– soit la ressource ne répond pas

Pour traiter le résultat du ping (voir figure 5 ci-dessus), la méthode passe par les étapes suivantes :
1) la méthode lance un programme shell (qui se trouve dans le fichier **Ping.Ksh :** voir annexe 1.5) dans lequel la méthode demande le nom d'une machine sur laquelle elle doit lancer le ping et qu'elle passe comme paramètre à l'appeler d'un autre programme AWK (dans le fichier **Ping.awk :** voir annexe 1.5). C'est ce programme AWK qui va analyser le résultat et renverra la valeur "1" lorsque la ressource répond et la valeur "2" lorsque la ressource ne répond pas. Dans le premier cas, le pourcentage des paquets perdus durant l'échange et les temps minimum, moyen et maximum de l'échange sont également fournis.

2) Après le lancement d'un Ping, la méthode obtient en retour l'adresse Internet de la machine vers laquelle on a expédié trois paquets, son état (accessible ou pas accessible), le nombre des paquets transmis (toujours 3), le pourcentage des paquets perdus, et ainsi que le temps moyen mis pour échanger ces paquets. Pour chaque ressource, une instance de chaque objet est écrite dans la MIB en utilisant la fonction de l'API **writeInstSock**.
3) La mise à jour de la MIB est faîte après le traitement de chaque requête SNMP reçue.

1.5.2 Implémentation des échanges.

Lorsque le manager veut par exemple savoir si une ressource precise est active, il envoi une requête de consultation au noyau de l'agent qui réalise cette fonction (ici agent PING). Le noyau transmet à son tour cette requête à sa méthode et c'est cette dernière qui ira questionner la ressource afin de savoir si elle est active ou pas.

Pour bien comprendre le fonctionnement des mécanismes qui permettent de mettre en oeuvre les requêtes SNMP, nous devons distinguer deux types d'échanges :
1) Echanges entre le noyau de l'agent et la méthode
2) Echanges entre le manager et la ressource administrée

1.5.2.1 Echanges entre le noyau de l'agent et la méthode.

Les échanges entre le noyau et sa méthode sont réalisées sous forme d'échanges méthodes, c'est-à-dire que ce sont des échanges qui sont mis en oeuvre à la demande de la méthode lorsque la ressource gérée par l'agent modifie un indicateur qui la caractérise, ou lorsque la méthode a besoin de consulter la valeur d'une instance gérée par le noyau.

Pour implémenter notre méthode, nous utiliserons deux types d'échanges entre le noyau et cette méthode (voir figure 4 ci-dessous):
1) nous aurons les échanges SNMP (échanges protocolaires), qui relèveront de l'initiative du noyau qui transmet aux méthodes les requêtes SNMP provenant du manager
2) les échanges méthodes, à l'initiative des méthodes pour accéder aux bases de données triées du noyau.

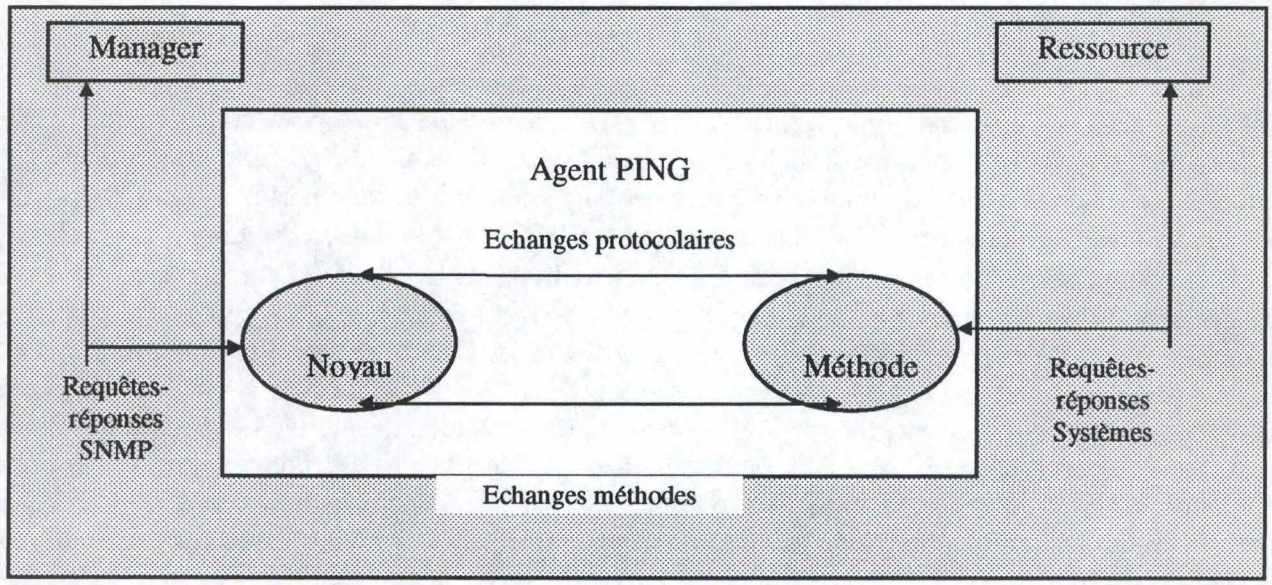


Figure 4 : Présentation des types échanges

1.5.2.1.1 les échanges protocolaires.

L'agent PING permet l'exécution, en parfaite conformité avec la norme SNMP, des primitives suivantes :
- GET : réalise la consultation d'une instance d'un objet
- GET–NEXT : réalise la consultation de l'instance de l'objet suivant par rapport à un ordre lexicographique
- SET : réalise la modification d'une instance d'un objet

Ces primitives sont supportées par les PDU suivants :
- SNMP–get–request
- SNMP–get–next–request
- SNMP–set–request
- SNMP–get–response

L'agent PING permet aussi d'émuler les verbes qui n'appartiennent pas obligatoirement à la norme SNMP, mais qui, vu leurs utilisations fréquentes dans le management des systèmes distribués, ont été introduits dans d'autres protocoles comme CMIP. Il s'agit des primitives suivantes :
- CREATE : création d'une instance dans une table (introduction d'une nouvelle adresse dans la table)
- DELETE : suppression d'une instance dans une table (suppression d'une adresse dans la table)

Ces deux nouvelles primitives sont supportées par les PDU suivants :
- SNMP–set–request
- SNMP–get–response

Pour implémenter ces transactions, deux modes d'interface d'échange sont possibles :
a) échanges protocolaires en mode interface directe
b) échanges protocolaires en mode interface sockets

a) <u>échanges protocolaires en mode interface directe</u>

Dans ce mode, les échanges protocolaires entre le noyau de l'agent et la méthode sont réalisés en utilisant des tubes nommés et les envois du signal SIGUSR2. L'utilisation du signal SIGUSR2 permet au noyau de l'agent et à la méthode de se consacrer à d'autres tâches.

Chaque écriture dans un tube doit être suivie d'un envoi du signal SIGUSR2 afin d'avertir le destinataire de la transmission d'un message. Afin de permettre au noyau et à la méthode de travailler indépendamment l'un de l'autre, les ouvertures des tubes doivent se faire dans un mode non bloquant.

Le toolkit GAM–OAT fournit à cet effet des primitives permettant de réaliser des lectures et des écritures dans le tube.

b) <u>échanges protocolaires en mode interface sockets</u>

Dans le mode interface sockets, le noyau et la méthode communiquent par l'intermédiaire des sockets réseau connectées, c'est-à-dire supportées au niveau transport par la couche TCP.

Les sockets sont des objets destinés à la communication entre processus dans différents espaces de définition appelés domaines. Elles peuvent être utilisées pour la communication entre processus s'exécutant sur la même machine, on utilise alors le domaine UNIX, ou bien si ces processus s'exécutent sur deux machines distantes reliées entre elles par l'Internet, on utilise alors des sockets sur le domaine Internet.

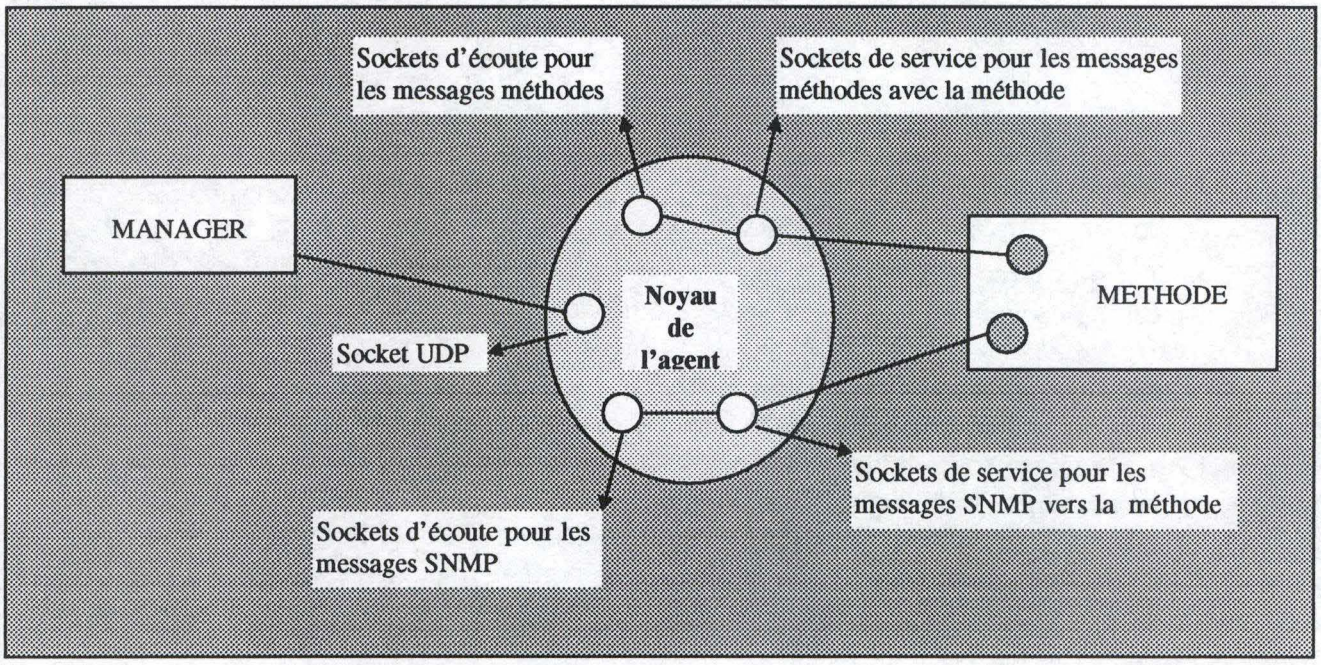


Figure 5 : présentation des sockets mis en jeu pour les échanges

L'architecture de notre agent PING, tout comme celle de tout agent développé avec GAM–OAT en interface sockets, utilise deux ports TCP sur lesquels le noyau crée deux sockets d'écoute. Ces ports TCP sont déclarés en utilisant un fichier « **services** » dans lequel on indique les noms de services Ping–meth et Ping–snmp.

Soit le contenu suivant : <Ping>_meth 1400/tcp
<Ping>_snmp 1401/tcp

1.5.2.1.2 les échanges méthodes.

Les échanges méthodes consistent à des lectures et écritures d'instances par la méthode. Ces lectures (READINST) et écritures (WRITEINST) se font directement dans les bases de données internes du noyau.

a) Les échanges méthodes en mode interface directe

se font par le biais d'envoi de message dans un tube nommé, couplé à l'émission d'un signal SIGUSR1. L'envoi du message et du signal SIGUSR1 est géré dans les fonctions et primitives de l'API de GAM–OAT **readInstDirect** ou **readNextDirect**.

Lors d'une réponse à un readInstDirect ou readNextDirect, le noyau ne renvoi pas de signal à la méthode. Cela permet au processus méthode de disposer du signal SIGUSR1 pour d'éventuelles interceptions indépendantes de GAM–OAT.

Il est possible pour la méthode de lire dans les bases internes du noyau l'instance suivante dans l'ordre lexicographique (READNEXT). Ce qui permet à la méthode de lire l'intégralité des instances d'une table.

b) Dans les échanges méthodes en mode interface sockets

Le noyau se met en écoute sur une liste de sockets. Lorsqu'aucune méthode n'est encore connectée, les sockets qui font partie de cette liste d'écoute sont :
- la socket d'écoute méthode
- la socket UDP

Lorsqu'une méthode se connecte, un événement se produit sur la socket d'écoute méthode, dès lors, le noyau ouvre une socket de service méthode et attend la connexion de la méthode sur la socket d'écoute SNMP. Une fois que la connexion est établie, le noyau ouvre une socket de service SNMP, avant de reprendre son écoute en ajoutant la nouvelle socket de service méthode à la liste d'écoute.

La déconnexion d'une méthode entraîne la fermeture de deux sockets de service qui lui sont associées et le retrait de la socket de service méthode de la liste d'écoute. Lorsqu'un message arrive sur une socket de service méthode, le noyau traite la requête avant de reprendre son attente sur la liste d'écoute. De même, lorsque le noyau reçoit sur la socket UDP une requête provenant du manager, il réalisera le traitement de cette requête avant de reprendre son attente sur la liste d'écoute. Ainsi, si la requête du manager doit être transmise à une méthode, le noyau attendra la réponse de cette dernière avec un timeout. Il reprendra son attente sur la liste d'écoute lorsque la méthode aura répondu ou que le timeout aura expiré. Durant cette attente de réponse, tout message, qu'il provienne du manager ou d'une méthode, est bufferisé sur la socket par laquelle il est transmis. On s'assure ainsi qu'il n'y aura aucune perte de requête.

Le mode interface sockets permet ainsi de faire cohabiter plusieurs méthodes avec le noyau d'un agent. Le noyau de l'agent PING doit avoir la possibilité de savoir à quelle méthode il doit transmettre une requête provenant du manager. Pour cela, le noyau utilise les identifiants des méthodes. Il s'agit d'un entier qui permet de distinguer de manière unique une méthode. De sorte que, lors de l'initialisation d'une méthode, celle-ci déclare au noyau son identifiant de manière à ce qu'il sache à quelle méthode correspond telle socket

de service SNMP. Cette déclaration est directement réalisée dans la fonction de l'API GAM–OAT : connectSock. Etant donné que l'agent PING n'a qu'une méthode, cela est plus aisé ici.

1.5.2.2 Echanges entre le noyau de l'agent et la station d'administration.

Les échanges entre le noyau d'un agent et le manager concernent les communications entre un manager et une ressource administrée par un agent construit avec GAM–OAT. Ces communications peuvent porter sur les consultations, les modifications, les créations et les suppressions. Ces opérations se font à l'aide des chaînes de liaison mises en oeuvre dans un agent construit avec GAM–OAT.

1.5.2.2.1 Les requêtes de consultation : GET et GET–NEXT.

Suite à une requête de consultation d'un manager, la valeur de l'instance à retourner peut-être soit directement lue dans les bases de données internes du noyau, soit soumise à la méthode. Dans l'implémentation de la méthode de l'agent PING, nous avons opté pour la soumission de la requête à la méthode. Ce qui permet de rafraîchir les différentes valeurs des instances de la MIB et donc, cela permet de garder dans la MIB les informations d'administration les plus récente à propos de la ressource administrée.

La méthode reçoit la requête en provenance du noyau de l'agent SNMP grâce à une fonction de l'API **receiveSock**. Dès qu'elle reçoit un message SNMP de consultation d'une instance, la méthode provoque la mise à jour des toutes les instances des objets de la MIB avant de rechercher et retourner la valeur demandée. Les figures suivantes présentent la chaîne de liaison complète, lors de l'envoi d'une requête **Snmpget** ou **Snmpgetnext** par le manager.

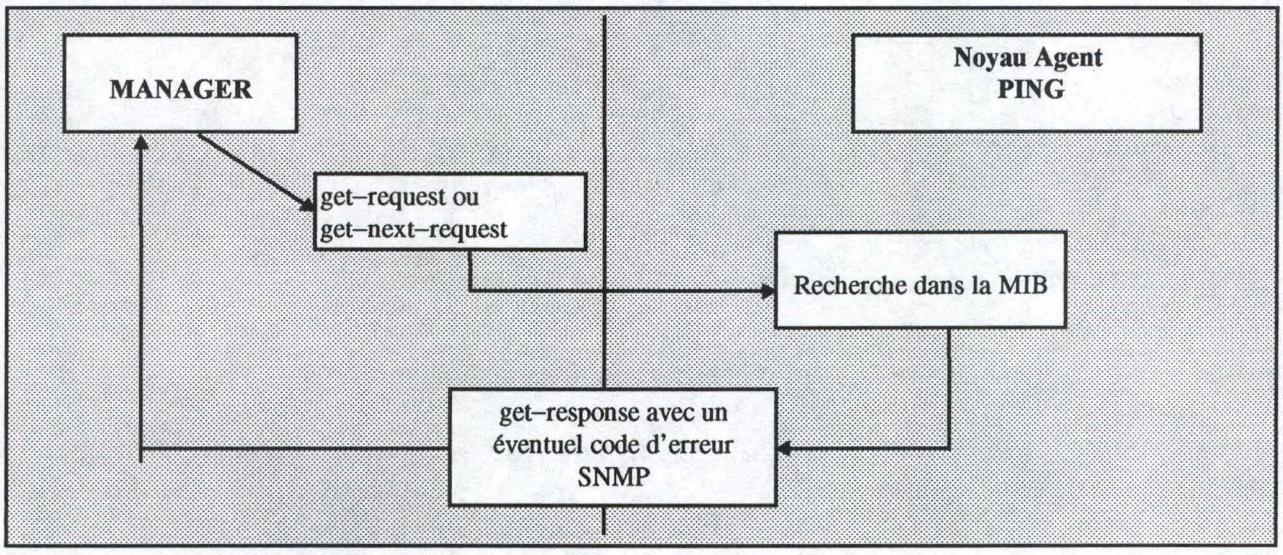


Figure 6 : **get** ou **get–next** en mode interface direct

Déroulement :

1. Pour réaliser la consultation d'une instance d'un objet, le manager envoi une requête snmpget au noyau de l'agent en indiquant l'instance qu'il veut consulter.
2. Trois cas peuvent se présenter à ce niveau :
a. la requête contient une erreur : un message d'erreur est retourné.
b. le noyau ne répond pas après un timeout : il y a renvoi d'un ger-response avec éventuel code d'erreur.
c. il n'y a pas d'erreur et la requête est bien transmise à la méthode
3. La méthode reçoit la requête et va interroger la ressource physique.
4. La ressource répond positivement ou négativement. Le code de retour est positionné à OK ou NOK.
5. En fonction de la réponse de la ressource, la méthode envoi un message de réponse au noyau.
6. Suivant le code retour, le noyau
a. met à jour toutes les instances dans la MIB (code retour=OK) avant de renvoyer le get-response au manager avec la valeur la plus fraîche.
b. renvoi le get-response avec code d'erreur (ce retour=NOK).

