INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE l'INPG

Spécialité: « Informatique : Systèmes et Communications »

préparée au laboratoire LSR-IMAG, projet SARDES,
dans le cadre de l'Ecole Doctorale
« Mathématiques Sciences et Technologies de l'Information »

présentée et soutenue publiquement par

Takoua Abdellatif Berrayana

Le 14 Septembre 2006

Apport des architectures à composants pour l'administration des intergiciels

Étude de cas : JonasALaCarte, un serveur d'applications J2EE administrable

---

Directeur de thèse :

Jacques Mossière

JURY

Prof.	Sacha	Krakowiak	Président
Prof.	Pierre	Sens	Rapporteur
Prof.	Lionel	Seinturier	Rapporteur
Prof.	Jacques	Mossière	Directeur de thèse
Dr.	François	Exertier	Examinateur
Prof.	Yassine	Lakhnech	Examinateur

Résumé

L'administration des systèmes informatiques modernes est une tâche de plus en plus complexe et coûteuse. En effet, l'architecture de ces systèmes n'est pas en géneral suffisamment modulaire pour permettre l'adaptation aux événements extérieurs (pannes, variation des performances, attaques de sécurité, etc.). Dans cette thèse, nous nous intéressons à l'administration des intergiciels qui a fait l'objet de moins de travail que l'administration des couches physiques et des systèmes d'exploitation.

Nous proposons, d'abord, une démarche pour rendre les intergiciels patrimoniaux administrables, c'est à dire facilement configurables, déployables et adaptables à l'exécution. Cette approche consiste à transformer l'architetcure des intergiciels en une architecture explicite à base de composants. Ensuite, nous proposons un système d'administration automatisant certaines fonctions de l'administration, telles que le déploiement, la mise à jour de code et la réparation des pannes.

Nous avons appliqué notre approche à l'administration d'un serveur d'applications J2EE patrimonial, JOnAS. JonasALaCarte est le nom du serveur J2EE administrable obtenu. Nous montrons que la configuration et le déploiement du serveur dans des environnements distribués sont simplifiés. De plus, de nouvelles propriétés de reconfiguration dynamique sont introduites comme la mise à jour dynamique de code et l'auto-réparation des pannes.

Abstract

The management of modern IT systems is increasingly costly and complex. Indeed, the architecture of these systems is generally monolithic and does not allow for dynamic adaptation to external changes (failures, performance variation, security attacks, etc). In this thesis, we are interested in the management of Middleware, subject of less work than hardware and operating systems management.

First, we propose an approach transforming legacy Middleware to administrable (easily configurable and adpatable) ones. This approach consists in changing their architecture to an explicit component-based one. Second, we propose a management system automating the deployment, the dynamic code update and the recovery from failure operations.

As a use case, we considered a legacy J2EE application server, JOnAS. JonasALaCarte is the name of the obtained J2EE server. We show that our approach to the administration simplifies the server configuration and deployment in distributed environments. Furthermore, we introduced new management features to the server such as the dynamic code update and the self-recovery from failures.

Remerciements

Je remercie chaleureusement toutes les personnes qui ont rendu cette thèse possible et notamment :

Mes premiers remerciements sont adressés à Sacha Krakowiak, Professeur à l'Université Joseph Fourier, pour m'avoir fait l'honneur de présider le jury de ma thèse.

Je remercie Lionel Seinturier, Professeur à l'Université des Sciences et Technologies de Lille et Pierre Sens, Professeur à l'Université Pierre et Marie Curie d'avoir accepté d'être rapporteurs de cette thèse. Ils ont évalué mon travail d'une manière approfondie et m'ont fait des remarques constructives. Je remercie également Yassine Lakhnech, Professeur à l'Université Joseph Fourier et François Exertier, chef du projet JOnAS à Bull, d'avoir accepté d'être examinateurs de ce travail.

J'exprime ma gratitude à Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, mon directeur de thèse, pour son encadrement, ses nombreux conseils, sa disponibilité, son soutien et sa gentillesse à mon égard.

Au delà du jury, je tiens à remercier tous ceux qui ont permis à ces travaux d'aboutir, par leurs conseils, leurs contributions et leurs encouragements.

- Merci à tous les membres de l'équipe JOnAS à Bull pour leur accueil et pour avoir été toujours disponibles pour répondre à mes questions sur JOnAS. Je remercie particulièrement Adriana Danes et François Exertier pour leur confiance et leur encouragement.
- Merci à Emmanuel Cecchet d'avoir accepté de diriger mes premiers pas dans la recherche et d'avoir trouvé le financement de ma thèse.
- Merci à Jean-Bernard Stéfani de m'avoir accueillie au sein du projet Sardes. Malgré son emploi de temps trés chargé, il a toujours été disponible pour discuter de ma thèse et pour m'aiguiller vers des pistes de travail intéressantes. Je lui suis reconnaissante de son encouragement et de ses conseils précieux.
- Merci à Sacha Krakowiak pour tout ce qu'il m'a appris. Grâce à Sacha, j'ai pris conscience durant cette thèse, qu'il est primordial de bien définir la problématique dans un travail de recherche, d'être rigoureux dans le choix de la terminologie et qu'une publication a toujours besoin de plusieurs itérations avant d'atteindre un bon niveau. De plus, depuis mon DEA, assister aux présentations et aux cours très pédagogiques de Sacha a toujours été pour moi un vrai plaisir.
- Merci à Jakub Kornaś pour son aide précieuse dans la conception et le développement de JonasALaCarte. J'ai eu le plaisir de travailler avec lui et j'ai toujours apprécié sa grande modestie.
- Merci à Vivien Quéma pour tout le temps qu'il a passé à mes cotés, pour la structuration et la correction de mon manuscrit de thèse et pour ses remarques constructives pour améliorer la qualité de mes transparents de soutenance. Grâce à Vivien, j'ai beaucoup amélioré

mon style d'écriture, particulièrement en français. Je le remercie également de m'avoir fait partager son amour de la musique et son expérience de post-doc.
- Merci à Fabienne pour son amitié et sa confiance. Sa présence dans le projet m'a beauoup aidée dans les moments difficiles. Je remercie également les autres membres permanents de l'équipe Jade (Baz, Sara et Nono) de l'intérêt qu'ils ont montré envers mon travail ainsi que les assistantes du projet (Valérie et Élodie) pour leur sympathie et leurs services.
- Merci à Simon et à Renaud, mes premiers collègues de bureau. En plus de leur gentillesse à mon égard, je leur dois beaucoup de ce que j'ai appris sur l'utilisation des grappes et l'évaluation des performances.
- Merci à Ouss, Matthieu et Erdem pour leur sympathie et leur encouragement.
- Merci aux jeunes thésards (Christophe, Jeremy, Sylvain) pour toutes les discussions intéressantes que nous avons eu et pour m'avoir aidée dans les évaluations de performance sur JonasALaCarte.
- Merci aux membres de l'équipe formelle (Alan, Serguei, Mikael) pour les moments très agréables que nous avons partagés au LRP et pendant les pauses café. Merci à Alan pour m'avoir suggéré d'introduire les lettrines dans ce manuscrit, pour toutes ses blagues (même les ratées) et pour ses petites astuces de savoir-vivre.
- Merci à toutes les personnes sympathiques et intéressantes que j'ai eu l'occasion de côtoyer aux cours de ces dernières années (au sein de HP, de Sardes, de l'Inria, des laboratoires LSR et ID, de l'Ensimag ou ailleurs) et qui voudront bien m'excuser de ne pas les nommer.

Enfin, je garde une place toute particulière pour ma famille et mes proches, pour tout ce qu'ils ont pu (et continuent à) m'apporter. Merci à mes beaux-parents pour leurs prières et pour avoir gardé ma fille pendant la période de rédaction. Merci à mes soeurs et mon frère, particulièrement à Tesnim et Tahia pour leur grand soutien et leur encouragement. Merci à mon mari Wassel pour m'avoir encouragée à oser quitter mon travail chez HP et faire cette thèse. Merci pour sa patience et pour sa présence inestimable à mes côtés. Merci à mes très chers enfants Lina et Marwen d'avoir supporté une maman souvent débordée. Enfin, merci à mes parents pour m'avoir fourni toutes les bonnes conditions pour réussir mes études et pour m'avoir apporté, en plus de leur amour, le goût du travail.

Table des matières

Table des figures

Chapitre 1

Introduction

Dans cette thèse, nous nous intéressons à l'automatisation des fonctions de l'administration des systèmes informatiques qui est devenue de plus en plus complexe et coûteuse. Dans cette introduction, nous commençons par une analyse des défis de l'administration des applications réparties. Nous présentons par la suite notre contribution ainsi que l'organisation de ce document.

1.1 Les défis de l'administration d'applications réparties

Les systèmes informatiques modernes intègrent un nombre croissant de processeurs hétérogènes interconnectés par des réseaux aux caractéristiques diverses comme les réseaux locaux, les satellites ou le réseau téléphonique commuté. Cette diversification des équipements engendre une complexification des applications réparties modernes. Celles-ci sont aujourd'hui constituées d'un grand nombre de composants logiciels développés de façon indépendante. Les environnements actuels sont également caractérisés par un dynamisme croissant, qu'il soit des utilisateurs, de leurs besoins ou encore des caractéristiques physiques du système sur lequel est déployée l'application.

Pour faire face à cette évolution des systèmes informatiques, il est primordial de développer des systèmes d'administration prenant en charge différentes fonctions.

- *Configuration et déploiement :* la première tâche de l'administration consiste à configurer et à déployer le système. Cela veut dire choisir et assembler les composants du système et les placer sur des sites physiques. Le système de déploiement doit gérer le code des composants depuis la phase d'empaquetage jusqu'au chargement et à l'instanciation des composants. Il doit également fournir des mécanismes pour la mise à jour de code.
- *Réparation des pannes :* l'occurrence d'une panne (machine, réseau ou logicielle) peut faire basculer l'application dans un état d'erreur et la rendre indisponible pour les besoins des usagers. L'application n'est alors plus à même d'assurer le minimum de qualité de service requis par ses utilisateurs. Il est donc important de réparer le système de façon à le ramener à la configuration qu'il avait avant la panne.
- *Gestion des variations de performances :* les systèmes doivent assurer une certaine qualité de service. Ils doivent, par conséquent, être adaptés aux différentes variations de performances et aux pics de charge.
- *Gestion des ressources :* les ressources utilisées dans un système, comme les processeurs, les systèmes de stockage, les E/S et la bande passante des réseaux peuvent être partagées par

plusieurs applications. La gestion des ressources qui a un impact important sur les performances doit tenir compte de la qualité de service et des priorités de chaque application.
- *Gestion de la sécurité :* un système informatique doit résister aux attaques malveillantes. Par conséquent, les acteurs impliqués dans un système (utilisateurs, administrateurs) doivent agir sous une ou plusieurs autorités.

La mise en œuvre de ces fonctions d'administration est une tâche complexe. En effet, elle nécessite la prise en compte de différents facteurs.

- *Complexité des systèmes administrés :* les systèmes administrés doivent fournir à l'administrateur des interfaces permettant de les observer et de les adapter. Néanmoins, plusieurs systèmes, notamment les systèmes patrimoniaux dont la conception n'a pas pris en compte les besoins d'administration, ne facilitent pas la mise en œuvre de certaines reconfigurations. Il est nécessaire que le système d'administration crée des méta-niveaux et des abstractions exposant une vue plus exploitable du système administré.
- *Distribution et passage à l'échelle :* les systèmes distribués à grande échelle nécessitent des infrastructures adéquates pour permettre à l'administrateur d'accéder aux différentes parties du système. Pour satisfaire cette contrainte, les systèmes d'administration sont, aujourd'hui, des applications réparties à part entière pouvant être complexes. Ces systèmes d'administration nécessitent d'être configurés, déployés et adaptés aux événements extérieurs, au même titre que les systèmes administrés.
- *Diminution du rôle de l'être humain :* avec la complexité croissante des systèmes distribués modernes, les compétences des administrateurs humains montrent leurs limitations. De plus, le coût de l'administration devient de plus en plus important. Il devient donc nécessaire d'automatiser les fonctions de l'administration qui peuvent l'être.

L'administration des couches basses (système d'exploitation, réseaux) a suscité beaucoup d'attention par le passé. En particulier, les travaux menés ont abouti à la création d'un standard largement utilisé, appelé SNMP(*Simple Network Management Protocol*). Dans cette thèse, nous nous intéressons à l'administration des intergiciels qui a fait l'objet de moins de travaux. Notre contribution se décompose en deux parties. D'une part, nous proposons une technique de construction de *systèmes administrables*. Un système administrable est un système sur lequel il est possible d'effectuer dynamiquement des opérations de gestion de configuration. D'autre part, nous proposons un *système d'administration* automatisant certaines fonctions de l'administration comme le déploiement, la mise à jour de code et la réparation des pannes. Ce système d'administration base ses actions sur une connaissance de l'architecture du système administré. Il est ainsi développé suivant le principe récemment proposé d'*administration fondée sur l'architecture* [DvdHT02].

Dans le cadre de nos expérimentations, nous avons appliqué les principes de l'administration fondée sur l'architecture aux serveurs d'applications J2EE. Ce choix a été motivé par deux raisons.

- Un serveur J2EE est un intergiciel formé de plusieurs *services* hétérogènes, possédant une architecture complexe qui peut être centralisée ou distribuée. Un service implante des propriétés dites non-fonctionnelles (transactions, sécurité, protocoles de communication, etc.). Dans les serveurs J2EE libres, ces services sont généralement implantés par différents contributeurs, ce qui complexifie leur intégration et leur mise-à-jour. De plus, les serveurs d'applications J2EE doivent maintenir une qualité de service prédéfinie en termes de performance et de disponibilité. Enfin, le coût de l'administration des serveurs J2EE augmente

proportionnellement au nombre des applications Web et de leurs utilisateurs. Par conséquent, les industriels cherchent à automatiser certaines fonctions de l'administration. Nous retrouvons donc, dans l'administration des serveurs d'applications J2EE, une grande partie des défis de l'administration des intergiciels d'une manière générale.

– Ce travail s'inscrit dans le cadre du consortium ObjectWeb [obj] qui travaille sur la future version de JOnAS, appelée JOnAS5. L'un des objectifs primordiaux de JOnAS5 est que son administration soit automatisée, notamment dans les environnements distribués de type grappes.

1.2 Construction d'intergiciels administrables

1.2.1 Principe

Un prérequis pour bâtir des systèmes administrables est l'adoption d'une architecture modulaire dans laquelle les interfaces et les dépendances entre les différentes parties sont clairement identifiées. Dans le cadre de cette thèse, nous adoptons un modèle à composants, appelé Fractal, autorisant la construction de structures distribuées hiérarchiques. Nous décrivons une démarche permettant de rendre administrable un logiciel patrimonial. En particulier, nous identifions plusieurs étapes dans le travail de réingénierie nécessaire, avec pour but de rendre le système administrable, tout en respectant les standards et en minimisant le surcoût en performances induit par la réingénierie.

De façon simplifiée, le travail de réingénierie consiste à encapsuler les éléments du système à administrer dans des composants. Pour ce faire, les étapes que nous avons retenues sont les suivantes : identification des parties du système et de leurs dépendances, identification des paramètres de configuration et empaquetage des composants.

1.2.2 JonasALaCarte : un serveur d'applications J2EE administrable

La démarche de transformation d'un logiciel patrimonial pour le rendre administrable présentée dans la section précédente a été validée par la construction d'un serveur d'applications J2EE [Sun] configurable. Ce serveur, appelé JonasALaCarte, est un serveur d'applications J2EE à composants, obtenu par un travail de réingénierie du serveur JOnAS. L'architecture de JonasALaCarte est explicite : les services sont des composants et leurs dépendances sont bien définies. Les composants sont structurés d'une manière hiérarchique et assemblés par des liens explicites (*bindings*). La configuration et le déploiement du serveur se basent sur un langage de description d'architecture ou ADL (*Architecture Description Language*). L'ADL nous permet d'obtenir des configurations *à la carte* selon l'environnement ciblé et d'automatiser le déploiement des composants dans cet environnement. La gestion des services hétérogènes est uniformisée grâce à l'abstraction fournie par les composants offrant des interfaces communes de supervision et de configuration. Les composants dans JonasALaCarte sont des unités de configuration, de déploiement et de reconfiguration.

1.3 Construction de systèmes d'administration

1.3.1 Principe

Nous proposons une infrastructure d'administration automatisant certaines fonctions de l'administration (déploiement, mise à jour de code, réparation de pannes). Cette infrastructure est basée sur le principe de l'*administration fondée sur l'architecture*. Son implantation repose sur Jade [BBH+05], un canevas d'administration de systèmes autonomes développé au sein du projet Sardes. Jade donne l'accès à une vue de l'architecture du système et fournit des mécanismes permettant de maintenir cette vue lorsque le système est modifié suite à des événements extérieurs. Par ailleurs, Jade fournit des protocoles de reprise après panne. Dans le cadre de cette thèse, nous avons étendu Jade afin qu'il permette de déployer des architectures distribuées et de procéder à la reconfiguration dynamique de leur implantation. Cette extension consiste principalement en la définition d'un format d'empaquetage des composants, ainsi que de mécanismes d'installation et de chargement de classes.

1.3.2 Administration de JonasALaCarte

Les outils de configuration et de déploiement utilisés dans les serveurs d'applications J2EE « libres », comme JOnAS [jon], JBoss [LF] et Geronimo [Mul04] sont généralement restreints à des scripts ad hoc. Ceci peut induire l'occurrence d'erreurs de configuration et nécessiter un temps considérable pour l'administrateur, notamment dans des environnements distribués de type grappe.

Nous avons intégré dans JonasALaCarte le système d'administration introduit dans la section précédente. Ce système permet d'automatiser un grand nombre de fonctions d'administrations liées à la tolérance aux pannes et au redimensionnement du serveur. Il permet notamment de superviser de façon autonome le serveur J2EE grâce à l'utilisation de boucles de commande semblables à celles utilisées dans la théorie de la commande [Oga97]. L'architecture à composants du système d'administation permet d'intégrer des politiques d'administration sophistiquées sous la forme de composants.

1.4 Évaluation

Outre une comparaison de notre travail avec l'état de l'art, nous avons réalisé diverses expériences permettant d'évaluer notre travail. Afin de prouver que les concepts défendus dans cette thèse étaient réalisables, nous avons configuré et déployé JonasALaCarte sur une grappe de machines. Cette expérience permet de montrer que la réalisation d'une version administrable de JOnAS s'est faite sans rompre les standards implantés par JOnAS. Elle permet également d'évaluer le coût de la réingénierie.

Par ailleurs, nous avons évalué le système d'administration proposé. Tout d'abord, nous avons montré qu'il fonctionne aussi bien en centralisé qu'en distribué, et qu'il permet de réduire les temps de déploiement de façon significative. Nous avons également mené des expériences pour illustrer les mécanismes de reconfiguration qu'il propose. Tout d'abord, nous illustrons le mécanisme de reconfiguration d'implantation à travers l'exemple de la reconfiguration du service Web (Tomcat) en cours d'exécution. D'autre part, nous présentons un scénario d'auto-réparation du serveur permettant de traiter le cas des pannes logicielles et matérielles.

Enfin, nous proposons une évaluation de performances de JonasALaCarte. Celle-ci montre que le travail de réingénierie à gros grain de JOnAS a un surcoût négligeable sur les performances comparé au serveur d'origine.

1.5 Organisation du document

Ce document s'organise en deux parties : la première partie (chapitre 2, 3 et 4) présente l'état des domaines considérés dans cette thèse et la deuxième partie (chapitre 5, 6, 7 et 8) est consacrée à la description du serveur J2EE JonasALaCarte et de son système d'administration intégré.

Le chapitre 2 est découpé en deux parties. La première définit les différentes fonctions de l'administration de systèmes avec un intérêt particulier pour l'administration d'intergiciels. Cette partie introduit également l'administration fondée sur l'architecture et les principes de l'administration autonome. La deuxième partie de ce chapitre présente des exemples récents de systèmes d'administration fondés sur l'architecture.

Le chapitre 3 est dédié à l'étude des architectures de composants pour la construction de systèmes administrables. Pour ce faire, nous avons étudié différents modèles de composants industriels et académiques selon les critères suivants : la capacité du modèle pour créer une architecture logicielle explicite et les outils qu'ils offrent pour faciliter l'administration, notamment les langages de description d'architectures pour le déploiement.

Le chapitre 4 est consacré à la présentation de l'environnement J2EE. Il introduit l'architecture J2EE et le déploiement des serveurs J2EE dans les environnements distribués. Il présente en particulier l'environnement grappe qui est actuellement le centre d'intérêt des industriels et nous décrivons les problèmes liés à la configuration et au déploiement dans cet environnement. Enfin, nous discutons le rôle de l'architecture dans l'administration des serveurs J2EE à travers une description de l'architecture des serveurs d'applications « libres ».

Le chapitre 5 est une présentation générale de notre approche à l'administration fondée sur l'architecture. Il présente l'architecture globale adoptée et les technologies sous-jacentes pour la mettre en œuvre. Le travail sur la réingénierie de JOnAS est décrit dans le chapitre 6 et nous décrivons le système d'administration intégré à JonasALaCarte dans le chapitre 7. Nous détaillons en particulier, les fonctions d'administration suivantes : la configuration, le déploiement, la reconfiguration dynamique et la tolérance aux pannes. Nous décrivons également le caractère autonome de JonasALaCarte.

Le chapitre 8 est dédié à l'évaluation de notre travail. Nous présentons une évaluation qualitative ainsi que les résultats quantitatifs des expériences menées dans cette thèse.

Enfin, nous présentons une conclusion de ce qui a été réalisé dans cette thèse et nous décrivons les perspectives de notre travail.

Chapitre 2

L'administration des intergiciels

2.1 Introduction

Comme représenté dans la figure 2.1, l'intergiciel est une couche logicielle située entre le système d'exploitation et les applications. Il contient un ensemble de services de gestion de la distribution et de la coopération entre les applications, et fournit à ces dernières un modèle et une interface de programmation. Les intergiciels peuvent être soit centralisés sur une même machine soit distribués.

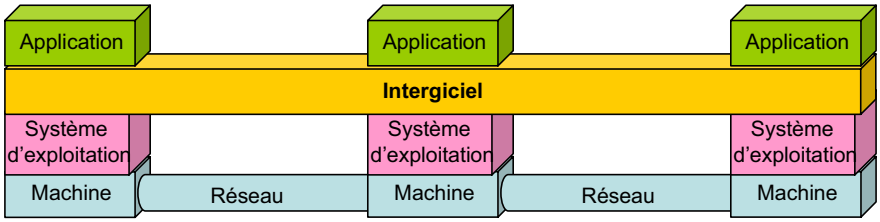

Fig. 2.1 – *Positionnement de la couche intergiciel.*

Plusieurs efforts ont été consacrés à l'élaboration de modèles et techniques pour l'administration des systèmes d'exploitation et des réseaux. L'administration des couches applicatives et de l'intergiciel a suscité moins d'intérêt. Néanmoins, la complexité croissante des intergiciels et la nouvelle tendance vers les intergiciels ouverts a encouragé les travaux des industriels et des chercheurs autour de cette problématique.

Ce chapitre est consacré à la définition et à l'analyse des différentes fonctions de l'administration, notamment l'administration des intergiciels[1]. Nous présentons tout d'abord les limitations

1. Les définitions et les exemples présentés dans ce chapitre sont inspirés d'un document non publié et écrit par Sacha Krakowiak.

du modèle agent-gestionnaire qui est classiquement adopté dans l'administration des systèmes d'exploitation et des réseaux. Nous détaillons ensuite les défis de l'administration. En particulier, nous introduisons les notions d'*administration fondée sur l'architecture* et d'*administration autonome* (*autonomic computing*) qui vont être traitées plus en détails dans les chapitres suivants. Enfin, nous présentons quelques systèmes d'administration récemment proposés qui respectent le principe de l'administration fondée sur l'architecture.

2.2 Fonctions de l'administration

Cette section présente différentes fonctions de l'administration, à savoir la configuration et le déploiement, l'observation, la reconfiguration, la gestion des ressources, la tolérance aux fautes et la sécurité.

2.2.1 Configuration et déploiement

Dans cette section, nous présentons la première fonction de l'administration qui consiste à configurer le système et à le déployer. Nous montrons la complexité de cette tâche généralement sous-estimée par les administrateurs d'intergiciels. En effet, outre la mise en place du système, nous montrons que les outils de configuration et de déploiement sont nécessaires pour reconfigurer dynamiquement le système en cours d'exécution.

Pour démarrer un intergiciel construit sous la forme d'un ensemble de composants, les tâches suivantes doivent être accomplies.

- Choisir les composants à utiliser (certains peuvent exister sous plusieurs versions) et positionner les paramètres initiaux. Les composants sont généralement installés dans un *dépôt* qui peut être centralisé ou distribué.
- Vérifier la cohérence du système, par exemple, des dépendances entre les composants.
- Déterminer les *cibles* qui sont les machines sur lesquelles le système doit être installé, placer chaque composant sur la cible correspondante et établir les connexions entre les composants. Les cibles sont généralement formées d'un ensemble de machines interconnectées par un réseau et contiennent déjà un minimum de logiciel installé (par exemple, des systèmes d'exploitation, des routeurs logiciels, etc.).
- Démarrer les composants dans un ordre approprié.

Le déploiement a été longtemps considéré comme une extension simple de la configuration des systèmes informatiques et pas vraiment comme une thématique importante. Les outils de déploiement sont généralement ad hoc et sont sous la forme de scripts d'installation écrits manuellement. Par conséquent, les incohérences dues à la configuration et au déploiement sont identifiées comme étant la cause principale des fautes dans les systèmes distribués [OGP03].

Avec la complexité croissante des plates-formes matérielles et des systèmes, une nouvelle génération d'outils est apparue basée sur des principes conceptuels et utilisant des représentations du système de haut niveau et plus explicites. Ainsi, la sélection et le placement des composants sont effectués selon une politique bien choisie, exprimée généralement selon une forme déclarative. La politique peut directement spécifier les composants et les emplacements à utiliser, exprimer des préférences ou alors définir des objectifs de haut niveau (par exemple, en termes de performance ou de disponibilité) à partir desquels le système d'administration tente de dériver des choix pour atteindre ces objectifs.

La figure 2.2 représente plus en détails les opérations liées à la configuration et au déploiement d'un système [Hal04] [HHW99].

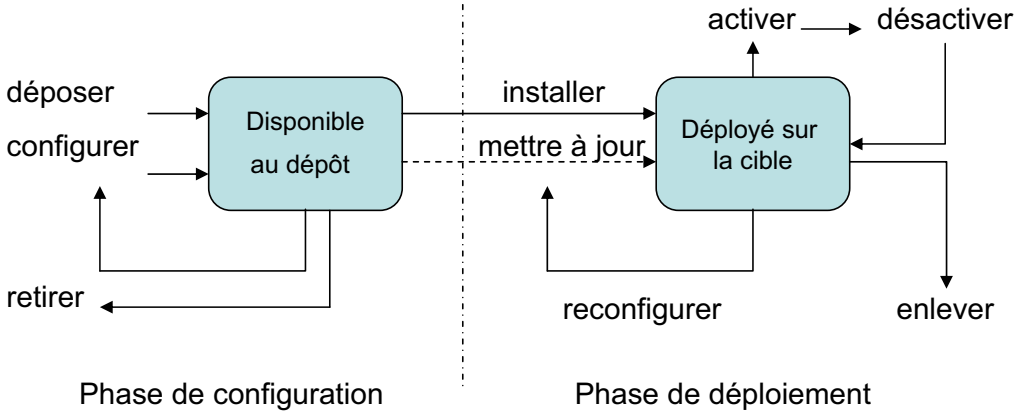

FIG. 2.2 – *Cycle de vie de la configuration et du déploiement d'un système.*

Afin de faciliter leur présentation, nous illustrons la configuration et le déploiement en deux phases dinstinctes. En pratique, ces phases sont fusionnées et respectent le cycle de vie suivant.

- *Déposer* et *retirer* le logiciel consiste à le rendre disponible sous plusieurs formes (des codes source, des modules ou des unités exécutables). Retirer un composant, devenu par exemple obsolète, le rend non disponible au niveau du dépôt.
- *Configurer* permet de définir les propriétés des composants à installer, par exemple, définir leurs paramètres au démarrage et leurs versions.
- *Installer* consiste à mettre les composants sur les machines cibles et à les assembler.
- *Activer* et *désactiver* permettent de lancer/stopper l'exécution des composants.
- *Retirer* un composant consiste à le désinstaller de la cible.
- *Reconfigurer* un composant en cours d'exécution consiste à en modifier la structure et l'organisation vis-à-vis des autres composants.

La configuration et le déploiement de systèmes distribués est une tâche très complexe notamment pour le passage à l'échelle et la tolérance aux fautes. De plus, les problèmes liés au versionnement (la coexistence de plusieurs versions d'un système) et à la cohérence de configurations sont loin d'être entièrement résolus. En effet, dans les systèmes traditionnels, on maintient généralement l'historique des différentes versions des composants mais on ne présente pas les dépendances entre ces versions dans une description à un plus haut niveau. Les dépendances qui ne sont pas exprimées d'une manière explicite peuvent induire des erreurs de compatibilité si on change une version d'un composant par une autre ou si on effectue une autre forme de reconfiguration statique ou dynamique. Ces problèmes ont mis en évidence le besoin d'unifier les outils de configuration et de déploiement [vdHHW98] et d'adopter des descripteurs d'architecture illustrant explicitement les dépendances entre les composants (en termes de liaisons et de compatibilité de versions). C'est ce qu'on entend par le *déploiement fondé sur l'architecture*.

2.2.2 Observation

La connaissance de l'état d'un système est un prérequis pour pouvoir l'administrer. Ceci inclut plusieurs aspects :

- l'information sur la configuration du système,

- l'information sur la charge et les performances du système,
- l'information sur l'évolution du système, c'est à dire les différents événements qui se passent dans le système au cours de son exécution.

Ces informations peuvent être présentées sous plusieurs formes : d'une manière instantanée, sous forme de graphes, par exemple, ou sous forme d'un ensemble de traces (*logs*) qui peuvent être exploitées a posteriori. Un aspect important de l'observation est la spécification de l'information à collecter. Cette spécification est utilisée, par exemple, pour la mise en place de filtres servant à éliminer les données et les événements considérés comme inutiles à analyser. Les systèmes d'observation ou *monitoring* doivent satisfaire les critères suivants.

- **Intrusivité minimale** : les sondes ou capteurs introduits dans les systèmes observés ne doivent pas être perturbateurs. Ils ne doivent pas changer le comportement des systèmes ni rajouter un surcoût en termes de ressources consomées car cela pourrait dégrader les performances du système.
- **Fidélité** : le système d'observation doit être fidèle à ce qui est réellement observé et livrer des résultats non modifiés.
- **Flexibilité** : le système d'observation doit être facilement adaptable durant son fonctionnement. Par exemple, il doit être possible de changer ses paramètres (période de remontée des évènements, critères de filtrage des informations, etc.). Il doit permettre également de rajouter dynamiquement des parties du système (sondes, filtres, etc.) sans arrêter totalement le système.
- **Passage à l'échelle** : le système d'observation doit passer à l'échelle pour l'observation de systèmes à grande échelle.

Face à ces défis, la construction de systèmes d'observation est encore un sujet de recherche actif. Des travaux récents proposent des systèmes d'observation flexibles [CLQ05], ou encore des systèmes se focalisant sur le passage à l'échelle [vRBV03].

2.2.3 Reconfiguration

La reconfiguration est la modification d'une configuration existante et déjà déployée. Les opérations suivantes relèvent de la reconfiguration :

- Changement des attributs d'un composant,
- Modification des liens entre composants,
- Ajout, suppression ou remplacement d'un composant sur un site,
- Migration d'un composant d'un site vers un autre.
- Modification de l'implantation d'un composant pour corriger une erreur dans le code ou pour installer une nouvelle version.

Notons que l'on distingue la *reconfiguration de structure* — qui correspond aux quatre premières opérations — de la *reconfiguration d'implantation* dont la définition est décrite dans le dernier point.

La reconfiguration suppose une connaissance du système administré puisqu'une modification d'une partie peut induire la modification d'autres parties de ce système. La reconfiguration doit respecter les propriétés suivantes.

- Les modifications du système doivent être exprimées en termes des éléments de sa structure, au moins à gros grains (nous parlons alors de *reconfiguration fondée sur l'architecture*).

Les spécifications d'une modification doivent être déclaratives, ce qui permet une séparation des spécification des changements (rôle de l'utilisateur) de l'implantation réelle de ce changement (rôle de l'outil de reconfiguration).
- Les spécifications doivent être indépendantes des algorithmes et des protocoles de l'application à reconfigurer. Ceci permet de bâtir des systèmes de reconfigurations génériques, des mécanismes et des patrons réutilisables.
- Les modifications apportées au système doivent le laisser dans un état cohérent. Concrètement, des invariants du système doivent être définis et préservés après toute modification.
- Toute modification du système ne doit pas trop dégrader les performances et ne doit pas faire dévier le système de son comportement prévu.

Le lecteur intéressé peut trouver dans [HW04] and [SAH+03] deux exemples de canevas de reconfiguration de systèmes essayant de tenir compte des propriétés citées précédemment.

2.2.4 Gestion des ressources

Lors de son fonctionnement, un système utilise plusieurs *ressources* comme les processus, les systèmes de stockage, les E/S et le réseau. L'allocation des ressources peut avoir un impact sur les performances de l'application. C'est une tâche traditionnellement effectuée par les systèmes d'exploitation et les utilisateurs des applications ont généralement peu de contrôle sur cette opération. Cette situation a changé pour les raisons suivantes :

- l'augmentation du nombre des applications ayant de fortes contraintes en termes de temps et d'espace comme les applications embarquées et les applications de gestion des données multimédia,
- la tendance vers les intergiciels ouverts adaptables à des environnements variables.

Ainsi, une grande partie de la gestion de ressources est déléguée aux couches supérieures, c'est à dire aux intergiciels ou aux applications elles-mêmes. Les couches basses du système prennent, dans ce cas, la responsabilité de partager équitablement les ressources entre les différentes classes de services. Le principe est de considérer l'application en meilleure position pour connaître précisément ses besoins et éventuellement adapter ses demandes en ressources selon son évolution. Ceci permet un usage optimisé des ressources tout en assurant un meilleur service de chaque application.

2.2.5 Tolérance aux fautes

Un système est sujet à une *panne* quant il dévie de son comportement attendu. La panne est la conséquence d'un état incorrect qui peut résulter d'une erreur humaine ou matérielle ou d'un événement naturel catastrophique tel qu'un incendie ou une inondation. Une cause de panne est généralement appellée une *faute*.

Plusieurs mesures de précaution peuvent être prises pour réduire la probabilité de l'occurrence des fautes. Néanmoins, les fautes ne peuvent être totalement éliminées et les systèmes informatiques doivent être conçus afin de permettre à leurs services de continuer à fonctionner même en présence de fautes. On parle alors de *disponibilité* comme un critère de qualité de service. La disponibilité est définie comme étant la fraction de temps pendant laquelle un système est à même d'offrir un service. Le but de la tolérance aux pannes est de garantir un degré de disponibilité, par exemple 99,999 %. La tolérance aux pannes est généralement assurée grâce à deux approches qui peuvent être combinées : la réduction de la probabilité d'interruption de services au moyen de la duplication, et la réduction du temps des réparations.

2.2.6 Sécurité

La sécurité est la qualité d'un système informatique capable de résister aux attaques malvaillantes. Ces attaques ont pour but de compromettre plusieurs propriétés du système, notamment :

- la **confidentialité** qui empêche que l'information soit consultée par des parties non autorisées,
- l'**intégrité** qui assure que toute modification de l'information est explicitement autorisée,

Afin d'atteindre ces propriétés, plusieurs techniques sont définies, par exemple :

- l'**authentification** qui assure qu'une personne (ou une autorité) est réellement celle qu'elle prétend être,
- la **certification** qui assure qu'un document ou une preuve d'identité est réellement valide (n'a pas été falsifiée ou n'a pas expiré).

Les propriétés précédentes imposent que les différents acteurs impliqués dans un système informatique (les administrateurs, les organisations ou leurs délégués logiciels) agissent sous une ou plusieurs *autorités* dont le rôle est de délivrer des autorisations. La spécification et l'implantation de la sécurité repose sur une relation de *confiance*.

2.3 Les défis de l'administration et les solutions existantes

Dans cette partie, nous nous intéressons aux défis de l'administration, notamment celle des intergiciels. Nous présentons le modèle d'administration le plus répandu : le modèle agent-gestionnaire. Nous montrons les limitations de ce modèle pour l'administration des intergiciels et nous introduisons les approches récemment proposées. En particulier, nous décrivons les concepts d'*administration fondée sur l'architecture* et d'*administration autonome*.

2.3.1 Le modèle agent-gestionnaire et ses limitations

L'approche actuelle pour l'administration des systèmes et des réseaux s'inspire des standards définis entre 1988 et 1990 : le protocole SNMP (*Simple Network Management Protocol*) pour le monde Internet et les protocoles CMIS/CMIP (*Common Management Information Services and Protocols*) spécifiés par les organisations ISO et ITU principalement pour le domaine des Télécommunications. Ces standards permettent l'inter-opérabilité entre les différents vendeurs. Ils sont basés sur un même modèle générique, appelé modèle agent-gestionnaire. Les éléments de ce modèle sont présentés sur la figure 2.3.

Le modèle repose sur les entités suivantes :

- Le **gestionnaire** agit en tant qu'interface pour l'administrateur humain. Il est généralement installé sur une machine dédiée à l'administration. Un gestionnaire peut intéragir avec un ou plusieurs agents.
- L'**agent** réside sur une machine administrée. Il est le délégué local du gestionnaire pour les objets administrés dont il consulte ou modifie l'état.
- La **MIB** (*Management Information Base*) définit l'organisation des objets administrés, ainsi que le modèle de données qui permet d'accéder à ces objets. Elle est utilisée à la fois par le gestionnaire et par les agents.

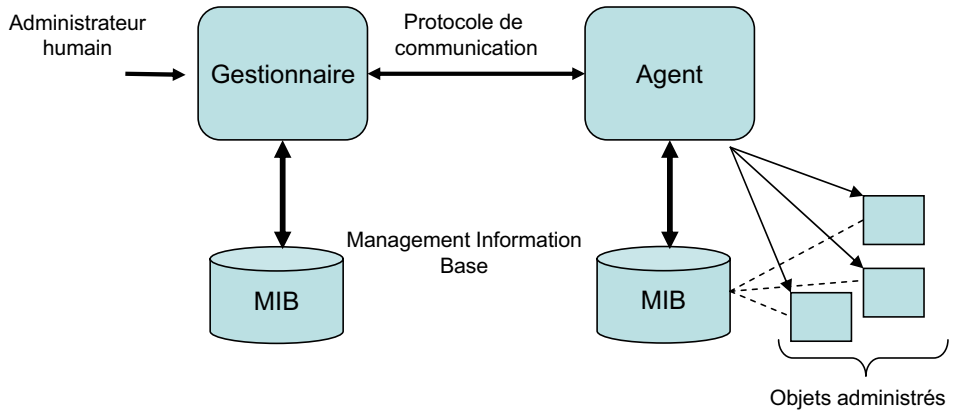

FIG. 2.3 – *Les éléments du modèle agent-gestionnaire.*

– Le **protocole d'administration de gestion** définit les interactions entre le gestionnaire et les agents.

Le principe du modèle agent-gestionnaire est de séparer le système d'administration des entités administrées. Il a été spécifié pour satisfaire plusieurs fonctions de l'administration telles que la configuration, la sécurité, la tolérance aux fautes et l'observation. Le lecteur intéressé pourra se référer à [JPMF] pour avoir une liste plus détaillée sur ces systèmes d'administration. Par ailleurs, dans [SPMF] sont listées les perspectives sur l'évolution des technologies d'administration spécifiques à Internet.

Une implantation typique de ce modèle est le standard SNMP (*Simple Network Management Protocol*) qui a été l'un des premiers protocoles permettant d'administrer des réseaux. Ses principaux avantages sont liés à sa simplicité. Les constructeurs de matériels peuvent superviser leurs appareils avec un minimum d'effort. SNMP permet d'effectuer les tâches de base d'administration en perturbant très peu les appareils et les réseaux administrés. Les ressources nécessaires pour mettre en place une infrastructure d'administration basée sur SNMP sont peu importantes. Ceci fait que le protocole SNMP s'est trés largement répandu et continue d'être utilisé intensivement.

Néanmoins, le protocole SNMP présente plusieurs inconvénients. SNMP manque d'automatisation. En effet, les opérations d'administration doivent être déclenchées manuellement par un administrateur. SNMP ne définit pas de méthodes permettant d'automatiser certaines tâches bien identifiées. Ceci est problématique pour les administrateurs devant contrôler des réseaux étendus composés de nombreux appareils. Par ailleurs, SNMP ne propose pas d'algorithme de déploiement et de reconfiguration. Les opérations d'administration doivent être mises en œuvre à la main. Le développement des agents SNMP et la cohérence des opérations d'administration sont laissés à la charge du programmeur. Le langage utilisé pour modéliser les ressources administrées, ASN1, n'est pas adapté à la modélisation d'applications réparties. Avec ce type de modélisation les administrateurs ont du mal à appréhender la structure complexe des applications réparties en cours d'exécution. Cette modélisation est orientée vers la spécification des protocoles de communication et non sur la spécification des applications réparties. De plus, les administrateurs n'ont pas à leur disposition une vision de la structure de l'application en cours d'exécution.

