Chapitre 6

JMX : un standard pour la gestion Java

6.1. Introduction

Dès son apparition, la technologie Java a suscité un intérêt extrêmement fort dans le domaine de la gestion des équipements, réseaux et services. Initialement mise en avant comme le langage de programmation du Web, cette technologie a été utilisée dans de nombreux prototypes liant les technologies Web aux plates-formes de gestion. Ce n'est qu'en 1999 que la technologie Java a vu son application dans le domaine de la gestion se développer fortement en s'imposant comme un langage de programmation de plus en plus efficace et populaire, mais surtout en offrant une infrastructure dynamique et ouverte disposant d'une palette très large d'interfaces de programmation, d'objets métiers, ainsi qu'un accès à de multiples types de communication.

JMX est l'acronyme de *Java Management Extensions*. C'est une initiative d'un consortium d'industriels dont les membres fondateurs sont *Bullsoft, Computer Associates, IBM, Powerware, Sun Microsystems, TIBCO* et *Alcatel/Xylan*. Ce consortium a été renforcé courant 2000 avec l'arrivée des sociétés[1] *BEA, Borland/Inprise, Gemstone, IONA, Iplanet, MGE, Motorola* et *Schmid Telecommunication*. Cet effort se place dans le contexte du Java Community Process (JCP) permettant à des tiers de participer à l'extension de la technologie Java. L'objectif initial de ce consortium est de définir une architecture ainsi qu'un

1. D'autres industriels contribuent dans des groupes de travail ou des JSR (*Java Specification Request*) dont les résultats sont en lien avec JMX. Ils seront mentionnés dans les sections relatives à ces technologies.

Chapitre rédigé par Olivier FESTOR et Laurent ANDREY.

ensemble d'interfaces de programmation standards pour l'utilisation de la technologie Java en gestion d'équipements, de réseaux, services et applications. Plus précisément, JMX a deux objectifs complémentaires : la gestion d'applications Java et la gestion en Java d'équipements, réseaux et services qui n'intègrent pas forcément cette technologie. Cette approche ouverte est aujourd'hui la plus aboutie dans le monde Java et des produits commerciaux ou libres extrêmement intéressants sont disponibles autour de cette technologie. Elle est à ce jour composée de deux phases. La première qui a abouti début 2000, s'est concentrée sur une architecture d'agent de gestion et d'instrumentation de composants. Elle correspond actuellement à la version 1.2 du jsr 03 (*java specification request*) [SUN 02]. La seconde phase qui vient de publier une première version publique d'une spécification (*public final draft*) se focalise sur l'interopérabilité, la transparence, la sécurité et la flexibilité des relations entre agents JMX et les applications de supervision clientes. Il s'agit du jsr 160 [SUN 03].

Afin de donner une vision globale, mais cependant assez détaillée de cette architecture, le chapitre est organisé de la manière suivante. La première section présente les grands objectifs de l'architecture et motive le choix de la technologie Java pour la réaliser. La section suivante présente l'architecture dans sa globalité et définit les composants de base de l'instrumentation : les *Mbeans*. Une fois la partie instrumentation détaillée, nous présentons les services disponibles dans un agent puis nous énumérons les services transversaux disponibles dans l'architecture pour s'intégrer avec des solutions standards de gestion. Nous poursuivons par une présentation des moyens « standards » d'interopérabilité entre agents et clients JMX. Finalement, nous dressons une liste des implantations et produits aujourd'hui disponibles autour de JMX avant de conclure ce chapitre par un résumé de l'architecture suivi d'un ensemble de défis que JMX se doit encore de relever pour s'imposer définitivement comme architecture de supervision du monde Java.

6.2. Les motivations de JMX

JMX adresse la supervision d'applications Java ainsi que la supervision par Java. L'objectif premier est donc d'encourager l'utilisation de la technologie Java en supervision d'équipements, de réseaux, de services et d'applications[2]. Le document d'introduction à l'architecture [SUN 02] présente l'ensemble des caractéristiques qu'offre l'approche JMX. Celles-ci sont guidées par les motivations suivantes :

– La dynamicité des services et des applications est en plein essor. Ceux-ci se conçoivent, se déploient et disparaissent à un rythme qui n'est plus compatible avec les solutions de supervision de seconde génération que nous connaissons

2. Dans la suite du chapitre, le terme gestion de réseaux sous-entend la gestion des équipements, réseaux, services et applications.

aujourd'hui. Des architectures plus souples et évolutives sont requises pour maîtriser ce nouveau niveau de complexité ;

– Les applications distribuées en Java se développent fortement. Ces applications doivent être supervisables et l'effort d'instrumentation de celles-ci doit être réduit au maximum. Pour répondre à ce besoin, une approche conceptuelle de supervision et un support logiciel extrêmement simples à aborder et à mettre en œuvre sont à proposer ;

– Le passage à l'échelle reste aujourd'hui l'un des problèmes majeurs en supervision. La technologie Java permet efficacement la conception, le développement et l'exploitation de logiciels basés sur des composants dynamiques capables de répondre à des besoins de distribution et de délégation ;

– Les paradigmes et modèles de supervision évoluent dans le temps. A titre d'exemple, l'avènement de l'approche WBEM introduit dans le monde de la supervision un nouveau modèle de l'information et un protocole spécifique qu'il faut pouvoir intégrer. Il est aujourd'hui nécessaire de disposer d'architectures de supervision pérennes capable d'intégrer les modèles et approches futures.

L'architecture de JMX propose des solutions à ces quatre besoins. Le premier est comblé dans JMX par une architecture d'agent léger, flexible et ouvert à une extensibilité dynamique de ses composants. Le second est comblé en proposant dans JMX d'une part un modèle d'objet géré facile à appréhender et en mettant à disposition des développeurs d'applications Java une architecture d'agent de supervision simple à instancier dans tout composant Java et/ou machine virtuelle. Pour supporter le passage à l'échelle, JMX s'appuie sur un modèle de composants indépendants, distribuables et instanciables à la demande parfaitement adapté à la gestion par délégation. L'évolutivité est supportée en séparant au niveau de l'architecture d'agent, la représentation interne des objets gérés de l'interface offerte aux applications de gestion. Ceci permet d'instancier dynamiquement de nouvelles interfaces protocolaires adaptées aux modèles requis. L'ensemble de ces composants sera détaillé dans les sections suivantes.

Une motivation supplémentaire pour une nouvelle architecture de supervision autour de la technologie Java est que celle-ci est aujourd'hui mature. Les architectures de supervision peuvent de plus, exploiter l'ensemble des technologies Java existantes. Nombreux sont les composants très intéressants et utiles dans le cadre de la supervision : les bus à message (JMS), l'interface d'accès aux services d'annuaires (JNDI, SLP), l'accès aux bases de données (JDBC), les services transactionnels (JTS), les architectures de serveurs applicatifs (EJB), tous les supports de communication (TCP, SNMP, HTTP…) et bus logiciels standards tels que RMI, CORBA/IIOP ainsi que les technologies émergeantes telles que JINI par exemple.

Afin de pérenniser l'architecture, et de permettre à des composants tiers actuellement basés sur des approches de supervision standards, de bénéficier de la technologie Java, JMX est proposé comme une extension de Java et offre un support des technologies de supervision déjà existantes (SNMP, CMIS, WBEM). Celui-ci est assuré au travers d'interfaces de programmation transversales. S'ajoute à ce processus de définition d'architecture et d'interfaces transversales la définition pour chaque composant de l'architecture, d'un ensemble de tests de conformité que doit passer toute implantation qui désire obtenir le label de conformité. De plus, l'ouverture aux partenariats et le large support industriel contribuent à renforcer l'impact de cette approche dans les réseaux et services de demain.

6.3. L'architecture générale

L'architecture de JMX repose sur le modèle organisationnel standard tel que défini par l'ISO et l'UIT-T dans le cadre général des architectures de supervision. On retrouve donc une répartition des composants suivant trois rôles : le superviseur, l'agent, les objets gérés. La figure 6.1 illustre les différents composants ainsi que leur localisation par rapport au modèle de gestion défini par l'UIT-T.

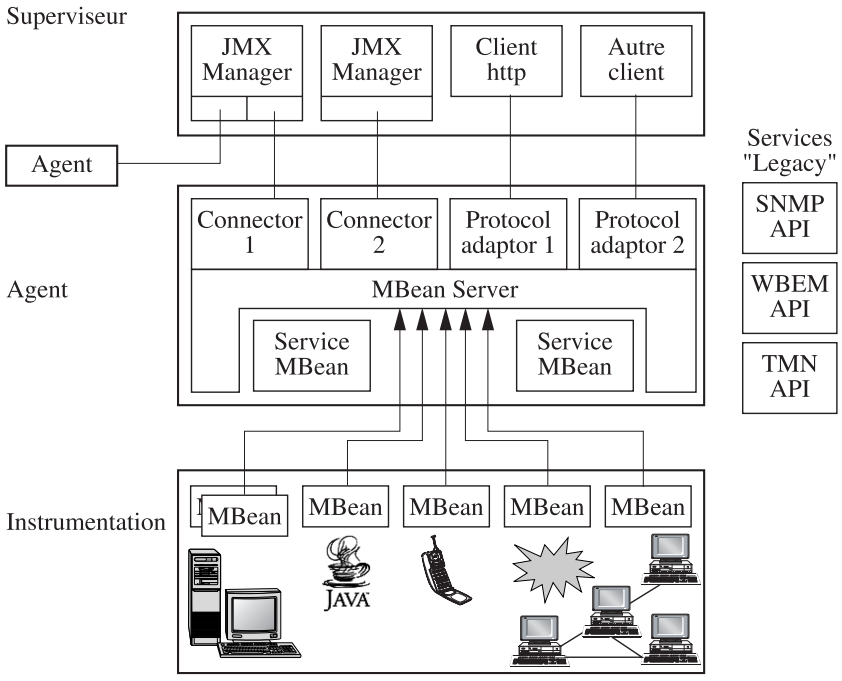

Figure 6.1. *L'architecture générale de JMX*

Le concept de base de l'architecture JMX est le MBean. Un MBean est littéralement un objet géré simple à implémenter qui représente une ressource pour ses besoins de supervision ou un service utile à la supervision. Concrètement, un MBean se traduit en un objet Java qui respecte un certain patron de programmation Java. Le MBean est utilisé pour instrumenter les ressources et forme la base du niveau instrumentation.

Les MBeans sont concentrés au niveau d'un agent au même titre que les objets gérés sont intégrés dans un agent dans l'approche OSI. Dans JMX, l'agent est défini comme un conteneur d'objets gérés. Il permet aux objets gérés d'être enregistrés, d'avoir un nom unique. Il dispose aussi d'un certain nombre de services offerts aux objets gérés mais également aux applications de gestion. Tout objet géré est accédé au travers d'un conteneur (appelé MbeanServer). Il offre à l'agent l'ensemble des méthodes pour créer, détruire un objet géré, lire des attributs, modifier des attributs et invoquer des méthodes sur les objets gérés. Un agent peut contenir plusieurs serveurs d'objets gérés. Afin de permettre aux applications de supervision d'accéder aux objets gérés des agents au travers d'un serveur d'objets gérés, l'architecture JMX propose deux types d'accès à distance : les connecteurs et les adaptateurs de protocoles. Les connecteurs permettent à un client de faire des appels de méthodes à distance sur un MBeanServer d'un agent. Typiquement un connecteur peut être bâti au dessus de RMI (*Remote Method Invocation*) de Java. Une implémentation particulière d'un connecteur impacte aussi bien le côté agent que le côté client/manageur. Les adaptateurs de protocoles sont des composants qui assurent la liaison entre des protocoles spécifiques (par exemple SNMP ou http) et les services locaux d'un serveur d'objets géré. Ces derniers permettent à des agents JMX et les Mbeans qu'ils contiennent d'être accessibles au travers d'approches existantes. Aussi, les adaptateurs offrent à l'architecture une capacité d'évolution et d'intégration des approches futures de supervision. Un adaptateur est un composant lié uniquement au côté agent. L'ensemble de ces composants (Agent, serveur d'objets géré, connecteurs et adaptateurs) forme le niveau agent de l'architecture.

