

**ECOLE DES MINES DE NANTES
DEPARTEMENT INFORMATIQUE
EQUIPE OBJET COMPOSANT MODELE**

Adaptabilité dynamique des services dans JOnAS (Java™ Open Application Server)

Présenté par
Zahi JARIR

Encadré par
Thomas LEDOUX

Résumé :

L'objectif que nous poursuivons dans le cadre de ce travail consiste à adapter dynamiquement les services associés aux applications à base de composants de type EJB. La plate-forme utilisée est JOnAS, un serveur d'applications open source à base de composants compatible avec les spécifications EJB 1.1. L'adaptabilité se fait en réponse à des changements dans le contexte d'exécution. Dans cette optique, nous introduisons la notion de politiques d'adaptation présentée au cours d'un travail de DEA [1]. Ces politiques sont utilisées pour configurer les associations entre les différents composants et les services en indiquant à JOnAS d'une part quelles sont les modifications de l'environnement à prendre en compte, et d'autre part comment réagir à ces modifications de façon appropriée.

Sommaire

1. Introduction
2. Présentation de JOnAS
3. Présentation d'une infrastructure pour middleware adaptable
4. Etude de l'adaptabilité dynamique des services dans JOnAS
5. Conclusions et perspectives

1 Introduction

Actuellement, le développement technologique des systèmes informatiques est en croissance progressive. Cette évolution, qui engendre la naissance d'une grande diversité de plates-formes d'exécution et d'environnement logiciels, rend la tâche des développeurs de plus en plus difficile. En outre, la nature de ces plates-formes se caractérise parfois par un changement dynamique du contexte d'exécution des applications (par exemple bande passante variable dans les applications de commerce électronique sur Internet).

Ainsi les développeurs se trouvent en face d'un grand défi qui leur oblige à mettre en place des méthodologies permettant de programmer des applications adaptables en tenant compte des différentes contraintes qui sont à la fois statique (liées aux plates-formes matérielles) et dynamique (liées à la disponibilité des ressources variables).

C'est dans le cadre de cette problématique que notre travail se situe. La solution que nous proposons consiste à mettre en place une infrastructure permettant d'écrire des applications adaptables vis à vis de leurs contextes d'exécution. Cette infrastructure est basée d'une part sur une infrastructure pour middleware adaptable [1], d'autre part sur la plate-forme JOnAS (Java TM Open Application Server) du consortium *Objectweb*, à base de composants.

L'organisation de notre rapport est présentée comme suit. La seconde section décrit la plate-forme JOnAS et ses spécificités. La troisième section présente brièvement une infrastructure pour middleware adaptable [1] basée sur le MOP (Meta-Object Protocol) RAM [2]. Enfin, la dernière section propose notre infrastructure qui se caractérise par le fait de conjuguer les richesses associées aux deux plates-formes citées .

2 Présentation de JOnAS

2.1 La technologie Entreprise Java Entreprise

Le modèle Entreprise Java Beans (EJB) est une spécification d'environnements de SUN Microsystems. Ce modèle est considéré comme un standard assez largement accepté dans le domaine de la programmation de serveurs d'applications réparties à base de composants. Le but de ce modèle est de simplifier le développement en le focalisant sur le côté fonctionnel de l'application et cela indépendamment de tout système. En outre, il fournit aux applications des facilités pour leur développement, leur déploiement, ainsi que leur maintenance.

La communication inter-composant se fait via RMI. Les problèmes de sécurité, d'accès concurrents et de persistance sont gérés par la plate-forme en offrant les services utiles tel que les services de nommage JNDI (Java Naming and Directory Interface), de sécurité (Java Security), gestion de transaction, de persistance, etc.

L'un des points forts du modèle EJB est qu'il partage les responsabilités vis à vis des applications en trois types d'entités : le serveur EJB, le conteneur (container) et l'Entreprise Java Bean.

a. Serveurs EJB

Les serveurs contiennent et exécutent un ou plusieurs EJB. Ces EJB ne communiquent pas directement avec le serveur d'EJB ni entre eux mais c'est à travers un médiateur appelé conteneur (Container).

b. Conteneur

Le conteneur EJB, en plus du rôle de médiateur entre le serveur EJB, le client du bean et d'autres EJB, fournit au bean qu'il représente de nombreux services de bas niveaux tels que la prise en charge de transactions, la gestion de stockage des données et leur extraction, etc.

c. Entreprise Java Bean

Les spécifications EJB distinguent les composants décrivant les procédures et les entités métier, et proposent trois types de beans correspondants :

- Les *session beans* modélisent des procédures métier, invoquées par les clients par invocation de méthodes à distance RMI,
- Les *entity beans* modélisent des entités métier persistantes, manipulées par RMI, et servent à spécifier les applications dans le point de vue information,
- Les *message-driven beans* modélisent des procédures métier, invoquées par les clients par envoi asynchrone de messages (signaux).

La persistance modélisant les entity beans sont de deux types :

- *Bean-Managed Persistence*, où le fournisseur de bean doit développer la méthode d'accès à la base de données en utilisant l'interface JDBC.
- *Container-Managed Persistence*, dont l'accès à la base et le stockage sont automatiquement géré par l'environnement EJB. Le fournisseur n'a qu'à spécifier un descripteur de bean contenant les informations à propos du mapping des champs dans le schème de la base.