Les défauts de SNMP peuvent être généralisés au modèle agent-gestionnaire. Ce modèle se

base sur une architecture centralisée qui ne passe pas à l'échelle pour supporter la charge et pour résister aux fautes. Sa structure statique manque de flexibilité et ne peut donc s'adapter à la complexité ni à la dynamicité des applications réparties. De nouvelles tendances apparaissent au delà de ce modèle telles que l'*administration basée sur l'architecture* et l'*administration autonome* que nous décrivons dans les sections suivantes.

2.3.2 L'administration fondée sur l'architecture

L'*architecture* d'un système informatique décrit ce système en tant qu'assemblage de composants. Cette description se base en général sur un *modèle* servant à définir l'organisation du système et son fonctionnement. Le modèle d'un système sert à mieux le comprendre, à prédire son comportement et sert également à sa conception et à son implantation. La notion d'architecture des systèmes a été surtout exploitée pour la conception des systèmes et ce n'est que récemment qu'on s'est rendu compte qu'elle pourrait servir de base pour l'administration. En particulier, les parties identifiées du système peuvent être des unités de déploiement, d'isolation de faute et de reconfiguration. La notion d'*administration fondée sur l'architecture* traduit l'idée que les modèles d'architecture peuvent servir de base pour la construction des fonctions d'administration. Ce principe est de plus en plus utilisé grâce aux efforts récents pour développer des architectures dirigées par les modèles (*model-driven architecture*) [omg].

2.3.3 Vers une administration autonome

Actuellement, les fonctions d'administration sont généralement effectuées par des administrateurs humains utilisant des outils appropriés pour assurer le fonctionnement du système administré. Une grande partie des tâches d'administration ne sont pas formalisées et dépendent plutôt du savoir-faire de l'administrateur et de son expérience.

Avec la complexité croissante des systèmes et des applications, le coût de l'administration devient de plus en plus considérable et les difficultés de l'administration commencent à dépasser les compétences des administrateurs. Par conséquent, une nouvelle tendance est apparue qui consiste à automatiser (au moins en partie) les fonctions de l'administration. C'est l'objectif de ce qui est appellé *autonomic computing* [WHW+04] [AG03].

Le but de l'*autonomic computing* est d'offrir aux systèmes des capacités d'auto-administration. Ceci inclut l'auto-configuration (configuration automatique suivant des règles prédéfinies), l'auto-optimisation (contrôle et adaptation du système pour assurer un certain niveau de performance), l'auto-réparation (détection de pannes et correction automatique) et l'auto-protection (prise des mesures nécessaires pour se protéger des attaques malveillantes et savoir se défendre contre ces attaques).

L'architecture d'un système autonome a initialement été proposée par IBM [WHW+04]. La figure 2.4 représente l'organisation des éléments d'une telle architecture. Les différents éléments communiquent suivant une *boucle de commande*. Nous distinguons principalement deux parties : l'élément administré et le gestionnaire. Le gestionnaire est formé d'un ensemble de composants.

- Le **moniteur** ou superviseur collecte les données qui caractérisent le comportement du système.
- L'**analyseur** interprète ces données en se basant sur une connaissance du système administré.
- Le **planificateur** utilise des algorithmes dont le rôle est de déterminer un plan d'action.
- L'**actionneur distant** met en œuvre le plan en envoyant des commandes aux actionneurs de l'élément administré.

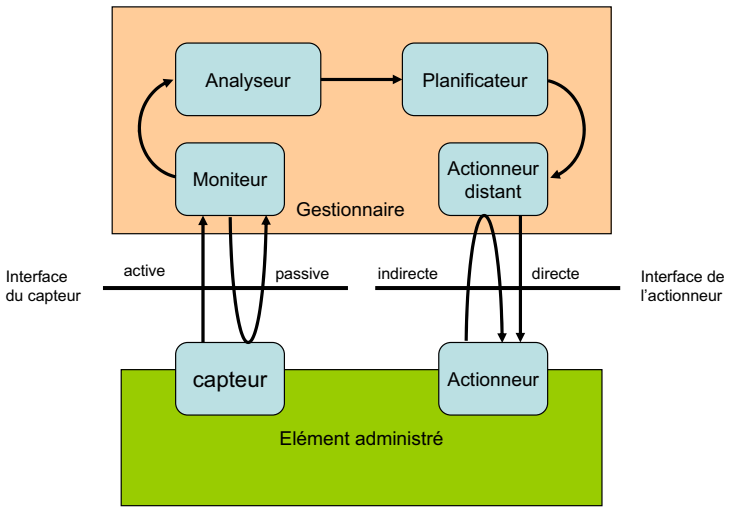

FIG. 2.4 – *Organisation d'un système autonome selon IBM.*

Le gestionnaire communique avec l'élément administré grâce aux **capteurs** et aux **actionneurs**. Les capteurs et les actionneurs offrent des interfaces principales implantées par le système administré pour observer son état et pour le modifier à des niveaux variables de granularité. Il y a deux modes de communication possibles : le mode *actif* — dans lequel les capteurs envoient régulièrement des informations de supervision — et le mode *passif* dans lequel les capteurs sont régulièrement consultés pour obtenir ces informations. De la même manière, l'actionneur distant et l'actionneur peuvent communiquer selon deux modes : le mode *direct* — quand l'actionneur distant envoie une requête de commande à l'actionneur — ou suivant le mode *indirect* (dit aussi de *callback*) quand l'actionneur distant demande plus d'instructions à l'actionneur pour accomplir une tâche.

L'*autonomic computing* est plus une vision future qu'une réalité industrielle. La preuve en est qu'il n'existe pas encore de standards dans ce domaine. Néanmoins, plusieurs initiatives et prototypes ont été débutés par les industriels pour mieux administrer leurs systèmes complexes et l'*autonomic computing* est devenu un sujet de recherche très actif. Un état de l'art sur les défis de l'*autonomic computing* est dressé dans [Kep05].

2.3.4 Synthèse

Les systèmes d'administration ont connu une grande progression depuis le modèle agent-gestionnaire jusqu'aux systèmes modernes. Le modèle agent-gestionnaire a été largement adopté pour l'administration des systèmes d'exploitation et des réseaux. Le protocole SNMP est l'exemple type du succès de ce modèle. Toutefois, face à la complexité des intergiciels et des applications modernes, ce modèle a montré ses limitations. En particulier, ce modèle ne permet pas de refléter l'architecture du système en cours d'exécution. Cependant, la description de l'architecture du système est fondamentale pour pouvoir le reconfigurer. L'approche actuellement adoptée est appellée administration fondée sur l'architecture. Dans cette approche, les fonctions d'administration, notamment la configuration, le déploiement et la reconfiguration, se basent sur une description de l'architecture et utilisent des éléments du système pour le mettre en place et l'adapter aux événements extérieurs. La tendance actuelle consiste également à minimiser l'intervention humaine dans l'adaptation du système en cours d'exécution. C'est le principe de l'administration auto-

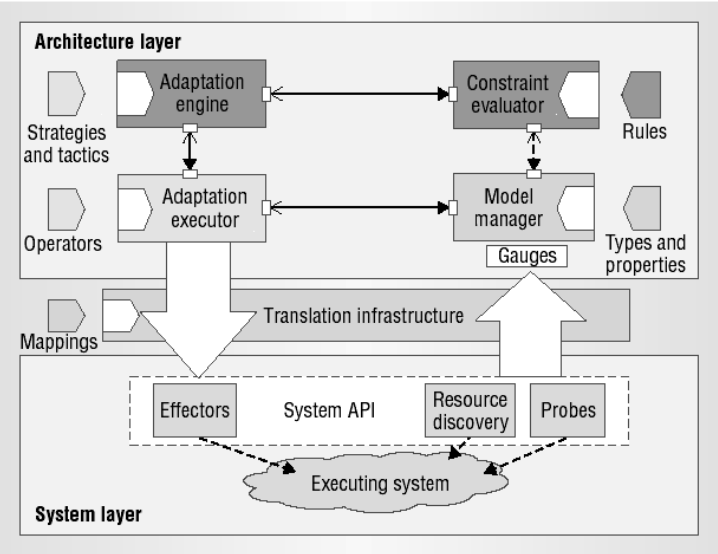

FIG. 2.5 – *Architecture du canevas Rainbow.*

nome que nous avons introduit dans cette section. Nous avons présenté également, l'architecture respectant une boucle de commande telle qu'elle a été proposée par IBM.

2.4 Systèmes d'administration fondés sur l'architecture

Dans cette section, nous présentons trois systèmes d'administration se basant sur les éléments d'architecture. Le premier système est Rainbow [CHG+04] qui est un canevas pour la contruction de systèmes auto-adaptables. SmartFrog [GGL+] est un canevas plutôt dédié au déploiement et au démarrage de systèmes distribués. Enfin, Nix [DJa04] offre une analyse rigoureuse du processus d'installation de code. Tous ces systèmes se basent sur un modèle de composants pour représenter l'architecture du système administré. Ils utilisent les éléments de cette architecture pour construire des mécanismes et des politiques d'administration.

2.4.1 Rainbow

Rainbow [CHG+04] est un canevas dédié à la construction de systèmes auto-adaptables. Rainbow distingue la couche d'administration de la couche du système administré pour permettre l'intégration de politiques d'administration indépendamment des mécanismes (e.g. sondes, canaux d'observation). Cette séparation permet également de réutiliser des parties de la couche d'administration (éventuellement étendus) pour différents types de systèmes.

L'adaptation dans Rainbow se base sur une boucle de commande. Les sondes envoient les informations d'observation à la couche administrative qui se base, d'une part, sur ces informations et, d'autre part, sur l'architecture du système, pour décider du type d'adaptation et de sa mise en œuvre.

La figure 2.5, extraite de [CHG+04], illustre la boucle de commande pour l'auto-adaptation. Rainbow se base sur un modèle abstrait d'architecture pour superviser le système à l'exécution. Ce modèle représente les différentes parties du système sous la forme de composants.

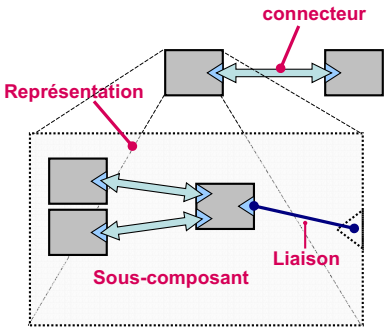

FIG. 2.6 – *Le modèle de composants utilisé dans Rainbow.*

La figure 2.6 représente le modèle de composants utilisé par Rainbow. Il s'agit d'un modèle hiérarchique qui représente le système administré à des niveaux d'abstractions différents. Les liaisons entre les composants sont représentées sous forme de connecteurs. Comme le montre l'exemple de la figure, les liaisons permettent d'exposer des interfaces internes au niveau d'un composant composite. Le modèle abstrait est utilisé pendant la conception du système à l'aide d'un ADL. Il est également consulté à l'exécution pour procéder à l'adaptation du système.

Une propriété intéressante de ce modèle est que les composants peuvent être annotés avec différentes propriétés telles que les débits attendus, les latences et les protocoles d'interaction. Il est également possible de rajouter des contraintes au niveau du modèle pour représenter l'ensemble des conditions qui doivent être vérifiées avant de valider un changement.

Un système intégrant Rainbow expose une architecture en plusieurs couches : la couche système, la couche d'architecture et la couche de traduction.

- **Infrastructure de la couche système.** A ce niveau, des interfaces d'accès sont implantées. Des sondes (*probes*) observent et mesurent les différents états du système. De plus, un service de découverte peut être sollicité pour allouer des ressources suivant des critères bien définis. Enfin, des actionneurs (*effectors*) permettent la mise en place des modifications.
- **Infrastructure de la couche d'architecture.** Des jauges (*gauges*) permettent d'agréger les informations envoyées par les sondes et de mettre à jour les propriétés appropriées dans le modèle d'architecture. Un gestionnaire de modèle (*model manager*) traite les informations et accède au modèle d'information. Un évaluateur de contraintes (*constraint evaluator*) vérifie périodiquement le modèle et détecte le besoin d'une adaptation si une contrainte est violée. Une machine d'adaptation (*adaptation engine*) détermine l'action à effectuer et met à jour l'adaptation correspondante.
- **Infrastructure de traduction.** Cette couche établit la correspondance entre le modèle et le système d'administration. Un dépôt dans l'infrastructure maintient la liste de correspondance entre la représentation en composants et l'implantation système. Par exemple, le système peut faire correspondre l'identificateur d'un composant à une adresse IP.

En se basant sur ces infrastructures, ajouter les politiques d'adaptation nécessite une connaissance spécifique du système telles que les cibles sur lesquelles ces systèmes sont déployés, les paramètres des composants, leurs types, leurs propriétés et les contraintes de leurs comportements ainsi que les stratégies d'adaptation. Suivant la politique d'administration et le contexte du travail, Rainbow définit un *style d'architecture*. Le style d'architecture identifie les propriétés

et les patrons communs à un ensemble de systèmes. Pour chaque style, Rainbow identifie les entités suivantes.

- **Les types de composants et de connecteurs** constituent une désignation des différentes parties du système et de leurs connections, par exemple Database, Client, Server et Filter. Des exemples de connecteurs incluent SQL, http, RPC et Pipe.
- **Les contraintes** permettent de déterminer les compositions autorisées des composants à partir de leurs types.
- **Les propriétés** sont des attributs des composants et les types des connecteurs. Elles fournissent des informations analytiques ou sémantiques sur ces composants. Par exemple, la charge et le temps de réponse sont des propriétés caractéristiques d'un serveur pour une analyse de performance.
- **Les analyses** sont construites suivant la nature du système administré. Par exemple, des analyses de performance peuvent être effectuées pour des applications de type client/serveur.

Pour définir les politiques d'adaptation, Rainbow définit :

- les **opérateurs d'adaptation** qui définissent les opérations de base comme ajouter un service ou éliminer un autre,
- les **stratégies d'adaptation** qui définissent les conditions dans lesquelles il faut effectuer les opérations d'adaptation.

Synthèse

Rainbow est un canevas dédié à la construction de systèmes auto-adaptables. Il respecte la boucle de commande introduite par IBM pour la construction de systèmes autonomes. Rainbow utilise un modèle de composants hiérarchique pour la représentation explicite de l'architecture du système administré. Le point fort de ce modèle est la possibilité de décrire les contraintes à vérifier régulièrement par le système d'administration avant et après la mise en place d'une politique d'adaptation.

Néanmoins, Rainbow présente quelques inconvénients. En effet, le canevas se focalise sur l'automatisation de l'adaptation et n'exploite pas, par exemple, l'ADL pour le déploiement des systèmes distribués. De plus, la partie instrumentation et la création des actionneurs se font d'une manière ad hoc. Le modèle de composants n'est exploité que pour la représentation du système administré et non pas pour la construction du système d'administration lui-même.

2.4.2 SmartFrog

SmartFrog (*Smart Framework for Object Groups*) [GGL+] est un canevas conçu pour la description, le déploiement, le démarrage et la gestion d'applications Java distribuées. SmartFrog a été développé par HP Labs et a été exploité en production.

Le déploiement dans SmartFrog s'effectue grâce à un descripteur de déploiement basé sur un langage propriétaire. Les composants à déployer doivent respecter un modèle bien précis en étendant une classe Java, ce qui leur confère un cycle de vie que nous présentons dans la suite. Des applications patrimoniales peuvent être intégrées dans SmartFrog en les encapsulant dans des composants suivant le modèle SmartFrog.

Langage de description

Le langage de description de SmartFrog permet de définir une description du système à déployer d'une manière déclarative. Une *description d'un composant* est une collection ordonnée d'attributs. Un attribut a la forme d'une paire nom-valeur. Il existe trois types de valeurs :

- des valeurs simples (entiers, chaînes de caractères, etc.),
- une référence vers un autre attribut,
- une autre description de composant.

Le dernier cas permet de définir une relation d'inclusion entre composants : un composant A *contient* un composant B lorsque la description de B est incluse dans la description de A. Le modèle définit également la notion de *template* qui permet par extension de définir des descriptions de composants. Cette extension consiste à modifier des valeurs d'attributs ou à ajouter de nouveaux attributs.

```
//webservertemplate.sf
webServerTemplate extends {
    sfProcessHost "localhost";
    port 80;
    useDB;
}
```

```
//dbtemplate.sf
dbTemplate extends {
    userTable extends {
        columns 4;
        rows 3;
    }
    dataTable extends {
        columns 2;
        rows 5;
    }
}
```

```
// List of templates
# include "webservertemplate.sf";
# include "dbtemplate.sf";

system extends {
    commonPort "8080";
    ws1 extends webServerTemplate {
        sfProcessHost "15.144.59.34";
        port ATTRIB commonPort;
        useDB LAZY ATTRIB db;
    }
    ws2 extends webServerTemplate {
        sfProcessHost "15.144.59.64";
        port ATTRIB PARENT:commonPort;
        type "backup";
    }
    db extends dbTemplate {
        userTable:rows 6;
    }
}
```

FIG. 2.7 – *Exemple de description d'un composant SmartFrog.*

La figure 2.7 donne un exemple de description d'un composant SmartFrog. Les composants présentés à gauche de la figure sont des *templates*. La description du composant à droite de la figure étend ces *templates*. Les composants *ws1*, *ws2* et *db* qui sont définis dans *system* illustrent cette extension: dans *ws2*, l'attribut *type* a été ajouté et des valeurs d'attributs ont été modifiés dans *ws1* et *ws2*, par exemple *sfProcessHost*.

La relation d'inclusion définit un système de désignation de noms hiérarchique. Chaque composant définit un contexte de désignation de noms permettant la définition de noms composites. Par exemple, *port* est un nom simple dans le contexte de *webServerTemplate* et *userTable:rows* est un nom composite dans le contexte de *dbTemplate*.

Enfin, nous décrivons les *références* qui représentent une partie importante du langage. Par exemple, dans le composant système, les attributs « *port ATTRIB commonPort* »et « *port ATTRIB PARENT:commonPort* » réfèrent au même attribut portant le nom « *commonPort "8080"* » dans le composant englobant. Le mot clé **LAZY** dans l'attribut *useDB* veut dire que

la référence doit être résolue au moment de l'exécution.

Il est important de noter que le langage permet de définir des informations liées à la configuration (en affectant des valeurs aux attributs) et au déploiement (en choisissant les machines cibles).

Modèle à composants et cycle de vie

Un composant SmartFrog fournit deux types d'interfaces : des interfaces fonctionnelles et des interfaces de gestion. La figure 2.8 représente trois types d'opérations : *Deploy*, *Start* et *Terminate* ainsi que le cycle de vie du composant.

Par ailleurs, il y a deux types de composants, primitif et composite. Le composant primitif est un objet Java implantant des interfaces de gestion et des interfaces fonctionnelles comme nous le représentons à gauche de la figure 2.8. La relation d'inclusion présentée précédemment

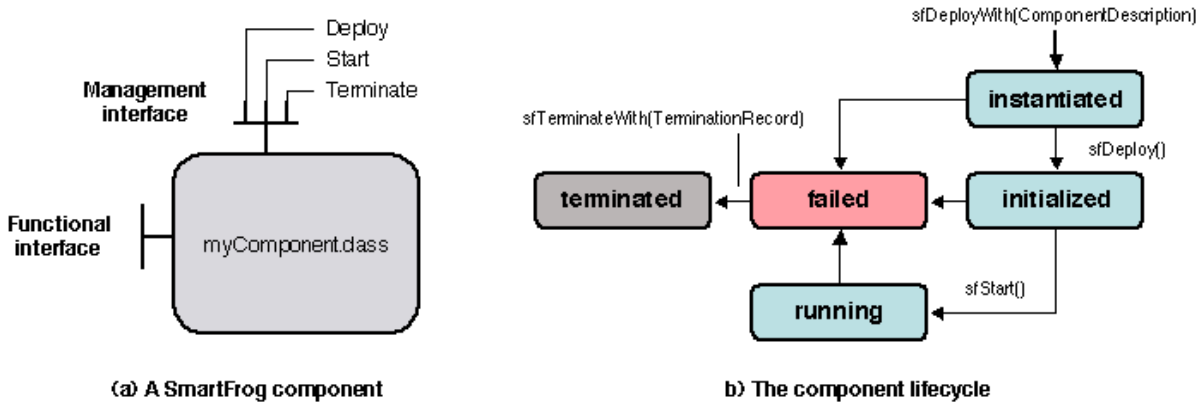

FIG. 2.8 – *Le modèle de composants SmartFrog.*

permet de définir des composants composites. Les composants fils partagent le même cycle de vie que les composants pères.

SmartFrog inclut des services pour faciliter le déploiement et l'administration, par exemple, un service de désignation de noms, de découverte dynamique, etc.

Déploiement

SmartFrog dispose d'un gestionnaire qui se base sur les descriptions de composants pour orchestrer les différentes tâches liées au déploiement. Ces tâches utilisent les interfaces de gestion fournies par les composants. Le gestionnaire est un système distribué et les différentes parties de ce système se coordonnent entre elles pour accomplir les tâches de déploiement.

Par ailleurs, le gestionnaire reste actif pendant l'exécution de l'application et se charge d'un certain nombre d'opérations telles que l'observation du système, la réaction aux fautes et l'arrêt des applications. De plus, pour répondre aux problèmes de sécurité, SmartForg offre un canevas de sécurité qui organise les différentes machines cibles sous la forme de domaines de sécurité, ce qui permet une répartition des responsabilités. De plus, la communication entre les nœuds est sécurisée.

SmartFrog a été utilisé dans un environnement industriel durant plusieurs années. En particulier, SmartFrog a été exploité pour l'administration de serveurs J2EE. Le langage de configuration permet une grande flexibilité et l'outil de déploiement a montré un passage à l'échelle raisonnable.

Synthèse

SmartFrog est un système servant à déployer et démarrer des systèmes distribués. Le système est modélisé sous la forme d'un ensemble de composants interconnectés. Le modèle à composants propose un cycle de vie et un ensemble d'interfaces fonctionnelles et de contrôle. Le modèle définit également la relation d'encapsulation entre composants. Pour le déploiement, SmartFrog se base sur un gestionnaire définissant les différentes tâches de déploiement et un ensemble de services comme le service de découverte et d'adressage.

SmartFrog présente par contre quelques limitations. Il suppose l'existence d'un environnement d'exécution déjà déployé sur les machines cibles et ne dispose pas d'un dépôt structuré des composants à déployer. D'autre part, les fonctions de reconfiguration sont très limitées et sont restreints au cycle de vie d'un composant. De plus, il n'est pas précisé comment effectuer des opérations de reconfiguration de structure ou d'implantation pour réparer une panne dans le système d'administration de SmartFrog. Enfin, le système d'administration est faiblement reconfigurable et ne profite pas, par exemple, du modèle à composants pour la gestion de son cycle de vie.

2.4.3 Nix

Nix [DJa04] est un canevas offrant plusieurs mécanismes de déploiement qui peuvent être utilisés afin de mettre en place des politiques de déploiement différentes. Ce canevas a été développé à l'université d'Utrecht. La principale contribution de Nix est un modèle pour un déploiement sûr et flexible, basé sur une analyse des dépendances entre les composants à déployer. La sûreté veut dire que toutes les dépendances sont satisfaites et la flexibilité veut dire la possibilité d'introduire un ensemble de politiques sans grande contrainte ainsi que la capacité du système à tolérer la coexistence de plusieurs versions d'un même composant.

Au départ, il s'agit surtout de remédier aux problèmes de cohérence et d'incompatibilité de versions rencontrés par des outils d'installation centralisée de logiciel, tels que RPM. Ceci a conduit à une analyse rigoureuse du processus d'installation, et à des propositions novatrices pour sa mise en œuvre.

L'approche proposée se base sur une analogie entre le déploiement de composants et la gestion de la mémoire [DVdJb04]. Deux problèmes majeurs sont identifiés.

- **Les dépendances non résolues**. A dépend de B (noté A->B) si le bon fonctionnement de A nécessite la présence de B. La *fermeture* d'un composant [2] de A (*closure*) est la fermeture transitive de la relation ->, issue de A, c'est à dire l'ensemble de composants dont A dépend (transitivement). Une dépendance n'est pas résolue lorsqu'un composant est déployé sans sa fermeture complète.
- **Les conflits**. Ceci fait référence au cas où deux versions différentes d'un même composant ne peuvent pas cohabiter parce qu'ils occupent la même place dans le système de fichiers.

Comme le présente la figure 2.9 la solution proposée est basée sur l'organisation d'un référentiel des composants sous la forme d'un espace d'adressage. Chaque objet répertorié possède un nom global unique appelé *store path*, similaire à un nom symbolique dans un système de fichiers hiérarchique.

Un composant est typiquement représenté par un répertoire sous lequel sont stockés tous ses éléments comme le code source, le code binaire, les librairies, etc. Un *store path* identifie

2. Nix utilise le terme « composant », mais dans un sens large (sans préjuger d'un modèle). Un composant est une unité de composition de logiciel et une unité de déploiement (une application, une partie bien identifiée d'application, une bibliothèque).

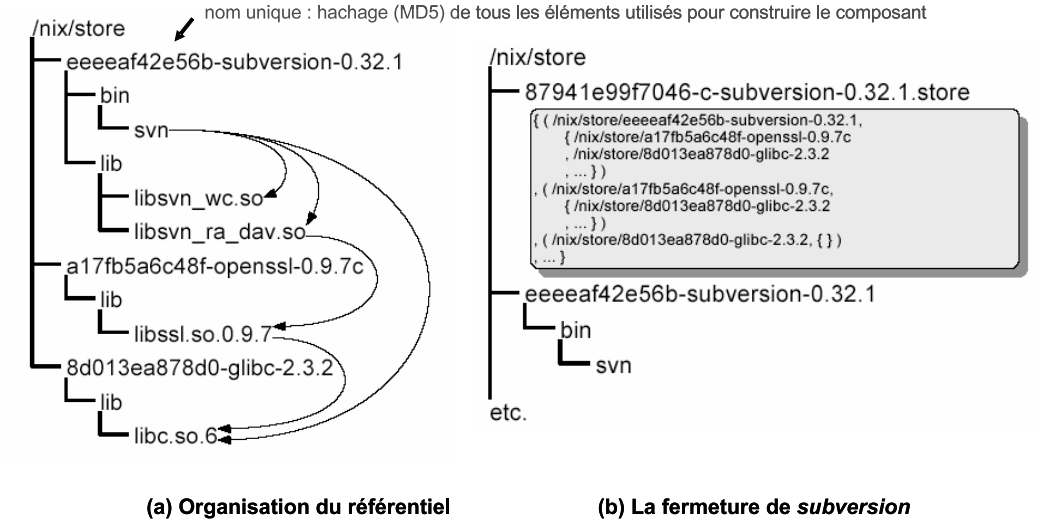

FIG. 2.9 – *Structure du dépôt des composants dans Nix.*

un composant d'une manière unique et inclut également un préfixe qui représente un hachage cryptographique de tous les éléments constituant le composant. Ainsi, une modification dans l'un des composants entraine la création d'un nouveau *store path*.

Afin de définir des dépendances, et donc de permettre aux fermetures d'être calculées, la règle suivante est appliquée: si un objet A dépend d'un objet B alors A contient le *store path* de B. Par exemple, les flèches de la figure 2.9(a) montrent les dépendances du composant `subversion`. La valeur de la fermeture correspondante est montrée dans la figure 2.9(b) comme étant un objet à part.

Les fermetures sont déterminées automatiquement en interprétant les *valeurs de dérivation*. Par exemple, la valeur de dérivation utilisée pour définir la fermeture de subversion est représentées dans le rectangle en haut de la figure 2.10.

Notons qu'une valeur de dérivation inclut un descripteur de déploiement (contenant par exemple, des noms de composants, une description de la plate-forme cible) et le programme des tâches de déploiement (sous la forme de scripts shell appellés *builders* dans Nix). Une valeur de dérivation est interprétée par l'équivalent d'un programme qui construit les fermetures et exécute les actions spécifiées par les *builders*.

La forme des valeurs de dérivation telle que présentée dans les figures précedentes est en fait cachée aux clients en utilisant divers mécanismes. En effet, les valeurs de dérivation ne sont pas conçues pour être écrites à la main. Elles sont générées à partir d'un langage de plus haut niveau.

Par ailleurs, le déploiement d'un composant suit un modèle fonctionnel (au sens des langages fonctionnels).

- Le résultat d'une opération ne dépend que de ses paramètres (si on recommence avec les mêmes paramètres, on a le même résultat). Par conséquent tous les paramètres sont explicites et il n'existe pas de dépendances cachées.
- Une fois le résultat obtenu, il ne change pas.

Les opérations de déploiement sont exprimées sous forme fonctionnelle comme des « expressions Nix » (*derivation expressions*) dans un langage développé à cet effet. Une description complète du langage (syntaxe et sémantique) se trouve dans [Dol05].

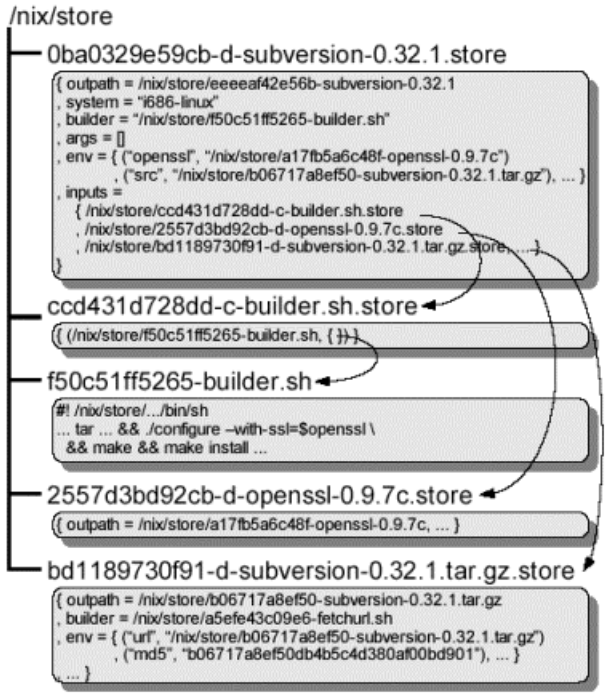

FIG. 2.10 – *Exemple de valeur de dérivation dans Nix.*

Synthèse

Nix offre une analyse rigoureuse du processus d'installation ainsi que des propositions novatrices pour sa mise en œuvre. La proposition la plus importante consiste en l'organisation du référentiel (noms internes uniques, gestion de versions, calcul de fermeture) pour le déploiement et la mise à jour. Ceci nous fait classer Nix parmi les systèmes dont l'administration est fondée sur l'architecture. En effet, toutes les dépendances entre les composants sont explicites et sont exposées à l'administrateur pour la gestion des versions.

Cependant, ce travail a été fait dans un contexte précis et pour un système d'exploitation précis. En effet, Nix est développé sous Linux et il n'est pas toujours possible de « nixifier » tous les systèmes d'exploitation. La mise en œuvre de Nix (par un administrateur) semble encore assez complexe (puisqu'il faut maîtriser le langage d'expressions). De plus, des progrès au niveau du langage sont nécessaires : le langage d'expression de Nix n'est pas typé et se réduit à l'utilisation de scripts *shell*. Par ailleurs, Nix n'est pas conçu pour un déploiement dans un environnement distribué comme les grappes et les grilles ; néanmoins, l'architecture de Nix pourrait être étendue pour prendre en compte ce genre d'environnements.

2.5 Synthèse

Dans ce chapitre, nous avons présenté les différentes fonctions de l'administration avec un intérêt particulier pour l'administration des intergiciels. Ceci est motivé par le fait que plusieurs standards ont été bâtis autour de l'administration des réseaux et des systèmes d'exploitation, mais que l'administration des intergiciels n'a suscité l'intérêt des industriels et des chercheurs que récemment. Cet intérêt est dû à la complexité croissante des applications qui nécessitent des fonctions d'administration sophistiquées. Dans ce chapitre, nous avons défini différentes fonctions

de l'administration (la configuration/déploiement, l'observation et la reconfiguration) et montré les défis que leur mise en œuvre soulève.

Nous avons, par ailleurs, expliqué que pour faire face à la complexité des systèmes actuels, les systèmes d'administration se basent de plus en plus sur la description de l'architecture des systèmes administrés. C'est le principe de l'*administration fondée sur l'architecture*. Une application de ce principe réside dans l'*administration autonome*.

Dans la deuxième partie de ce chapitre, nous avons présenté trois exemples de systèmes d'administration fondés sur l'architecture : Rainbow, SmartFrog et Nix. Chaque système se focalise sur un aspect particulier de l'administration : Rainbow offre un canevas pour la construction de systèmes auto-adaptables à l'exécution grâce à une infrastructure respectant une boucle de commande, SmartFrog se focalise principalement sur les aspects déploiement et démarrage des systèmes distribués et Nix élabore la phase d'installation dans le déploiement grâce à une structure originale du référentiel.

Nous avons constaté que les modèles de composants utilisés par chaque système étaient ad hoc et limités aux fonctions d'administration implantées. Dans le cadre de cette thèse, nous intéressons à un modèle plus général permettant de couvrir les fonctions d'administration de base dans un même système. Le chapitre suivant dresse un état de l'art des modèles de composants les plus utilisés dans les milieux industriels et académiques.

Chapitre 3

Modèles de composants pour la construction de systèmes administrables

Comme nous l'avons mentionné dans l'introduction, nous désignons par *administrables* les sytèmes qui offrent la possibilité de configurer et de déployer leurs différentes parties dynamiquement. Depuis les années 90, la construction de systèmes administrables repose sur l'utilisation de *composants*. Dans ce chapitre, nous dressons un état de l'art de différents modèles de composants existants. Divers modèles ont été proposés ; nous les classons en deux catégories : ceux qui se focalisent sur le déploiement [Ope03, MFH01, ACN02, ASCN03] — notamment pour l'installation et la mise à jour — et ceux qui permettent des reconfigurations dynamiques de structure [PBJ97, PBJ98, MDEK95, CBCP01, BCS].

Dans ce chapitre, nous avons fait le choix de présenter les modèles EJB, Fractal et OSGi. En effet, la description des EJB est nécessaire pour comprendre le cadre applicatif (serveurs J2EE) de cette thèse. Par ailleurs, les modèles Fractal et OSGi sont utilisés dans la contribution de notre thèse.

Ce chapitre est structuré de la façon suivante : nous le débutons par une définition des besoins en termes d'architecture pour la construction de systèmes administrables. Nous présentons ensuite divers modèles à composants industriels et académiques.

3.1 Quelle architecture adopter pour la construction de systèmes administrables ?

Pour construire des systèmes administrables, les administrateurs se basent, aujourd'hui, sur les éléments d'architecture afin de mettre en œuvre les fonctions d'administration. C'est le principe de l'administration fondée sur l'architecture dont nous identifions les deux propriétés suivantes.

- L'architecture du système doit être explicite en termes des ressources administrées et des relations entre elles. Nous distinguons deux types de relations : les connexions et les relations d'encapsulation. De plus, cette architecture doit être dynamiquement modifiable.
- Chaque ressource administrée doit être une unité isolée de configuration et de déploiement.

Ces principes peuvent être appliqués à des granularités différentes, selon le niveau d'adaptation souhaité.

Pour construire de telles architectures, il est logique d'adopter les composants qui permettent d'isoler les différentes parties d'un système et d'exposer leurs dépendances. En effet, « un composant est une unité de composition qui peut être déployée et configurée d'une façon indépendente » [Szy02]. Par conséquent, un composant est une brique de base configurable permettant de construire des logiciels par composition. Un composant a des interfaces permettant de modéliser des services *fournis* et des services *requis*.

Par ailleurs, pour construire des systèmes administrables, le modèle de composants doit permettre la mise en place des fonctions d'administration détaillées dans le chapitre précédent :

- **Configuration et déploiement.** Le modèle de composants doit offrir les interfaces nécessaires pour la configuration des objets encapsulés allant de quelques attributs jusqu'à l'expression des relations avec d'autres composants. Pour définir des architectures, nous avons besoin des deux types de relations : les liaisons et l'encapsulation. De plus, le modèle de composants doit offrir les outils pour exprimer cette architecture et automatiser son déploiement. Pour atteindre cet objectif, plusieurs modèles de composants définissent un langage de description d'architecture ou ADL (*Architecture Description Language*).
- **Observation.** Le modèle de composants doit offrir les interfaces nécessaires à l'introspection des ressources encapsulées/représentées par les composants.
- **Reconfiguration.** L'assemblage des composants doit être dynamiquement modifiable. Ceci permet d'assurer la reconfiguration de structure. Par ailleurs, le modèle doit tolérer également la reconfiguration d'implantation en offrant les propriétés nécessaires en termes de paquetage et de gestion de versions.

Dans ce chapitre, nous décrivons un certain nombre de modèles de composants ayant des utilisations différentes. Tout d'abord, nous présentons les modèles standards EJB [Ent02] et CCM [Gro02] qui permettent la création d'applications d'entreprises. Ces modèles supposent l'existence d'un intergiciel offrant des propriétés non-fonctionnelles comme la persistance, les transactions, la sécurité, etc. Ensuite, nous décrivons le modèle OSGi qui permet de gérer les dépendances entre classes Java patrimoniales via une notion de paquetage étendue. Enfin, nous présentons les modèles DCUP et Fractal qui ciblent la construction de systèmes reconfigurables à l'exécution grâce à un ensemble de contrôleurs.

3.2 Les EJB

Le modèle de composants EJB (*Enterprise Java Beans*) est spécifié par Sun Microsystems [Ent02]. Les EJB (appelés aussi *beans*) sont des objets Java dédiés à la construction d'applications J2EE. Ils implantent les interfaces nécessaires pour interagir avec leur environnement d'exécution, le conteneur des EJB. Le code des EJB représente la partie métier d'une application J2EE, alors que le conteneur offre des propriétés dites non-fonctionnelles, telles que la communication, les transactions et les aspects sécurité. Dans le chapitre 4, nous décrivons les propriétés des conteneurs EJB et plus généralement l'intergiciel J2EE. Dans cette section, nous nous concentrons sur le modèle EJB et la description de ses interfaces.

La figure 3.1 représente les différentes interfaces et les descripteurs nécessaires à l'implantation d'un EJB. Un composant EJB se présente sous la forme d'un ensemble de classes Java regroupées dans un fichier de description. Chaque composant forme ainsi une unité réutilisable qui peut être

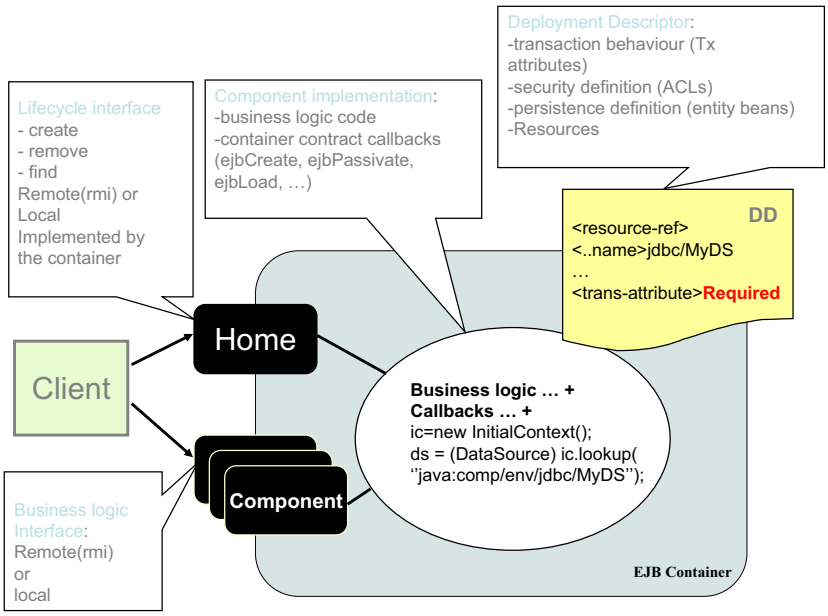

FIG. 3.1 – *Architecture et implantation des EJB.*

combinée à d'autres pour former une application. La spécification EJB 2.x définit trois principaux types d'EJB : les beans sessions, les beans entités et les beans pilotés par messages.

Chacun de ces composants est un programme de type serveur, destiné à traiter les requêtes d'autres programmes, comme des servlets ou des JSP ou toute autre application Java[1].

Les beans sessions Un *bean session* est un composant dont chaque instance est associée à un unique client. La durée de vie d'une instance est donc limitée au temps que dure la requête cliente. On distingue deux sortes de beans sessions :

- le *bean session avec état* conservant des données entre les appels de méthodes du client,
- le *bean session sans état* ne conservant pas d'état au-delà du traitement d'un appel de méthode.

Les beans entités Un *bean entité* est une représentation dans le serveur d'applications d'une donnée stockée dans une base de données. Il peut être utilisé et modifié simultanément par plusieurs clients. Sa durée de vie correspond à celle de la donnée dans la base. Nous distinguons deux sortes de beans entités :

- le bean dont la persistance est programmée par le développeur (*bean-managed persistence* ou BMP),
- le bean dont la persistance est gérée automatiquement par le conteneur EJB (*container-managed persistence* ou CMP).

Les beans pilotés par message Un *bean piloté par message* est un programme qui se comporte comme un consommateur de messages *Java Message Service* (JMS). C'est donc un composant qui n'est pas piloté directement par des programmes clients. Il effectue des traitements

1. Nous présentons les servlets et les JSP dans 4.1.

à la réception d'un message en s'abonnant à une destination. Ces messages peuvent avoir été envoyés par n'importe quel client JMS.

Interfaces des beans sessions et des beans entités

Dans la figure 3.1 sont représentées les différentes interfaces d'un EJB. Du point de vue du client, les beans entités et les beans sessions sont composés chacun de deux interfaces : la partie `EJBHome` et la partie `EJBObject`, ou `EJBLocalHome` et `EJBLocalObject`, selon que l'interface est distante ou locale. L'interface `Home` sert à gérer le cycle de vie des instances du bean alors que la partie métier réside dans l'objet implantant l'interface `Object`.