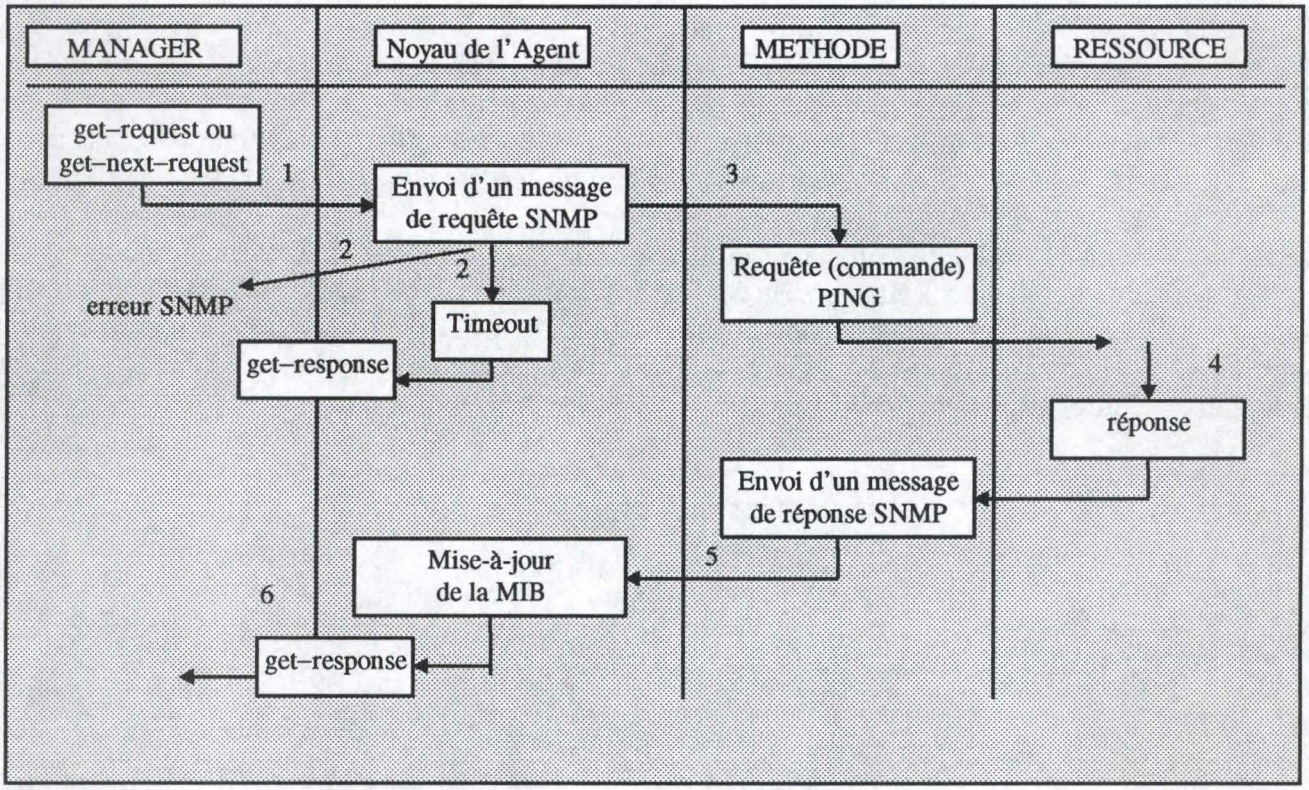


Figure 7 : **get** ou **get–next** en mode interface sockets

Dans le cas d'un **get-next** provenant du manager, la recherche de l'instance suivante est réalisée par le noyau.

1.5.2.2.2 requête de modification : SET.

La requête de modification porte sur la modification de la valeur d'une instance de la MIB. Dans le cas de l'agent PING, cette requête porte sur la modification d'une addresse dans la table PingmibTable. A la réception de la requête de modification, la méthode va modifier la valeur de l'adresse dans la table et lance une commande Ping sur la machine associée à cette adresse. Puis seulement les valeurs des instances concernées par cette adresse sont réactualisées.

Déroulement :

1. Pour réaliser la modification de l'instance d'un objet, le manager envoi une requête snmpset au noyau de l'agent en indiquant l'instance qu'il veut modifier, son type et la nouvelle valeur.
2. Trois cas peuvent se présenter à ce niveau :
a. la requête contient une erreur : un message d'erreur est retourné.
b. le noyau ne répond pas après un timeout : il y a renvoi d'un get-response avec éventuel code d'erreur.
c. il n'y a pas d'erreur et la requête est bien transmise à la méthode
3. La méthode reçoit la requête et va interroger la ressource physique.
4. La ressource répond positivement ou négativement. Le code de retour est positionné à OK ou NOK.
5. En fonction de la réponse de la ressource, la méthode envoi un message de réponse au noyau.
6. Suivant le code retour, le noyau
a. modifie l'instance dans la MIB (code retour=OK) avant de renvoyer le get-response au manager

a. renvoi le get-response avec code d'erreur (ce retour=NOK).

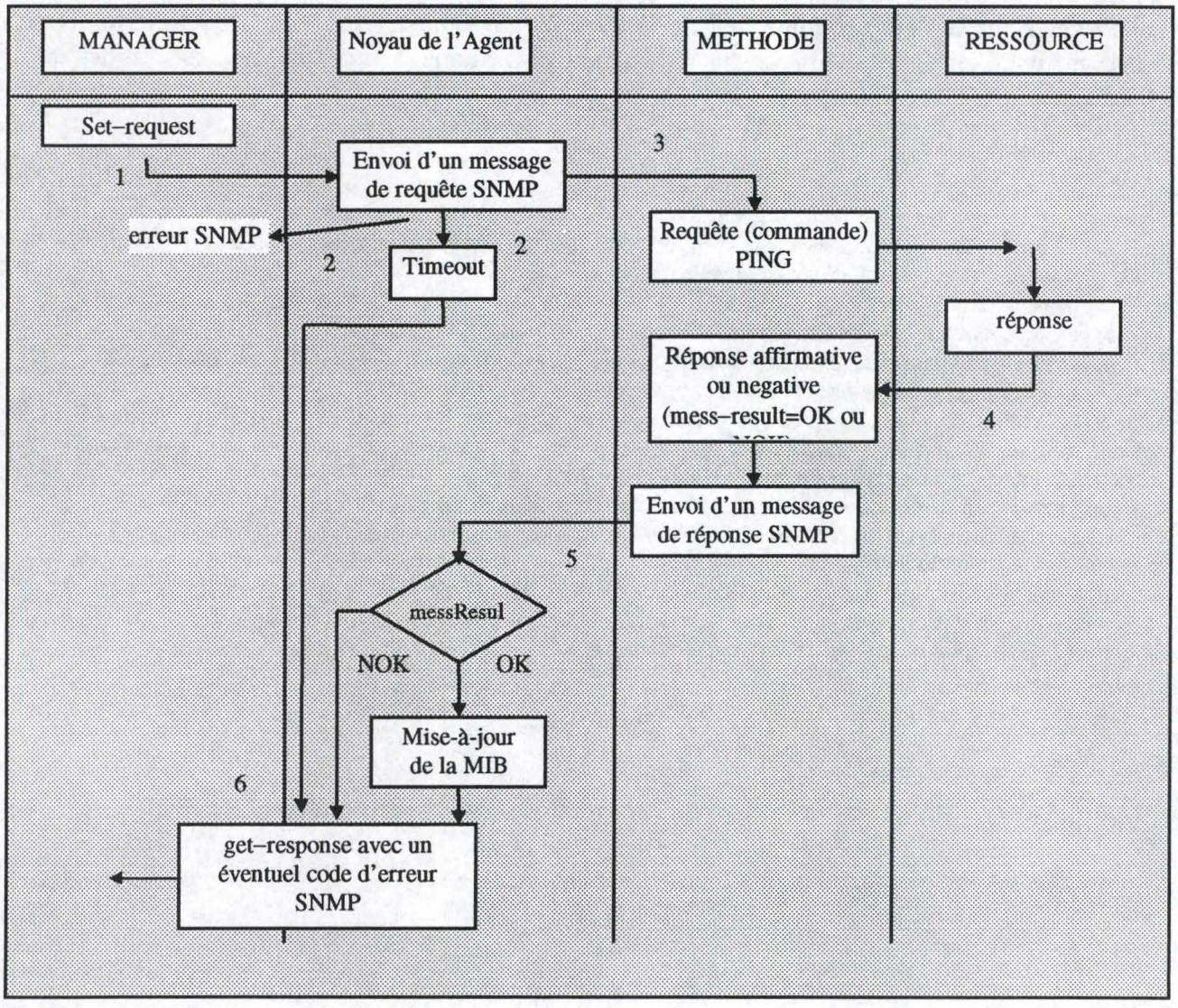


Figure 8 : **set**

1.5.2.2.3 requête de création : CREATE.

La chaîne complète lors d'un create lancé par le manager se traduit par l'envoi d'une requête Snmpset portant sur une instance non existante dans la table des objets de la MIB.

Pour l'agent PING, la création de l'instance d'un objet correspond à l'ajout dans la table PingmibTable d'une adresse de type IpAddress. Le problème à ce niveau est qu'il faut ajouter dans le fichier « MachineReseau » (qui contient les noms de toutes les machines présentes dans réseau) le nom en toute lettre de la machine (encore inconnu car on ne connaît que l'adresse) dont on vient d'ajouter l'adresse dans la table. Pour retrouver le nom en toutes lettres à partir de l'adresse, on utilise la fonction UNIX getHostByName.

A la réception de la requête Snmpset en vue d'une création, la méthode commence par convertir l'adresse Ip de la machine en son nom en toutes lettres. Puis, ce nom est ajouté dans le fichier « **MachineReseau** ». Enfin, la méthode lance l'initialisation de la MIB par la méthode. C'est à ce moment que les instances relatives à la nouvelle machine seront ajoutées dans la MIB.

Déroulement :

1. Pour réaliser la création d'une instance d'un objet, le manager envoi une requête snmpset au noyau de l'agent en indiquant l'instance qu'il veut créer. Dans le cas de notre agent, il ne peut qu'ajouter une nouvelle ressource en donnant l'instance address +l'adresse IP de cette nouvelle ressource.
2. Trois cas peuvent se présenter à ce niveau :
   a. la requête contient une erreur : un message d'erreur est retourné.
   b. le noyau ne répond pas après un timeout : il y a renvoi d'un ger-response avec éventuel code d'erreur.
   c. il n'y a pas d'erreur et la requête est bien transmise à la méthode
3. A la réception de la requête, la méthode va rechercher le nom en toute lettre de la ressource qu'on ajoute en fonction de son adresse IP et l'ajoute dans le fichier qui contient les noms de toutes les ressources présentes dans le réseau.
4. La méthode interroge toutes ces ressources en lançant une série des ping.
5. Les ressources répondent positivement ou négativement. Le code de retour est positionné à OK ou NOK.
6. En fonction des réponses de ces ressources, la méthode envoi des messages de réponse au noyau.
7. Suivant le code retour, le noyau
   a. met à jour les instances dans la MIB (code retour=OK) avant de renvoyer le get-response au manager. C'est à ce moment que les valeurs se rapportant sur la ressource qu'on vient d'ajouter seront instanciées.
   b. renvoi le get-response avec code d'erreur (ce retour=NOK).
8. Le noyau renvoi le get-response au manager

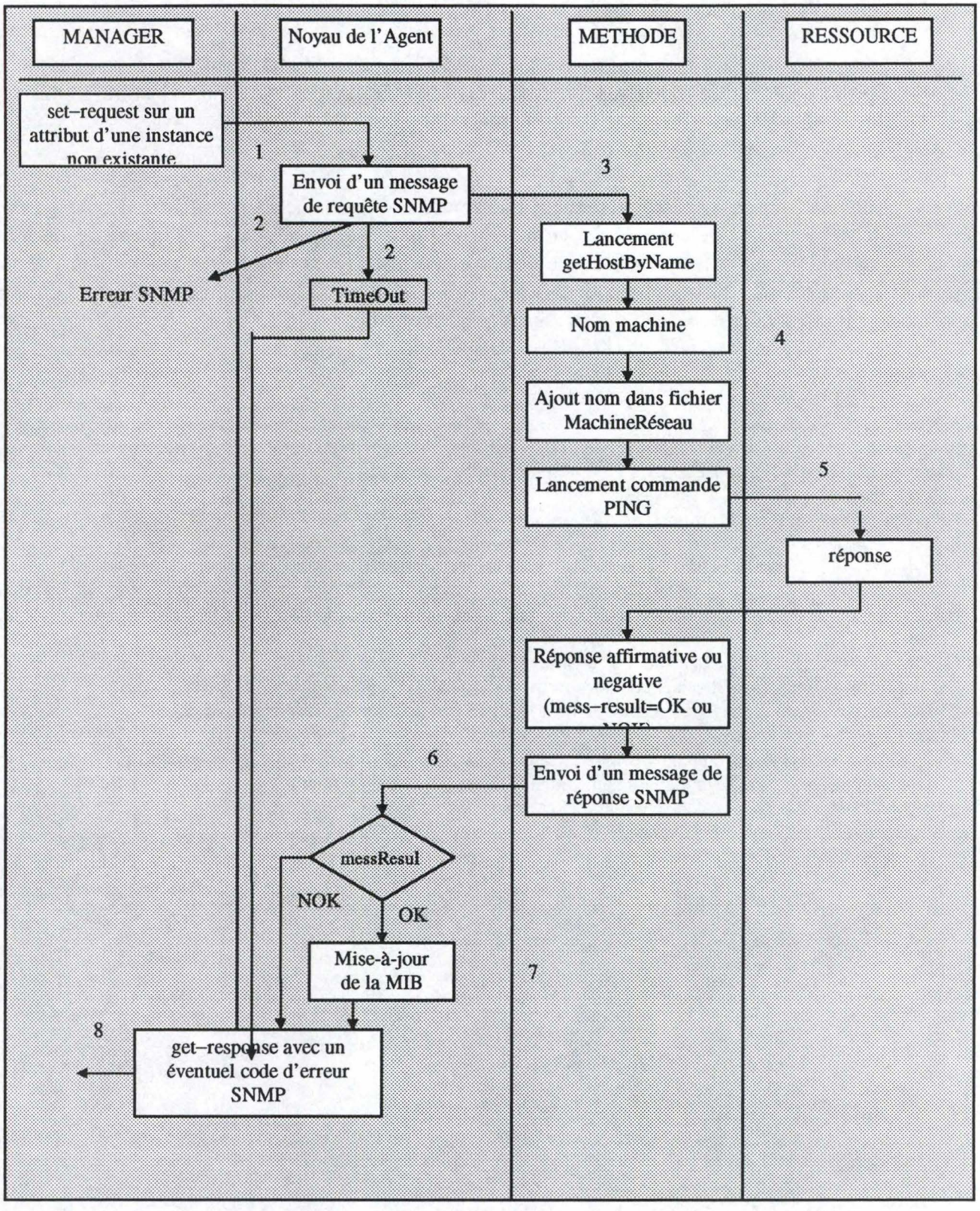


Figure 9 : Create

1.5.2.2.4 requête de suppression : DELETE.

La chaîne de liaison complète d'un DELETE (requête Snmpset en vue d'une suppression) envoyé par le manager se traduit par la suppression dans la MIB de l'instance d'un objet qui représente un attribut qu'on veut supprimer à l'aide d'une requête Snmpset. Ici également, on gère le problème de conversion de l'adresse en nom en toutes lettres.

Lorsque la méthode reçoit la requête, elle commence par détacher le suffixe de l'OID qu'on veut supprimer. Cela lui permet de retrouver le nom en toutes lettres de la ressource qu'on veut supprimer du réseau. La méthode supprime ce nom dans le fichier « **MachineReseau** » qui contient les noms de toutes les ressources du réseau. Puis, la méthode relance son initialisation. Ce n'est qu'à ce moment que toutes les instances relatives à cette ressource seront effacées de la MIB.

Déroulement :

1. Pour réaliser la suppression d'une instance d'un objet, le manager envoi une requête snmpset au noyau de l'agent en indiquant l'instance qu'il veut supprimer et la valeur NULL.. Dans le cas de notre agent, il ne peut que supprimer une ressource en donnant l'instance address +l'adresse IP de cette ressource.
2. Trois cas peuvent se présenter à ce niveau :
   a. la requête contient une erreur : un message d'erreur est retourné.
   b. le noyau ne répond pas après un timeout : il y a renvoi d'un ger-response avec éventuel code d'erreur.
   c. il n'y a pas d'erreur et la requête est bien transmise à la méthode
3. A la réception de la requête, la méthode va rechercher le nom en toutes lettres de la ressource qu'on veut supprimer en fonction de son adresse IP et la supprimer dans le fichier qui contient les noms de toutes les ressources présentes dans le réseau.
4. La méthode interroge toutes ces ressources en lançant une série des ping.
5. Les ressources répondent positivement ou négativement. Le code de retour est positionné à OK ou NOK.
6. En fonction des réponses de ces ressources, la méthode envoi des messages de réponse au noyau.
7. Suivant le code retour, le noyau
   a. met à jour les instances dans la MIB (code retour=OK) avant de renvoyer le get-response au manager. C'est à ce moment que les valeurs se rapportant à la ressource qu'on vient de supprimer ne seront pas instanciées.
   b. renvoi le get-response avec code d'erreur (ce retour=NOK).
8. Le noyau renvoi le get-response au manager

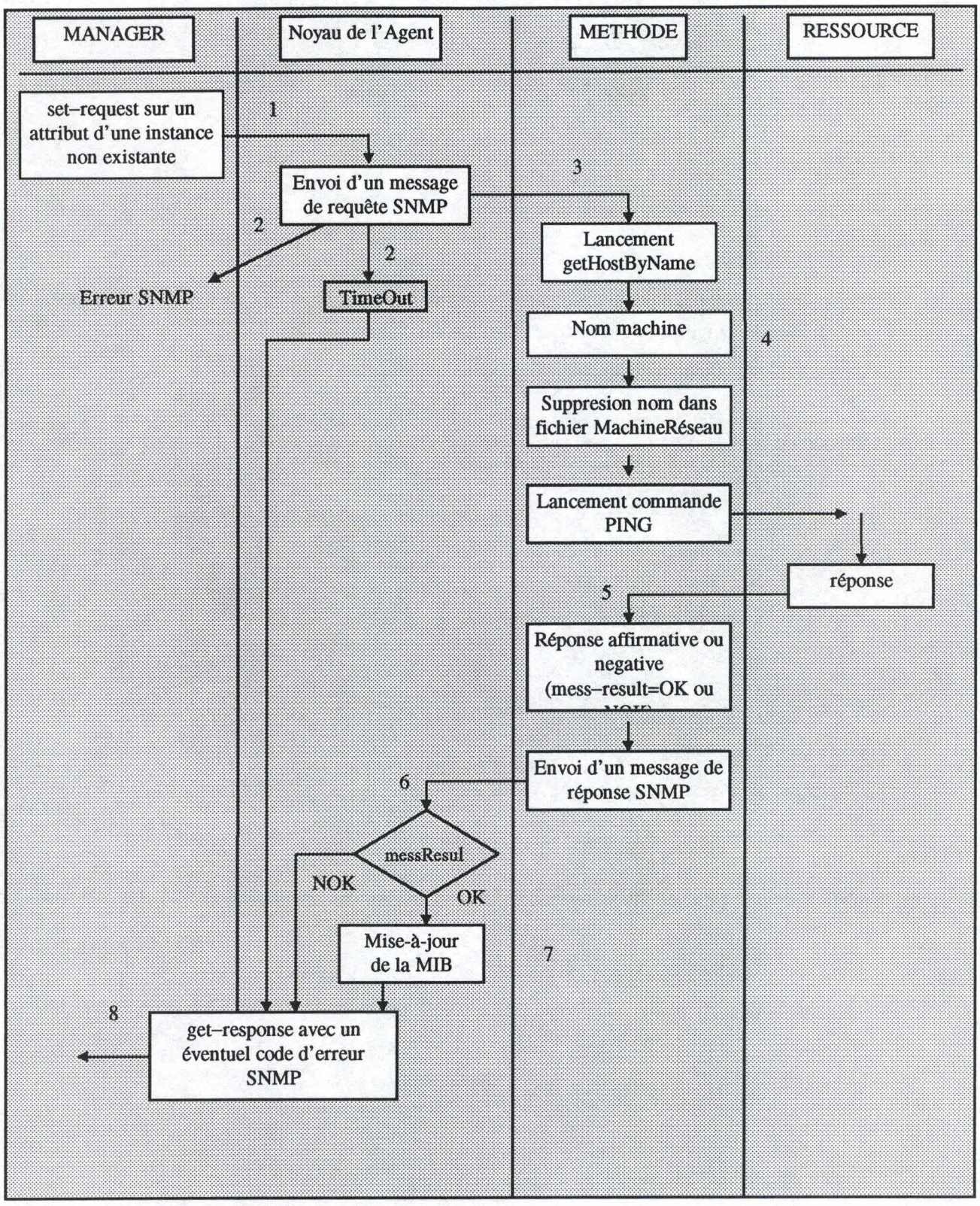


Figure 10 : Delete

1.6 Simulation du fonctionnement de l'agent PING.

Pour simuler le fonctionnement de l'agent PING, nous allons lancer une série des requêtes SNMP pour voir comment réagit notre agent. Il s'agit des requêtes suivantes :