Un EJB est constitué des éléments suivantes :

- **Classe du bean**, qui encapsule les données associées au bean et contient les méthodes métiers implémentées par le développeur qui permettent d'accéder à ces données. Elle renferme aussi les différentes méthodes utilisées par le conteneur pour gérer le cycle de vie de ses instances.
- **Interface Home**, qui définit les méthodes utilisées par le client pour créer, rechercher, et supprimer des instances du bean. Elle est mise en œuvre par le conteneur lors du déploiement dans une classe appelé EJB Home.
- **Interface Remote**, qui est mise en œuvre par le conteneur lors du déploiement dans une classe appelée EJB Object. Le client du bean après avoir utilisé l'interface EJB Home pour accéder au bean, utilise cette interface pour appeler indirectement les méthodes métiers implémentées dans la classe bean.

- **Classe Primary Key**, qui est défini seulement pour un entity bean. Elle enferme une ou plusieurs variables qui identifient de manière unique une instance de bean d'entité spécifique. Cette classe contient en plus les méthodes qui permettent de gérer des objets Primary Key.

Le scénario d'un appel d'une méthode métier d'un bean par un client est illustré par la figure 1.

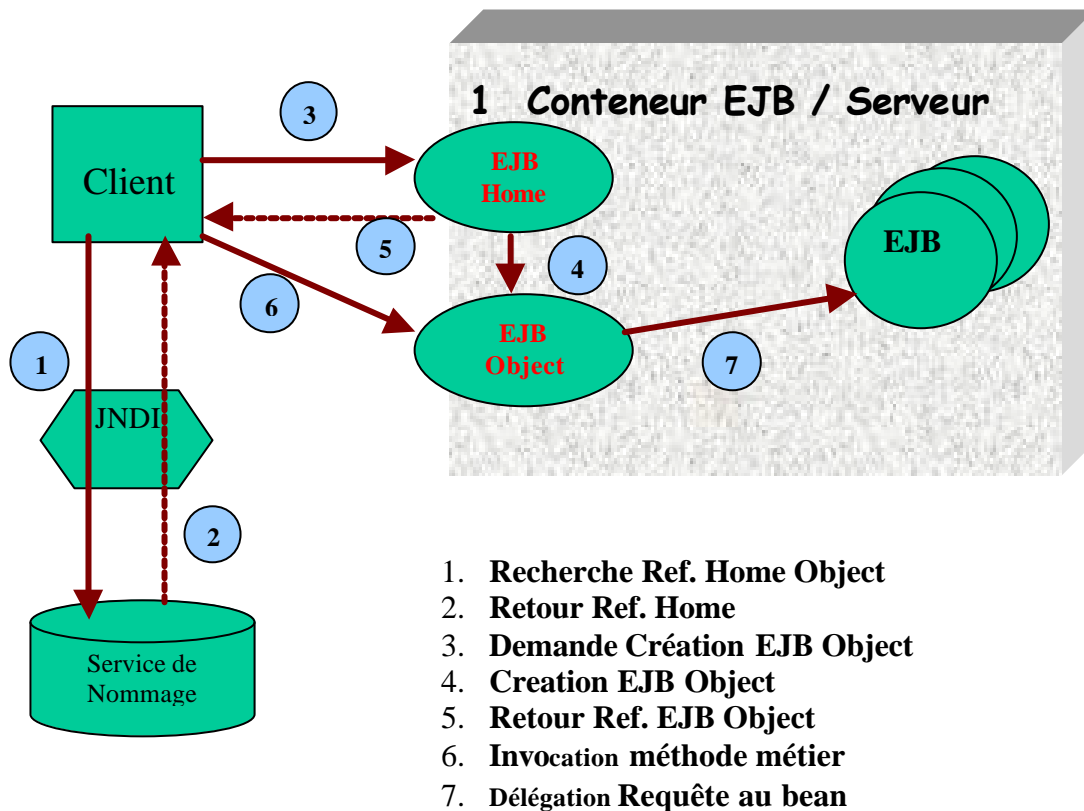


Figure 1 - Scénario relatif à l'appel d'une méthode bean par un client

d. Spécification des aspects techniques relative aux EJB

Les spécifications EJB, comme cela a été indiquées, mettent l'accent sur la séparation du code des composants (le code métier), de leur assemblage et de leurs aspects techniques. Ces aspects techniques sont décrits de manière déclarative dans un descripteur de déploiement, et sont mis en oeuvre par un support d'exécution spécifique, le conteneur EJB.

Ce descripteur, représenté par un fichier XML (eXtensible Markup Language), est utilisée dans deux contextes :

- pour chaque bean : pour spécifier les propriétés techniques du composants.

- Pour l'application entière : pour spécifier l'assemblage des composants, et les propriétés globales de l'application, comme le type de persistance, les rôles de sécurité, etc.