Les méthodes de l'interface `Home` d'un bean session permettent de créer/détruire des instances de beans, d'obtenir des informations sur le type de bean géré par cette interface et d'obtenir une clé sérialisable permettant de récupérer un talon vers l'interface Home. L'interface `EJB(Local)Object`, permet d'effectuer des appels de méthode sur l'instance de bean référencée. Elle permet en outre de détruire cette instance, de récupérer l'interface `Home`, d'obtenir une clé sérialisable permettant d'obtenir une nouvelle référence vers l'instance de bean et d'obtenir une clé primaire permettant de le localiser ultérieurement.

L'accès aux beans Contrairement aux beans pilotés par messages, les beans sessions et entités peuvent être directement appelés par des clients. L'accès peut être effectué à distance en utilisant un protocole de communication à travers le réseau, comme RMI. Depuis la spécification 2.0, il est également possible de mettre des interfaces locales sur des beans : elles ne permettent leur utilisation qu'à l'intérieur du même conteneur d'EJB, mais offrent de meilleures performances (grâce aux échanges de paramètres par référence plutôt que par valeur entre les clients et les EJB).

Les interfaces `Home` sont également un moyen d'obtenir des références vers des instances de bean. Pour ce faire, le conteneur d'EJB utilise un service de nommage JNDI (*Java Naming and Directory Interface*). Celui-ci permet à un client d'obtenir la référence qu'il cherche à partir d'un nom symbolique.

Administration des EJB Rappelons que les propriétés non-fonctionnelles comme la sécurité et les transactions des EJB sont gérées par le conteneur qui constitue l'environnement d'exécution et qui gère le cycle de vie des EJB. Comme nous le décrivons dans 4.3, l'administration des applications J2EE est standardisée. C'est principalement le déploiement qui est spécifié et peu de détails sont donnés sur les fonctions d'observation et encore moins sur la reconfiguration. Le déploiement, que nous décrivons en détails dans 4.3, consiste à regrouper un ensemble d'EJB avec leurs descripteurs dans une même unité de déploiement, appellée *module*. Pour cela, le conteneur doit implanter un certain nombre d'interfaces permettant, d'une part, de récupérer les informations nécessaires à la configuration des EJB (en utilisant les descripteurs) et, d'autre part, de permettre l'installation des modules.

Cependant, une fois déployés, les EJB n'offrent pas le moyen de connaître leur état à l'exécution ou de reconfigurer les liaisons entre eux. En effet, le modèle ne prévoit pas d'interfaces particulières d'introspection et n'expose pas les dépendances entre les EJB d'une manière explicite pour des fins d'observation et de reconfiguration. La liaison entre les modules et les ressources extérieures ne sont pas non plus exposées à l'administrateur pour adapter par exemple l'allocation et l'utilisation des ressources en fonction de l'évolution des EJB au sein d'une application J2EE.

Synthèse

Les EJB représentent une solution intéressante pour développer des applications Web de façon modulaire. Les propriétés non-fonctionnelles des EJB sont prises en charge par le conteneur qui gère leur cycle de vie. Cela facilite le développement des applications Web du fait que le développeur peut se concentrer sur l'aspect métier.

Le déploiement des EJB est couvert par les spécifications J2EE. Cependant, une fois déployée, une application J2EE est une boîte noire dont la reconfiguration et la gestion des dépendances avec les ressources extérieurs sont trés réduites. En effet, le modèle des EJB ne permet pas d'exposer l'architecture d'une application J2EE d'une manière explicite.

3.3 CCM

Le CCM (*Corba Component Model*) [Gro02] est un modèle de composants proposé par l'OMG. Nous commençons par présenter la structure des composants. Nous décrivons ensuite le rôle des conteneurs de composants. Enfin, nous présentons les langages de description de composants et les descripteurs de déploiement.

Structure d'un composant

Un composant interagit avec les autres composants en utilisant des interfaces fonctionnelles (ou *ports*) dont il existe quatre types différents :

- une **facette** est une interface serveur qui peut être utilisée par des clients en mode synchrone,
- un **réceptacle** est une interface cliente en mode synchrone,
- un **puits** d'événements est une interface serveur qui peut être utilisée par des clients en mode asynchrone,
- une **source** d'événements est une interface cliente en mode asynchrone.

Par ailleurs, un composant possède des *attributs* qui permettent de le configurer lors de son déploiement ou en cours d'exécution.

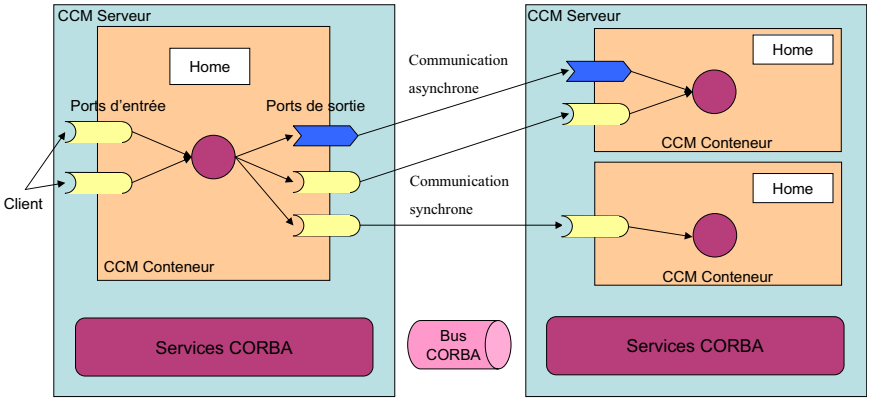

FIG. 3.2 – *Architecture des conteneurs CCM.*

Le conteneur de composants

La figure 3.2 représente l'architecture de conteneurs CCM. Les composants CCM s'exécutent au sein de conteneurs qui fournissent certains services système (transactions, sécurité, tolérance aux fautes, etc.) et qui utilisent un bus CORBA pour l'acheminement des communications entre les ports. Les conteneurs prennent en charge un seul type de composants. Ils contiennent une *usine de composants* qui se charge de la création et de la destruction d'instances de ce type de composant. Ils contiennent également des objets d'interposition pour les ports des composants. Les conteneurs permettent de gérer ces objets d'interposition à l'aide de deux interfaces : une interface d'introspection qui fournit des informations sur le type des composants, sur ses ports, ainsi que sur l'état de ses connexions aux autres composants ; une interface de gestion qui permet de créer des connexions entre les composants.

Description de composants

Le type et l'implantation des composants CCM sont définis à l'aide de langages déclaratifs. Les types de composants sont décrits en utilisant une extension de l'IDL (*Interface Definition Language*) de CORBA [cor02], ce qui permet à la fois de programmer les composants dans différents langages et de les exécuter dans des environnements hétérogènes. Les implantations de composants sont décrites à l'aide du langage de description CIDL (*Component Implementation Definition Language*). CIDL permet de décrire la structure logicielle de l'implantation des composants, ainsi que certains de leurs aspects non-fonctionnels : persistance, transaction et sécurité. La compilation des descriptions IDL et CIDL fournit, entre autres, un squelette prenant en charge la partie non-fonctionnelle d'un composant et un descripteur XML regroupant les contraintes techniques à respecter lors de son déploiement.

Déploiement de composants

Concernant le déploiement des composants, CCM utilise des *descripteurs de déploiement*. On distingue deux types de descripteurs.

- Les **descripteurs de composants** permettent de décrire des composants "isolés". La description d'un composant contient sa structure logicielle, son type en termes de ports d'entrée et de sortie, ainsi que la catégorie à laquelle il appartient (processus, entité, session, service). Ces descripteurs sont générés lors de la compilation de la description des composants (faite à l'aide de CIDL), mais peuvent être modifiés pour paramétrer la gestion des aspects non-fonctionnels.
- Les **descripteurs d'assemblage de composants** permettent de décrire l'architecture initiale d'une application. Ils spécifient les instances de composants constituant l'application, définissent des règles de placement sur des sites d'exécution, et indiquent les connexions à établir entre les différentes instances.

Synthèse

Le modèle CCM est destiné à la construcion d'applications ayant besoin de divers services pour s'exécuter. Des exemples typiques de telles applications sont les applications Web. L'avantage de cette approche est qu'elle simplifie le développement d'applications. La complexité de la prise en charge des services non fonctionnels est masquée par des interfaces simplifiées et elle est prise en charge, d'une part, par les conteneurs et, d'autre part, par les parties du composant qui sont générées par les compilateurs. Un des apports du modèle CCM est de proposer des

outils nécessaires aux différentes étapes du cycle de construction d'une application (IDL, CIDL, descripteurs de déploiement).

Néanmoins, le modèle CCM, comme les EJB a un certain nombre de limites ; certaines sont citées dans [MM00]. Tout d'abord, le modèle CCM a un modèle de composition limité : il n'est possible de créer ni des composants composites, ni des composants partagés. D'autre part, les capacités d'administration sont limitées : les seules possibilités de reconfiguration sont fournies par les interfaces d'introspection et de gestion de ports implantées par les conteneurs. Une autre limitation du modèle CCM est son faible niveau de configurabilité : les conteneurs sont monolithiques et ne supportent qu'un ensemble figé de propriétés non-fonctionnelles. Enfin, les outils fournis permettent uniquement de générer et déployer du code.

3.4 OSGi

OSGi (*Open Services Gateway initiative*) [Ope03] est un standard définissant initialement une plate-forme orientée services. OSGi fournit l'environnement nécessaire pour installer/desinstaller et mettre à jour des unités de déploiement, appelées*bundles*. Les *bundles* communiquant via des *services*. Pour effectuer la fonction de déploiement, OSGi définit trois propriétés :

- un format pour les paquetage des *bundles*,
- un mécanisme de chargement de classes pour isoler les *bundles*,
- un dépôt permettant aux composants d'enregistrer leurs services et d'utiliser des services d'autres composants.

Structure d'un *bundle*

Un *bundle* est un fichier JAR contenant du code binaire ainsi que les ressources nécessaires au déploiement d'une application. Le *bundle* contient, également, un fichier *manifest* et une classe d'activation. Le *manifest* décrit, entre autres, les dépendances du code du composant sous la forme d'*import/export* de paquetages Java.

La figure 3.3 représente la structure d'un *bundle*. Nous distinguons les parties suivantes :

- les ressources qui nécessitent des services extérieurs,
- le fichier *manifest* décrivant les dépendances du *bundle*,
- les interfaces *Services* fournies et requises,
- les interfaces pour importer et exporter les paquetages,
- les objets métiers qui représentent le code à déployer,
- un activateur dont le rôle est d'instancier le code.

Déploiement et mise à jour des *bundles*

Chaque bundle possède son propre *class loader*. Ce dernier permet de charger toutes les ressources nécessaires pour les classes Java contenues dans le *bundle*. La mise à jour d'un *bundle* consiste à désinstaller l'ancienne version et à installer la nouvelle. Par conséquent, la mise à jour d'un *bundle* nécessite un nouveau *class loader*. Si ce *bundle* exporte des ressources, alors tous les autres qui les importent doivent être redémarrés également. En effet, si les ressources importées sont chargées par un nouveau *class loader*, une exception `ClassCastException` est levée par les *bundles* qui les importent. Pour cela, la plate-forme OSGi enregistre toutes les dépendances *import/export* des *bundles* déployés et en tient compte au moment de la mise à jour.

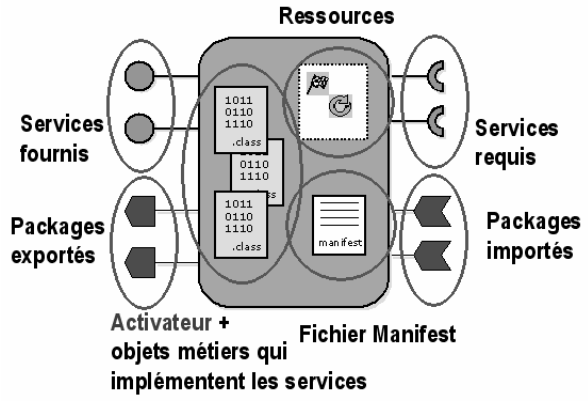

FIG. 3.3 – *Le bundle OSGi.*

Les services OSGi

Outre les propriétés d'empaquetage et de chargement de classes, OSGi définit des interfaces de *services* permettant l'interaction entre les *bundles* déployés. Un service est publié en précisant une interface de service, une référence vers le composant qui implante le service et un ensemble de propriétés. Ces propriétés, de type clé-valeur, permettent aux clients de distinguer deux offres de service ayant la même interface. De plus, le référentiel fournit des moyens pour effectuer des recherches contraintes par des filtres respectant une syntaxe LDAP. L'enregistrement et le retrait de services peuvent être signalés aux composants par l'envoi de notifications. Par ailleurs, l'assemblage des parties d'une application a lieu lors de l'exécution comme résultat de l'interaction entre les composants et les services.

Synthèse

OSGi est un standard définissant une plate-forme conçue pour le déploiement des applications Java. Pour cela, le standard définit trois parties : le paquetage des applications, le chargement de classes et les services. L'unité de déploiement est le *bundle* et chaque *bundle* est chargé par un *class loader* séparé. Des *bundles* peuvent échanger des ressources grâce à des relations *import/export* définies dans un fichier *manifest*. De plus, les *bundles* peuvent échanger des services à l'exécution.

OSGi présente un ensemble de limitations. Tout d'abord, le standard ne propose pas pour l'instant d'interfaces de programmation pour gérer la liaison dynamique aux services qui reste bien difficile à maîtriser pour le développeur d'applications. De plus, la plate-forme ne prend pas en charge des aspects non fonctionnels comme le cycle de vie, la configuration, la journalisation, la persistance, la sécurité, la distribution et la gestion des sessions. Ces aspects font partie de la responsabilité du développeur. De plus, le standard ne propose pas d'ADL pour décrire l'architecture du système à déployer et ne permet pas d'observer le système à l'exécution. En effet, les outils de déploiement et d'observation sont à la charge du développeur et se font d'une manière ad hoc.

Concernant l'architecture des applications deployées par OSGi, l'utilisation des contextes entre les *bundles* rend les dépendances cachées et ne permet pas l'exposition de l'architecture des applications d'une manière explicite. Il y a eu plusieurs travaux pour résoudre ce problème (par exemple, le projet ServiceBinder [HC04]) mais le modèle lui-même ne permet pas une gestion explicite des dépendances. Par ailleurs, la communication entre *bundles* ne peuvent être établies qu'au sein d'une même JVM. Les architectures distribuées ne sont pas prises en compte

par le modèle. De plus, il n'est pas possible de représenter des relations d'encapsulation entre composants, ce qui rend le modèle faible pour la représentation des architectures de systèmes.

3.5 DCUP

DCUP (*Dynamic Component UPdating*) [PBJ97, PBJ98] fait partie du projet SOFA/DCUP issu de l'université de Prague. DCUP est un modèle de composants qui a été conçu pour permettre la mise à jour dynamique des composants. Mettre à jour un composant consiste à le remplacer par une version plus récente.

Structure d'un composant DCUP

Une application DCUP est constituée par une hiérarchie en arbre de composants imbriqués. Chaque composant est composé d'un ensemble d'objets (Java dans la version actuelle de la plate-forme). Les services fournis et requis par les composants sont représentés par des objets d'indirection appellés adaptateurs (*wrappers*). Ces indirections permettent de modifier facilement les références correspondant aux services requis ou fournis par le composant. Toutes les communications entre ces composants passent par ces objets d'indirection. Un composant est séparé en une partie permanente qui existe pour toute la durée de vie du composant et une partie remplaçable qui peut être modifiée à l'exécution par une nouvelle version lors d'une mise à jour.

La figure 3.4 montre ces deux parties : la partie permanente est représentée par un *ComponentManager* (CM) et la partie remplaçable est composée d'un *componentBuilder* (CB) et d'un ensemble de composants fonctionnels. Le rôle du CM est de coordonner le processus de mise à jour et de maintenir les références vers les divers services fournis par le composant. Le CB contient la logique permettant de créer les sous-composants. Il est associé à une version particulière du composant. Ses rôles sont (1) de construire et de détruire des composants de la version à laquelle il est associé et (2) de rendre accessible et de permettre la restauration de l'état du composant.

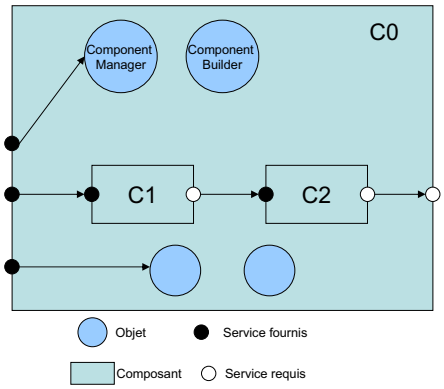

FIG. 3.4 – *Structure d'un composant DCUP.*

Mise à jour d'un composant

La mise à jour d'un composant est réalisée par la coopération du fournisseur du composant, du CM et des CB des deux versions du composant concerné par la mise à jour. Le CM héberge

une entité responsable des mises à jour. Celle-ci interagit avec le fournisseur du composant pour obtenir les nouvelles versions du composant. Cette interaction peut se faire à l'initiative du CM (mode *pull*) ou à l'initiative du fournisseur du composant (mode *push*).

La mise à jour se fait suivant les étapes suivantes.

- Le CM bloque toutes les communications entrantes en verrouillant les adaptateurs correspondants. Le CM envoie alors la requête de mise à jour au CB.
- Le CB arrête l'exécution des composants fonctionnels (de la partie remplaçable), capture l'état du composant et détruit les objets encapsulés.
- Une fois le composant détruit, le CM instancie le CB de la nouvelle version du composant et lui ordonne la création de la nouvelle instance du composant à mettre à jour.
- Le CB crée la nouvelle structure en instanciant les objets internes et les sous-composants. Le CB injecte alors l'état du composant et peut effectuer des transformations dans cet état pour l'adapter aux nouveaux composants.
- Quand l'ensemble des composants est prêt, le CM débloque tous les adaptateurs.

Synthèse

DCUP propose un modèle de composants qui permet la mise-à-jour dynamique des composants. Cependant, la mise à jour est limitée au contenu d'un composant et ne couvre pas les services fournis et requis. Par ailleurs, DCUP ne définit pas de fonction de surveillance pour les composants et ne permet pas d'automatiser certaines opérations de reconfiguration. De plus, malgré la propriété de hiérarchie dans DCUP, le modèle ne permet pas aux administrateurs d'obtenir une vision de l'architecture de l'application en cours d'exécution.

3.6 Fractal

Fractal [BCS] est un modèle de composants indépendant du langage d'implantation, hiérarchique et séparant clairement la partie fonctionnelle de la partie de contrôle. La spécification de Fractal a été conçue conjointement par France Telecom R&D et l'INRIA. Nous commençons par présenter le modèle. Nous présentons ensuite l'implantation de référence, Julia [jul02]. Nous décrivons également les outils pour le déploiement des composants Fractal. Enfin, nous présentons le langage de description d'architecture (ADL) de Fractal et la gestion de chargement de classes.

Composants et liaisons

Un composant Fractal est une entité d'exécution qui possède une ou plusieurs interfaces. Une interface est un point d'accès au composant. Une interface implante un type d'interface qui spécifie les opérations supportées par cette dernière. Il existe deux catégories d'interfaces : les interfaces *serveur* — qui sont des points d'accès acceptant des appels de méthodes — et les interfaces *client* qui sont des points d'accès émettant des appels de méthodes.

Comme représenté sur la figure 3.5, un composant Fractal est généralement composé de deux parties : *une membrane* — qui possède des interfaces fonctionnelles et de contrôle — et un *contenu* qui est constitué d'un ensemble de sous-composants.

Les interfaces d'une membrane sont soit *externes*, soit *internes*. Les interfaces externes sont accessibles de l'extérieur du composant, alors que les interfaces internes sont accessibles par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de *contrôleurs*. Les contrôleurs peuvent être considérés comme des méta-objets. Chaque contrôleur

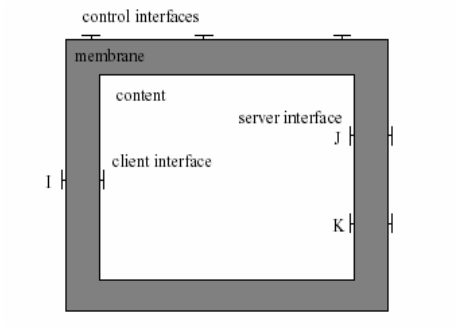

FIG. 3.5 – *Le modèle Fractal.*

a un rôle particulier. Par exemple, certains contrôleurs ont la charge de fournir une représentation causalement connectée de la structure d'un composant (en termes de sous-composants). D'autres contrôleurs permettent de contrôler le comportement d'un composant et/ou de ses sous-composants. Les contrôleurs peuvent également jouer le rôle d'intercepteurs. Les intercepteurs permettent d'intercepter les appels de méthodes entrant et sortant des interfaces d'un composant. Il est alors possible de contrôler ces appels par le moyen des contrôleurs.

Le modèle Fractal fournit deux mécanismes permettant de définir l'architecture d'une application: l'imbrication (à l'aide de composants *composites*) et la *liaison*. La liaison est ce qui permet aux composants Fractal de communiquer. Fractal définit deux types de liaisons, primitives et composites. Les liaisons *primitives* sont établies entre une interface client et une interface serveur de deux composants résidant dans le même espace d'adressage. Une liaison primitive est implantée à l'aide d'un pointeur (en C) ou d'une référence (en Java). Les liaisons *composites* sont des chemins de communication arbitrairement complexes entre deux interfaces de composants. Les liaisons composites sont constituées d'un ensemble de composants de liaison (par exemple stub, skeleton) reliés par des liaisons primitives.

Niveaux de contrôle

Le modèle de composants Fractal n'impose la présence d'aucun contrôleur dans la membrane d'un composant. Il permet, au contraire, de créer des formes arbitrairement complexes de membranes implantant diverses formes et sémantiques de contrôle et d'interception. Pour faciliter la tâche du développeur d'applications, la spécification Fractal définit différents contrôleurs :

- le **contrôleur d'attributs** permet de configurer les attributs d'un composant;
- le **contrôleur de liaisons** permet de créer/rompre une liaison primitive entre deux interfaces de composants;
- le **contrôleur de contenu** permet d'ajouter/retrancher des sous-composants au contenu d'un composant composite;
- le **contrôleur de cycle de vie** permet de contrôler les principales phases comportementales d'un composant. Par exemple, les méthodes de base fournies par un tel contrôleur permettent de démarrer et stopper l'exécution du composant.

Implantations

Il existe différentes implantations du modèle Fractal : Think [FSLM02], Plasma [LH05], ProActive [pro05], AOKell [aok05], FracTalk [fra05] en SmallTalk, etc. Julia est l'implantation

de référence du modèle. Julia est un canevas logiciel écrit en Java qui permet de programmer les membranes des composants et de les déployer programmatiquement ou à l'aide d'un langage de description d'architecture (ADL). Julia fournit un ensemble de contrôleurs que l'utilisateur peut assembler pour créer les intercepteurs et contrôleurs de son choix. Par ailleurs, Julia fournit des mécanismes d'optimisation qui permettent d'obtenir un continuum allant de configurations entièrement statiques et très performantes à des configurations moins performantes permettant des reconfigurations dynamiques. Le développeur d'application peut ainsi choisir l'équilibre performance/dynamicité dont il a besoin.

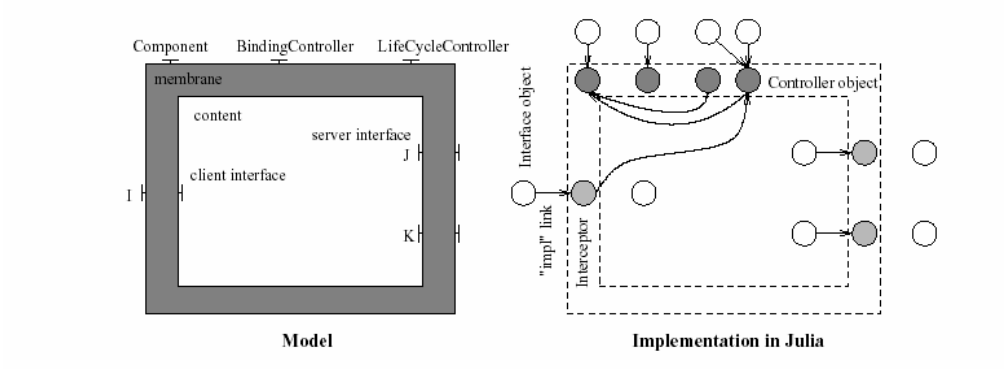

FIG. 3.6 – *Julia : implantation de référence de Fractal.*

Comme le représente la figure 3.6, un composant Fractal est formé de plusieurs objets Java que l'on peut séparer en trois groupes.

- Les objets qui implémentent le contenu du composant. Ces objets n'ont pas été représentés sur la figure. Ils peuvent être des sous-composants (dans le cas de composants composites) ou des objets Java (pour les composants primitifs).
- Les objets qui implantent la partie de contrôle du composant (représentés en noir et en gris). Ces objets peuvent être séparés en deux groupes : les objets implémentant les interfaces de contrôle, et des intercepteurs optionnels qui interceptent les appels de méthodes entrant et sortant sur les interfaces fonctionnelles. Comme les aspects de contrôle ne sont généralement pas indépendants, les contrôleurs et les intercepteurs ont généralement des références les uns vers les autres.
- Les objets qui référencent les interfaces du composant (en blanc). Ces objets sont le seul moyen pour un composant de posséder des références vers un autre composant.

La mise en place de ces différents objets est effectuée par des fabriques de composants. Celles-ci fournissent une méthode de création qui prend en paramètre une description des parties fonctionnelles et de contrôle du composant.

Déploiement

Nous présentons deux aspects du déploiement : le langage de description d'architecture et le méchanisme de chargement de classes.

Fractal ADL

Fractal fournit un langage de description d'architectures [fraa] constitué de deux parties : un langage basé sur XML et une usine qui permet de traiter les définitions faites à l'aide du langage. Le langage ADL de Fractal est extensible : il est constitué d'un ensemble de modules

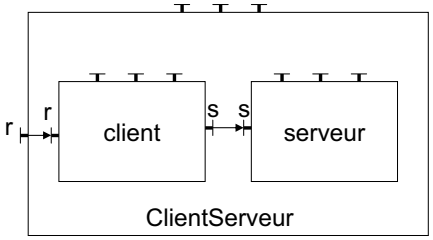

FIG. 3.7 – *Architecture du composant ClientServeur.*

permettant la description de divers aspects de l'application. La figure 3.7 présente la définition ADL du composant de la figure 3.8. Nous distinguons un composant composite `ClientServeur` et deux sous-composants `client` et `serveur`. L'interface client de `client`, `s`, est explicitement liée à l'interface serveur, `s` de `serveur`.

```
<definition name="ClientServeur">
<interface name="r" role="server" signature="java.lang.Runnable"/>
<component name="client">
   <interface name="r" role="server" signature="java.lang.Runnable"/>
   <interface name="s" role="client" signature="Service"/>
   <content class="ClientImpl"/>
</component>
<component name="serveur">
   <interface name="r" role="server" signature="java.lang.Runnable"/>
   <interface name="s" role="client" signature="Service"/>
   <content class="ClientImpl"/>
</component>
<binding client="this.r" server="client.r"/>
<binding client="client.s" server="serveur.s"/>
</definition>
```

FIG. 3.8 – *Description en ADL de l'exemple de composant ClientServeur.*

Chargement de classes dans Fractal ADL

Fractal ADL permet de spécifier le *class loader* à utiliser pour créer les composants. Ceci est fait à l'aide du code représenté dans la figure 3.9.

```
ClassLoader cl = new MySimpleClassLoader();
Map context = new HashMap();
context.put("classloader", cl);
Factory f = FactoryFactoty.getFactory();
Object c = f.newComponent("ClientServeur", context);
```

FIG. 3.9 – *Chargement de classes dans Fractal : exemple du composant ClientServeur.*

Dans cet exemple, toutes les classes utilisés par le composant `ClientServeur` sont chargées à l'aide d'un *class loader* `MySimpleClassLoader`. Notons qu'il serait possible de charger les composants `Client` et `Serveur` à l'aide d'autres *class loader*. Les composants liés doivent avoir les interfaces chargées par le même *class loader*, sinon une exception `ClassCastException` est levée. En revanche, ces deux composants peuvent être chargés par des *class loaders* différents, ce qui permet le changement de versions et le redéploiement.

Une limitation actuelle de ce méchanisme est qu'il n'offre pas de solutions pour le paquetage des composants. Dans le cadre de cette thèse, nous avons proposé une solution [AKS05], dont le

principe est d'utiliser Fractal pour le paquetage et pour le déploiement.

3.7 Synthèse

Pour construire des systèmes administrables, il est nécessaire d'adopter une architecture modulaire qui permet d'automatiser le déploiement et la configuration des différentes parties du système, ainsi que d'effectuer des opérations de reconfiguration dynamiques. Il est aujourd'hui communément admis que les modèles de composants forment une base intéressante pour la construction de systèmes administrables.

Dans ce chapitre, nous avons présenté un ensemble de modèles de composants. Les deux premiers modèles présentés permettent de simplifier le développement d'applications Web. Ils se basent sur les intergiciels pour la gestion du cycle de vie des composants et pour disposer de services dits non-fonctionnels. Ces modèles ne sont pas suffisamment généraux pour être utilisés dans la construction de systèmes administrables.

Les autres modèles que nous avons étudiés peuvent être classés en deux catégories : ceux qui se focalisent sur le déploiement — notamment pour l'installation et la mise à jour du code — et ceux qui visent la reconfiguration de structure à l'exécution. Dans la première catégorie, nous avons décrit OSGi, un modèle utilisant une notion de paquetage évolué pour procéder à un chargement de classes avec prise en compte de versions. Dans la seconde catégorie, nous avons présenté DCUP et Fractal. Fractal semble être le modèle le plus approprié pour construire des systèmes administrables. Il permet la description explicite des architectures et autorise des formes arbitraires d'opérations de contrôle grâce à un ensemble de contrôleurs extensibles. Une limitation actuelle de Fractal est qu'il n'offre pas d'outils pour l'empaquetage et pour la mise à jour de composants. Nous pensons que l'enrichissement du modèle Fractal avec une infrastructure de type OSGi permettrait de combler ces lacunes. C'est la solution que nous avons adopté dans cette thèse. Le chapitre 5 présente et justifie le choix de cette solution.

Chapitre 4

Administration de serveurs J2EE

Sommaire

Ce chapitre est consacré à la présentation des serveurs d'applications J2EE (Java 2 Enterprise Edition) [1] et à leur administration.

Un serveur d'applications J2EE est un assemblage de *services* qui offrent des propriétés non fonctionnelles aux applications hébergées par le serveur. Par défaut, un serveur J2EE est déployé sur une seule JVM. Cependant, afin d'améliorer les performances et tolérer les pannes, les serveurs J2EE sont déployés dans des environnements distribués comme les grappes. La gestion des grappes est actuellement une tâche trés complexe et préoccupe les chercheurs et les industriels. Dans ce chapitre, nous nous concentrons sur les serveurs d'applications « libres », avec un intérêt particulier pour leur administration dans les environnements grappes.

L'architecture des serveurs J2EE est complexe et l'administration de l'intergiciel présente actuellement plusieurs défis. Pour cette raison, nous pensons que l'adoption de cet environnment comme étude de cas pour cette thèse permet de couvrir plusieurs aspects de l'administration de systèmes distribués d'une manière générale.

Le chapitre est structuré de la manière suivante. En 4.1, nous présentons les principes du standard J2EE et nous décrivons en 4.2 l'architecture des serveurs d'applications J2EE. Nous décrivons en 4.3 les standards d'administration J2EE et nous montrons qu'ils ne couvrent pas tous les besoins. Nous décrivons en 4.4 l'architecture des trois serveurs d'applications J2EE libres

1. Le prototype développé dans cette thèse est basé sur la version 1.4 de J2EE. La version J2EE 1.5 n'était pas encore implantée par les serveurs d'applications J2EE et ne remet pas en cause l'architecture de l'intergiciel et son administration.

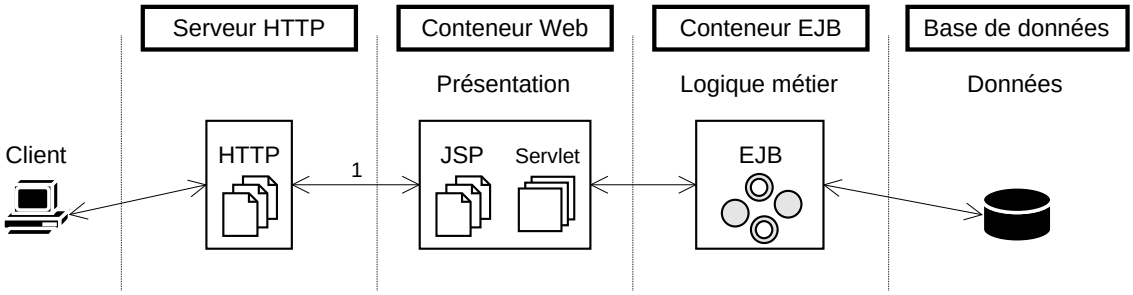

FIG. 4.1 – *Logiciels formant un serveur J2EE*

les plus connus[2] et nous justifions pourquoi les limitations de l'administration de ces serveurs sont liées à leurs architectures. Enfin, nous décrivons en 4.5, les défis de l'administration de serveurs J2EE.

4.1 Le standard J2EE

Le but de cette section est d'expliquer les principes du standard J2EE. La description détaillée de l'architecture de l'intergiciel est plutôt présentée dans la section 4.2.

Comme le montre la figure 4.1, un serveur d'application J2EE est généralement composé de plusieurs étages :

- un serveur HTTP qui sert à stocker les pages Web statiques,
- un conteneur de servlets et de pages JavaServer (JSP), qui exécute le code de présentation de l'application J2EE,
- un conteneur d'Enterprise Java Beans (EJB) qui exécute le code métier de l'application,
- un serveur de base de données qui permet de stocker les données.

Ces logiciels communiquent entre eux pour permettre le déroulement de l'application complète. Ces communications peuvent se faire par divers moyens, le plus répandu étant basé sur TCP/IP comme RMI. Grâce à ce protocole de communication à distance, les programmes peuvent être répartis sur des ordinateurs différents. Par conséquent, chaque niveau peut être hébergé par son propre ordinateur.

Nous décrivons brièvement, dans la suite, les différents logiciels implantant l'architecture J2EE.

4.1.1 Le serveur HTTP

Un serveur Web est basé sur le protocole de communication HTTP qui est mis en œuvre sur TCP/IP. Le protocole HTTP permet de formuler des requêtes dont on souhaite recevoir le résultat, sous la forme d'un document qui peut être aussi bien du texte que des images ou du son, de la vidéo ou des animations interactives. Le serveur HTTP libre le plus connu est Apache [Apa], de la Fondation Apache. Dans une architecture J2EE, le rôle d'un serveur HTTP est d'héberger les pages Web statiques. Pour le traitement des données dynamiques, la requête du client est transférée au conteneur Web comme le montre la flèche 1 dans la figure 4.1.

2. Nous nous limitons aux serveurs d'applications libres car nous n'avons pas l'accès à suffisamment d'informations sur les serveurs propriétaires comme Websphere ou Weblogic.

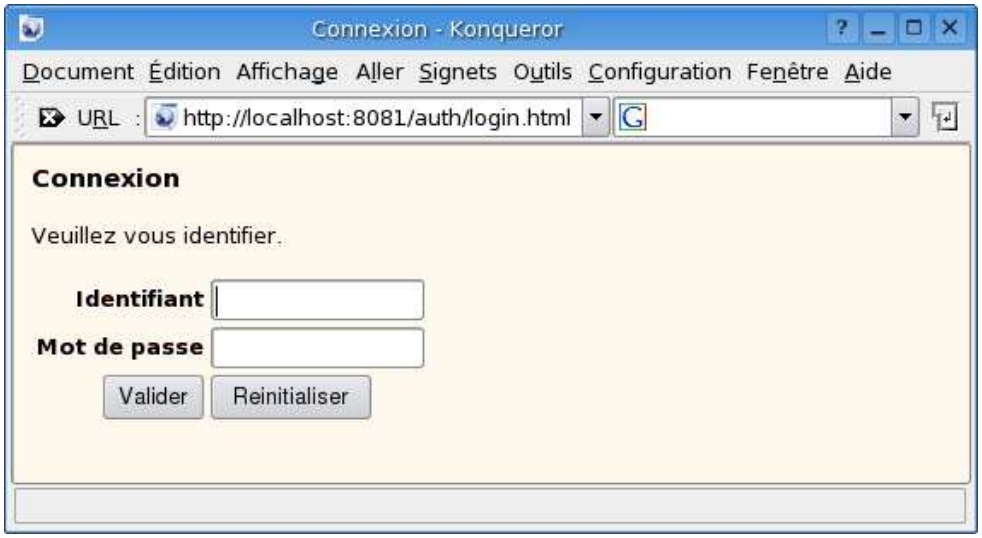

FIG. 4.2 – *Exemple de formulaire.*

La figure 4.2 présente un exemple de formulaire dont la validation (en appuyant sur le bouton « valider ») précédée par le remplissage des champs vides permet d'appeller une JSP (Java Server Page). La JSP prend les données en entrée et affiche une autre page Web après traitement des données.

4.1.2 Le conteneur Web

Le conteneur Web est un programme qui est au cœur de la spécification J2EE puisqu'il met en œuvre deux types de composants J2EE : les servlets et les *JavaServer Pages* ou JSP. Ces spécifications définissent des moyens de générer des contenus de pages Web dynamiquement. Les programmes hébergés par le conteneur Web (Servlets et JSP) sont appelés des composants Web.

Nous décrivons, dans ce qui suit, les principes des JSP et des Servlets en utilisant l'exemple du formulaire de la figure 4.2 .

Les servlets

L'interface de programmation des servlets encapsule entièrement le protocole HTTP sous la forme d'objets et de méthodes le représentant. Il est possible d'obtenir très simplement des informations sur la requête reçue et la connexion TCP entre le client et le serveur HTTP, ainsi que de lire les données envoyées par le client et d'écrire les données retournées au client. La rédaction d'une réponse à une requête se fait en employant l'interface `java.io.PrintWriter`.

D'autre part, une servlet doit implanter des fonctions appelées par le conteneur Web pour gérer son cycle de vie. Les fonctions fournies permettent d'initialiser une instance de la servlet, de l'appeler pour traiter une requête reçue et de la détruire. Une instance de servlet peut servir au traitement de plusieurs requêtes HTTP. Sa destruction ne survient généralement qu'après une longue période d'inactivité ou à la terminaison du conteneur.

On distingue deux types de servlets : celles dont une instance supporte plusieurs traitements simultanés de requêtes et celles sur lesquelles les appels doivent être sérialisés par le conteneur Web, car un état interne à l'instance ne supporte pas des accès concurrents de plusieurs fils d'exécution (*threads*). Pour les servlets du premier type, un conteneur Web doit gérer une réserve

d'instances pour pouvoir paralléliser le traitement de requêtes simultanées, alors que tous les appels de servlets du deuxième type peuvent être dirigés sur une unique instance.

Bien que les servlets sont intéressantes pour les performances, on leur préfère maintenant les JSP qui sont en fait des extensions qui les remplacent. En effet, les JSP sont compilées par le conteneur Web sous la forme de servlets pour les exécuter.

Les JSP se révèlent un moyen de créer des servlets beaucoup plus simplement qu'en les écrivant directement. Elles permettent d'écrire tout le code HTML ou XML d'un document sans avoir à l'encapsuler dans des appels fastidieux à l'interface `PrintWriter`. Elles dispensent également l'auteur de l'apprentissage d'une bonne partie de l'interface de programmation des servlets.

Les JavaServer Pages

Les pages JavaServer (ou JSP) sont des documents Web au format HTML ou XML contenant par endroits des marqueurs spéciaux délimitant du code Java qui doit être exécuté dans le conteneur pour générer le contenu de cette zone du document.

Le code de la figure 4.3 est une JSP qui est appelée quand on valide le formulaire de la figure 4.2. On y voit du code Java délimité par les marqueurs `<%` et `%>` à l'intérieur du code HTML de la page. Les variables `request`, `session` et `out` utilisées dans cette page constituent une partie des variables prédéfinies dans toute JSP. La variable `request` sert à récupérer les paramètres qui ont été saisis dans le formulaire et envoyés lors de la requête. Dans l'exemple, nous montrons trois façons différentes d'inclure du texte dans la réponse : en incluant la section java, en utilisant le délimiteur `<%=` ou en appelant une méthode d'écriture sur l'objet `out`, de type `java.io.PrintWriter`. Par ailleurs, du code HTML peut être inclus à l'intérieur de sections conditionnelles ou de boucles.

La figure 4.4 est l'affichage d'une réponse à une requête réussie exécutant le code de la figure 4.3.

Synthèse

Le conteneur Web, en reposant principalement sur la gestion du protocole HTTP pour l'envoi de documents HTML, est un outil destiné à la gestion de la présentation des applications d'entreprise. La génération des données à présenter peut elle-même être effectuée directement par des servlets ou des JSP, car ce sont des programmes Java à part entière, disposant de toute l'interface de programmation de base de J2EE. Néanmoins, les interfaces de programmation fournies par les serveurs d'EJB incitent les développeurs à préférer ces derniers pour la partie métier de l'application.