Le niveau superviseur comporte l'ensemble des applications de supervision et outils d'accès aux informations fournies par les agents. Ce niveau supporte *a priori* n'importe quel type d'applications et est ouvert à de nombreux protocoles. Actuellement, JMX ne propose pas de structure Java pour ce niveau. Ceci fait l'objet de la seconde phase de l'architecture qui a démarré début 2000. Cette phase adresse de manière générale les services de supervision distribués (au sens services offerts aux applications de supervision) et se focalise dans un premier temps sur des services de localisation des agents ainsi que sur des mécanismes d'accès à distance aux fonctions de supervision offertes par les agents. Elle a débouché sur la version courant du jsr160 [SUN 02].

De manière transversale à cette architecture Gestionnaire/Agent/Instrumentation, JMX propose un ensemble d'interfaces de programmation protocolaires dites

transversales. Ces interfaces sont mises à disposition des développeurs d'agents et/ou d'objets gérés pour faciliter la programmation de l'échange d'informations de gestion avec des ressources distantes accessibles uniquement au travers d'un protocole de gestion standard, c'est-à-dire SNMP, CMIP ou WBEM. Ces interfaces sont dédiées à l'intégration. Leur utilisation encourage la mise en place d'agents de gestion JMX et permet une transition graduelle vers cette technologie.

6.4. Les objets génériques

Tout objet géré de l'approche JMX est un objet Java respectant un patron de conception strict. Les concepteurs de JMX ont défini quatre types d'objets gérés adaptés chacun à un profil d'instrumentation donné. Ces Mbeans sont : le MBean standard, le MBean dynamique, le MBean modèle et le MBean ouvert.

6.4.1. *Les Mbeans standards*

Le MBean standard est le composant de supervision (objet géré) le plus simple. Celui-ci est préconisé pour la réalisation d'objets gérés dont l'interface de supervision (opérations et actions exposées au travers du serveur d'objets gérés[3]) est très stable et ne change pas d'une instance d'objet à l'autre. L'interface de supervision offerte par ce type d'objet géré est fixée à la compilation. Le serveur (inclus dans un agent) découvre l'interface par introspection sur leur classe et toutes les instances de l'objet géré considéré auront la même interface de supervision.

6.4.1.1. *Composition*

Un MBean standard est composé de deux entités Java : une interface qui définit les attributs et méthodes exposés au serveur d'objets gérés pour des besoins de supervision et un objet Java qui implémente ces attributs et méthodes ainsi que le comportement associé. Les attributs de supervision sont définis dans l'interface par des méthodes de lecture et ou d'affectation en respectant le patron suivant :

– pour définir un attribut autorisé en lecture, le concepteur du MBean doit définir dans l'interface de supervision de l'objet une méthode dont la signature est la suivante : TypeAttribut get*NomAttribut*(). Lors de l'introspection, le serveur de code va identifier cette méthode comme la définition d'un attribut de supervision autorisé en lecture. Le nom de cet attribut est celui post-fixant l'opération dans la définition de la méthode ;

3. Afin d'éviter toute ambiguïté entre les attributs et méthodes d'un objet Java et les attributs et méthodes définis dans l'interface de gestion qui sont également des attributs et méthodes Java, nous ferons toujours référence aux attributs de supervision en termes d'opération de gestion et dénommerons les méthodes offertes à l'interface de supervision, des actions.

– pour définir un attribut sur lequel on autorise une opération d'affectation, l'interface de l'objet doit contenir une méthode dont la signature est la suivante : void set*NomAttribut*(TypeAttribut). Cette méthode sera identifiée lors de l'introspection par le serveur d'objets gérés comme un support de l'opération d'affectation sur l'attribut de nom *NomAttribut* ;

La présence des deux méthodes ci-dessus pour un même attribut indique que l'accès à celui-ci est autorisé à la fois en lecture et en écriture. Les méthodes offertes à l'interface de gestion (actions au sens OSI) constituent le reste des méthodes définies dans l'interface. Pour des raisons de cohérence avec les opérations de gestion, les actions ne peuvent pas commencer par un préfixe *get* ou *set*.

Afin de permettre au serveur d'objets de découvrir l'interface de supervision offerte par l'objet, l'interface Java qui la définit doit se conformer à la convention de nommage suivante *NomObjet*MBean où *NomObjet* est le nom effectif de la classe qui implante cette interface. Sur la base de cette convention de nommage, le serveur est capable de retrouver l'interface de supervision associée au MBean standard.

6.4.1.2. *Un exemple*

Afin d'illustrer la programmation d'objets JMX, nous prendrons tout au long du chapitre un même modèle d'objet. Celui-ci modélise un dispositif de surveillance et de contrôle très simple d'un serveur Web. L'objet géré associé exhibe les attributs et méthodes suivants :

– `MaxConnexions` : un attribut qui indique le nombre maximal de connexions simultanées autorisées. Cet attribut est de type entier. Il est autorisé en lecture et en écriture ;

– `IPHit` : un attribut qui renvoie la liste des adresses IP (sous forme d'un objet Java), classées par nombre de connexions qui se sont connectées sur le site depuis le lancement du serveur. Cet attribut est en lecture seule ;

– `ResetServer(DateTime t, Integer m)` : action qui relance le serveur Web à la date et heure donnée en paramètre, positionne le nombre maximal de connexions simultanées à la valeur donnée dans le second paramètre de l'action.

A partir de cette spécification informelle, nous pouvons construire le MBean standard correspondant. La figure 6.2 comprend la définition de l'interface. Celle-ci porte le nom de l'objet, ici `ObjetWeb`, suivi du patron MBean comme l'exige la convention JMX.

L'interface définit les attributs et méthodes qui sont exposés à la supervision. On retrouve deux opérations (`Get` et `Set`) pour l'attribut `MaxConnexions`, une

opération qui indique que l'attribut `IPHit` est en lecture et que celui-ci renvoie une table.

```
1.public interface ObjetWebMBean
2.{
3.  public Integer getMaxConnexions() ;
4.  public void setMaxConnexions(Integer i) ;
5.  public IPHitItem[] getIPHit()
6.  public void resetServer(DateTime t, Integer m);
7.}
```

Figure 6.2. *Définition d'interface de Mbean standard*

Une seule action est offerte à l'interface de supervision. Elle est matérialisée par la méthode `resetServer` qui a deux paramètres : la date à laquelle le serveur doit être relancé et la valeur maximale du nombre de connexions que doit autoriser le serveur après son redémarrage.

La figure 6.3 montre le code Java de l'objet implantant cette interface. On retrouve les conventions de nommage définies précédemment : l'objet porte le même nom que l'interface sans le suffixe MBean. Il implante un constructeur public par défaut ainsi que toutes les méthodes définies dans l'interface de gestion. Nous avons ajouté deux attributs internes publics : l'attribut `maTable[]` qui maintient la tables des statistiques de connexions et `maxConnect` qui maintient en interne le nombre maximal de connexions simultanées ouvertes. Ces deux attributs, bien que publics, ne sont pas visibles à l'interface de supervision car ils ne sont pas définis dans l'interface du MBean. Il en est de même pour la méthode `planInternalReset`.

Lors du chargement de cette classe, le serveur d'objets implanté dans l'agent va effectuer une introspection sur la classe. De cette introspection, il va reconstituer l'interface de supervision (interface `ObjetMBean` plus toutes les interfaces héritées de super-classes qui seraient également des Mbeans standards). L'ensemble des attributs (opérations `get` et/ou `set`) et actions découvertes dans cette interface reconstituée seront offertes aux applications de gestion sur ce type d'objets au travers du serveur. Les constructeurs publics d'une classe concrète implantant cette interface MBean seront accessibles *via* les méthodes *Create(….)* d'un MBeanServeur afin d'y instancier des Mbean.

Le MBean standard est le moyen le plus simple de créer un objet géré dans l'architecture JMX. Simples, ces objets ont cependant une limite : leur interface de supervision ne peut pas changer dynamiquement et toutes les instances de cet objet

exposeront exactement la même interface. Afin de palier cette limitation, JMX définit un second type d'objet géré : le MBean dynamique.

```
1. public class ObjectWeb implements ObjectWebMBean
2. {
3. public ObjectWeb()
4. {
5.  maxConn = 5;
6.  maTable = new IPHitTable();
7. };
8.
9. public Integer getMaxConnexions()
10.{
11. return maxConn;
12.};
13.
14.public void setMaxConnexions(Integer I)
15.{
16. maxConn = I;
17. if (connexionsCourantes >= maxConn)
18.  {
19.    fermerDesConnexions() ; }
20.  }
21.};
22.
23.public IPHitTable getIPHit()
24.{
25.  return maTable;
26.};
27.
28.public void resetServer(DateTime t, Integer m)
29.{ planInternalReset(t,m);
30.};
31.
32.public void planInternalReset(DateTime t,Integer m)
33.{
34. …
```

Figure 6.3. *Mbean standard*

6.4.2. *Les Mbeans dynamiques*

La dénomination dynamique pour ces Mbeans vient du fait que l'interface de supervision de ces objets n'est pas figée à la compilation mais définie par les objets gérés eux-même à l'exécution. Alors que dans le MBean standard, le conteneur d'objets gérés se charge de découvrir par introspection les attributs et méthodes de l'objet, il va demander à l'objet géré dynamique réaliser ce travail de création des méta-données (description de l'interface exposée à la supervision) à sa place et de lui fournir cette interface de supervision à la demande. L'instance d'objet est donc responsable de la construction de l'interface de gestion qu'elle désire offrir et ceci à

l'exécution et non plus à la compilation. Dans ce cas, deux instances d'une même classe d'objet géré peuvent exposer des interfaces de supervision totalement différentes.

L'utilisation de ce type d'objet géré est intéressante lorsque l'interface de supervision d'une classe d'objet géré ne peut-être statique. Ceci se présente principalement dans deux cas :

– les attributs et méthodes offertes à l'interface de gestion sont fonction du superviseur qui les a créés (une notion de droit d'accès). Dans ce cas, l'objet crée à l'instanciation l'interface en fonction d'un contexte externe, par exemple l'identification de l'administrateur ;

– l'objet ne peut construire son interface qu'au vu des informations disponibles sur la ressource qu'il modélise. Par exemple, on peut imaginer qu'une implantation IP supervisée permet de remettre le compteur de paquets reçus à zéro et pas une autre. Bien que la classe d'objet géré qui représente la couche IP soit unique, les deux instances n'offriront pas exactement la même interface.

Ce dernier point rappelle étrangement le concept de paquetage optionnel et conditionnel dans la modélisation des ressources gérées de l'approche OSI. Ce mécanisme est parfaitement adapté pour implanter ce genre de concept comme nous le verrons plus tard.

