La gestion des réseaux IP basée sur les technologies Web et le modèle *push*

Jean-Philippe Martin-Flatin
École Polytechnique Fédérale de Lausanne (EPFL)
Institut pour les Communications informatiques et leurs Applications (ICA)
1015 Lausanne, Suisse
Email: martin-flatin@epfl.ch Web: http://icawww.epfl.ch

Résumé

La gestion des réseaux IP est actuellement fondée sur le protocole SNMP, ainsi que sur des plateformes de gestion coûteuses implémentant le paradigme gestionnaire-agent du cadre de gestion SNMP. Cette gestion repose sur deux types de transfert de données: un protocole de type requête/réponse, pour recueillir des données statistiques et pour surveiller le réseau, et du *push* non sollicité, pour envoyer les notifications. Ce modèle conceptuel souffre de plusieurs carences, notamment en ce qui concerne le temps de mise sur marché du logiciel de gestion propriétaire, la gestion de versions multiples, l'efficacité du codage de l'information transportée par le protocole, etc. Dans ce papier, nous présentons une nouvelle approche pour la gestion des réseaux IP, basée sur les technologies Web et le modèle *push*. Ce modèle suit le paradigme publication-souscription, et utilise la même méthode de transfert quel que soit le type des données de gestion. Nous décrivons comment l'implémenter avec des technologies Web pour (i) envoyer des notifications, (ii) distribuer les données de gestion relatives à la surveillance du réseau et à l'accumulation de données statistiques, et (iii) mettre à jour les interfaces graphiques servant au *monitoring*.

Mots-clés: Gestion de réseaux, Réseaux IP, Web, Push, Java, HTTP, RMI.

1. Introduction

La plupart des réseaux IP sont actuellement gérés avec des plateformes de gestion dédiées telles que HP OpenView, Cabletron Spectrum, IBM Netview ou Sun Solstice. Dans une récente publication [5], nous avons analysé les problèmes liés à cette approche, et avons mis en évidence les avantages de la gestion des réseaux IP basée sur les technologies Web par rapport à l'approche classique fondée sur SNMP. Nous avons proposé de répartir les principales tâches de gestion assurées par ces plateformes en deux groupes, en fonction du paradigme utilisé pour transférer des données de gestion entre l'agent et le gestionnaire ; le premier groupe suit le modèle *pull*, le second le modèle *push*. Nous avons montré que ces deux modèles pouvaient rapidement être implémentés et déployés dans la plupart des équipements actuellement disponibles sur le marché.

Le modèle *pull* est fondé sur le paradigme requête-réponse. C'est une généralisation du *polling* réalisé par SNMP. À chaque cycle de collecte des données, le gestionnaire (c'est-à-dire le client SNMP) interroge les agents pour connaître la valeur de plusieurs variables de MIB (on a un serveur SNMP par agent). Chaque agent répond alors séparément à chaque requête du gestionnaire. Lors du transfert des données de gestion, tout se déroule donc comme si le gestionnaire "tirait" à lui les données stockées dans les agents. Dans ce modèle, le transfert de données est toujours initié par le gestionnaire.

Inversement, le modèle *push* est basé sur le paradigme publication-souscription. Seules les notifications suivent ce modèle dans le cadre de gestion standard SNMP. Les applications de gestion de réseaux conçues à partir de ce modèle passent par trois étapes:

- *publication*: chaque agent annonce les MIB qu'il gère et les notifications qu'il est susceptible d'envoyer ;
- *souscription*: agent par agent, l'administrateur (la personne[1]) abonne le gestionnaire (le programme) à différentes variables de MIB et notifications ; pour les variables de MIB, la fréquence à laquelle leurs valeurs doivent être envoyées au gestionnaire est également indiquée ;

1. Pour la terminologie, voir [6,10].

- *distribution*: chaque agent gère indépendamment ses cycles d'information du gestionnaire ; les données sont donc "poussées" à l'initiative de l'agent ; dans le cas du *monitoring* et de la collecte de données à des fins d'analyse statistique, cette opération s'effectue à intervalles réguliers, grâce à un ordonnanceur ; dans le cas des notifications, elle s'effectue de façon asynchrone.

Nous avons également montré que le *monitoring* et la collecte de données peuvent aussi bien être réalisés avec des technologies *pull* que *push* [5]. L'avantage du *push* en gestion régulière est qu'il génère moins de trafic sur le réseau, et qu'il diminue la charge CPU du gestionnaire en déléguant une partie du travail aux agents (cf. la gestion par délégation de Goldszmidt [3] ou le "mythe des agents idiots" de Wellens and Auerbach [12]). Le *pull*, lui, reste plus indiqué pour la gestion *ad hoc* [5].

Dans cet article, nous présentons en détail comment utiliser le modèle *push* pour gérer des réseaux IP avec les technologies Web. Nous démontrons notamment que les agents peuvent employer les mêmes techniques de communication pour envoyer des données à intervalles réguliers (via l'ordonnanceur) ou irréguliers (notifications) au gestionnaire. Les techniques considérées sont HTTP, les socquettes (*sockets*) et Java RMI. Nous expliquons à quel point le traitement des notifications, le *monitoring* et la génération de rapports d'utilisation du réseau sont découplés, et montrons que l'application de gestion de réseaux peut être répartie entre plusieurs machines.

Les contraintes que nous tentons de satisfaire sont les suivantes:

- proposer des solutions simples pouvant être développées et déployées en moins d'un an ;
- résoudre les principaux problèmes rencontrés avec SNMP [5] ;
- respecter le modèle de sécurité des applettes Java défini dans le JDK 1.1.6 ;
- rendre la traversée des coupe-feu aisée ; le problème classique est celui où le gestionnaire est situé à l'intérieur d'un intranet sécurisé, alors que certains des agents qu'il gère se trouvent dans des sites distants et sont accédés via un réseau étendu non sécurisé comme l'Internet ;
- proposer au moins une solution qui ne requiert pas une JVM complète et le JDK complet (avec RMI, Java IDL, etc.), au cas où certains agents seraient embarqués et ne pourraient offrir qu'une JVM restreinte (cf. par exemple les JVM légères proposées dans l'environnement EmbeddedJava).

La suite de cet article est structurée en cinq chapitres. Dans le ch. 2, nous décrivons les phases de publication et de souscription du modèle *push*. Dans le ch. 3, nous présentons la phase de distribution. Dans le ch. 4, nous étudions différentes techniques de communication entre agent et gestionnaire. Enfin, nous concluons dans le ch. 5 en présentant des perspectives d'avenir.