La séparation entre la partie logique et la partie présentation de l'application permet à des développeurs spécialistes, appelés concepteurs Web de se concentrer sur la tâche d'organisation de l'affichage des données à l'écran de l'utilisateur et aux développeurs d'EJB de se focaliser sur les aspects métiers des applications J2EE.

4.1.3 Le conteneur d'EJB

Dans le chapitre 3, nous avons présenté les EJB parmi d'autres modèles à composants. Dans cette section, nous nous concentrons plutôt sur le rôle de l'intergiciel et sur les différents services qui le composent.

Le code métier d'une application d'entreprise est la partie des programmes qui effectue les traitements remplissant les objectifs de l'application. Le rôle d'un conteneur d'EJB est de prendre

```
<%@ page language="java" pageEncoding="UTF-8" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.naming.InitialContext" %>
<%@ page import="fr.imag.*" %>
<html>
<head>
<title>Page principale</title>
</head>
<body bgcolor="#FEF8ED">
<%
String login = (String) request.getParameter("login");
String password = (String) request.getParameter("password");
if (login != null) {
if (password == null) password = "";
AuthArgs a = new AuthArgs(login, password);
session.setAttribute("auth", a);
}
AuthArgs a = (AuthArgs) session.getAttribute("auth");
if (a == null) { %>
Informations d'identification manquantes.
<% return;
} else {
InitialContext ctx = new InitialContext();
UserManagerHome h = (UserManagerHome) ctx.lookup(UserManagerHome.JNDI_NAME);
try {
ArrayList l = h.create().getGroups(a);
UserData user = (UserData) l.get(0); %>
<h2><%=user.getLogin()%> : <%=user.getName()%> (<%=user.getEmail()%>)</h2>
<table border="1">
<tr><th>Groupe</th><th>Utilisateurs dans le groupe</th>
<% for (int i=1; i<l.size(); i++) {
ArrayList list = (ArrayList) l.get(i);
GroupData group = (GroupData) list.get(0); %>
<tr><td><%=group.getName()%> (<%=group.getDescription()%>)</td><td>
<%  for (int j=1; j<list.size(); j++) {
UserData ud = (UserData) list.get(j);
if (j > 1) out.print(", ");
out.print(ud.getName());
} %>
</td></tr>
<% } %>
</table>
<% } catch (AuthenticationException e) { %>
Informations d'identification erronees.
<% }
}
%>
</body>
</html>
```

FIG. 4.3 – *Code de la JSP* /auth/doLogin.jsp.

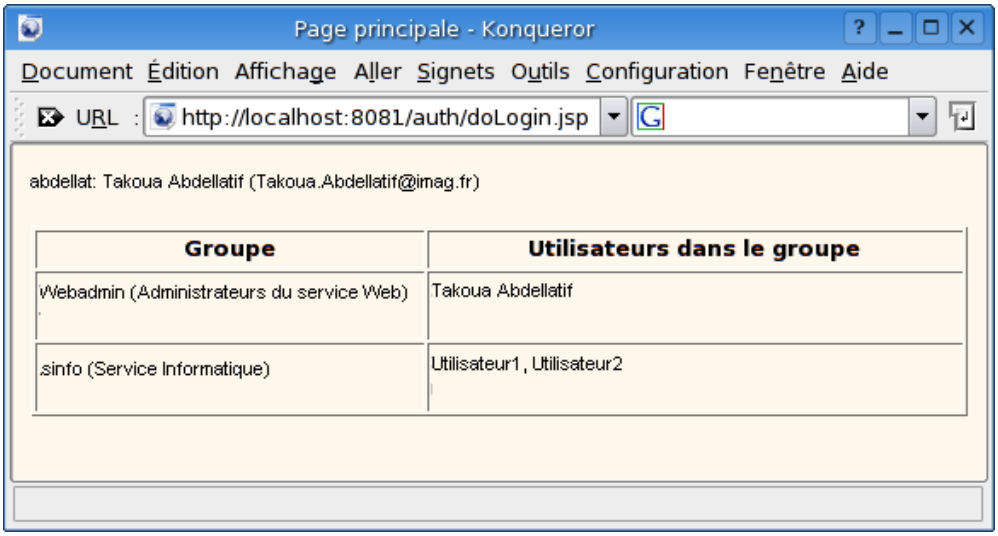

FIG. 4.4 – *Un affichage de la page JSP* /auth/doLogin.jsp.

en charge l'exécution de ce code métier. Pour cela, le code doit être construit sous la forme de composants EJB et suivre des règles d'interfaçage avec le conteneur.

Le but de l'utilisation d'un conteneur d'EJB est de simplifier la tâche du développeur d'applications en lui évitant d'écrire du code non fonctionnel, c'est-à -dire des programmes généralement complexes et non directement liés à la nature de l'application, mais nécessaires pour sa mise en œuvre. Pour ce faire, le conteneur fournit un ensemble de services couramment utilisés dans les applications d'entreprise. Parmi eux, nous citons les services suivants.

La persistance Les mécanismes de gestion de la persistance des beans entités par le conteneur évitent au développeur d'écrire des programmes dans un langage de base de données pour stocker l'état de ces objets. Les gains en temps de développement sont importants, car ces langages sont complexes, et les risques d'erreur sont élevés.

C'est le conteneur d'EJB qui est chargé de les définir lui-même avec l'aide d'informations de configuration pour déterminer où sont stockées les données. Il peut également effectuer des optimisations sans changer le code de l'application. Par exemple, il peut mettre en œuvre un cache qui améliore les performances de l'application sans demander aucun effort supplémentaire pour le programmeur.

Les relations Des beans entités peuvent avoir des relations entre eux. Ces relations reflètent les relations entre les données de la base de données représentées par les beans.

Les transactions La gestion des transactions demande une coopération parfois complexe entre les applications, les ressources et le coordinateur de transaction. Grâce au conteneur d'EJB, il est possible de gérer les transactions en déclarant dans le descripteur de déploiement de chaque bean la méthode de transaction à mettre en œuvre par le serveur d'application. La gestion des erreurs dans l'exécution des beans est entièrement homogène et prise en compte dans la gestion des transactions.

La sécurité De la même manière, les contrôles d'accès aux beans peuvent être vérifiés par le serveur d'application ; la technique pour le faire étant spécifique au conteneur.

Cycles de vie et réserves de ressources La nature des beans définit précisément leur durée de vie dans le conteneur d'EJB. Un bean entité a une durée de vie liée à celle de la

donnée qu'il représente et un bean session n'existe que pendant la session d'un client avec le serveur d'applications. Quand un bean n'existe plus, le conteneur d'EJB prend en charge la libération de la mémoire et des ressources utilisées. Dans un souci d'efficacité, il peut les garder en réserve pour les réutiliser rapidement pour d'autres requêtes. Cette technique peut diviser par un facteur très important le nombre d'allocations effectuées dans le conteneur d'EJB et éviter, par conséquent, des exécutions nombreuses et très coûteuses du ramasse-miettes Java.

Dans la section 4.2, nous montrons l'architecture de ces différents services ainsi que leur assemblage dans un serveur d'applications.

4.1.4 La base de données

Pour les applications d'entreprise, les systèmes de stockage généralement utilisés sont des bases de données. Pour les applications J2EE, c'est le conteneur d'EJB qui se charge de la persistance des beans entités dans des bases de données. Pour ce faire, il lui est possible d'utiliser des interfaces de programmation spécifiques aux bases de données utilisées. Une interface de programmation a été définie pour uniformiser les accès en Java aux bases de données relationnelles : *Java DataBase Connectivity* (JDBC). Tous les conteneurs d'EJB permettent d'accéder à des bases de données en utilisant JDBC.

JDBC

L'interface de programmation JDBC permet d'établir des connexions avec des bases de données pour générer et exécuter des requêtes SQL et en récupérer les résultats. Elle fournit un moyen de convertir des types SQL en objets Java directement manipulables par l'application. Les logiciels qui implantent l'interface JDBC sont appelés des pilotes. Ce sont, en effet, des programmes qui, en implantant tous la même interface, permettent d'uniformiser les moyens d'accès aux différentes bases de données. A chaque type de base de données correspond donc au moins un pilote JDBC.

Les requêtes SQL peuvent être interprétées à chaque requête, ou précompilées par le pilote pour être exécutées plusieurs fois plus efficacement. On parle alors d'instructions *préparées*, dans lesquelles il est toutefois possible de remplacer des paramètres par des valeurs différentes à chaque appel. Enfin, il est possible d'exécuter des procédures stockées dans la base de données.

Les mécanismes fournis par JDBC permettent de regrouper plusieurs requêtes d'une connexion dans une unique transaction qu'il est possible de valider ou d'annuler.

Des extensions ont été spécialement ajoutées à la spécification JDBC pour les environnements J2EE, dans lesquels elles doivent toujours être fournies. Elles concernent en particulier la gestion de réserves de connexions et les transactions réparties.

Les réserves de connexions Comme nous l'avons évoqué pour le conteneur d'EJB, les procédures de création et de destruction d'objets sont généralement coûteuses. C'est évidemment le cas pour les connexions aux bases de données, d'autant plus qu'elles demandent généralement l'ouverture de connexions réseau. Il n'est donc pas rare de voir des créations de connexions durer un temps de l'ordre de la seconde. L'extension à JDBC fournie par les serveurs J2EE permet de partager un certain nombre de connexions à l'intérieur d'une application de type serveur, comme un conteneur d'EJB. Lorsqu'un programme demande à un conteneur d'EJB une connexion à une base de données, le conteneur lui fournit un objet qui présente la même interface qu'une connexion JDBC. L'appel d'une fermeture d'une connexion est généralement intercepté pour

placer la connexion en réserve pour une réutilisation future. Cette optimisation est nécessaire pour une exécution efficace d'un serveur d'application soumis à une grande charge. C'est à la charge de l'administrateur du serveur d'applications de préciser des informations comme les nombres minimum et maximum de connexions en réserve ainsi que leur durée de vie.

4.1.5 Synthèse

L'architecture J2EE est composée de plusieurs étages : le serveur HTTP pour stocker les pages Web statiques, le conteneur Web pour exécuter les pages Web dynamiques grâce aux JSP et aux Servlets, le conteneur d'EJB pour exécuter le code métier et la base de données. Le but de cette répartition est de simplifier le développement des applications J2EE. En effet, la spécification sépare la partie présentation de la partie métier. Chaque partie peut être développée séparément par des spécialistes du domaine. De plus, pour la partie EJB, c'est à dire la partie métier, le conteneur d'EJB se charge de gérer les propriétés non-fonctionnelles telles que la persistance, les transactions, la sécurité et le cycle de vie des ressources.

4.2 Architecture de serveurs d'applications J2EE

Dans cette section, nous présentons l'architecture des serveurs d'applications d'une manière générale et en 4.4, nous décrivons les architectures des serveurs d'applications « libres ». Nous distinguons deux architectures : une centralisée où tous les services sont localisés dans une même JVM et une architecture distribuée où les services sont répartis sur des machines différentes. Pour la configuration centralisée, nous décrivons les mécanismes de chargement de classes Java dans un serveur d'applications et son impact sur le déploiement et le redéploiement des services. Pour la configuration distribuée, nous nous concentrons sur le cas des grappes et nous introduisons un modèle de déploiement à plus grande échelle, conçu récemment par IBM et appelé *edge computing*.

4.2.1 Configuration centralisée

Comme le montre la figure 4.5, un serveur d'applications est formé d'un ensemble de *services*. Le conteneur Web et le conteneur EJB sont deux services importants puisqu'ils représentent les environnements d'exécution des composants J2EE (JSP/servlets et EJB).

Les autres services interagissent avec les conteneurs Web et EJB, via des interfaces bien spécifiées, afin de fournir des propriétés non-fonctionnelles aux applications. Les services les plus utilisés sont les suivants.

- Le service *Registry* implante l'interface *Java Naming and Directory Interface* (JNDI). Il offre un système de nommage pour les applications.
- Le service de base de données gère les *datasources* qui sont des objets gérant les connections aux bases de données utilisant *Java DataBase Connectivity* (JDBC).
- Le service de Transaction implante la spécification *Java Transaction API* (JTA) pour la gestion des transactions.
- Le service *Java Message Service* (JMS) spécifie, comme son nom l'indique, un moyen d'échanger des messages entre des applications en suivant un modèle de publication et d'abonnement.
- Le service JMX permet la supervision à distance des différentes parties du serveur.
- Le service EAR est utilisé pour déployer les applications J2EE complètes (c'est à dire les applications archivées dans des fichiers EAR qui contiennent des fichiers EJB-JAR et des

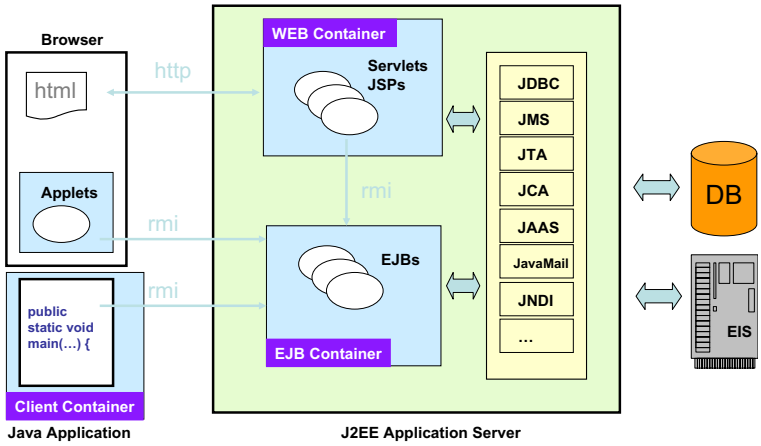

FIG. 4.5 – *Architecture d'un serveur d'application : un assemblage de services*

fichiers WAR). Le service délègue le déploiement de la partie Web au conteneur Web et la partie EJB au conteneur EJB.

Intérêt de la configuration centralisée

L'utilisation d'un serveur d'applications sur une seule machine présente plusieurs avantages.

D'une part, la localisation sur une même JVM des différents logiciels du serveur d'applications est une commodité pendant la phase de développement d'applications. En effet, ceci est plus pratique pour le test et le redémarrage des applications ainsi que la mise à jour du code du serveur.

D'autre part, la configuration centralisée du serveur peut donner de meilleures performances pour certaines applications. C'est le cas, par exemple, quand le traitement d'une requête nécessite plusieurs accès successifs à la base de données. Mettre la base de données sur la même machine que le conteneur EJB permet alors de réduire la bande passante et le temps de réponse aux requêtes.

Par ailleurs, dans une configuration centralisée, il est possible d'améliorer les performances en utilisant des interfaces locales pour accéder aux beans plutôt que des interfaces distantes. Dans ce cas, le conteneur d'EJB enregistre les beans directement dans le système de nommage JNDI au lieu de mettre des talons vers des serveurs. Par conséquent, les appels de méthode sont effectués directement et les paramètres sont passés par référence plutôt que par copie. Aucune sérialisation d'objet n'est nécessaire.

Cas de JOnAS

La figure 4.6 présente l'architecture de JOnAS. Les services sont développés par des contributeurs différents. Par exemple, le service Web encapsule le serveur Tomcat ou Jetty, le service Transaction est implanté par JOTM, le service JMS est fourni par JORAM, le service Web Services est offert par Axis, etc.

Comme tous les serveurs d'applications J2EE, les communications entre les conteneurs Web et EJB et les autres services se font en utilisant des interfaces standards. Des éléments de gestion des services, non représentés dans la figure, permettent de configurer et de démarrer les services.

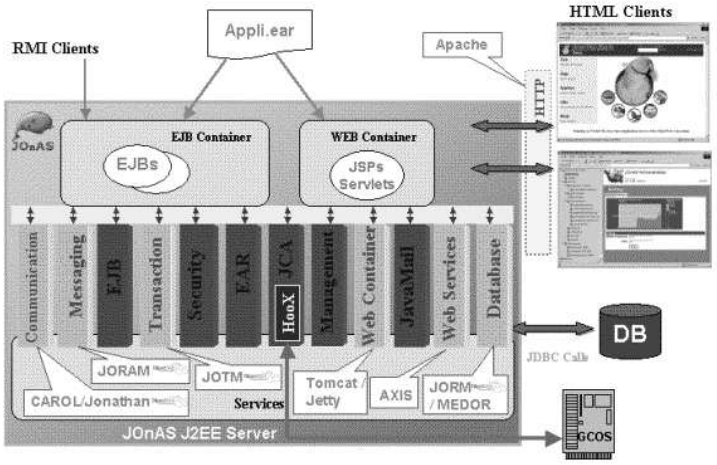

FIG. 4.6 – *Architecture du serveur JOnAS*

Un fichier de configuration global permet à l'administrateur de choisir les services à démarrer avec le démarrage de JOnAS et chaque service dispose d'un fichier de configuration séparé.

Plus de détails sur l'implantation de cette architecture seront donnés dans le chapitre 6 traitant la réingénierie de JOnAS.

Chargement de classes et déploiement

Il est important de comprendre le chargement de classes dans J2EE pour comprendre les problèmes liés à la reconfiguration d'implantation du serveur et des applications.

Rappelons d'abord le chargement de classes dans Java. Le *class loader* Java permet de charger dynamiquement des classes Java au sein d'une machine virtuelle, à partir d'une source définie : système de fichiers, réseau, etc. Le rôle d'un *class loader* est de créer des objets `Class` à partir de fichiers .class. Pour ce faire, chaque *class loader* implante une méthode `loadClass(String name)` appelant, à son tour, la méthode `findLoadedClass(String name)` pour vérifier que la classe n'a pas été chargée et stockée dans un cache. Si ce n'est pas le cas, `loadClass` utilise `defineClass` pour construire un objet `Class` à partir d'un tableau d'octets. Notons qu'un *class loader* peut également déléguer le chargement d'une classe à un autre *class loader*.

Il est important de noter qu'une classe est liée au *class loader* qui l'a chargée : deux exemplaires d'une même classe chargée par deux *class loaders* différents sont considérés comme étant deux classes différentes. En conséquence, l'utilisation d'une classe à la place de l'autre lève une exception `ClassCastException`.

Dans J2EE, plusieurs approches de chargement de classes existent pour gérer le déploiement des composants d'une application [SB98]. Ces approches se basent sur le même principe du chargeur-par-déploiement (*loader-per-deployment*). La figure 4.7 présente le cas de JOnAS. Le mécanisme de chargement de classes respecte une hiérarchie en arbre de *class loaders* : chaque *class loader* a un père auquel il peut déléguer le chargement. La délégation se fait dans le sens

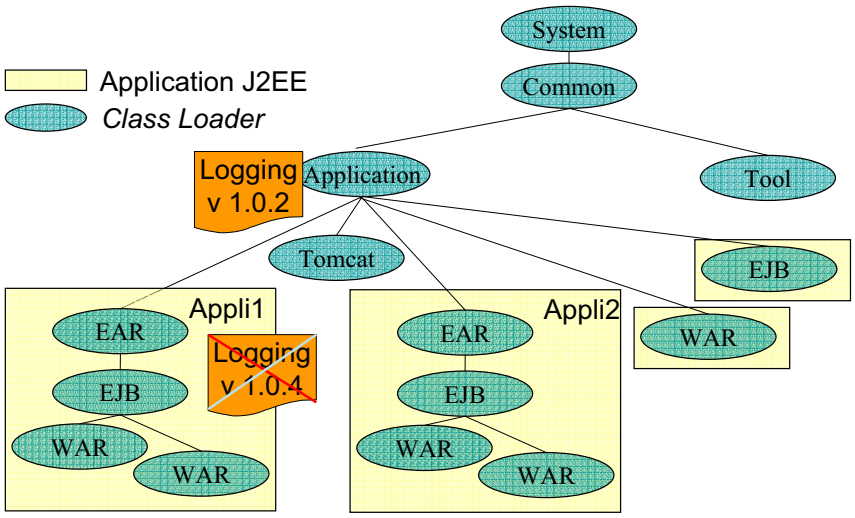

FIG. 4.7 – *Mécanisme de chargement de classes dans J2EE: exemple de JOnAS.*

partant des feuilles vers la racine, ce qui veut dire que les classes sont chargées par les racines des sous-arbres avant d'être chargées par les fils.

Dans JOnAS, le *class loader* `common` permet de charger toutes les classes des services du serveur. Un autre *class loader* fils, appelé `application`, charge les applications. À chaque module simple (EJB-JAR ou WAR) est associé un *class loader* unique. Dans le cas de EAR (contenant des modules EJB et Web), plusieurs *class loaders* peuvent être créés et collaborer pour charger l'application. La motivation derrière la séparation des *class loaders* consiste principalement en l'isolation des applications.

Néanmoins, cette solution présente plusieurs inconvénients. D'abord, le chargement de classes des services est simpliste ; il ne permet pas la mise à jour du code d'un service sans arrêter les autres. En effet, le code de la plupart des services est chargé par un *class loader* commun. Par conséquent, il n'est pas possible de charger deux versions différentes d'un même service et donc d'effectuer des reconfigurations d'implantation. D'autre part, les classes Java de l'intergiciel sont *accessibles* aux applications. Vue la hiérarchie des *class loaders*, ceci implique que les classes des applications utilisent les librairies de l'intergiciel et pas celles empaquetées dans leurs paquets. En effet, quand les applications sont déployées avec des classes ayant les mêmes noms que des classes de l'intergiciel, ce sont les classes de l'intergiciel qui sont prises en compte (car déjà chargées). Par exemple, dans la figure 4.7, l'application `Appli1` arrive avec la version `v1.0.4` de la librairie `Logging`. La même librairie étant déjà chargée par l'intergiciel mais en version `v1.0.2`, sera prise en compte par `Appli1`. Par conséquent, des conflits de versions peuvent avoir lieu. Ensuite, des problèmes de sécurité peuvent avoir lieu (des applications malveillantes peuvent appeller, par exemple, les méthodes `public static` de l'intergiciel et endommager le serveur).

Synthèse

Un serveur d'applications J2EE est un assemblage de services offrant des propriétés non fonctionnelles aux applications. Les standards J2EE ne définissent pas les interfaces des services ni la manière dont ils doivent communiquer. La conception de l'architecture du serveur est laissée

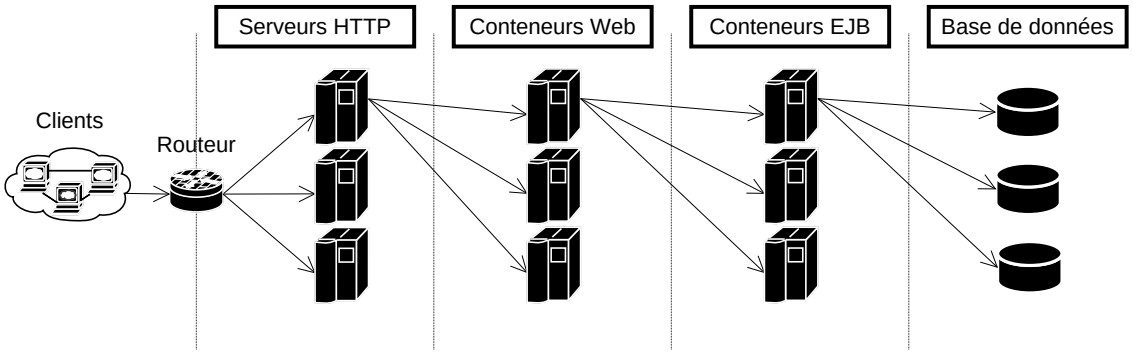

FIG. 4.8 – *Répartition de serveur J2EE en grappe*

à la charge des fournisseurs du serveur J2EE. Par défaut, un serveur d'applications est installé sur une même machine virtuelle Java. Cette configuration centralisée peut être intéressante pour faciliter le développement et pour les performances de certaines applications.

Le déploiement des services et des applications repose, en général, sur une hiérarchie de *class loaders*. Cette solution présente deux inconvénients majeurs. D'une part, elle ne permet pas de gérer proprement la mise à jour d'un service sans affecter les autres parties du serveur. D'autre part, les classes du serveur sont accessibles aux applications, ce qui peut entrainer des problèmes de cohérence et de sécurité.

4.2.2 Configuration distribuée

Les serveurs d'applications J2EE doivent assurer une certaine qualité de service. Pour ce faire, il faut éviter les ruptures des services provenant des pannes ou des ralentissements des serveurs occasionnés par le traitement de multiples requêtes simultannées. Il faut, également, assurer un temps de réponse acceptable aux utilisateurs. Pour fournir ces propriétés de *tolérance aux fautes* et de *support de la charge*, les serveurs d'applications sont distribués sur plusieurs machines.

Parmi les environnements distribués, nous nous intéressons d'abord aux grappes de machines et nous introduisons, par la suite, le modèle du *Edge-computing*.

Les grappes J2EE

Une grappe est un ensemble de machines interconnectées par un réseau local à haut débit. Les machines sont en général homogènes quant à leurs configurations matérielles ou logicielles. Le principe d'une grappe J2EE est d'étaler chaque étage sur plusieurs machines comme le présente la figure 4.8. Les données d'un même étage sont partagées de manière à ce qu'aucune donnée utile ne se trouve inaccessible ou perdue à la suite d'une panne.

Le partage de données entre des nœuds peut se faire de manière fiable de différentes manières.

- Les données peuvent être localisées sur un espace de stockage fiable partagé. Il peut être alors nécessaire de synchroniser les nœuds entre eux lors des accès concurrents pour les gérer correctement. C'est le cas, par exemple, pour les entity beans persistant dans une même base de donnée.
- Les données peuvent être dupliquées sur les nœuds. Chaque modification est alors signalée par un message à tous les nœuds qui en contiennent une copie. Pour obtenir des copies cohérentes, il est nécessaire que les messages soient tous reçus et dans le même ordre. Un

système de communication de groupe est utilisé pour assurer l'ordre et la cohérence des opérations de mise à jour. Cette technique est généralement utilisée pour les étages Web et EJB.

La grappe est un moyen d'augmenter la capacité de traitement des requêtes Web par rapport à l'utilisation d'une unique machine. Cependant, ceci n'est vrai que dans la mesure où les communications entre les nœuds pour leur synchronisation ne pénalisent pas trop le traitement des requêtes. Dans le cadre de cette thèse, nous avons effectué une évaluation de performance de JGroups [ACL04] qui est le système de communication de groupe le plus utilisé dans les serveurs d'applications « libres ». Le lecteur intéressé peut trouver dans [ACL04] les résultats obtenus. Nous montrons que, dans une grappe J2EE, l'utilisation de IP_multicast dans le protocole de communication de groupes n'est pas toujours efficace. En effet, dans le cas d'un très grand nombre de requêtes, nous observons une forte perte de messages entre les nœuds membres d'un groupe, ce qui conduit à plusieurs retransmissions de messages et donc à une dégradation de la latence et du temps de réponse. L'utilisation d'une configuration de JGroups basée sur une communication point-à-point utilisant TCP, est dans ce cas plus efficace. Nous montrons également, que les couches applicatives Java au dessus de JGroups masque l'effet du réseau et qu'une communication au dessus de FastEthernet est aussi efficace qu'une communication au dessus de GigabitEthernet.

Le choix de la répartition des requêtes peut s'effectuer à divers niveaux. Pour cela, des programmes intermédiaires doivent intervenir pour recevoir les requêtes et permettre de les distribuer vers les machines cibles selon des politiques différentes.

Répartition au niveau des routeurs

Le routeur peut aiguiller les requêtes d'un client Web vers des serveurs HTTP différents sans que le client s'en aperçoive. La caractéristique commune des solutions existantes est que l'adresse IP cible, souvent appelée une adresse virtuelle, ne désigne réellement aucune machine en particulier, mais peut être aiguillée par le routeur vers n'importe quelle machine de la grappe. Une technique trés utilisée est celle de la répartition par traduction d'adresse IP (plus exactement par la traduction d'adresse réseau NAT pour *Network Address Translation*). Pour chaque nouvelle connexion entrante destinée à l'adresse virtuelle, le routeur peut choisir une adresse IP de destination dans celles de la grappe.

Cisco LocalDirector et Linux Virtual Server/NAT sont des exemples de routeurs mettant en œuvre de la traduction d'adresse IP.

Liaison entre serveur HTTP et conteneur Web

Le but de cette liaison est de transférer les requêtes reçues par le serveur HTTP vers les serveurs Web d'une manière à équilibrer la charge sur ces serveurs.

Nous considérons ici le cas de Jakarta de la fondation Apache, qui développe le principal conteneur Web libre et le serveur Web libre le plus connu, Tomcat. L'ajout de nouvelles propriétés dans Apache, par exemple l'aiguillage de requêtes HTTP, est possible en développant des modules qui étendent le serveur HTTP. L'utilisation d'un module JK, nécessitant plusieurs informations de configuration au niveau du serveur, permet d'associer à certaines URL l'adresse d'un conteneur Web qui pourra les traiter. Quand le serveur reçoit une requête qui correspond à une de ces URL, le serveur HTTP la transfère à un conteneur Web.

Le transfert de requête s'effectue par le protocole spéficique ajp13 (*Apache JServ Protocol version 1.3*) : une connexion TCP est ouverte vers le conteneur Web et toutes les données qui

composent la requête sont passées dans une forme binaire pour un transfert rapide. Le protocole supporte la réutilisation de connexion pour plusieurs requêtes. Quand la version cryptée de HTTP est utilisée, toutes les informations de sécurité exploitables dans les servlets sont également passées au conteneur. Le client et l'application ne peuvent pas distinguer si la connexion s'est faite à travers un serveur HTTP intermédiaire ou si elle est directement reçue par un conteneur Web.

Par ailleurs, ajp13 permet d'attribuer des poids à chaque conteneur Web et de distribuer les requêtes en fonction de ces poids.

Afin que chaque requête puisse être traitée par un serveur Web, les données à états doivent être mises à jour d'une manière cohérente sur chaque serveur Web. Ceci est assuré dans Tomcat en dupliquant la session à chaque écriture sur tous les conteneurs Web de la grappe. Un système de communication de groupe, tel que JGroups, assure la cohérence entre les réplicas.

Interfaçage avec le conteneur EJB

Pour exécuter la logique métier de l'application J2EE, les servlets et JSP font des appels de méthodes sur des beans déployés dans le conteneur d'EJB. Ces appels nécessitent un mécanisme de communication entre les deux serveurs.

Nous rappelons que l'accès à l'interface `Home` d'un EJB passe par un système de nommage accessible par l'interface de programmation JNDI. Cette interface présente au programme client une arborescence, similaire à l'organisation classique d'un système de fichiers, dans laquelle il est possible de rechercher des éléments, d'en enregistrer ou de les lister. Quand un conteneur d'EJB déploie un bean, il enregistre son interface `Home` dans l'arborescence avec la fonction `bind()` et le client peut la récupérer avec la fonction `lookup()`. JNDI est généralement utilisé pour partager des informations entre des programmes potentiellement séparés dans des JVM différentes indépendamment du protocole de communication. Dans le cas de J2EE, le protocole de communication utilisé entre les EJB est RMI. Les objets enregistrés dans l'arborescence sont des talons qui permettent d'accéder aux implantations d'interface `Home` des beans. Il est possible de choisir l'implantation de JNDI utilisée par un programme client sans modifier son code. En effet, le système JNDI obtient le nom de cette classe dans la variable d'environnement standard : `java.naming.factory.initial` et la valeur de cette variable est généralement positionnée en modifiant des fichiers de configuration externes à l'application. En particulier, dans l'environnement grappe, JNDI peut être implanté de telle sorte que les requêtes soient servies par des serveurs EJB différents. En effet, le choix d'un serveur de destination peut-être effectué par le talon à chaque appel de méthode et ce sans que l'application cliente n'ait à être modifiée pour fonctionner en grappe.

Plusieurs politiques de dispersion sont possibles. Par exemple, le talon peut détecter qu'un serveur est en panne et décider de lui-même de diriger l'appel vers un autre serveur de sa liste et aucune erreur n'est remontée au client. Dans JOnAS, un simple algorithme de tourniquet *(round-robin)* sur les conteneurs EJB est actuellement implanté.

Actuellement, le nombre de machines utilisées par une grappe J2EE ne dépasse pas quelques dizaines de machines. Un autre modèle de distribution est proposé par IBM pour gérer la distribution de serveurs J2EE à une plus grande échelle.

Edge Computing

Plusieurs travaux [ADZ00] [CIG+03] [AFF+01] ont été menés dans l'industrie et dans la recherche pour augmenter les capacités des serveurs J2EE en utilisant les grappes ou les centres

de données (*data centers*). Ces travaux se focalisent principalement sur l'allocation optimale des ressources.

Une autre approche est récemment proposée par IBM qui a introduit un nouveau modèle, appelé *edge-computing* pour déployer les serveurs J2EE [edga][edgb][edgc]. En plus des infrastructures centralisées (les centres de données contenant les bases de données et les systèmes propriétaires), d'autres serveurs appellés *edge servers* (ES) ou « serveurs aux bords », localisés géographiquement à proximité des clients, permettent de délivrer les services d'une manière efficace. En effet, une partie des applications Web, ou même la totalité du serveur, peut être distribuée au niveau des ES pour réduire la bande passante du réseau utilisée et raccourcir le temps de réponse. Le système du *edge-computing* peut être vu comme un seul serveur d'applications virtuel où les conteneurs Web et les conteneurs EJB sont déployés sur le ES. Les bases de données sont en général déployées au niveau du centre de données. La manière avec laquelle les différents étages sont déployés dépend en fait de la nature des applications. Plusieurs travaux ont été réalisés pour définir la structure des applications J2EE et les architectures efficaces à déployer dans cet environnement [edga].

Néanmoins, l'administration du système distribué présente plusieurs défis que nous résumons dans les points suivants.

- **Hétérogénité et dynamicité** Contrairement aux grappes où les nœuds sont généralement homogènes, les ES peuvent être hétérogènes en termes de logiciels ou de matériels utilisés. De plus, un ES peut a priori quitter ou rejoindre le système dynamiquement. Certains départs peuvent être prévisibles à l'expiration de la période de disponiblité prévue des ES mais ils peuvent aussi être inattendus puisque personne ne peut empécher un administrateur de ES de retirer sa machine.
- **Passage à l'échelle** Le nombre des ES peut être très important, jusqu'à quelques millions et par conséquent le nombre de composants des applications et de l'intergiciel à administrer est très important également.
- **Gestion des fautes** La distribution à très grande échelle des ES augmente la probabilité des fautes à cause des goulots d'étranglements qui peuvent se produire au niveau du réseau. Un autre type de fautes est la panne d'un ES.
- **Sécurité** Les ES peuvent a priori appartenir à des personnes ou à des autorités différentes. Il est donc plus complexe de gérer les droits d'accès. Les problèmes de sécurité peuvent se produire, également, au niveau du réseau pour interconnecter les différents composants de l'intergiciel et des applications.
- **Découverte ou localisation** Pour déployer une application, les ES doivent être choisis de telle sorte que les applications soient déployées à proximité des clients. Il faut également que les machines cibles satisfassent des contraintes de ressources logicielles et matérielles nécessaires au bon fonctionnement de ces applications.

Face à cette complexité, il est nécessaire de bâtir des systèmes d'administration sophistiqués. En effet, il est nécessaire de construire des services de découverte et d'allocation de ressources ; le système de déploiement et d'observation doivent passer à l'échelle et le système d'administration doit tolérer les pannes et assurer la sécurité du système administré.

Synthèse

Les serveurs d'applications sont déployés sur des grappes de machines afin d'augmenter les performances des applications J2EE et d'assurer la continuité des services. Le principe des grappes est de dupliquer les conteneurs EJB, les conteneurs Web et dans certains cas les serveurs HTTP et la base de données. Cette duplication permet d'équilibrer la charge et de traiter les

requêtes même en cas de panne de l'une des machines. Pour ce faire, de nouveaux éléments sont introduits entre les étages J2EE comme les équilibreurs de charges et les sytèmes de communication de groupes dupliquant les données avec états d'une manière fiable.

À plus grande échelle, le *edge computing* est un modèle récent consistant à déployer les applications J2EE à l'échelle de l'Internet. Bien que ce modèle semble intéressant pour mieux gérer les pics de charge et optimiser l'utilisation des ressources, il introduit, en contre-partie, des problèmes liés à l'administration comme la sécurité et la vulnérabilité aux fautes.

4.3 Les standards de l'administration de plates-formes J2EE

Les standards J2EE se sont focalisés principalement sur l'instrumentation des serveurs d'applications et sur le déploiement des applications. L'instrumentation des serveurs est basée sur le standard JMX [jmx04] qui définit également l'accès à distance aux serveurs à administrer. Utilisant cette instrumentation, JSR77 [jsra] spécifie le modèle d'informations que les serveurs doivent exposer à l'administrateur. Enfin, JSR88 [jsrb] spécifie le déploiement des applications.

Nous décrivons dans cette section ces standards.

4.3.1 JMX

L'extension de Java pour l'administration (*Java Management Extension*) [jmx04] définit une architecture et un ensemble de services permettant d'administrer les applications Java[3].

La figure 4.9 illustre l'architecture de JMX qui est composée de quatre niveaux.

- **Le niveau instrumentation** Ce niveau correspond à l'instrumentation des ressources sous la forme de MBeans.
- **Le niveau agent** Les agents d'administration contrôlent les MBeans et les rendent accessibles aux applications d'administration distantes. Chaque agent a la responsabilité d'un ensemble de MBeans. Toutes les opérations d'administration requises sur un MBean doivent passer par son agent.
- **Le niveau connexions** Ce niveau spécifie des composants particuliers, appelés *connecteurs* ou *adaptateurs* de protocoles, permettant aux applications d'administration d'accéder à distance aux agents JMX en respectant divers protocoles de communication.
- **Le niveau gestionnaires** Ce niveau représente les gestionnaires qui communiquent avec les applications via les connecteurs.

Le niveau instrumentation

Pour être administrables, les ressources logicielles ou matérielles doivent être encapsulés dans des composants appelés MBeans (*Manageable Beans*). Ces composants définissent des interfaces de contrôle et permettent d'exposer les interfaces fonctionnelles à la couche administrative. La standardisation des interfaces permet une interopérabilité entre les produits développés et instrumentés avec JMX et les outils d'administrations faits par de tierces parties.

Les interfaces d'un MBean exposent :

- les attributs qui peuvent être lus ou modifiés,
- les méthodes permettant de modifier le comportement d'un MBean (par exemple une opération de réinitialisation),
- les méthodes envoyant des notifications correspondant à des événements particuliers.

3. Les librairies JMX sont maintenant intégrées à JDK1.5.

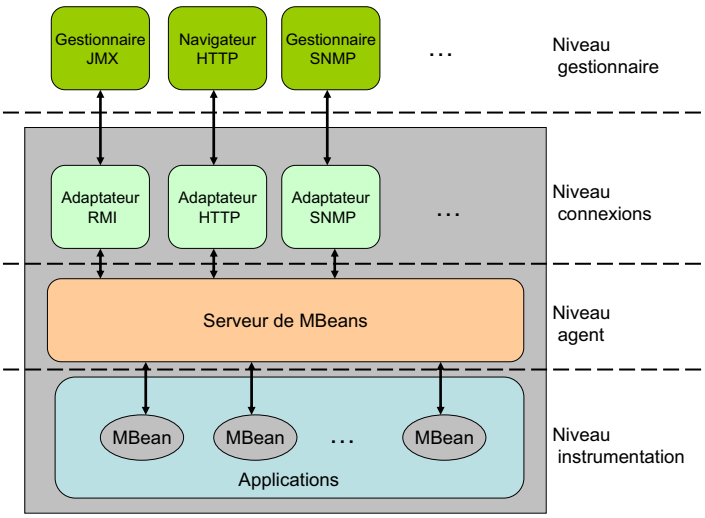

FIG. 4.9 – *Architecture d'un système JMX.*

Il existe deux types de MBeans : les MBeans standards (*Standard MBean*) et les MBeans Dynamiques (*Dynamic MBean*). Les MBeans standards se basent sur une interface d'administration fixée au moment de la compilation. Les attributs et les méthodes d'administration associés à une ressource sont déterminés de manière statique. Inversement, les MBeans dynamiques permettent de déterminer en cours d'exécution les attributs et les opérations d'administration associés à une ressource.

Dans un MBean standard, le nom de l'interface correspond au nom de la classe du MBean suffixé par la chaîne `MBean`. Ainsi le nom de l'interface d'administration correspondant au MBean `MyClass` serait `MyClassMBean`. Pour chaque attribut public représentant la ressource administrée, il doit exister une méthode de type `get/set` permettant respectivement de lire ou de modifier cet attribut. Le nom de ces méthodes dépend du nom de l'attribut associé. La syntaxe de déclaration de ces méthodes est décrite dans 4.10.

```
public AttributeType getAttributeName();
public void setAttributeName(AttributeType value);
```

FIG. 4.10 – *Accès aux noms d'attributs dans un MBean standard.*

Pour toute opération d'administration de la ressource, il doit exister une méthode correspondante dans l'interface d'administration du MBean.

L'exemple suivant définit un MBean de classe `MyClass` possèdant un attribut d'administration de type entier appelé `state`. Cet attribut peut être lu et modifié. De plus, notre MBean met en œuvre une méthode *reset* permettant de réinitialiser le MBean. L'interface d'administration et le code de la classe du MBean sont représentés dans 4.11

Un MBean dynamique est un objet java classique qui doit mettre en œuvre une interface prédéfinie appelée *DynamicMBean*. Cette interface permet de découvrir les attributs et les opérations d'administration associés au MBean en cours d'exécution. Au lieu d'exposer directement ces opérations au travers d'une interface d'administration, le MBean dynamique fournit une méthode permettant d'obtenir la liste de tous les attributs et la signature des méthodes associées. L'interface DynamicMBean est représentée dans la figure 4.12.