L'un des principaux avantages des EJB est leurs capacités à gérer des transactions d'une manière déclarative, et non pas codé dans l'application serveur, au moment de déploiement. C'est ce qu'on appelle le «Container-managed transaction demarcation». Le comportement d'une transaction est défini au moment de la configuration, et dans le descripteur de déploiement du bean.

2.2 La plate-forme JOnAS

JOnAS, développé par la société française Evidian (anciennement BullSoft), est un serveur d'application open source compatible avec les spécifications EJB 1.1. Il s'inscrit dans une initiative dénommée ObjectWeb en faveur du logiciel open source de type middleware.

JOnAS intègre actuellement les spécifications *JMS* (*Java Messaging System*) qui permettent aux EJB d'échanger des messages d'une manière asynchrone à travers un module dénommé *JORAM* (*Java Open Reliable Asynchronous Messaging*). Les transactions distribuées sont aussi gérées, par l'intermédiaire d'un coordinateur implémentant le standard *JTA* (*Java Transaction API*).

Cette plate-forme fournit aussi plusieurs outils tel que le générateur d'EJB, un module de mapping objet/relationnel relatif aux bases de données, etc.

JOnAS est construit sur la base de *JTM* (*Java Transaction Manager*) qui gère les transactions distribués. Ainsi les transactions peuvent impliquer plusieurs beans situés sur des serveurs EJB différents.

Quant au service de sécurité, JOnAS utilise TOMCAT pour l'identification et l'authentification de l'utilisateur. Cependant, cette sécurité est disponible seulement si le client est un servlet dans TOMCAT.

3 Brève présentation d'une infrastructure pour middleware adaptable

Cette infrastructure a été développée au cours d'un sujet de DEA [1]. Ce travail a permis de concevoir et développer une infrastructure pour le middleware adaptable, capable d'adapter de façon la plus transparente possible (pour le programmeur d'application) les applications à des conditions changeantes d'exécution, aussi bien statiquement (configuration au déploiement) que dynamiquement (reconfiguration automatique à l'exécution).

Cette infrastructure repose sur un protocole à métaobjets de type runtime nommé RAM (la sigle ??? ??()[2]). Ce protocole a permis de séparer le code fonctionnel des services non-fonctionnels des applications et de modifier dynamiquement leurs associations. Le point fort de cette infrastructure est basé sur le fait qu'elle fait appel à des *politiques*

d'adaptation. Ces politiques décrivent quand et comment effectuer ces configurations à la volée.

Ces politiques peuvent avoir différentes formes mais leur rôle est d'indiquer au middleware deux informations :

- Quelles sont les variations de l'environnement qui ont un impact sur les applications en cours d'exécution.
- Quelles mesures il doit prendre pour adapter ces applications aux nouvelles conditions.

Plus précisément, ce système repose sur les trois principaux concepts d'adaptabilité qui sont :

- **Observation** de l'environnement et du système lui-même.
- **Décision** de déclencher une adaptation
- **Action** pour modifier le système.

L'adaptation est réalisée par une entité appelée le *moteur d'adaptation*. Les modifications que ce dernier est capable de réaliser consiste à reconfigurer les associations entre les composants fonctionnels formant l'application et les composants non-fonctionnels qui représentent les services. Plus précisément, le moteur d'adaptation attache, détache ou reconfigure dynamiquement des services non-fonctionnels aux composants fonctionnels constituant l'application en question. De cette manière, le système modifie dynamiquement le comportement relatif à ces divers composants fonctionnels.

L'approche retenue pour effectuer cette adaptation est basée sur les techniques de réflexion. Plus précisément, l'infrastructure se retrouve avec deux niveaux, un premier niveau, dit de base, qui englobe les composants fonctionnels alors que le second niveau, dit méta niveau, représente les services non-fonctionnels. Ainsi et grâce aux techniques réflexives, le système se trouve capable de modifier dynamiquement le comportement des composants fonctionnels durant l'exécution de l'application.

La figure 2 suivante illustre les différents composants cités auparavant.