1) snmpmany
2) snmpget
3) snmpset

1.6.1 snmpmany

cette requête permet d'obtenir toutes les instances des objets de la MIB. Si l'on suppose que notre réseau est composé des machines suivantes (avec leurs adresses Internet) :

NOMS	ADRESSE
Belle	129.182.59.94
Wire	129.182.59.155
Virenque	129.182.165.6
Tupa	129.182.165.27
Remus	129.182.165.44
Zoe2000	129.182.175.127

La requête SNMP suivante permet d'obtenir toutes les instances des objets de l'agent PING :

>% snmpmany PING

Le résultat correspond au contenu du fichier suivant :

```
Name :   address.129.182.59.94    Kind : IpAddress    Value : 129.182.59.94
Name :  address.129.182.59.155    Kind : IpAddress    Value : 129.182.59.155
Name :  address.129.182.165.6     Kind : IpAddress    Value : 129.182.165.6
Name :  address.129.182.165.27    Kind : IpAddress    Value : 129.182.165.27
Name :  address.129.182.165.44    Kind : IpAddress    Value : 129.182.165.44
Name :  address.129.182.59.126    Kind : IpAddress    Value : 129.182.59.126

Name :  state.129.182.59.94       Kind : StateSyntax  Value :  1
Name :  state.129.182.59.155      Kind : StateSyntax  Value :  1
Name :  state.129.182.165.6       Kind : StateSyntax  Value :  1
Name :  state.129.182.165.27      Kind : StateSyntax  Value :  1
Name :  state.129.182.165.44      Kind : StateSyntax  Value :  1
Name :  state.129.182.59.126      Kind : StateSyntax  Value :  1

Name :  Ptransmis.129.182.59.94       Kind : Integer   Value :  3
Name :  Ptransmis.129.182.59.155      Kind : Integer   Value :  3
Name :  Ptransmis.129.182.165.6       Kind : Iinteger  Value :  3
Name :  Ptransmis.129.182.165.27      Kind : Integer   Value :  3
Name :  Ptransmis.129.182.165.44      Kind : Integer   Value :  3
Name :  Ptransmis.129.182.59.126      Kind : Integer   Value :  3

Name :  Pperdu.129.182.59.94      Kind : Integer   Value :  0
Name :  Pperdu.129.182.59.155     Kind : Integer   Value :  0
Name :  Pperdu.129.182.165.6      Kind : Iinteger  Value :  0
Name :  Pperdu.129.182.165.27     Kind : Integer   Value :  0
Name :  Pperdu.129.182.165.44     Kind : Integer   Value :  0
Name :  Pperdu.129.182.59.126     Kind : Integer   Value :  0

Name :  Time.129.182.59.94        Kind : TimeTicks  Value :  2
Name :  Time.129.182.59.155       Kind : TimeTicks  Value :  3
Name :  Time.129.182.165.6        Kind : TimeTicks  Value :  8
Name :  Time.129.182.165.27       Kind : TimeTicks  Value :  2
Name :  Time.129.182.165.44       Kind : TimeTicks  Value :  2
Name :  Time.129.182.59.126       Kind : TimeTicks  Value :  3
End of MIB.
```

Fichier 5 : contenue de la MIB

1.6.2 Snmpget

Cette requête permet de consulter les valeurs de différentes instances des objets de la MIB. Il s'agit d'une opération atomique c'est-à-dire que l'une ou l'autre de toutes les valeurs est retrouvée ou pas du tout. Lorsqu'au moins une des valeurs ne peut être retrouvée, aucune valeur ne sera retournée. Dans ce cas, un message d'erreur sera renvoyé. Les erreurs suivantes peuvent apparaître :

– un objet nommé dans le champ variable bindings (voir figure 3 chapitre 2 première partie) ne correspond pas à un identifiant d'objet de la MIB

– un objet nommé n'est pas d'un type agrégé et par conséquent n'a donc aucune valeur. Le message retourné est noSuchName et une valeur correspondant à l'index de l'objet qui pose problème est aussi retournée dans le champ error–index.

Il n'est pas possible de retrouver, par exemple, une ligne entière d'une table en référençant juste l'entrée de l'objet. Pour le faire, on peut ajouter à la requête Snmpget plusieurs instances à retourner. Dans ce cas, lorsqu'aucune réponse n'est possible pour au moins un des objets, alors aucune valeur ne sera renvoyée.

Exemple : Suite au dump par Snmpmany précédent, pour connaître l'état de la machine Wire, on lance la requête suivante :

>% snmpget State.129.182.59.155

et on obtient la réponse suivante :

```
Name: State.129.182.59.155

Kind: StateSyntax

Value: 1
```

1.6.3 snmpgetnext.

Elle est presque identique à la requête Snmpget. La seule différence est qu'on retourne la valeur de l'instance d'un objet qui est le suivant d'après l'ordre lexicographique. Elle est aussi atomique. Ceci permet à la station d'administration de découvrir la structure d'une vue de la MIB d'une manière dynamique. La requête snmpmany lance en fait une série des requêtes snmpget afin de consulter toutes les instances des objets de la MIB.

1.6.4 snmpset.

Cette requête permet de modifier une valeur de l'instance, d'en créer une nouvelle ou de supprimer une qui existe déjà. Le format de la requête contient l'identifiant de l'instance de l'objet et la valeur à affecter à l'instance de l'objet. Il s'agit encore une fois d'une opération atomique. En cas de valeur inconsistante (en fonction du type), une erreur et renvoyée (noSuchName).

1.6.4.1 modification d'une valeur.

Se fait grâce à la requête snmpset en précisant l'attribut qu'on veut modifier, le type de sa valeur et la valeur à remplacer.

Exemple :

snmpset address.129.182.59.155 IpAddress 129.182.59.155

affecte l'adresse **129.182.59.155** à l'instance **address.129.182.59.155** dans la MIB.

1.6.1.2 création d'une instance.

Il suffit de lancer la requête snmpset en précisant un attribut non encore instancié, le type de sa valeur et la valeur qu'on veut lui assigner. Si l'on suppose maintenant que la station d'administration voudrait ajouter une nouvelle entrée dans la table (une nouvelle ressource dans le réseau) pour l'adresse suivante 129.182.52.45, l'index suivant **address.129.182.52.45** sera ajouté dans la table. On lance la requête suivante :

>% snmpset address.129.182.52.45 IpAddress 129.182.52.45

Address est l'index de la table. Le noyau de l'agent ne connaît pas l'identifiant d'instance 129.182.52.45.
A la réception de cette requête, le noyau va créer une nouvelle entrée dans la table.
La méthode qui permet de manipuler les instances des objets de la MIB va ajouter le nom en toutes lettres de la machine correspondant à l'adresse 129.182.52.45 dans le fichier qui contient les noms de toutes les ressources du réseau, puis lancera le rafraîchissement des valeurs des instances des objets de la MIB. C'est à ce moment que les instances correspondant à la ressource qu'on vient d'ajouter dans le réseau seront instanciés dans la MIB.

1.6.4.3 suppression d'une instance.

La requête snmpset peut aussi être utilisée pour supprimer une entrée dans une table. Dans ce cas, la valeur NULL est fournie à l'index à supprimer.
A la réception de la requête, la méthode qui permet de manipuler les instances des objets de la MIB va supprimer le nom de la ressource qu'on veut ôter du réseau dans le fichier qui contient les noms de toutes les machines du réseau, puis relance la mise à jour des valeurs des objets de la MIB. C'est à ce moment que les instances correspondant à la ressource qu'on a supprimé du réseau vont disparaître dans la MIB.

Exemple : la requête suivante supprime la machine d'adresse 129.182.52.45 du réseau.

>% snmpset address.129.182.52.45 NULL

CHAPITRE 2 : SPECIFICATION ET DEVELOPPEMENT DE LA CHAINE AUTOMATIQUE DE TEST

2.1 Introduction.

L'agent PING que nous venons de construire respecte le protocole snmp. Il permet de répondre aux sollicitations de la station d'administration et le cas échéant, effectue sur l'élément les opérations demandées. Son noyau permet de manipuler les instances des tables de la MIB et assure les échanges entre la station d'administration et la ressource qu'il gère. Lors du développement de la chaîne automatique de tests, nous allons essayer de le rendre le plus général possible en tenant compte de toutes les caractéristiques fonctionnelles possibles des agents SNMP. Ce chapitre présente un plan de tests permettant la validation des agents construits suivant le protocole snmp.

On traitera essentiellement de tests fonctionnels. Les tests d'endurance, de robustesses et de performances sont bien entendu indispensables à la complétude de la validation.

Le test d'endurance concerne la nécessité de s'assurer qu'un agent fonctionne bien durant une longue période sur une machine de production. Le test de robustesse permet de savoir si l'agent résiste à une utilisation maximale de ses capacités, compte tenu de la richesse en nombre d'objets de sa MIB, en nombre d'instances de ces objets et compte tenu de la multiplication des requêtes en provenance d'un manager ou d'une méthode. Les tests de performance ont pour but de tuner l'agent.

Nous allons parcourir successivement les étapes suivantes :
1) la spécification, dans laquelle nous relevons et explicitons les différents tests à réaliser;
2) le développement de l'automate, dans lequel nous expliquons l'implémentation de la chaîne;
3) la validation, dans laquelle nous simulons le test de l'agent que nous avons développé . Ce qui nous permettra de valider cet agent et la chaîne que nous venons de développer.

2.2 Spécification.

L'automate développé ne réalise que le test des agents snmp. Compte tenu des fonctions dédiées à un agent snmp, les tests que nous allons spécifier ici porteront sur le contenu de la MIB et sur le bon fonctionnement des requêtes snmp.

Dans le premier test, on va vérifier la validité des valeurs des instances des objets de la MIB. Une valeur sera valide lorsqu'elle sera contenu dans un intervalle de validité lié à un objet et dont les bornes sont fixées par le développeur de l'agent. Dans le deuxième test, on s'assurera que toutes les requêtes snmp fonctionnent correctement.

Par rapport aux caractéristiques que nous avons ci-dessus relevées, nos tests porteront essentiellement sur les types d'opérations suivantes :

1) test de la validité des contenus de la MIB :

- tests de la validité de toutes les instances de la MIB de l'agent
- test de la validité de certaines instances de la MIB

2) tests des requêtes snmp :

- interrogation d'une instance d'un objet de la MIB
- modification d'une instance d'un objet de la MIB
- création d'une instance d'un objet de la MIB
- suppression d'une instance d'un objet de la MIB

Pour spécifier chaque test, nous allons décrire l'identification du test, son scénario et le résultat attendu.

2.2.1 test sur la validité des contenus de la MIB

Ces tests portent sur la signification sémantique des instances des objets de la MIB. Ces tests vont permettre au développeur de l'agent de repérer certaines instances qui paraissent incohérentes par rapport aux limites de validité fixées par ce même développeur.

2.2.1.1 tests de la validité de toutes les instances de la MIB de l'agent.

a) Identification.
Il s'agit d'un test sur toutes les instances des objets de la MIB.
Ce test devra se faire en fonction des bornes fixées par le développeur de l'agent.
Il faut interroger toute la MIB gérée par un agent.

b) Scénario.
Il faut interroger toutes les instances de la MIB en lançant à répétition la requête Snmpget.
Puis on compare chaque valeur renvoyée par la requête snmpget aux valeurs des bornes liées au type d'objet dont la valeur est instanciéé.

c) Résultat attendu.
Chaque valeur retournée par le snmpget doit être contenue dans l'intervalle défini par les bornes liées au type d'objet dont elle est l'instance. Dans le cas contraire, toutes les instances consultées dont le test aura échoué seront qualifiées de non cohérentes et elles seront signalées au développeur pour des éventuelles corrections.

2.2.1.2 test de la validité d'une instance de la MIB

a) Identification.
Il s'agit d'un test sur une instance particulière d'un objet de la MIB.
Ce test devra se faire en fonction des bornes fixées par le développeur de l'agent.
Au contraire du test précèdent, ce test ne portera que sur une instance d'un objet précis de la MIB.

b) Scénario.
Il faut interroger une instance de la MIB en lançant la requête Snmpget.
Puis on compare la valeur renvoyée par le snmpget aux valeurs des bornes liées au type d'objet dont la valeur est instanciéé

c) Résultat.
La valeur retournée par le snmpget doit être contenue dans l'intervalle défini par les bornes liées au type d'objet dont elle est l'instance. Dans le cas contraire, cette instance dont le test a échoué sera qualifiée de non cohérente et elle sera signalée au développeur pour une éventuelle correction.

2.2.2 tests sur les requêtes snmp

Ces tests concernent les échanges des requêtes entre la station d'administration et l'agent.

2.2.2.1 interrogation d'une instance d'un objet de la MIB

a) Identification.

Il s'agit d'un test sur la requête snmpget.

Ce test se fera en deux étapes :

- on vérifie d'abord que la requête fonctionne correctement
- on vérifie ensuite que la valeur de l'instance consultée est cohérente

b) Scénario.

On interroge une instance de la MIB à l'aide des requêtes snmpget.

Puis, on vérifie la validité de la valeur retournée par rapport aux bornes liées au type d'objet de l'objet qu'on instancie.

c) Résultat.

La valeur retournée par la requête snmpget doit être contenue dans l'intervalle défini par les deux bornes du type d'objet. Dans le cas contraire, ce type d'objet sera considéré comme incohérent et il sera signalé au développeur pour correction.

2.2.2.2 modification d'une instance d'un objet de la MIB

a) Identification.

Il s'agit d'un test sur la requête snmpset utilisée en mode modification.

Lorsque cette requête est utilisée en vue de modifier la valeur d'une instance, elle doit contenir la valeur qu'il faut substituer à la valeur qu'on veut changée.

Ce test se fera en trois étapes différentes :

- modification de la valeur de l'instance
- consultation de la même instance après l'étape de modification
- comparaison des valeurs

b) Scénario.

- on commence par lancer la requête snmpset de modification en précisant la nouvelle valeur à remplacer.
- on interroge ensuite la même instance pour obtenir la valeur qui est effectivement dans la MIB
- on compare enfin la valeur retournée par la requête snmpget avec la valeur précisée dans la requête snmpset de modification.

c) Résultat.

La valeur retournée par la requête snmpget et la valeur précisée dans la requête snmpset de modification doivent être les même pour pouvoir conclure que la requête de modification snmpset fonctionne correctement. Lorsque les deux valeurs ne sont pas identiques, on conclura que la requête ne fonctionne pas comme il faut.

2.2.2.3 création d'une instance d'un objet de la MIB

a) Identification.

Il s'agit d'un test sur la requête snmpset utilisée en mode création.

Lorsque cette requête est utilisée en vue de la création de la valeur d'une instance, elle doit contenir la valeur d'une instance qui n'existe pas encore dans la MIB.

Ce test se fera en trois étapes différentes :

- création de la valeur de l'instance
- consultation de la même instance après l'étape de modification
- comparaison des valeurs

b) Scénario.

- on commence par lancer la requête snmpset de création en précisant la valeur d'une instance qui n'existe pas encore dans la MIB.
- on interroge ensuite la MIB pour obtenir la valeur de l'instance qu'on vient de créer
- on compare enfin la valeur retournée par la requête snmpget avec la valeur précisée dans la requête snmpset de création.

c) Résultat.

La valeur retournée par la requête snmpget et la valeur précisée dans la requête snmpset de création doivent être les même pour pouvoir conclure que la requête de création snmpset fonctionne correctement. Lorsque les deux valeurs ne sont pas identiques, on conclura que la requête ne fonctionne pas comme il faut.

2.2.2.4 suppression d'une instance d'un objet de la MIB

a) Identification.

Il s'agit d'un test sur la requête snmpset utilisée en mode suppression.

Lorsque cette requête est utilisée en vue de la suppression de la valeur d'une instance, elle doit contenir la valeur NULL comme valeur de remplacement de la valeur qui est déjà dans la MIB.

Ce test se fera en trois étapes différentes :

- suppression de la valeur de l'instance
- consultation de la même instance après l'étape de modification
- comparaison des valeurs

b) Scénario.

- on commence par lancer la requête snmpset de suppression en précisant la valeur NULL comme valeur de remplacement.
- on interroge ensuite la MIB pour obtenir la valeur de l'instance qu'on vient de supprimer
- on compare enfin la valeur retournée par la requête snmpget avec la valeur précisée dans la requête snmpset de suppression.

c) Résultat.

La requête snmpget doit retourner une erreur pour signaler que l'instance qu'on essaye de consulter n'existe pas dans la MIB. Lorsque la requête renvoie une valeur, ce qu'il y a un problème et donc la requête ne fonctionne pas correctement.

2.3 Développement de la chaîne automatique.

2.3.1 Principe.

Dans la définition des objets de la MIB, seuls les types de base suivants peuvent être utilisés :

- INTEGER
- Octet String
- IpAddress
- Counter
- Gauge
- TimeTicks

Pour plus de pertinence, nous limiterons nos tests aux objets d'un type mathématiquement testable, c'est-à-dire les objets ayant l'un des types suivants : INTEGER, Counter, Gauge ou TimeTicks.

L'idée est de demander à l'utilisateur (développeurs d'agents), qui connaît mieux la signification sémantique des instances des objets de l'agent, de fixer une valeur minimale et une valeur maximale pour tout attribut de la MIB d'un des types ci-dessus et entre lesquelles, les différentes valeurs des instances des objets seront considérées comme valides et correctes.

Pour cela, nous allons réaliser les étapes suivantes :

1) parcourir tout le fichier qui contient la définition de la MIB de l'agent afin de rechercher tous les types simples et composés mathématiquement testables.
2) les types ainsi trouvés sont écrits dans une liste chaînée quelconque.
3) Ensuite, on parcourt la MIB et on demande à l'utilisateur d'initialiser tout objet d'un des types ainsi trouvés dans la MIB avec une borne inférieure et une borne supérieure, et on recopie le tout dans un fichier <agent>.BORNE. Chaque fois qu'on voudra tester un agent ou certaines instances de la MIB de cet agent, on commencera par vérifier si tous les objets de la MIB de l'agent sont déjà bornés ou pas. A cet effet, on recherchera dans le répertoire courant s'il existe déjà un fichier du nom <NomAgent>.BORNE, car ce fichier n'est créé que lors de l'affectation des bornes à un agent. Cela permet d'éviter la fastidieuse tâche d'affecter à chaque reprise des bornes aux objets d'un agent, d'autant que le nombre des objets de la MIB peut varier grandement. L'automate développé offre aussi la possibilité de modifier les bornes des objets qui ont déjà été bornés.