Le méthode `getMBeanInfo` retourne un objet de type `MBeanInfo` contenant la définition des

```
public interface MyClassMBean {
public Integer getState();
public void setState(Integer s);
public void reset();
}

public class MyClass implements MyClassMBean {
private Integer state = null;
private ... // autres attributs non relatif à l'administration
public Integer getState() {
return(state);
}
public void setState(Integer s) {
state = s;
}
public void reset() {
state = null;
}
}
```

FIG. 4.11 – *Exemple de code d'un MBean standard.*

```
Public MBeanInfo getMBeanInfo():
Public Object getAttribute(String attribute)
Public AttributeList getAttributes(String[] attributes)
Public void setAttribute(Attribute attribute)
Public AttributeList setAttributes(AttributeList attributes)
Public Object invoke(String actionName, Object[] params,
String[] signature)
```

FIG. 4.12 – *Interfaces d'un MBean dynamique.*

attributs et des méthodes d'administration du MBean. Les méthodes de type `getAttribute` permettent de lire les valeurs des attributs dont le nom est passé en paramètre. Les méthodes de type `setAttribute` permettent d'écrire la valeur des attributs dont le nom et la valeur sont passés en paramètre. La méthode `invoke` permet d'exécuter la méthode dont le nom est passé en paramètre avec les paramètres correspondants. Le paramètre signature est un tableau contenant le type des objets passés en paramètre lors de l'appel de la méthode. Le paramètre `params` est un tableau contenant la valeur des objets passés en paramètre.

Afin de permettre aux applications d'administration de réagir à des changements d'état ou à d'autres événements se produisant au niveau des ressources administrées, le modèle de notification JMX définit un mécanisme permettant aux MBeans de diffuser ces événements (appelés notifications). Ce modèle est de type *publish/subscribe* et fait intervenir les éléments suivants.

- Une classe (*Notification*) implante la nature de l'événement.
- Une interface (*NotificationBroadcaster*) permet d'émettre les événements. Elle définit principalement deux méthodes (*addNotificationListener* et *removeNotificationListener*) permettant respectivement à un objet de s'abonner à un événement ou de s'en désabonner. La méthode *handleNotification* implante le traitement à faire à la réception d'un événement.

Le niveau agent

Dans l'architecture JMX, toutes les opérations d'administration passent par le biais des agents. Les agents sont similaires aux agent SNMP ; ils servent d'intermédiaires entre le système administré et les gestionnaires. Un agent est composé d'un serveur de MBeans et d'un ensemble

de services aidant à la configuration, au déploiement et au contrôle des MBeans. Pour ce faire, ils présentent les interfaces nécessaires pour effectuer les opérations de contrôle suivantes :

- la lecture et la modification des attributs du MBean,
- l'appel de l'une des opérations définies par l'interface du MBean,
- la réception des notifications émises par le MBean,
- l'utilisation des services fournis par l'agent.

Le serveur de MBeans (*MBean Server*) est l'élément central d'un agent. Il maintient une liste de tous les MBeans contrôlés. Toutes les opérations d'administration appliquées sur un MBean sont effectuées à partir de l'interface `MBeanServer` du serveur de MBean. Cette interface définit des méthodes permettant d'instancier et d'enregistrer un nouveau MBean dans l'agent, d'accéder aux attributs du MBean et d'appeler les opérations qu'il implante.

Le serveur de MBeans émet la notification `jmx.mbean.created` lorsqu'un MBean a été enregistré. De même, lorsqu'un MBean est détruit, la notification `jmx.mbean.delete` est émise.

Concernant les services fournis par les agents, ils sont implantés sous la forme de MBeans et enregistrés au niveau du serveur de MBeans. Les services de téléchargement à distance et le service de surveillance sont les plus utilisées dans les applications utilisant JMX. Le service de téléchargement (`m-let service`) permet d'instancier et de télécharger de nouvelles classes de MBeans. Ce service utilise un fichier XML (`MLET tags`) contenant les informations nécesaires au téléchargement des MBeans. Le service de surveillance permet d'observer les changements d'états des MBeans et d'envoyer les notifications aux MBeans intéressés en passant par le serveur des MBeans.

Le niveau des connexions et des gestionnaires

Les connecteurs et les adaptateurs de protocole rendent les agents accessibles à distance. Cette partie est spécifiée par le standard JSR 160 [jsrc] qui définit les protocoles de communication et le format des messages. Les interfaces définies au niveau des gestionnaires sont trés proches des MBeans et les notifications émises par les MBeans sont propagées jusqu'aux gestionnaires. Le standard permet également de sécuriser les connexions entre les gestionaires et les agents et de définir des droits d'accès aux ressource administrées. Des solutions liées à la découverte de serveurs distribués sont proposés comme le protocole SLP (*Service Location Protocol*), Jini ou JNDI (*Java naming and Directory Interface*).

Synthèse

JMX définit une architecture d'administration flexible sous la forme de trois niveaux d'administration : le niveau instrumentation, le niveau agent et le niveau connexions. L'instrumentation est assurée en encapsulant le code à administrer sous forme de MBeans. Le niveau agent gère le cycle de vie des MBeans et définit des services de bases utiles pour l'administration des applications comme, par exemple, le service de téléchargement de MBeans et le service de surveillance. Enfin, pour accéder à distance aux MBeans, le standard JSR160 définit les interfaces et les protocoles des connecteurs entre l'agent et les gestionnaires.

Cependant, l'utilisation de JMX comporte certains inconvénients. Tout d'abord, la modélisation de l'application administrée par JMX reste insuffisante puisqu'elle ne permet pas de refléter la structure de l'application sous-jacente. En effet, la communication entre les MBeans n'est pas définie par le modèle et n'est pas exposée à l'administrateur de manière explicite. De plus, le modèle des MBeans est plat et il n'y a pas de notion de hiérarchie qui est très importante pour représenter des architectures logicielles. À cause des relations implicites entre les MBeans,

la reconfiguration de structure ou d'implantation d'un MBean n'est pas très bien définie. En pratique, la reconfiguration est généralement limitée à la modification dynamique de certaines valeurs d'attributs. De plus, le standard ne définit pas le déploiement distribué sur plusieurs JVM et ne spécifie pas la connexion entre un gestionnaire et plusieurs agents.

4.3.2 JSR77

Pour administrer les serveurs d'applications avec des outils développés par tierces personnes, le standard JSR77 [jsra] définit un modèle d'information que chaque serveur doit exposer à l'administrateur. Ce modèle représente tous les types d'objets faisant partie d'un serveur d'application J2EE. Ces objets sont appellés objets administrés ou *Managed Objects*. La figure 4.13 représente la hiérarchie d'objets administrés selon le modèle d'information de la spécification.

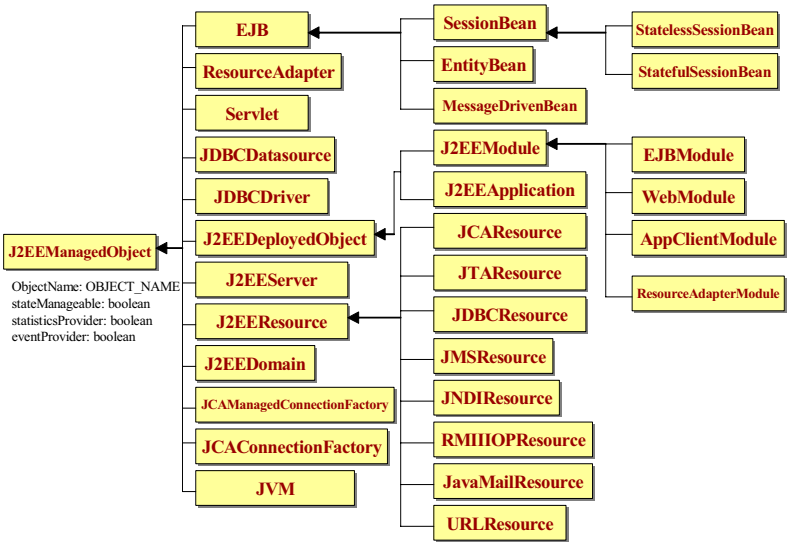

FIG. 4.13 – *Le modèle d'information du JSR77.*

Un objet administré est une abstraction d'une ressource, par exemple le serveur J2EE, le *J2EEDeployedObject* (qui peut être un EAR, un WAR ou un RAR), le pilote JDBC, etc. Chaque objet administré peut implanter un ensemble d'interfaces pour envoyer des notifications ou d'autres événements sur l'état de l'objet ainsi que des interfaces permettant de contrôler son cycle de vie et de fournir des informations liées aux performances. La figure 4.14 illustre le cycle de vie que doit implanter un objet administré.

JSR77 repose sur le standard JMX pour l'instrumentation et la connection du serveur à un ou plusieurs gestionnaires.

Enfin, dans J2EE, les MBeans encapsulent les entités à administrer et le serveur MBean doit être sous la forme d'un composant EJB. Ce composant est appellé le **MEJB**. À part le MEJB qui sert de point d'entrée au serveur, un ensemble de connecteurs JMX sont généralement implantés dans les serveurs.

Observer un serveur J2EE suivant la spécification du JSR77 implique les étapes suivantes :

- récupérer l'URL d'un des connecteurs JMX du serveur à administrer,
- créer un connecteur client en utilisant l'URL JMX,
- communiquer avec le serveur MBean (ou le MEJB) via le connecteur pour récupérer les informations sur les objets administrés à partir de leur encapsulation (les MBeans).

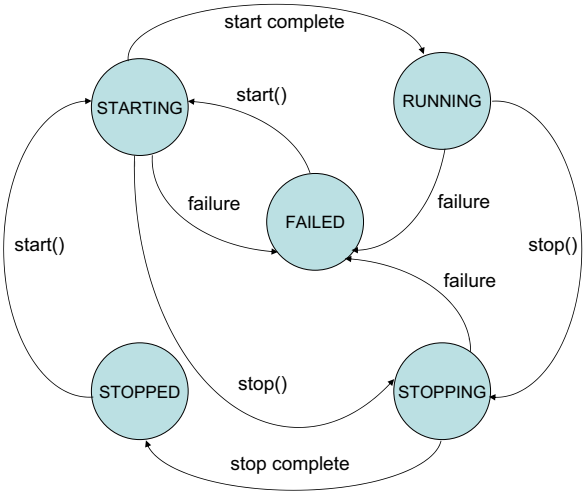

FIG. 4.14 – *Le cycle de vie défini par JSR77.*

La communication via les connecteurs est aujourd'hui standardisée (JSR160) et on a deux modes de communication : pull et push. Le manager a le choix entre interroger régulièrement le serveur des MBeans et s'abonner à la réception de certains évènements suivant le modèle de notification JMX.

La spécification JSR77, bien qu'elle permette de visualiser l'ensemble des objets constituant un serveur d'application, ne définit pas l'architecture de ce serveur. En particulier, le modèle de la spécification ne décrit pas les dépendences entre les services et la reconfiguration des objets administrés est restreinte à la modification de quelques paramètres. Par exemple, il n'est pas précisé comment reconfigurer la structure ou l'implantation d'un service du serveur suite à une panne ou à un pic de charge.

4.3.3 JSR88

La spécification de déploiement J2EE ou JSR88 [jsrb] est disponible depuis J2EE 1.4. La spécification permet une séparation entre l'intergiciel J2EE et les outils utilisés pour déployer les applications sur cet intergiciel. La spécification définit l'ensemble d'interfaces que doivent implanter d'un côté les outils de déploiement et de l'autre, les intergiciels J2EE. Le but est d'assurer une compatibilité entre n'importe quel outil de déploiement J2EE avec n'importe quel serveur d'applications conforme au standard J2EE.

L'interaction entre un outil de déploiement et le serveur est assurée grâce à un composant appellé *DeploymentManager* que chaque serveur J2EE doit fournir. Ce composant permet de déployer des unités de déploiement appelées *modules*. Nous distinguons quatre types de modules.

- Les fichiers EJB-JAR sont des archives contenant les classes des EJBs et leurs descripteurs de déploiement.
- Les fichiers WAR sont des archives contenant les applications Web formées de Servlets/JSPs et de descripteurs de déploiement.
- Les fichiers EAR sont des archives Java pour les applications J2EE contenant les fichiers EJB-JAR, WAR et leurs descripteurs de déploiements.
- Les fichiers RAR qui sont des archives Java contenant les classes de connecteurs J2EE et de descripteurs de déploiement XML.

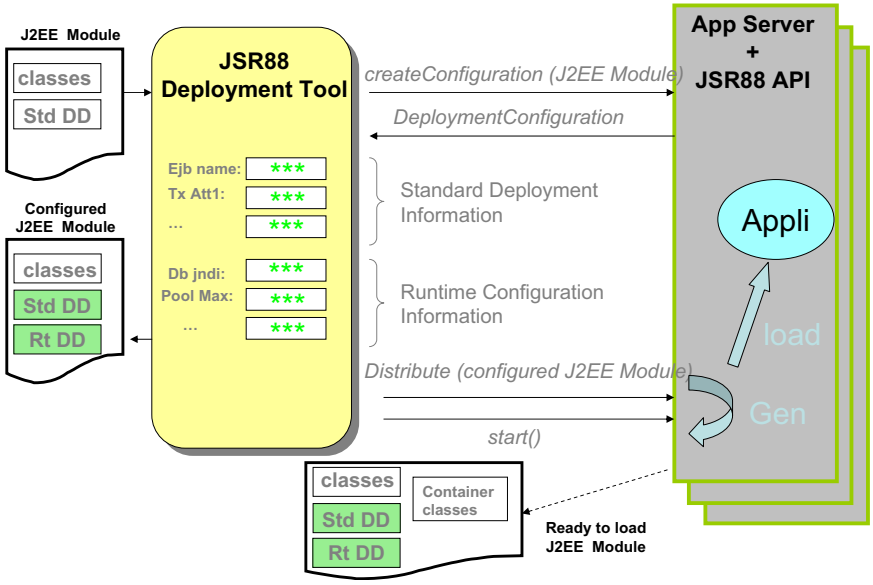

FIG. 4.15 – *Scénario de déploiement d'un module J2EE.*

Le *DeploymentManager* fournit les interfaces nécessaires pour effectuer les deux opérations de déploiement définies par la spécification du JSR88 : la configuration et la distribution. Nous pouvons résumer ces opérations dans les points suivants.

- Configuration de l'application. Le *DeploymentManager* permet de mettre à jour les descripteurs de déploiement avec les informations nécessaires au déploiement d'applications. Une interface graphique est généralement utilisée pour interroger le serveur sur les éléments propriétaires pour remplir les champs correspondants dans le descripteur de déploiement. Cette étape s'appelle, selon la spécification, la « phase de déploiement ». À partir du descripteur de déploiement et des classes initialement développées, d'autres classes d'interposition (*stubs*) spécifiques au conteneur sont générées. L'ensemble des classes et des descripteurs de déploiements forment le module à déployer.
- Distribution sur des cibles. Cette phase consiste à installer le module à déployer sur une ou plusieurs cibles.
- Démarrage/arrêt d'une application.
- Désinstallation de l'application.

Suite aux opérations liées au démarrage, à l'arrêt, à la distribution et à la désinstallation des modules, le *DeploymentManager* retourne un objet appellé *ProgressObject* à l'outil de déploiement. Cet objet permet de tracer et de donner les détails de l'exécution du déploiement qui peut avoir une longue durée de vie.

La figure 4.15 présente un scénario de déploiement d'un module J2EE. L'outil de déploiement, à gauche de la figure, offre une interface graphique pour remplir les champs des descripteurs de déploiement (DD dans la figure) du module à déployer. À droite de la figure est représenté le serveur d'application qui implante l'interface du JSR88 servant à installer le module. Le scénario de déploiement comprend les étapes suivantes:

- L'outil de déploiement prend en entrée un module à déployer contenant initialement les classes des EJBs et le descripteur de déploiement standard (Std DD). Si le module est un WAR, un JAR ou un RAR, l'outil de déploiement crée une instance de

l'objet *J2EEDeployable Object* et si le module est un EAR, il crée plutôt un objet *J2EEApplicationObject*. La phase de configuration consiste à mettre à jour le descripteur de déploiement propriétaire ou *Runtime* (Rt DD).

- L'outil de déploiement utilise une usine *DeploymentFactory* pour créer une instance du *DeploymentManager*. Pour récupérer les informations propres au serveur d'applications, l'outil de déploiement appelle la méthode *createConfiguration* du *DeploymentManager* avec le *J2EEDeployableObject* comme paramètre.
- Le *DeploymentManager* retourne l'objet *DeploymentConfiguration* contenant les informations nécessaires au descripteur de déploiement propriétaire ou (Rt DD).
- L'outil de déploiement offre l'interface à l'administrateur pour appeler la méthode *distribute* du *DeploymentManager* afin d'installer le module sur la machine cible et exécute la méthode *start()*.

Comme c'est décrit dans [Exe04], la spécification JSR88 n'offre pas de solution à un nombre de problèmes que nous listons ici.

D'abord, le modèle de déploiement J2EE n'offre pas le moyen d'exprimer les dépendances entre les modules. Par exemple, il est possible qu'un RAR soit partagé par plusieurs applications. Il est plus logique que ce RAR soit déployé une seule fois au lieu d'être inclus dans le paquetage de chaque EAR de l'application qui l'utilise. Ceci vient du manque d'outils, dans les serveurs d'applications, vérifiant qu'un module est déjà déployé.

De plus, la notion de cible ou *target* dans le modèle de déploiement est trés simpliste. Il est indiqué qu'une cible est soit un serveur soit un ensemble de serveurs ce qui suppose la duplication des modules sur les différentes cibles. Par contre, il n'est pas précisé comment définir les dépendances entre les différents modules distribués et il n'est pas spécifié comment distribuer un même module sur des serveurs différents.

Enfin, il est important de noter que la spécification couvre le déploiement des applications et n'aborde pas le déploiement de l'intergiciel. Le modèle suppose que l'intergiciel (le serveur J2EE) est déjà déployé. En particulier, la spécification ne définit pas les paquetages des services de l'intergiciel et ne décrit pas leur installation sur une même machine virtuelle ou en distribué.

4.3.4 Synthèse

JSR77 et JSR88 sont les deux spécifications principales de l'administration des serveurs J2EE. JSR77 définit un modèle d'informations que doit exposer chaque serveur à l'administrateur. L'implantation de JSR77 repose sur une instrumentation JMX du serveur et à un accès distant via les connecteurs JMX. JSR88 traite le déploiement des applications et spécifie les interfaces nécessaires pour la communication entre les outils de déploiement et les serveurs.

En revanche, les standards ne couvrent pas l'architecture ni le déploiement de l'intergiciel. Ceci est laissé à la charge des fournisseurs des serveurs.

4.4 Administration des serveurs d'applications J2EE « libres »

Nous présentons dans cette partie les approches adoptées par les serveurs d'applications « libres » pour l'administration de l'intergiciel, l'administration des applications étant en général restreinte à l'implantation du standard JSR88. Nous décrivons, pour chaque serveur, son architecture et discutons des outils d'administration utilisés dans les environnements distribués.

4.4.1 JBoss

JBoss [FR03] est basé sur le patron du micro-noyau (*Micro-kernel*) dont le principe est de fournir un noyau minimal servant de bus logiciel permettant d'intégrer des extensions à la demande et d'une manière incrémentale. Ces extensions peuvent être fournies par différents développeurs. Pour des raisons d'interopérabilité, le bus impose des interfaces bien définies aux extensions.

Pour réaliser cette architecture, JBoss repose sur le modèle JMX. Le serveur de MBeans constitue le noyau et les services de l'intergiciel sont intégrés au noyau sous la forme de MBeans. Toutefois, le modèle JMX ne définit pas le cycle de vie pour les MBeans, ni de relations exprimant les dépendances entre les MBeans. Le paquetage et le déploiement ne sont pas non plus couverts par la spécification. Face à ces limitations, le modèle MBeans a été étendu dans JBoss pour donner lieu aux *services MBeans*. Les services MBeans respectent un cycle de vie bien défini et constituent des unités de déploiement. Par ailleurs, JBoss définit les SAR (*Service archives*) qui sont des paquetages pour un ou plusieurs MBeans. Un SAR contient les binaires pour un ou plusieurs services MBeans ainsi qu'un *descripteur de service* qui fournit les informations nécessaires au déploiement.

Cycle de vie des services

Un service peut être dans un état « arrêté » ou « actif ». Pour chaque transition d'état, l'une des opérations suivantes est invoquée.

- `create` est appelé une fois que le MBean associé au service est enregistré au niveau du serveur MBean ; cette opération permet de compléter l'initialisation du MBean et de positionner son état à « arrêté ».
- `start` modifie l'état d'un service de « arrêté » à « actif » ; cette opération permet d'effectuer ce qui est nécessaire pour démarrer convenablement le MBean.
- `stop` modifie l'état d'un service de « actif » à « arrêté » et permet également d'arrêter les parties appropriées du service.
- `destroy` permet de nettoyer toutes les ressources utilisées par le MBean, cette opération n'est appelée que lorsque le MBean est dans l'état « arrêté » et lorsqu'il est désenregistré du serveur de MBeans.

Les opérations gérant le cycle de vie d'un service doivent être exposées via les interfaces d'administration du MBean associé. Le MBean ne doit pas obligatoirement implanter toutes les opérations citées de ce cycle de vie.

Les descripteurs de services

Chaque paquetage de services (chaque SAR) contient un descripteur de déploiement des MBeans sous la forme d'un fichier XML. Pour chaque service, un élément `mbean` de ce fichier permet de spécifier les informations suivantes :

- la classe Java du service et le nom d'objet JMX, c'est à dire la désignation JMX du MBean associé au service,
- les valeurs initiales pour les attributs d'administration,
- les dépendances avec d'autres `mbeans`.

La figure suivante présente un exemple de descripteurs pour un SAR servant à déployer cinq services. Chaque élément `mbean` donne les informations nécessaires au déploiement d'un service. Les attributs `code` et `name`, obligatoires pour les éléments `mbeans` spécifient respectivement la classe qui implante le service et le nom d'object JMX. Par exemple les chaînes de caractères `jboss:service=WebService` et `jboss:service=XidFactory` représentent des noms d'objets JMX.

```
<server>
<!-- Web server for class loading -->
<mbean code="org.jboss.web.WebService"
name="jboss:service=WebService">
<attribute name="Port">8083</attribute>
<attribute name="DownloadServerClasses">true</attribute>
</mbean>
<!-- XID factory -->
<mbean code="org.jboss.tm.XidFactory"
name="jboss:service=XidFactory">
<attribute name="Pad">true</attribute>
</mbean>
<!-- Transaction manager -->
<mbean code="org.jboss.tm.TransactionManagerService"
name="jboss:service=TransactionManager">
<attribute name="TransactionTimeout">300</attribute>
<depends optional-attribute-name="XidFactory">
jboss:service=XidFactory</depends>
</mbean>
<!-- EJB deployer -->
<mbean code="org.jboss.ejb.EJBDeployer"
name="jboss.ejb:service=EJBDeployer">
<attribute name="VerifyDeployments">true</attribute>
<attribute name="ValidateDTDs">false</attribute>
<attribute name="VerifierVerbose">true</attribute>
<depends>jboss:service=TransactionManager</depends>
<depends>jboss:service=WebService</depends>
</mbean>
<!-- RMI/JRMP invoker -->
<mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
name="jboss:service=invoker,type=jrmp">
<attribute name="RMIObjectPort">4444</attribute>
<depends>jboss:service=TransactionManager</depends>
</mbean>
</server>
</body>
</html>
```

FIG. 4.16 – *Exemple de fichier descripteur de services dans JBoss.*

Chaque élément `mbean` peut contenir un ou plusieurs éléments `depends`. Les éléments `depends` sont optionnels et contiennent des chaînes de caractères spécifiant des noms d'objets JMX. Ces noms d'objets correspondent à des MBeans dont dépend le MBean englobant. Dans l'exemple précédent, il y a des dépendances entre le `transaction manager` et le `XID factory` et entre le `EJB deployer` et le `web server`.

Gestion des dépendances

Le déploiement des paquets SARs est géré par le `SARDeployer`. Le `ServiceController` joue le rôle de dépôt de composants services, il contrôle les dépendances entre les MBeans et n'autorise pas les MBeans dont les dépendances ne sont pas satisfaites de passer à l'état actif. Le `SarDeployer` et le `ServiceController` collaborent pour maintenir le cycle de vie des MBeans en respectant le protocole suivant.

- Quand l'opération `create` est appelée sur un MBean à déployer, la même opération est systématiquement invoquée sur tous les autres MBeans dont dépend ce MBean. Les valeurs des attributs sont positionnées à ce stade et le MBean peut vérifier si les autres MBeans dont il dépend sont disponibles.
- L'opération `start` est invoquée sur le MBean à déployer ainsi que sur tous les autres MBeans dont il dépend.
- Quand l'opération `stop` est appelée sur un MBean à déployer, la même opération est systématiquement invoquée sur tous les autres MBeans dont dépend ce MBean.
- Quand l'opération `destroy` est appelée sur un MBean à déployer, la même opération est systématiquement invoquée sur tous les autres MBeans dont dépend ce MBean.

Les relations entre les services sont établies au déploiement et il n'est pas possible de les changer dynamiquement à l'exécution.

Déploiement et chargement de classes

Les unités de déploiement (les modules pour les applications et les archives pour l'intergiciel) sont déposées dans un répertoire bien défini. Une classe, le *DeploymentScanner* contrôle le répertoire de déploiement de la façon suivante. Quand un nouveau fichier est ajouté (respectivement retiré) au répertoire, le *DeploymentScanner* avertit le *MainDeployer* qui appelle alors la méthode `deploy` (respectivement `undeploy`). Une classe est dite *visible* au sein d'un serveur soit parce que la classe est déjà chargée dans la machine virtuelle Java soit parce que un des *class loaders* utilisés par le serveur contient cette classe dans sa liste. Par exemple, une classe dans le `classpath` n'est pas forcément chargée mais, elle est visible.

Suite à une opération `deploy` (ou `undeploy`), la liste des classes visibles est mise à jour. À chaque nouveau fichier installé, un nouveau nom correspondant à sa version est généré ce qui identifie la classe d'une manière unique[4]. JBoss utilise une nouvelle architecture qui facilite le partage des classes entre les unités de déploiement. Les *unified class loaders* sont un ensemble de *class loaders* partageant un même ensemble de classes chargées.

Un ensemble de *unified class loaders* agit comme un unique *class loader* permettant la coexistence de plusieurs versions de classes portant le même nom. Cependant, la mise à jour des classes n'est pas résolue avec cette solution. Par conséquent, la gestion de versions des services n'est pas supportée dans JBoss.

Pour l'administration des environnements distribués, JBoss n'offre pas d'outil particulier pour automatiser le déploiement de l'intergiciel ou des applications. Des scripts ad hoc sont généralement adoptés pour la partie déploiement. L'observation et la reconfiguration ne sont pas traités.

4. rappellons qu'une classe Java est définie d'une manière unique par la combinaison d'un *class loader* et d'un nom de classe

Il existe une extension de JBoss, appellée JAGR [CKZ+03] permettant de gérer les fautes au niveau des applications en redémarrant la partie fautive. Ce travail est très intéressant et a été exploité dans la thèse comme nous le montrerons dans les chapitres sur la contribution. Cependant, le résultat obtenu sur la pertinence du redémarrage partiel ou *microReboot* est restreint aux applications. La faute au niveau de l'intergiciel n'est pas gérée dans JAGR ainsi que tous les problèmes liés au déploiement et à la configuration dans les environnements distribués.

Synthèse

JBoss est un serveur d'applications libre reposant sur une architecture à micro-noyau. Le serveur JMX constitue le cœur du serveur et les services sont ajoutés au serveur sous la forme de MBeans. Cette architecture est relativement modulaire puisque les services sont initialement archivés dans des paquets séparés et peuvent être ajoutés dynamiquement. Cependant, à l'exécution, les dépendances entre les services ne sont pas explicites et il n'est pas possible de modifier la structure du serveur, notamment dans des environnements distribués comme la grappe. De plus, dans ces environnements, l'administration est restreinte à quelques scripts manuellement écrits pour la configuration ad hoc et le déploiement de l'intergiciel.

JAGR est un prototype de recherche qui étend JBoss pour gérer les fautes des applications J2EE. Mais JAGR ne traite pas les fautes de l'intergiciel.

4.4.2 Geronimo

Geronimo [Mul04] présente comme JBoss une structure de micro-noyau. L'unité de configuration est le GBean alors que l'unité de déploiement est appellée *configuration*. Une configuration peut assembler plusieurs GBeans. Un plan de déploiement (*deployment plan*) permet d'exprimer les dépendances entre les services ainsi que d'autres paramètres de configuration. Nous ne détaillons pas ici la syntaxe de ces plans de déploiement qui sont proches des descripteurs de déploiement déjà présentés pour JBoss.

La figure 4.17 représente les étapes de déploiement dans Geronimo. Comme dans JBoss, ces dépendances sont établies au déploiement. Néanmoins, contrairement à JBoss, la communication entre les services ne fait pas intervenir le noyau mais à la place, des *proxy* sont créés. Les *proxy* permettent une dépendance faible entre les services. Par contre, ces liaisons restent locales à une seule machine virtuelle.

Geronimo ne dispose pas d'outil particulier pour gérer le déploiement et la configuration dans des environnements distribués comme la grappe.

4.4.3 JOnAS

Contrairement aux autres serveurs d'applications libres, JOnAS [Pro] intègre un système d'administration permettant le déploiment des applications et la supervision de l'intergiciel à l'exécution. Le coeur de ce système d'administration est basé sur une application Web, appellé **jonasAdmin**, offrant une console graphique à l'administrateur. JonasAdmin permet d'effectuer les opérations suivantes.

- Obtenir une liste des services présents dans le serveur et visualiser leurs paramètres de configuration.
- Déployer ou désinstaller les applications J2EE. Ceci suppose que les modules associés soient déjà présents sur la machine du serveur.

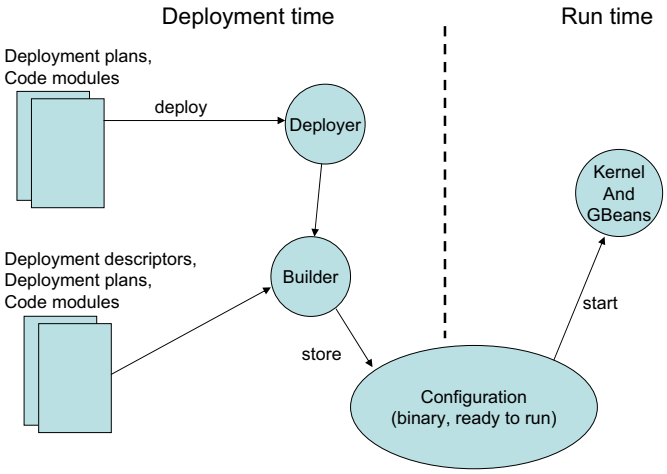

FIG. 4.17 – *Phases de déploiement dans Geronimo.*

– Obtenir des statistiques et des informations liées aux performances. Par exemple, la console de JonasAdmin permet d'afficher la mémoire utilisée par le serveur et le nombre des beans dans les réserves.

Pour réaliser ces propriétés, l'application Web de jonasAdmin communique avec le serveur JMX intégré dans JOnAS grâce au service JMX. Par conséquent, JonasAdmin expose tous les MBeans instrumentant le serveur et les applications au niveau de la console.

La première version du système d'administration se focalise uniquement sur la configuration centralisée de JOnAS. Nous avons contribué à étendre le système d'administration pour gérer des instances de JOnAS regroupées dans des domaines dans des environnements distribués [AD06].

La figure 4.18 présente l'architecture du système d'administration obtenu.

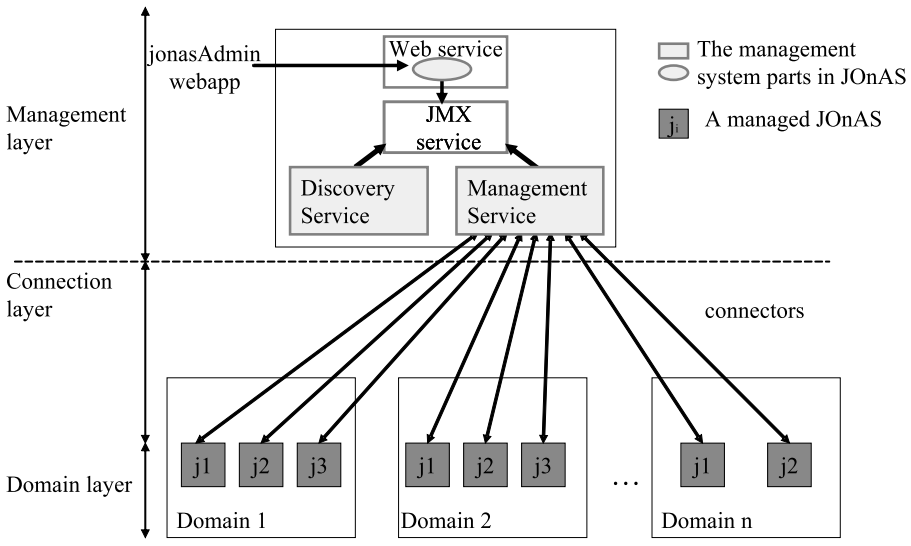

FIG. 4.18 – *Système d'administration de JOnAS.*

Nous distinguons trois couches :

– la couche des domaines ou *domain layer*,

– la couche des connexions ou *connection layer*,
– la couche d'administration ou *management layer*.

Les domaines représentent un ensemble d'instances de JOnAS soumises à une même autorité ou une même politique d'administration. La couche de connexions est formée d'un ensemble de connecteurs JMX reliant la couche d'administration aux MBeans des serveurs JOnAS administrés. Enfin, la couche d'administration est constituée de jonasAdmin, du service de découverte et du service de *management*. Le service de *management* regroupe l'ensemble des programmes recevant les événements d'observation ou envoyant les messages de commande pour modifier les serveurs quand ceci est nécessaire. La console offerte par jonasAdmin est enrichie pour afficher les informations sur les serveurs des différents domaines et pour permettre de modifier leurs paramètres.

Le but du service de découverte ou *service discovery* est de découvrir les instances de JOnAS démarrées dans un réseau local et de les enregistrer au niveau du serveur gestionnaire. L'architecture de ce service est présentée dans 4.19 [5].

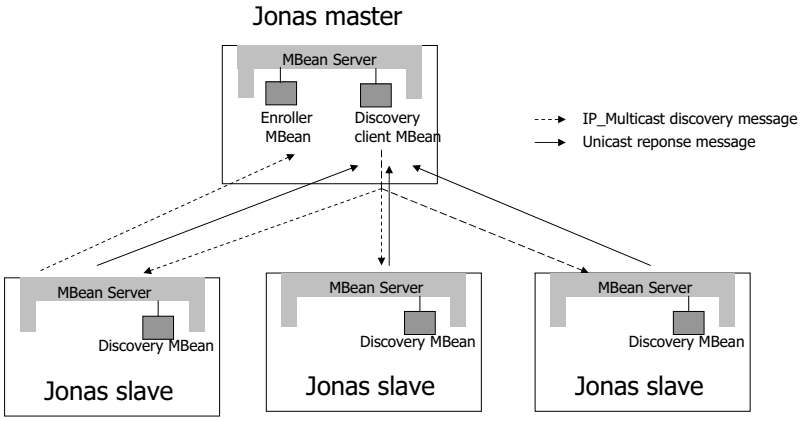

FIG. 4.19 – *Le service de découverte dans JOnAS.*

La solution que nous avons adoptée est basée sur le standard JMX : les éléments de notre service de découverte sont des MBeans qui sont activés lorsque le service démarre. Selon le MBean activé, JOnAS peut se comporter en maître (*master*) ou en esclave (*slave*). Avec la configuration de maître, JOnAS joue le rôle de gestionnaire. Les MBeans activés sont alors le *DiscoveryClient* et *Enroller*. La configuration par défaut est celle de l'esclave, c'est-à-dire sujet à être administré et, dans ce cas, le MBean activé est *Discovery*. JOnAS peut être à la fois maître et esclave, dans un tel cas, il s'administre lui-même. Un maître et ses serveurs esclaves appartiennent à un même groupe IP-multicast formant ainsi un même domaine. Un algorithme simple peut résumer le fonctionnement du service. L'algorithme distingue 2 modes : la découverte serveurs par le maître et la notification aux esclaves.

1. **Découverte des serveurs par le maître** Pour découvrir les instances de JOnAS à administrer, le MBean *DiscoveryClient* du maître envoie une requête de découverte contenant les informations nécessaires (adresse IP, port) pour lui envoyer des messages de réponse. La réponse doit contenir des informations sur l'état du serveur (démarré, en panne ou arrêté), le nom du domaine et la liste des URL correspondant aux connecteurs JMX permettant de se connecter au serveur.

5. Le service discovery est introduit dans JOnAS depuis la version 4.6.

Le *DiscoveryClient* attend les réponses pendant un temps configurable. Du côté des esclaves, les MBeans *Discovery* répondent suivant une communication point à point avec un message contenant les informations attendues.

2. **Notification aux esclaves** Au démarrage ou à l'arrêt d'un serveur JOnAS, le serveur diffuse un message avec IP-multicast en utilisant le MBean *Discovery*. L'adresse multicast est configurable et définit le domaine du serveur. Le MBean envoie dans le message de diffusion les informations nécessaires au maître pour pouvoir se connecter au serveur. Du côté du maître, le MBean *Enroller* attend les messages des serveurs JOnAS et les transfère localement à tous les programmes d'administration dans JOnAS intéressés par cet événement. En particulier, JonasAdmin est averti pour mettre à jour la liste des serveurs dans le domaine.

Une fois que les serveurs sont enregistrés au niveau de jonasAdmin, il est possible d'effectuer le déploiement d'applications sur des domaines (ce qui entraine leur déploiement sur les serveurs faisant partie de ces domaines).

En revanche, l'administration de JOnAS représente encore plusieurs limitations, liées principalement à son architecture. D'abord, à cause de l'architecture complètement cachée à l'exécution, il n'est pas possible d'ajouter ou de changer dynamiquement un service à l'exécution. Ceci est dû également au fait que les services dans JOnAS sont livrés ensemble dans un même paquetage et sont démarrés au démarrage du serveur. Par conséquent, il n'est pas envisageable d'ajouter un service alors que le serveur est déjà démarré. Ensuite, la majorité de la configuration des services est basée sur des fichiers spécifiques et hétérogènes et l'administrateur doit être familier avec la syntaxe de chaque type de fichier de configuration. Enfin, le système d'administration de JOnAS ne gère pas le déploiement des serveurs qui sont supposés être déjà déployés sur les machines cibles. Le déploiement des serveurs est généralement effectué, comme dans la plupart des serveurs J2EE « libres », manuellement ou à l'aide de scripts ad hoc.

4.4.4 Synthèse

Les serveurs d'applications libres (JOnAS, JBoss et Geronimo) reposent, d'une manière ou d'une autre, sur le standard JMX pour l'administration des serveurs et des applications. En effet, JBoss adopte une architecture respectant le patron du micro-noyau où le noyau est le serveur JMX et les services intégrés comme des MBeans. Dans Geronimo, le même patron est utilisé et les services sont des GBeans, une extension des MBeans pour avoir une dépendance moins forte entre les services.

Dans JOnAS, les services sont encapsulés dans des MBeans et un service JMX permet de les exposer aux administrateurs. L'architecture de JOnAS est considérée moins modulaire que celle des autres serveurs d'applications puisqu'il n'est pas possible de déployer dynamiquement les services alors que le serveur est démarré, ni de les archiver dans des paquets séparés. Cependant, JOnAS se distingue des autres serveurs par une console d'administration intégrée et un service de découverte que nous avons contribué à définir et à implanter.

Par ailleurs, les trois serveurs ne disposent pas d'outils pour automatiser le déploiement notamment dans les environnements distribués comme les grappes, ni d'effectuer des opérations de reconfiguration pour la réparation des pannes et l'adapatation des serveurs aux pics de charge. Nous attribuons ces limitations à l'architecture de ces serveurs, qui repose sur JMX et qui n'est pas complètement explicite à l'exécution. Nous invitons le lecteur à revoir notre discussion sur les limitations de l'architecture JMX dans 4.3.1.

Dans les environnements distribués, des fichiers ad hoc et écrits manuellement sont utilisés pour configurer et déployer les serveurs. Ceci peut entrainer des erreurs et nécessite un temps

considérable à l'administrateur. De plus, la reconfiguration n'est pas possible car l'administrateur ne dispose pas de vision explicite de l'architecture globale du système à l'exécution.

L'administration des serveurs J2EE soulève encore plusieurs défis.

4.5 Les défis de l'administration de serveurs J2EE

Les standards J2EE se focalisent particulièrement sur le déploiement des applications. La configuration et le déploiement de l'intergiciel sont laissés à la charge des fournisseurs des serveurs. Nous avons présenté des solutions adoptées dans les serveurs d'applications libres pour administrer les services. Néanmoins, plusieurs problèmes d'administration sont encore ouverts. Nous résumons ces problèmes dans les points suivants.

- **Déploiement** Plusieurs serveurs d'applications offrent la possibilité d'archiver les services dans des paquets indépendants. Ils offrent également des outils de déploiement. Cependant, l'architecture d'un serveur J2EE est généralement implicite et limitée à une seule machine virtuelle Java. Il n'est pas actuellement possible de dupliquer les services soit dans une même machine virtuelle soit sur des nœuds différents sans construire des outils supplémentaires. De plus, dans les environnements distribués, l'unité de déploiement est tout le serveur et non pas le service, ce qui peut engendrer une utilisation non optimale des ressources.
- **Configuration** Dans plusieurs serveurs d'applications J2EE, des outils sont mis en place pour une configuration plus uniforme des services de l'intergiciel. En revanche, il n'existe pas d'outils qui automatisent le déploiement et la configuration dans des environnements distribués comme les grappes.
- **Performance et gestion des fautes** Ces tâches impliquent la reconfiguration d'une partie du serveur pour réagir à des événements extérieurs comme les pannes matérielles ou logicielles, les attaques de sécurité ou les pics de charges. Comme exemple de reconfigurations, nous pouvons citer le redémarrage d'un service ou la migration d'un service d'une machine à une autre peu chargée. Comme nous l'avons évoqué plus haut, ces reconfigurations doivent avoir un impact minimal sur le serveur alors qu'actuellement ils impliquent en général le redémarrage du serveur.