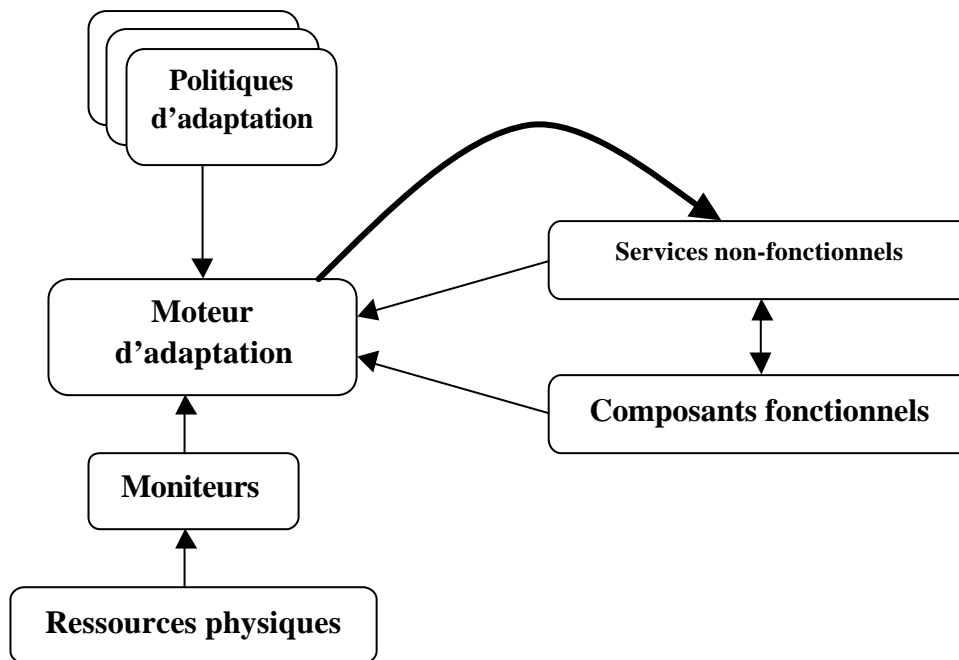


Figure 2 – Décomposition fonctionnelle de l'infrastructure pour middleware adaptable.

4 Etude de l'adaptabilité dynamique des services dans JOnAS

L'étude menée dans cette section consiste à intégrer les politiques d'adaptation de l'infrastructure présentée à la section précédente au sein de JOnAS. Ainsi la plate-forme JOnAS aura la capacité d'adapter dynamiquement des services aux profit des applications à base de composants selon les changements de leurs contextes d'exécution.

Vu d'une part notre objectif principal qui est de réaliser cette intégration de politiques d'adaptation et d'autre part le caractère «réflexif» du service d'assemblage dynamique de service pour l'adaptabilité dynamique, nous avons dans un premier temps réaliser nos propres services¹ «maisons» pour valider l'infrastructure modulaire et adaptable. Ce choix d'utiliser ce type de service est lié aussi à la nature des services prédéfinis de JOnAS qui sont implémentés dans un même bout de code et imbriqués les uns dans les autres[3,4]. Ces services se passent successivement la main par des fonctions non standardisées et leurs entrelacements est par suite complexe. Cela rend l'ordre de leurs utilisations déjà préétablis et donc leurs compositions très complexes. Ainsi pour ne pas être confronté à ce problème de composition de services, nous avons décidé de mettre en place des services maisons.

Comme les travaux présentés en [1] sont basés sur une infrastructure adaptable possédant un couplage lâche entre les différents modules décrits comme suit :

- a. Le moteur d'adaptation des politiques
- b. L'infrastructure d'observation des ressources
- c. Les composants fonctionnels des applications

¹ La réutilisation des services préfinis de JOnAS fera l'objet de nos futurs travaux.

- d. Les composants non fonctionnels représentant les services systèmes contrôlés par des méta-objets. Le MOP (MetaObject Protocol) utilisé est appelé RAM [2].

Il nous a semblé pertinent de remplacer ce MOP par le container du serveur JOnAS jouant le rôle de tisseur dynamique entre les services et les composants. Après la prise de cette décision, notre prochain objectif consiste à analyser la plate-forme JOnAS pour savoir quels sont les modules clefs qui pourront intervenir dans la mise en œuvre de l'adaptation. Cette analyse ainsi faite nous a permis de déterminer le principal acteur pour répondre à ce besoin. Il s'agit de *l'objet d'interposition*, généré par l'outil *GenIC* «Generate Interposition Classes», jouant le rôle du container vis à vis des composants beans.

Une fois le point d'entrée repéré, l'idée de base est articulée autour de deux étapes distinctes. La première consiste à introduire une indirection au niveau de cet objet d'interposition, alors que la seconde a pour but d'intégrer l'infrastructure d'observation et le moteur d'adaptation des politiques réalisés au cours du travail cité en [1].

4.1 Mise en place d'une indirection dans JOnAS

Les différents services de base de JOnAS sont fournis aux composants à travers les objets d'interposition qui leur sont associés. Cela nous a fait penser aussi à déléguer la tâche de composition dynamique des services à ce niveau. Afin de «respecter» la spécification EJB dédié à l'objet d'interposition, la solution qui nous semble être la meilleure est le fait de mettre en place une indirection au niveau de cet objet vers un autre composant, appelé *DynamicComposite*, qui jouera donc le rôle de compositeur dynamique de services.

Grâce à l'outil GenIC offert en open source et au travail réalisé en [3], la réalisation de cette indirection s'est avérée possible. Elle consiste en un certain nombre de tâches séquentielles qui sont :

a. Intégration de l'indirection au niveau du fichier de déploiement

Comme la génération de l'objet d'interposition dépend des fichiers de déploiement `ejb-jar.XML` et `JOnAS-ejb-jar.XML`, nous avons ajouté cette intégration à ce niveau pour que l'outil GenIC produise ou non une indirection au sein de l'objet d'interposition lors de la phase de génération. Cette option rend notre architecture plus flexible.

b. Ajout d'une nouvelle balise au sein du fichier de déploiement JOnAS

Puisque cette intégration est liée à la plate-forme JOnAS, nous avons donc modifié le fichier de déploiement correspond `JOnAS-ejb-jar.xml` en faisant ajouter une nouvelle balise

```
<JOnAS-dynamic-composite> true or false </JOnAS-dynamic-composite>
```

qui indiquera si cette indirection doit être prise en compte ou non lors de la génération des objets d'interposition.

