RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Administration fondée sur l'architecture des serveurs d'applications J2EE patrimoniaux

Takoua Abdellatif[1], Fabienne Boyer[2], Jakub Kornas[3] et Jean-Bernard Stefani[3]
[1]Bull [2]UJF [3]INRIA

Project Sardes, INRIA Rhône-Alpes
655, avenue de l'Europe 38334 Saint-Ismier Cedex, France
prenom.nom@inrialpes.fr

Résumé
Dans ce papier, nous présentons JonasALaCarte, un système d'administration pour des serveurs d'applications J2EE patrimoniaux en se basant sur la description de leurs architectures. Nous montrons qu'adopter une approche fondée sur l'architecture de systèmes distribués apporte des avantages immédiats en termes de flexibilité, de facilité d'utilisation, de gestion de configuration et de propriétés d'autonomie. La ré-ingénierie d'un système d'applications (le serveur libre JOnAS dans notre cas) peut être réalisée avec un effort minimal et avec un surcoût négligeable sur les performances comparées à celles du serveur d'origine.

Mots-clés : Administration autonome, serveurs J2EE, intergiciels à composants, Fractal.

---

1. Introduction

Les serveurs d'applications J2EE [2] présentent des architectures complexes orientées *services*. Par exemple, le serveur JOnAS [3] contient un conteneur Web tel que Tomcat [1], un serveur EJB, un serveur de transaction encapsulant JOTM [4], etc. Par ailleurs, les serveurs d'applications sont déployés dans des environnements de types grappes, ce qui rend leur administration un vrai défi. Les standards existants [2] gèrent uniquement une partie de l'administration, notamment le déploiement des applications. La gestion de l'intergiciel est laissée à la charge des fournisseurs de serveurs. Ceci laisse ouverts un ensemble de difficultés que nous résumons dans les points suivants.

- **Déploiement** : Plusieurs serveurs d'applications fournissent des outils de paquetage et de déploiement. Cependant, ce déploiement est généralement réstreint à une seule JVM et l'architecture des serveurs reste cachée à l'exécution. Il n'est pas possible de distribuer des serveurs sur les machines d'une grappe par exemple sans ajouter des outils supplémentaires. De plus, l'unité de déploiement considère actuellement tout le serveur et non pas le service ce qui est insuffisant pour une gestion optimale des ressources.
- **Configuration** : Plusieurs serveurs libres offrent des interfaces de configuration uniformes pour masquer l'hétérogéinité des services. Cependant ces interfaces ne sont pas généralisées pour configurer les serveurs dans des environnements distribués.
- **Performance et gestion des fautes** : Ces tâches nécessitent la mise en place de mécanismes de reconfiguration pour réagir aux pannes logicielles et matérielles, aux attaques de sécurité et aux pics de charge. Actuellement, ces opérations nécessitent le redémarrage complet du serveur alors qu'elles devraient avoir un impact minimal.

Nous croyons que pour résoudre ces problèmes, il est nécessaire que l'architecture logicielle interne du serveur, en termes des services qui le composent et leurs dépendances, soit explicite et reconfigurable à l'exécution. La configuration et les paramètres de déploiement doivent être décrits en utilisant des éléments de l'architectre du système. Cette description sert alors de base pour automatiser et implanter différentes politiques de déploiement et de reconfiguration. Ce sont les principes de ce qu'on appelle *administration fondée sur l'architecture* [14].

Afin de rendre explicite l'architecture logicielle interne d'un serveur d'applications, nous avons adopté un modèle à composants pour le système administré. Le même modèle est également adopté pour le système d'administration. Dans cette approche, les différentes parties logicielles et matérielles sont représentées grâce à des composants. Ceci apporte les avantages suivants.

- **Configuration uniforme** : la configuration et le déploiement du système administré et du système d'administration sont simplifiés grâce à l'utilisation d'un même modèle à composants offrant le même type d'interfaces de contrôle.
- **Administration automatisée** : Les entités administrables peuvent être supervisées grâce à des boucles de contrôle qui sont organisées sous la forme de composants. Ceci permet d'automatiser un ensemble de fonctions classiques d'administration traitant la mise à jour de code, la tolérance aux pannes, le redimentionnement, etc.
- **Administration flexible** : des politiques d'administration autonomes sophistiquées [18] peuvent être facilement intégrés sous la forme de composants.

Dans ce papier, nous décrivons comment nous avons appliqué l'approche d'une administration basée sur l'architecture à un serveur d'applications J2EE patrimonial. Les contributions de ce papier peuvent être résumées dans les points suivants.

- Nous montrons comment appliquer un canevas d'administration basé sur l'architecture sur un serveur d'applications patrimonial.
- Nous montrons les avantages d'une telle approche en termes d'uniformité, d'administration distribuée (à l'échelle d'une grappe) et automatisée.
- Nous montrons que la ré-ingénierie d'un serveur d'applications patrimonial à gros grains (à la granularité des services) peut s'effectuer avec un effort minimal et avec un surcoût négligeable sur les performances.