Pour atteindre ces objectifs, nous pensons que l'architecture des serveurs en termes des services qui le composent et de leurs relations doit être exposée d'une manière explicite. De plus, les paramètres de configuration et de déploiement doivent être décrits en utilisant des éléments de l'architecture.

Pour ce faire, l'architecture du système doit reposer sur un modèle à composants. Les différentes parties du logiciel ainsi que les machines sur lesquelles elles se déploient doivent être représentées avec ce même modèle pour plus d'uniformité. Ceci apporte les avantages suivants.

- **Configuration uniforme** La configuration et le déploiement du système administré et du système d'administration sont unifiées en utilisant le même modèle de composants.
- **Automatisation des fonctions d'administration** L'abstraction de l'administration à la manipulation de composants permet de construire facilement des boucles de commande automatisant la configuration, le déploiement et la reconfiguration.
- **Flexibilité** L'architecture explicite des serveurs d'applications permet de séparer les mécanismes des politiques d'administration. Les politiques d'administration peuvent être ajoutées ou changées à l'exécution sous la forme de composants.

4.6 Synthèse

Les serveurs d'applications J2EE sont un assemblage de services offrant des propriétés non fonctionnelles aux applications comme la gestion des transactions, les protocoles de communication, la sécurité, etc. L'architecture est très complexe et elle est répartie en plusieurs étages pour séparer le développement de la partie présentation de la partie métier. Par conséquent, l'administration des serveurs d'application est une tâche difficile. Les standards J2EE couvrent surtout la configuration et le déploiement des applications et n'abordent pas la configuration et le déploiement de l'intergiciel. Ils définissent un modèle d'information que tous les serveurs d'applications doivent exposer grâce à une instrumentation JMX mais ne précisent pas l'architecture de ces serveurs.

Les serveurs d'applications libres (JBoss, Geronimo et JOnAS) implantent les standards d'administration J2EE et offrent des approches propriétaires pour la gestion de l'intergiciel. L'architecture de ces serveurs se base, d'une manière ou d'une autre, sur le modèle JMX. Néanmoins, l'architecture des serveurs ne permet pas d'avoir une vue explicite des dépendances entre les différentes parties et d'effectuer des opérations de reconfiguration comme la gestion des fautes et l'adaptation aux pics de charge. La reconfiguration d'implantation dans les serveurs qui consiste principalement à la mise à jour du code des services, est limitée également à cause de la gestion de chargement de classes actuellement adoptée dans J2EE. De plus, les outils d'administration dans les environnements distribués comme les grappes sont réstreints à l'écriture de scripts ad hoc.

L'administration des serveurs d'applications J2EE pose encore plusieurs défis. Le déploiement et la configuration doivent être automatisés surtout dans des environnements distribués comme les grappes. L'architecture du serveur doit être explicite afin d'exposer les dépendances entre les différentes parties du logiciel en cours d'exécution. De plus, la politique de chargement de classes dans J2EE doit être revue afin de tolérer la mise à jour dynamique des services sans arrêter les serveurs. Enfin, vu le coût exorbitant de l'administration dont souffrent actuellement les industriels, il faut simplifier le rôle des administrateurs et adopter une administration autonome où la gestion des fautes et de la charge est traitée par le système lui-même.

Chapitre 5

Présentation de la contribution

Sommaire

Dans ce chapitre, nous introduisons la contribution de cette thèse. Nous le débutons par une synthèse de l'état de l'art établi dans les chapitres précédents. Nous présentons ensuite les deux grands axes de notre contribution : une démarche pour la construction de *systèmes administrables* et la conception d'un système d'*administration*. Enfin, nous décrivons l'organisation de la suite de ce rapport.

5.1 Synthèse de l'état de l'art

Dans le chapitre 2, nous avons défini les différentes fonctions de l'administration. Nous avons identifié trois fonctions principales : la configuration/déploiement, l'observation et la reconfiguration. D'autres fonctions, orthogonales à celles-ci, font également partie de l'administration : la sécurité, la gestion de ressources, etc. Néanmoins, nous nous intéressons dans cette thèse aux fonctions de bases qui constituent encore un défi pour l'administration des systèmes distribués.

Pour faire face à la complexité des systèmes distribués modernes, les administrateurs proposent de baser les fonctions de l'administration sur la description et l'utilisation des éléments de l'architecture des systèmes administrés. C'est ce qui est appelé *administration fondée sur l'architecture*.

Nous avons structuré l'état de l'art en trois parties : la description de systèmes d'administration fondés sur l'architecture, une étude de modèles de composants permettant la construction de systèmes administrables et la discussion de l'administration des serveurs J2EE. Nous présentons, dans ce qui suit, une synthèse de chaque partie.

5.1.1 Administration fondée sur l'architecture

Nous avons présenté, dans le chapitre 2, des travaux récents portant sur la construction de systèmes d'administration fondés sur l'architecture. Les constats que nous pouvons dresser sont les suivants.

- **Les systèmes administrés sont structurés sous forme de composants**. Les modèles utilisés sont généralement simplistes, ce qui limite les fonctions d'administration qui peuvent être réalisées. Par exemple, dans SmartFrog, le cycle de vie des composants est figé ; de fait, les opérations de reconfigurations se limitent à des transitions d'états du cycle de vie. Dans Nix, les composants représentent un simple répertoire dans lequel sont stockés les binaires et les sources du code à déployer. Les dépendances entre composants ne sont pas prises en compte dans le modèle et sont définies dans des scripts shell complexes. De plus, ces dépendances sont locales, ce qui empêche d'utiliser Nix pour effectuer un déploiement distribué.
- **Les systèmes d'administration ne couvrent qu'un sous-ensemble des fonctions de base de l'administration**. Chaque système est généralement dédié à une fonction de l'administration particulière. Par exemple, Nix et SmartFrog ne traitent que le déploiement. Rainbow se limite à la reconfiguration dynamique à l'exécution et suppose que le système administré est déjà déployé. Dans certains cas, les limitations des systèmes d'administration sont dûes à la simplicité du modèle de composants du système administré. Dans Nix, par exemple, le modèle utilisé ne permet pas d'effectuer des reconfigurations de structure comme déplacer un composant d'une machine à une autre.
- **Les systèmes d'administration ne sont pas *administrables***. Les systèmes d'administration sont des applications distribuées complexes dont il est nécessaire d'automatiser le déploiement et de permettre l'adaptation à l'évolution des systèmes administrés et aux événements extérieurs. Toutefois, les systèmes d'administration actuels négligent, en général, la prise en compte de leur propre administration. Par exemple, SmartFrog ou Nix ne définissent pas le déploiement du système de déploiement lui-même. De même, Rainbow ne permet pas son auto-reconfiguration dynamique. Pour remédier à ces défauts, une possibilité serait d'utiliser le même modèle de composants que les systèmes administrés dans la construction des systèmes d'administration afin de bénéficier des mêmes outils d'administration.

Cette étude a permis d'identifier deux besoins essentiels :

- l'utilisation d'un modèle de composants flexible, facilitant la construction de systèmes administrables ; notons que ce modèle doit être utilisé pour construire les systèmes d'administration eux-mêmes,
- la construction des systèmes d'administrations couvrant toutes les fonctions de l'administration de base et autorisant leur propre administration.

5.1.2 Construction de systèmes administrables

Dans le chapitre 3, nous avons dressé un état de l'art de différents modèles de composants existants ; en particulier, nous avons étudié leur utilisation dans la construction des systèmes administrables.

Nous avons distingué deux catégories de modèles de composants :

- les modèles de composants permettant de faciliter le déploiement et la reconfiguration d'implantation,
- les modèles de composants traitant plutôt les reconfigurations de structure une fois que le système à administrer est en cours d'exécution.

Dans la première catégorie, OSGi est le modèle de composants qui fournit les propriétés d'empaquetage et de déploiement les plus évoluées. D'une part, il est possible d'exprimer les dépendances entre les paquetages (appelés *bundles*). D'autre part, le chargement de classes utilisé dans OSGi permet la mise à jour dynamique du code des composants.

Concernant les modèles dédiés à la construction de systèmes dont la structure peut être dynamiquement reconfigurée, Fractal est le modèle de composants le plus approprié. D'une part, il permet de construire des architectures explicites dans lesquelles il est possible de maîtriser les dépendances (liaisons ou encapsulations) entre les composants pendant une opération de reconfiguration. D'autre part, Fractal se distingue des autres modèles de composants par la propriété de partage qui lui permet de modéliser un grand nombre de systèmes.

Pour couvrir toutes les fonctions de l'administration des systèmes distribués, il est nécessaire de disposer d'un modèle de composants permettant : (1) d'effectuer des opérations de déploiement flexibles permettant la mise à jour dynamique du code des composants et (2) d'obtenir une architecture explicite à l'exécution dont il est possible de changer la structure sans arrêter la totalité du système. Dans le cadre de cette thèse, nous proposons d'étendre le composant Fractal avec des caractéristiques du modèle OSGi afin d'obtenir le modèle de composants répondant aux deux besoins.

5.1.3 Étude de cas : administration des serveurs d'applications J2EE

Pour étudier les problèmes d'administration des systèmes distribués actuels, nous considérons le cas de l'administration des serveurs J2EE. Le chapitre 4 a présenté l'architecture et les solutions adoptées pour l'administration des serveurs d'applications « libres » actuels (JBoss, Geronimo et JOnAS). Le constat que nous pouvons dresser est double :

- l'administration des serveurs J2EE est limitée par l'architecture monolithique des serveurs administrés,
- les outils d'administration utilisés ne suffisent pas pour faire face à la complexité croissante de la gestion des serveurs J2EE, notamment dans les environnements distribués et le coût de l'administration est, par conséquent, de plus en plus exorbitant.

Concernant l'architecture des serveurs J2EE, elle est généralement basée sur JMX et n'offre pas, de fait, une vue explicite à l'exécution des dépendances entre les différentes parties du serveur. Par conséquent, il n'est pas possible d'effectuer des opérations de reconfiguration de structure (pour adapter les serveurs à des événements extérieurs telsque les pics de charges et les fautes). De plus, les mécanismes de chargement de classes adoptés ne permettent pas d'effectuer des reconfigurations d'implantation (mise à jour dynamique de code). Il est donc nécessaire de modifier l'architecture des serveurs afin de la rendre explicite à l'exécution ainsi que le mécanisme de chargement de classes pour permettre la mise à jour dynamique de code.

Concernant les outils d'administration des serveurs, ils sont généralement restreints à quelques scripts ad hoc pour la configuration et le déploiement dans les environnements distribués. Dans des environnements complexes de type grappe, l'administrateur passe un temps important pour gérer manuellement les dépendances de configuration entre les différentes parties. De plus, les moyens de reconfiguration se limitent à la modification de quelques paramètres de configuration, ce qui ne suffit pas pour adapter les serveurs aux événements extérieurs d'une manière efficace. Face au coût exorbitant de l'administration, il est nécessaire d'automatiser le déploiement des serveurs J2EE notamment dans les environnement distribués ainsi que certaines opérations de reconfiguration comme la réparation des pannes.

5.2 Présentation de la contribution

Cette thèse présente deux contributions à la construction de systèmes d'administration. La première contribution est une démarche permettant de rendre administrable un système distribué patrimonial. Cette démarche est illustrée par le travail de réingénierie du serveur d'applications J2EE JOnAS. La seconde contribution est la présentation d'un système d'administration permettant d'automatiser la configuration et le déploiement de systèmes distribués, ainsi que de mettre en œuvre des politiques de reconfiguration de structure et d'implantation. Dans la suite de cette section, nous détaillons ces deux contributions.

5.2.1 Construction de systèmes administrables

Nous proposons, dans cette thèse, une démarche permettant de transformer l'architecture d'un système patrimonial existant en une architecture administrable, c'est-à-dire modulaire et reconfigurable. Le travail de réingénierie doit respecter les contraintes suivantes :

- **Respecter les standards**. Si l'implantation d'un système respecte une spécification, il faut que l'implantation modifiée respecte également cette spécification. Typiquement, lorsque des interfaces standardisées sont présentes dans le logiciel de départ, celles-ci doivent être présentes également dans la version modifiée.
- **Minimiser le surcoût induit par la réingénierie**. Il faut que les modifications effectuées dans le système patrimonial ne dégradent pas les performances par rapport au système initial.

Par ailleurs, il est important d'évaluer le temps nécessaire pour la réingénierie. Lorsque la réingénierie peut être effectuée à différentes granularités, il est important de trouver le bon compromis entre la granularité choisie (qui correspond généralement au niveau de reconfiguration désiré) et le temps qu'elle demande.

Le travail de réingénierie consiste à encapsuler les éléments du système à administrer dans des composants. Suite à l'étude effectuée dans l'état de l'art, nous avons adopté Fractal comme modèle de composants. Pour pallier son absence de système d'empaquetage de composants, nous proposons d'utiliser OSGi [1] qui définit le standard actuel d'empaquetage d'applications Java. Les étapes du travail de réingénierie sont les suivantes :

- **Identification des parties du système et de leurs dépendances**. Cette tâche peut s'avérer complexe lorsque la communication entre deux éléments d'un système n'est pas

1. Nous utilisons OSCAR comme implantation de la spécification OSGi (http://oscar.objectweb.org/).

explicite et ne passe pas par des interfaces clairement identifiées. Une relation de contenance est exprimée sous la forme d'encapsulation de composants Fractal et une relation de dépendance est traduite par des liaisons. Une communication est exposée sous la forme de liaisons Fractal.

- **Identification des paramètres de configuration**. Ces paramètres doivent être exposés au niveau de contrôleurs Fractal.
- **Empaquetage des composants**. Une fois « fractalisé », le code doit être archivé dans des paquetages de sorte que chaque unité de déploiement corresponde à un paquetage. La répartition du code dans des paquetages peut être soumise à plusieurs contraintes. Par exemple, il n'est pas toujours possible de faire coexister deux codes de licences differentes dans un même paquetage. De plus, les dépendances entre les paquetages doivent être clairement identifiées et exprimées d'une maniere explicite.

Dans le cadre de cette thèse, nous validons cette démarche de transformation d'un logiciel patrimonial en présentant JonasALaCarte, une version « fractalisée » du serveur d'applications JOnAS.

5.2.2 Système d'administration

Notre seconde contribution est le développement d'un système d'administration. Celui-ci est basé sur Jade [BBH+05], un canevas d'administration de systèmes autonomes développé au sein du projet Sardes. Nous avons étendu Jade afin qu'il permette de déployer des architectures distribuées et de procéder à la reconfiguration dynamique de leur implantation.

Nous débutons cette section par une brève présentation de Jade. Nous décrivons, ensuite, les extensions effectuées dans Jade. Enfin, nous présentons l'utilisation de Jade pour administrer JonasALaCarte.

Jade : canevas d'administration de systèmes autonomes

Jade fournit des patrons de conception et des outils pour la construction de systèmes autonomes. L'architecture de Jade est modulaire et permet de créer d'une manière systématique des boucles de commande [Oga97]. Ces boucles sont constituées de plusieurs éléments : les *capteurs* collectent des informations du système administré, véhiculées via des canaux de communication à des composants *gestionnaires*. Les gestionnaires implantent les politiques d'administration et décident des actions à effectuer sur le système. Ces actions sont effectués grâce aux composants *actionneurs* colocalisés généralement avec le système administré.

Jade utilise Fractal pour la construction des éléments de ses boucles de commande et suppose que le système administré est lui-même construit avec Fractal. Par ailleurs, Jade a une structure réflexive, c'est-à-dire qu'en cours d'exécution, le système administré avec Jade utilise une représentation de lui-même : le SR (*System Representation*). Le SR sert de méta-niveau et réifie l'architecture logicielle interne du logiciel. Le système administré et le SR sont causalement liés : toute modification de l'un se répercute sur l'autre. La connaissance de cette architecture facilite la mise en œuvre de politiques de reconfigurations; elle permet notamment d'identifier les éléments à reconfigurer et de les introduire en conformité avec le reste de la structure du système administré. Jade offre des outils pour générer le SR à partir de la description du système (via l'ADL) lors du déploiement.

L'utilisation de Jade permet principalement de bénéficier de la génération automatique du SR à partir d'une architecture Fractal et de disposer de mécanismes maintenant la cohérence

entre le SR et le système administré. L'implantation des éléments des boucles de commande est laissée à la charge de l'utilisateur de Jade puisque la sémantique de ces élements est liée à la nature de l'application à administrer et aux politiques d'administration à mettre en place.

Extension de Jade

Nous avons étendu Jade afin d'autoriser le déploiement et la reconfiguration d'implantation des systèmes administrés. Jade, dans sa version initiale, se basait sur l'infrastructure de déploiement Fractal qui suppose que le code des composants est déjà installé sur les machines cibles et qui n'impose pas de format de paquetage. Nous avons étendu le système de déploiement en définissant l'empaquetage des composants et les mécanismes d'installation et de chargement de classes. Nous définissons dans le chapitre 7 les abstractions utilisées pour la construction du système de déploiement et nous montrons une implantation de ces abstractions combinant les infrastructures de déploiement Fractal et OSGi.

Par ailleurs, notons que nous décrivons également une extension du SR, appelée *carte* ou SM (*System Map*). La carte implante un cache permettant d'accéder à certaines informations du système administré. Le cache réduit l'accès au système administré et au réseau. De plus, il est possible de definir des domaines d'administration permettant de regrouper un ensemble de composants sous une même autorité administrative.

Administration de JonasALaCarte

Nous avons utilisé Jade pour administrer JonasALaCarte. Pour illustrer les fonctions d'administration obtenues, nous décrivons les trois scénarios suivants :

- le déploiement automatisé d'une grappe J2EE comme exemple de déploiement distribué,
- la mise à jour dynamique du code d'un service pour illustrer la reconfiguration d'implantation,
- la réparation automatique de pannes pour illustrer la reconfiguration de structure.

5.2.3 Évaluation

Nous avons évalué notre travail suivant des critères qualitatifs et quantitatifs. Concernant les critères qualitatifs, nous montrons que, contrairement aux systèmes étudiés dans l'état de l'art, notre système d'administration est construit avec le même modèle de composants que le système administré. Par conséquent, le système d'administration est à son tour *administrable* :

- les composants de la boucle de commande (capteurs, actionneurs et gestionnaires) sont configurés comme des composants Fractal ;
- ces composants sont empaquetés et déployés avec les mêmes outils que les composants du système administré ;
- il est possible de changer dynamiquement la configuration du système d'administration pour définir de nouvelles politiques, ajouter et mettre à jour à l'exécution des capteurs ou des actionneurs respectant l'évolution du système administré.

L'évaluation quantitative montre que la modification d'un système patrimonial comme JOnAS nécessite un temps de réingénierie raisonnable et ne viole pas les standards. Par ailleurs, nous montrons que les performances du système initial sont conservées. Enfin, notre évaluation montre que JonasALaCarte permet d'obtenir des gains significatifs sur les temps d'administration, notamment dans les environnements distribués de type grappe.

5.3 Organisation des chapitres de la contribution

La suite de ce document s'organise de la façon suivante : le chapitre 6 présente notre approche de la construction de systèmes administrables et explique en détails le travail de réingénierie effectué sur JOnAS pour obtenir JonasALaCarte. Le chapitre 7 décrit le canevas Jade, notre extension de ce canevas, notamment le processus de déploiement mis en place utilisant Fractal et OSGi. Nous présentons également dans ce chapitre l'utilisation du système d'administration pour configurer, déployer et adapter JonasALaCarte. Enfin, le chapitre 8 présente l'évaluation de notre travail.

Chapitre 6

Construction de systèmes administrables : cas des serveurs d'applications patrimoniaux

Sommaire

Dans le chapitre 4, nous avons vu que l'administration des serveurs J2EE est limitée par leur architecture monolithique qui ne permet pas d'effectuer des opérations de mise à jour du code dynamiquement, ni d'adapter les serveurs aux modifications de leur environnement. Dans ce chapitre, nous présentons le travail de réingénierie effectué dans JOnAS afin de rendre son architecture explicite. Nous commençons par présenter l'architecture de JOnAS. Nous présentons ensuite JonasALaCarte[1], la réingénierie de JOnAS. Enfin, nous montrons comment l'approche que nous proposons peut être généralisée afin de s'appliquer à d'autres intergiciels.

6.1 Architecture de JOnAS

L'architecture de JOnAS est présentée sur la figure 6.1. Nous distinguons deux types d'entités : les *services* (par exemple le service JMX, le service Web, le service EJB, etc.) et les entités responsables de leur *gestion* et de leur configuration. Il existe trois entités de gestion :

– le **Service manager** gère le cycle de vie des services,

1. Nous ne présentons dans ce chapitre que la partie intergicielle de JonasALaCarte qui est en fait la partie de JOnAS « fractalisé ». En effet, JonasALaCarte inclut également un système d'administration que nous introduisons dans le chapitre 7.

– le **Loader manager** est responsable du chargement de classes des services dans JOnAS,
– le **Configuration Manager** configure les services à partir de fichiers de configuration installés dans un répertoire locale au serveur.

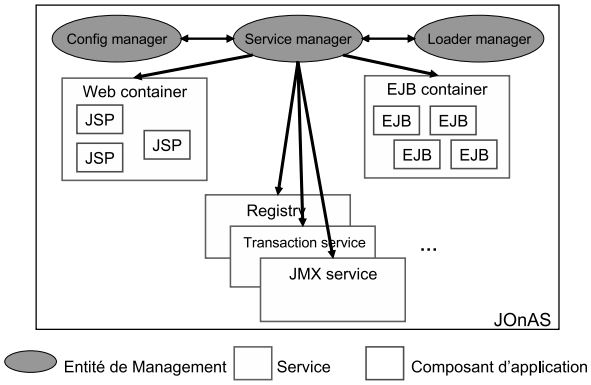

FIG. 6.1 – *Architecture du serveur JOnAS.*

Pour accéder aux autres services, chaque service effectue un appel statique à la méthode `ServiceManager.getInstance().getService(X)`. Les services communiquent alors en appellant des méthodes sur les références obtenues. Nous notons deux inconvénients à cette implantation.

– Les communications entre les services ne passent pas par des interfaces bien définies et il n'est donc pas possible de changer dynamiquement les dépendances entre ces services ou de les contrôler pour effectuer des opérations de reconfiguration.
– L'architecture de JOnAS suppose que tous les services sont localisés dans la même JVM. Pour démarrer un service dans une autre JVM (ou sur un autre nœud), une nouvelle instance de JOnAS contenant ce service doit être déployée. C'est le cas par exemple dans les grappes J2EE, ce qui n'est pas généralement optimal.

6.2 Réingénierie de JOnAS

6.2.1 Identification des parties à modifier dans JOnAS

Nous proposons de considérer les services comme des unités de configuration, de déploiement et de reconfiguration. Ce choix est motivé par les raisons suivantes :

– Les services encapsulent du code venant généralement de contributeurs différents. Le code de chaque service doit, de fait, être empaqueté et mis à jour dynamiquement de façon indépendante.
– Les services doivent être déployés à la demande suivant le contenu des modules applicatifs déployés dans le serveur. Par exemple, il est inutile de déployer le service Web si les modules déployés dans le serveur sont uniquement des JARs (contenant des EJB).
– Certains services (encapsulant typiquement les conteneurs EJB et Web) consomment plus de ressources que les autres services. Il est souhaitable de pouvoir dupliquer indépendemment chaque service afin de permettre à certains services d'avoir plus de duplicas que d'autres.

Afin que les services soient des unités de configuration et de déploiement séparées, il est nécessaire que ces derniers communiquent en utilisant des interfaces bien définies. Il est également nécessaire que leurs liaisons soient explicites à l'exécution. Par ailleurs, il est nécessaire d'archiver chaque service dans un paquetage indépendant. Enfin, le code de chaque service doit être chargé par un *class loader* indépendant. Ceci permet la mise à jour et le changement de versions des services. En effet, mettre à jour le code d'un service suppose la destruction du *class loader* associé et son remplacement par un autre. Si ce *class loader* est responsable du chargement du code d'autres services, ce code va être inutilement rechargé et les autres services inutilement arrêtés.

Par ailleurs, pour simplifier la configuration des services, il faut disposer d'interfaces uniformes et communes permettant de positionner ou de récupérer les paramètres de configuration. De plus, les fichiers de configuration doivent avoir la même syntaxe pour tous les services afin de masquer leur hétérogénéité.

Pour répondre à ces besoins, nous avons identifié deux phases dans le travail de réingénierie de JOnAS :

- la phase de « fractalisation » des services pour disposer d'une architecture explicite à l'exécution,
- la phase d'empaquetage et de chargement de classes pour permettre le déploiement et la mise à jour de code.

6.2.2 « Fractalisation » de JOnAS

Cette section présente JonasALaCarte, une version « fractalisée » de JOnAS. La « fractalisation » de JOnAS consiste à encapsuler les services dans des composants Fractal et à exposer toutes les dépendances sous forme de liaisons Fractal.

La figure 6.2 représente une version simplifiée[2] de l'architecture de JonasALaCarte. JonasALaCarte est un composite Fractal contenant des services et des entités d'administration sous la forme de sous-composants. Nous présentons les différents types de composants ainsi que la nature de leurs liaisons.

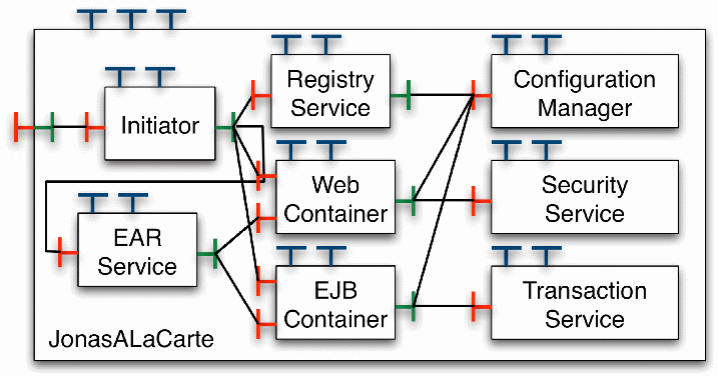

FIG. 6.2 – *Architecture Fractal de JonasALaCarte.*

Nous identifions deux types de composants : les *services* et des *composants de gestion* du serveur. Le composant *configuration manager* est un exemple de composant de gestion : il fournit une configuration par défaut aux services auxquels il est relié. Notons que ce composant peut être localisé dans la même machine que le composant à configurer ou sur une machine distante.

2. Pour plus de clarté, nous ne représentons sur la figure qu'une partie des services.

Un autre exemple de composant de gestion est le composant *initiator* qui permet de démarrer les services dans un ordre défini (e.g. le service *Security* avant le service Web). Parmi les composants implantant les services de JOnAS, citons le service EAR qui encapsule les applications J2EE. Ce composant est lié à un ensemble de composants conteneurs (Web et EJB) dans lesquels il déploie des applications à exécuter.

Les liaisons dans JonasALaCarte peuvent être locales ou distantes. Par conséquent, JonasALaCarte peut être distribué, i.e. les services peuvent s'exécuter sur des machines différentes. Néanmoins, il est important de noter que certains services doivent être co-localisés du fait qu'ils échangent des données non sérialisables. C'est le cas, par exemple, des services Web et *Security*. Il est également indispensable que le service JMX (non représenté dans la figure) soit installé dans toutes les JVM exécutant les services de JOnAS. En effet, le serveur de MBeans doit être dans la même JVM que le code à instrumenter (les MBeans).

JonasALaCarte peut être décrit à l'aide de l'ADL de Fractal. La figure 6.3 présente un extrait de sa description ADL. Dans ce fichier, nous précisons les différents composants, leurs liaisons, ainsi que les machines cibles sur lesquelles sont déployés les composants. Par exemple, le service EJB, les composants *configurator* et *initiator* doivent être déployés sur le nœud1. Les nœud x représentent des nœuds virtuels, c'est-à-dire des composants Fractal dont la configuration est dirigée par l'ADL[3]. Dans notre cas, un nœud virtuel correspond à une machine virtuelle Java.

6.2.3 Empaquetage et chargement des classes des services

L'empaquetage et le chargement de classes des services doit permettre (1) le déploiement des services d'une manière indépendante et (2) la mise à jour dynamique de leur code. Pour ce faire, nous avons besoin que chaque service soit archivé dans son propre paquetage et que chaque paquetage possède son propre *class loader*[4]. En effet, lorsqu'un paquetage a besoin d'être changé, son *classloader* est détruit et un autre *classloader* est créé pour charger le nouveau code.

Concernant le paquetage, un code appartenant à un paquetage peut importer le code d'un autre paquetage. Si le code importé est modifié (mis à jour), il est nécessaire que le code qui l'importe soit mis à jour également. Pour éviter cette contrainte, nous avons choisi que les interfaces permettant la communication entre les services soient archivées dans un paquetage séparé. Ceci évite que la mise à jour d'un service entraine la mise à jour des services dont il exporte du code. Nous nous permettons de faire ce choix car nous considérons que le code des interfaces entre les services est relativement stable, contrairement au code des services qui change plus fréquemment.

Outre le code des services et ceux de leurs interfaces communes, nous avons besoin d'archiver le code des librairies partagées (ASM, logging, etc.).

Nous identifions donc 3 types de paquetages :

- les paquetages contenant le code des services,
- les paquetages contenant le code des librairies utilisées par les services
- les paquetages contenant les interfaces servant à la communication entre les services.

Nous avons effectué deux expériences autour de l'empaquetage et du chargement de classes. Dans la première, nous avons adopté OSGi et dans la deuxième, nous utilisons le même modèle de composants Fractal pour les paquetages et les composants à l'exécution. Nous présentons ces deux approches dans les sections suivantes et nous les comparons.

3. Le déploiement distribué est expliqué en détails dans le prochain chapitre.
4. Cette partie du travail a été effectuée en collaboration avec Jakub Kornas de l'équipe Sardes.

```
<definition name="org.objectweb.jonasALaCarte.Server">
   <component name="ejb" definition="org.objectweb.jonasALaCarte.EJB">
      <virtual-node name="node1" />
   </component>
   <component name="web" definition="org.objectweb.jonasALaCarte.Web">
      <virtual-node name="node2" />
   </component>
   <component name="ear" definition="org.objectweb.jonasALaCarte.EAR">
      <virtual-node name="node2" />
   </component>
   <component name="security" definition="org.objectweb.jonasALaCarte.Security">
      <virtual-node name="node2" />
   </component>
   <component name="transaction" definition="org.objectweb.jonasALaCarte.JTM">
      <virtual-node name="node3" />
   </component>
   <component name="initiator" definition="org.objectweb.jonasALaCarte.Initiator">
      <virtual-node name="node1" />
   </component>
   <component name="registry" definition="org.objectweb.jonasALaCarte.Registry">
      <virtual-node name="node3" />
   </component>
   <component name="configurator" definition="org.objectweb.jonasALaCarte.Configurator">
      <virtual-node name="node1" />
   </component>

   <!-- List of bindings -->
   <binding client="initiator.service-web" server="web.service" />
   <binding client="initiator.service-ejb" server="ejb.service" />
   <binding client="initiator.service-tm" server="transaction.service" />
   <binding client="initiator.service-reg" server="registry.service"/>
   <binding client="initiator.service-sec" server="security.service" />
   <binding client="initiator.service-ear" server="EAR.service" />
   <binding client="web.confItf" server="configurator.confItf" />
   <binding client="initiator.confItf" server="configurator.confItf" />
   <binding client="security.confItf" server="configurator.confItf" />
   <binding client="registry.confItf" server="configurator.confItf" />
   <binding client="ear.confItf" server="configurator.confItf" />
   <binding client="transaction.confItf" server="configurator.confItf" />
   <binding client="ejb.transactItf" server="transaction.transactItf" />
   <binding client="ejb.tm" server="jtm.tm" />
   <binding client="web.sec" server="security.sec" />
   <binding client="ear.deploy-web server="web.deploy" />
   <binding client="ear.deploy-ejb" server="ejb.deploy" />
</definition>
```

FIG. 6.3 – *Extrait d'un fichier ADL de JonasALaCarte.*

Solution basée sur OSGi

Comme nous l'avons présenté dans l'état de l'art, les paquetages OSGi sont appelés *bundles*. Chaque *bundle* est un fichier JAR contenant un code binaire, ainsi que des fichiers *manifest* décrivant la dépendance entre les *bundles* sous la forme d'*import/export*.

Dans le cadre de JonasALaCarte, chaque service est archivé dans un *bundle* séparé. Par ailleurs, les interfaces de communication entre les services sont également archivées dans des *bundles* séparés, ainsi que les librairies partagées par les services. La figure 6.4 donne un exemple de *manifest* correspondant au service Web. Nous précisons dans ce fichier les *paquetages importés* (`org.objectweb.jonas_lib.security.jacc` dans ce cas) et la liste des paquetages contenus dans le *bundle*.

```
Manifest-Version: 1.0.2
Bundle-Activator: fr.test.catalina.Activator
DynamicImport-Package: *
Import-Package: org.objectweb.jonas_lib.security.jacc,
 javax.servlet.jsp,
 javax.servlet.jsp.el,
 javax.servlet.jsp.resources,
 javax.servlet.jsp.tagext,
 javax.servlet,
 javax.servlet.http,
 javax.servlet.resources
Bundle-Name: Catalina Bundle
Bundle-Description: catalina module
Bundle-Manifestversion: 2
Bundle-ClassPath: .,
 etc/,
 /catalina/server/lib/servlets-invoker.jar,
 /catalina/server/lib/catalina-cluster.jar,
 /catalina/server/lib/tomcat-coyote.jar,
 /catalina/server/lib/catalina.jar,
 /catalina/server/lib/catalina-storeconfig.jar,
 /catalina/server/lib/commons-modeler.jar,
 /catalina/server/lib/tomcat-ajp.jar,
 /catalina/server/lib/catalina-ant.jar,
 /catalina/server/lib/catalina-optional.jar,
 /catalina/server/lib/tomcat-http.jar,
 /catalina/server/lib/servlets-webdav.jar,
 /catalina/server/lib/tomcat-util.jar,
 /catalina/server/lib/servlets-default.jar,
 /catalina/server/lib/ow_jonas_tomcat55.jar
Bundle-Version: 1.0.0
```

FIG. 6.4 – *Extrait d'un fichier Manifest.*

Dans la version actuelle de JonasALaCarte, nous n'avons archivé sous la forme de *bundles* qu'une partie des services. Le reste est regroupé dans un même *bundle*. De même, nous avons regroupé toutes les librairies communes aux services (logging, ASM, Utils, etc.). Pour une gestion plus fine du code, ces librairies doivent être réparties sur des *bundles* différents.

Solution basée sur Fractal

L'intérêt de cette approche (décrite dans [AKS05]) est de disposer d'une gestion uniforme des composants de JonasALaCarte dans au long de leur cycle de vie : des paquetages à l'exécution. Nous avons choisi de considérer comme dépendances entre les paquetages les dépendances qui existent entre les services à l'exécution. Nous avons donc fait le choix d'exprimer ces dépendances

dans les mêmes fichiers ADL servant à décrire l'architecture de JonasALaCarte à l'exécution. Pour ce faire, nous avons étendu l'ADL avec des informations indiquant l'identificateur du paquetage de chaque service, ainsi que sa version.

La figure 6.5 représente l'architecture de notre proposition et montre que l'on peut utiliser les mêmes outils pour gérer les services dans leur formats paquetages et à l'exécution. Nous distinguons trois couches :

- **la couche des paquetages** contenant les paquetages à déployer et exprimant les dépendances entre eux,
- **la couche de déploiement** formée par les composants responsables du processus de déploiement,
- **la couche intergicielle** représentant les composants Fractal de JonasALaCarte à l'exécution.

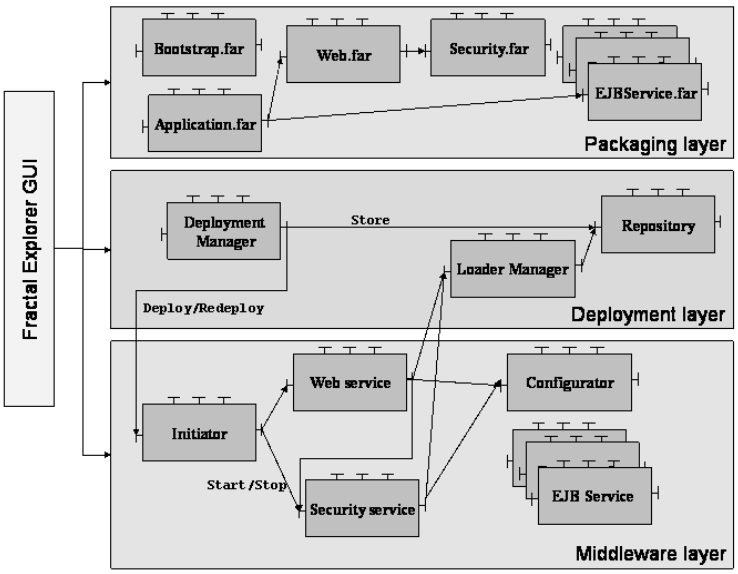

FIG. 6.5 – *Empaquetage basé sur Fractal.*

Dans la couche de déploiement, le composant *Deployment Manager* est responsable de l'analyse du contenu des paquetages, de la vérification de la compatibilité entre leurs versions et de l'identification des liaisons entre eux. Lorsqu'aucun problème n'est détecté (paquetages non disponibles, incompatibilité entre les versions des paquetages, etc), le *Deployment Manager* extrait les contenus des paquetages et demande au composant *Repository* de les stocker localement. Suivant l'implantation du composant *Repository*, plusieurs supports de stockage sont utilisables (fichiers systèmes, base de données, etc.). Le *deployment manager* demande au composant *Loader Manager* de charger le code. Enfin, le *Loader Manager* appelle la méthode `start` sur le composant *initiator* permettant de démarrer tous les services de JonasALaCarte.

Chaque paquetage Fractal ou FAR (*Fractal Archive*) contient les éléments suivants :

- un ou plusieurs fichiers ADL étendus pour décrire l'empaquetage et les dépendances entre les paquetages,
- le code des composants sous forme d'archives Java classiques (JAR).

Dans l'exemple de la figure 6.6, nous montrons un extrait du paquetage `WebContainer-1.0.far`. Les dépendances entre le paquetage du service Web et les autres paquetages exprimés dans le fichier ADL correspondant aux dépendances entre les composants à l'exécution. Par exemple, le paquetage WebContainer-1.0.far dépend de tout paquetage fournissant l'interface `JmxServiceItf` et de tout paquetage fournissant l'interface `SecurityServiceItf`, etc.

```
package WebContainer-1.0.far
WebContainer.fractal:
   <definition name="org.objectweb.jonasALaCarte.WebContainer" version="1.0">
   <interface name="jmx" role="client" signature="org.objectweb.jonas.jmx.JmxServiceItf"
         compatibility="[1.0, *]"/>
   <interface name="security" role="client"
         signature="org.objectweb.jonas.security.SecurityServiceItf" compatibility="[1.0, *]"/>
   <interface name="jprop" role="client"
         signature="org.objectweb.jonasALaCarte.configurator.JProperty" compatibility="[1.0, *]"/>
   <interface name="lmgr" role="client"
         signature="org.objectweb.jonasALaCarte.loaderManager.LManager" compatibility="[1.0, *]"/>
   <interface name="service" role="server" signature="org.objectweb.jonas.service.ServiceItf"/>
   <content
      class="org.objectweb.jonas.web.wrapper.catalina55.WebContainerWrapper"/>
   ... the rest of the ADL file
</definition>

catalina.jar
tomcat-coyote.jar
... other, non ADL files ...
```

FIG. 6.6 – *Dépendances entre les archives Fractal dans JonasALaCarte.*

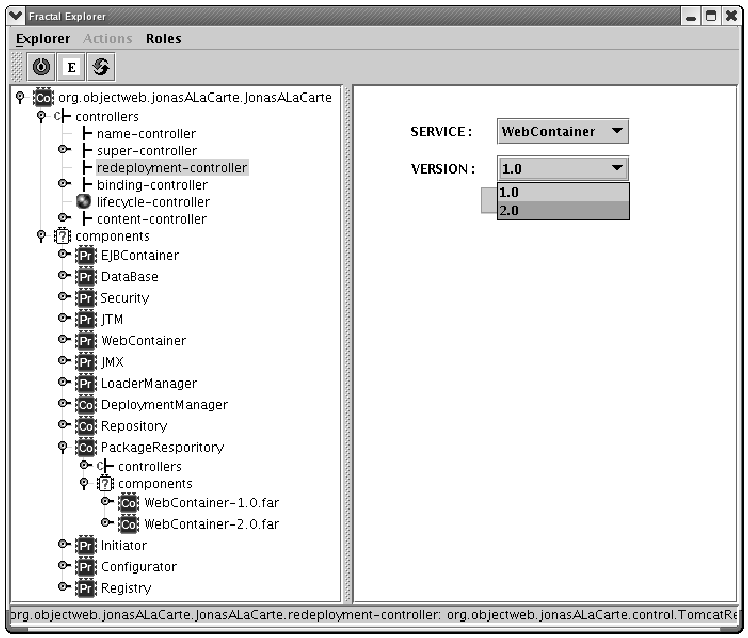

FIG. 6.7 – *Navigation dans les paquetages et les composants avec Fractal Explorer.*

L'adoption de Fractal de la phase d'empaquetage à la phase d'exécution permet d'observer et de reconfigurer le système administré dans toutes les étapes de son cycle de vie en utilisant

les mêmes outils. En effet, nous avons enrichi le navigateur de composants Fractal, *Fractal explorer* [frab] de sorte à offrir les mêmes interfaces graphiques pour les paquetages et les composants à l'exécution. La figure 6.7 montre une navigation dans les différentes versions des paquetages disponibles d'un service ainsi que les composants Fractal à l'exécution.