4) Tout test sur une instance de la MIB portera au moins sur la présence ou non de la valeur de cette instance entre les bornes relatives à l'objet dont on instancie la valeur. A la fin de la simulation des tests, tous les objets qui paraissent incohérents seront signalés à l'utilisateur.

2.3.2 Description de l'automate de test.

L'automate qu'on a développé lance une série des processus qui dialoguent les uns avec les autres. Il permet le dialogue via le protocole SNMP.
Les principales fonctions de l'automate sont les suivantes :

a) Affectation des bornes.
Pour un agent donné, cette méthode lance l'agent et demande à l'utilisateur de donner les bornes de tous les objets (si ce n'est pas encore fait) ou plutôt demande à l'utilisateur de valider ou de modifier les bornes de certains objets.

b) Test de l'agent.

Cette fonction permet à l'utilisateur de tester toutes les valeurs des instances des objets de la MIB d'un agent donné. Dans ce cas, toutes les instances des objets de la MIB sont comparées aux bornes et les incohérences relevées sont signalées à l'utilisateur.

c) Test des requêtes snmp.

Cette fonction permet de dire à l'utilisateur comment fonctionnent les transactions entre la station d'administration et un agent. Toutes les transactions qui posent des problèmes sont signalées à l'utilisateur.

L'automate reçoit le nom de l'agent et commence par vérifier si l'agent a déjà été borné. Lorsque l'agent est déjà borné, il doit exister dans le répertoire courant un fichier du nom <NomAgent>.BORNE. Sinon, pour chaque valeur de type mathématiquement testable d'une instance d'un objet de la MIB, on demande à l'utilisateur de fournir une borne inférieur et une borne supérieure. On recopie ces valeurs dans le fichier <NomAgent>.BORNE.

Dès que l'automate est sûr que l'agent est borné, on offre la possibilité à l'utilisateur :

1) de tester toutes les instances des objets de la MIB. Dans ce cas, toutes les instances sont comparées aux bornes des objets qu'elles instancient. Les cas incohérents sont signalés à l'utilisateur.

2) de tester le fonctionnement des échanges snmp. Il s'agit des échanges mis en oeuvre suite à des requêtes Snmp en provenance du manager. On tentera de consulter, de modifier, de créer ou de supprimer les valeurs de certaines instances de la MIB d'une part et d'autre part, on vérifiera si l'opération a réussi en accédant à la MIB par une consultation. Les valeurs ainsi consulter, modifier, créer ou supprimer seront comparées aux bornes affectées aux objets dont elles sont les instances. Ceci, afin de s'assurer de leurs validités.

2.3.3 Affectation des bornes aux types d'objets de l'agent.

Cette fonction de l'automate sert à borner les objets d'un agent, c'est-à-dire à fixer une borne inférieure et une borne supérieure de chaque objet de la MIB. Etant donné la diversité des agents SNMP et la variété sémantique des objets qui les composent, et ainsi que la nécessité pour l'utilisateur d'adapter l'agent à ses besoins, on a jugé qu'il était plus pertinent de laisser à l'utilisateur la responsabilité de fixer les bornes des objets des agents d'après sa maîtrise de la fonction de l'agent. En plus, pour des raisons de pertinence évidentes, seuls les objets de type mathématiquement testable seront bornés. En effet, on ne voit pas l'importance d'effectuer un test sur les instances d'un objet de type « chaîne de caractère », par exemple.

L'affectation des bornes aux objets de la MIB du noyau d'un agent s'effectue en trois étapes suivantes (voir figure 2.1 ci-dessous) :

– la réalisation d'un dump snmp
– la détermination d'une liste des types mathématiquement testables
– l'affectation des bornes aux objets de la MIB

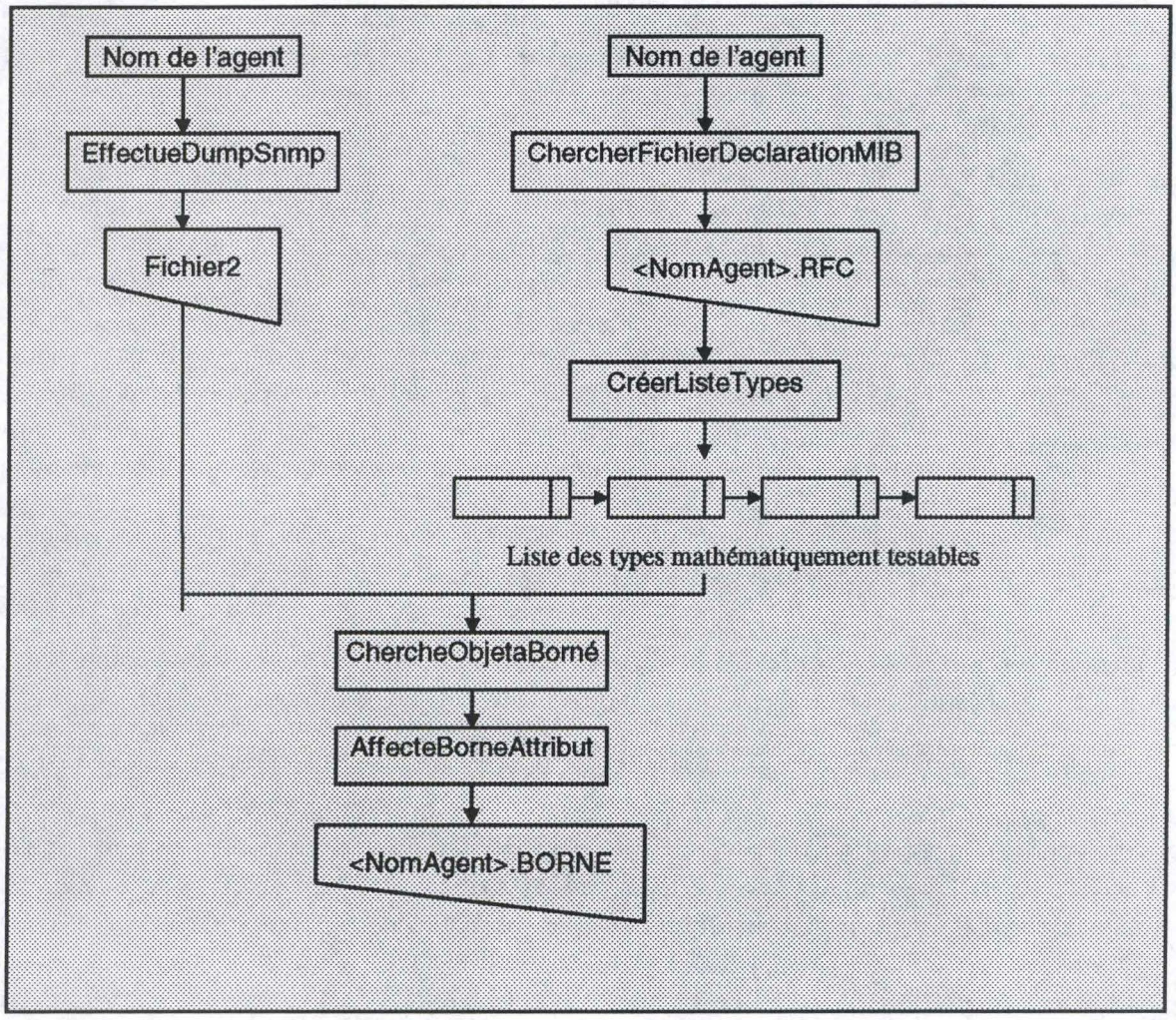


Figure 2.1: fonctionnement de l'affectation des bornes aux objets de l'agent

2.3.3.1 la réalisation d'un dump snmp.

Cette étape permet d'obtenir toutes les instances des objets de la MIB. En connaissant le nom de l'agent, on lance la requête Snmpmany –h <NomAgent>. La requête Snmpmany réalise la consultation de toutes les instances des objets de la MIB suivant l'ordre lexicographique. Elle lance donc à répétition la requête Snmpgetnext. Le résultat est rédirigé dans le fichier "fichier2".

2.3.3.2 la détermination d'une liste des types mathématiquement testables.

Connaissant le nom de l'agent, cette étape recherche dans le répertoire courant le fichier de déclaration de la MIB de l'agent (soit <NomAgent>.RFC), le parcoure en recherchant tous les types simples ou composés mathématiquement testables et les place dans une liste chaînée.

2.3.3.3 l'affectation des bornes aux objets de la MIB.

On recherche dans le fichier de déclaration de la MIB tous les objets ayant un type contenus dans la liste chaînée ci-dessus créée et on demande à l'utilisateur de lui affecter une borne inférieure et une borne supérieure. Cet objet, son type et ses deux bornes sont enfin recopiés dans le fichier <NomAgent>.BORNE.

2.3.4 Tests.

L'automate que nous avons développé offre la possibilité de faire trois types des tests différents :

1) tester toutes instances des objets de la MIB de l'agent
2) tester le bon fonctionnement des échanges snmp

Chaque type de test ci-dessus est terminé par une conclusion sur la validité du test qu'on vient de réaliser. Ainsi, on aura en plus, les fonctions suivantes :

– conclusion sur la validité d'un agent
– conclusion sur la validité des échanges snmp

2.3.4.1 Tester toutes les instances des objets de la MIB d'un agent.

Ce test permet de parcourir toutes les instances des objets de la MIB, de les comparer aux bornes associées à ces objets et de relever les instances qui paraissent incohérentes.

Tel qu'illustrer dans la figure 2.2 ci-dessous, pour instancier tous les objets de la MIB, on lance la requête Snmpmany –h <NomAgent> et on redirige le résultat dans le fichier "fichier2". Le fichier <NomAgent>.BORNE contient tous les objets de type mathématiquement testable, leurs types et leurs bornes inférieures et supérieures. Le principe du test consiste à comparer la valeur de chaque instance contenue dans fichier2 aux valeurs des bornes de l'objet instancié. Lorsque la valeur de l'instance est comprise entre les deux bornes, on stocke la valeur "0" dans la ième position d'un tableau qui contient les résultats de tous les tests sur les différentes instances testées. La ième position correspond au ième test. lorsque le test est négatif, on stocke la valeur "1" au lieu de zéro. Conjointement, on stocke aussi dans une liste chaînée les noms de tous les attributs dont les tests sur les instances n'ont pas été positifs. Ceci, afin de proposer cette liste à l'utilisateur pour des éventuelles corrections.

La procédure de conclusion sur la validité compte la fréquence des résultats négatifs et par rapport à une limite préalablement fixée par l'utilisateur, la construction de l'agent qu'on vient de tester sera appréciée. Lorsque la fréquence des tests négatifs est importante, il faudra peut-être revoir toute la construction de l'agent, peut-être même, adopter une autre démarche de construction de la MIB et de construction de la méthode qui l'anime. Lorsque le nombre des tests négatifs est peu important, seule la révision des objets incohérents s'avère pertinente. Dans ce cas, ces objets seront proposés à l'utilisateur pour lui permettre de revoir la manière dont ils sont manipulés.

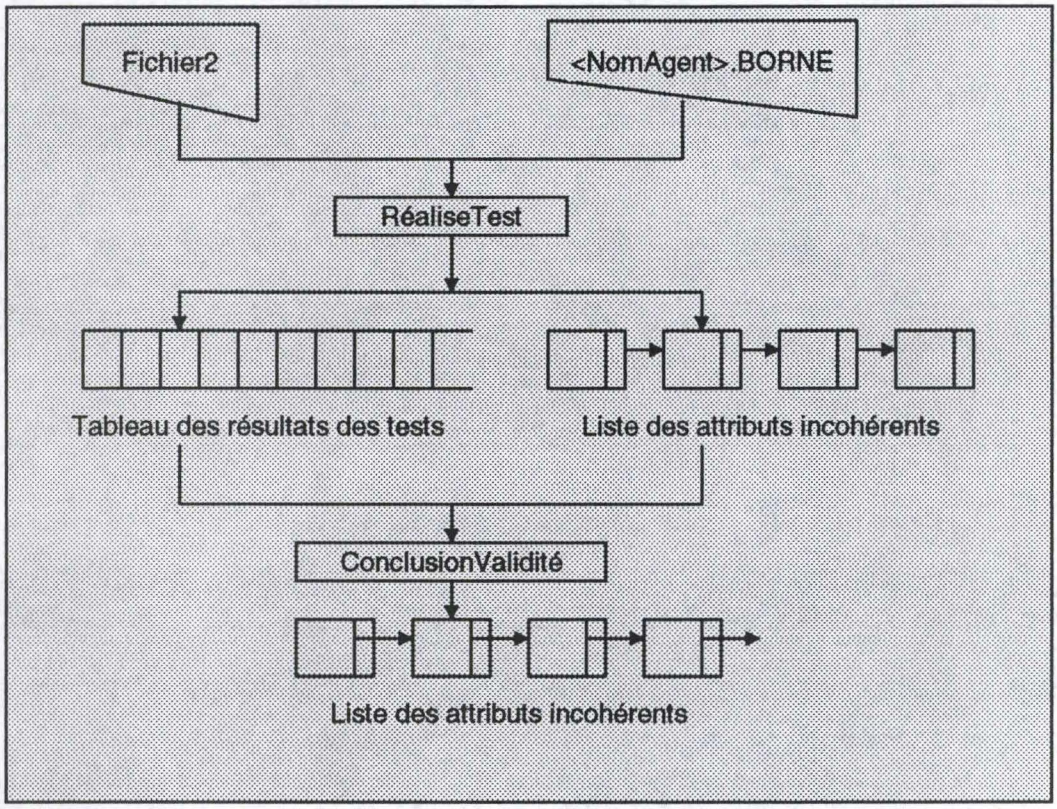


Figure 2.2 : fonctionnement du test d'un agent

2.3.4.2 Tester le bon fonctionnement des échanges snmp.

Les échanges snmp sont mis en oeuvre par le lancement des requêtes SNMP en provenance du manager.

L'agent développé permet l'exécution des primitives suivantes :

- GET : permet la consultation d'une instance d'un attribut
- GET–NEXT : permet la consultation de l'instance suivante d'un attribut
- SET : réalise la modification, la création et la suppression de l'instance d'un attribut

Les requêtes SNMP en provenance du manager sont transmises à la méthode par l'intermédiaire du noyau de l'agent. Toute opération provoque la recherche de la valeur à retourner directement dans la ressource et la mise à jour des valeurs des instances de la MIB avant le renvoi de la réponse au manager.

Dans l'illustration de la figure 2.3 ci-dessous, l'automate reçoit la requête de l'utilisateur. L'automate lance la requête de l'utilisateur. Lors des opérations de consultation (Snmpget) et de modification (Snmpset), la méthode met à jour les valeurs de différentes instances de la MIB en allant rechercher directement leurs valeurs dans la ressource. Puis seulement, en renvoi la réponse à l'utilisateur. De la sorte, les valeurs des instances de la MIB sont toujours rafraîchies. Ce qui permet de suivre l'état du réseau couvert.

Lors des opérations de création et de suppression (Snmpset) (voi figure 2.4 ci-dessous), nous utilisons le fichier "MachinesRéseau" dans lequel on a stocké la liste de toutes les machines présentes dans le réseau. L'exécution de la requête de création aura pour effet d'ajouter le nom d'une nouvelle machine dont l'adresse est donnée dans le fichier ce fichier. Pour y parvenir, nous utilisons la fonction UNIX getHostbyName qui permet de trouver le nom de la machine à partir de son adresse. Une fois que le nom est ajouté dans ce fichier, nous relançons la mise à jour de la MIB. C'est en ce moment que les instances des objets relatives à la nouvelle machine seront ajoutées dans la MIB.

De même, la requête de suppression aura pour effet d'effacer le nom d'une machine du fichier ci-dessus et dont l'adresse est donnée dans la requête de suppression. Pour y parvenir, nous utilisons la fonction UNIX getHostbyName qui permet de trouver le nom de la machine à partir de son adresse. Une fois que le nom est supprimé dans ce fichier, nous relançons la mise à jour de la MIB. C'est en ce moment que les instances des objets relatives à la machine supprimée seront supprimées dans la MIB.

Après la création ou la suppression, le test portera sur la certitude que les instances ont été réellement ajoutées ou supprimées de la MIB. Pour cela, on lance la requête de consultation snmpget sur des instances qu'on a voulu créer ou supprimer et on compare le résultat aux valeurs passées comme paramètres aux requêtes de création et/ou suppression. Dans le cas de l'agent PING, un test supplémentaire portera sur l'assurance que le nom de la machine a été supprimé ou ajouté dans le fichier "MachinesRéseau".