Ce papier est structuré comme suit. La section 2 présente plus en détails le contexte et la motivation de notre travail. Nous introduisons les principes d'une administration basée sur l'architecture et son implantation dans la section 3. La section 4 présente l'application de notre approche à un serveur d'applications patrimonial. Dans la section 5, nous présentons l'évaluation de notre travail. Nous concluons à la fin du papier et nous présentons les perspectives de ce travail.

2. Contexte et motivation

Nous discutons dans cette section, les architectures des serveurs d'application J2EE libres les plus connus (JOnAS, JBoss, Geronimo)[1] et nous montrons que ces architectures cachées à l'exécution ne permettent pas d'appliquer l'approche de l'administration basée sur l'architecture. Nous discutons également des outils que ces serveurs offrent pour l'administration dans une grappe.

2.1. Les architectures des serveurs d'applications J2EE

Les serveurs d'applications représentent des environnements d'exécution des applications J2EE. Ils offrent un ensemble de services pour des propriétés dites non-fonctionnelles comme la persistance, les transactions, etc. Dans les serveurs d'applications libres, plusieurs approches sont adoptées pour l'assemblage des services, leurs configurations et leurs déploiements.

JOnAS

Dans JOnAS, l'abstraction *service* est utilisée pour offrir une abstraction uniforme ainsi que des interfaces communes pour le démarrage et l'arrêt des services. Ces interfaces, ainsi que l'administration du serveur sont basées sur le standard JMX. Cependant, l'architecture du serveur est complètement cachée à l'exécution et les services communiquent via des références statiques. La configuration des services utilise des fichiers scripts différents et l'administrateur doit maîtriser la syntaxe de ces fichiers. De plus, les services ne sont pas archivés dans des paquets séparés et doivent démarrer en même temps que le serveur. Par conséquent, il n'est pas possible d'ajouter dynamiquement un service après le démarrage du serveur. A l'exécution l'architecture est difficile à modifier (par exemple pour rebooter un service après une panne ou changer la version de son code) car les dépendances entre les services sont cachées.

[1] Nous ne discuttons pas les serveurs propriétaires tels que Websphere de IBM par manque d'informations sur leurs architectures internes.

JBoss

JBoss [19] dispose d'une architecture plus modulaire que JOnAS, baséé sur la philosophie du micro-noyau (*micro-kernel*). Le micro-noyau de serveur est basé sur un serveur JMX et les services sont greffés comme étant des MBeans. Le modèle MBean de JMX est amélioré dans JBoss afin que chaque service ait son propre cycle de vie et puisse être archivé dans un paquet séparé. Un descripteur de service permet de configurer chaque service et de lister les MBeans requis qui doivent être démarrés à l'avance. Un MBean de gestion de dépendance (*dependency manager*) se charge de vérifier les dépendances entre les services pendant le temps du déploiement et au sein d'une même JVM. Cependant, les relations entre les services sont cachées à l'exécution et ne sont pas modifiables. Par ailleurs, comme dans JMX, il n'est pas possible de capturer les relations d'encapsulation entre les MBeans (par exemple pour détecter les dépendances de pannes entre un service et la machine sous-jacente). Le déploiement est effectué dans JBoss dans une seule JVM et le modèle ne permet de décrire un déploiement distribué dans une grappe par exemple, où les dépendances d'encapsulation entre les services et les machines cibles ainsi que les connections distantes doivent être décrites d'une manière explicite.

Geronimo

Geronimo [20] présente comme JBoss une structure de micro-noyau. Les services à l'intérieur de Geronilmo sont appellés GBeans et sont des unités de déploiement et de configuration. Ils peuvent être archivés dans des paquets séparés et un plan descripteur permet d'exprimer les dépendances entre les services qui sont résolues au moment du déploiement, comme dans JBoss. Geronimo permet aux services de communiquer sans l'intervention du noyau en créant dynamiquement des éléments de communication intermédiaires (*proxy*) entre les GBeans. Comparé à JBoss, ceci offre une dépendance faible entre les services. Cependant, ces dépendances restent locales à une seule JVM et la relation d'encapsulation ne peut pas être capturée non plus entre les GBeans. Le modèle ne permet pas par conséquent d'automatiser le déploiement dans un environnement distribué.

2.2. Les grappes J2EE

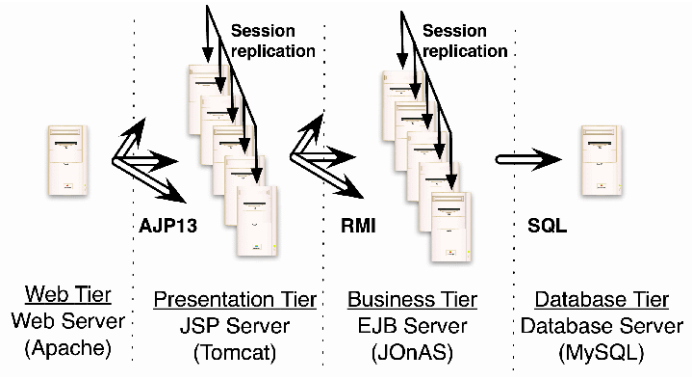

FIG. 1 – Environnement de grappes J2EE.