2. Phases de publication et de souscription

Dans la phase de publication, chaque équipement annonce les MIB qu'il gère (MIB génériques comme la MIB-II ou la MIB RMON, ou MIB propriétaires telle la MIB Cisco), ainsi que les notifications qu'il peut envoyer au gestionnaire (par exemple, "température de la carte-mère trop élevée" ou "interface xxx hors service"). Pour réaliser cela facilement, nous proposons de stocker deux pages HTML dans chaque agent. Les URL correspondantes doivent être standard, de façon à simplifier la tâche des administrateurs et à faciliter l'automatisation partielle de la phase de souscription. La première URL affiche toutes les applettes de gestion stockées dans l'agent:

<`http://agent.domain/mgmt/mibs.html`>

où `agent.domain` est le nom DNS complet (*fully qualified domain name*) de l'agent. Ceci nécessite que le répertoire `mgmt` soit réservé sur tous les agents à des fins de gestion (gestion de réseaux, gestion de systèmes...). Une fois cette page chargée par l'administrateur dans son butineur Web, il lui suffit de cliquer sur l'icône ou le nom d'une MIB pour télécharger l'applette de souscription correspondante. À partir de là, le nom des URL n'est plus imposé: libre à chaque vendeur de choisir ce qu'il veut.

La seconde URL prédéfinie permet de souscrire aux notifications:

<http://agent.domain/mgmt/notifications.html>

La page HTML correspondante est propriétaire. Elle peut être unique pour un vendeur donné, quel que soit l'équipement, ou elle peut être spécifique à un équipement donné. Par exemple, les notifications générées par des routeurs IP ou des commutateurs ATM d'une même marque seront vraisemblablement différentes, alors que celles générées par un routeur haut de gamme et un routeur bas de gamme seront probablement identiques. Cette page HTML peut contenir une applette offrant une interface graphique sophistiquée, ou elle peut se limiter à un formulaire avec des boutons-poussoirs et des cases à cocher.

Comme on peut le voir sur la Fig. 1, le mode opératoire pour la première phase est de (i) télécharger l'applette représentant la vue d'ensemble du réseau dans le butineur Web d'une machine quelconque, (ii) sélectionner un agent sur le plan du réseau, et (iii) télécharger depuis cet agent les deux pages HTML décrites ci-dessus ; ces dernières sont stockées en EPROM dans l'agent.

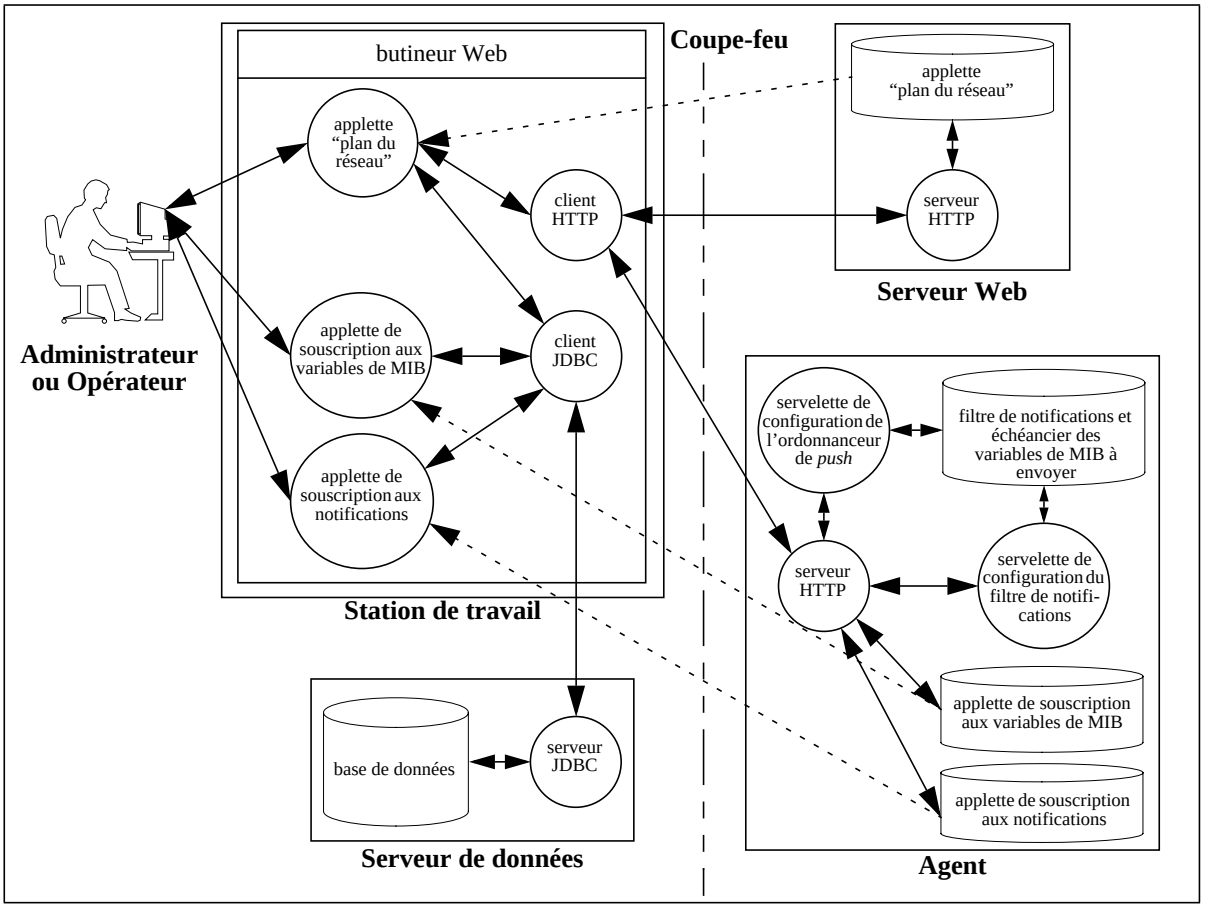

Fig. 1. Phases de publication et de souscription

Dans le seconde phase, l'administrateur abonne le gestionnaire à des variables de MIB et des notifications. Les applettes de souscription aux variables de MIB permettent de sélectionner les variables qui nous intéressent et de spécifier la fréquence à laquelle elles doivent être envoyées. Cette fréquence de *push* peut être propre à chaque variable ; elle peut aussi être la même pour toutes les variables d'une MIB donnée, ou pour toutes les variables de toutes les MIB d'un agent. Cette politique est définie par le vendeur, et constitue un critère de différenciation de l'offre entre vendeurs. La granularité plus ou moins fine de la fréquence de *push* permet aussi aux vendeurs de mettre à jour le logiciel de gestion de leurs équipements en plusieurs étapes.

L'applette de souscription aux notifications, pour sa part, n'a pas besoin de gérer de fréquences, puisque les notifications sont par essence asynchrones et imprévisibles. En fait, cette applette ne sert qu'à définir un filtre.

Les notifications auxquelles l'administrateur n'a pas souscrit sont purement et simplement jetées par l'agent, sans laisser de trace.