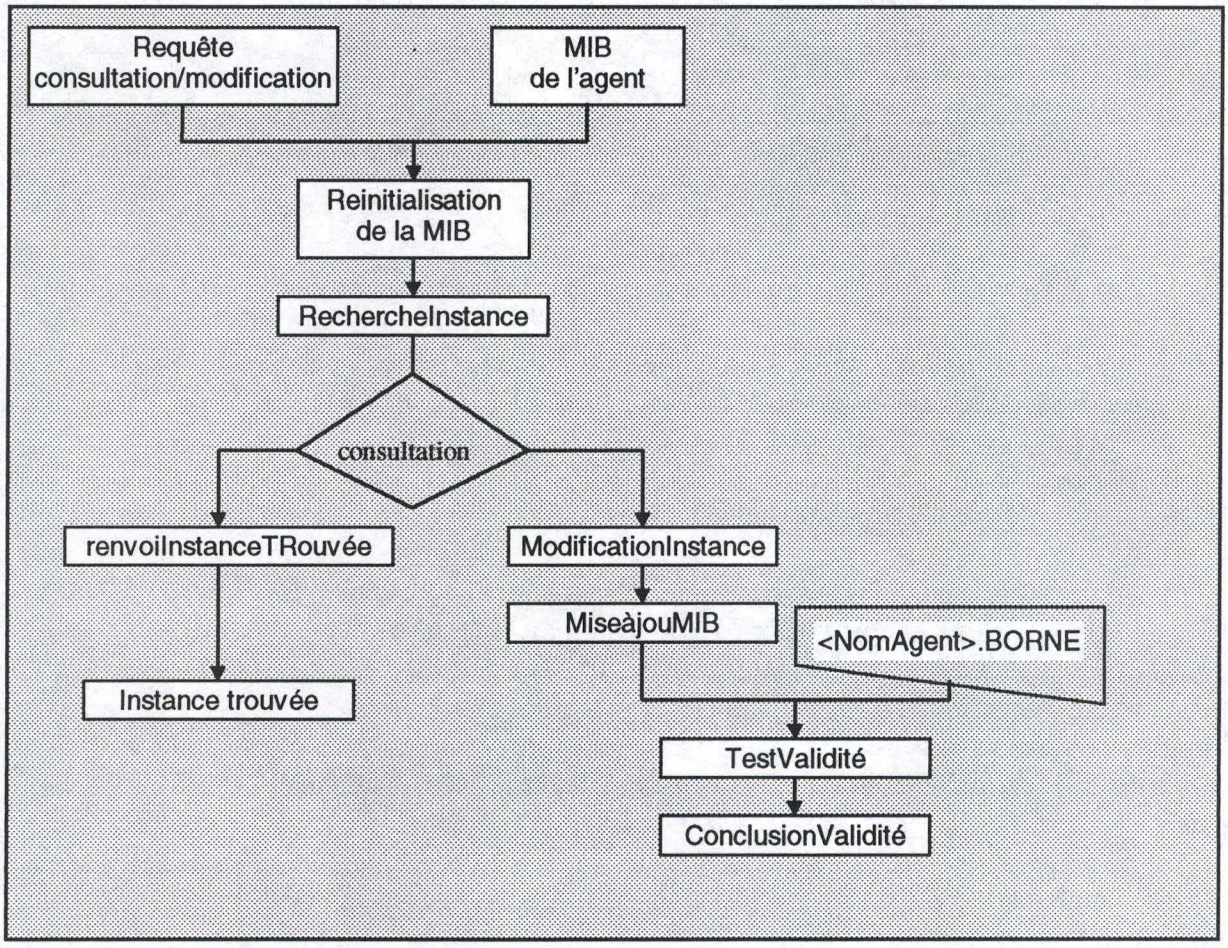


Figure 2.3 : Fonctionnement des tests de consultation /modification

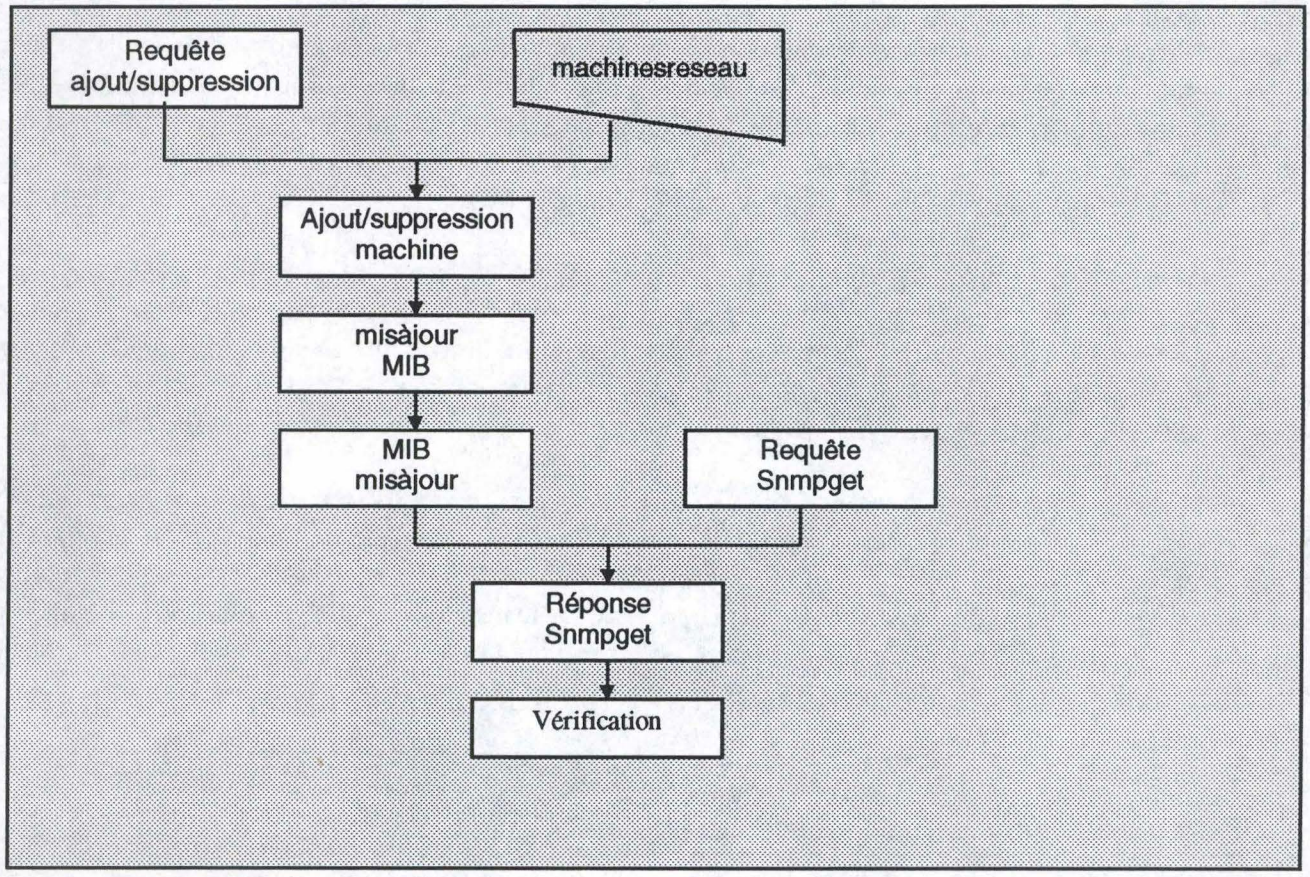


Figure 2.4 : Fonctionnement des tests de création/suppression

2.4 Simulation des tests sur l'agent PING.

Une fois l'automate développé, nous allons simuler son fonctionnement sur l'agent que nous créé afin de bien comprendre son fonctionnement et de réaliser un essaye de validation.

2.4.1 Affectation des bornes aux objets de l'agent PING.

L'automate reçoit le nom de l'agent et commence par vérifier si l'agent PING a déjà été borné. Lorsque l'agent est déjà borné, il doit exister dans le répertoire courant un fichier Ping.BORNE. Sinon, pour chaque type d'objet mathématiquement testable d'un objet de la MIB, on demande à l'utilisateur de fournir une borne inférieur et une borne supérieure.

Dans la déclaration de la MIB de l'agent PING, les types d'objets mathématiquement testables sont les suivants :

- state de type StateSyntax
- ptransmis de type INTEGER
- perdu de type INTEGER
- Time de type TimetIcks

Pour chacun de ces types d'objets, on demande à l'utilisateur de fournir une borne inférieure et une borne supérieure et on recopie ces valeurs dans le fichier Ping.BORNE. Pour l'agent PING, ce fichier se présente comme suit :

```
State      StateSyntax  1   2
Ptransmis  INTEGER      3   3
Pperdu     INTEGER      0   0
Time       TimeTicks    1   30
```

Fichier 1 : contenu du fichier ping.BORNE

2.4.2 Tests.

Dans la phase des tests, le développeur de l'agent PING a la possibilité de tester la validité de toutes les instances de l'objet de sa MIB, de n'en tester que quelques-unes et de tester le fonctionnement des requêtes snmp pour cet agent.

2.4.2.1 Tester toutes les instances de l'objet de la MIB.

Pour rappel, la MIB de l'agent PING ne contient qu'un objet : pingmibTable. Cet objet a les types d'objet suivants : address, state, ptransmis, perdu et time. Les instances de l'objet pingmibTable correspondent aux différentes valeurs que prennent les types d'objet ci-dessus.

Dans cette partie des tests, il s'agit de vérifier la cohérence des valeurs de ces instances par rapport aux bornes qui ont été affectées aux différents types d'objet de l'objet pingmibTable.

Pour le faire, on va parcourir les étapes suivantes :

1) En connaissant le nom de l'agent PING, on instancie toutes les valeurs de la MIB. Pour cela, on effectue un dump snmp. Cette action nous permet d'obtenir toutes les instances dans le fichier FICHIER2 suivant :

```
Name :   address.129.182.59.94    Kind : IpAddress    Value : 129.182.59.94
Name :  address.129.182.59.155    Kind : IpAddress    Value : 129.182.59.155
Name :  address.129.182.165.6     Kind : IpAddress    Value : 129.182.165.6
Name :  address.129.182.165.27    Kind : IpAddress    Value : 129.182.165.27
Name :  address.129.182.165.44    Kind : IpAddress    Value : 129.182.165.44
Name :  address.129.182.59.126    Kind : IpAddress    Value : 129.182.59.126

Name :  state.129.182.59.94       Kind : StateSyntax  Value :  1
Name :  state.129.182.59.155      Kind : StateSyntax  Value :  1
Name :  state.129.182.165.6       Kind : StateSyntax  Value :  1
Name :  state.129.182.165.27      Kind : StateSyntax  Value :  1
Name :  state.129.182.165.44      Kind : StateSyntax  Value :  1
Name :  state.129.182.59.126      Kind : StateSyntax  Value :  1

Name :  Ptransmis.129.182.59.94      Kind : Integer   Value :  3
Name :  Ptransmis.129.182.59.155     Kind : Integer   Value :  3
Name :  Ptransmis.129.182.165.6      Kind : Iinteger  Value :  3
Name :  Ptransmis.129.182.165.27     Kind : Integer   Value :  3
Name :  Ptransmis.129.182.165.44     Kind : Integer   Value :  3
Name :  Ptransmis.129.182.59.126     Kind : Integer   Value :  3

Name :  Pperdu.129.182.59.94      Kind : Integer   Value :  0
Name :  Pperdu.129.182.59.155     Kind : Integer   Value :  0
Name :  Pperdu.129.182.165.6      Kind : Iinteger  Value :  0
Name :  Pperdu.129.182.165.27     Kind : Integer   Value :  0
Name :  Pperdu.129.182.165.44     Kind : Integer   Value :  0
Name :  Pperdu.129.182.59.126     Kind : Integer   Value :  0

Name :  Time.129.182.59.94        Kind : TimeTicks    Value :  2
Name :  Time.129.182.59.155       Kind : TimeTicks    Value :  3
Name :  Time.129.182.165.6        Kind : TimeTicks    Value :  8
Name :  Time.129.182.165.27       Kind : TimeTicks    Value :  2
Name :  Time.129.182.165.44       Kind : TimeTicks    Value :  2
Name :  Time.129.182.59.126       Kind : TimeTicks    Value :  3
End of MIB.
```

Fichier 2 : les instances de tous les objets de la MIB

2) Ensuite, on va comparer la valeur de chaque instances de la MIB aux bornes liées au type d'objet dont elle est la valeur.

Par exemple, la valeur de l'instance Time.129.182.165.27 (=2) est comparée aux bornes du type d'objet Time (1 et 30 voir fichier 1, voir dans ci-dessus). Pour que cette instance soit considérée comme cohérente, sa valeur doit être contenue entre les valeurs 1 et 30. La valeur "0" sera positionnée dans la ième position du tableau qui contient les résultats de tous les tests réalisés. Dans le cas contraire, le test aura échoué et la valeur "1" sera positionnée dans la ième position du tableau qui contient les résultats de tous les tests réalisés. Ce tableau permettra de retrouver les instances incohérentes afin de le signaler au développeur de l'agent PING pour une éventuelle correction. Ce tableau a la forme suivant :

Instances	address.129.182.59.94	address.129.182.59.155	address.129.182.165.6	etc
résultat test	1	1	0	

Par exemple, en fournissant à l'automate le nom de l'agent PING que nous avons développé précédemment et par rapport aux bornes qu'on a affectées aux objets de notre MIB (voir fichier 1 ping.BORNE ci-dessus), l'automate peut retourner la valeur : Time.129.182.59.126. Ce qui signifie que cette instance n'est pas valide par rapport aux bornes concernant le temps.
Ce qui signifie que le temps que met un paquet pour atteindre la ressource (supérieur à 30 millisecondes) dont l'adresse est 129.182.59.126 est trop long. Cela peut être normal si cette ressource est géographiquement située à un lieu éloigné de la station d'administration.

2.4.2.2 tester une instance de la MIB de la l'agent PING.

Plutôt que de tester toutes les instances de la MIB, cette partie ne réalise le test que pour une seule instance. La démarche demeure la même que celle du test précèdent, sauf qu'ici, au lieu d'instancier toutes les instances de la MIB, on ne réalise que la consultation de l'instance dont on veut tester la cohérence. Le reste de la procédure est la même.

Par exemple, lorsqu'on veut tester la cohérence de l'instaance time.129.182.165.44, on commence par consulter la valeur de cette instance en lançant la requête snmpget. Puis on compare la valeur retournée (3) aux bornes (1 et 30) liées au type d'objet time. Si la valeur contenue dans la MIB est située dans l'intervalle que défini les deux bornes, donc cette instance est cohérente. Dans le cas contraire, il y a un problème et cela est signalé au développeur de l'agent.

2.4.2.3 tester le fonctionnement des requêtes snmp avec l'agent PING.

L'agent PING qu'on a construit permet de répondre aux requêtes snmpget, snmpset (en mode modification, création et suppression) en provenance de la station d'administration. Dans cette partie des tests, on va tester le bon fonctionnement de l'agent PING par rapport à ces requêtes.

1) la requête de consultation.

On voudrait s'assurer que la valeur retournée par la requête snmpget lorsqu'on consulte une instance correspond bien à la valeur de cette instance qui est dans la MIB. Il faudrait pouvoir accéder en mode lecture dans la MIB autrement qu'en utilisant la requête de consultation snmpget. Tenir compte de cette exigence lors du développement de l'automate nous aurait obligé de le réaliser pour un environnement particulier ; ce qui aurait eu pour conséquence d'en réduire la généralisation. C'est pourquoi nous avons décider de ne pas développer ce type d'action.

Par exemple, l'utilisateur peut vouloir consulter la valeur de l'instance Time.129.182.165.44. Si l'agent fonctionne avec une méthode réelle, en lançant la requête protocolaire de consultation :

snmpget Time.129.182.165.44

la valeur retournée doit correspondre à la valeur la plus récente sur le temps de transaction sur la ressource dont l'adresse est 129.182.165.44. En fait, la méthode va interroger la ressource physique

avant de mettre à jour la MIB et de retourner la valeur sollicitée. Ensuite seulement cette valeur sera comparée aux bornes qui en fixent la limite de validité.

2) la requête de modification.

Lorsqu'on veut tester la modification de la valeur de l'instance : Time.129.182.59.126, on doit d'abord lancer la requête « snmpset Time.129.182.59.126 TimeTicks 5 ». L'agent va tenter de modifier la valeur de l'instance. Puis on va consulter la valeur de la même instance dans la MIB en lançant la requête de consultation « snmpget Time.129.182.59.126 ».
On compare la valeur qu'on voulait avoir dans la MIB après modification (valeur fournie à la requête snmpset) à la valeur qu'on a obtenue suite à la consultation. La requête fonctionne bien lorsque les deux valeurs sont identiques. Dans le cas contraire, il y a un problème et cela doit être signalé au développeur de l'agent PING. On aura compris qu'une modification n'est pas possible pour le type d'objet time car il est défini dans la déclaration de la MIB comme ayant statut read-only.

3) la requête de création.

D'après la déclaration de la MIB de l'agent PING, seul le type d'objet address peut être créé. Ce qui correspond à l'ajout d'une ressource dans le réseau.
Pour tester le bon fonctionnement de la requête snmpset lorsqu'elle est utilisée dans le mode création, essayons d'ajouter dans le réseau une ressource dont l'adresse Internet est 129.182.54.45. On lance la requête suivante « snmpset address.129.182.54.45 IpAddress 129.182.54.45 ». A la réception de cette requête, l'agent ajoute cette ressource dans le réseau et ajoute les instances liées à cette nouvelle ressource dans la MIB.
Puis, on va consulter la valeur de l'instance address.129.182.54.45 et la valeur retournée doit correspondre à la valeur « 129.182.54.45 ». Dans le cas contraire, la création n'a pas eu lieu et il faut le signaler au développeur de l'agent PING.

4) la requête de suppression.

Autant la création ne peut porter que sur l'ajout d'une ressource dans le réseau, autant la suppression ne peut porter que sur le retrait d'une ressource dans le réseau.
Pour tester le bon fonctionnement de la requête snmpset lorsqu'elle est utilisée dans le mode suppression, essayons d'enlever dans le réseau une ressource dont l'adresse Internet est 129.182.54.45. On lance la requête suivante « snmpset address.129.182.54.45 NULL ». A la réception de cette requête, l'agent supprime cette ressource dans le réseau et supprime les instances liées à cette ressource dans la MIB.
Puis, on va consulter la valeur de l'instance address.129.182.54.45 et la valeur retournée doit correspondre à un message signalant qu'on essaye de consulter les instances liées à une ressource qui n'existe plus dans le réseau. Lorsque la requête renvoie une adresse correspondant à cette ressource, ce qu'il y a un problème et il faut le signaler au développeur de l'agent PING.

2.5 Validation.

La validation des tests qu'on a spécifiés et développés ci-dessus se fait grâce à une interface utilisateur. L'utilisateur de cet automate a le choix sur le test à réaliser et après chaque test, l'automate lui signale les anomalies rencontrées par rapport au comportement normal espéré. De sorte que c'est finalement l'utilisateur qui décide de valider ou pas son agent. L'outil que nous avons développé apparaît comme un système d'aide à la validation des agents snmp.

IV. CONCLUSION

Dans un contexte d'administration de réseau par le protocole SNMP, les différentes ressources du réseau sont administrées par la station d'administration. Celle-ci a besoin d'accéder à certaines informations pertinentes qui lui sont utiles pour réaliser sa tâche d'administration. Pour cela, chaque ressource présente dans le réseau est modélisée par un agent grâce auquel la ressource peut répondre aux requêtes snmp en provenance de la station d'administration et lui signaler la survenance inattendue de certains événements. L'agent sert aussi à manipuler les informations d'administration contenues dans sa base de données des informations d'administration de la ressource qu'il modélise. Il s'instaure un échange protocolaire entre la station d'administration et l'agent. Le développement d'un agent pour animer un tel modèle revêt une importance particulière. C'est pour répondre à cette nécessité que nous avons développé la chaîne automatique permettant de tester et de garantir le bon fonctionnement des agents développés suivant le protocole d'administration snmp et réalisant les fonctionnalités ci-dessus décrites.

Le développement des agents snmp qui modélisent les différentes ressources du réseau utilise de l'expertise en matière de télécommunication, de réseau et de système d'exploitation. Autant, le développement des agents snmp exige une maîtrise de ces différents domaines, autant le développement de l'automate de test des agents développés suivant ce protocole exige la maîtrise de mêmes concepts. C'est pourquoi, après avoir introduit notre travail par une présentation des concepts liés à ces domaines, nous avons développé un agent modélisant une ressource virtuelle quelconque (l'agent PING). L'objectif était de nous permettre de mieux appréhender les différents concepts et structures qui peuvent être mis en oeuvre pour modéliser une ressource et générer un agent qui soit en mesure d'animer sa base d'informations d'administration. La connaissance de ces techniques est considérée comme un prérequi en vue du développement de notre automate de test des agents snmp.

Durant le développement de cet automate de tests, nous devions concevoir et réaliser un outil le plus général possible, c'est-à-dire indépendant d'une technologie particulière de développement des agents snmp. La technologie selon laquelle peuvent être développés les agent snmp peut varier d'une organisation à une autre. C'est pourquoi, dans la démarche adoptée pour la réalisation de notre objectif, nous avons conçu et implémenté notre automate sans tenir compte de l'agent snmp que nous avions développé auparavant en utilisant la technologie mise en oeuvre par GAM–OAT.

Toutefois, le développement d'un agent met en oeuvre une technologie particulière liée à l'environnement particulier dans lequel il est développé (par exemple, BULL crée ses agents snmp à l'aide du toolkit GAM–OAT qui met en oeuvre une technologie particulière). L'agent que nous avons développé a été fait en utilisant la technologie GAM–OAT. Cette technologie permet de créer des agents snmp modélisant une ressource et pouvant animer la base d'informations d'administration liée à cette ressource. L'agent ainsi développé permet de répondre aux différentes requêtes snmp en provenance de la station d'administration et de générer une requête trap lui permettant de signaler à la station d'administration la survenance d'un événement particulier.