La figure 1 présente une architecture typique d'une grappe de serveurs J2EE. Le but de la grappe est d'offrir des propriétés d'équililbrage de charges et de tolérance aux pannes en dupliquant les serveurs et leurs données à états. La figure 1 montre la duplication des conteneurs Web à l'aide de l'équilibreur de charge mod_jk qui permet de transférer les requêtes HTTP du serveur Apache aux conteneurs Web. Dans le cas de JOnAS, CMI (*Clustered Method Invocation*) permet d'équilibrer la charge entre les conteneurs EJB. Un système de communication de groupes permet la duplication des sessions HTTP et les EJB stateful.
Administrer une grappe J2EE nécessite une vue générale du système et la possibilité d'agir d'une manière cohérente sur toutes les ressources et les fonctions mises en place. Par exemple, comme dans

JAGR [11], gérer une panne logicielle dans l'étage EJB pourrait nécessiter une intervention en amont pour bloquer les appels entrant pendant le temps de la réparation.
Dans le cas des trois serveurs d'applications J2EE discutés ci-dessus, la gestion d'une grappe J2EE est actuellement une tâche manuelle qui nécessite encore des supports et des outils d'observation. Le déploiement par exemple, est actuellement une tâche difficile réallisée à l'aide de fichiers scripts sujets aux fautes vue le nombre des ressources et leurs inter-dépendances (par exemple, les connections entre les étages J2EE et les équilibreurs de charge). Ces éléments ne sont pas actuellement capturés dans les descripteurs de déploiement des serveurs d'applications.
De plus, comme discuté dans 2.1, à cause de leur configuration locale à une seule JVM, l'unité de duplication est la totalité du serveur. Ceci conduit à la duplication pas forcément nécessaire des services autres que les conteneurs. Par exemple, il suffit dans certains cas de dupliquer le service de transaction seulement deux fois pour la tolérance aux pannes alors que les conteneurs Web et EJB auraient besoin d'être dupliqués sur une dizaine de machines afin d'augmenter les performances.

3. L'administration fondée sur l'architecture

Dans cette section, nous définissons les principes de l'administration fondée sur l'architecture et nous présentons par la suite le modèle à composants Fractal [9, 10] que nous avons adopté dans notre travail. Par la suite, nous présentons le canevas d'administration basé sur Fractal que nous avons utilisé pour l'administration des serveurs J2EE.

3.1. Les principes de l'administration fondée sur l'architecture

Dans une administration fondée sur l'architecture, la configuration du système et les paramètres de déploiement sont décrits en utilisant les éléments de l'architecture du système. Afin d'atteindre cet objectif, nous avons identifié deux principes.

- L'architecture du système doit être explicite en termes des ressources à administrer (les services, les noeuds, etc) ainsi que leurs relations (encapsulations et liaisons). Elle doit être également modifiable à l'exécution.
- Chaque ressource administrée doit être une unité séparée de configuration et de déploiement. Ceci est nécessaire pour la reconfiguration afin de pouvoir remplacer une ressource administrée par une autre.

Ces principes peuvent être appliqués à des granualrités différentes selon le niveau d'adaptation souhaité. De plus, ces principes sont appliquables au système d'administration lui-même afin de permettre l'intégration et la modularité des politiques d'administration.
La section suivante introduit Fractal le modèle à composants que nous avons choisi et justifie notre choix.

3.2. Le modèle à composants Fractal

Fractal [9] est un modèle à composants extensible qui distingue deux types de composants : primitifs et composites. Les composants primitifs sont des classes Java standard respectant des conventions de codage. Les composants composites encapsulent un groupe de composants primitifs et/ou d'autres composants composites. Les systèmes construits avec le modèle Fractal sont adaptables considérant le composant comme étant une unité de déploiement et de configuration. L'architecture d'un système est décrite dans des fichiers ADL en termes de relations entre les composants (liaisons et encapsulations). Les liaisons peuvent être locales à une seule JVM ou distantes. Elles peuvent être aussi implantées sous la forme de composants pour définir plusieurs sémantiques et protocoles de communication adaptables. Ces propriétés sont en fait spécifiques à Fractal, comparées à d'autres modèles à composants comme c'est expliqué dans [9].
Avec Fractal, l'architecture du système est explicite en termes de relations d'encapsulation et de connections locales à une même JVM ou distantes. Elle est modifiable à l'exécution grâce aux controleurs Fractal, ce qui permet d'avoir des systèmes dynamiquement adaptables où les composants sont des unités de configuration et de déploiement. A l'exécution, les liaisons entre les composants peuvent être modifiées tout en tenant compte des contraintes du cycle de vie du composant.

3.3. Canevas réflectif pour l'administration fondée sur l'architecture

Dans cette section, nous présentons un canevas basé sur Fractal appellé Jade [8] construit suivant les principes d'une administration fondée sur l'architecture. Le canevas complète les propriétés de base du modèle Fractal et son ADL avec un ensemble d'outils et de règles aidant à la construction de systèmes administrables. Une description détaillée de Jade peut se trouver dans [8].