Exemple de fichier de déploiement JOnAS:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<JOnAS-ejb-jar>
  <JOnAS-session>
    <ejb-name>Op</ejb-name>
    <jndi-name>OpHome</jndi-name>
    <JOnAS-dynamic-composite>true</JOnAS-dynamic-composite>
  </JOnAS-session>
  <JOnAS-session>
    <ejb-name>Opbis</ejb-name>
    <jndi-name>OpbisHome</jndi-name>
    <JOnAS-dynamic-composite>false</JOnAS-dynamic-composite>
  </JOnAS-session>
</JOnAS-ejb-jar>
```

Pour pouvoir valider le fichier de déploiement JOnAS modifié, une modification sur la DTD associée (JOnAS-ejb-jar.dtd) est faite.

Un extrait de JOnAS-ejb-jar.dtd :

```
.....
<!ELEMENT JOnAS-entity (ejb-name, jndi-name, JOnAS-dynamic-composite?, JOnAS-
resource*, JOnAS-resource-env*, JOnAS-ejb-ref*, is-modified-method-name?, passivation-
timeout?, jdbc-mapping?)>

<!ELEMENT JOnAS-dynamic-composite (#PCDATA)>
.....
<!ELEMENT JOnAS-session (ejb-name, jndi-name, JOnAS-dynamic-composite?, JOnAS-
resource*, JOnAS-resource-env*, JOnAS-ejb-ref*, session-timeout?)>
.....
```

c. Génération du fichier package

org.objectweb.JOnAS_ejb.deployment.xml

Afin de faciliter la génération des classes formant le package `org.objectweb.JOnAS_ejb.deployment.xml` servant de parsing du fichier `JOnAS-ejb-jar.xml`, nous avons généré en premier temps grâce au produit *XML Spy (IDE)* le fichier XML Schema Definition `JOnAS-ejb-jar.xsd` à partir de la DTD (Document Type Definition) de JOnAS ainsi modifiée (cf Figure 3). Ce fichier qui décrit le schéma de la structure XML liée à la DTD de JOnAS, est utilisé en second lieu par un autre outil *Castor* qui nous a permis finalement d'avoir l'ensemble des différentes classes formant le package `org.objectweb.JOnAS_ejb.deployment.xml`.

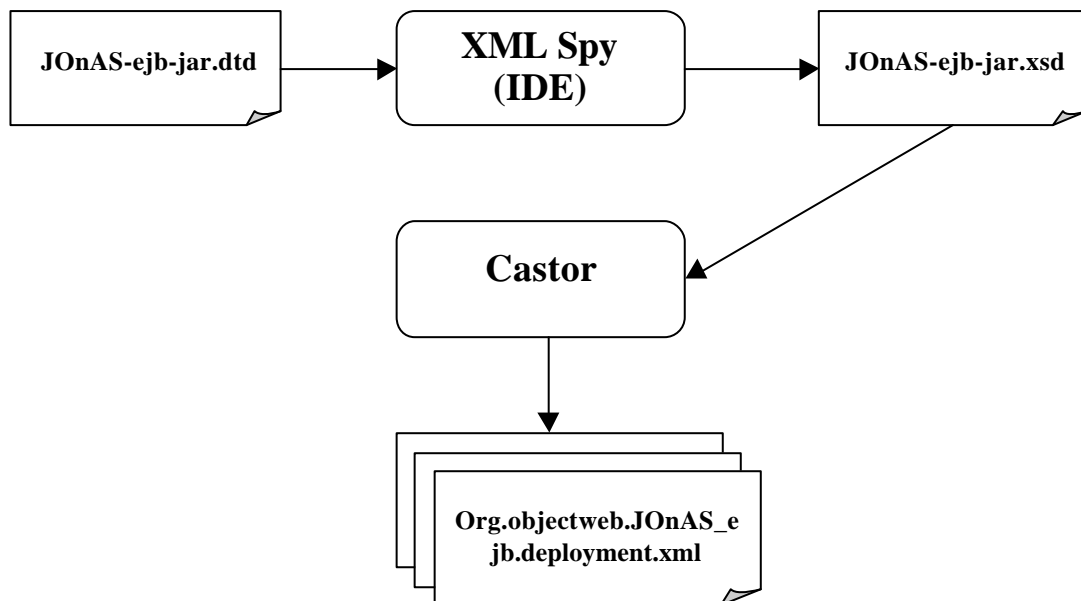


Figure 3 – Démarches de génération du package org.objectweb.JOnAS_ejb.deployment.xml

d. Mise à jour de quelques classes du package org.objectweb.JOnAS_ejb.deployment.api

Suite à l'insertion d'une nouvelle balise dans le fichier de déploiement JOnAS, il s'est avéré nécessaire de modifier certaines classes du package org.objectweb.JOnAS_ejb.deployment.api. Ce package représente en fait la mémoire de ce qui a été parsé lors de l'analyse des différentes balises des fichiers de déploiement grâce au package déjà généré (org.objectweb.JOnAS_ejb.deployment.xml). Il sert de principale entrée pour l'outil GenIC afin de générer les classes d'interposition.

Pour que GenIC soit au courant de la présence et/ou de la valeur attribuée à la balise ajoutée <JOnAS-dynamic-composite> dans le fichier de déploiement JOnAS, certaines classes ont été modifiées. Ces classes sont BeanDesc, SessionDesc et EntityDesc (voir Annexe).

e. Modification des classes servant de générateur de code

Pour mettre en place une indirection dans l'objet d'interposition, il a fallu modifier certaines classes, servant de générateur de code associé à l'objet d'interposition, pour introduire le code supplémentaire associé à cette indirection. Ces classes qui font partie du package org.objectweb.JOnAS_ejb.tools sont GenICRemote, GenICSessionRemote et GenICEntityRemote.

4.2 Mise en place de l'adaptabilité dynamique des services dans JOnAS

Grâce à l'indirection déjà réalisée, l'intégration de l'infrastructure d'observation et le moteur d'adaptation des politiques a été réalisée en réutilisant la quasi-totalité des codes associés. Cette intégration est prise en compte lors du lancement du serveur JOnAS avec le chargement des différents services prédéfinies dans JOnAS. Le service JOnAS ainsi ajouté, appelé *LoadPolicies*, consiste à charger les politiques liées aux fichier XML suivants :

- *Service.xml*, qui fournit au moteur d'adaptation la liste des différents services utilisés.
- *System.xml*, qui représente une politique de bas niveau. Cette politique définit des règles d'adaptation (Quels services doivent être utilisés et avec quels paramètres en fonction des conditions environnementales) de type *Condition --> Action*.
- *Application.xml*, qui décrit comment vont être appliquées les politiques systèmes et à quels composants.

Les démarches suivies afin de réaliser cette tâche sont décrites par les sous paragraphes suivants. Le premier sous-paragraph explicite comment intégrer le service *LoadPolicies* dans JOnAS. Le deuxième paragraphe décrit comment l'objet *DynamicComposite* est créé et ensuite comment on lui associe les politiques d'adaptation qui le concernent prises en compte par le service *LoadPolicies* au chargement?

4.2.1 Mettre en place le service *LoadPolicies* au sein de JOnAS

Afin de mettre en place le service *LoadPolicies* responsable du chargement des politiques d'adaptation au sein du serveur EJB, nous avons créé une classe qui implémente l'interface *org.objectweb.JOnAS.service.Service*. Le fragment de code suivant fait intervenir les différentes méthodes décrites dans l'interface :