6.4.2.1. *Composition*

Un Mbean dynamique doit se conformer (en l'implantant) à une interface Java définie dans l'environnement JMX. Cette interface est l'interface DynamicMBean. Elle comprend :

– une méthode d'interrogation de l'interface de gestion (`getMBeanInfo`). Cette méthode renvoie un objet de type `MBeanInfo`. Celui-ci comporte une description complète de l'interface de gestion de l'objet géré, à savoir : les attributs et les opérations autorisées, les actions et leurs signatures, les notifications que peut émettre l'objet, les constructeurs possibles pour l'instanciation *via* le MBeanServer, ainsi qu'une description textuelle de chaque composant. Cette méthode en autre est invoquée par le serveur (MBeanServer) lorsqu'il a besoin de connaître l'interface de gestion qu'une instance d'objet est prête à offrir au travers du serveur ;

– des méthodes d'exécution. Elles servent au serveur pour demander à l'objet l'exécution d'opérations (lecture, écriture sur des attributs) et de méthodes (actions de gestion). Ces méthodes sont :

- `Object invoke(String,Object[])` : méthode invoquée par le serveur pour demander à l'objet géré l'exécution de l'action dont le nom et les paramètres sont fournis comme paramètres de la méthode. Le résultat de l'invocation de la méthode est renvoyé dans l'objet résultat de la méthode `invoke` ;

- `Object getAttribute(String)` : méthode invoquée par le serveur pour récupérer la valeur de l'attribut dont le nom (tel qu'il est défini dans l'interface de gestion) est donné en paramètre ;
- `AttributeList getAtributes(String[])` : méthode invoquée par le serveur pour récupérer la valeur d'une liste d'attributs d'un objet géré. Les attributs sont identifiés par leurs noms et placés dans le tableau paramètre sous forme de chaîne de caractères. La réponse comporte pour chaque attribut : son nom et sa valeur ;
- `setAttribute(Attribute)` : méthode invoquée par le serveur sur l'objet pour modifier la valeur d'un attribut. Le paramètre Attribut comprend le nom de l'attribut à modifier ainsi que la valeur à lui affecter ;
- `AttributeList SetAttributes(AttributeList)` : méthode identique à la méthode précédente mais s'appliquant sur plusieurs attributs en une seule fois.

Le descripteur d'interface de supervision comprend un ensemble d'objets qui permettent de spécifier tous les composants de cette interface. Le modèle objet de ce composant est illustré dans la figure 6.4.

Pour chaque composant de l'interface de gestion (attribut, constructeur, méthode, notification), l'objet géré indique dans le `MBeanInfo` son nom ainsi qu'une description textuelle (`MbeanFeatureInfo`). Pour chaque attribut, il construit un objet `MbeanAttributeInfo` qui comporte : le type Java de l'attribut, son nom et ses droits d'accès (`isReadable()`, `isWritable()`, `iIs()`). Pour chaque constructeur offert à l'interface de supervision, le `MBeanInfo` contient sa signature (liste des paramètres dans la classe `MBeanConstructorInfo`). Les notifications que l'objet est susceptible d'émettre sont décrites dans la classe `MBeanNotificationInfo`. Leur description comprend principalement la liste des types Java de celles-ci. Pour chaque méthode offerte, l'information contenue dans la description comporte son nom, la liste de ses paramètres, le type de retour ainsi que l'impact de la méthode. Quatre types d'impact sont définis : si la méthode engendre des modifications des valeurs dans l'objet ou dans la ressource associée son type doit être `ACTION` ; si la méthode est une méthode de consultation qui n'engendre pas de modification sa classification est `INFO` ; si la méthode est une méthode de consultation mais qu'elle engendre également des modifications, son type sera `ACTION_INFO` ; si finalement son impact est inconnu, elle sera classifiée `UNKNOWN`. Ce dernier type d'impact doit être utilisé avec parcimonie. Il est destiné à être retourné lorsque l'on ne peut pas connaître l'impact. Ceci peut arriver dans des cas ou du code d'instrumentation est généré automatiquement par exemple.

L'ensemble de ces informations sont contenues dans le MbeanInfo de l'objet géré qui y indique en plus le nom de sa classe Java ainsi qu'une description textuelle de son rôle. Afin de permettre au serveur d'objets de bien constituer l'interface de

gestion de l'objet géré, le concepteur de l'objet géré doit consciencieusement remplir cet objet d'informations.

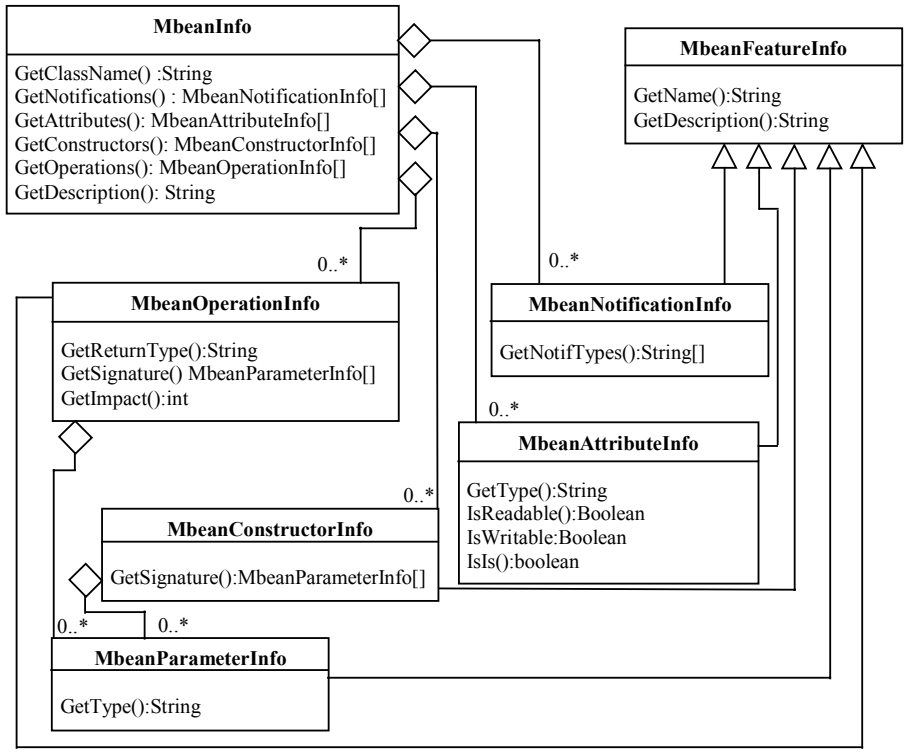

Figure 6.4. *Les composants de l'interface MbeanInfo*

Il faut souligner que le type des attributs paramètres et retour de fonctions sont des classes Java. Les types primitifs Java ne peuvent pas être utilisés en tant que tels dans les Mbeans dynamiques. Il faut pour ces types passer par des wrappers présents dans les bibliothèques standard Java.

6.4.2.2.*Un exemple*

Reprenons l'exemple du superviseur de serveur Web décrit dans la section précédente en rajoutant au cahier des charges la contrainte suivante : tous les serveurs ne peuvent maintenir la liste des clients et les scores associés. C'est lors de l'initialisation de l'objet géré que le choix de supporter ou pas cette fonction est arrêté. Dans l'exemple, ce choix est fixé par un paramètre booléen du constructeur de l'objet. Si ce paramètre est fixé à vrai, alors la fonction est supportée, sinon elle

ne l'est pas En résumé, certaines instances de l'objet offriront l'accès à l'attribut IPHit, d'autres non.

Ci-dessous, nous détaillons une partie[4] du code Java qui implante cet objet géré comme un MBean dynamique. Celui-ci est découpé en trois parties : la déclaration de la classe, la construction dynamique de l'interface de supervision et la réalisation de la méthode d'accès à un attribut. La figure 6.5 comporte la déclaration de la classe ainsi que la définition du constructeur.

La première remarque est que contrairement au MBean standard, l'objet géré dynamique ne définit pas son interface de gestion sous forme d'une interface Java mais implémente l'interface `DynamicMBean` définie dans JMX (`java.management.DynamicMBean`).

```
public class ObjectWebDynamique implements DynamicMBean {

public ObjectWebDynamique(boolean pSupport)
    {
     buildMBeanInfo(pSupport); // voir Figure 6
     }
```

Figure 6.5. *La déclaration du MBean dynamique et son constructeur*

Le constructeur appelle la méthode de création de l'interface de gestion en lui indiquant au travers du booléen `pSupport` si l'attribut `IPHit` doit être offert à l'interface ou non. C'est au sein de cette méthode que l'instance décide quels attributs et méthodes il exhibe à l'interface. Le code de celle-ci est donné dans la figure 6.6. Elle vise à construire l'objet de description de l'interface `MBeanInfo`. La méthode de construction commence par créer la liste des attributs que l'objet va exhiber à l'interface de gestion (lignes 3-14). Le premier attribut est l'attribut `MaxConnexions`. Il est autorisé en lecture et en écriture (remplissage des champs `read` et `write` du descripteur de l'attribut à `true`).

Le second attribut (`IIPHit` lignes 7 à 13) n'est mis à disposition de l'interface de gestion que si le paramètre `pSupport` a été positionné à vrai lors de l'appel de la méthode de construction de l'interface. Ceci illustre parfaitement l'intérêt des MBeans dynamiques car cette mise à disposition conditionnelle n'est pas possible dans un MBean standard.

4. Pour des raisons de place et de clarté du texte, nous ne donnons dans l'exemple que le code de la déclaration de la classe, la méthode de construction de son interface de gestion dynamique ainsi que la méthode d'accès à un attribut que doit implanter l'objet pour rester conforme à l'interface de MBean dynamique. Les autres méthodes suivent un schéma similaire à ceux illustrés ici.

```
1.private void buildMBeanInfo() {
2.
3. dAttributes[0] = new MBeanAttributeInfo("MaxConnexions",
4.                                        "java.lang.Integer",
5.      "Nombre maximal de connexions simultannees autorisees",
6.                                          true,true);
7.if (fSupport)
8. {
9. dAttributes[1] = new MBeanAttributeInfo("IPHit",
10.                                       "IPHitTable",
11.                           "la liste des meilleurs clients.",
12.                                         true,false);
13. }
14.
15. MBeanParameterInfo[] constructorParams = new MBeanParameterInfo[1];
16.
17. constructorParams[0] = new MBeanParameterInfo("java.lang.Boolean",
   "support de la table de hit","");
18.
19. dConstructors[0] = new MBeanConstructorInfo("ObjectWebDynamique:
    construteur de l'objet",constructorParam);
20.
21. MBeanParameterInfo[] params = new MBeanParameterInfo[2];
22. params[0] = new MBeanParameterInfo("DateTime","la date…","");
23. params[1] = new MBeanParameterInfo("Integer","conn max…","");
24.
25. dOperations[0] = new MBeanOperationInfo("resetServer",
26.                    "relance le serveur a la date donnee… ",
27.                    params ,
28.                    "void",
29.                     MBeanOperationInfo.ACTION);
30.
31. dMBeanInfo = new javax.management.MBeanInfo(
32.                    this.getClass().getName(),
33.                    new String("mon objet dynamique"),
34.                    dAttributes,
35.                    dConstructors,
36.                    dOperations,
37.                    new MBeanNotificationInfo[0]);
38. }
```

Figure 6.6. *La méthode de construction dynamique de l'interface de supervision d'un objet géré dynamique*

Les lignes 15 à 19 comprennent la création de l'interface de supervision concernant les constructeurs. Dans un premier temps (ligne 15), la description du paramètre booléen du constructeur offert à l'interface de supervision est construite. Puis cette information est utilisée pour construire la description du constructeur lui-même (ligne 19). Les lignes 21 à 29 comportent le code associé à la construction de la description de la méthode `resetServer` offerte à l'interface de gestion. Dans un premier temps, une description pour chacun de ses paramètres est construite puis cette information est utilisée pour construire le descripteur de la méthode (lignes 25

à 29). Finalement l'ensemble de ces informations sont regroupées dans le descripteur de l'objet géré, construit dans les lignes 31 à 38.