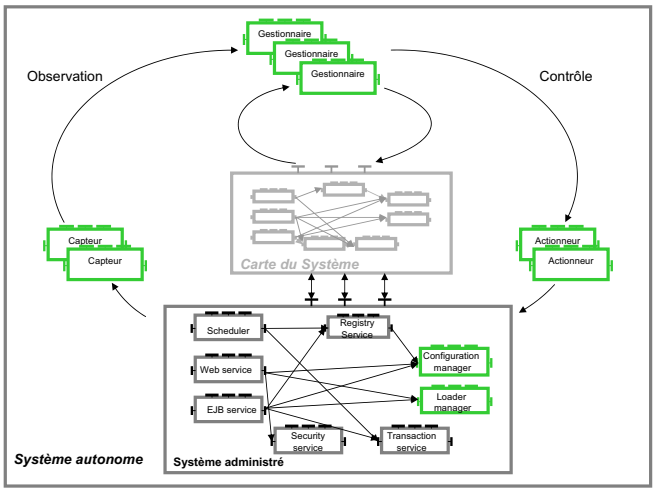

FIG. 2 – Architecture d'un système autonome.

La figure 2 illustre les éléments principaux du canevas qui organise les activités d'administration sous la forme de boucles de contrôle comprenant les éléments suivants :

- Le système administré qui doit être organisé sous la forme de composants Fractal.
- Les capteurs et les actionneurs qui fournissent des interfaces d'observation et de contrôle. Les sondes sont responsables de la détection des changements d'états telle ques la violation des contraintes de qualité de services ou les pannes. Les actionneurs fournissent les mécanismes nécessaires pour agir sur le système comme rebooter un composant fautif ou mettre à jour le code d'un composant. Les capteurs et les actuateurs peuvent être implantés sous la forme de composants Fractal ou sous la forme de controlleurs Fractal.
- Les composants gestionnaires (*managers*) qui implantent les politiques d'administration et les décisions à prendre selon l'état observé suivant les boucles de commande. Les gestionnaires réagissent aux évènements remontés par les sondes et les décisions sont exécutées à travers les actionneurs.
- La carte du système (*System Map*) qui fournit une représentation causalement connectée du système administré dont les gestionnaires se servent pour la supervision de l'état du système.

En tant que composants Fractal, les capteurs, actuateurs et gestionnaires peuvent être configurés et déployés en utilisant les mêmes outils que les composants du système à administrer.

Afin d'utiliser ce canevas, le développeur doit implanter les composants gestionnaires, les capteurs et les actuateurs qui sont spécifiques au système administré et écrire les fichiers ADL décrivant l'architecture du système à déployer.

4. Application à un serveur d'application patrimonial

Nous présentons dans cette section JonasALaCarte, notre personalisation du canevas Jade au serveur d'applications JOnAS. Ceci comprend (1) la ré-ingénierie de JOnAS afin de rendre explicite l'architecture du serveur sous la forme de composants Fractal et (2) l'implantation des composants d'administration (les capteurs, les actuateurs et les gestionnaires) selon la politique d'administration souhaitée.

La description de la ré-ingénierie de JOnAS est illustrative de l'effort nécessaire pour appliquer notre approche à un serveur d'applications patrimonial. En effet, un tel travail de ré-ingénierie doit être plus simple pour des serveurs d'applications modulaires comme JBoss et Geronimo.

4.1. Ré-ingénierie de JOnAS

Dans JOnAS, un service étend une classe `AbsServiceImpl` qui permet de démarrer et d'arrêter un service utilisant la classe `ServiceManager`. Un service accède à un autre service X à travers un appel statique à la classe `ServiceManager` : `ServiceManager.getInstance().getService(X)`. En plus des relations implicites, il y a l'hypothèse que ces services et leurs entités d'administration s'exécutent dans la même machine virtuelle. Dans JonasALaCarte, un service doit étendre une classe `AbsServiceWrapper` qui exécute un service JOnAS dans un composant Fractal primitif. Cette classe abstraite implante principalement quatre types d'interfaces :

– l'interface fonctionnelle d'un service,
– le `LifeCycleController` offrant à un service un cycle de vie indépendant respectant la spécification du JSR77,
– le `ServiceConfController` servant à positionner les paramètres de configuration d'un service,
– le `BindingController` permettant de relier un service à d'autres services ou à d'autres composants d'administration.

L'adaptateur (*Wrapper*) lance la classe d'un service JOnAS. Cette classe est modifiée de telle sorte que tous les appels statiques sont remplacés par des appels sur les références passées par l'adaptateur. La figure 3 illustre la transformation que nous avons effectuée au niveau de l'implantation d'un service (le service JOnAS est représenté à gauche alors que le service adapté est représenté à droite).

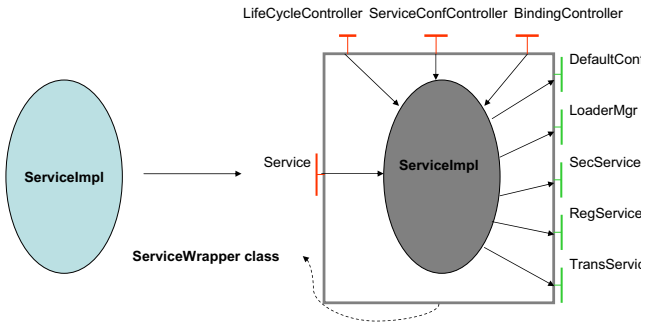

FIG. 3 – Ré-ingénierie d'un service JOnAS.