Comparaison des deux propositions Par rapport à Fractal, OSGi a l'avantage d'offrir un format standardisé des paquetages. Ceci est important pour un logiciel « libre » où plusieurs contributeurs participent au code.

L'avantage de la solution Fractal est la gestion uniforme des composants, de l'empaquetage à l'exécution, ce qui permet d'utiliser des outils d'administration communs.

6.3 Mise en œuvre de la « fractalisation »

Dans cette section, nous décrivons l'implantation de la « fractalisation » de JOnAS.

Dans JOnAS, chaque service étend une classe abstraite `AbsServiceImpl` implantant les interfaces nécessaires pour démarrer, arrêter et configurer un service.

Dans JonasALaCarte, un service doit étendre une classe `AbsServiceWrapper` qui exécute un service JOnAS dans un composant Fractal primitif. Cette classe implante quatre types d'interfaces :

- le `LifeCycleController` offrant à un service un cycle de vie indépendant respectant la spécification du JSR77,
- le `ServiceConfController` qui étend l'interface Fractal `AttributeController` et qui sert 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,
- des interfaces fonctionnelles dépendant du service encapsulé.

Nous appellons adaptateur (*wrapper*) toute classe étendant `AbsServiceWrapper`. Un *wrapper* permet de lancer la classe implantant le service JOnAS. Cette classe est modifiée de telle sorte que tous les appels statiques soient remplacés par des appels sur les références passées par l'adaptateur.

La figure 6.8 illustre la transformation que nous avons effectuée au niveau de l'implantation d'un service (le service JOnAS est représenté à gauche et le service adapté est représenté à droite).

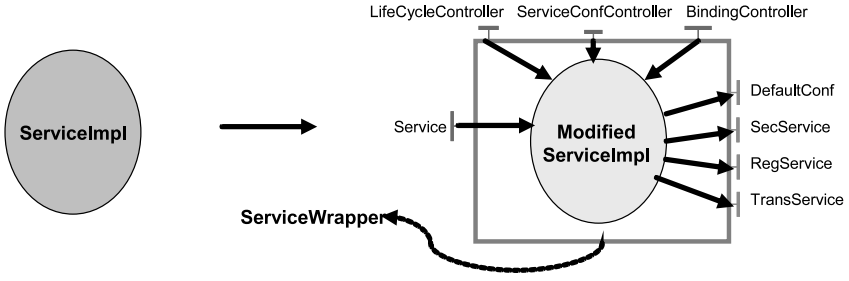

FIG. 6.8 – *Réingénierie d'un service JOnAS : passage de ServiceImpl à ModifiedServiceImpl.*

Exemple du service EJB

Pour illustrer ces transformations, considérons l'exemple du service EJB dont le *wrapper* est représenté dans la figure 6.9.

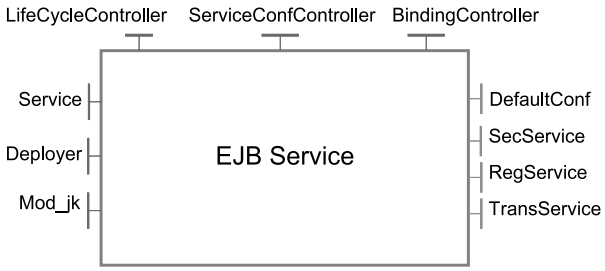

FIG. 6.9 – *Exemple du service EJB.*

Outre les interfaces de contrôle décrites ci-dessus, nous avons implanté les interfaces fonctionnelles requises et fournies spécifiques au service.

- L'interface `Service`, implantée par tous les services, permet de démarrer ou d'arrêter un service à partir d'un composant extérieur (que nous avons appelé `Initiator` dont le rôle est de démarrer les services dans un ordre bien défini).
- L'interface `Deployer` permet de déployer des modules d'applications (les JAR).
- L'interface Mod_jk est utilisée pour relier le conteneur EJB au serveur Web Apache via le connecteur Mod_jk. Nous décrirons plus en détails ce type de connexion dans la partie 7.4.1 décrivant la configuration et l'architecture du serveur en grappes.
- Un ensemble d'interfaces requises sont implantées : `SecService`, `RegService` et `TransService` permettant au service EJB de communiquer respectivement avec les services Security, Registry et Transaction. L'interface `DefaultConf` permet de lier le service à un composant `Configurator` qui lui offre une configuration par défaut.

Configuration du service EJB La figure 6.10 présente un exemple de fichier ADL permettant de configurer le service EJB. Nous identifions 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 client et serveur permettant de connecter le service à d'autres composants,
- le marqueur `virtual node` qui permet de spécifier la désignation de la machine virtuelle sur laquelle le service doit être déployé.

Modifications du service EJB Dans la suite, nous montrons à l'aide d'extraits de code l'encapsulation du code du service EJB dans un composant Fractal.

La figure 6.11 montre l'implantation du service EJB dans JOnAS avant sa modification. Notons que l'accès aux classes de gestion, le `LoaderManager` responsable du chargement de classes des services et `JProp` responsable de la configuration est fait d'une manière statique. De plus l'accès aux autres services est fait en récupérant leurs références du `ServiceManager`. Les dépendances entre les services sont donc cachées dans le code et il n'est pas possible de les changer dynamiquement à l'exécution.

```
<?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="secItf" role="client"
               signature="org.objectweb.jonasALaCarte.SecItf"/>
    <interface name="transactionItf" role="client"
               signature="org.objectweb.jonasALaCarte.TransactItf"/>
    <interface name="registryItf" role="client"
               signature="org.objectweb.jonasALaCarte.RegistryItf"/>
   <interface name="service" role="server"
               signature="org.objectweb.jonasALaCarte.ServiceItf"/>
   <interface name="deployer" role="server"
               signature="org.objectweb.jonasALaCarte.DeployerItf"/>
   <interface name="mod_jk" role="server"
               signature="org.objectweb.jonasALaCarte.ModjkItf"/>
    ...
    <virtual-node name="EJBNode"/>
</definition>
```

FIG. 6.10 – *Extrait du fichier ADL du service EJB.*

```
public class EJBServiceImpl extends ServiceImpl implements EJBServiceMBean, ...
{
   // Access to the classloader
   LoaderManager lm = LoaderManager.getInstance();
   appsClassLoader = lm.getAppsLoader();

   // Access to the configuration parameters
   String JONAS_BASE = JProp.getJonasBase();
   ...
   // Access to services
   ...
   serviceManager = ServiceManager.getInstance();
   ...
   securityService = (SecurityService)
   serviceManager.getSecurityService();
   registryService = (RegistryService)
   serviceManager.getRegistryService();
   jtmService = (TransactionService)
      serviceManager.getTransactionService();
   // Some functional code
      ...
   public void start();
      ...
}
```

FIG. 6.11 – *Extrait de l'implantation du service EJB dans JOnAS.*

Nous avons changé ce code de la manière suivante. D'abord, nous avons implanté la classe `AbsService` que tous les *wrappers* des services doivent étendre. Un extrait de code de cette classe est représenté dans la figure 6.12.

Cette classe implante les interfaces suivantes :

- `ServiceItf` qui étend l'interface LifeCycleController dans l'implantation par défaut de Julia pour couvrir tous les états du cycle de vie défini dans le standard JSR77,
- `ServiceConf` qui étend `AttributeController` permettant de positionner les paramètres de configuration du service,
- `BindingController` permettant de définir les interfaces requises connectant le service aux autres composants du serveur.

Nous montrons dans la figure 6.13 un extrait de la classe `EJBServiceWrapper`. Cette classe étend la classe abstraite `AbsService` et implante un ensemble d'interfaces fonctionnelles comme l'interface `Deployer` dans le cas du service EJB.

Notez que cette classe implante `EJBServiceMBean` comme c'est le cas dans la classe `EJBServiceImpl` de JOnAS. Ceci permet d'instrumenter le service utilisant les MBeans JMX et donc de respecter le standard JSR77.

6.4 Généralisation de notre approche

Le travail de réingénierie illustré dans ce chapitre est en fait généralisable aux autres serveurs d'applications « libres » ainsi que d'autres types d'intergiciels.

Dans le cas de JBoss ou de Geronimo, il suffit de considérer respectivement les MBeans et les GBeans associés aux services et d'effectuer le même travail que nous avons fait sur les services de JOnAS. De la même façon que dans JOnAS, il faut d'abord identifier les interfaces de liaison entre les services et utiliser ces interfaces dans l'établissement des liaisons de communication. La deuxième étape consiste à écrire les fichiers ADL pour la configuration de chaque service et pour l'architecture du serveur.

Pour la phase d'empaquetage, chaque serveur offre en fait un format propriétaire de paquetages pour les services. Le passage de ce format à celui des *bundles* OSGi par exemple suppose l'écriture des fichiers de méta-données appropriés.

Plus généralement, ce travail est également applicable aux intergiciels composés de parties similaires aux services JOnAS correspondant à des entités fonctionnelles séparées. Ces parties doivent être encapsulées dans des composants Fractal et leur configuration doit s'effectuer via les contrôleurs Fractal. L'architecture de l'intergiciel modifié est décrite dans un ou plusieurs fichiers ADL ou en se liant (au sens Fractal) à un composant de configuration par défaut. Les dépendances entre les parties doivent s'exprimer en termes d'encapsulation ou de liaison Fractal afin d'obtenir une architecture explicite du système : une relation d'inclusion est exprimée sous la forme d'encapsulation et une relation de dépendance est traduite par des liaisons Fractal. Une liaison peut être implantée sous forme de composants Fractal implantant le protocole de communication initial.

Une fois « fractalisé », le code doit être archivé dans des paquetages séparés. Il faut définir comment répartir ce code dans les différents paquetages de sorte que chaque unité de déploiement corresponde à un ou plusieurs paquetages. Les dépendances entre les paquetages doivent être également identifiées et exprimées. Dans le cas de OSGi, ces dépendances sont exprimées dans des fichiers *manifest*.

Enfin, des interfaces du code d'origine peuvent coexister avec les interfaces Fractal. Ceci est utile pour préserver l'implantation de standards par exemple. En effet, nous avons conservé dans

```
public abstract class AbsService implements ServiceItf, ServiceConfController,
      BindingController {
      // ServiceConf extends AttributeController for configuration setup and LifeCycle Controller
      public void start() throws ServiceException {
                  ...
      }
      public void stop() throws ServiceException {
                  ...
       }
      public boolean isStarted() {
         return this.started;
      }
      public String getServiceName() {
         return this.name;
      }
      public void setName(String name) {
         this.name = name;
      }
      public String getDomainName() {
         return this.domainName;
      }
      public String getJonasServerName() {
         return this.jonasServerName;
      }
      // ServiceConf implementation
     public void setConfUrls(String urls) {
         // some code
                  ....
      }
      public String getConf() {
          return conf;
      }
      public void setConf(String conf) {
         this.conf = conf;
      }
      public String getConfUrls() {
         ...
      }
      public void setAttribute(String name, String value){
         ...
      }
      public String getAttribute(String name){
        ...
      }

      // BindingController interface implementation
      public String[] listFc() {
          return new String[] { "transactionItf", "secItf", "registryItf",
                     "EJBItf", "WebItf", "confItf", "component" };
      }
      public Object lookupFc(String itfName) throws NoSuchInterfaceException {
         if (itfName.equals("confItf"))
             return conf;
         if ... // test the other binding interfaces
   return null;
      }
      public void bindFc(String itfName, Object itf)
            throws NoSuchInterfaceException, IllegalBindingException,
                 IllegalLifeCycleException {
                  ...
          }
      public void unbindFc(String itfName) throws NoSuchInterfaceException,
              IllegalBindingException, IllegalLifeCycleException {
if (itfName.equals("confItf"))
                     conf = null;
                       ...
       }
}
```

FIG. 6.12 – *Extrait du code de la classe AbsService.*

```
   public class EJBServiceWrapper extends AbsService implements Deployer, EJBServiceMBean
   {
      // The references to the other services and default configurator are obtained thanks
      // to the implementation of the AbstractService interfaces
            transactionItf, secItf, confItf
            ...

            ...
      //-------------------------------------------------------------------------------
      // Deployer interface implementation
      //-------------------------------------------------------------------------------
      public void deploy(EJBApp Jar) throws ServiceException {
         ...
      }
      public void deploy(URL EJBUrl) throws ServiceException {
         ...
      }
     //-------------------------------------------------------------------------------
     // Service interface implementation
    //--------------------------------------------------------------------------------
      public void start() throws NoSuchInterfaceException,
            IllegalBindingException, IllegalLifeCycleException {
         ...
      }
      // code from EJBServiceImpl class
}
```

FIG. 6.13 – *Extrait de l'implantation de l'adaptateur du service EJB.*

les services de JonasALaCarte l'implantation d'interfaces MBeans afin de satisfaire le standard JSR77.

6.5 Synthèse

Dans ce chapitre, nous avons montré le travail de réingénierie effectué dans JOnAS et donnant lieu au prototype JonasALaCarte. Le but de ce travail est de rendre le serveur *administrable* c'est à dire de rendre son architecture suffisamment flexible pour automatiser sa configuration et sa reconfiguration.

Pour effectuer ce travail, nous avons identifié deux phases : la phase de transformation de l'architecture du serveur en une architecture explicite à l'exécution et la phase d'empaquetage et de chargement de classes. La première phase est réalisée grâce à l'encapsulation des services dans des composants Fractal. Les dépendances sont exprimées en termes de liaisons Fractal dynamiquement reconfigurables, locales à une seule JVM ou distantes. Pour la phase d'empaquetage et de chargement de classes, nous avons mené deux expériences : la première utilise le standard OSGi et la deuxième est entièrement basée sur Fractal. L'avantage de la première solution est de faciliter l'interopérabilité entre plusieurs fournisseurs de code grâce au format standard des paquetages. L'avantage d'une solution entièrement basée sur Fractal est d'uniformiser les outils de gestion des composants du paquetage à l'exécution. Grâce à ces deux expériences, nous avons pu définir des abstractions du processus de déploiement indépendamment de l'implantation; ceci sera décrit dans le chapitre 7. Enfin, ce travail de réingénierie est applicable à d'autres serveurs d'applications et d'autres intergiciels.

Ce chapitre a montré notre démarche pour transformer un intergiciel patrimonial en un logiciel modulaire et donc *administrable*. Le chapitre 7 est dédié à la présentation du système d'administration et à l'illustration des propriétés permettant d'automatiser le déploiement distribué et la reconfiguration.

Chapitre 7

Système d'administration fondée sur l'architecture

Sommaire

Dans le chapitre 6, nous avons présenté JonasALaCarte, un serveur d'applications J2EE administrable constuit par une réingénierie du serveur patrimonial JOnAS. Dans ce chapitre, nous décrivons un système d'administration pour JonasALaCarte. Ce système d'administration permet trois types de fonctions d'administration : déploiement distribué, reconfiguration d'implantation et reconfiguration de structure.

Ce chapitre est structuré de la façon suivante. Nous rappellons les besoins de l'administration des serveurs J2EE dans la section 7.1. La section 7.2 décrit Jade, un canevas d'administration

qui nous a servi de base pour administrer JonasALaCarte. Nous décrivons dans la section 7.3 nos extensions au système Jade permettant d'effectuer un déploiement distribué, ainsi que des reconfigurations d'implantation et de structure. Enfin, nous décrivons l'utilisation du système d'administration obtenu pour déployer et reconfigurer JonasALaCarte dans les environnements distribués.

7.1 Besoins de l'administration des serveurs J2EE

Nous rappelons dans cette section les deux besoins de l'administration des serveurs J2EE décrits dans 4.5 : le déploiement distribué et la reconfiguration dynamique (d'implantation et de structure).

7.1.1 Déploiement de l'intergiciel J2EE

JSR88, le standard de déploiement J2EE, spécifie seulement le déploiement des applications ; le déploiement de l'intergciel est laissé à la responsabilité des fournisseurs des serveurs. Le déploiement de l'intergiciel est généralement effectué manuellement ou en écrivant des scripts dans le cas du déploiement distribué. Ces scripts sont toutefois écrits d'une manière ad hoc, ce qui nécessite un temps considérable dans le cas des architectures complexes de type grappe. En effet, en plus des éléments de l'intergiciel, de nouveaux éléments sont introduits dans les grappes J2EE pour équilibrer la charge et pour assurer la cohérence entre les données dupliquées, ce qui rend complexe l'architecture distribuée à administrer. La figure 7.1 présente un exemple d'une telle architecture, celui de JOnAS en grappe. Des équilibreurs de charges sont introduits en amont des duplicas des conteneurs Web et EJB : mod_jk et CMI (*Clustered Method Invocation*). Mod_jk est un module du serveur Apache permettant de transférer les requêtes HTTP du serveur Apache vers les conteneurs Web ; le protocole utilisé est appelé `AJP13`. CMI est une extension de RMI qui permet, lorsqu'il est utilisé dans les *Registry* des serveurs Web, d'équilibrer la charge entre les conteneurs EJB.

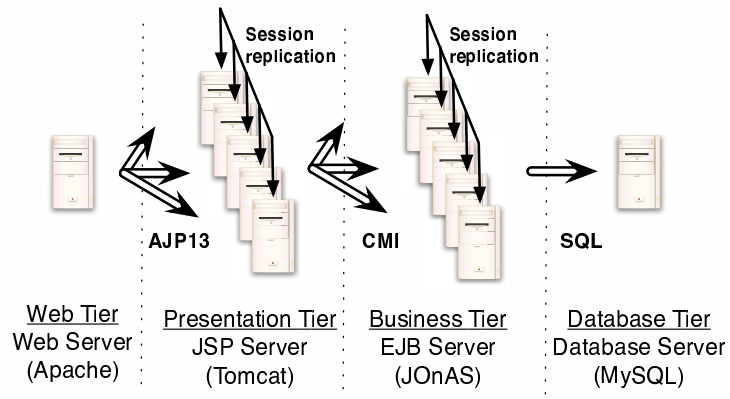

FIG. 7.1 – *Environnement de grappes J2EE.*

Face à la complexité des serveurs d'applications J2EE sur grappe de machines, il est nécessaire d'automatiser le déploiement des différentes parties du serveur. L'administrateur doit être capable de décrire l'architecture distribuée à déployer et de disposer des outils nécessaires pour installer les éléments de cette architecture sur les machines cibles.

7.1.2 Reconfiguration

Nous distinguons deux types de reconfiguration : la reconfiguration de structure et la reconfiguration d'implantation.

Reconfiguration de structure À l'exécution, un système peut recevoir plusieurs événements extérieurs nécessitant la modification de sa structure. Nous identifions deux causes à ces événements :

- **Prévention de la surcharge des serveurs** : la surcharge des systèmes J2EE se produit quand le système est incapable de supporter un pic de requêtes clientes, ce qui peut provoquer la dégradation de la qualité de service. Dans un environnement grappe, la duplication des conteneurs Web et EJB permet de supporter la charge. Pour une utilisation optimale des ressources, le nombre de duplicas doit être adapaté de sorte qu'il reste proportionnel à la charge.
- **Réaction aux fautes** : la duplication des services permet d'assurer la continuité des services quand une panne logicielle ou matérielle se produit. Cependant, pour maintenir le même niveau de performance, le nœud fautif doit être réparé ou remplacé.

Pour accomplir ces tâches, l'administrateur doit disposer d'une vue générale du système et de moyens d'agir d'une manière cohérente sur toutes les ressources mises en place. Par exemple, gérer une panne logicielle dans l'étage EJB peut nécessiter une intervention en amont pour bloquer les appels entrants pendant le temps de la réparation. De plus, arrêter ou démarrer un service nécessite la gestion des dépendances avec les autres services.

Reconfiguration d'implantation La reconfiguration d'implantation que nous considérons dans cette thèse consiste à mettre à jour dynamiquement le code des services dans JOnAS. Celle-ci n'est pas réalisable actuellement dans JOnAS pour les raisons suivantes : la plupart des services sont chargés par un même *class loader* ; il n'est donc pas possible d'avoir deux versions d'un même service chargées. Par ailleurs, comme nous l'avons évoqué au chapitre 4, la communication entre les services n'est pas explicite. Il n'est donc pas possible d'arrêter un service pour le mettre à jour, sans impacter les autres services.

7.2 Jade : un canevas réflexif pour l'administration fondée sur l'architecture

Pour construire notre système d'administration, nous nous sommes basés sur Jade [BBH+05], un canevas logiciel dédié à la construction de systèmes distribués administrables. Jade est construit suivant les principes de l'administration fondée sur l'architecture. En effet, Jade suppose que le système administré est construit à l'aide de composants Fractal et utilise l'ADL Fractal pour décrire l'architecture à déployer. Toutes les ressources du système (nœuds, entités logicielles, etc.) sont des composants Fractal et leurs dépendances sont exprimées à l'aide de propriétés d'encapsulation et de liaisons. L'architecture est par conséquent explicite et elle est dynamiquement reconfigurable.

Les éléments principaux du canevas forment une architecture de boucle de commande. Cette structure est semblable à celle des boucles de commande utilisées dans la théorie de la commande [Oga97]. Ces boucles sont en charge de la régulation et de l'optimisation du comportement du système administré et tous les éléments intervenant dans la boucle sont sous forme de composants Fractal.

Nous commençons par une description de l'architecture de Jade et nous présentons, par la suite, les outils d'administration que Jade met à la disposition de l'administrateur. Enfin, nous discutons des parties manquantes que nous avons ajoutées dans le cadre de cette thèse.

7.2.1 Architecture

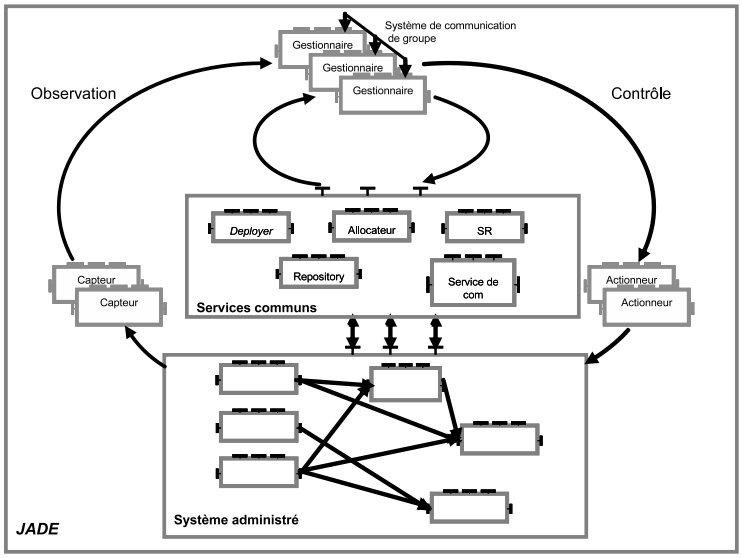

FIG. 7.2 – *Architecture de Jade, un canevas de systèmes d'administration autonomes.*

La figure 7.2 présente l'architecture de Jade. Celle-ci est organisée sous forme de trois types de composants : les élements administrés (*managed elements*), les gestionnaires autonomes (*autonomic managers*) et les services communs (*common services*). Les nœuds physiques hébergeant les composants sont appelés *managed nodes*. Ils sont distincts dans Jade des nœuds hébergeant les services et les gestionnaires. Ces derniers nécessitent, en effet, un traitement particulier comme la duplication des nœuds sous-jacents pour tolérer la panne du système d'administration. Nous décrivons dans ce qui suit les composants de Jade.

Eléments administrés Un élément administré est un composant Fractal qui encapsule une entité à contrôler. Ce composant doit offrir des *capteurs* pour collecter les informations sur l'entité à contrôler et des *actionneurs* pour changer son état. Les *capteurs* et les *actionneurs* peuvent être implantés sous forme d'interfaces au niveau du composant administré ou sous forme de composants liés aux éléments à administrer par des liaisons Fractal. Un élément administré peut être une ressource logicielle ou matérielle, un composant primitif ou composite, un composant localisé dans un même espace d'adressage ou un composite distribué.

Les *managed nodes* sont des cas particuliers de composants à administrer. Ils offrent les interfaces nécessaires pour installer les paquetages des composants et pour créer les éléments administrés sur les nœuds.

Gestionnaires Ce sont des composants Fractal implantant la politique d'administration, c'est à dire les phases d'analyse, de décision, de planification et d'exécution de la boucle de commande. Ils observent les éléments administrés, analysent les événements provenant des capteurs, font des diagnostics sur l'état du système, planifient des actions en réponse aux événements observés

et, respectant les contraintes des politiques d'administration de plus haut niveau, ils exécutent leur plan d'action. Un gestionnaire est aussi un élément administré. Par conséquent, il peut être ajouté ou retiré à distance et il peut être contrôlé par un autre gestionnaire.

Services communs Les services communs dans Jade fournissent aux composants gestionnaires un ensemble d'outils de base pour effectuer des fonctions de configuration comme l'installation, le déploiement des composants et la communication avec les actionneurs. Les composants fournissant ces services sont des éléments administrés et, donc, sujets à l'administration comme les autres composants du système administré. Les services actuellement fournis dans Jade sont les suivants :

- Le **service de nommage** offre la possibilité de retrouver les composants dans le domaine de Jade.
- Le **service de découverte** est utilisé pour enregistrer et désenregistrer de nouveaux nœuds dans le système administré par Jade. Il s'agit d'un point d'entrée pour Jade servant à modifier l'ensemble de nœuds physiques administrés.
- Le **service allocateur** est utilisé pour allouer des noeuds physiques connus, c'est à dire des noeuds enregistrés au niveau du service de découverte. L'allocation de noeuds peut prendre plusieurs formes et peut offrir des niveaux plus ou moins sophistiqués de virtualisation de ressources. Le service allocateur est typiquement utilisé par les composants gestionnaires pour allouer ou réallouer des nœuds.
- Les **services de communications** offrent des facilités de communication entre les composants de Jade. Ils implantent des usines de liaisons Fractal servant de canaux de communication. Ces canaux sont utilisés pour transmettre les données d'observation et les messages de commande. Ils sont également développés sous forme de composants Fractal implantant différents protocoles de communication. Le choix des composants de communication dépend de la nature des messages envoyés, du niveau de sécurité demandé et des performances recherchées.
- Le **service de représentation du système** (*System Representation* ou SR) fournit une vue du système administré causalement connectée à ce dernier. Les gestionnaires se servent du SR pour connaître l'architecture du système (y compris en cas de panne). Jade offre la possibilité de générer systématiquement le SR à partir de l'ADL lors du déploiement. Pour générer le SR, il est nécessaire d'utiliser une *usine de composants réflexive*, appelée `ReflexFactoryFactory`, étendant l'usine Fractal par défaut. L'*usine réflexive* dipose d'une *usine duale* qui reproduit les mêmes opérations de création de composants. Lorsque l'*usine réflexive* crée un composant, l'*usine duale* crée également un composant ayant les mêmes contrôleurs ; ce composant est appelé *composant dual*.
  Tout appel de méthode effectué au niveau des contrôleurs d'un composant est dupliqué sur le composant dual. En revanche, un composant dual ne contient les interfaces fonctionnelles du composant. Par conséquent, les appels des méthodes fonctionnelles ne sont pas reproduits au niveau des composants.

Tolérance aux pannes du système d'administration Afin d'assurer la fiabilité du système d'administration lui-même, les composants gestionnaires sont dupliqués et une boucle de commande de méta-niveau assure l'auto-réparation des gestionnaires. Ceci est assuré par deux mécanismes : la duplication active des composants gestionnaires et l'intégration dans le SR d'une représentation des composants gestionnaires eux-mêmes (avec leurs configurations et leurs liaisons comme les autres composants administrés).

7.2.2 Outils pour l'administration

Afin d'utiliser le canevas Jade, trois conditions doivent être satisfaites :

- le système administré doit être structuré sous forme de composants Fractal et implanter les capteurs et les actionneurs qui lui sont spécifiques ;
- le développeur doit implanter les composants gestionnaires décrivant les politiques d'administration désirées ;
- le développeur doit écrire les fichiers ADL décrivant l'architecture du système à déployer.

Le canevas offre l'ensemble des services permettant d'allouer les nœuds et de configurer et déployer le système à administrer.

Déploiement Le déploiement dans Jade se base sur les composants *Deployer* et *Repository* (Figure 7.2). Le *Deployer* prend en entrée le fichier ADL décrivant l'architecture du système à déployer, extrait les binaires qui sont déposés dans le *Repository* et les installe sur les machines cibles. Il procède ensuite à l'instanciation des composants et à la création des liaisons entre composants. Afin de permettre la création de composants, chaque nœud administré contient un *bootstrap* Fractal.

Reconfiguration Grâce à l'architecture explicite du système administré, la *reconfiguration de structure* est facilement réalisable. En effet, ajouter un composant dans une structure nécessite trois phases :

- déployer le nouveau composant,
- ajouter le composant au sein d'un composite,
- établir les liaisons avec les autres composants.

En revanche, Jade ne permet pas de faire des *reconfigurations d'implantation*. En effet, le déploiement tel qu'il est fait dans Jade ne permet pas de procéder à des mises à jour ultérieures de code. Les deux limitations principales sont qu'il n'est pas possible de gérer les dépendances entre les paquetages et qu'il n'y a pas de mécanismes évolués de chargement de classes permettant la mise à jour du code des composants. En effet, le système de déploiement de Jade n'utilise qu'un seul *class loader* par machine cible. Dans le cas de JOnAS, il est important de pouvoir exprimer les dépendances entre les paquetages des services et de disposer de mécanismes évolués de chargement de classes permettant la mise à jour du code d'un service sans redémarrer la totalité du serveur.

Dans la suite de ce chapitre, nous présentons deux extensions au système Jade. La première extension est un nouveau processus de déploiement intégrant notamment les phases d'empaquetage et de chargement de classes. La seconde extension est un mécanisme de reconfiguration d'implantation.

7.3 Système de déploiement et de reconfiguration

Dans cette section nous présentons le système de déploiement que nous avons intégré à Jade, ainsi que les mécanismes de reconfiguration mis en place[1]. Nous commençons tout d'abord par présenter une extension du composant SR qui permet de faciliter l'implantation des fonctions de déploiement et de reconfiguration. Nous présentons ensuite la définition des abstractions d'un système de déploiement d'une manière générale en 7.3.2, ainsi que son implantation à l'aide de

1. Ce travail a été réalisé en collaboration avec Jakub Kornaś et Jean-Bernard Stefani.

Fractal et OSGi. Nous présentons en 7.3.3 les mécanismes de reconfiguration possibles grâce à notre système de déploiement.

7.3.1 Extension du composant SR

Le SR tel qu'il est défini dans Jade a un rôle passif. En effet, les politiques d'administration sont appliquées directement sur le système administré et le rôle du SR est réduit à la représentation des modifications de l'architecture et de la configuration du système administré. Dans le cadre de notre travail, nous avons ajouté de nouvelles fonctionnalités au SR.

Implantation de caches. Les composants duaux conservent les informations sur l'état du composant auquel ils sont associés. Ceci permet de minimiser l'intrusitivité sur le système administré et sur le réseau. Comme les informations d'observation sont fournies au niveau du SR, les gestionnaires n'ont à connaître qu'une seule adresse, celle du SR, pour obtenir une vue de l'architecture et de la configuration courante du système administré.

Exposition de l'état des composants du système. Nous avons étendu le cycle de vie des composants au niveau de la carte afin de pouvoir présenter l'état des composants fautifs. Ceci évite que les gestionnaires envoient des requêtes aux composants fautifs dès qu'ils découvrent leur état au niveau de la carte.

Définition de nouvelles politiques d'administration. Nous avons étendu le SR afin de pouvoir définir des *domaines d'administration*. La notion de *domaine* [ST94] permet de regrouper des entités sous une même autorité. Cette notion a été initialement introduite pour définir des aires de protection à travers des droits d'accès partagés. Elle a été étendue par la suite pour regrouper plus généralement des entités partageant un ensemble de propriétés et pour administrer des ressources physiques ou logiques avec les objectifs suivants : (1) définir des politiques communes et des règles d'usage des éléments d'un même domaine et (2) définir un contexte de nommage pour les objets dans le domaine.

Un domaine peut être associé à un emplacement géographique ou refléter la structure d'une application complexe. Dans le SR, nous présentons les domaines sous forme de composites englobant des composants du SR. Ces composites n'ont pas de correspondants au niveau du système administré. Leur but est de ramener les fonctions d'administration à des opérations plus simples et de plus haut niveau. Nous avons, par exemple, utilisé la notion de domaines dans JonasALaCarte pour regrouper tous les exemplaires de services ayant besoin de subir un même traitement. Ces services sont, par conséquent, exposés à l'administrateur sous forme d'un seul service virtuel et chaque opération sur le service virtuel est dupliquée au niveau des services encapsulés.

Par analogie avec la carte géographique, le SR étendu offre une vue du système administré et de son état ; il permet de définir des domaines d'administration en superposant des frontières entre les composants. Nous avons donc choisi d'appeler *carte* le SR étendu et nous utiliserons ce nom dans le reste de ce document.

7.3.2 Système de déploiement

Nous avons vu en 2.2.1 que la complexité des systèmes actuels a mis en évidence le besoin d'unifier les outils de configuration et de déploiement [vdHHW98] et d'adopter des descripteurs

d'architecture illustrant explicitement les dépendances entre les composants (en termes de liaisons et de compatibilité de versions). C'est ce que nous entendons par *déploiement fondé sur l'architecture*. Nous présentons d'abord l'architecture d'un système de déploiement et ses différentes abstractions d'une façon générale. Nous décrivons, par la suite, les étapes d'un processus de déploiement ainsi que notre implantation. Nous finissons cette partie par un exemple simple.

Architecture d'un système de déploiement

Nous identifions dans un système de déploiement les quatre éléments principaux présentés dans la figure 7.3 :

1. le descripteur de déploiement (*deployment descriptor*),
2. les cibles (*targets*),
3. le dépôt des paquetages (*package repository*),
4. le programme de déploiement (*deployment engine*).

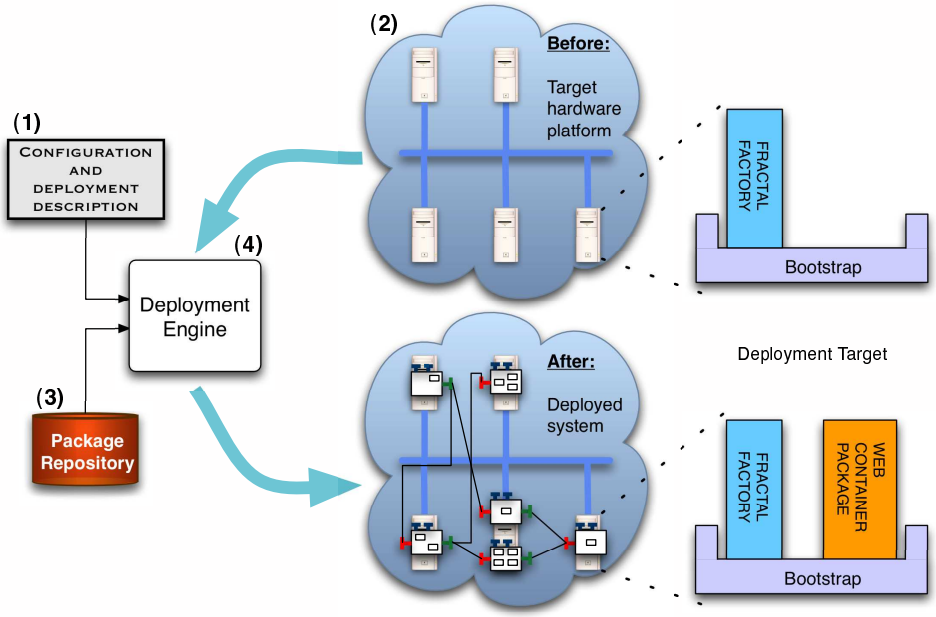

FIG. 7.3 – *Architecture d'un système de déploiement.*

Le descripteur de déploiement est constitué d'un ou de plusieurs fichiers ADL permettant de décrire les composants du système, leurs paramètres de configuration et leurs relations. Le but du système de déploiement est de transformer l'architecture décrite dans le descripteur de déploiement en un système distribué sur les machines cibles. Le dépôt des paquetages contient les paquetages disponibles pour installation. Le *deployment engine* est le programme qui analyse le descripteur de déploiement et qui déclenche le processus de déploiement en appelant les interfaces appropriées au niveau des cibles. Les cibles sont formées d'un ensemble de nœuds, éventuellement hétérogènes sur lesquels les composants sont installés et instanciés. Chaque nœud dispose d'un

logiciel de démarrage (*bootstrap*) implantant diverses interfaces permettant d'installer/charger des paquetages, ainsi que de créer des composants.

Ces différents éléments coopèrent pour mettre en œuvre le processus de déploiement décrit dans le paragraphe suivant.

Processus de déploiement

Les objectifs du processus de déploiement sont :

- l'installation des paquetages,
- la désinstallation des paquetages inutilisés,
- l'instantiation des composants respectant l'architecture décrite dans le fichier ADL.

Pour ce faire, nous avons défini un ensemble d'interfaces coopérant pour réaliser les opérations de déploiement. La figure 7.4 représente les étapes du processus de déploiement. Les interfaces **Generic Factory**, **Installer**, **Loader Factory** et **Component Factory** sont implantées par le *bootstrap*.

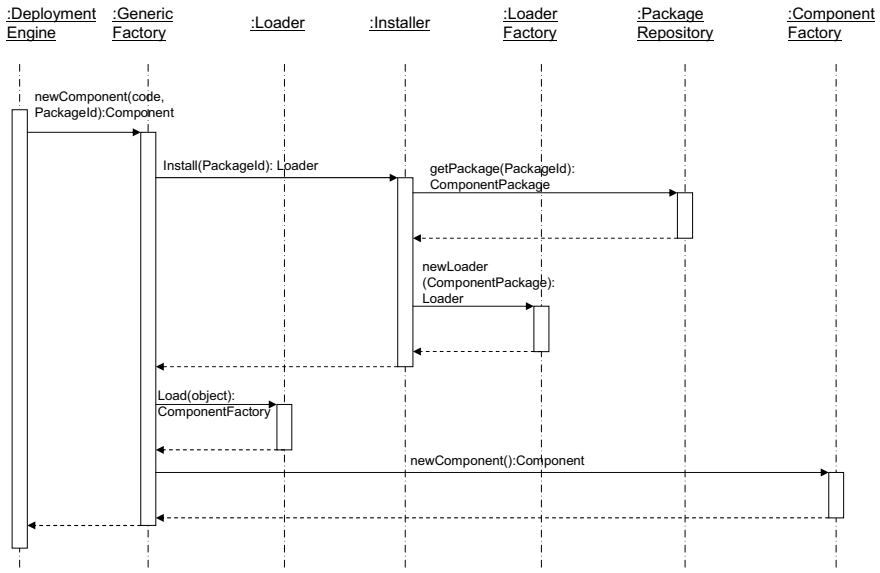

FIG. 7.4 – *Etapes du processus de déploiement.*

- Le *deployment engine* prend en entrée l'ADL du descripteur de déploiement. Pour chaque composant, il extrait l'adresse de la machine cible, des informations sur le code de l'application (e.g. le type du composant) et l'identificateur du paquetage que nous nommons *packageId*. Le *deployment engine* appelle la méthode *newComponent(code, packageId)* sur l'interface *GenericFactory* de la machine cible.
- Le *Generic Factory* de la machine cible envoie à l'*Installer* le *packageId* du composant à déployer. L'*Installer* contacte le dépôt des paquetages pour télécharger les paquetages correspondant au *packageId* et les installe localement sur la machine.
- L'*Installer* demande au *LoaderFactory* de créer un chargeur du composant à partir de son paquetage installé et le retourne au *Generic Factory*.

– Le *Generic Factory* utilise le chargeur pour créer une instance du composant. Il configure cette instance à l'aide du *Component Factory*.

Une fois les composants installés, instanciés et configurés, le *deployment engine* établit les liaisons entre les composants comme elles sont définies dans le descripteur de déploiement. Enfin, le *deployment engine* démarre l'application en utilisant les interfaces de cycle de vie des composants.

Cycle de vie des paquetages

La gestion des paquetages sur les machines cibles est effectuée selon une technique de désinstallation paresseuse. Les paquetages qui ne sont plus utilisés sont supprimés de la machine cible par un **ramasse-miettes (*Garbage collector*)**. Un paquetage peut être dans l'un des états suivants : *installé*, *désinstallé* ou *marqué*. Il est *installé* lorsqu'il est déposé sur la machine cible, il occupe alors une place dédiée de l'éspace disque (par exemple un répertoire contenant le code du paquetage). L'installation d'un paquetage entraîne l'installation de tous les paquetages dont il dépend. La relation entre des paquetages installés est exprimée en termes de « pointeurs ». Le paquetage est *désinstallé* lorsqu'il n'est plus utilisé dans le chargement de composants sur une machine cible ; la place mémoire qu'il occupe est alors libérée par le ramasse-miettes. L'*Installer* peut décider de désinstaller un paquetage lorsqu'un composant est supprimé de l'application instanciée. Lorsqu'un paquetage est désinstallé, tous les paquetages dont il dépend ne sont plus pointés. Le paquetage est *marqué* lorsque l'administrateur souhaite le garder installé sur la machine cible même s'il n'est pas utilisé dans le chargement d'un composant. Le paquetage pourrait servir au chargement d'un autre composant.