Passons maintenant aux méthodes d'accès et de traitement que doit supporter un MBean dynamique. Ces méthodes sont : la lecture d'attribut(s), l'affectation d'attribut(s) et l'invocation de méthode. La figure 6.7 comprend le code relatif à la lecture d'un attribut. Celle-ci prend en paramètre un nom d'attribut sous forme de chaîne de caractères et renvoie, sauf erreur lors du traitement, l'objet associé (ligne 1).

```
1. public Object getAttribute(String attribute_name) throws
2.       AttributeNotFoundException, RunTimeOperationsException
3.        IllegalArgumentException
4.  {
5.     if (attribute_name == null)
6.     {
7.         throw new RuntimeOperationsException(new
IllegalArgumentException("Null Name"),"Null Name is not permitted");
8.     }
9.     if (attribute_name.equals("MaxConnexions")) {
10.        return fMaxConnexions;
11.    }
12.    if (attribute_name.equals("IPHit")) {
13.            if (fSupport == true)
14.                   return fIPHit;
15.    }
16.    throw(new AttributeNotFoundException(name + "not found"));
17. }
```

Figure 6.7. *Le code de la méthode de lecture d'attributs dans un MBean dynamique*

Dans le cas des MBeans standards, l'interface de gestion est fixée par le concepteur de l'objet à la compilation. Dans le cas des MBeans dynamiques, la définition de cette interface ainsi que l'exécution des opérations est sous la responsabilité de l'objet géré et ces deux tâches sont assurées à l'exécution, il est vrai au prix d'un effort de codage supplémentaire (implantation par l'objet de toutes les méthodes d'accès et d'invocation et construction du descripteur). Dans ces deux cas, le concepteur doit implanter l'objet géré. Pour éviter cette tâche au concepteur d'une application, JMX propose les Mbeans modèles, définis dans la section suivante.

6.4.3. *Les Mbeans modèles*

Les Mbeans modèles apportent un degré supplémentaire de facilité d'usage de l'architecture JMX pour les développeurs d'applications. En effet ce concept d'objet géré permet aux développeurs de rendre leurs composants supervisables avec un minimum d'effort. Le principe qui a guidé la conception de ce type de MBean est le

suivant : un développeur d'application Java qui désire que son application soit supervisable, n'a pas besoin de savoir comment sont construits des objets gérés dans un agent JMX, ni comment ils sont maintenus par le serveur. Sa seule préoccupation pour la supervision de son application est de définir quels sont les points d'entrée dans celle-ci et quelle vision il souhaite donner de son composant aux applications de supervision (le modèle de l'information). Sur la base de ce besoin, l'architecture JMX intègre dans tout serveur d'objet un mécanisme qui permet à une application Java de :

– dire à l'agent quelle interface de supervision il souhaite exposer (noms des attributs, types, signature des méthodes, notifications émises) ;

– dire à l'agent comment cette interface s'instrumente dans son application (donner la liste des correspondances entre les attributs et méthodes offertes à l'interface de gestion par l'objet géré et les appels réels à effectuer sur l'application).

Concrètement ce service permet de déléguer la création et la mise en forme d'un objet géré au serveur. Le développeur de l'application ne se préoccupe plus de cette phase. Il n'a pas à implanter de MBean. Un MBean modèle peut donc être vu comme un objet géré générique dont l'interface peut être construite à la demande d'une application (une factory dans le jargon des patrons de conception). Dans l'approche JMX, tout agent se doit de fournir ce service (classe `javax.management.modelmbean.RequiredModelMBean`).

Composition

Afin d'offrir ce service, tout agent dispose d'un MBean générique (modèle). Un composant applicatif externe peut demander l'instanciation d'un MBean à partir de ce MBean générique *via* un service d'initialisation. L'initialisation consiste à définir les composants (attributs, méthodes et notifications) que ce MBean doit offrir à l'interface de supervision. Le MBean modèle est un MBean dynamique qui étend l'interface de celle-ci et implémente les interfaces de persistance et de diffusion de notifications étendue (`ModelMBeanNotificationBroadcaster`).

En supplément des informations que l'on doit spécifier dans un MbeanInfo, le MBean modèle comprend un descripteur pour chaque attribut, opération et notification qu'il expose à l'interface de supervision. Ce descripteur permet de définir les caractéristiques suivantes sur le MBean lui-même, ses attributs et méthodes.

Pour les attributs ce descripteur comprend les champs suivants :

– le type de persistance souhaitée pour les valeurs d'attributs: à chaque modification, sur déclenchement d'un timer interne au MBean, uniquement à des dates spécifiques ou aucune ;

– la fréquence (en milliseconde) ainsi que le type d'action à entreprendre sur un get ou un set d'attribut : cette description permet de calibrer le cache d'accès à la ressource réelle ;

– le mapping vers les opérations réelles de l'application. Pour chaque attribut et méthode, le descripteur doit, dans le cas ou celui-ci est en lien avec une ressource réelle, spécifier l'appel effectif à la ressource correspondante ;

– des valeurs par défaut que peuvent prendre les attributs ;

– des paramètres de visibilité (1 à 4) qui permettent aux adaptateurs de filtrer suivant ce critère les attributs qu'ils exposent ;

– une chaîne XML qui décrit la manière dont une valeur d'attribut doit être présentée.

Il existe des champs similaires pour le MBean lui-même, les actions et les notifications. Certains champs sont positionnables par l'application (période, présentation, mapping). Il existe des champs supplémentaires qui sont positionnés uniquement par le MBean et ouverts en lecture au travers de l'interface de gestion. Par exemple, le descripteur de tout attribut comprend un champ indiquant la date de la dernière modification de l'attribut et celui d'une méthode maintient la valeur renvoyée lors de la dernière invocation.

La version initiale de la spécification permettait au concepteur d'application de spécifier l'interface du MBean de gestion suivant trois supports externes : XML, IDL et un support textuel. S'ajoute à ces supports la possibilité de créer le descripteur au sein d'un programme Java en instanciant l'objet et en positionnant l'ensemble de ses champs. La dernière version du standard ne fait plus de référence explicite à ces trois modes externes. Cependant il est probable que ceux-ci vont perdurer car ils offrent un moyen convivial et intégrateur de définir les caractéristiques de gestion pour un MBean spécifique à une application.

Une application qui souhaite être supervisable par JMX au travers d'un ou plusieurs Mbeans modèles doit effectuer les deux étapes suivantes :

– elle instancie un MBean modèle dans l'agent JMX de son choix ;

– elle configure 6.le MBean créé au travers du ModelMBeanInfo. Il suffit pour cela d'invoquer la méthode :

`setModelMBeanInfo(ModelMBeanInfo)` sur le MBean créé.

Une fois ces étapes réalisées, l'application est instrumentée et son interface de supervision est accessible par des applications de gestion au travers du serveur d'objets gérés. Pour couper ce lien de supervision, il suffit que l'application désenregistre l'objet géré associé.

6.4.4. *Les Mbeans ouverts*

Dans les *MBeans* standards, tout comme dans les *MBeans* dynamiques et modèle, les types des attributs des paramètres et résultats de méthodes peuvent être n'importe quel type ou objet Java. Ceci est extrêmement puissant mais peut poser problème à certaines applications de supervision génériques telles que des navigateurs de MIB notamment pour l'affichage des valeurs. Afin de faciliter la réalisation de ces outils et pour leur assurer une certaine généricité, l'architecture JMX propose aux développeurs d'objets gérés l'utilisation de *MBeans* ouverts.

Un *MBean* ouvert est un *MBean* dynamique qui respecte un certain nombre de contraintes sur :

– le type de ses attributs, paramètres et résultats de méthodes de gestion ;

– la description des attributs, paramètres, méthodes et notifications.

Les types autorisés sont tous les types Java standard (les types de base de `java.lang`, c'est-à-dire `String, Character, Boolean, Byte, Short, Integer, Long, Float, Double`), les tableaux contenants ces types, ainsi que deux types supplémentaires : `CompositeData` et `TabularData` qui permettent de représenter des types complexes. `CompositeData` est l'équivalent d'une table de hachage statique. Les attributs et paramètres de ce type doivent fournir dans le `MBeanInfo` correspondant un champ donnant une description des données contenues dans cette table. Le type `TabularData` est une table d'objets composites pouvant contenir un nombre indéterminé de lignes et indexé par un nombre indéterminé de colonnes. La structure d'un tel objet est définie à l'instanciation (nombre et types des colonnes). Une fois instanciées, ces caractéristiques ne peuvent plus changer dans l'objet. La structure de tableau pour ces deux types est également autorisée dans les *MBeans* ouverts.

En plus d'une restriction de types, les MBeans ouverts se distinguent des MBeans dynamiques par des descripteurs (`MBeanInfo`) plus riches. En effet, les descripteurs que doivent fournir les différents éléments d'un MBean ouvert sont ceux hérités des MBeans dynamiques avec en plus :

– pour tout attribut et paramètre de type simple : une description de la valeur par défaut et une énumération des valeurs possibles. Ces deux champs peuvent également être renvoyés dans un tableau si l'attribut ou le paramètre sont de type `CompositeData` ;

– pour tout attribut ou paramètre de type tabulaire : la liste des colonnes du tableau (le nom de chaque colonne).

L'ensemble des informations fournies dans le descripteur de l'objet géré ainsi que la restriction des types supportés permet la construction d'applications génériques opérant sur l'ensemble des caractéristiques de ces objets. Par exemple, il

est facile de développer une application de supervision qui affiche sous forme de tableau les valeurs des attributs de type `TabularData`, la structure étant fixée à l'avance.

Il est inutile de donner ici un exemple de MBean ouvert, celui-ci étant très proche du MBean dynamique présenté précédemment. Si les Mbeans ouverts sont aujourd'hui partie intégrante de la norme, des évolutions de ces derniers sont encore envisagées et des types supplémentaires pourraient être ajoutés dans des versions ultérieures.

6.4.5. *Les MBeans de service et de communication*

Il existe dans JMX des objets particuliers qui ne représentent pas une ressource « réelle », mais qui permettent soit à d'autres objets d'utiliser des services utiles, soit d'offrir une interface de communication vers l'extérieur (MBean adaptateurs de protocoles ou MBeans connecteurs). Au sein de l'architecture JMX, ces composants sont des MBeans comme les autres, et par conséquent leur administration peut se faire par JMX. Ils sont détaillés dans les sections suivantes dédiées aux services d'un agent et aux services transversaux.

6.5. Les services d'un agent

Le service de base d'un agent est d'offrir au travers du serveur d'objets gérés une structure d'accueil pour les objets ainsi qu'une interface de service permettant de manipuler ces objets gérés indépendamment de la manière dont ils sont implémentés (MBeans standards, dynamiques, ouverts ou modèles). Le service offert permet :

– d'instancier et de détruire des objets gérés,

– d'énumérer les objets gérés instanciés dans le serveur et de découvrir pour chacun d'eux son interface de supervision ;

– de lire, modifier les attributs et invoquer des opérations (méthodes) sur les objets ;

– de collecter les notifications émises par les objets.

Dans la version initiale de la spécification JMX, ce service est défini de manière locale. Son extension comme une interface distante se fait au travers d'un connecteur JMX dont les principes sont décrits dans [SUN 03].

En complément de la structure d'accueil pour les objets gérés et le service de manipulation de ces objets, un agent JMX assure leur nommage et implante un ensemble de services utiles dans de nombreuses opérations de gestion. Ces services peuvent être bénéfiques aux objets gérés de l'agent mais également aux fonctions de

supervision qui inter-opèrent avec cet agent. Les services supplémentaires sont comme tous les autres composants d'un agent JMX définis comme des objets gérés (MBeans). A ce jour, les services disponibles sont au nombre de cinq. Il s'agit d'un service de timer, un service de scrutation de variables, un service d'exécution de requêtes, un service de chargement dynamique d'objets gérés et tout récemment un service de relations. Ces services sont détaillés dans cette section.