On pourrait imaginer d'avoir une seule applette de souscription pour toutes les MIB et notifications d'un agent donné. Les MIB pourrait alors être dépliées comme dans le butineur de MIB (*MIB browser*) d'une plateforme de gestion SNMP classique. Mais de même que les gens préfèrent souvent les interfaces graphiques évoluées de CiscoWorks et consorts à des interfaces génériques souvent plus austères [6], de même on peut s'attendre à ce que des applettes spécifiques se généralisent à travers le marché, MIB par MIB.

Il est à noter que la phase de souscription décrite ci-dessus est entièrement réalisée manuellement. Si un agent venait à perdre sa configuration, par exemple parce qu'il stocke celle-ci en mémoire volatile, il serait fastidieux pour l'administrateur d'avoir à redéfinir entièrement cette configuration à la main. C'est pourquoi il est important de garder une copie de la configuration de chaque agent de façon persistante. Comme il apparaît sur la Fig. 1, nous proposons d'utiliser pour cela JDBC et une base de données relationnelle. Si le nombre d'agents à gérer est limité, on peut aussi travailler avec des fichiers textuels. Dans un cas comme dans l'autre, la configuration de l'agent peut être téléchargée automatiquement sans intervention humaine, par exemple quand l'agent se réinitialise (*reboot*).

La base de données, qui peut être distribuée, doit assurer la persistance de trois types de données [6]:

- l'échéancier des variables de MIB à envoyer, agent par agent ;
- le filtre de notifications, agent par agent ;
- la topologie du réseau utilisée par l'applette "plan du réseau" pour construire son interface graphique.

3. Phase de distribution

Dans la phase de distribution, le transfert des données pour assurer le *monitoring* et la collecte de données diffère peu de l'envoi des notifications. Les problèmes de communication sont identiques ; seules les servelettes Java tournant sur le gestionnaire et l'agent diffèrent.

Penchons-nous tout d'abord sur l'envoi des notifications et le traitement des évènements. Ces tâches sont décrites Fig. 2. Sur chaque agent, le contrôleur de santé vérifie en permanence que tout va bien sur cet équipement: les interfaces Ethernet détectent bien une porteuse, la ventilation de l'alimentation fonctionne comme prévu, etc. Quand une situation anormale est détectée, le contrôleur de santé contacte le générateur de notifications, qui traduit une structure de données propriétaire en mémoire en une notification au format standard SNMP. Cette dernière est envoyée au gestionnaire par l'expéditeur de notifications. À son arrivée au gestionnaire, la notification est reçue par le receveur de notifications. Celui-ci est un objet d'une servelette Java tournant sur le gestionnaire. De là, la notification est transmise au filtre de protection. Le rôle de ce filtre est de protéger l'application de gestion contre des agents la "bombardant" de notifications, soit parce qu'ils sont mal configurés, soit parce qu'ils sont bogués, soit parce qu'ils ont été reprogrammés par des *hackers* à des fins de déni de service. Au delà d'un certain seuil de trafic par unité de temps, le filtre de protection désactive l'agent en jetant automatiquement tous les paquets en provenance de cet agent. Les agents ne peuvent être réactivés que par l'administrateur ; celui-ci est prévenu dès que la désactivation a eu lieu.

S'il n'y a pas de problème, le filtre de protection envoie les notifications au corrélateur d'évènements. Comme dans les plateformes de gestion classiques, ce dernier est la pierre angulaire de tout le *monitoring*. Les notifications ne sont pas le seul type d'évènements reçus par le corrélateur. Celui-ci reçoit également des évènements en provenance de l'interpréteur de données, les recoupe avec les notifications, et jette les évènements masqués par la topologie du réseau (si un routeur tombe en panne, toutes les machines situées derrière lui apparaissent hors service, mais seul le problème du routeur est gardé). Lorsque le corrélateur d'évènements n'a plus que des évènements irréductibles, il invoque pour chacun d'entre eux un traiteur d'évènement (*event handler*) défini par le degré d'urgence de cet évènement. Par exemple, il déclenche une sirène d'alarme pour un évènement critique, un bip pour un évènement important, ou un *email* pour un évènement informationnel.