La figure 4 donne un exemple de configuration d'un service adapté de JOnAS, utilisant Fractal ADL. Cette description d'architecture contient les informations suivantes :

– la classe d'implantation du service adapté délimitée par le marqueur `<content>`,
– les valeurs de configuration delimitées par `<attributes>`,
– la liste des interfaces clientes et serveurs permettant de connecter le service à d'autres composants,
– le marqueur `virtual node` qui permet de spécifier la machine virtuelle sur laquelle le service doit être déployé.

JonasALaCarte est à l'exécution un composite Fractal contenant des services et des entités d'administration sous la forme de sous-composants. La figure 5 représente une version simplifiée de l'architecture de JonasALaCarte. Pour la clarité de la figure, nous ne représentons qu'une partie des services.

Un composant `configuration manager` permet de fournir une configuration par défaut pour les services JOnAS. Un service peut , en effet, récupérer sa configuration par défaut à partir de ce composant (par exemple, dans la figure, les composants Web container et EJB container sont liés au composant `configuration manager`). L'autre possibilité consiste à positionner des valeurs de configuration via `attribute controller`. Il est possible de reconfigurer un service soit en modifiant l'`attribute controller` soit en changeant sa liaison vers un autre composant `configuration manager`.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD Fractal ADL 2.0//EN"
  "classpath://org/objectweb/fractal/adl/xml/standard.dtd">

<definition name="org.objectweb.jonasALaCarte.EJB">
    <!-- Content class -->
    <content class="org.objectweb.jonasALaCarte.EJBServiceWrapper"/>

    <!-- Attribute Controller -->
    <attributes signature="org.objectweb.jonasALaCarte.ServiceConfController">
       <attribute name="conf"  value="mdbminthreadpoolsize 10;
                               mdbmaxthreadpoolsize 25; ...  "/>
       <attribute name="serviceName" value="ejb "/>
    </attributes>

    < !-- Interfaces -->
    <interface name="confItf" role="client"
               signature="org.objectweb.jonasALaCarte.ConfItf"/>
    <interface name="loaderItf" role="client"
               signature="org.objectweb.jonasALaCarte.LoaderItf"/>