```
public class LoadPolicies implements Service {
    private final int trace = Trace.DB_30;
    private String name = null;
    private boolean started = false;
    Context ictx = null;

    .....
    /**
     * Init LoadPolicies
     * Configuration information is passed thru a Context object.
     */

    public void init(Context ctx) throws ServiceException {
        Trace.outln(trace, "LoadPolicies - initialized");
    }

    public void start() throws ServiceException {

        Trace.outln(trace, "LoadPolicies - starting");
        File configDir = new File("config-server");
```

```

    LoadPolicies(configDir);
    this.started = true;
}
public void stop() throws ServiceException {
    if (this.started) this.started = false;
}
public boolean isStarted() {
    return this.started;
}
public String getName() {
    return this.name;
}
public void setName(String name) {
    this.name = name;
}
}
.....

```

Une fois la classe défini, nous avons modifié le fichier *JOnAS.properties* en introduisant les informations associées *au service LoadPolicies* :

```

.....
JOnAS.services                LoadPolicies,jmx,security,jtm,dbm,ejb

#
#           JOnAS LoadPolicies service configuration
# Set the name of the implementation class of the LoadPolicies service
JOnAS.service.LoadPolicies.class      proto.LoadPolicies

```

4.2.2 Mise en place de l'objet DynamicComposite

Dans ce sous-paragraphe, nous avons pris un exemple de session bean, appelé *Op*, qui appartient au package *sb* et qui implémente une méthode métier décrite comme suit :

- méthode *buy* (int *s*) qui fait ajouter la valeur de la variable *s* au total général.

L'indirection ainsi greffée est mise en place grâce à l'insertion d'une méthode « **execute** » à la place de l'appel standard généré par GenIC (e.g. *buy(p1)* dans cette exemple) de chaque méthode métier du beans au sein de l'objet d'interposition. Le fragment de code suivant illustre le code lié à cette indirection.