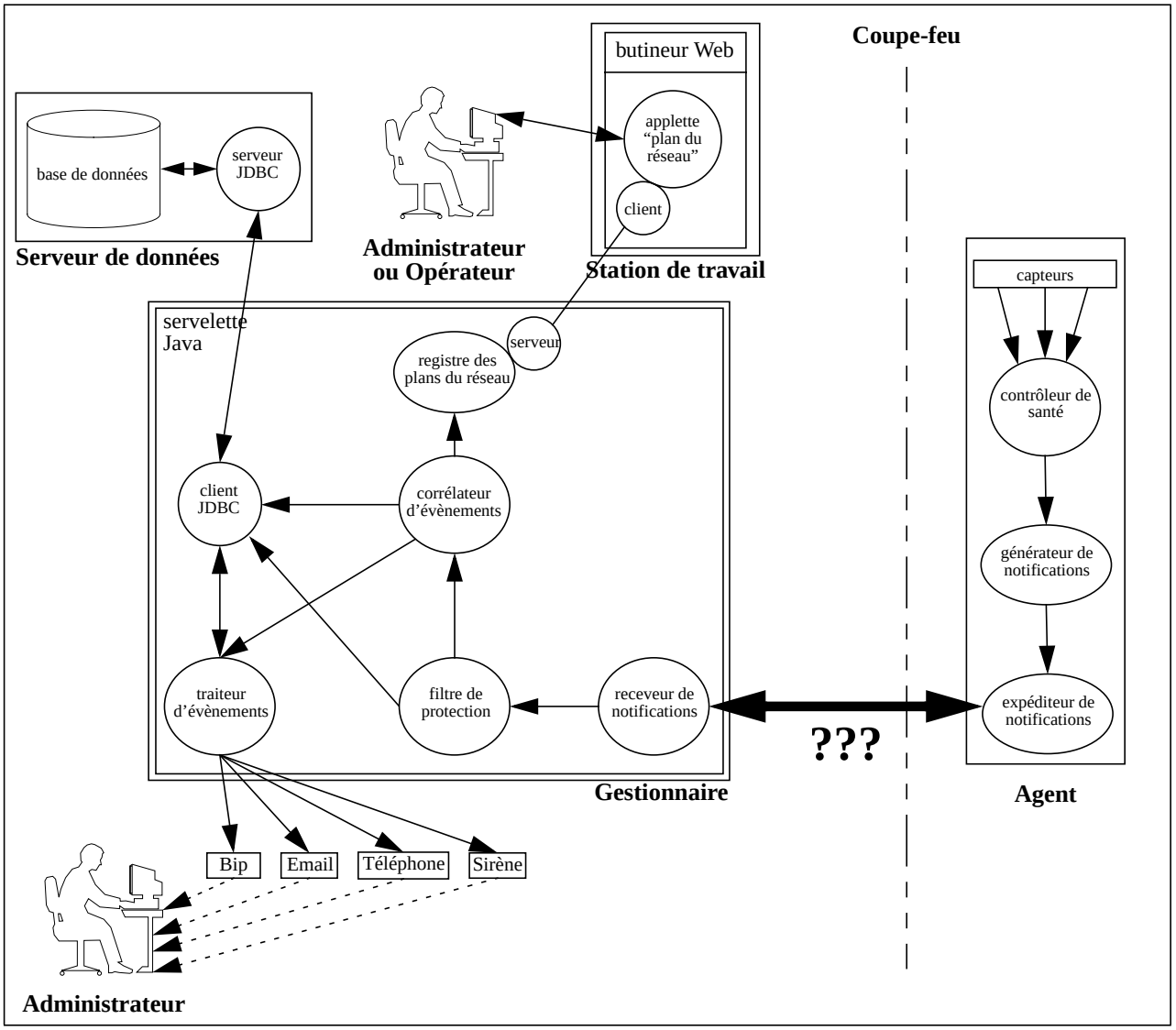

Fig. 2. Phase de distribution: notifications

Si un administrateur ou un opérateur veut suivre en temps réel les problèmes du réseau, il lui faut enregistrer l'applette "plan du réseau" tournant dans son butineur auprès du gestionnaire (registre des plans du réseau). Le cas échéant, le corrélateur d'évènements met alors à jour tous les plans du réseau enregistrés. Les icônes représentant les agents deviennent rouges quand un problème est détecté, vertes lorsque le problème disparaît. Pour éviter des phénomènes d'oscillations, il est recommandé de définir une couleur intermédiaire, par exemple orange, pour indiquer qu'une machine est dans une phase transitoire incertaine.

Certains évènements peuvent être simplement enregistrés dans la base de données, sans qu'aucune mesure ne soit prise dans l'immédiat. Plus tard, par une analyse statistique automatisée, on détermine si cet évènement se produit souvent ou rarement ; dans le premier cas, l'information est remontée à l'administrateur, par exemple par *email* ; dans le second, tous les enregistrements sont jetés de façon transparente pour l'administrateur.

La base de données des Fig. 2 et 3 comporte maintenant sept types de données:

- l'échéancier des données à envoyer, agent par agent ;
- le filtre de notifications, agent par agent ;
- la topologie du réseau utilisée par l'applette "plan du réseau" pour construire son interface graphique ;
- la définition des traiteurs d'évènements ;
- l'enregistrement de certains évènements ;

- les données récoltées à des fins d'analyse statistique ;
- une trace (*log*) de l'appel des traiteurs d'évènements.

Nous venons de présenter en détail la gestion des évènements. Si l'on se penche maintenant sur le *monitoring* et la collecte de données, la principale différence réside dans la servelette Java qui tourne sur le gestionnaire. La servelette côté agent est également différente au niveau du code, mais au niveau logique, elle a la même structure que précédemment: le générateur de notifications correspond au préparateur de données, et l'expéditeur de notifications correspond à l'expéditeur de données. La communication entre l'expéditeur et le receveur est identique. Cette similarité est très différente de ce que l'on rencontre dans le cadre classique SNMP, où la communication est totalement différente entre une opération `get-next` et une opération `snmpv2-trap` [10].

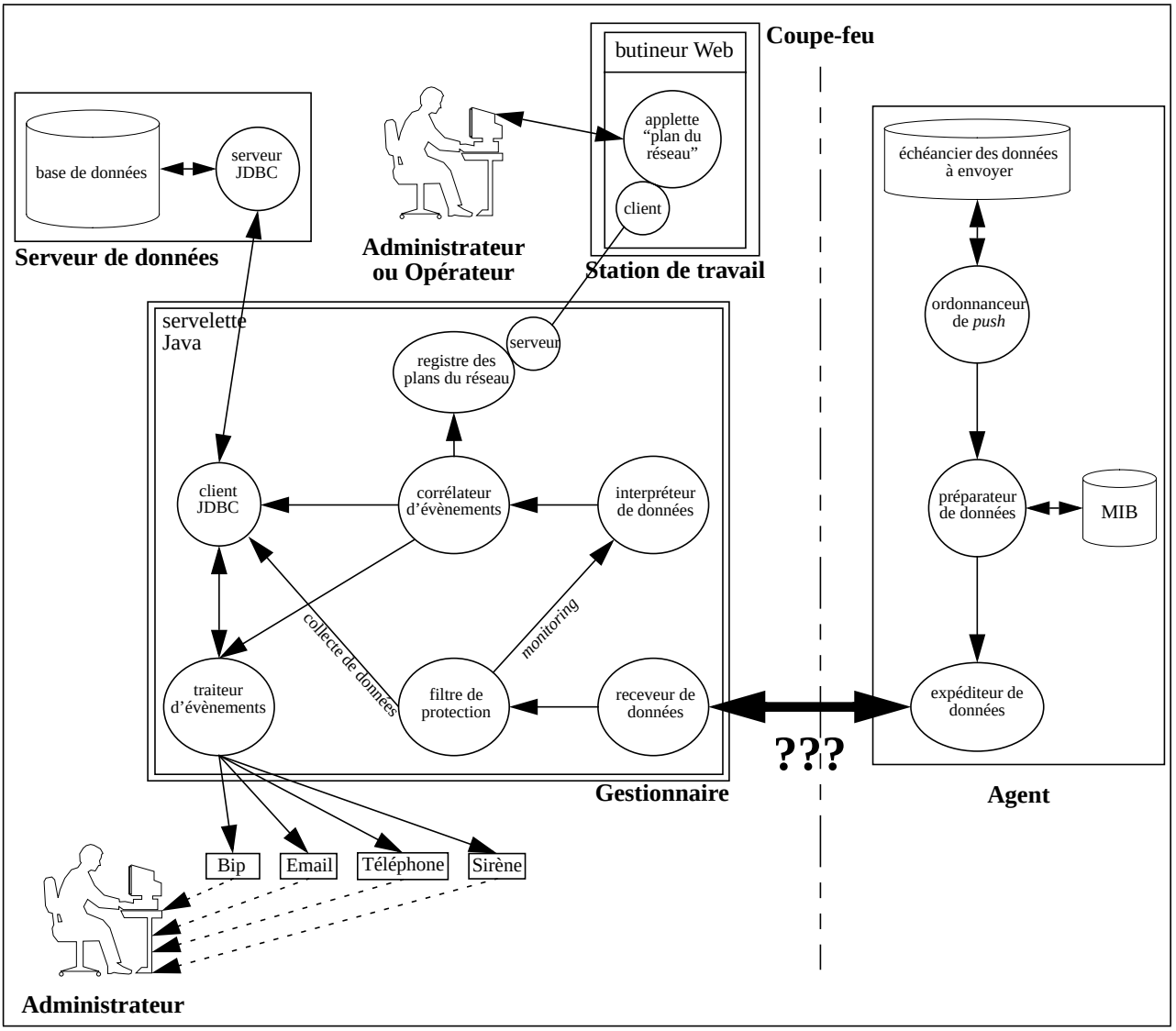

Fig. 3. Phase de distribution: *monitoring* et collecte de données

Pour le *monitoring* comme pour la collecte de données, les données de gestion arrivent au niveau du receveur de données côté gestionnaire. Comme précédemment, ces données transitent via un filtre de protection pour la même raison. Pour la collecte de données, les données ne sont pas traitées: elles sont directement stockées dans une base de données relationnelle via JDBC. Leur traitement statistique se fait de façon différée, grâce à une servelette non représentée ici. En revanche, les données récoltées pour le *monitoring* sont aussitôt envoyées à l'interpréteur de données. Celui-ci s'appuie sur sa connaissance de la topologie du réseau et sur une base de

règles écrites par l'administrateur pour déterminer s'il y a un problème dans le réseau. Lorsqu'un problème est détecté, un évènement est généré et envoyé au corrélateur d'évènements, où il est traité comme décrit précédemment.