6.5.1. *Le nommage des objets*

Dans JMX, les instances d'objets sont accessibles *via* le serveur d'objets (MBeanServer) au travers d'un nom unique. Toute opération sur un objet au travers du serveur se fait *via* ce nom. Il n'est notamment pas possible d'obtenir *via* le serveur une référence directe sur un objet géré. Un nom d'instance d'objet est défini par : un nom de domaine suivi d'une liste de couples <*attribut,valeur*>.

Par exemple, la séquence :

```
loria.fr:machineName=bousbach,diskName=hd1,partition=1,
```

identifie l'instance qui représente la partition 1 sur le disque hd1 de la machine `bousbach` dans le domaine `loria.fr`.

JMX ne fait aucune assomption sur la sémantique du nom de domaine ni sur celle des attributs. De plus, JMX n'impose aucun ordre d'apparition des couples attributs valeurs dans la liste. En fait, JMX se base sur l'ordre lexicographique des noms d'attributs pour construire la forme normale des noms. Par exemple les noms `loria.fr:machineName=bousbach, diskName=hd1, partition=1` et `loria.fr: diskName=hd1, machineName=bousbach, partition=1` sont identiques pour JMX, le second étant la représentation canonique du premier. Ceci implique que l'organisation logique d'une MIB dans JMX n'est *a priori* pas hiérarchique.

Sur cette structure des noms, JMX supporte un mécanisme de *pattern matching* utile à de nombreux services comme par exemple le service de requêtes décrit plus loin dans cette section. Le *pattern matching* s'appuie sur deux types de jokers : l'un pour un caractère (`?`), l'autre pour une chaîne de caractères (`*`). Par exemple, le nom `loria.fr:*` correspond à tous les objets dans le domaine `loria.fr` et `loria.fr:machineName=bousbach,diskId=??` va correspondre à tous les objets du domaine `loria.fr` qui ont une machine dont le nom est `bousbach` et qui exhibent un attribut d'identité de disque sur deux caractères. Comme le montrent ces exemples, le mécanisme de *pattern matching* sur les noms des objets gérés peut être très utile pour de la sélection de collections d'objets.

Le concept de nom d'objet dans JMX est finalement très proche du concept de nom distinctif défini dans la gestion OSI. Il a cependant trois différences principales par rapport au nommage OSI :

– dans JMX, l'attribut, identifié par son nom, d'un item de nommage ne doit pas nécessairement exister dans l'objet correspondant alors que dans l'OSI l'attribut d'une classe utilisé pour le nommage doit nécessairement exister dans toute instance de l'objet (un attribut de nommage ne peut par exemple pas faire partie d'un paquetage conditionnel dans cette approche) ;

– l'ordre d'apparition des couples <attribut,valeur> est primordial dans l'OSI car il définit la contenance des objets dans l'arbre de la MIB. Ceci n'est pas le cas dans JMX ;

– le concept de domaine n'est pas présent dans l'OSI. Il peut cependant être émulé par l'introduction d'un objet de premier niveau « domaine » dont l'attribut de nommage comporte le nom.

6.5.2. *Le support de notifications*

L'architecture d'un agent JMX offre un mécanisme d'échange de notifications entre objets gérés. Le mécanisme offert est assez simple. Il est basé sur les concepts d'événements tels qu'on les retrouve dans la programmation événementielle de l'*Abstract Window Toolkit* de Java (AWT) à savoir : des sources d'événements, des événements typés et des listeners.

Dans JMX, ces concepts sont repris comme suit :

– un objet qui souhaite émettre des notifications implémente l'interface `NotificationEmitter` (auparavant `NotificationBroadcaster` dans JMX 1.1) de JMX. Cette interface permet à des consommateurs de s'abonner ou se désabonner aux notifications de la source et à cette dernière de les émettre vers les consommateurs ;

– tout objet souhaitant recevoir une notification d'un émetteur doit implémenter l'interface `NotificationListener`. Celle-ci définit les méthodes appelées par les producteurs de notifications lors de l'émission d'une notification vers le consommateur.

JMX définit une racine d'héritage pour toutes les notifications. Celle-ci comporte des informations génériques telles que son type (une chaîne de caractères définissant un type suivant un certain patron), une estampille, un numéro de séquence dans le contexte de la source, un message textuel et des données utilisateurs sous forme d'un objet Java fourni par la source. Cette notification peut être sous-classée par un développeur d'agent pour construire des notifications propres.

JMX définit également un patron de types pour les notifications. Celui-ci impose la structure de la chaîne de caractères qui définit un type. Aujourd'hui deux types sont définis : les types dont la caractérisation commence par `jmx.` et les types externes. Pour ces derniers, JMX impose que le type commence par le nom de l'entreprise qui a défini le type de notifications. Par exemple, `jmx.attribute.change` est un type de notification standard de JMX utilisé pour indiquer le changement de valeur d'un attribut ; `loria.cmis.association` est un type de notification défini par le `loria` et qui concerne le traitement des associations CMIS.

Afin de permettre à des objets consommateurs de ne pas être obligés de recevoir toutes les notifications des objets auxquels ils s'abonnent, le mécanisme de notification de JMX dispose également d'un support de filtrage. Tout objet peut définir un filtre. Celui-ci est réalisé en implémentant l'interface `NotificationFilter`. Celle-ci ne comporte qu'une méthode qui est appelée par la source de la notification afin de vérifier si la notification qu'elle souhaite envoyer à la destination passe le filtre ou non. Si oui, la notification est émise ; si non, elle est détruite pour l'instance cible considérée. Les filtres sont donnés aux sources par les consommateurs. Lors de la phase d'abonnement, les consommateurs indiquent aux sources quels filtres ils désirent voir appliquer.

Le mécanisme de notifications est utilisé dans de nombreux autres services de JMX, notamment dans le service de timer et dans le service de scrutation d'attributs. Chaque service définit ses propres notifications et filtres.

6.5.3. *Le service de Timer*

Le service de timer permet à des *MBeans* de lui déléguer la fonction de réveil. Pour cela, il permet l'envoi de notifications à l'ensemble de ses abonnés à des dates et temps spécifiques. Deux modes de notifications sont supportés : des notifications uniques et des notifications périodiques. Le fonctionnement du service est illustré dans la figure 6.8. Une fois créé et activé (méthode `start`) par une source, le service de timer autorise des *MBeans* à lui déposer des notifications (méthode `addNotification`). Le dépôt d'une notification consiste à donner au timer le type de notification à émettre et le mode dans lequel celle-ci s'inscrit (périodique ou unique). Dans le cas d'une notification périodique, l'ajout comprend également la spécification de l'intervalle de la période ainsi que le nombre maximal de répétitions. Dans le cas d'une notification à déclenchement unique, la date et l'heure de ce déclenchement sont également fournies lors de l'ajout. Une fois ces notifications ajoutées au service, celui-ci se charge de les émettre en fonction de leurs paramètres de déclenchement. Pour recevoir ces notifications, un objet (ici un puit : *sink*) doit spécifier au serveur la liste des notifications auxquelles il souhaite souscrire. Ceci est réalisé en donnant au serveur un *listener* d'événement adapté à la ou les notifications que l'objet souhaite recevoir (utilisation d'un filtre pour le *listener*).

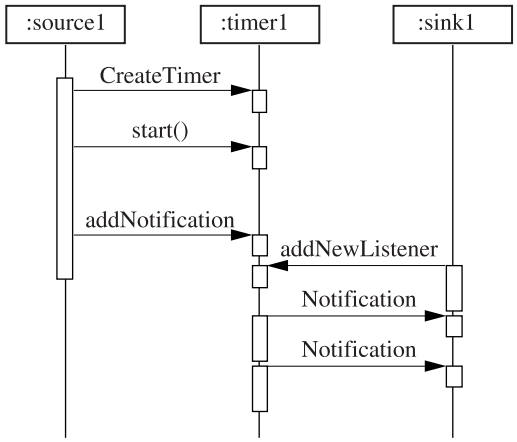

Figure 6.8. *L'instanciation et l'exploitation du service de timer*

Le service se charge ensuite de notifier les objets ayant souscrit en émettant la ou les notifications en accord avec les spécifications des caractéristiques temporelles de celles-ci. Le service supporte l'envoi de notifications multiples et permet à des objets qui s'enregistrent pour la réception d'une notification de récupérer l'ensemble des notifications liées à un timer.

Tout objet géré peut instancier un service de timer, le déclencher et bien sûr l'arrêter. Ce service offre également une routine aux objets gérés afin que ceux-ci l'interrogent sur son état (actif, arrêté). Il faut bien noter qu'un timer est un MBean. Toutes les interactions y afférant peuvent se faire *via* l'interface du MBeanServer qui le contient.

Ce service est très utile aux objets gérés leur facilitant par délégation la gestion des temporisateurs. Par exemple un objet géré devant déclencher en local des tâches en mode batch (par exemple, enregistrer tous les soirs une image des journaux locaux) peut utiliser ce service et de ce fait ne coder dans un listener que les actions relatives à la sauvegarde sans se préoccuper de la gestion du temps. Ce service est également utilisé par d'autres services standards de JMX comme le service de scrutation décrit dans la section suivante.

6.5.4. *Le service de monitoring*

Le service de scrutation est un service qui permet la délégation de scrutation périodique de valeurs d'attributs offert aux applications de gestion mais également à tous les objets gérés d'un agent. Il permet également de maintenir des valeurs d'attributs à la demande. Soit il maintient la dernière valeur observée, soit il

maintient la différence entre les deux dernières valeurs observées sur un même attribut.

Ce service offre des facilités de scrutation pour deux types d'attributs : des attributs numériques ou des attributs de type chaîne de caractères. Pour les premiers il offre deux services : un service de type surveillance de Gauge et un service de type surveillance de compteur. Pour chacun de ces types de surveillance, le service définit des notifications adaptées.

Le service de scrutation de compteurs supporte la surveillance d'attributs de type `Byte, int, short et long`. Ces attributs ont une valeur toujours supérieure à 0 et ne peuvent être qu'incrémentés. Pour surveiller ce type d'attribut, le client du service instancie un objet de scrutation (`CounterMonitor`), et lui donne les paramètres suivants : l'attribut à surveiller (le nom de l'objet géré visé et le nom de l'attribut), la périodicité des surveillances et le seuil de déclenchement d'une alarme. En plus le service permet de déclencher des alarmes sur des seuils multiples *via* la définition d'un offset qui calcule un nouveau seuil dès que le seuil précédent a été atteint par l'attribut. Lors du dépassement du seuil, le service émet la notification `threshold` définie dans le standard.

Le service d'observation de type Gauge supporte les mêmes types d'attributs que le service de scrutation de compteurs. Les paramètres de ce service sont : l'objet géré cible, l'attribut visé, une périodicité de scrutation, un seuil de déclenchement de notification bas et un seuil de déclenchement de notification haut. Lorsque la valeur de l'attribut surveillé passe sous la barre du seuil bas (la précédente valeur était au dessus du seuil), la notification de ce seuil est émise. Lorsque la valeur dépasse le seuil haut (la précédente valeur était sous le seuil), la notification du niveau haut est émise. Le service définit deux notifications pour ce type de scrutation : `high` (dépassement de seuil haut) et `low` (dépassement de seuil bas). Ce service est similaire au mécanisme d'hystérésis que l'on trouve dans la MIB RMON dans le monde SNMP.

Le service d'observation de chaînes de caractères permet d'observer l'égalité entre la valeur de l'attribut observé et la chaîne donnée en paramètre lors de son initialisation. Deux notifications spécifiques sont définies pour ce service : `matches` est une notification qui indique que la chaîne observée est identique à la chaîne demandée ; `differs` est une notification qui indique que les deux chaînes de caractères sont différentes. Ces notifications ne sont émises que lors d'un changement dans l'observation, par exemple si la chaîne observée est différente de la chaîne demandée durant plusieurs observations successives, la notification `differs` n'est envoyée qu'une seule fois lors de la première observation.