Durant le développement de notre automate, nous avions eu à faire face à un certain nombre des problèmes. Le développement d'un agent snmp met en oeuvre certaines fonctions de télécommunication, de réseau et de systèmes d'exploitation lors de l'implémentation des échanges qui s'opèrent entre la ressource modélisée par un agent et la station d'administration. Un des avantages qu'offrent le toolkit GAM–OAT développé par BULL est de décharger le développeur de l'implémentation de ces fonctions. Nous avons utilisé les fonctions de l'API de GAM–OAT réalisant ce traitement (voir annexe 2).

Les agents snmp développés avec GAM–OAT permettent de répondre aux sollicitations du manager qui leur sont adressées par des requêtes snmp SNMPGET, SNMPSET et SNMPSET(en mode CREATE et DELETE). Ces agents permettent aussi de signaler à la station d'administration, à leurs initiatives, les survenances de certains événements dans les ressources qu'ils modélisent, grâce aux trappes (POLLING). L'agent que nous avons développé et présenté ci-dessus ne met pas en oeuvre cette fonctionnalité. C'est pourquoi, dans le développement de notre automate de tests, nous n'avons ni spécifié, ni développé des tests liés à cette fonctionnalité. Dans une amélioration ultérieure de cet automate, la prise en compte des tests liés à cette possibilité d'un agent snmp contribuerait à la complétude de l'automate que nous avons réalisé.

Bien que nous ayons essayé de développer notre chaîne automatique en envisageant tous les comportements possibles des agents SNMP construits avec différents outils de développement, nous aurions souhaité simuler le fonctionnement de l'automate sur un agent non construit avec GAM–OAT. Nous n'avons pas pu réaliser ce type de test pour tenir compte des différentes subtilités techniques et conceptuels intervenant dans le développement des agents en utilisant une autre technologie que celle mise en oeuvre par GAM–OAT.

Comme vous avez pu le remarquer, pour vérifier le bon fonctionnement de l'agent développé, nous ne recourons qu'à ses autres possibilités (par exemple, consultation par une requête snmpget) pour tester ce qu'on vient de faire. Il aurait été plus intéressant de consulter les instances des objets de la MIB en utilisant un autre moyen, par exemple, le système *OpenMaster* de BULL. Ce système permet, entre autres fonctions, de consulter, de modifier, de créer ou de supprimer les instances des objets de la MIB. Il serait donc intéressant de tester l'automate dans un tel environnement.

Le travail que nous avons réalisé nous a permis de répondre à l'objectif qui nous a été fixé. En particulier, cette chaîne permet de répondre à un besoin exprimé par les développeurs des agents snmp de BULL. Sa réalisation nous a permis de mettre en pratique les différents concepts appris durant notre formation et de vivre une expérience de développement très enrichissante.

V.ANNEXE

Annexe 1.1 : syntaxe des objets de la MIB.

```
IMPORT ObjectName, ObjectSyntax FROM RFC-1155-SMI

OBJECT-TYPE MACRO ::=
BEGIN
        TYPE NOTATION ::=  "SYNTAX "   type(TYPE ObjectSyntax)
                           "ACCESS"  Access
                           "STATUS"   Statut
                           DescrPart
                           ReferPart
                           IndexPart
                           DefValPart
         VALUE NOTATION ::=value (VALUE ObjectName)
        Access::= "read-only" | "read-write" | "write-only" | "not-accessible"
        Statut ::= "mandatory" | "optional" | "obsolete" | "deprecated"
        DescrPart ::= "DESCRIPTION" value (description DisplayString) | empty
        ReferPart ::= "REFERENCE" value (reference DisplayString) | empty
        IndexPart ::= "INDEX" "{" IndexTypes "} "
        IndexTypes ::= IndexType | IndexType " , " IndexType
        IndexType ::= value(indexobject ObjectName)     --if indexobject, use the SYNTAX
                                                        --value of the correspondent
                                                        --OBJECT-TYPE invocation
                          | type (indextype)
                                                        --otherwise use names SMI type
                                                        --must conform to IndexSyntax below

        DefValPart ::= "DEFVAL" " { " value (defvalue ObjectSyntax) " } " | empty

        DisplayString ::=OCTET STRING SIZE (0..255)

 END

 indexSyntax ::=CHOICE {number  INTEGER (0..MAX),
                        string OCTET STRING,
                        object OBJECT IDENTIFIER,
                        address NetworkAddress,
                        IpAddress IpAddress }


RFC1155-SMI DEFINITIONS ::= BEGIN

Exports --- EVERYTHING
            ineternet, directory, mgmt,experimental, private, enterprises,OBJECT-TYPE,
            ObjectName, oBJECTSyntax, SimpleSyntax, ApplicationSyntax, NetworkAddress,
            IpAddress, Counter, Gauge,timeTicks, Opaque;

--the path to the root

internet          OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1}
directory         OBJECT IDENTIFIER ::= { internet 1}
mgmt              OBJECT IDENTIFIER ::= { internet 2}
experimental      OBJECT IDENTIFIER ::= { internet 3}
private           OBJECT IDENTIFIER ::= { internet 4}
enterprises       OBJECT IDENTIFIER ::= { private }
```

```
--definition of object types

OBJECT-TYPE MACRO ::=
BEGIN
        TYPE NOTATION ::= "Syntax"  type(TYPE ObjectSyntax)
                                  "ACCESS" Access
                                  "STATUT" Statut
        VALUE NOTATION ::= value(VALUE ObjectName)
        Access ::= "read-only" | "read-write" | "write-only" | "not-accessible"
        Status  ::= "mandatory" | "optional" | "obsolete "
END
--names of objects in the MIB
ObjectName ::=OBJECT IDENTIFIER
--syntax of objects in the MIB
ObjectSyntax ::= CHOICE {simple SimpleSyntax,
        --note that simple SEQUENCEs are not directly mentioned here to keep things simple
        --(i.e., prevent misuse). However, applicationwide types that are IMPLICITY encoded
        --simple SEQUENCEs may appear in the following CHOICE

                                  applicationwide ApplicationSyntax}

SimpleSyntax ::=CHOICE { number INTEGER,
                         string OCTET STRING,
                         object OBJECT IDENTIFIER,
                         empty NULL }

ApplicationSyntax ::=CHOICE { address NetworkAddress,
                              counter Counter,
                              gauge Gauge,
                              ticks TimeTicks,
                              arbitrary Opaque
                        --other applicationwide types, as they are defined, will be added here
                              }

--applicationwide types
NetworkAddress ::=CHOICE {internet IpAddress}
IpAddress ::=[APPLICATION 0]                                    --in network-byte order
                        IMPLICIT OCTET STRING (SIZE (4))
Counter::=[APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
Gauge::= [APPLICATION 2] IMPLICIT INTEGER (0..4294967295)
TimeTicks::= [APPLICATION 3] IMPLICIT INTEGER (0..4294967295)
Opaque::= [APPLICATION 4] OCTET STRING   --arbitrary ASN.1 value, "double-wrapped "
END
```

Annexe 1.2 : Code source de la méthode à associer au noyau de l'agent PING

```
/*****************************************************************/
/* Product Name  : Methode to use by PING agent                  */
/* File Name     : Methode.c                                     */
/* Type          : C source code file                            */
/* Function      : Implemente the use of agent PING              */
/* Author        : Maurice NDAYE-MUKUNA                          */
/* Copyrirght    : BULL 1999                                     */
/*****************************************************************/
```

```
static char *table[]={"address","state","ptransmis","pperdu","time"};

/*************************************************************/
/* Procédure d'initialisation par la méthode                 */
/*************************************************************/
static int init_meth()
{
 int i,j,k,l,m,rate,nboctet;
 struct table *ptr1;
 FILE *f1;
 int octet1,octet2,octet3,octet4;
 char toto[200],titi[10];
 fp=fopen("fconf","r");
 /* je compte le nombre des ressources du r,seau */
 for (i=0;fgets(host,62,fp)!=NULL;i++);
 nohost=i;
 ptr=malloc()sizeof(struct table)*nbhost);
 fp=fopen("fconf","r");
 for (ptr1=ptr;fgets(host,62,fp)!=NULL;ptr1++)
 {
  sprintf(cde,"ping -c 1 %s",host);
  fp2=popen(cde,"r");
  i=1;
  while (i<=10)
  {
   i++;
  }
  /* je r,cupŠre le r,sultat du ping */
  fgets(buf,127,fp2);
  sscanf(buf,"%s %s (%d.%d.%d.%d) : %d %s %s",toto,toto,&(ptr1->address[0],
  ptr1->address[1],ptr1->address[2],ptr1->address[3]);
  sprintf(cmde,"ksh ping.ksh %s",host);
  pclose(fp2);
  f1=popen(cmde,"r");
  fgets(buf,62,f1);
  sscanf(buf,"%d",&nboctet);
  if (nboctet==1){
     ptr1->state=1;
     fgets(buf,62,f1);
     sscanf(buf,"%s",toto);
     l=0;
     while (toto[l]='%')
     {
      l++;
     }
toto[l]='\0';
```

```
ptr1->pperdu=atoi(toto);
 ptr1->ptransmis=3;
 fgets(buf,62,f1);
 sscanf(buf,"%s",toto);
 l=0;
 while (toto[l]!='/')
 {
  l++;
 }
 l++;
 m=0;
 while (toto[l]!='/')
 {
  titi[m]=toto[l];
  l++;
  m++;
 }
 titi[m]='\0';
 ptr1->time=atoi(titi);}
 else { if (nboctet==2){
          ptr1->state=2;
          ptr1->ptransmis=3;
          ptr1->pperdu=0;
          ptr1->time=0;}
      }
 fclose(fq);
 pclose(f1);
 j=0;
 while (j<5)
 {
  switch(j)
  {
   case 0: sprintf(mess.val,"%d.%d.%d.%d",ptr1->address[0],
                ptr1->address[2],ptr1->addres[3]);
          break;
   case 1: sprintf(mess.val,"%d",ptr1->state);
          break;
   case 2: sprintf(mess.val,"%d",ptr1->ptransmis);
          break;
   case 3: sprintf(mess.val,"%d",ptr1->pperdu);
          break;
   case 4: sprintf(mess.val,"%d",ptr1->time);
          break;
  }
```

```
      strcpy(attName,tab[j]);
       for (i=0;i<4;i++)
       {
        name[i]=ptr1->address[i];
       }
       buildByName(mess.inst,attName,name,4);
       /* je fais la mise ... jour du noyau de l'agent */
       mess.result=OK;
       result=writeInstSock(mess,sock_meth);
       printMess(mess);
       j++;
      }
 }
 fclose(fp);
}

/*************************************************************/
/* Méthode qui traite les requêtes snmp reçues                */
/*************************************************************/
static CVoidType snmpReceive(CIntType sock)
{
 char *strcpy();
 int Ctrcmp(),rate;
 result=receiveSock(&mess,sock);
 if (result==GAM_ERR_CORE_NOT_READY)
 {
  return;
 }
 if (result != GAM_NO_ERROR){
    return;}
 if (mess.operation==DELET)
    result=buildName(mess.val,attName,name,&namelen);
    else result=buildName(mess.inst,attName,name,&namelen);
 if (result)
 {
  printGamError(result);
  return;
 }
```

```
switch(mess.operation)
{
/* je traite le GET ici */
case GET :{
        int i;
        struct table *ptr1;
        init_meth;
        z=0;
        i=0;
        /* Je recherche et met le pointeur sur la ressource interrog, */
        for (ptr1=ptr;z<nbhost;ptr1++)
        {
         if (memcmp(name,ptr1->address,sizeof(OidType)*4)==0)
            {
             i=z;
             z=nbhost+1;
            }
           else {z++;
                 i=z;
                }
        }
        /*je v,rifie si j'ai trouv, au moins une ressource*/
        if (i==0)
        {
         mess.result=NOK;
         break;
        }
        if (i>nbhost)
        {
         mess.result=NOK;
         break;
        }
        /*je cherche l'attribut qu'il faut retoun,*/
        i=0;
        while (i<5)
        {
         if (strcmp(attName,tab[i])!=0)
         {
          i++;
          else break;
         }
         /*je stocke les valeurs dans mess.val selon l'attribut*/
```

```
switch(i)
        {
         case 0: sprintf(mess.val,"%d.%d.%d.%d",ptr1->address[0],
                  ptr1->address[2],ptr1->addres[3]);
                 break;
         case 1: sprintf(mess.val,"%d",ptr1->state);
                 break;
         case 2: sprintf(mess.val,"%d",ptr1->ptransmis);
                 break;
         case 3: sprintf(mess.val,"%d",ptr1->pperdu);
                 break;
         case 4: sprintf(mess.val,"%d",ptr1->time);
                 break;
        }
        result=0;
        break;
       }
      }
     /*je traite le SET ici*/
case SET:
case SETLAST:
case CREATE:{
         int i,x,l,m,j,z;
         char voici[100];
         char name1[100];
         struct table *ptr1;
         char toto[200];
         char *ipaddress;
         i=0;
         z=0;
         for (ptr1=ptr;z<=nbhost;ptr1++)
         {
          if (memcmp(name,ptr1->address,sizeof(OidType)*4)==0)
          {
           nbh=nbhost;
           i=z+1;
           z=nbh+1;
           break;
          }
          else {
               z++;
               i=z;
              }
```

```
        if ((i==0)II(i>nbhost))
          {
          /*d,but du CREATE*/
          fp3=fopen("fconf","a");
          ipaddress=(char *)malloc(5);
          sprintf(ipaddress,"%c%c%c%c",name[0],name[1],name[2],name[3]);
          hp=gethostbyaddress(ipaddress,4,AF_INET);
          strcpy(voici,hp->h_name);
          l=0;
          while (voici[l]!='.')
          {
           name1[l]=voici[l];
           l++;
          }
          name1[1]='\0';
          fputs(name1,fp);
          fclose(fp3);
          init_meth();
          mess.result=OK;
          break;
          }
          if (i<=nbhost)
          {
          init_meth();
          mess.result=OK;
          break;
          }
case DELET :{
          /*je traite le DELETE ici*/
          struct table *ptr1;
          char voici[100];
          char name1[10];
          char *ipaddress;
          int l;
          fp4=fopen("fconf","r");
          if (fp4==NULL)
          {
           fq=fopen("fconf1","a");
           if (fq==NULL)
           {
            ipaddress=(chat *)malloc(5);
            sprintf(ipaddress,"%c%c%c%c",name[0],name[1],name[2],name[3]);
            hp=gethostbyaddress(ipaddress,4,AF_INET);
            strcpy(voici,hp->h_name);
```

```
            l=0;
              while (voici[l]!='.')
              {
               name1[l]=voici[l];
               l++;
              }
              name1[l]='\0';
              while (fgets(host,62,fp4)!=NULL)
              {
               if (memcmp(host,name1,strlen(name1))!=0)
                  fputs(host,fq);
                  else break;
              }
              fclose(fq4);
              fclose(fq);
              system("cp fconf1 fconf");
              system("rm fconf1");
              init_meth();
             }
             break;
            }
            }
           }
  }
  /*Envoi de la r,ponse*/
  if (mess.operation !=NOTIFY)
  {
   switch(result)
   {
    case 0: mess.reult=OK;
          break;
    case 1: if (mess.operation==GET)
              mess.result=OK;
              else mess.result=GAM_ERR_NOT_TABLE;
              break;
    case 2: mess.result=GAM_ERR_NO_SUCH;
          break;
   }
   printMess(mess);
   writeRespSock(mess,sock);
  }
 }
return;
```

```
/* MAIN */
main (int argc,char **argv)
{
 fd_set read_mask;
 CIntfType rc;
 exten short traceLevel;
 if (argc==2)
 {
  traceLevel=(short)atoi(argv[1]);
 }
 /* INITIALISATION */
 result=connectSock(agtName,oaservName,METHID,&sock_meth,&sock_snmp);
 if (result)
 {
  printGamError(result);
  exit(result);
 }
 init_meth;
 /* boucle pour attendre les messages venant du noyau GAM */
 while (1)
 {
  FD_ZERO(&read_mask);
  FD_SET(sock_snmp,&read_mask);
  rc=select(sock_snmp+1,&read_mask,(fd_set *)NULL,(fd_set *)NULL,NULL);
  if (rc==0)
  {
   break;
  }
  if (rf==1)
  {
   break;
  }
  /*un message est en attente */
  if (FD_ISSET(sock_snmp,&read_mask))
  {
   snmpreceive(sock_snmp);
  }
 }
}
```