4. Communication entre agent et gestionnaire

Maintenant que nous avons montré que les aspects de communication étaient similaires pour les trois tâches principales réalisées par une plateforme de gestion de réseaux IP, penchons-nous plus en détail sur cette communication. Comment est-elle réalisée ? Que ce soit pour l'envoi des notifications, pour le *monitoring* ou pour la collecte de données, on a un expéditeur côté agent et un receveur côté gestionnaire. Mais quelle technologie peut-on utiliser entre les deux ? Par rapport au cadre de gestion classique SNMP, les rôles du client et du serveur sont inversés dans le modèle *push*: le transfert de données est maintenant initié par l'agent et non par le gestionnaire. Hors ceci pose des problèmes de sécurité s'il faut traverser un coupe-feu entre l'agent et le gestionnaire. En effet, il est nettement préférable qu'un transfert soit initié par le gestionnaire, connecté à un intranet sécurisé, et en qui l'administrateur a confiance car il le contrôle localement, plutôt que par un agent distant qui a pu être compromis par un *hacker*, et sur lequel l'administrateur n'a aucun contrôle direct. On rend ainsi l'usurpation d'identité (*impersonation*) bien moins aisée.

Pour résoudre ce problème, nous proposons d'utiliser des connexions TCP persistantes. Ceci permet de découpler l'initiative de la création de la socquette de l'initiative du transfert de données via cette socquette. Ainsi, c'est toujours le gestionnaire qui crée la socquette, ce qui est préférable en termes de sécurité, alors que toutes les données transitent par la suite de l'agent vers le gestionnaire, sous le contrôle de l'agent. Un autre avantage des connexions persistantes est de diminuer l'*overhead* réseau et CPU de la création puis de la suppression de multiples socquettes TCP [5].

Un argument souvent opposé à l'usage de connexions persistantes est celui de leur mauvaise *scalability*. Pour chaque connexion persistante, le gestionnaire doit en effet bloquer une certaine quantité de mémoire pour les structures de données utilisées par le noyau pour gérer la connexion TCP. Lorsque le nombre d'agents gérés par un même gestionnaire devient élevé, on a tôt fait de saturer sa mémoire. Cette critique était recevable lorsque SNMP a été conçu, à la fin des années 1980: la mémoire des équipements réseaux et des systèmes était alors rare et chère. Aujourd'hui, ce raisonnement ne tient plus: les gestionnaires disposent typiquement d'au moins 64 Mo de mémoire, ce qui est largement suffisant pour garder en permanence plusieurs milliers de connexions TCP persistantes. En termes de *scalability*, ce n'est pas la persistance des connexions qui est aujourd'hui critique, mais plutôt des facteurs comme le temps d'accès à la base de données, le temps de calcul de corrélation des évènements, ou le goulet d'étranglement que constitue le segment de réseau auquel le gestionnaire est connecté. Ce sont ces facteurs qui déterminent la règle empirique actuelle selon laquelle un gestionnaire ne doit pas gérer plus de quelques centaines d'agents. Au delà, il faut distribuer la gestion entre plusieurs gestionnaires.

Les technologies Web nous offrent trois options pour assurer une connexion persistante entre un agent et un gestionnaire: les socquettes, Java RMI et HTTP [9]. Nous allons maintenant les étudier tour à tour, en payant une attention toute particulière aux problèmes liés à la traversée d'un coupe-feu.

4.1. Socquettes

Les socquettes présentent une propriété intéressante de bidirectionalité. Lorsqu'une socquette est créée, le côté client de la socquette contacte le coté serveur, comme il est d'usage dans le modèle client-serveur. Mais une fois que la socquette est établie, qu'elle soit de type TCP ou UDP, le client peut envoyer des données au serveur via celle-ci, mais le serveur peut également et indépendamment envoyer des données au client via cette même socquette. Cette propriété permet de résoudre notre paradoxe client-serveur apparent: le gestionnaire crée une socquette vers chaque agent qu'il gère, puis chaque agent envoie ultérieurement des données à travers ce tuyau virtuel. Ces données peuvent être des variables de MIB ou des notifications: tout transite via le même tuyau virtuel. Pour garantir la persistance de toutes ces connexions, le receveur, côté gestionnaire, doit déclarer une

durée de vie infinie pour chaque socquette qu'il crée. Si la connexion TCP est rompue pour une raison quelconque, le gestionnaire doit le détecter immédiatement, puis se reconnecter à l'agent en créant une nouvelle socquette.

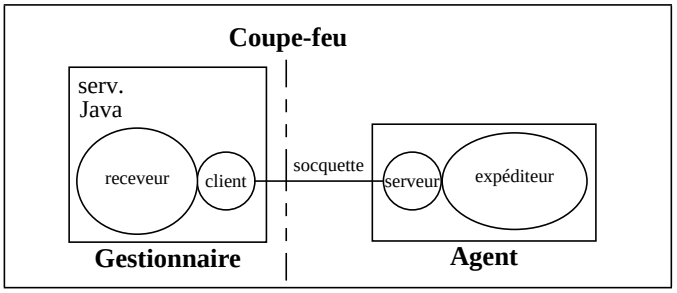

Fig. 4. Distribution via socquettes

Cette solution présente un grand avantage: sa simplicité. La programmation des socquettes est très facile, tout particulièrement en Java. Malheureusement, elle présente aussi plusieurs inconvénients, dont nous allons voir les trois principaux.

Tout d'abord, si une connexion persistante est rompue, le gestionnaire ne le détecte pas automatiquement, à moins qu'il envoie régulièrement des *keepalives*. L'envoi des notifications, qui peuvent véhiculer des messages urgents, dépend par conséquent d'une connexion versatile, ce qui pose un problème conceptuel.