    <interface name="transactionItf" role="client"
               signature="org.objectweb.jonasALaCarte.TransactItf"/>
    <interface name="registryItf" role="client"
               signature="org.objectweb.jonasALaCarte.RegistryItf"/>
    ...
    <virtual-node name="EJBNode"/>
</definition>
```

FIG. 4 – Extrait d'un fichier ADL pour la configuration du service EJB.

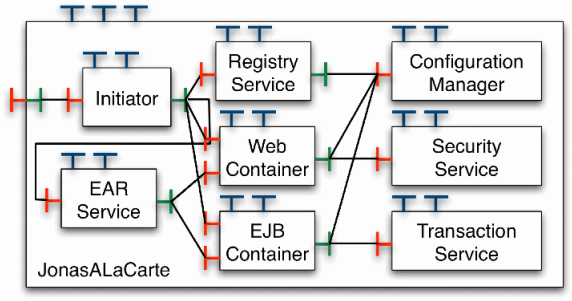

FIG. 5 – Architecture à base de Fractal de JonasALaCarte.

Nous avons introduit un composant `initiator` qui permet de démarrer les services dans un ordre bien défini. Par exemple, il est nécéssaire de démarrer le service *Security* avant de démarrer le service Web. L'architecture de JonasALaCarte expose les éléments internes du serveur d'applications d'une manière explicite. Les liaisons entre les services ainsi que les liaisons entre les services et les composants d'administration sont exprimées sous la forme de liaisons Fractal représentées par les flèches sur la figure 5. Par exemple, le service EJB est explicitement lié au service Transaction.

4.2. Administration des grappes J2EE

Pour une exploitation en grappes, l'architecture de JonasALaCarte est spécifiée, comme en centralisé, dans un fichier ADL. L'instantiation de cette description permet de configurer et de déployer les composants du serveur d'applications sur les machines cibles. Par exemple, dans la figure 4, nous précisons que le conteneur EJB doit être déployé sur le nœud "EJBNode". Les nœuds virtuels sont à leur tour des composants Fractal dont la configuration est dirigée par l'ADL.

Pour déployer JonasALaCarte sur une grappe, l'administrateur doit décrire dans l'ADL l'architecture du système dans l'environnement distribué, notamment la spécification des nœuds sur lesquels les différents services et composants vont être déployés. Le déploiement est alors automatisé grâce au déployeur Fractal qui analyse l'ADL.

Contrairement aux grappes actuelles de JOnAS, l'unité de duplication de JonasALaCarte est le service (encapsulé dans un composant Fractal) et non pas la totalité du serveur. Cette duplication sélective est importante puisque les conteneurs EJB et Web représentent généralement les goulots d'étranglement et il est nécessaire d'avoir plus de réplicas pour ces conteneurs comparés à d'autres services de JOnAS.

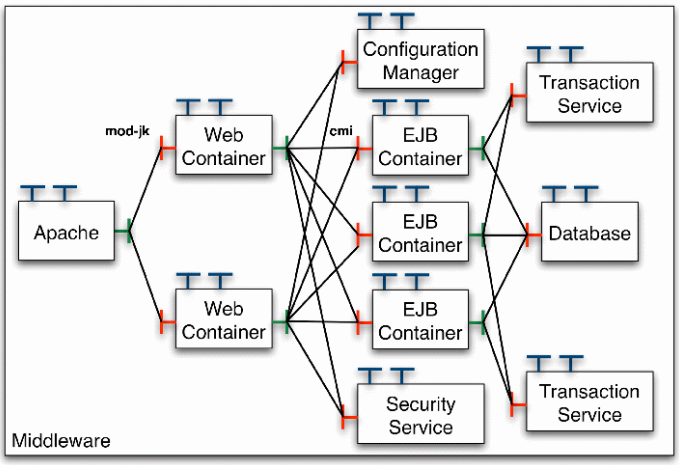

FIG. 6 – Environnement de grappes J2EE : la vue composants.

La figure 6 représente un exemple d'architecture à composants Fractal correspondant au système représenté dans la figure 1. Deux points méritent d'être soulignés :

- La configuration et le déploiement des serveurs J2EE sont abstraits à la manipulation de composants Fractal.
- La configuration d'une grappe est juste une configuration particulière du serveur où les composants sont distribués et dupliqués sur des JVM différentes. Les mêmes outils sont utilisés pour administrer le serveur sur une seule JVM ou dans une grappe.

4.3. Administration autonome

Notre implantation du canevas réflexif présenté dans la section 3.3 nous permet d'introduire des politiques d'administration différentes dans JonasALaCarte. En effet, nous avons implanté des composants gestionnaires permettant d'effectuer un démarrage partiel du serveur pour réparer des pannes logi-

cielles ou des pannes matérielles. La politique consiste ici à redémarrer seulement les services et composants impactés par une faute. Cette politique est similaire à celle de JAGR [11] bien que le niveau de la reconfiguration soit l'application dans JAGR et pas l'intergiciel.
Notre implantation comprend les composants suivants :
– Les capteurs : nous avons considéré deux types de sondes matérielles et logicielles. Les sondes matérielles se chargent d'envoyer des requêtes régulières aux machines à administrer pour vérifier qu'elles sont toujours opérationnelles. Pour illustrer les pannes logicielles, considérons la panne du conteneur Web. Nous avons implanté une sonde qui interroge régulièrement une servlet légère qui tourne dans le conteneur Web. Si au bout d'un temps configurable, la cible (la machine ou le service) ne répond pas, elle est jugée en panne. Nous ne considérons dans ce papier que les pannes franches.
– Un ensemble d'actuateurs : leur rôle est de rebooter les services fautifs et de relier à nouveau les composants dissociés.
– Les gestionnaires réparant les pannes : ils implantent la politique de démarrage partiel (*micro-reboot*). Quand une partie du système tombe en panne, ceci pourrait avoir un impact sur les autres parties du serveur. Par exemple, dans le cas d'une panne d'un nœud, tous les services s'exécutant sur cette machine ne sont plus accessibles. Gérer ce genre d'évènements dans notre contexte, implique généralement l'exécution d'un certain nombre d'actions (la reconfiguration, le déploiement, le redémarrage, l'ajout de nouveau composants, etc). Par exemple, dans une grappe, la panne d'un service peut conduire à agir au niveau du composant Apache pour bloquer les appels entrants le temps de la reconfiguration.

Pour illustrer le comportement autonome de notre système, considérons le cas de la panne du conteneur Web. Les actions principales prises par le gestionnaire de fautes est (1) d'activer le proxy au niveau du composant Apache, introduit pour stocker les appels entrants dans une liste le temps de la reconfiguration, (2) nettoyer le composants fautif (en tuant des threads en cours d'exécution et en coupant les liaisons) et démarrer un nouveau composant conteneur Web sur la machine, (3) établir les liaisons entre ce nouveau composant et les autres et (4) débloquer les requêtes au niveau de Apache. Le gestionnaire a la connaissance des liaisons à établir entre les composants du système grâce à la carte. Il est également possible de dupliquer les infomations avec états (les sessions dans le cas du conteneur Web) au niveau de la carte qui sert alors de cache pour ces informations. Dans le cas d'une panne matérielle, le gestionnaire a également la connaissance des composants sur le nœud ainsi que leurs relations. Il effectue les actions décrites ci-dessus, instancie les composants sur un autre nœud alloué et rétablit les liaisons nécessaires.