Annexe 1.3 : Code source de l'automate

```
/*****************************************************************/
/* Product Name : Automate for Test-Agent SNMP                   */
/* File Name    : automate3.c                                    */
/* Type         : C source code file                             */
/* Function     : Test if SNMP agent is made OK                  */
/* copyright    : BULL 1999                                      */
/*****************************************************************/
#include <gamApi/GAMAPI.h>
#include <stdio.h>
#include <errno.h>
int reussi,compte,inv[100],pentier,choix,bonchoix;
char agent[25],cestout[25],ferme[25];
char *faux[100],agent[25];
struct table {
            char genretype[25];
            struct table *suivant;
           };
struct table *ptr1,*ptr12,*ptr2,*ptr21;
char attribut[25];
char *tbreq[]={"Snmpget","Snmpset"};
char *tabpt3[]={"            ","NULL"};
/*****************************************************************/
/* Cette procédure modifie les bornes d'une s,rie d'attributs     */
/*****************************************************************/
static ChangerAttribut(char agent10[25])
{
 int present,min,max;
 char fini[25],termine[25],buf[62],att[25],type[25],var[62];
 FILE *p1,*p2;
 sprintf(var,"%s.BORNE1",agent10);
 p2=fopen(var,"a");
 sprintf(termine,"y");
 printf("Entre le nom de l'attribut dont il faut changer les bornes :\n");
 printf("Entre 'y' s'il n'y a plus d'autres attributs :\n");
 scanf("%s",fini);
 while (strncmp(fini,termine,strlen(termine))!=0)
 {
  present=0;
  sprintf(var,"%s.BORNE",agent10);
  p1=fopen(var,"r");
```

```
while (fgets(buf,120,p1)!=NULL)
 {
 sscanf(buf,"%s %s %d %d",att,type,&min,&max);
 if (strcmp(att,fini)==0){
  printf("%s\n",att);
  printf("%s\n",type);
  printf("Minimum %d\n",min);
  scanf("%d",&min);
  printf("Maximum %d\n",max);
  scanf("%d",&max);
  sprintf(buf,"%s %s %d %d\n",att,type,min,max);
  fputs(buf,p2);
  present=1;
  break;}
  else fputs(buf,p2);
 }
 fclose(p1);
 if(present==0){
  printf("Cet attribut n'existe pas dans la MIB\n");
  printf("Entrer le nom de l'attribut dont il faut changer les bornes\n");
  printf("Entree 'y' s'il n'y a plus d'autres attributs \n");
  scanf("%s",fini);
  else{printf("Entrer le nom de l'attribut dont il faut changer les bornes\n");
       printf("Entrer 'y' s'il n'y a plus d'autres attributs\n");
       scanf("%s",fini);}
 }
 fclose(p2);
 sprintf(var,"mv %s.BORNE1 %s.BORNE",agent10,agent10);
 system(var);
}
/*************************************************************************/
/* Cette procédure vérifie s'il existe d,j... un fichier
*/
/* <agent>.BORNE correspondant au nom d'un agent donné
*/
/*************************************************************************/
static int VerifiExistenceAgent(char agent11[25])
{
 int i,j,k,reponse;
 char var[62],buf[512],nom[25];
 FILE —p4;
 system("rm FICHIERRepertoire");
 sprintf(var,"%s.BORNE >FICHIERRepertoire",agent11);
 system(var);
 p4=fopen("FICHIERRepertoire","r");
 reponse=0;
```

```
while (fgets(buf,120,p4)!=NULL)
 {
 i=0;
 j=0;
 while (buf[i]!='.')
 {
  nom[j]=buf[i];
  i++;
  j++;
 }
 nom[j]='\0';
 if (strcmp(nom,agent11)==0){
    reponse=1;
    break;}
    else reponse=0;
 }
 return(reponse);
}

/*************************************************************************/
/* Cette procédure effectue la validation des initialisations            */
/* d,j... faites d'un agent. Elle reçoit comme entrée le nom             */
/* de cet agent                                                          */
/*************************************************************************/
static ValideAttribut(char agent10[25])
{
 int mini,maxi;
 char avant[62],buf[512],att[25],type[25];
 FILE *p5,*p6;
 sprintf(avant,"%s.BORNE",agent10);
 p5=fopen(avant,"r");
 sprintf(avant,"%s.BORNE1",agent10);
 p6=fopen(avant,"a");
while (fgets(buf,62,p5)!=NULL)
 {
 sscanf(buf,"%s %s %d %d",att,type,&mini,&maxi);
 printf("%s\n",att);
 printf("%s\n",type);
 printf("Minimum %d\n",mini);
 scanf("%d",&mini);
 printf("Maximum %d\n",maxi);
 scanf("%d",&maxi);
 sprintf(buf,"%s %s %d %d\n",att,type,mini,maxi);
 fputs(buf,p6);
 }
```

```
fclose(p5);
 fclose(p6);
 sprintf(avant,"mv %s.BORNE %s.BORNE",agent10,agent10);
 system(avant);
}

/*****************************************************************/
/* Cette procédure fait un dump de snmpmany et redirige          */
/* les instances de la MIB dans le fichier FICHIER2              */
/*****************************************************************/
static EffectueDumpSnmpmany(char agent2[25])
{
 char cde[25];
 FILE *f1;
 sprintf(cde,"snmomany %s > FICHIER2",agent2);
 f1=popen(cde,"r");
 pclose(f1);
}
/*****************************************************************/
/* Cette procédure crée une liste des types                      */
/* mathématiquement testables d'un agent. Il faut créer          */
/* un tableau qui contient tous les types qui sont               */
/* contenus dans la MIB                                          */
/*****************************************************************/
static struct table *CreerTableTypes(char agent[25])
{
 int i;
 static char *t[]={"INTEGER","Counter","Gauge","TimeTicks"};
 char buf1[512],buf2[512],attribut1[25],truc[25],vraitruc[25];
 char type1[25],rien[25],accolade[1],rfc[25],suite[25],avant[40];
 struct table *ptr10,*ptr11;
 FILE *p10,*p11;
 sprintf(vraitruc,"::=");
 sprintf(rfc,"%s",agent);
 sprintf(avant,"../rfc/");
 sprintf(suite,".rfc");
 strcat(rfc,suite);
 strcat(avant,rfc);
 p10=fopen(avant,"r");
 ptr10=malloc(sizeof(struct table));
 ptr11=malloc(sizeof(struct table));
 ptr11=ptr10;
```

```
for (i=1;i<=4;i++)
 {
 switch(i)
 {
  case 1: sprintf(ptr11->genretype,"INTEGER");
          ptr11->suivant=malloc(sizeof(struct table));
          ptr11=ptr11->suivant;
          break;
  case 2: sprintf(ptr11->genretype,"Counter");
          ptr11->suivant=malloc(sizeof(struct table));
          ptr11=ptr11->suivant;
          break;
  case 3: sprintf(ptr11->genretype,"Gauge");
          ptr11->suivant=malloc(sizeof(struct table));
          ptr11=ptr11->suivant;
          break;
  case 3: sprintf(ptr11->genretype,"TimeTicks");
          ptr11->suivant=malloc(sizeof(struct table));
          ptr11=ptr11->suivant;
          break;
 }
}
sprintf(accolade,"{");
sprintf(rien,"                ");
sprintf(type1,"                ");
ptr11->suivant=malloc(sizeof(struct table));
while (fgets(buf1,120,p10)!=NULL)
{
 sscanf(buf1,"%s %s %s",attribut1,truc,type1);
 if (strcmp(truc,vraitruc)==0){
    for (i=0;i<=4;i++)
    {
     switch (i)
     {
      case 0: if (strcmp(type1,t[i]==0){
                strcpy(ptr11->genretype,attribut1);
                ptr11->suivant=malloc(sizeof(struct table));
                ptr11=ptr11->suivant;
                sprintf(type1,"                ");
                break;
      case 1: if (strcmp(type1,t[i]==0){
                strcpy(ptr11->genretype,attribut1);
                ptr11->suivant=malloc(sizeof(struct table));
                ptr11=ptr11->suivant;
                sprintf(type1,"                ");
        break;
```

```
case 2: if (strcmp(type1,t[i]==0){
            strcpy(ptr11->genretype,attribut1);
            ptr11->suivant=malloc(sizeof(struct table));
            ptr11=ptr11->suivant;
            sprintf(type1,"                    ");
            break;
      case 3: if (strcmp(type1,t[i]==0){
            strcpy(ptr11->genretype,attribut1);
            ptr11->suivant=malloc(sizeof(struct table));
            ptr11=ptr11->suivant;
            sprintf(type1,"                    ");
            break;
      case 4:if (strcmp(type1,rien)!=NULL){
            fgets(buf1,120,p10);
            sscanf(buf1,"%s %s",type1,accolade);
            for (i=0;i<=3;i++)
            {
             switch(i)
             {
              case 0:if (strcmp(type1,t[i])==0){
                     strcpy(ptr11->genretype,attribut1);
                     ptr11->suivant;
                     ptr11->suivant=malloc(sizeof(strcut table));
                     break;
                     else break;
              case 1:if (strcmp(type1,t[i])==0){
                     strcpy(ptr11->genretype,attribut1);
                     ptr11->suivant;
                     ptr11->suivant=malloc(sizeof(strcut table));
                     break;
                     else break;
              case 2:if (strcmp(type1,t[i])==0){
                     strcpy(ptr11->genretype,attribut1);
                     ptr11->suivant;
                     ptr11->suivant=malloc(sizeof(strcut table));
                     break;
                     else break;
```

```
case 3:if (strcmp(type1,t[i])==0){
                    strcpy(ptr11->genretype,attribut1);
                    ptr11->suivant;
                    ptr11->suivant=malloc(sizeof(strcut table));
                    break;
                    else break;

            }
            }

    }
   }
 }
 ptr11->suivant=NULL;
 ptr1=ptr10;
 returne(ptr1);
 }
/*************************************************************/
/* Cette procédure permet de fixer les bornes                */
/* des attributs testables d'un agent.                       */
/*************************************************************/
static AffecteBorneAttribut(char agent1[25], struct table **ptr1)
{
 FILE *p8,*p9;
 char buf1[512],buf2[512],fin[10];
 char objet[25],objet1[25],syntaxe[25],attribut1[25];
 char suite[25],rfc[25],type1[25],kind[25],bout[25];
 char avant[62],borne[62],effacer[62];
 int l,present,minimum,maximum;
 struct table *ptr11;
 ptr11=*ptr1;
 sprintf(rfc,"%s",agent1);
 sprintf(effacer,"rm %s.BORNE",agent1);
 system(effacer);
 sprintf(suite,".rfc");
 strcat(rfc,suite);
 sprintf(borne,"%s.BORNE",agent1);
 p8=fopen(avant,"r");
 p9=fopen(borne,"a");
 sprintf(objet1,"OBJECT-TYPE");
 sprintf(bout,"END");
```

```
fgets(buf1,80,p8);
 while (fgets(buf1,80,p8)!=NULL)
 {
 if (strncmp(bout,buf1,strlen(bout))==0){
    fclose(p8);
    fclose(p9);
    break;
    else {sscanf(buf1,"%s %s",attribut1,objet);
         if (strcmp(objet,objet1)==0){
            fgets(buf1,"%s %s",syntaxe,type1);
            present=0;
            ptr11=*ptr1;
            while ((ptr11->suivant!=NULL)II(present==1))
            {
             if (strcmp(ptr11->genretype,type1)!=0){
               ptr11=ptr11->suivant;}
               else {present=1;
               strcpy(kind,type1);
               break;}
            }
            if (present==1){
              printf("Donnez les bornes de cet attribut : %s\n",attribut1);
              for (l=1;l<=2;l++)
              {
               fgets(buf1,80,p8);
              }
              fgets(buf1,80,p8);
              printf("Son type est : %s\n",kind);
              printf("Sa description est : %s\n",buf1);
              printf("Minimum =\n");
              scanf("%d",&minimum);
              printf("Maximum =\n");
              scanf("%d",&maximum);
              sprintf(buf2,"%s %s %d %d\n",attribut1,kind,minimum,maximum);
              fputs(buf2,p9);}}}
 }
}
```

```
/*****************************************************************/
/* Cette proc,dure permet de tester si les valeurs des           */
/* attributs dans la MIB sont contenues ou pas dans les          */
/* bornes                                                        */
/*****************************************************************/
static struct table *RealiseTest(char agent10[25],int compteur,int inv[100])
{
 int k,l,r,l,d,s,m,min,max,value2;
 char value2[20],nom1[25],kind1[25],tnom2[25],nom2[25];
 char buf1[512],buf[512],nom1[25];
 FILE *f8,*f9;
 struct table *ptr20,*ptr21;
 ptr20=malloc(sizeof(struct table));
 ptr21=ptr20;
 f8=fopen("FICHIER2","r");
 sprintf(fin,"End of MIB");
 r=0;
 k=1;
 compteur=0;
 while (fgets(buf1,120,f8)!=NULL)
 {
  if (strncmp(buf1,fin,strlen(fin))!=0)
  k++;
  sscanf(buf1,"%s %s %s %s %s %s",tnom2,nom2,kind2,tvalue2,value2);
  i=0;
  while (nom2[i]!='.')
  {
   nom3[i]=nom2[i];
   i++;
  }
  nom3[i]='\0';
  sprintf(avant,"%s.BORNE",agent10);
  f9=fopen(avant,"r");
  while (fgets(buf2,82,f9)!=NULL)
  {
   sscanf(buf2,"%s %s %d %d",nom1,kind1,&min,&max);
   if (strncmp(nom1,nom3,strlen(nom1))==0){
     if (atoi(value2)<min){
        inv[r]=0;
        compte++;
        strcpy(ptr21->genretype,nom2);
        ptr21->suivant=malloc(sizeof(struct table));
        r++;
```

```
        ptr21=ptr21->suivant;
        break;}
        else {if (value2>max){
        else {if (atoi(value2)>max){
                inv[i]=0;
                compte++;
                strcpy(ptr21->genretype,nom2);
                ptr21->suivant=malloc(sizeof(struct table));
                r++;
                ptr21=ptr21->suivant;
                break;
                else {inv[r]=1;
                r++;}}}
 }
 fclose(f9);}
 else break;
 }
 r--;
 compteur++;
 ptr21->suivant=NULL;
 fclose(f8);
 fclose(f9);
 return(ptr20);
}

/*****************************************************************/
/* Cette procédure permet de tirer la conclusion sur la          */
/* validité ou non de l'agent qu'on voulait tester.              */
/* A partir de compte, inv et faux, on sait conclure.            */
/*****************************************************************/
static ConclureValidite(int compte,int inv[100],struct table **ptr3)
{
 int j;
 struct table *ptr30;
 ptr30=*ptr3;
 if (compte<=10){
    printf("cet agent est valide\n");
    printf("voici la liste des attributs incoh,rents :\n");
    while (ptr30->suivant!=NULL)
    {
     printf(cet attribut '%s' est incoh,rent\n",ptr30->genretype);
     ptr30=ptr30->suivant;
    }}
    else {printf("cet agent est invalide\n");
         printf("voici la liste des attributs incoh,rents\n");
  while (ptr30->suivant!=NULL)
```

```
{
        printf("cet attribut '%s' est incoh,rent\n",ptr30->genretype);
        ptr30=ptr30->suivant;
        }}
}

/*****************************************************************/
/* Cette procédure lance l'affectation des bornes                */
/* aux types d'objet de la MIB                                   */
/*****************************************************************/
static AffecteBorneAgent(char agent20[25])
{
 int pentier,choix,bonchoix;
 EffectueDumpsnmpmany(agent20);
 pentier=VerifiExistenceAgent(agent20);
 if (pentier ==0){
    ptr1=CreerTableType(agent20);
    AffecteBorneAttribut(agent20,&ptr1);}
   else {
        printf("cet agent a d,j... ,t, initialis, en bornes!\n");
        printf("Pour valider chaque objet, taper 1 + puis Enter\n");
        printf("Pour changer un attribut, taper 2 + Enter\n");
        scanf("%d",&choix);
        bonchoix=0;
        while (bonchoix==0)
        {
         switch(choix)
         {
         case 1: ValiderAttribut(agent20);
                 bonchoix=1;
                 break;
         case 2: ChangerAttribut(agent20);
                 bonchoix=1;
                 break;
         default : break;
         }
        }
       }
}
```

```
/*****************************************************************/
/*Cette procédure vérifie si la requête fournie est correcte     */
/*****************************************************************/
static int VerifiSiBonneRequete(char req[20])
{
 int e;
 while (estla !=0)
 {
  if (strcmp(req,tabreq[e])==0){
     estla=1;}
     else{
         e++
         }
 }
 return estla;
}

/*****************************************************************/
/*Cette procédure r,alise le test d'une requête snmp             */
/*****************************************************************/
static TesteRequeteSNMP(char agent23[25],rt1[20],rt2[20],rt3[20],rt4[20])
{
 int e,k,finfichier;
 char v1[10],v2[15],buffer[62],fich[20],suite[5];
 char inst[7],inst2[15];
 FILE *fichierp,*fichier;
 e=1;
 /*Je recherche la bonne requ^te dans cette boucle*/
 while (strcmp(tabreq[e],pt1)!=0))
 {
  switch (e)
  {
   case 1: /*Il s'agit de la requete Snmpget*/
         sprintf(commande,"%s %s",pt1,pt2);
         sprintf(suit,".int");
         fichier=popen(commande,"r");
         k=1;
         while (k<3)
         {
          fgets(buffer,62,fichier);
          k++;
         }
```

```
        fgets(buffer,62,fichier);
        sscanf(buffer,"%s %s",v1,v2);
        printf("La valeur de l'instance dans la MIB est %s\n",v2);
        printf("La valeur de votre requete est %s\n",v2);
        pclose(fichier);
  case 2: /*Il s'agit de la requete Snmpset*/
        sprintf(commande,"%s %s %s",pt1,pt2,pt3);
        system(commande);
        sprintf(requconsult,"Snmpget");
        sprintf(commande,"%s %s",reqconsult,pt2);
        sprintf(suit,".int");
        fichier=popen(commande,"r");
        k=1;
        while (k<3)
        {
         fgets(buffer,62,fichier);
         k++;
        }
        fgets(buffer,62,fichier);
        sscanf(buffer,"%s %s",v1,v2);
        printf("La valeur de l'instance dans la MIB est %s\n",v2);
        printf("La valeur de votre requete est %s\n",pt3);
        pclose(fichier);
  default : printf("La requete n'est pas correcte");
          break;
 }
}

}

/*****************************************************************/
/*cette procédure lance le test d'une requête snmp                */
/*****************************************************************/
static TraitementTest(char agent22[25])
{

 int vraierequete,descision,jaichoisi,compte,inv[100];
 char stop[1],arreter[1],requete[62];
 char t1[20],t2[20],t3[20],t4[20];
 sprintf(stop,"o")
 sprint
```

```
while (strcmp(stop,arreter)!=0)
{
printf("Entre 1 si tu veux tester toutes les instances de la MIB :\n");
printf("Entre 2 si tu veux tester un type d'objet de la MIB :\n");
printf("Entre 3 si tu veux tester une instance de la MIB :\n");
printf("Entre 4 si tu veux tester un requˆte snmp :\n");
scanf("%d",jaichoisi);
printf("Entre la requete ... tester :");
scanf("%d",requete);
while (bonchoix!=1)
{
 switch(jaichoisi)
 {
  case 1: RealiseTest(agent22,compte,inv);
          ConcluValidite(compte,inv,&ptr2);
          bonchoix=1;
          break;
  case 2: RealiseTestUnTypeObjet(agent22,compte,inv);
          ConcluValidite(compte,inv,&ptr2);
          bonchoix=1;
          break;
  case 3: RealiseTestUneInstance(agent22);
          bonchoix=1;
          break;
  case 4: printf("Veuillez fournir votre requˆte snmp :\n");
          sscanf(requete,"%s %s %s %s",t1,t2,t3,t4);
          vraierequete=VerifiSiBonneRequete(requete);
          if (vraierequete==1){
             TesteRequeteSNMP(agent22,t1,t2,t3,t4);}
             else {printf("Votre requˆte n'est pas correcte \n");
 }
 }
}
}
```

```
main()
{
 ptr12=malloc(sizeof(struct table));
 sprintf(cestout,"q");
 int quoi;
 /*
  *on v,rifie si il n'a plus envie de faire quelque chose
  */
 while (strncmp(ferme,cestout,strlen(cestout))!=0)
 {
 printf("Entre 1 si tu veux affecter des bornes ou changer une borne : ");
 printf("Entre 2 si tu veux r,aliser un test : ");
 printf("Entre 3 si tu veux quitter : ");
 scanf("%s",quoi);
 /*
  *on aiguille selon ce qu'il veut faire*/
  */
 int non=0;
 while (non==0)
 {
  switch(quoi)
  {
   case 1: printf("Entre le nom de l'agent que tu veut borner :");
           scanf("%s",agent);
           AffecteBornesAgent(agent);
           non=1;
           break;
   case 2: printf("Entre le nom de l'agent que tu veux tester :");
           scanf("%s",agent);
           TraitementTest(agent);
           non=1;
           break;
   case 3: printf("Entre le nom de l'agent que tu veux valider :");
           scanf("%s",agent);
           ValidationAgent(agent);
           non=1;
           break;
   default : printf("Veuillez entre un bon nombre !\n");
             break;
  }
  printf("Un autre agent ... tester(y)? sinon(q)");
  scanf("%s",ferme);
  }

 }
}
```

Annexe 1.4 : Fichier rc.Pingmib **: contenu du script de lancement du noyau de l'agent PING**

```
Unset LIBPATH
gamAgDir=/users.people/maurice/gamagt/agts
gamInstDir=$gamAgtDir/inst

gamAgtaPort=4010

pid=‘ps guww|grep ″$ganAgtDir/pingmibd -p ″ |grep -v ″grep″ |awk ´{ print $2 }´ `
if [ ! -z ″pid″ ] ; then
echo ″Agent pingmibd is already running (pid=$pid) . . . ″
echo ″Exiting ! ″
exit 1
fi
$gamAgtDir/pingmibd -p ^$gamAgtPOrt
-df $gamAgtDir/pingmib.snmpdef
-cf $gamAgtDir/pingmib.Communauté
-gd $ gamAgtDir
-id $ gamAgtDir
-tp 1
-tl 0
```