Un second problème peut survenir lorsque le nombre d'agents à gérer est élevé. Le nombre de socquettes TCP étant élevé côté gestionnaire, le nombre de descripteurs TCP nécessaires en mémoire peut excéder le nombre maximum accepté par le système d'exploitation. Sur de nombreux systèmes Unix, ce problème peut être résolu en modifiant un paramètre de configuration du noyau et en recompilant celui-ci. Loin d'être irréaliste, cette solution est appliquée de façon assez systématique aux serveurs Web très utilisés. Les vendeurs ont d'ailleurs souvent simplifié la tâche des administrateurs ces dernières années: sur SGI, par exemple, la recompilation du noyau est automatique lorsqu'on réinitialise une machine, dès lors que le fichier de configuration du noyau a été édité.

Le troisième problème est lié aux coupe-feu. La plupart des coupe-feu sont configurés pour ne pas laisser passer de trafic UDP autre que DNS (et parfois aussi NNTP), et pour ne laisser passer que quelques ports TCP [1]. Que l'on utilise des socquettes TCP ou UDP, les coupe-feu ne laisseront donc pas passer notre trafic de gestion par défaut: il faudra explicitement modifier la configuration du coupe-feu. Ceci ne pose pas de problème aux grandes entreprises, qui ont les moyens financiers de s'offrir l'expertise technique requise, soit en interne, soit en externe. Il en va tout autrement pour les PME qui, en général, n'ont pas cette expertise en interne, et pour qui la moindre modification du coupe-feu nécessite le recours à des consultants externes très coûteux – une opération à éviter autant que faire ce peut.

4.2. Java RMI

Tout comme les socquettes, Java RMI offre une association bidirectionnelle: une fois que le client RMI s'est connecté au serveur RMI, chacun d'entre eux peut envoyer des données à l'autre. RMI est une solution élégante au niveau conceptuel, car elle offre une vision entièrement orientée objets de la gestion de réseaux [7]. Elle offre une sémantique plus riche au concepteur de l'application de gestion distribuée, qui n'est plus cantonné à des variables de MIB de bas niveau et à une poignée d'opérations SNMP également de bas niveau. Des structures de données et des méthodes plus élaborées peuvent être définies, mieux adaptées à la conception d'applications de gestion de réseaux de plus en plus complexes.

Avec RMI, le traitement des notifications et des variables de MIB est légèrement différent côté agent, car les processus qui tournent sur l'agent n'ont pas le même temps de réaction. Pour les notifications, le *push* est déclenché par le contrôleur de santé, qui est propriétaire, existe déjà dans la plupart des équipements, et n'a généralement pas été écrit en Java. Pour les variables de MIB, le *push* est déclenché par l'ordonnanceur. Ce

dernier n'existe pas aujourd'hui dans les équipements, et il n'est pas gouverné par le même degré d'urgence que le contrôleur de santé ; il peut donc être écrit en Java. Dans le cas des notifications, une partie seulement du traitement est effectué par l'application Java au sein de laquelle tourne le serveur RMI (cf. Fig. 5) ; pour les variables de MIB, c'est l'intégralité (cf. Fig. 6). À part ça, dans les deux cas, le transfert de données entre l'agent et le gestionnaire est identique, et se fait du serveur RMI vers le client RMI.

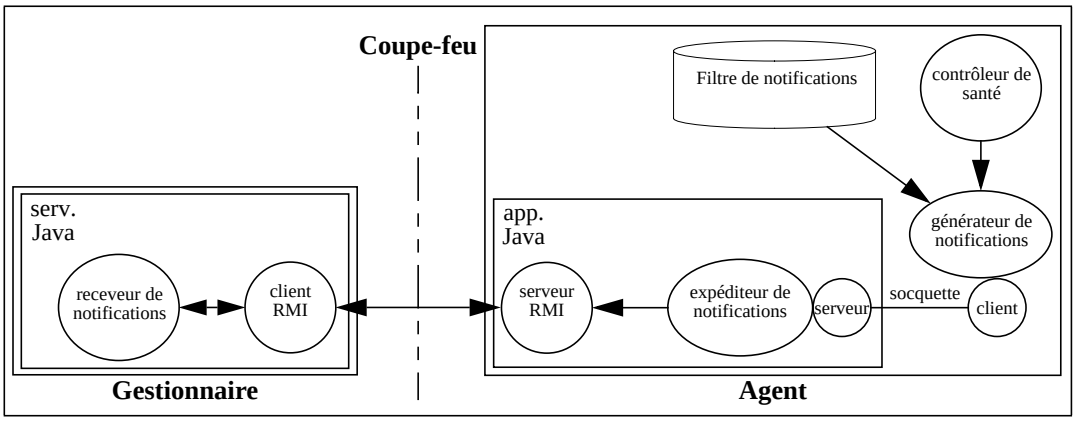

Fig. 5. Distribution via RMI: notifications

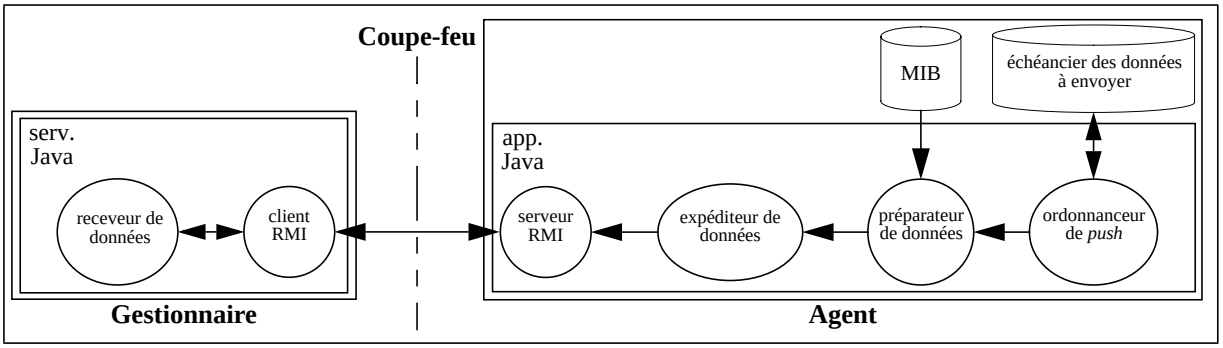

Fig. 6. Distribution via RMI: *monitoring* et collecte de données

Cette solution, pour élégante qu'elle puisse paraître, n'en présente pas moins un inconvénient majeur: RMI. Si la disponibilité de RMI ne pose aucun problème côté gestionnaire, il en va tout autrement pour les agents. Le monde Internet est dominé par des agents relativement bon marché. Pour certains équipements peu coûteux comme les contrôleurs d'imprimantes ou les routeurs d'accès de bas de gamme, la guerre des prix est telle que les marges sont très serrées, et chaque dollar qui peut être économisé doit l'être. Du coup, les ressources en CPU, mémoire vive et mémoire morte sont rognées autant que possible, et limitées au strict minimum. Sur de tels équipements, le support d'une JVM légère est déjà un problème en soi ; *a fortiori,* le support d'une JVM complète et d'un JDK complet sont hors du champ des possibles. Un problème similaire se pose pour les équipements embarqués peu coûteux, tels les téléphones mobiles ou ordinateurs portables de bas de gamme. Sun travaille actuellement sur ce problème avec l'architecture EmbeddedJava, qui permettra à terme de ne mettre dans ces équipements que le strict nécessaire pour le JDK et la JVM. Il est aussi envisagé de télécharger des extensions à la demande, ce qui nécessite la présence de ressources non utilisées dans les équipements embarqués – une hypothèse dont le réalisme reste à prouver...