```

public synchronized void buy(int p1) throws java.rmi.RemoteException {
    TraceEjb.debugGenIC("JOnASOpRemote.buy(int)");
    String methodSignature = "null" ;
    RequestCtx rctx = preinvoke(2, methodSignature) ;
    JContextSession ctx = (JContextSession) rctx.getEJBContext();
    sb.OpBean eb = (sb.OpBean)ctx.getInstance();
    try {

```

```

Object[] args = new Object[]{ new Integer(p1) };
java.lang.reflect.Method meth=eb.getClass().getMethod("buy", new Class[]{Integer.TYPE });
dynamicComposite.execute(eb, meth, args);
} catch (NoSuchMethodException nsme) {
    throw new RemoteException("RuntimeException thrown by an enterprise Bean", nsme);
} catch (RuntimeException e) {
    rctx.setSysExc(e);
    throw new RemoteException("RuntimeException thrown by an enterprise Bean", e);
} catch (Error e) {
    rctx.setSysExc(e);
    throw new RemoteException("Error thrown by an enterprise Bean", e);
} finally {
    postinvoke(rctx);
}
}
}

```

Outre cet ajout d'indirection, nous avons inséré un code supplémentaire au sein de la méthode constructeur de l'objet d'interposition EJB Object. Ce code correspond aux fonctionnalités suivantes :

- **Création d'un objet DynamicComposite** jouant le rôle de compositeur dynamique de services, juste après la création de l'objet d'interposition par la fabrique Home.
- Recherche auprès du serveur de nommage JNDI, de la **localisation du service « LoadPolicies »** qui est responsable du chargement des différents politiques (Services.xml, System.xml, Applications.xml). La mise en place de ce service au sein du serveur JOnAS est décrit au dessus.
- **Attachement de l'objet DynamicComposite** par « LoadPolicies » au sein du groupe adéquat selon la politique application.xml chargée. Cette étape est illustré ci dessous.

Un exemple de code extrait de la classe d'interposition, associé à l'exemple cité associé au bean Op (JOnASOpRemote), décrit les différentes fonctionnalités citées auparavant :

```

public JOnASOpRemote(JSessionHome home) throws RemoteException {
    super(home, true);
    TraceEjb.debugGenIC("JOnASOpRemote constructor");
    //Attach DynamicComposite
    String beanName = "sb.OpBean";
    String dynaCompName = beanName;
    dynamicComposite = new DynamicComposite(dynaCompName);
    //search LoadPolicies service
    //Get a reference on loadPolicies service
    try {
        loadPolicies=(proto.LoadPolicies) serviceManager.getInstance().getService("LoadPolicies");
    } catch (ServiceException e) {
        System.out.println ("Cannot find LoadPolicies Service :"+e);
    }
    loadPolicies.addDynamicComposite(beanName,dynamicComposite);
}

```

Afin de mieux illustrer cette étape, prenons l'exemple de politique application.xml et system.xml présentés respectivement ci-dessous :

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<application-policy>
  <group name="groupbean1">
    <select from="all">
      <or>
        <equals>
          <property-value name="className"/>
          <string value="sb.OpBean"/>
        </equals>
        <equals>
          <property-value name="className"/>
          <string value="sb.OpbisBean"/>
        </equals>
      </or>
    </select>
    <bind policy="tracer"/>
  </group>
</group>
</application-policy>

<?xml version="1.0" encoding="ISO-8859-1"?>
<config>
  <system-policy name="tracer">
    <rule>
      <when>
        <less-than>
          <property-value name="/system/network.bandwidth"/>
          <number value="40000"/>
        </less-than>
      </when>
      <ensure>
        <attached service="misc.trace" role="main">
          <parameter name="prefix" value="TRACE"/>
        </attached>
        <attached service="servicelog" role="main">
          <parameter name="prefix" value="TRACE"/>
        </attached>
      </ensure>
    </rule>
  </system-policy>
</config>

```

Une fois l'objet `dynamicComposite` créé et le service `LoadPolicies` localisé, nous allons décrire le scénario qui fait associer à cet objet les politiques systèmes qui le concerne grâce à la description introduit par la politique applicative (cf. `application.xml`).

D'après l'exemple de la politique applicative décrite ci-dessus, le bean dont le nom de la classe est «`sb.OpBean`» (qui représente notre bean) appartient au groupe nommé «`groupebean1`». L'ensemble des beans qui appartiennent à ce groupe sont attaché à la politique système «`tracer`» (cf. `System.xml`).

Puisque que l'objet `dynamicComposite` ainsi créé et qui représente le compositeur dynamique des services vis à vis du bean «`Op`», nous l'avons associé au groupe «`groupebean1`» représentant le bean `Op`. Ainsi et grâce au service `LoadPolicies` «chargeur de politiques d'adaptation» lancé dans le serveur, l'objet `dynamicComposite` possèdera les capacités d'assumer son rôle de compositeur.

L'infrastructure ainsi réalisée est décrite par la figure 4 suivante.