En plus des notifications d'information, ce service définit également un ensemble de notifications d'erreurs permettant de remonter au client de la scrutation

des problèmes à l'exécution tels que l'inexistance de l'objet cible ou de l'attribut demandé, ou une incompatibilité entre le type effectif de l'attribut à surveiller et les types supportés par le service.

Ce service est extrêmement utile aux développeurs d'agents. Il est également parfaitement adapté pour de la délégation et permet la limitation du trafic de supervision en autorisant les applications à pousser des traitements de surveillance dans les agents plutôt que de faire du polling périodique à distance.

6.5.5. *Le service de requêtes*

JMX propose un service de requêtes au sein d'un agent (MBeanServer). Celui-ci permet la récupération de collections de MBeans sur un ensemble de critères de sélection. Ce service offre les méthodes queryMBeans ou queryNames. De façon similaire à l'approche OSI[5] celles-ci prennent en paramètre un filtre et une portée. La portée est définie sur un *pattern* de noms d'objets. Elle permet de sélectionner un ensemble d'objets candidats à l'application de la requête sur la base de leur nom. Une fois ces objets sélectionnés, le service leur applique le filtre défini sous forme d'une expression booléenne. Tout objet qui passe le filtre est renvoyé à la source de la requête.

Un filtre est défini par une expression booléenne comportant des opérateurs relationnels (`and,or`) et des opérateurs sur attributs et valeurs (`string, attribute, number, matches,…`). Les filtres sont construits sous forme d'objets Java pour lesquels le service de requête offre des constructeurs statiques. La figure 6.9 comporte un exemple de filtre. Celui-ci sélectionne les objets à ceux dont l'attribut `sysUpTime` est supérieur à 1 heure et dont la personne responsable (attribut `sysContact`) est Tuppence Beresford.

```
QueryExp maRequete =
 Query.and(
       Query.gt( Query.attr(″sysUpTime″),
                 Query.value(3600000)),
       Query.match(Query.attr(″sysContact″),
                   Query.value(″Tuppence Beresford″)
           ) ;
```

Figure 6.9. *Un exemple de filtre*

5. A notre sens d'une manière moins évoluée dans JMX. JMX propose ce mécanisme de filtre de sélection uniquement pour la recherche d'objets gérés, l'approche OSI la propose pour la plupart des opérations du service CMIS.

L'invocation d'une requête avec ce filtre et le nom loria.fr :* par exemple renverra tous les objets du domaine loria.fr qui répondent au critère du filtre.

L'invocation d'une requête se fait sur le serveur au travers des méthodes `queryMBeans` ou `queryNames`. Ces méthodes prennent donc en paramètre la portée (pattern de nom) et le filtre défini par l'expression de la requête. Ce service est requis dans toute implantation conforme du serveur d'objets JMX. Son accès peut être local ou distant *via* un connecteur.

6.5.6. *Le service de chargement d'objets gérés à distance*

Dans l'objectif d'encourager la gestion par délégation, l'agent JMX supporte un service permettant de télécharger le code nécessaire afin d'instancier en son sein des objets gérés provenant d'un site distant. Ce service s'appelle M-Let acronyme de *Management Applet*.Il prend en compte une URL distante pointant sur un fichier texte. Ce fichier comprend les instructions nécessaires au chargement des classes requises ainsi qu'à l'instanciation des objets demandés. Ce fichier dont la structure est basée sur un langage d'étiquette à la HTML proche de la borne <APPLET> comprend pour chaque objet à instancier : son nom de classe, le nom de l'archive (fichier jar) dans laquelle se trouvent les fichiers compilés (.class) nécessaires, la localisation de cette archive (URL) et le nom JMX que l'on souhaite donner à l'instance.

La figure 6.10 illustre le mécanisme de chargement et d'instanciation d'objet géré à distance. Dans cet exemple, l'agent instancie, au travers du service M-Let, deux objets gérés comme cela est demandé dans le fichier `mesBeans.txt`. Celui-ci indique que l'agent doit instancier les objets gérés `newMB1` et `newMB2` respectivement de classe `MB1` et `MB2` et que les classes sont disponibles sur l'URL `http://www.loria.fr/~festor` dans le fichier `mesMBeans.jar`.

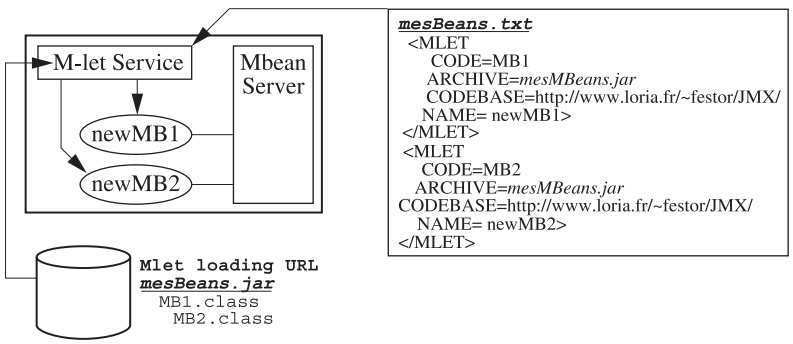

Figure 6.10. *Chargement d'objets gérés distants par le service Mlet*

Sur la base de ces informations, le service télécharge les classes[6], et instancie les deux objets dans le serveur d'objets avec les noms donnés en paramètre.

Le service de téléchargement à distance est un service obligatoire dans toute implantation d'agent conforme à l'architecture JMX.

6.5.7. *Le service de relations*

Le service de relations est le service le plus récent introduit dans le niveau agent de l'architecture JMX. Il permet de gérer des dépendances entre objets gérés au sein d'un agent. Le modèle de relations proposé est un modèle proche du modèle relationnel de CORBA. Il définit la notion de relation comme une entité nommée qui regroupe un ensemble de rôles. Un rôle est défini par une description, le type de classe des instances de MBeans qui peuvent prendre ce rôle et une cardinalité minimale et maximale.

Ce service offre les fonctions suivantes :

– création et suppression de types de relations,

– ajout et retrait d'instances de relations,

– méthodes d'interrogation et de parcours de relations : accès à tous les objets dans un rôle donné, découverte de toutes les relations dans lesquelles un objet participe, découverte de toutes les instances d'une relation donnée… ;

– accès à toutes les méta-données concernant les relations et les rôles (spécification des types, cardinalités…).

Le service assure également la maintenance des relations et émet des notifications (lui-même est un MBean) lorsque les caractéristiques d'une relation sont violées. Par exemple, la destruction d'un MBean qui participait à une relation peut entraîner une violation de la cardinalité d'un rôle et donc déclencher une notification.

6.5.8. *Enregistrement et destruction de MBeans*

Nous avons vu les différents services qu'offre un agent JMX. Pour donner une vision complète du fonctionnement d'un agent JMX, il nous reste à présenter les procédures d'enregistrement et de destruction d'objets gérés. Pour l'enregistrement et pour la suppression de MBeans, deux scenarii sont possibles. Le premier est celui où l'on demande à l'agent l'instanciation, resp. la destruction d'un objet géré. Dans ce cas, l'instanciation/destruction d'un *MBean* et le nommage/suppression du nom

6. JMX [1, §8][2, §2.11] définit finement les politiques de chargement à utiliser. L'usage d'un certain nombre de « ClassLoaders » est précisé.

d'*MBean* sont effectués de manière atomique. Le second cas est celui où un objet déjà existant souhaite s'enregistrer ou se désenregistrer du serveur. Pour cela le serveur offre les méthodes `register` respectivement `deregister` qui peuvent être invoquées par les objets eux-mêmes.

De plus, JMX offre un mécanisme de contrôle sur l'enregistrement ou le désenregistrement d'objets en proposant à ceux-ci d'implémenter des pré/post conditions pour ces opérations. Ces pré/post conditions sont matérialisées par des méthodes `pre/postRegister` et `pre/postDeregister` définies dans l'interface `MbeanRegistration`. Elles peuvent être programmées par le concepteur de l'objet géré. Elles renvoient toutes une valeur booléenne et sont invoquées par le serveur lors d'une opération d'enregistrement resp. de désenregistrement. Si, au cours d'une de ces phases, une de ces méthodes renvoie la valeur `false`, la procédure est arrêtée au niveau du serveur de MBeans.

6.6. *Les services transversaux*

Les services transversaux permettent d'offrir au développeur d'un agent basé sur l'architecture JMX, une panoplie d'interfaces de programmation pour accéder depuis les objets JMX à des ressources distantes *via* un protocole de gestion standard, par exemple SNMP, CMIP). Suivant les objectifs d'intégration, ces APIs fournissent une composante superviseur, agent ou les deux. Dans cette section, nous présentons les différents services existants ainsi que les travaux en cours au sein du consortium.

6.6.1. *SNMP*

SNMP a été la première approche pour laquelle des travaux d'intégration avec JMX ont été menés. Ceux-ci ont conduit à la création d'une interface de superviseur. Cette interface Java permet de manipuler des agents SNMP distants *via* le protocole SNMP. Elle offre un support Java pour tous les types ASN.1 définis pour SNMP, définit un modèle de session pour la communication avec ces agents et propose un modèle de traitement évolué pour les requêtes, réponses et traps.

Les objets de base de cette interface sont :

– `SNMPPeer` : Objet du superviseur représentant un agent distant (adresse IP, numéro de port de l'agent, taille maximale des PDU…) ;

– `SNMPSession` : Objet du superviseur qui représente une session avec un agent donné. Cet objet gère le type de communication (synchrone, asynchrone), offre un mécanisme sympathique pour le traitement des cas d'erreurs de communication. Il met également à disposition du programmeur une batterie de méthodes conviviales

pour la manipulation de requêtes et de listes de variables, pour le multiplexage/ démultiplexage et la segmentation ou le réassemblage de requêtes dans un but d'efficacité de traitement et d'économie maximale de bande passante ;

– `SNMPParameters` : Objet du superviseur qui comporte les informations de version et de sécurité pour les échanges avec un ou plusieurs agents ;

– `SNMPRequest` : Objet représentant une requête en cours dans une session. En plus des requêtes de base définies dans le protocole SNMP, l'API offre au sein d'une session des requêtes supplémentaires telles que des requêtes de polling périodique gérées par la session et des parcours de MIB conditionnels.

Pour le traitement des traps, l'API offre un mécanisme d'événements dans lequel un `EventReportDispatcher` instancié dans le superviseur va transmettre sous forme d'événements Java les traps qu'il reçoit des agents vers les objets Java qui s'y sont abonnés au travers d'un *listener* spécifique.

L'API de service SNMP est aujourd'hui très complète. Elle supporte notamment les version 1 et 2 du protocole SNMP. Cependant seul le côté superviseur existe à ce jour ce qui semble normal dans l'optique d'une intégration par JMX. Cette interface n'est pas actuellement disponible dans l'implantation de référence de JMX mais disponible dans plusieurs produits conformes à l'architecture JMX (voir section implantations).

Des travaux sur le côté agent sont en cours au sein du consortium pour permettre à un agent JMX d'exhiber certains de ces objets à un gestionnaire SNMP. Ceux-ci devraient aboutir très rapidement à des propositions et à une implantation de référence.