Annexe 1.5 : fichier rc.methode1 : contenu du script de lancement de la méthode de l'agent PING

```
GamMetDir=/users/people/maurice/gamagt/methodes
case "$1" in
'start')
pid=`ps -e | grep pingmibd | grep -v grep | awk '{ print $1 }'`
if [ "n$pid" = "n" ] ; then
echo "Noyau pingmib is not running..... "
echo   "Exiting ! "
exit 1
fi

$gamMetDir/Methode1 2>>/tmp/methode.log >>/tmp/methode.log &
#$gamMetDir/Methode

echo "methode--PING"
;;
'stop')
pid=`ps -e | grep Methode1 | grep -v grep | awk ' { print $1 }'`
if [ "n$pid" = "n" ] then
        echo "methode Methode1 is not running..... "
        echo   "Exiting ! "
        exit 1
fi
kill $pid
echo  "methode-methode1 shutdown"

        ;;
*)
        echo  "Usage : $o [ start | stop ] "
        exit 1
        ;;
esac

exit o
```

Annexe 1.6 : fichier Ping.Ksh : programme de lancement de la commande Ping

```
/%/          { if  ($7=="100%" )
print 2
else  {
print 1
print $7
}
}
/ //         { print $4 }
/NOT FOUND/ {print 2}
```

Fichier **Ping.awk** : programme de traitement du résultat de la commande Ping

```
# !/usr :bin/Ksh
#Ce shell lance un Ping sur la machine qui lui est passé
#en paramètre, puis traite le résultat de ce Ping.
#à l'aide de AWK, pour donner en sortie :
# -si la machine répond : 1
#                          pourcentPacketsPerdus
#                          tpsMin/tpsMax/tpsAvg
#
#si la machine est inaccessible : 2

Ping -c 3 $1 2>&1  | awk -f Ping.awk
```

Annexe 2 : présentation de l'API de GAM

2.1 Initialisation.

1) Nom : **methInit**
Type retourné : GAMResultType
Paramètres : const char *gamAGTDir ;
const char *pipesDir ;
const char *agtName ;
long *pipeRespSnmp ;
long *pipeRespSnmp ;
long *pipeRespSnmp ;
long *pipeRespSnmp ;

Description : Initialise la méthode.
On doit préciser le répertoire du noyau de l'agent GAM dans gamAgtDir, le répertoire des pipes utilisés par l'agent GAM dans pipesDir et le nom de l'agent dans agtName.
La fonction methInit ouvre les tubes nommés utilisés pour communiquer avec le noyau de l'agent GAM et renvoie des pointeurs sur les descripteurs de ces tubes.
Il est important de noter que les paramètres de la fonction methInit correspondant aux descripteurs de ces tubes sont des pointeurs, car ils sont valorisés durant l'exécution de la fonction. La fonction methInit envoie au noyau de l'agent GAM un message de type WRITEINST valorisant l'instance gamInit-<agtName>à METHBEGIN.
Si le noyau n'est pas encore initialisé, seule la création et l'ouverture des tubes en lecture est réalisé dans la fonction methInit, dans ce cas une erreur GAM_ERR_CORE_NOT_READY est renvoyée et les descripteurs des tubes en écriture sont valorisés à 0.
Lorsque le noyau s'initialisera, la méthode en sera prévenue par la réception d'une requête SET valorisant l'instance gamInit-<agtName> à COREBEGIN.
La méthode devra alors ouvrir en écriture non bloquante les tubes pipeRespSnmp et pipeReqMeth.
Si le noyau est déjà initialisé, l 'envoi du WRITEINST valorisant l'instance gamInit-<agtName> à METHBEGIN déclenchera le mode MUTE.
Il est impératif que la fonction methInit soit appelée par la méthode dès son lancement.

Code retour : GAM_NO_ERROR en cas de réussite si le noyau est déjà initialisé, GAM_ERROR_CORE_NOT_READY en cas de réussite si le noyau n'est pas encore initialisé, GAM__ERRPIPE pour une erreur sur un tube nommé, GAM_ERR_SIGNAL pour une erreur de signal et GAM_ERR_UNIX pour une erreur système.

2) Nom : **connectSock**
Type : const char *agtName ;
const char *serverName ;
const int methId ;
int *sock_meth ;
int *sock_snmp ;

Description : réalise la connexion d'une méthode au noyau GAM. Le nom de l'agent est passé dans agtName, le nom de la machine sur laquelle tourne le noyau dans serverName et l'identifiant de la méthode (choisi arbitrairement par le développeur de manière à ce qu'il soit unique pour le même noyau GAM) dans methId. Le descripteur de la socket sur laquelle la méthode transmettra ses requêtes (writeInst , readInst) au noyau est retourné dans sock_meth et le descripteur de la socket sur laquelle le noyau soumettra à la méthode les requêtes provenant du manager est retourné dans sock_snmp. La fonction connectSock réalise les connexions des deux sockets TCP et envoie au noyau un message lui faisant prendre connaissance de l'identifiant de la méthode.

Code retour : GAM_NOERROR en cas de réussite, GAM_ERR_SOCVK si l'une ou l'autre socket est impossible à créer ou à connecter (noyau absent).

3) Nom : initFusion
Type retourné : GAMResultType

Paramètres : const char *gamAgtDir ;
const char *instDir ;
const char *tmpDir ;
const char *agtName ;

Description : réalise une fusion entre fichiers d'instances. On doit préciser le répertoire du noyau de l'agent GAM dans gamAgtDir le répertoire où se trouvent les fichiers d'instances et incrémental dans instDir et le nom de l'agent dans agtName.
La fusion nécessite également un répertoire de travail temporaire qu'on doit donc préciser dans le paramètre tmpDir.
La fonction initFusion est bloquante jusqu'à ce qu'un fichier d'instances fusionné soit disponible.
Il est impératif que la fonction initFusion soit appelée par la méthode si celle-ci doit s'initialiser par rapport au fichier d'instances.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_FUSION sinon.

2.2 Lecture et écriture d'instances.

Nom : readInstDirect
Type retourné : GAMResultType

Paramètres : GAMMessType *messPipe ;
const char *gamAgtDir,
const char *agtName ;
const long pipeReqMeth ;
const long pipeRespMeth ;
const long timeOut

Description : lit la valeur d'une instance.
Le nom de l'instance avec son suffixe doit être placé dans messPipe.inst.
L'opération messPipe.operation est positionnée à READINST par la fonction.
La valeur de l'instance est retournée dans messPipe.val.
On doit préciser le répertoire du noyau de l'agent GAM dans gamAgtDir et le nom de l'agent dans agtName.
Il faut également indiquer les descripteurs des tubes nommés piperReqMeth et pipeRespMeth.
Ces tubes auront pu être ouverts dans la méthode grâce à la fonction methInit.
Le message de la requête est écrit par la fonction dans le tube nommé pipeReqMeth.
Le message de réponse est lu par la fonction dans le tube nommé pipeRespMeth.
Les échanges entre la méthode et le noyau de l'agent GAM sont synchronisés par la fonction grâce au signal réservé SIGUSR1.
S'il n'y a pas de réponse après une durée de timeOut secondes, la fonction renvoie une erreur.
Le code d'erreur renvoyé est également reporté dans messPipe.result.
Il est important de noter que le premier paramètre de la fonction readNextDirect est un pointeur sur une structure de type GAMMessType, car différents champs de cette structure sont positionnés durant l'exécution de la fonction.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_NO_SUCH si l'instance précédents n'existe pas, GAM_ERR_NO_SUFFIX si l'on omet de préciser le suffixe de l'instance précédente, GAM_ERR_PIPE pour une erreur sur un tube nommé, GAM_ERR_SIGNAL pour une erreur de signal, GAM_ERR_END_TEMPO si le noyau n' a pas répondu après l'expiration du timeOut, et GAM_ERR_UNIX pour une erreur système.

2)

Nom	: **readNextDirect**
Type retourné	: GAMResultType
Paramètres	: GAMMessType *messPipe ; const char *gamAgtDir, const char *agtName ; const long pipeReqMeth ; const long pipeRespMeth ; const long timeOut
Description	: lit la valeur de l'instance suivante dans l'ordre. Le nom de l'instance précédente avec son suffixe doit être placé dans messPipe.inst. On peut chercher la première instance gérée par le noyau da l'agent GAM en indiquant root.O dans le paramètre messPipe.Inst. L'opération messPipe.operation est positionnée à READNEXT par la fonction. Le nom de l'instance suivante est retourné dans mess.Pipe.inst. La valeur de l'instance suivante est retournée dans messPipe.val. On doit préciser le répertoire du noyau de l'agent GAM dans gamAgtDir et le nom de l'agent dans agtName. Il faut également indiquer les descripteurs des tubes nommés piperReqMeth et pipeRespMeth. Ces tubes auront pu être ouverts dans la méthode grâce à la fonction methInit. Le message de la requête est écrit par la fonction dans le tube nommé pipeReqMeth. Le message de réponse est lu par la fonction dans le tube nommé pipeRespMeth. Les échanges entre la méthode et le noyau de l'agent GAM sont synchronisés par la fonction grâce au signal réservé SIGUSR1. S'il n'y a pas de réponse après une durée de timeOut secondes, la fonction renvoie une erreur. Le code d'erreur renvoyé est également reporté dans messPipe.result. Il est important de noter que le premier paramètre de la fonction readNextDirect est un pointeur sur une structure de type GAMMessType, car différents champs de cette structure sont positionnés durant l'exécution de la fonction.
Code retour	: GAM_NO_ERROR en cas de réussite, GAM_ERR_NO_SUCH si l'instance précédents n'existe pas, GAM_ERR_NO_SUFFIX si l'on omet de préciser le suffixe de l'instance précédente, GAM_ERR_PIPE pour une erreur sur un tube nommé, GAM_ERR_SIGNAL pour une erreur de signal, GAM_ERR_END_TEMPO si le noyau n'a pas répondu après l'expiration du timeOut, et GAM_ERR_UNIX pour une erreur système.

3)

Nom : writeInstDirect

Type retourné : GAMResultType

Paramètres : GAMMessType *messPipe ;
const char *gamAgtDir,
const char *agtName ;
const long pipeReqMeth ;

Description : écrit une instance dans les bases de données triées du noyau de l'agent GAM.
Le nom de l'instance avec son suffixe doit être placé dans messPipe.inst et sa valeur dans messPipe.val.
L'opération messPipe.operation est positionnée à WRITEINST par la fonction.
On doit préciser le répertoire du noyau de l'agent GAM dans gamAgtDir et le nom de l'agent dans agtName.
Il faut également indiquer les descripteurs des tubes nommés piperEqMeth et pipeRespMeth.
Ces tubes auront pu être ouverts dans la méthode grâce à la fonction methInit.
Le message de la requête est écrit par la fonction dans le tube nommé pipeReqMeth.
L'envoi du message de la requête vers le noyau de l'agent GAM est synchronisé par la fonction grâce au signal SIGUSR1.
Aucun message d'acquittement de la part du noyau n'est attendu. Ainsi, le code retour e la fonction rend uniquement compte de l'envoi de message au noyau de l'agent GAM. En particulier, si l'on tente de créer une instance non tabulaire, le noyau de l'agent empêchera bien cette création illicite, mais la fonction writeInstDirect n'aura pas renvoyé d'erreur.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_NO_SUCH si l'instance précédents n'existe pas, GAM_ERR_NO_SUFFIX si l'on omet de préciser le suffixe de l'instance précédente, GAM_ERR_Pipe pour une erreur sur un tube nommé,, GAM_ERR_END_TEMPO si le noyau n' a pas répondu après l'expiration du timeOut, et GAM_ERR_UNIX pour une erreur système.

4)

Nom : readInstSock
Type retourné : GAMResultType

Paramètres : GAMMessType *mess ;
int sock_meth,
long timeOut

Description : Lit la valeur de l'instance en mode sockets.
Le nom de l'instance avec son suffixe doit être placé dans mess.inst.
L'opération mess.operation est positionnée à WRITEINST par la fonction.
La valeur de l'instance est retournée dans mess.val.
On doit préciser le descripteur de la socket sock_meth.
Si le noyau n'a pas répondu après une durée de timeOut secondes, la fonction renvoie une erreur.
Le code d'erreur renvoyé est également reporté dans mess.result.
Il est important de noter que le premier paramètre de la fonction readInstSock est un pointeur sur une structure de type GAMMessType, car différents champs de cette structure sont positionnés durant l'exécution de la fonction.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_NO_SUCH si l'instance précédents n'existe pas, GAM_ERR_NO_SUFFIX si l'on omet de préciser le suffixe de l'instance précédente, GAM_ERR_SOCK pour une erreur de manipulation de la socket, GAM_ERR_END_TEMPO si le noyau n' a pas répondu après l'expiration du timeOut, et GAM_ERR_UNIX pour une erreur système.

5)

Nom : readNextSock
Type retourné : GAMResultType

Paramètres : GAMMessType *mess ;
int sock_meth,
long timeOut

Description : Lit la valeur de l'instance suivante dans l'ordre lexicographique en mode sockets.
Le nom de l'instance avec son suffixe doit être placé dans mess.inst.
On peut rechercher la première instance gérée par le noyau de l'agent en indiquant root.O dans le paramètre mess.inst.
L'opération mess.operation est positionnée à WRITENEXT par la fonction.
Le nom de l'instance suivante est retournée dans mess.inst.
La valeur de l'instance suivante est retournée dans mess.val.
On doit préciser le descripteur de la socket sock_meth.
Si le noyau n'a pas répondu après une durée de timeOut secondes, la fonction renvoie une erreur.
Le code d'erreur renvoyé est également reporté dans mess.result.
Il est important de noter que le premier paramètre de la fonction readInstSock est un pointeur sur une structure de type GAMMessType, car différents champs de cette structure sont positionnés durant l'exécution de la fonction.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_NO_SUCH si l'instance précédents n'existe pas, GAM_ERR_NO_SUFFIX si l'on omet de préciser le suffixe de l'instance précédente, GAM_ERR_SOCK pour une erreur de manipulation de la socket, GAM_ERR_END_TEMPO si le noyau n' a pas répondu après l'expiration du timeOut, GAM_ERR_LAST_INST si l'instance suivante n'existe pas et GAM_ERR_UNIX pour une erreur système.

6)
Nom : writeInstSock
Type retourné : GAMResultType

Paramètres : GAMMessType mess ;
int sock_meth,

Description : Erict une instance dans les bases de données triées du noyau de l'agent GAM en mode socket.
Le nom de l'instance avec son suffixe doit être placé dans mess.inst et sa valeur dans mess.val.
L'opération mess.operation est positionnée à WRITEINST par la fonction.
On doit préciser le descripteur de la socket sock_meth.
Aucun message d'acquittement de la part du noyau n'est attendu.
Ainsi, le code retour dee la fonction rend uniquement compte de l'envoi du message au noyau de l'agent GAM
En particulier, si l'on tente de créer une instance non tabulaire, le noyau de l'agent GAM empêchera bien cette création illicite, mais la fonction writeInstSock n'aura pas renvoyé d'erreur.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_SOCK si une erreur se produit dans l'écriture du message sur le socket.

7)
Nom : PRINTMess
Type retourné : void

Paramètres : const GAMMessType messPipe;

Description : Affiche sur la sortie standard les champs du message messPipe.
Cette fonction peut être utilisée dans la méthode afin de tracer les messages échangés avec le noyau de l'agent GAM.

Code retour : Aucun.

2.3 Manipulation de sockets.

1)
Nom : receiveSock
Type retourné : GAMResultType

Paramètres : GAMMessType *messSock;
const int sock ;

Description : lit un message *messSock dans la socket sock.
Cette fonction peut être utilisée par la méthode pour lire les requêtes SNMP que lui transmet le noyau de l'agent GAM.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_SOCK pour une erreur sur la socket et GAM_ERR_UNIX pour une erreur système.

2)
Nom : writeRespSock
Type retourné : GAMResultType

Paramètres : GAMMessType messSock;
IntfType sock ;

Description : Répond à une requête SNMP en mode socket.
Les champs operation et inst doivent être identiques à ceux transmis par le noyau dans la requête SNMP.
Ce message de requête aura pu être lu par la méthode grâce à la fonction receiveSock.
La valeur de l'instance doit être placée dans messSock.vall
On doit préciser le descripteur de la socket sock.
Cette socket aura pu être ouverte et connectée dans la méthode grâce à la fonction connectSock.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_SOCK pour une erreur sur la socket et GAM_ERR_UNIX pour une erreur système.

2.4 Opérations diverses.

1)
Nom : buildName
Type retourné : GAMResultType

Paramètres : const char *inst ;
char *attName ;
OidPtrType name ;
OidLenghPtrType namelenp ;

Description : construit à partir du nom complet d'une instance avec son suffixe (contenu dans inst) le nom de l'attribut (sans suffixe) dans attName, un tableau name contenant *namelenp éléments représentatnt les subids successifs du suffixe.
Exemple : inst = « trucmuche.67.567 » => attName= »trucmuche », name[0]=67, name[1]=567 et *namelenp=2.

Code retour : GAM_NO_ERROR en cas de réussite, GAM_ERR_SOCK pour une erreur sur la socket et GAM_ERR_UNIX pour une erreur système.

2)
Nom : buildByName
Type retourné : GAMResultType

Paramètres : char *inst ;
const char *attName ;
const OidPtrType name ;
const OidLenghType namelen ;

Description : construit le nom complet d'une instance avec son suffixe (inst) à partir du nom de l'attribut sans suffixe (attName) et d'un tableau (name)de namelen éléments contenant les subids du suffixe (opération inverse de la pécédente).
Exemple : inst = « trucmuche.67.567 » => attName= »trucmuche », name[0]=67, name[1]=567 et *namelenp=2.

Code retour : GAM_NO_ERROR en cas de réussite et GAM_ERR_UNIX pour une erreur système.

3)
Nom : printGamError
Type retourné : void

Paramètres : GamResultType ;

Description : imprime sur la sortie d'erreur standard un message d'erreur associé au code d'erreur passé dans le paramètre error

Code retour : Aucun, si le code d'erreur ne correspond à aucune erreur GAM connue, le message Undefined error est affiché.

```
#ifndef _GAMTYPES_H_
#define _GAMTYPES_H_

typesdef CIntsType GAMItemType ;
typesdef CIntsType GAMIMessOpType ;
```

VI. REFERENCES BIBLIOGRAPGIQUES.

[1] Stalling William, "SNMP, SNMPv2 and CMIP : The practical guide to NetWork-Management Standards"
In Addison-Wesley Publishing Company, 1993.

[2] Cornafion, "Systèmes Informatiques Répartis : concepts et techniques"
In Dunod Informatique,1981.

[3] Sloman Morris and Kramer Jeff, "Distributed Systems and Computer Networks"
In C.A.R. Hoare, series editor, 1993.

[4] RFC 1155, "Structure and Identification of Management Information for TCP/IP – Based Internets"
M. Rose, K. McCloghrie, Hughes LAN Systems
Mai 1990
http://www.ietf.org/rfc/rfc1155.txt

[5] RFC 1156, " Management Information Base for Network Management of TCP/IP-based internets"
K. McCloghrie, Hughes LAN Systems and M. Rose
Mai 1990
http://www.ietf.org/rfc/rfc1156.txt

[6] RFC 1157, " A Simple Network Management Protocol (SNMP)"
J. Case, M. Fedor,M. Schoffstall and J. Davin
Mai 1990

[7] Sedrati Gerald, "Documentation de GAM–OAT : version 3.0"
Bull*Soft*
Février 19998

[8] Moreno Jean Michel, "UNIX Administration 2ème édition"
in Ediscience International 1998

[9] Tanenbaum Andrew, "Réseaux"
in InterEditions et Prentice Hall, 1997.