5. Evaluation

Nous avons évalué notre travail selon plusieurs critères : le coût du travail de ré-ingénierie, le temps de la configuration et du déploiement, la pertinence de l'approche du redémarrage partiel pour l'intergiciel, la compatibilité avec les standards et l'évaluation de JonasALaCarte comparé à JOnAS.

5.1. Coût du travail de ré-ingénierie

La partie la plus dure du travail consistait à trouver les dépendances entre les services afin dechanger les références statiques vers des liaisons Fractal. Nous avons ajouté en plus un nouveau paquetage contenant les adaptateurs Fractal des services ainsi que les composants d'administration. Ceci représente approximativement 3300 lignes de code :
– Onze classes Java sont dédiées aux entités d'administration et aux composants de configuration.
– Un adaptateur par service (nous avons encapsulé 8 services dans des composants Fractal).

La totalité du travail de ré-ingénierie a nécessité un mois pour un programmeur connaissant Fractal.

5.2. Temps de configuration et de déploiement

Cette phase consiste principalement à écrire les fichiers ADL. Cette phase est actuellement manuelle puisque nous ne disposons pas actuellement d'outils de haut niveau permettant de générer les fichiers ADL ou de vérificateurs de validité sémantique comme celui présenté dans [7]. La configuration et le déploiement d'une grappe nécessite quelques heures (le temps nécessaire pour écrire les ADL) à la place de quelques jours comme c'est le cas actuellement pour les grappes JOnAS. Les outils de déploiement Fractal permettent de déployer automatiquement les composants sur les machines appropriées.

5.3. La pertinence du démarrage partiel

Dans notre expérience, le redémarrage d'un service ainsi que l'établissement des liaisons et la récupération de l'état ont nécessité 3 secondes alors que le redémarrage de tout JOnAS nécessite près d'une minute. Ceci valide la pertinence de l'approche de redémarrage partiel pour assurer la continuité de service. Ce résultat a été d'ailleurs montré dans JAGR [11], où le temps de redémarrage d'un service est de l'ordre de la seconde.

5.4. la compatibilité avec les standards J2EE

Notre travail de ré-ingénierie est compatible avec les standards J2EE : JSR77 et JSR88 implantés dans JOnAS. JSR77 décrit un modèle d'information que tous les serveurs d'applications doivent exposer grâce à une instrumentation avec JMX. Dans JonasALaCarte, nous avons gardé toutes les interfaces des MBeans. Nous avons exposé ces interfaces comme étant des `attribute controllers` afin de pouvoir les manipuler comme des interfaces Fractal. JonasALaCarte expose donc le modèle d'information du JSR77. De la même manière, les interfaces compatibles avec JSR88 permettant de déployer les applications sont aussi préservées et exposées comme des interfaces Fractal. Le travail de ré-ingénierie sur l'intergiciel ne remet pas en cause le paquetage et le déploiement des applications J2EE respectant JSR88.

5.5. L'évaluation des performances

Nous considérons deux métiriques : le nombre de requêtes par seconde servies et les ressources (CPU et mémoire) utilisées par le serveur. Notre évaluation de performance est effectuée utilisant RUBiS [12].
Nous avons utilisé trois machines : une machine pour émuler les clients, une machine pour le serveur d'applications (JOnAS ou JonasALaCarte) et une machine pour la base de données (MySQL) :

– Machines de l'émulateur des clients et du serveur d'applications : SMP Bi-Processor AMD Athlon 1800+ (1533 MHz), 1Gb RAM.
– Machine de la base de données : 2x Intel Xeon 1.8 GHz, 1 Gb

Les machines tournent la version 2.4.26 du noyau Linux. Nous avons utilisé la version 4.4.0 de JOnAS (nous avons modifié cette version pour JonasALaCarte), la version 4.1.3 de MySQL, la version 1.4.2 de RUBiS et la machine virtuelle Java JDK 1.4.1_01. Comparé à JOnAS, JonasALaCarte ne présente pas de surcoût en termes de consommation CPU et de mémoire. Avec les deux serveurs, la moyenne de la consommation CPU est autour de 70%. Une moyenne de la mémoire consommée sur 10 expériences donne un surcoût de 0,8% avec JonasALaCarte. Nous pensons que ce surcoût est dû à la création de nouveaux objets résultant du travail de ré-ingénierie. Par ailleurs, nous avons enregistré le même nombre de requêtes par seconde (9 requêtes/s pour les 2 serveurs). Nous expliquons ce surcoût négligeable par le fait que le travail de ré-ingénierie est effectué à gros-grain et le nombre d'objets crées est négligeable devant les objets du serveur.

6. Travaux similaires

Nous listons trois types de travaux liés au notre : l'administration des serveurs J2EE, les systèmes dont l'administration est fondée sur l'architecture et l'administration des grappes.
L'administration des serveurs d'applications J2EE (JOnAS, Geronimo et JBoss) est basée sur JMX pour l'instrumentation du serveur. Cependant, comme expliqué dans la section 2, l'architecture reste cachée à l'exécution ce qui limite la reconfiguration à la granularité du service. Ceci est dû à la limitation du modèle JMX à exprimer des architetcures explicites en termes de liaisons entre les MBeans et d'encapsulations. Par ailleurs, ces serveurs sont exécutés dans une seule machine virtuelle Java et des fichiers de scripts ad hocs sont écrits pour le déploiement dans des environnements distribués. Dans notre cas, le déploiement est automatisé avec Fractal ADL et l'unité de déploiement est le service.

Concernant, la tolérance des pannes, JAGR [11] a montré que le redéploiement partiel est pertinent pour la réparation des pannes dans les applications J2EE. Nous montrons ce résultat pour l'intergiciel.
Par ailleurs, plusieurs travaux adoptent une administration fondée sur l'architecture [14], ce qui signifie que les fonctions d'administration se basent sur un modèle causalement connecté au système à l'exécution. Nous citons par exemple les systèmes basés sur Darwin [16] et SmartFrog [17].