6.6.2. *WBEM*

Initialement lancé comme groupe de travail au sein de JMX, le succès croissant de WBEM et le champ d'application plus large que simplement celui de l'intégration ont mené le groupe de travail chargé de traiter WBEM à se regrouper au sein d'un JSR[7] spécifique appelé WBEM Services. En plus des membres de JMX, les entreprises ou consortiums suivants contribuent à ce groupe : *Caldera*, *CISCO*, *Compaq*, *Lucent Technologies*, *Nokia*, *SNIA*[8], *Progress Software*, et *Veritas*. Dans ce groupe, l'activité autour de WBEM est double. D'une part les efforts visent à fournir une interface de programmation permettant à des agents JMX d'interopérer avec des agents WBEM au même titre que SNMP. D'autre part, ce groupe a pour objectif de définir une

7. *Java Specification Request* : groupe de travail au sein du processus de communauté Java.
8. *Storage Network Industry Association* est un consortium qui regroupe plus de 100 industriels. Son but est de développer des standard et des implantations de référence pour l'industrie de la sauvegarde et du stockage de données au travers des réseaux.

architecture et de réaliser un environnement complet autour des services WBEM, notamment un agent CIMOM (voir chapitre sur WBEM) ce qui est en dehors des objectifs de JMX et qui justifie la création du JSR spécifique à WBEM.

Les travaux de ce JSR qui intéressent directement JMX et auxquels des experts de JMX participent sont ceux relatifs aux interfaces de programmation d'accès à un CIMOM ainsi que les interfaces d'accès à un *provider* (agent JMX) depuis un CIMOM. Ces travaux ont déjà abouti à une première spécification Java. Celle-ci définit l'API d'accès à un CIMOM. Elle permet à un agent JMX au travers de MBeans d'interroger un agent WBEM.

Cette interface est aujourd'hui composée de trois paquetages : un paquetage pour les types génériques utilisés pour la manipulation d'objets CIM, un paquetage qui définit les objets Java à utiliser pour réaliser un client CIM et un paquetage qui permet la réalisation d'un provider CIMOM. Le paquetage générique comporte l'ensemble des objets Java qui représentent des composants CIM à savoir les classes, attributs, espaces de nommage, propriétés, qualifieurs, méthodes et types CIM (voir chapitre WBEM/CIM). L'interface client offre un ensemble de méthodes pour accéder à un agent CIM suivant les opérations définies dans le standard (création/suppression de classe et d'instances, invocation de méthodes, lecture de spécifications CIM, d'attributs d'instances, parcours de MIB...). Un support pour la gestion de connexion avec des agents CIM est également fourni dans ce paquetage. Le dernier paquetage permet d'enregistrer des objets sur un CIMOM et de réaliser des opérations sur ces objets. Trois types de providers sont supportés : des fournisseurs d'instances, de méthodes ou de propriétés. En fonction du type de fournisseur, les méthodes correspondantes doivent être implantées.

Cette interface est aujourd'hui extrêmement complète, mais reste fidèle à la norme du DMTF ce qui n'est pas sans poser quelques problèmes. En effet cette dernière est encore mal spécifiée et comporte encore de nombreuses erreurs. Celles-ci sont détaillées dans les chapitres dédiés à WBEM. Du point de vue réalisation, cette interface est aujourd'hui implantée dans l'environnement WBEM de SUN Microsystems. Elle est également reprise dans des implantations tierces comme celle du SNIA par exemple. Le code de celle-ci n'est à ce jour pas disponible dans l'implantation de référence de JMX.

6.6.2. *CMIS*

Les efforts sur l'intégration CMIS/JMX sont développés au sein d'un groupe de travail spécifique de JMX. A ce jour, un jeu d'interfaces de programmation aux deux niveaux est utilisable dans un contexte JMX. Les deux niveaux proposés sont un niveau d'objets gérés et un niveau protocolaire CMIS/ASN.1.

Le premier permet aux développeurs de ne manipuler que des objets gérés OSI sans se préoccuper de la couche protocolaire CMIS sous-jacente. Ce niveau de programmation est proche de l'API TMN++ développée en C++ au sein du *Tele Management Forum* (TMF[9]) mais reformulée et reprogrammée en Java. Elle offre un côté gestionnaire et un côté agent.

Une alternative à cette approche de programmation est définie sous forme d'une API duale Superviseur/Agent permet aux développeurs de manipuler en Java directement une pile OSI ainsi que les primitives de service CMIS. Elle comprend une abstraction de pile qui fonctionne comme une *factory* de requêtes. Elle dispose d'un mécanisme d'évènements identique à celui de l'AWT de Java (Event/Listener) pour la réception d'indications et de confirmations. Cette abstraction CMIS et l'interface Java associée sont indépendantes de toute pile commerciale sous-jacente (elle est disponible sur plusieurs implantations) et dispose de mécanismes de transport des requêtes/réponses CMIS alternatifs tels que RMI ou HTTP. Cette interface a été réalisée par les chercheurs de l'INRIA Lorraine en coopération avec les ingénieurs de la division OpenMaster de BullSoft. Elle est disponible sur la plate-forme OpenMaster et une implantation utilisant le transport RMI de Java est disponible en OpenSource[10]. L'ensemble des spécifications et des exemples sont fournies dans la distribution. Une présentation détaillée de l'interface et de sa mise en œuvre est disponible dans [SUN 02, SUN 03].

Aucune des deux interfaces Java existantes pour CMIS n'est à ce jour standardisée dans le cadre du jsr.

6.6.3. *Autres services transversaux*

JMX est ouvert à l'intégration d'autres services d'accès aux informations de supervision. Récemment un nouveau groupe autour de CORBA s'est créé dans le JSR. Celui-ci vise d'une part à développer un adaptateur de protocole IIOP pour les agents JMX et, d'autre part, à développer des propositions communes avec le groupe de travail de l'OMG11 sur les besoins et les approches de supervision des applications distribuées. L'adaptateur IIOP permettra à des plates-formes spécifiques (non compatibles JMX) d'accéder à des agents JMX *via* le protocole IIOP et au travers de la définition d'une interface standard pour la gestion d'applications distribuées en IDL, les applications de supervision JMX pourront soit au travers d'un agent JMX, soit directement superviser des ressources accessibles *via* CORBA.

En résumé, cette interface transversale a un objectif triple : offrir une intégration des agents JMX dans des plates-formes de supervision CORBA (interface haute ou

9. http://www.tmforum.org
10. http://www.loria.fr/~festor/JTMN/
11. http://www.omg.org

agent), permettre à des agents et à des superviseurs JMX d'accéder à des ressources CORBA (interface basse ou superviseur) et finalement de définir un modèle IDL fédérateur pour la supervision d'applications distribuées.

6.6.4. *Synthèse des interfaces transversales*

Comme nous l'avons vu dans les différentes sections précédentes, il existe aujourd'hui un nombre important d'interfaces dites transversales permettant à la technologie JMX de s'ouvrir ou plutôt d'intégrer d'autres approches de supervision. La figure 6.11, replace les différents composants dans l'architecture de supervision (Superviseur, Agent, Instrumentation) afin d'illustrer les différents endroits dans lesquels ces interfaces sont utiles, à savoir à la frontière haute de l'agent sous forme d'adaptateur de protocole, ou à la frontière basse permettant à des *MBeans* d'agir comme proxys vers des agents standards.

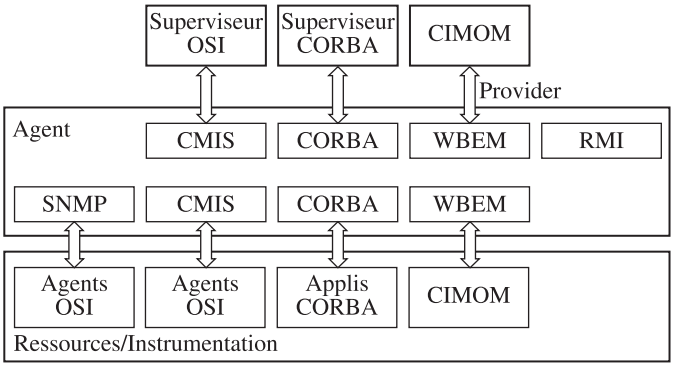

Figure 6.11. *Interopérabilité JMX / approches standard*

Cette illustration démontre parfaitement l'ouverture de la technologie JMX vers le monde de la supervision standard. En effet, un agent JMX est capable au travers de ces interfaces d'accéder à l'ensemble des agents du marché basés sur une approche de supervision standard (interfaces basses de l'agent JMX). De plus, JMX permet à ses agents de s'intégrer avec la plupart des systèmes de supervision existants aux travers d'adaptateurs de protocoles (interfaces hautes de l'agent). Nous n'avons pas mis dans cette interface haute la liaison vers un superviseur SNMP car celle-ci n'est pas disponible à ce jour dans JMX. On peut noter également que les interfaces transversales dans un rôle superviseur (interfaces basses de l'agent sur la figure) peuvent également être utilisées pour la conception de superviseurs qui ne sont pas forcément des agents JMX.

6.7. Limites de l'architecture

Nous avons identifié à ce jour trois limites à cette architecture en pleine évolution. La première est l'absence de support explicite pour la persistance d'objets gérés dans un serveur. Si celle-ci est abordée dans le document de spécification, aucune spécification précise d'un service support n'est proposée et rien n'est actuellement fourni dans l'implantation de référence. Ceci peut être contourné en couplant lors de la conception d'un agent, les objets gérés avec un mécanisme de persistance orthogonale ou en exploitant soi-même la sérialisation Java par exemple. L'utilisation de ces technologies reste dans ce cas, malheureusement à la charge du concepteur d'agent. Cette limitation actuelle devrait disparaître dans les implantations futures de JMX ou la persistance doit être offerte, au moins pour les MBeans modèles.

Le second point qui nous apparaît limité dans l'architecture d'agent est le mécanisme de notification offert. En effet, celui-ci se base sur un mécanisme d'abonnement à la source qui est très lourd à gérer dans certaines situations. Par exemple, il est très complexe de mettre en place un objet géré dont la tâche est de journaliser par exemple des notifications de type Alarme quels que soient les objets qui les émettent. Pour cela, il faut actuellement que cet objet s'abonne systématiquement à tous les objets susceptibles d'émettre de telles notifications et le maintien de la cohérence avec les sources potentielles est difficile. Pour résoudre ce problème, un service générique de notifications pourrait être fourni dans l'agent. Tout objet pourrait souscrire à ce service en s'abonnant à un type de notification et non plus à une source. Les sources de notifications pourraient elles-mêmes émettre toutes leurs notifications par défaut vers ce service. Un tel mécanisme pourrait être mis en place très facilement en étendant l'interface du NotificationEmitter et en insérant le service comme MBean directement dans le serveur d'objets par exemple.

La troisième critique que nous pouvons formuler sur l'architecture d'agent est relative au choix de l'ordre lexicographique pour la sémantique des noms d'objets. En effet, ce choix certes simple, ne permet pas naturellement de représenter des liens de contenance entre objets comme cela est défini dans l'approche OSI, ni de hiérarchie de type système de fichier comme cela est défini dans WBEM. En conséquence, le *mapping* d'autres approches vers JMX est rendu plus délicat. On peut cependant utiliser le service de relations défini dans JMX pour reconstituer au sein de l'agent une organisation hiérarchique. Mais ce mécanisme est plus lourd car l'accès aux objets ne se fait plus que par navigation dans les relations ajoutant un niveau d'indirection pas forcément nécessaire. D'une manière générale, la tendance actuelle tend à fédérer des ensembles d'agents. Une mise en œuvre étendue de JMX nécessite donc l'adoption de convention de nommage stricte au sein d'un MBeanServer. Mais il faut aller au-delà : [SUN 03] fournit les moyens d'enregistrer les MBeanServers dans des systèmes d'annuaires (LDAP, JINI, SLP…) ; les développements devront aussi adopter des conventions de nommage homogènes avec le niveau JMX pour des entrées dans ces répertoires. Nous devons donc nous attendre à voir apparaître nombre de pratiques divergentes.