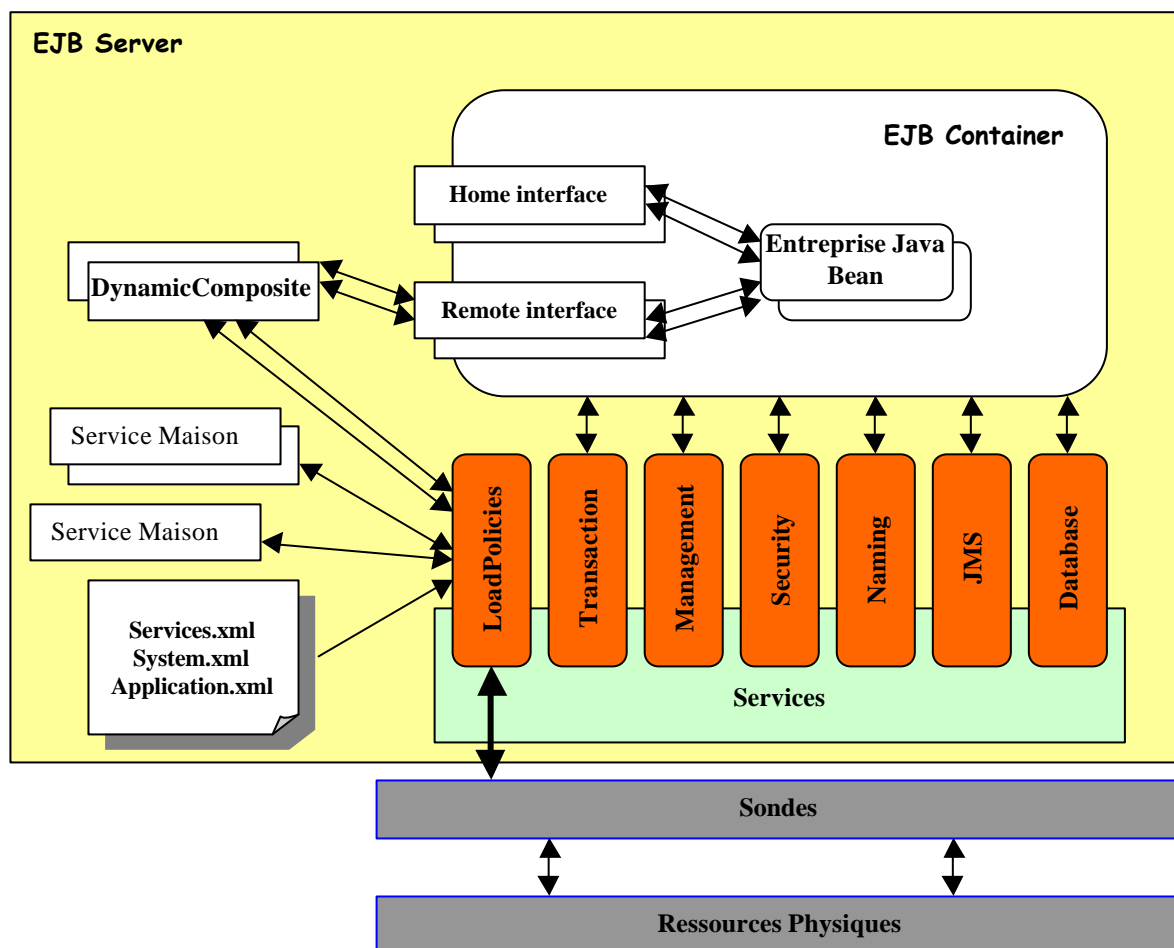


Figure 4 - Infrastructure pour middleware adaptable basée sur JOnAS.

5 Conclusions et Perspectives

Comme cela a été décrit, nous avons pu intégrer une infrastructure pour middleware adaptable au sein du serveur d'applications JOnAS. Cette intégration a rendu donc la plateforme JOnAS capable d'adapter dynamiquement les associations entre les services et les composants beans. Cette adaptation, comme cela a été citée, fait appel à des services maillons très simples.

Afin de tirer profit de cette infrastructure adaptable, il serait intéressant dans un futur travail de se pencher sur l'utilisation des services plus avancés et en particulier sur les services prédéfinis de JOnAS.

Remerciements

Au terme de ce travail, j'exprime ma profonde reconnaissance au Pr. Pierre COINTE de m'avoir permis de travailler au sein de l'équipe OCM du département Informatique de l'Ecole des Mines de Nantes.

Je désire aussi exprimer ma profonde gratitude envers mon directeur de recherche, Mr. Thomas LEDOUX, dont les directives et la disponibilité pour répondre à mes questions, m'ont été du plus grand profit malgré ses nombreuses occupations. Ses qualités humaines, son soutien et son aide à l'accomplissement de ce travail m'ont permis de le mener dans les meilleures conditions.

Je souhaite remercier Pierre-Charles DAVID pour ses prestigieuses remarques et les discussions fructueuses partagées ensemble autour de ce sujet.

Je remercie également Ocello AUDREY pour ses précieuses aides et remarques.

Je tiens aussi à remercier Melle Christine VIOLEAU d'abord pour sa sympathie et puis pour l'attention particulière qu'elle m'a accordée pour l'aboutissement de mes affaires administratives.

Références

- [1] DAVID, P. C., « Une infrastructure pour middleware adaptable », DEA, Université de Nantes, Ecole des Mines de Nantes, septembre 2001.
- [2] La référence exacte de RAM
- [3] OCCELLO Audrey, « Développement du service d'interactions dans les Entreprise Java Beans », ESSI/I3S.
- [4] Mickael Bartorello, Hélène Maguin, Audrey Ocello, Mireille Blay-Fornarino, Anne-Maroe Dery, Michel Riveill, « Intégration de services au sein d'un serveur d'EJB », LMO'2002, pages 169 à 183.