Il y a deux différences entre Darwin et notre travail. D'abord, nous exploitons le modèle Fractal qui offre plus de flexibilité à l'exécution. Ensuite, notre système d'administration présente une approche réflexive où les entités d'administration (les sondes, les gestionnaires et les actionneurs) sont aussi des composants Fractal et peuvent être supervisés et reconfigurés au même titre que les autres composants. Enfin, comme notre canevas, Darwin maintient une représentaion explicite du système à l'exécution (réstreinte aux applications et ne couvre pas l'infrastructure sous-jacente). SmatFrog offre un langage de configuration qui peut être utilisé pour décrire l'application à déployer et un langage de *workflow* permettant de décrire des opérations de déploiement complexe. Contrairement à notre travail, Smartfrog est limité au déploiement et les opérations de reconfiguration sont limitées et fixés au cycle de vie des composants.
Par ailleurs, il y a plusieurs travaux autour des grappes et des environnements distribués à grande échelle. En revanche, la majorité [6][5][15][13] traite plutôt l'aspect gestion des ressources. Nous pensons que nous pouvons facilement appliquer ces algorithmes et politiques en les intégrant comme des composants dans notre canevas.

7. Conclusion and perspectives

Les serveurs d'applications libres constituent un assemblage complexe de services. L'administration est particulièrement difficile dans des environnements distribués comme les grappes. Dans ce papier, nous avons présenté une approche d'administration fondée sur l'architecture pour simplifier et automatiser les fonctions d'administration des serveurs, notamment dans les environnements grappes. Dans notre approche, l'architecture du système à administrer est exposée d'une manière explicite sous la forme de composants hiérarchiques, explicitement liés. De plus, cette architecture est flexible et modulaire ce qui permet d'effectuer des opérations de reconfiguration en réponse à des évènements extérieurs comme les pannes.
Nous avons appliqué notre approche à JOnAS en encapsulant les différents services du serveur dans des composants Fractal et en traduisant les relations entre eux utilisant des liaisons Fractal explicites, localisées dans une même JVM ou distantes. A cette granularité, nous avons effectué ce travail de réingénierie avec un surcoût minimal et en conservant les performances du serveur JOnAS. Nous avons obtenu un déploiement automatisé dans des environnements distribués comme les grappes et des capacités de reconfiguration autonomes comme la réparation des pannes.
Cette approche peut être facilement appliquée à d'autres types d'intergiciels ou systèmes distribués composés par un ensemble de services ayant des interfaces standardisées. Nous travaillons actuellement sur l'application de notre approche à la granularité des EJB ainsi que dans des environnements distribués à très grande échelle comme les grilles d'entreprises.

Bibliographie

1. Apache Tomcat. http ://jakarta.apache.org/tomcat.
2. J2EE : Java 2 Platform, Enterprise Edition. http ://java.sun.com/j2ee/index.jsp.
3. Java Open Application Server (JOnAS). http ://jonas.objectweb.org.
4. JOTM. http ://jotm.objectweb.org.
5. K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA based management of a computing utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.
6. Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves : a mechanism for resource management in cluster-based network servers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 90–101, Santa Clara, California, June 2000.
7. P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.-B. Stefani. Dream Types – A Domain Specific Type System for Component-Based Message-Oriented Middleware. In *4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05*, Lisbon, Portugal, September 2005.
8. S. Bouchenak, F. Boyer, D. Hagimont, S. Krakowiak, A. Mos, N. de Palma, V. Quéma, and J.-B. Stefani. Architecture-Based Autonomous Repair Management : An Application to J2EE Clusters. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS)*, Orlando, Florida, USA, October 2005.
9. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In *Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004)*, Edinburgh, Scotland, 2004.

10. E. Bruneton, T. Coupaye, and J.-B. Stefani. The Fractal Component Model Specification - ObjectWeb Consortium. http ://www.objectweb.org, June, 2005.
11. G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox". JAGR : An Autonomous Self-Recovering Application Server. In *Proceedings of the 5th International Workshop on Active Middleware Services*, 2003.
12. E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and Scalability of EJB Applications. In *Proceedings of OOPSLA'02*, Seattle, WA, USA, November 2002.
13. Jeffrey S. Chase, David E. Irwin, Laura E. Grit, Justin D. Moore, and Sara E. Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proceedings 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, Seattle, Washington, June 2003.
14. E. Dashofy, A. van der Hoek, and R. Taylor. Towards Architecture-Based Self-Healing Systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*. ACM, 2002.
15. Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP : an architecture for secure resource peering. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 133–148. ACM Press, 2003.
16. I. Georgiadis, J. Magee, and J. Kramer. Self-organizing software architecture for distributed systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*. ACM, 2002.
17. P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog : Configuration and Automatic Ignition of Distributed Applications. HP OVUA, 2003.
18. J. Kephart and D. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1), 2003.
19. J. Lindfors and M. Fleury. JMX-Managing J2EE with Java Management Extensions. Sams, The JBoss Group, 2005.
20. A. Mulder. *Apache Geronimo Development and Deployment*. Pearson Education, Inc, 2004.