S'ajoute à ces limites, une interrogation ; JMX ne dispose pas d'un langage de spécification des interfaces de gestion. C'est l'un de ses avantages car tout est Java et Java est le seul langage à connaître pour faire de la supervision. Cependant, nous nous nous demandons si il est réaliste d'envisager qu'une interface Java ou qu'un *MBeanInfo* puisse être une base de standardisation et de discussion entre des personnes d'horizons différents ne connaissant pas forcément la technologie Java et dont les documents se veulent indépendants de toute technologie. La réponse est probablement non. Cependant, il est parfaitement possible de prendre un langage de spécification d'informations de gestion existant (GDMO, IDL, …) et de fournir un compilateur capable de générer des *MBeans*.

En résumé, l'architecture de l'agent et le niveau d'instrumentation de JMX ont très peu d'inconvénients. Les services s'enrichissent au fur et à mesure des avancées des travaux sans cependant alourdir trop fortement l'environnement. Les limites identifiées seront certainement comblées dans les versions futures de l'architecture fournissant ainsi un environnement quasi parfait.

6.8. Les implantations de JMX

6.8.1. *Références SUN*

La première implantation conforme à l'architecture JMX est l'implantation de référence liée au jsr 03 disponible sur le site de *Sun Microsystems*. Celle-ci comporte le code relatif au serveur d'objets gérés. Il supporte les *MBeans* standards, dynamiques, modèles et ouverts et implante la plupart des services d'agents (timer, monitoring, relations…). Cette l'implantation est complétée par l'implantation de référence liée au jsr 160. Le couplage des deux permet donc à un manager distant d'accéder à toutes les facilités obligatoires de JMX au travers d'un connecteur utilisant RMI (*Remote Method Invocation*, l'appel à distance de Java) comme transport. Le principal point noir de l'implantation reste cependant à ce jour, la non-disponibilité des services transversaux, qui existent dans des implantations produits de l'architecture mais pas dans la référence, ce qui est regrettable. De nombreux exemples sont fournis avec cette implantation qui forme, malgré les absences, une excellente base pour l'évaluation de la technologie. Il faut noter que ces implantations ne sont pas du logiciel libre. Les contraintes d'utilisation et d'accès au code sont souples, mais restent sous licence particulière de SUN.

6.8.2. *Implantations « libres »*

MX4J est une implantation libre de JMX sous licence de type Apache (http://mx4j.sourceforge.net). Elle fournit un support correct de JMX 1.1. JBossMX (http://www.jboss.org) en est une autre sous licence LGPL. Cette dernière semble implanter partiellement JMX 1.2. La particularité de JBossMX est qu'à l'origine le

groupe JBoss a développé ce support JMX car il a fait le choix d'utiliser les MBeans comme *modèles de composants* logiciels pour la réalisation de leur offre principale : le serveur j2ee JBoss, une idée géniale à la base.

6.8.3. *Implantations commerciales*

En plus de son implantation de référence *Sun Microsystems* commercialise un produit intitulé JDMK[12] (*Java Dynamic Management Kit*) qui respecte les interfaces et l'architecture JMX. Ce produit SUN est en réalité une implantation beaucoup plus complète que l'implantation de référence de JMX. Elle comporte notamment plus d'adaptateurs de protocoles et d'outils. Par exemple, JDMK fournit un environnement pour créer un connecteur Java à l'aide d'un compilateur (ProxGen) qui génère une interface Java d'un MBean pour le côté manager. L'infrastructure SNMP est également fournie notamment au travers d'un compilateur (MIBGen) qui génère des MBeans à partir de spécifications SNMPv1, v2 & v3 ainsi qu'un mapping de noms symboliques vers des identificateurs d'objets SNMP (OID en *dotted string notation*). JDMK fourni également un adaptateur HTML. Celui-ci offre l'accès à un agent JDMK depuis n'importe quel navigateur Web au travers de *JavaScript*. L'adaptateur fonctionne comme un serveur Web qui génère des pages HTML décrivant les objets de la MIB et permet, au travers de formes dynamiques, d'administrer l'agent (affecter des variables, invoquer des méthodes offertes par les MBeans à l'interface de supervision). Les types des attributs et paramètres supportés par l'adaptateur sont cependant limités aux types de base de Java, aux tableaux de types de base ainsi que certains types spécifiques de JMX tel que `ObjectName`. Une version d'évaluation de ce produit est téléchargeable depuis le site de *Sun*.

En février 2001, Tivoli a produit une implantation complète de JMX appelée tmx4j. Cette implantation est téléchargeable depuis le site d'Alphaworks[13]. Cette réalisation implémente toutes les fonctionnalités de la spécification JMX (tous les services de base y compris le chargement dynamique de MBeans au travers du service Mlet. Extrêmement complet et bien documenté, cette implantation fournit en plus du core JMX, des démons JMX pour NT et Unix qui hébergent un agent JMX. Un connecteur RMI ainsi qu'un adaptateur http sont également fournis dans l'implémentation. La distribution comprend également un excellent tutoriel.

6.9. Utilisations de JMX

JMX commence réellement à être utilisé. Principalement pour son but initial : l'instrumentation d'applications Java.

---

12. http://www.sun.com/software/java-dynamic/
13. http://www.alphaworks.ibm.com

6.9.1. *J2ee*

Pratiquement tous les serveurs d'applications j2ee présentent une administration accessible *via* JMX. Ceci n'est pas étonnant, j2ee définit un cadre pour son administration dans [HRA 02] et JMX est le support de choix pour l'implanter.

6.9.2. *Autres applications*

La « *apache fondation* » (www.apache.org) promoteur du célèbre serveur web semble vouloir utiliser JMX pour certains de ses projets : Tomcat (serveur de servlet, une extension dynamique de serveur web en Java), James (serveur de courrier électronique en Java). JMX est également utilisé dans plusieurs toolkits agent et pour l'instrumentation de nombreuses applications. Le site de référence de JMX maintient la liste exhaustive des applications instrumentées par JMX.

6.10. Conclusion

Au travers de l'architecture et des interfaces JMX, la technologie Java démontre parfaitement son adéquation aux besoins des architectures de supervision. JMX innove fortement dans le monde de la supervision en ne prônant plus de langage de spécification pour les modèles de l'information et en offrant une architecture d'agent souple et extensible fondée sur la technologie Java. Ce choix technologique est de notre point de vue un avantage énorme car il assure une pérennité à la solution en ne se soumettant pas à une approche de gestion donnée mais en permettant son adaptation à toutes les nouvelles modélisations qui émergent comme WBEM par exemple.

Comme nous avons pu le relever tout au long de ce chapitre, de nombreux concepts retenus dans JMX sont fortement inspirés de la gestion OSI. JMX a réussi à les simplifier au maximum sans cependant les restreindre trop fortement et a su mieux les mettre en œuvre d'un point de vue technologique, ceci de façon relativement simple. Elle hérite de l'approche OSI toutes les caractéristiques qui en ont fait l'approche la plus complète en lui ajoutant en plus le support de la dynamicité requis aujourd'hui.

Initialement orientée vers la supervision des applications Java pour laquelle JMX offre une palette extrêmement riche de composants, cette technologie s'ouvre au travers des interfaces transversales à la gestion de systèmes standards et permet à celle-ci de bénéficier de son architecture novatrice. Mais surtout, l'architecture permet à des approches standards de bénéficier de toute la richesse et la force des composants Java dans le domaine de la distribution (bus logiciels), de la composition (outils autour des *Beans*) et de l'accès à l'information (annuaires, bases de données) disponibles pour cette technologie.

Les apports de cette architecture sont multiples. Le modèle à base de composants est très utile et favorise la gestion par délégation. Celle-ci est amplifiée par le support de chargement dynamique à distance d'objets gérés dans l'agent. Le concept d'objet géré étendu à celui d'adaptateur de protocoles permettant à un agent d'offrir dynamiquement plusieurs interfaces de communications garantit une ouverture aux développeurs d'objets et d'agents. Finalement, les quatre types de *MBeans* et les mécanismes intégrés dans le serveur d'objets offrent une approche extrêmement facile à mettre en œuvre pour instrumenter des applications Java mais également grâce aux interfaces de programmation de services de supervision transversales, tout autre composant distant. Cette simplicité facilite énormément l'acquisition des concepts et l'utilisation de ceux-ci dans des applications.

Aujourd'hui JMX est dans une phase de croissance très forte. Preuve en sont la forte croissance du nombre d'industriels qui y contribuent, l'élargissement des champs d'application (J2EE, J2SE, J2ME[14]) ainsi que les efforts d'intégration (WBEM, SNMP, TMN, CORBA). Si l'architecture de l'agent est très plaisante, le grand défi que JMX se doit de relever pour s'imposer réellement comme standard dans l'industrie est la mise à disposition d'interfaces de très grande qualité pour les services de supervision côté gestionnaire. Le jsr 160 est une très bonne base de départ pour cela.

Finalement, les efforts entrepris pour l'instrumentation par JMX de l'ensemble des composants disponibles aujourd'hui dans le standards Java (tous les composants du framework Java disposeront à court terme de *MBeans* pour les superviser) font de cette approche, une solution incontournable pour la supervision Java.

6.11. Pour en savoir plus

JMX possède une page Web dédiée à l'adresse suivante :

http://java.sun.com/products/JavaManagement/

Le lecteur y trouvera la spécification complète (actuellement 170 pages) ainsi qu'un résumé stratégique et technique d'une vingtaine de pages. De plus, l'implantation de référence est également téléchargeable depuis ce site. Elle est accompagnée de toute la documentation technique des classes sous forme de *Javadoc*. Les discussions autour de cette approche sont très actives et les participants très réactifs aux problèmes éventuels rencontrés par les développeurs.

14. L'intégration dans J2ME pose des problèmes intéressants liés notamment aux restrictions de mémoire, de CPU dans les équipements légers ainsi que le non-support de l'introspection et de la sérialisation dans la machine virtuelle pour ces équipements. Pour J2SE et J2EE, l'intégration consiste principalement à définir des MBeans et de fournir l'instrumentation pour l'ensemble des composants de ces plates-formes.

Ces discussions ont lieu sur la liste de diffusion dédiée : jmx-spec-comments@sun.com.

Finalement, le lecteur intéressé trouvera sur la page de l'auteur (http://www.loria.fr/~festor) l'ensemble des supports de cours (transparents, TD, sujets de TP et d'examen) liés à cette technologie. Celle-ci est enseignée en Maîtrise d'IUP Réseaux numériques de communication de l'Université Henri Poincaré Nancy I ainsi que dans le module de supervision de réseaux de la troisième année option télécoms de l'école supérieure d'informatique et applications de lorraine (ESIAL).

En plus des différents jsr cités et des documentations des implémentations testées nous avons particulièrement apprécié deux livres. L'un [KRE 02] pour un recul certains sur les pratiques en instrumentation Java, et l'autre [FLE 02] pour sa présentation claire de JMX.

6.12. Bibliographie

[SUN 02] SUN, *Java™ Management Extensions, Instrumentation and Agent Specification, v1.2*, http://jcp.org/en/jsr/detail?id=3, Maintenance Release, octobre 2002.

[SUN 03] SUN, *Java™ Management Extensions(JMX™) Remote API 1.0 Specificatio*n, http://www.jcp.org/en/jsr/detail?id=160, Proposed Final Draft June, juin 2003.

[HRA 02] HRASNA H., *Java™ 2 Platform, Enterprise Edition Management Specification JSR-77*, http://www.jcp.org/en/jsr/detail?id=77, Final Release v1.0, 18 juin 2002.

[KRE 02] KREGER H., WARD Harold K., WILLIAMSON L., *Java and JMX: Building Manageable Systems, Addison Wesley Professional*; ISBN : 0672324083, 1re édition décembre 2002.

[FLE 02] FLEURY M. *et al.*, *Jmx: Managing J2ee Applications with Java Management Extensions*, Sams, ISBN : 0672322889, 1re édition janvier 2002.