D'autres problèmes rendent cette solution problématique aujourd'hui. Tout d'abord, le "bureau d'enregistrement" (*registry*) de RMI est nécessairement une application Java avec le JDK 1.1.6 [9]. Cette application est très gourmande en termes de ressources CPU et mémoire. Ensuite, la vitesse d'exécution du code RMI est lente, bien plus que celle du code Java classique qui n'est déjà pas rapide. Enfin, RMI s'appuyant en fait sur la technologie des socquettes [9], on a le même problème que précédemment via-à-vis des coupe-feu. En fait le problème est pire, car on ne contrôle plus les numéros de ports alloués côté client. En effet, les socquettes RMI

sont transparentes pour l'application Java: si, côté serveur, on est sûr que RMI utilisera toujours le port numéro 1099 [4], côté client, en revanche, RMI peut utiliser n'importe quel numéro de port ; ce dernier n'est plus choisi par l'administrateur, comme dans la solution précédente, ce qui rend la configuration du coupe-feu encore plus délicate... et coûteuse !

Pour toutes ces raisons, nous pensons que RMI n'est pas une solution adéquate actuellement, bien qu'elle présente un potentiel intéressant au niveau conceptuel.

4.3. HTTP

HTTP n'offre pas la propriété de bidirectionalité que nous avons exploitée avec les socquettes et Java RMI. Cette fois, les connexions sont orientées: il n'est plus possible de créer une connexion persistante dans un sens, puis d'envoyer des données dans l'autre. Pour qu'un serveur HTTP puisse envoyer un message à un client HTTP, il faut qu'il ait reçu au préalable une requête de ce client. Un serveur HTTP ne peut donc pas spontanément envoyer des messages non sollicités à un client.

En cela, SNMP et HTTP se comportent différemment. SNMP implémente un modèle de communication client-serveur généralisé, où la requête du client peut être explicite (modèle *pull* des opérations `get`, `set`...) ou implicite (modèle *push* de l'opération `snmpv2-trap`). HTTP, quant à lui, implémente un modèle client-serveur strict: toutes ses méthodes suivent le modèle requête-réponse ; la requête du client ne peut pas être implicite. Avec HTTP, on ne peut donc pas réaliser du *push* naturellement.

Comment résoudre ce problème ? Peut-on envoyer une seule requête côté client HTTP, puis envoyer un nombre infini de réponses à cette requête coté serveur ? L'astuce consiste à envoyer une réponse de longueur infinie côté serveur, et à inclure des séparateurs dans la charge utile (*payload*) des messages HTTP. Netscape s'est déjà penché sur cette question dans le contexte du Web, pour mettre à jour l'interface graphique d'une applette dans un butineur Web [8]. La solution préconisée était d'utiliser le type `multipart` de MIME [2]. Nous proposons d'utiliser cette même solution en gestion de réseaux. À chaque cycle de *push*, un agent envoie une partie MIME comportant le descriptif et la valeur de toutes les variables de MIB spécifiées par l'ordonnanceur. Le séparateur MIME sert alors de séparateur entre deux cycles de *push* consécutifs ; c'est une méta-donnée signifiant "fin de cycle de *push*". Quant aux notifications, elles sont envoyées une à la fois, de manière asynchrone. Dans ce cas, le séparateur MIME est interprété par l'agent comme une méta-donnée signifiant "fin de notification".

Cette solution présente deux gros avantages: elle est simple à implémenter, et elle permet de traverser facilement les coupe-feu. En effet, presque tous les coupe-feu du monde sont déjà configurés pour laisser passer le trafic HTTP, à cause du Web. Le serveur Web externe de l'entreprise n'étant pas nécessairement (et même probablement pas) la machine hôte du gestionnaire représentée Fig. 7, il suffit quasiment de faire du copier-coller dans le fichier de configuration du coupe-feu pour permettre au trafic de gestion de passer à travers ce coupe-feu. Dans bien des entreprises, y compris des PME, ceci ne nécessite pas le recours à un consultant extérieur.

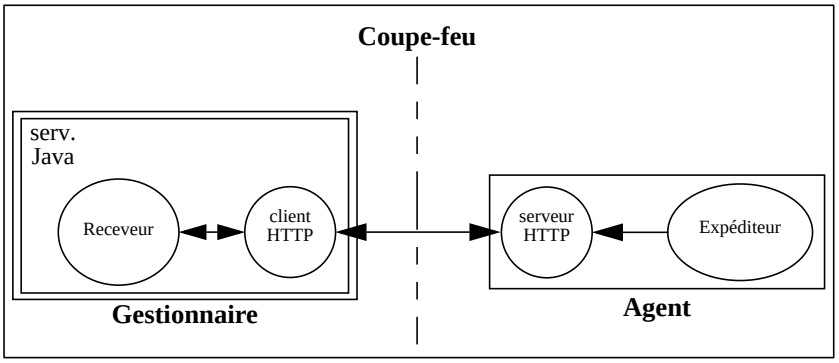

Fig. 7. Distribution via HTTP

Le principal problème à résoudre est celui du contrôle des temporisations. En effet, la robustesse des systèmes d'exploitation et des services Web repose en partie sur le ménage automatique des socquettes TCP et UDP jugées inactives. Ce ménage repose sur un certain nombre d'hypothèses. Étudions-les et vérifions si elles sont compatibles avec notre utilisation de connexions TCP persistantes pour réaliser du *push*.

En ce qui concerne le système d'exploitation, sa conception de l'activité d'une socquette est très particulière. Il ne fait rien si aucune donnée ne transite à travers la socquette, ni dans un sens, ni dans l'autre. Autrement dit, même si nous avons de nombreuses connexions persistantes entre le gestionnaire et les agents, aucune d'entre elles ne sera rompue par le système d'exploitation du fait de son inactivité. En revanche, si le système d'exploitation envoie des données à travers la socquette et si la couche TCP ne reçoit pas de ACK durant un certain laps de temps, cette fois le système d'exploitation estimera que la socquette est inactive et la fermera automatiquement. Cette temporisation est en général de 9 minutes [11, p. 299]. Elle est codée en dur dans le noyau. Sur la plupart des machines, elle n'est pas modifiable [*ibid.*]. Dans le cas de gestion de réseaux qui nous intéresse, on n'a rien à craindre des systèmes d'exploitation de l'agent et du gestionnaire tant qu'on n'a pas de problème avec le réseau. Si on a une panne réseau et qu'elle dure plus de 9 minutes, la socquette sera fermée côté agent, mais le gestionnaire n'en sera pas averti. Pour que ce dernier s'en rende compte, nous préconisons d'envoyer un *keepalive* côté gestionnaire à intervalle régulier dès qu'on n'a plus reçu de données d'un agent pendant plus d'un certain temps, par exemple 9 minutes. Dès que le réseau sera rétabli, le gestionnaire recevra alors un RST de la part de l'agent, ce qui lui indiquera clairement que la socquette a été fermée par l'agent et qu'il faut en créer une nouvelle. La fréquence des *keepalives* et le temps au bout duquel il faut commencer à en envoyer sont des paramètres à déterminer expérimentalement.

En ce qui concerne le client HTTP, il n'y a aucun risque de rompre la connexion puisqu'on est toujours en mode lecture sur la socquette dans la phase distribution.

En ce qui concerne le serveur HTTP dans le cas général du Web, le principal ennemi est l'impatience des utilisateurs quand les transferts sont lents. En effet, beaucoup d'utilisateurs n'attendent pas plus de 10 secondes avant de recharger une même page, ce qui peut causer (en fonction des butineurs et de la stabilité des liens réseaux) des connexions dites "à moitié ouvertes": un côté a fermé la socquette sans faire un `shutdown` (ou sans que le `shutdown` atteigne sa destination), ce qui empêche l'autre côté de fermer proprement la socquette et de libérer les ressources correspondantes. Ce problème est particulièrement critique pour les serveurs Web très sollicités, recevant plus de 100 requêtes par seconde. À ce rythme, on a tôt fait de saturer tous les descripteurs TCP disponibles si l'on ne prend pas garde à faire du ménage rapidement. (Ce problème est bien connu des *hackers*, qui l'exploitent à des fins de déni de service.) Les développeurs de serveurs HTTP ont adopté différentes stratégies pour circonvenir ces problèmes. Les deux principales sont (i) d'assigner une borne supérieure à la durée de vie d'une socquette, et (ii) d'assigner une borne supérieure à la période d'inactivité d'une socquette, c'est-à-dire au temps durant lequel aucun trafic ne transite via la socquette. Du temps de HTTP/1.0, les temporisations étaient souvent codées en dur dans les serveurs. Avec l'avènement des connexions persistantes de HTTP/1.1, la gestion des temporisations s'est affinée. La plupart des serveurs HTTP/1.1 permettent aujourd'hui à l'administrateur de contrôler depuis le fichier de configuration du serveur HTTP l'une ou l'autre des bornes supérieures précisées ci-dessus.

En ce qui concerne le serveur HTTP dans le cas particulier qui nous intéresse, celui de la gestion des réseaux IP, on ne court plus le risque d'avoir 100 requêtes par seconde, et on bénéficie toujours du contrôle des temporisations. Il suffit donc de donner à ces temporisations des valeurs supérieures à la période de *push* la plus petite pour cet agent (celle qui garantit que la socquette est active) pour que l'on ne court pas le risque de voir la connexion persistante rompue par le serveur HTTP de l'agent. Il est à noter qu'il n'est pas nécessaire que l'administrateur gère explicitement ces temporisations: la gestion des fréquences de *push* via les applettes présentées ch. 2 peut automatiquement mettre à jour ces temporisations.

5. Conclusion

Dans cet article, nous avons décrit un modèle conceptuel de haut niveau pour gérer des réseaux IP à l'aide des technologies Web et du modèle *push*. Cette approche présente plusieurs avantages par rapport à l'approche classique basée sur SNMP, et ne nécessite pas de plateforme de gestion de réseaux coûteuse. Nous avons détaillé les trois phases du modèle *push* (publication, souscription et distribution), puis nous sommes penchés sur les aspects de communication entre agent et gestionnaire. Nous avons étudié trois technologies (les socquettes, Java RMI et HTTP), et présenté leurs avantages et inconvénients respectifs. En prenant en compte les contraintes de sécurité imposées par la traversée d'un coupe-feu, la meilleure solution selon nous est HTTP.

À l'avenir, il serait intéressant d'étudier l'utilisation possible de bases de données actives en gestion de réseaux IP. Dans le modèle conceptuel que nous avons présenté, les agents envoient des données à une servelette Java tournant sur le gestionnaire, puis s'en remettent à l'application de gestion pour prendre les mesures qui s'imposent et stocker les données de gestion dans une base de données. On pourrait changer ce modèle pour que les agents envoient directement leurs données de gestion dans une base de données active, et exploiter alors les actions déclenchées (*trigger-based actions*) offertes par ce type de base de données.

Remerciements

Cette recherche a été en partie financée par le Fonds National de la Recherche Scientifique (FNRS) grâce au projet SPP-ICS 5003-45311. L'auteur souhaite remercier G. Madhusudan pour des discussions très enrichissantes sur la communication dans des applications distribuées écrites en Java, ainsi que L. Bovet sur MIME et W. Almesberger sur TCP.

Références

[1] D.B. Chapman et E.D. Zwicky. *Building Internet Firewalls*. O'Reilly & Associates, Sebastopol, CA, USA, 1995.

[2] N. Freed et N. Borenstein (Eds.). *RFC 2046. Multipurpose Internet Mail Extensions (MIME). Part Two: Media Types*. IETF, novembre 1996.

[3] G. Goldszmidt. *Distributed Management by Delegation*. Thèse de doctorat, Columbia University, New York, NY, USA, décembre 1995.

[4] IANA. *Protocol Numbers and Assignment Services*. Disponible depuis <http://www.iana.org/numbers.html>. Ce site Web remplace la RFC 1700 qui est maintenant obsolète.

[5] J.P. Martin-Flatin. *Push vs. Pull in Web-Based Network Management*. Rapport technique SSC/1998/022, version 3, SSC, EPFL, Lausanne, Suisse, novembre 1998.

[6] J.P. Martin-Flatin. *IP Network Management Platforms Before the Web*. Rapport technique SSC/1998/021, version 2, SSC, EPFL, Lausanne, Suisse, décembre 1998.

[7] J.P. Martin-Flatin, S. Znaty et J.P. Hubaux. "A Survey of Distributed Enterprise Network and Systems Management". *Journal of Network and Systems Management*, vol. 7, no. 1, mars 1999.

[8] Netscape. *An Exploration of Dynamic Documents*. 1995. Disponible depuis <http://home.mcom.com/assist/net_sites/pushpull.html>.

[9] P. Sridharan. *Advanced Java networking*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[10] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. 3ème édition. Addison-Wesley, Reading, MA, USA, 1999.

[11] W.R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison-Wesley, Reading, MA, USA, 1994.

[12] C. Wellens et K. Auerbach. "Towards Useful Management". *The Simple Times*, vol. 4, no. 3, juillet 1996.