La figure 7.5 illustre un exemple de ramasse-miettes de paquetages. Nous considérons le cas de deux composants `a` et `b` correspondant respectivement aux paquetages `1` et `4`. Afin de fonctionner correctement, le paquetage `1` a besoin que les paquetages `2` et `3` soient installés. Les flèches dans la figure 7.5 représentent ces dépendances. La suppression du composant `a` entraine la désinstallation du paquetage `1`. Le paquetage `2` qui n'est plus utilisé sera désinstallé par le ramasse-miette, contrairement au paquetage `3` qui est encore sollicité par le paquetage `4`. Notons que nous supposons que le paquetage `2` n'est pas marqué ; le cas échéant, il serait gardé installé sur la machine cible.

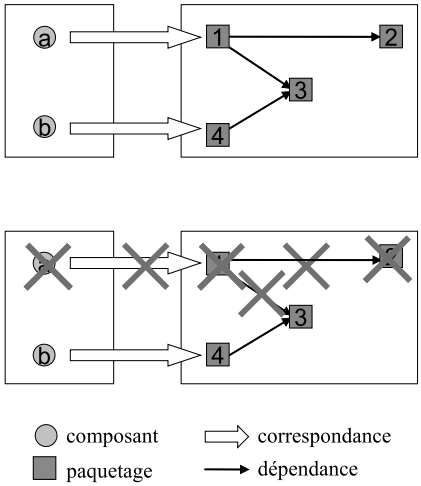

FIG. 7.5 – *Exemple de ramasse-miettes de paquetages.*

Implantation du processus de déploiement

Le système de déploiement décrit dans les sections précédentes a été implanté à l'aide de Fractal et OSGi. Les paquetages sont implantés sous forme de *bundles* OSGi et les dépendances entre les paquetages sont exprimées dans les fichiers *manifest* sous forme de relations *import/export*. Le *packageId* est actuellement une simple chaîne de caractères concaténant le nom du composant correspondant au paquetage au mot « Package ». Nous envisageons, dans une version future, de calculer le *PackageId* en appliquant, comme dans Nix, un hash sur le paquetage à installer et de tous les paquetages dont il dépend. Ceci permettra d'obtenir un identificateur unique par composant.

L'*Installer* est implanté par un composant Fractal encapsulant le démon OSGi. Lorsque la méthode `install(PackageId)` est appelée, le composant *Installer* télécharge les paquetages correspondant au *PackageId* du dépôt des paquetages et les installe localement sur la machine. L'installation dans OSGi nécessite que chaque paquetage soit associé à un répertoire dans le système de fichiers de la machine cible ainsi qu'à un *class loader*. Chaque *class loader* est configuré de sorte qu'il trouve les ressources (les répertoires correspondant au code des paquetages) et le *class loader* à qui il va déléguer le chargement des classes. Nous avons modifié le code d'OSGi afin que chaque *bundle* ait une méthode *getClassLoader()* retournant le *classloader* qui l'a créé.

Le dépôt de paquetages utilise *Oscar Bundle Repository* (OBR)[2]. Ce dépôt contient les paquetages des composants à installer. La communication entre le composant *Installer* et le dépôt est implantée à l'aide d'une liaison Fractal afin de la rendre explicite. Ceci permet de configurer l'accès au dépôt en fonction des besoins, par exemple pour le sécuriser ou adapter son protocole de communication (RMI, HTTP, TCP, etc).

Par ailleurs, nous avons étendu l'ADL Fractal pour permettre de spécifier, pour chaque composant à déployer, le *packageId* de son paquetage. Le *Generic Factory* est un composant Fractal installé sur la machine cible et encapsulant l'usine Fractal. Le *deployment engine* étend l'usine de composants Fractal. Il permet d'analyser les fichiers ADL de l'architecture globale de l'application et d'envoyer les fichiers ADL correspondant aux composants à déployer au *Generic Factory* de chaque machine cible.

Exemple

Considérons le système composé de 3 composants (C_1, C_2 et C_3) représenté sur la figure 7.6. A chaque composant C_i correspond un paquetage que l'administrateur a installé au niveau du dépôt de paquetages. La figure 7.7 illustre le descripteur de déploiement de cette architecture. A l'exécution, C_1 encapsule C_2 et C_3. Par ailleurs, les composants C_2 et C_3 sont liés par une laison Fractal.

La figure 7.6 illustre la correspondance entre les paquetages des composants C_i et les composants à l'exécution. Chaque nœud cible doit contenir un *bootstrap* (contenant *Generic Factory*, *Installer*, *LoaderFactory*). Les composants C_1 et C_2 sont déployés sur le nœud1 et le composant C_3 sur le nœud2. Dans cet exemple, nous faisons l'hypothèse que le code des interfaces entre les composants est stable et qu'il n'est pas sujet aux modifications aussi fréquemment que le code des composants[3]. Nous avons fait le choix, par conséquent, d'archiver le code de toutes les interfaces dans un paquetage séparé et de charger ce code par un même *class loader*.

2. `http://oscar-osgi.sourceforge.net/`

3. Cette hypothèse est valable dans le cadre des serveurs d'applications où les interfaces entre les services évoluent rarement et où le code des services change beaucoup plus fréquemment.

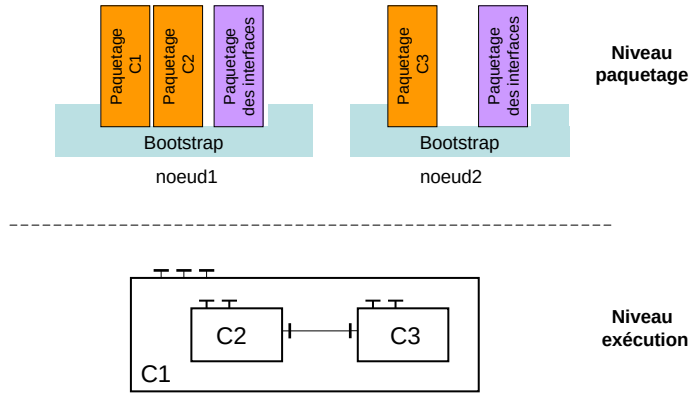

FIG. 7.6 – *Exemple de déploiement de composants.*

```
<definition name="C1">
  <component name="C2">
    <interface name="service"
      role="client" signature="Service"/>
    <content class="C2Impl"/>
    <package name="C2 Package"/>
    <virtual-node name="node1"/>
  </component>
  <component name="C3">
    <interface name="service"
      role="server" signature="Service"/>
    <content class="C3Impl"/>
    <package name="C3 Package"/>
    <virtual-node name="node2"/>
  </component>
  <binding from="C2.service" to="C3.service"/>
  <package name="C1 Package"/>
  <virtual-node name="node1"/>
</definition>
```

FIG. 7.7 – *Exemple de fichier ADL de descripteur de déploiement.*

7.3.3 Reconfiguration

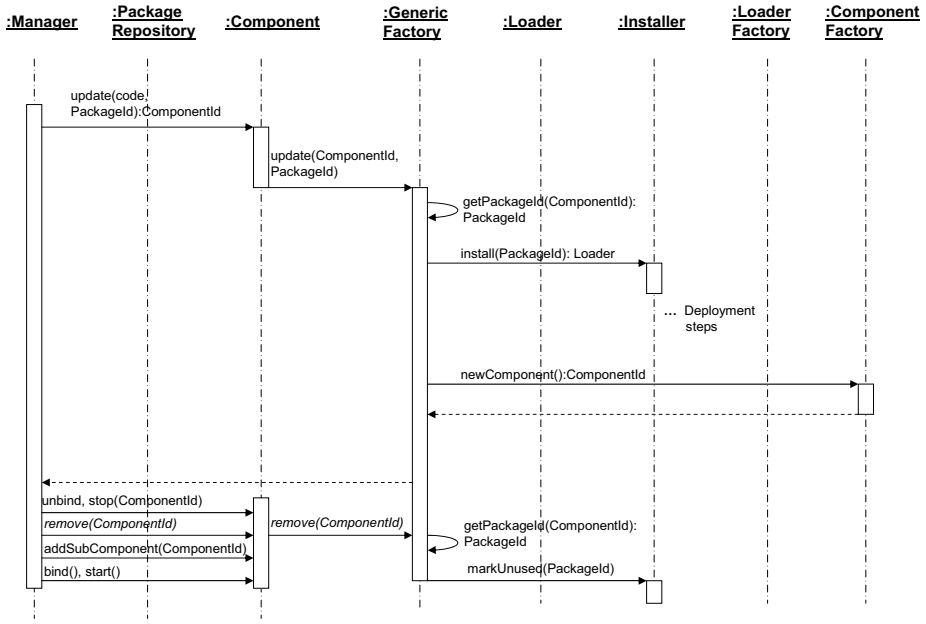

FIG. 7.8 – *Etapes de la reconfiguration.*

Les opérations de reconfiguration comme la mise à jour du code d'un composant ou le remplacement d'un composant fautif sont implantées au niveau des gestionnaires Jade. Pour illustrer les éléments intervenant dans une opération de reconfiguration, considérons l'exemple de la figure 7.6[4]. Nous proposons de traiter deux cas : le cas de l'insertion d'un composant C_4 entre C_2 et C_3 pour illustrer la reconfiguration de structure et le cas où le gestionnaire décide de mettre à jour le code de C_3 pour illustrer la reconfiguration d'implantation.

Reconfiguration de structure Pour insérer un composant C_4 entre C_2 et C_3, il faut effectuer les étapes suivantes.

- Déployer le composant C_4 sur une machine cible, par exemple sur un nœud3, utilisant le processus de déploiement défini dans 7.3.2.
- Ajouter C_4 dans C_1 en utilisant l'interface *content controller* de C_1.
- Couper la liaison entre C_2 et C_3.
- Établir les liaisons entre C_2 et C_4, d'une part, et entre C_4 et C_3, d'autre part.

Dans cet exemple, nous ne montrons que les mécanismes de la reconfiguration. Il est nécessaire de tenir compte de l'état des composants pendant l'opération de reconfiguration. Nous donnerons un exemple de prise en compte de l'état des composants dans notre description de la reconfiguration du service Web dans JOnAS.

4. Notons que cet exemple est simple ; l'architecture ne présente qu'un seul niveau de hiérarchie et ne présente pas de composants partagés. Cette architecture correspond à celle de JonasALaCarte et les mécanismes de reconfiguration sont donc applicables à notre étude de cas. Les cas les plus complexes (avec plusieurs niveaux de hiérarchie et des composants partagés) nécessitent une étude plus fine des mécanismes de reconfiguration.

Reconfiguration d'implantation La mise à jour d'un composant nécessite d'effectuer les étapes suivantes :

- Déployer le composant du nouveau paquetage,
- Isoler l'ancien composant en coupant les liaisons avec les autres composants,
- Supprimer le composant contenant l'ancien code,
- Établir les liaisons avec le nouveau composant installé suivant l'architecture initiale.

La figure 7.8 présente les éléments et les intéractions nécessaires à l'établissement d'une reconfiguration d'implantation. Le composant gestionnaire appelle la méthode `Component update(Code C3Code,PackageId newC3PackageId)` sur le composant C_3 en passant comme paramètres les informations sur le nouveau composant à installer (son type) et l'identificateur de son paquetage.

Le composant C_3 contacte son *Generic Factory* en appelant la méthode `update(ComponentId self, PackageId newCodePackage)`. Le *Generic Factory* effectue les étapes de déploiement nécessaires pour installer et instancier le nouveau composant ayant pour but de remplacer C_3. Il retourne, alors, la référence *componentId* du nouveau composant que nous désignons par C_3'.

Le gestionnaire casse la liaison entre l'ancien composant et les autres composants (dans notre cas la liaison entre C_3 et C_2) en appelant la méthode `unbind()` (sur le composant `C2`) et établit les mêmes liaisons avec le nouveau composant déployé (entre C_2 et C_3'). Enfin, il supprime l'ancien C_3 en appelant la méthode *remove* implantée par le composant. Le composant appelle alors `remove(ComponentId selfId)` sur son *Generic Factory*.

Le *Generic Factory* qui maintient une table de correspondance entre les *componentId* et *packageId* retourne le *packageId* correspondant au composant C_3 et appelle la méthode `markUnused(packageId)` du composant *Installer*. Le paquetage correspondant au composant désinstallé est nettoyé par la suite par le ramasse-miettes suivant le protocole décrit dans 7.3.2.

Enfin, le gestionnaire ajoute le composant C_3' au composite C_1 et procède à l'établissement des liaisons entre le composant C_3' et les autres C.

7.3.4 Synthèse

Nous avons présenté un système de déploiement permettant d'installer et d'instancier des composants Fractal sur des systèmes distribués. Les paquetages sont sous forme de *bundles* OSGi et sont initialement installés dans un dépôt de paquetages. Un *deployment engine* se charge d'installer les paquetages sur les machines cibles à partir d'un dépôt. Nous avons défini des interfaces que doit implanter chaque machine cible pour charger le code des composants. L'isolation des composants dans des paquetages différents et leur chargement d'une manière séparée nous a permis d'effectuer des reconfigurations d'implantation (mise à jour du code des composants) et de structure (ajout ou suppression de composants).

Nous montrons dans la section suivante l'utilisation du processus de déploiement et des mécanismes de reconfiguration dans JonasALaCarte.

7.4 Administration dans JonasALaCarte

Dans le chapitre 6, nous avons présenté la réingénierie de JOnAS à l'aide du modèle de composants Fractal. Le prototype obtenu, appelé JonasALaCarte, encapsule les services dans des composants Fractal et modèlise les dépendances entre ces services à l'aide de liaisons Fractal.

Dans cette section, nous décrivons l'intégration de JonasALaCarte au sein du système Jade. La figure 7.9 représente l'architecture du système résultant. Pour simplifier la figure, nous ne représentons pas tous les services de Jade. L'architecture illustrée est composé principalement des éléments suivants :

- le système administré (constituant une configuration de JOnAS « fractalisé »),
- la carte représentant une vue du système administrée,
- un ensemble de capteurs permettant de récupérer des informations d'observation,
- un ensemble de composants gestionnaires implantant les politiques d'administration,
- un ensemble d'actionneurs exécutant les commandes envoyées par les gestionnaires.

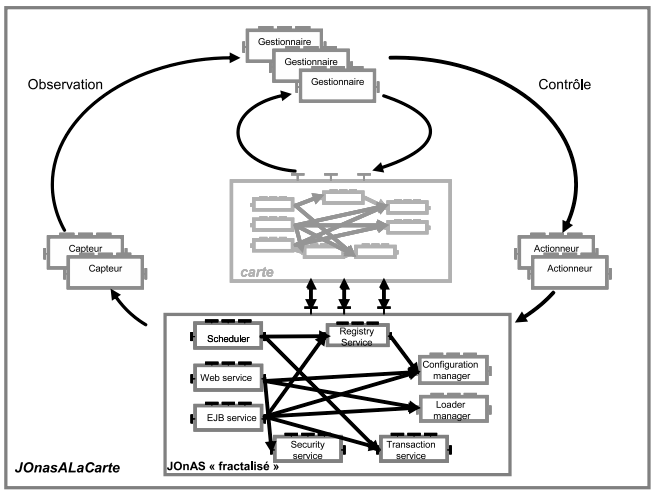

FIG. 7.9 – *Intégration de JonasALaCarte et de Jade.*

Dans la suite de cette section, nous montrons que l'utilisation de Jade (étendu avec les mécanismes décrits dans les sections précédentes) permet de résoudre les problèmes d'administration J2EE présentés en 7.1. Pour ce faire, nous commençons par décrire le déploiement de JonasALaCarte sur une grappe. Ensuite, nous montrons les propriétés de mise à jour dynamique du code des services utilisant les mécanismes de reconfiguration. Enfin, nous décrivons un scénario de réparation automatique de pannes.

7.4.1 Déploiement d'une grappe J2EE

Pour déployer une grappe J2EE d'une manière uniforme, il est nécessaire d'encapsuler dans des composants Fractal, en plus des services du serveur d'applications, tous les autres éléments de la grappe (le serveur HTTP et la base de données). Ceci permet d'avoir une même unité de déploiement, le composant Fractal, et d'utiliser donc les mêmes outils pour tous les étages J2EE.

Nous avons donc encapsulé dans des composants Fractal le serveur HTTP Apache et la base de données, au même titre que les services de JOnAS. Des interfaces de contrôle Fractal permettent de modifier les fichiers de configuration d'Apache et de la base de données. Par exemple, le contrôleur d'attributs associé au serveur HTTP Apache définit comme attributs l'ensemble des propriétés qui peuvent être configurées via le fichier `http.conf` (`User`, `Group`, `Port`, etc.).

Par ailleurs, les dépendances entre les éléments extérieurs et les services de JonasALaCarte sont réifiés sous forme de liaisons Fractal. La reconfiguration d'une liaison permet de modifier

les adresses et les numéros de ports aiguillant les requêtes. La définition de ces interconnexions se trouve dans les fichiers de configuration des logiciels introduits (`http.conf` pour Apache par exemple). Dans JonasALaCarte, nous distinguons alors deux types de liaisons Fractal : des liaisons de configuration (comme celles entre Apache et les services JOnAS) et des liaisons de communication (comme celles entre les services) qui correspondent à des canaux de communications ou des appels de méthodes.

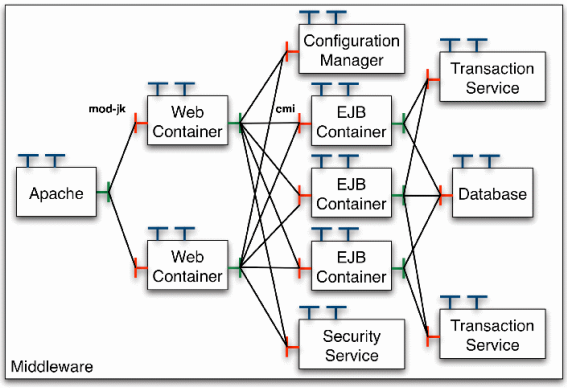

FIG. 7.10 – *Environnement de grappes J2EE : la vue composants.*

Pour déployer JonasALaCarte sur une grappe, l'administrateur doit, comme pour une configuration centralisée, décrire dans un ou plusieurs fichiers ADL l'architecture du système distribué. Ce fichier contient la désignation des machines cibles, les composants à déployer avec les identificateurs de leurs paquetages et les liaisons reflétant les dépendances entre les composants.

La figure 7.10 représente un exemple d'architecture à composants d'une grappe J2EE. Cette architecture est décrite dans le fichier de déploiement représenté dans la figure 7.11. Le déploiement du système se déroule de la même façon que le déploiement d'une application Fractal distribuée suivant le processus décrit dans 7.3. Tous les composants sont archivés dans des paquetages séparés (des *bundles* dans notre cas) et les interfaces entre les composants, supposées stables, sont dans un paquetage à part.

Outre le déploiement de l'intergiciel, l'administrateur doit déployer les applications J2EE. Pour ce faire, les conteneurs Web et EJB exposent des interfaces serveurs pour le déploiement des applications. L'administrateur (ou le composant gestionnaire) appelle la méthode `deploy(URL applicationUrl)` au niveau de ces interfaces en passant l'adresse à laquelle les modules de l'application à déployer sont déposés. Les interfaces de déploiement sont en fait implantées dans JOnAS suivant le standard de déploiement des applications, JSR88. Nous avons juste exposé ces interfaces sous forme d'interfaces Fractal et nous réutilisons les mécanismes de déploiement des applications implantés dans JOnAS [5].

Notons que la carte peut être utilisée afin de définir plusieurs politiques de déploiement (en définissant des domaines d'administration au niveau de la carte). En effet, une grappe peut héberger plusieurs applications J2EE. Il est pratique de regrouper les différents services Web et EJB exécutant une même application dans un même composite. Ce composite est alors vu par l'administrateur comme un conteneur virtuel. Une interface de déploiement d'applications est définie au niveau du composite. L'appel de la méthode `deploy` au niveau de cette interface entraine l'appel de la même méthode au niveau des sous-composants.

5. Une étude de la conformité de notre solution aux standards J2EE est présentée au chapitre 8. En particulier, nous discutons de la possibilité d'implanter le standard JS88 en utilisant les abstractions de déploiement introduites dans ce chapitre.

```
<definition name="org.objectweb.jonasALaCarte.Server">
  <component name="Apache" definition="org.objectweb.jonasALaCarte.Apache">
      <virtual-node name="node8" />
   <component name="ejb1" definition="org.objectweb.jonasALaCarte.EJB">
      <virtual-node name="node1" />
   </component>
<component name="ejb2" definition="org.objectweb.jonasALaCarte.EJB">
      <virtual-node name="node2" />
   </component>
<component name="ejb3" definition="org.objectweb.jonasALaCarte.EJB">
      <virtual-node name="node3" />
   </component>
   <component name="web1" definition="org.objectweb.jonasALaCarte.Web">
      <virtual-node name="node4" />
   </component>
   <component name="web2" definition="org.objectweb.jonasALaCarte.Web">
      <virtual-node name="node5" />
   </component>
   <component name="security" definition="org.objectweb.jonasALaCarte.Security">
      <virtual-node name="node6" />
   </component>
   <component name="transaction1" definition="org.objectweb.jonasALaCarte.TransactionManager">
      <virtual-node name="node1" />
   </component>
   <component name="transaction2" definition="org.objectweb.jonasALaCarte.TransactionManager">
      <virtual-node name="node2" />
   </component>
   <component name="registry" definition="org.objectweb.jonasALaCarte.Registry">
      <virtual-node name="node7" />
   </component>
   <component name="database" definition="org.objectweb.jonasALaCarte.DB">
      <virtual-node name="node7" />
   </component>
   <component name="configurator" definition="org.objectweb.jonasALaCarte.Configurator">
      <virtual-node name="node1" />
   </component>

   <!-- List of bindings -->
   <binding client="initiator.service-web" server="web.service" />
   <binding client="initiator.service-ejb" server="ejb.service" />
   <binding client="initiator.service-tm" server="transaction.service" />
   <binding client="initiator.service-reg" server="registry.service"/>
   <binding client="initiator.service-sec" server="security.service" />
   <binding client="initiator.service-ear" server="EAR.service" />
   <binding client="web1.confItf" server="configurator.confItf" />
   <binding client="Apache.modjk1" server="web1.modjk" />
   <binding client="web1.cmi" server="ejb1.cmi" />
   <binding client="web1.cmi" server="ejb2.cmi" />
   <binding client="web1.cmi" server="ejb3.cmi" />
   <--idem pour web2 -->
   <binding client="initiator.confItf" server="configurator.confItf" />
   <binding client="security.confItf" server="configurator.confItf" />
   <binding client="registry.confItf" server="configurator.confItf" />
   <binding client="ear.confItf" server="configurator.confItf" />
   <binding client="transaction.confItf" server="configurator.confItf" />
   <binding client="ejb1.transactItf" server="transaction.transactItf" />
   <binding client="ejb1.tm" server="jtm.tm" />
   <binding client="ejb1.db" server="database.db1" />
   <-- idem pour ejb2 -->
   <binding client="web.sec" server="security.sec" />
   <binding client="ear.deploy-web server="web.deploy" />
   <binding client="ear.deploy-ejb" server="ejb.deploy" />
</definition>
```

FIG. 7.11 – *Extrait du fichier ADL de JonasALaCarte en grappe.*

Par rapport au déploiement actuel des grappes de serveurs « libres », notre solution se distingue par les propriétés suivantes.

- La configuration des serveurs J2EE est simplifiée grâce à la représentation uniforme en composants Fractal des différents éléments du système. En effet, la configuration d'une grappe est simplement une configuration particulière du serveur où les composants sont distribués et dupliqués sur des JVM différentes.
- L'administrateur utilise les mêmes outils pour la configuration de la grappe que pour la configuration centralisée du serveur. De la même manière, il suffit de décrire l'architecture désirée et la configuration des parties du système dans des fichiers ADL. Les étapes du déploiement sont automatisées par le système de déploiement.
- L'unité de déploiement est plus fine — le service — ce qui permet une meilleure gestion des ressources.
- Il est possible de définir plusieurs politiques de déploiement en définissant des domaines de déploiement au niveau de la carte.

7.4.2 Mise à jour dynamique de code

La mise à jour des services dans les serveurs « libres » est une propriété très importante car le code provient de contributeurs différents et évolue, par conséquent, d'une manière indépendante. Dans JOnAS, la mise à jour d'un service nécessite l'empaquetage d'une nouvelle version de tout le serveur puisque les services ne sont pas archivés dans des paquetages différents. Il faut que tout le serveur s'arrête pour remplacer l'ancien code du service par le nouveau. En effet, les services sont chargés par le même *class loader* et il n'est pas possible de créer deux versions d'un même service.

Dans notre travail, nous avons isolé les services dans des composants séparés et toutes les communications entre les services passent par des interfaces explicites. Nous avons adopté une approche simple où le code des interfaces et le code des services sont archivés dans des paquetages différents et sont chargés par des *class loaders* différents. Ceci permet de ne redémarrer que le code du service à mettre à jour et de contrôler l'impact sur les autres services.

Comme exemple de reconfiguration d'implantation, nous avons traité la mise à jour du code du service Web qui encapsule le code de Tomcat. Nous avons considéré une architecture J2EE complète où un serveur Apache HTTP est en amont pour recevoir les requêtes HTTP, un serveur d'applications JOnAS contenant le serveur Web et une base de données. Le système, entièrement « fractalisé », est déployé à partir d'une description ADL. Après le démarrage du système, nous proposons de passer de la version 4.x de Tomcat à la version 5.x.

Le service Web contenant la version de Tomcat 5.x est encapsulé dans un composant Fractal comme tous les services de JonasALaCarte. La reconfiguration consiste alors à remplacer l'ancien composant Web par le nouveau. C'est une application directe du mécanisme de reconfiguration présenté en 7.3.3. Toutefois, il faut tenir compte des applications en cours d'exécution et des requêtes clientes qui sont en cours. En effet, il est nécessaire de tenir compte de deux éléments principaux :

- les appels entrant durant le temps de la reconfiguration,
- l'état des composants Web (JSP et Servlets) qu'il faut retrouver après la reconfiguration.

Concernant les appels entrants, un module inséré dans Apache permet de les stocker dans une file le temps de la reconfiguration. L'activation de cette fonctionalité se fait au niveau d'une interface Fractal du composant encapsulant Apache. Pour retrouver le conteneur Web dans un

état cohérent, plusieurs techniques sont possibles. Après le blocage des appels entrants, on peut attendre le temps nécessaire pour l'exécution d'une application en cours avant d'effectuer l'opération de reconfiguration. Si ce temps est trop long, c'est le cas des applications de commerce électronique nécessitant des transactions coûteuses, il est possible de sauvegarder l'état des composants Web en dupliquant les données des sessions HTTP au niveau de la carte[6]. Dans notre expérience, nous avons choisi la première solution.

Grâce à ces techniques assurant la cohérence du système après une opération de reconfiguration, les étapes de la reconfiguration d'implantation sont effectuées exactement comme celles présentées en 7.3.3.

- L'administrateur ou un composant gestionnaire appelle la méthode `Component update (Code nouvelleVersionTomcat, PackageId newCodePackageId)` sur le composant encapsulant le service Web.
- L'ancien composant est isolé par le gestionnaire, c'est à dire qu'il est (1) arrêté, (2) les liaisons avec les autres services sont coupées et (3) il est enlevé du composite qui le contient (en général, ce composite est JOnAS). La figure 7.12 présente un extrait du code réalisant cette isolation. Si le conteneur Web est partagé par plusieurs composites (les composants appelés `parents` dans le code), il doit être enlevé de tous les composites. L'*Installer* se charge de désinstaller le paquet de l'ancien code et d'installer le paquetage du nouveau service Web. La figure 7.13 montre le code correspondant à cette opération. Nous utilisons des services offerts par le `ShellService` d'OSGi pour récupérer, installer ou désinstaller des paquetages.
- Le *Generic Factory* effectue les étapes nécessaires au déploiement du nouveau composant Web et retourne la référence *componentId* du nouveau composant Web installé. Ce composant est par la suite, ajouté aux composites encapsulant le service Web et les liaisons sont établies avec les autres services. La figure 7.14 présente le code associé à cette phase.
- Le composant Apache est activé pour bloquer les appels entrants.

Le gestionnaire démarre alors le nouveau service Web ainsi que les applications Web. Le composant Apache débloque l'arrivée des requêtes clientes.

7.4.3 Réparation automatique des pannes

Nous avons utilisé la boucle de commande de JonasALaCarte, représentée dans la figure 7.9, pour introduire des politiques d'administration autonome. Nous considérons la réparation de pannes logicielles ou matérielles et nous ne considérons que les pannes franches. Nous avons choisi une politique de *redémarrage partiel* ou *micro-reboot* du serveur pour réparer des pannes logicielles ou des pannes matérielles. La politique consiste à redémarrer seulement les services et composants impactés par une faute. Cette politique est similaire à celle de JAGR [CKZ+03] bien que, dans JAGR, le niveau de la reconfiguration soit l'application et pas l'intergiciel.

Nous n'avons considéré dans ce travail que les pannes franches. Nous considérons fautif tout composant logiciel ou matériel (machine) ne répondant pas à une requête au bout d'un temps configurable. Pour détecter les pannes, nous avons implanté deux types de capteurs, ceux qui détectent les pannes matérielles et ceux qui détectent les pannes logicielles. La détection de pannes matérielles est effectuée en envoyant régulièrement des requêtes (*ping*) aux machines à administrer pour vérifier qu'elles sont toujours opérationnelles. Une machine qui ne répond pas

6. La duplication des données des sessions HTTP est une technique utilisée dans les grappes J2EE.

```
// Stop the Web component
LifeCycleController lcc = (LifeCycleController) this.webComponent
                                        .getFcInterface("lifecycle-controller");
      try {
            lcc.stopFc();
      } catch (IllegalLifeCycleException e) {
            e.printStackTrace();
       }
// Unbind from other services
BindingController bc = (BindingController) this.webComponent
                                    .getFcInterface("binding-controller");
      try {
            bc.unbindFc("configurator");
            bc.unbindFc("jmx");
            bc.unbindFc("sec");
      } catch (IllegalBindingException e) {
            e.printStackTrace();
      } catch (IllegalLifeCycleException e) {
            e.printStackTrace();
      }
// Remove from JonasALaCarte composites
for (int i = 0; i < parents.length; i++) {
      ContentController cc = (ContentController) parents[i]
                                    .getFcInterface("content-controller");
      LifeCycleController compositeLcc = (LifeCycleController) parents[i]
                                    .getFcInterface("lifecycle-controller");
      try {
            cc.removeFcSubComponent(webComponent);
      } catch (IllegalContentException e) {
             e.printStackTrace();
      } catch (IllegalLifeCycleException e) {
            e.printStackTrace();
      }
}
```

FIG. 7.12 – *Isolation de l'ancien code du conteneur Web.*

```
// uninstall the old Tomcat bundle
       try {
           _this_bundleCtx.getBundle(Tomcat4xPackageId).uninstall();
       } catch (BundleException e) {
           e.printStackTrace();
       }

// install the new Tomcat bundle
       ShellService shellService = null;
       ServiceReference[] services = null;
       try {
           services = _this_bundleCtx.getServiceReferences(
                   "org.ungoverned.osgi.service.shell.ShellService", null);
       } catch (InvalidSyntaxException e) {
           e.printStackTrace();
       }

       if (services != null && services.length > 0) {
           Object shellObj = _this_bundleCtx.getService(services[0]);
           if (shellObj != null && shellObj instanceof ShellService) {
               shellService = (ShellService) shellObj;
               try {
                   shellService.executeCommand(
                           "start file:/home/abdellat/Repository/WebService5x.jar",
                           System.out, System.err);
               } catch (Exception e) {
                   e.printStackTrace();
               }
           }
       }
```

FIG. 7.13 – *Gestion de paquetages pour la mise à jour du service Web.*

```
// create the new Tomcat component
      Object params = null;
      services = null;
      try {
      services = _this_bundleCtx.getServiceReferences(
                  "java.lang.ClassLoader", "(ClassLoader=web)");
      } catch (InvalidSyntaxException e) {
            e.printStackTrace();
      }
      if (services != null && services.length > 0) {
            Object clObj = _this_bundleCtx.getService(services[0]);
            if (clObj != null && clObj instanceof ClassLoader) {
             ClassLoader cl = (ClassLoader) clObj;
            System.err
                        .println("ClassLoader for a current component: " + cl);
            params = new Object[] { cl, webControllerDesc };
            }
      }
      try {
            this.webComponent = (Fractal.getGenericFactory(boot))
                  .newFcInstance(webType, params, webContentDesc);
      } catch (InstantiationException e) {
            e.printStackTrace();
      } catch (NoSuchInterfaceException e) {
            e.printStackTrace();
      }

      try {
            for (int i = 0; i < parents.length; i++) {
                  ContentController cc = (ContentController) parents[i]
                              .getFcInterface("content-controller");
                   try {
                          cc.addFcSubComponent(webComponent);
                    } catch (IllegalContentException e) {
                          e.printStackTrace();
                   } catch (IllegalLifeCycleException e) {
                          e.printStackTrace();
                  }
            }


            Object obj = this.webComponent.getFcInterface("service");
            NameController nc = (NameController) this.webComponent
                    .getFcInterface("name-controller");
            nc.setFcName("web");
            // Binding establishement
            BindingController bc = (BindingController) this.webComponent
                    .getFcInterface("binding-controller");
            Object srvItf = jpComponent.getFcInterface("configurator");
            bc.bindFc("configurator", srvItf);
            srvItf = jmxComponent.getFcInterface("jmx");
            bc.bindFc("jmx", srvItf);
            srvItf = secComponent.getFcInterface("sec");
            bc.bindFc("sec", srvItf);
            // start the Web service
            LifeCycleController lcc = (LifeCycleController) this.webComponent
                    .getFcInterface("lifecycle-controller");
            try {
                lcc.startFc();
            } catch (IllegalLifeCycleException e) {
                e.printStackTrace();
            }
```

FIG. 7.14 – *Création du nouveau composant du conteneur Web.*

est considérée en panne. La panne peut être due à une défaillance matérielle de la machine ou à un problème réseau isolant la machine du reste du système.

Comme exemple de panne logicielle, nous avons considéré la panne du conteneur Web. Pour détecter cette panne, nous avons implanté une Servlet « légère », c'est à dire avec un code fonctionnel minimal dont le seul rôle est d'acquitter la réception des requêtes reçues. Un capteur a pour rôle d'interroger régulièrement la Servlet s'exécutant dans le conteneur Web. Si au bout d'un temps configurable, la Servlet n'acquitte pas les requêtes, le service Web qui l'héberge est considéré en panne. En effet, c'est le conteneur Web qui représente l'environnement d'exécution de la Servlet. Si celle-ci ne répond pas et tenant compte du fait que son code est minimal (réponse à une requête), il y a une forte probabilité pour que le serveur Web ait un problème (en état de saturation ou dans un état d'interblocage) l'empêchant de recevoir ou d'envoyer correctement les requêtes.

Nous avons implanté un composant gestionnaire mettant en œuvre la politique de redémarrage partiel (*micro-reboot*). Lorsque le gestionnaire reçoit l'événement de la panne d'une machine ou d'un service, il demande aux actionneurs de redémarrer les composants fautifs. Considérons le cas d'une panne logicielle. Lorsque le capteur ne reçoit pas d'acquittement de la Servlet s'exécutant dans le service Web, il envoie un événement au gestionnaire qui établit les opérations suivantes.

1. Activation d'une interface du composant Apache pour stocker dans une liste les appels entrants durant le temps de la reconfiguration.
2. Redémarrage du composant Web fautif.
3. Débloquage des requêtes au niveau du composant Apache.

```
<definition name="org.objectweb.jonasALaCarte.Probe">
        <!-- probe component -->
        <interface name="starter" role="server"
                   signature="org.objectweb.jonasALaCarte.StarterItf"/>
        <interface name="manager" role="client"
                   signature="org.objectweb.jonasALaCarte.manager.ManagerItf"/>
        <content class="org.objectweb.jonasALaCarte.probe.Probe"/>
        <attributes signature="org.objectweb.jonasALaCarte.probe.ProbeConf">
                <attribute name="Timeout"
                           value="1000"/>
                <attribute name="PingFrequency"
                           value="15000"/>
                <attribute name="Address"
                           value="http://localhost:9000/jonasAdmin/"/>
        </attributes>
</definition>
```

FIG. 7.15 – *Extrait du fichier ADL pour la configuration du capteur.*

La figure 7.15 présente la configuration du capteur. Le temps d'attente d'une requête, `Timeout`, est mis par défaut à 1 sec avant de juger la Servlet fautive. L'envoi d'une requête, `PingFrequency` se fait toutes les 15 secondes. Enfin, le fichier ADL précise l'adresse de la Servlet qu'il faut régulièrement interroger. Le code du capteur est très simple : il consiste en un client HTTP interrogeant régulièrement une Servlet à distance. Le capteur lève une exception s'il n'obtient pas de réponse.

Le code représentant la gestion de la panne est représenté dans la figure 7.16. Le composant représentant la partie de JOnAS « fractalisé » est appelé « middleware ». Pour récupérer une référence vers ce composant, nous utilisons une méthode utilitaire `getComponentByName` qui

retourne la référence d'un composant à partir de son nom. Nous avons ajouté un contrôleur `redeployment-controller` au composant « middleware » dont le rôle est de redémarrer le service Web en appelant la méthode `restartWebService()`. Cette méthode arrête et redémarre le service.

La panne du service Web peut avoir lieu pendant que des applications s'exécutent. Si les données des sessions HTTP sont régulièrement dupliquées (soit au niveau de la carte, soit au niveau d'un autre conteneur comme c'est le cas dans les grappes), c'est la dernière copie qui est récupérée. Sinon, la requête en cours pendant la panne n'est pas traitée.

```
public void handleError(String errorMsg) {
        if (errorMsg.equals (WebError)
        try {
                // get the middleware component from JonasALaCarte system
                SuperController superController =  (SuperController)
                                comp.getFcInterface("super-controller");
                Component cmp = superController.getFcSuperComponents()[0];
                Component middleware = getComponentByName(Fractal.getContentController(cmp),
                                "middleware");
                WebServiceRedeploymentController trc = (WebServiceRedeploymentController)
                                middleware.getFcInterface("redeployment-controller");
                // Restart the Web service
                trc.restartWebService();
        } catch (NoSuchInterfaceException e) {
                System.out.println("Unable to find the parent composite");
                e.printStackTrace(); }
        }
}
```

FIG. 7.16 – *Extrait du code de l'actionneur.*

7.5 Synthèse

Nous avons présenté, dans ce chapitre, un système d'administration permettant d'automatiser la configuration, le déploiement et l'adaptation des systèmes distribués. Ce système d'administration utilise un canevas appelé Jade dont les éléments d'architecture forment des boucles de commande dans le but de construire des systèmes autonomes. De plus, Jade dispose d'une structure réflexive basée sur l'utilisation d'un composant de représentation du système. Tous les éléments du système d'administration et du système administré sont des composants Fractal.

Nous avons contribué à la construction d'un système de déploiement et de mécanismes de reconfiguration d'implantation pour Jade. Nous avons utilisé le système d'administration obtenu pour le déploiement et la reconfiguration de la version « fractalisée » de JOnAS. Pour illustrer les fonctions d'administration introduites, nous avons décrit le déploiement dans une grappe, la mise à jour de code d'un service et la réparation automatique de pannes. Pour chaque opération de reconfiguration, en plus de l'utilisation des mécanismes de reconfiguration de Jade, il était nécessaire de tenir compte de l'état du système et des requêtes en cours pour maintenir la cohérence du système adapté. Nous avons fait ce travail pour le service Web de JOnAS, mais il est nécessaire de le faire pour les autres services de JOnAS.

Dans ce chapitre, nous avons montré qu'il était possible d'automatiser le déploiement sur une grappe d'un serveur J2EE, réduisant le rôle de l'administrateur à la description de l'architecture souhaitée dans un fichier ADL. Nous avons également montré que la mise à jour du code des services était réalisable sans redémarrer tout le serveur et enfin, qu'il était possible d'implanter des politiques d'administration autonomes comme la réparation automatique de pannes.

Chapitre 8

Évaluation

Sommaire

Nous présentons dans ce chapitre une évaluation qualitative ainsi qu'une évaluation quantitative de notre travail. Pour les aspects qualitatifs, nous discutons des trois points suivants : la compatibilité de JonasALaCarte avec les standards de l'administration J2EE, la pertinence de la politique du redémarrage partiel pour l'auto-réparation des pannes dans les serveurs J2EE et une comparaison de notre travail avec les travaux de l'état de l'art. Pour les aspects quantitatifs, nous avons étudié le coût du travail de réingénierie du point de vue du développeur de JOnAS, le gain en temps de configuration et de déploiement du point de vue de l'administrateur et, enfin, une évaluation des performances de JonasALaCarte comparées à celles de JOnAS.

8.1 Aspects qualitatifs

8.1.1 Compatibilité avec les standards

Notre travail de réingénierie est compatible avec les standards d'administration J2EE (JSR77 et JSR88) implantés dans JOnAS. Rappellons que JSR77 décrit un modèle d'informations que tous les serveurs d'applications doivent exposer grâce à une instrumentation JMX. Dans JonasALaCarte, nous avons procédé incrémentalement. Nous avons gardé l'instrumentation JMX en gardant les interfaces des MBeans du serveur. D'autre part, nous avons ajouté à coté les interfaces Fractal. Par conséquent, les mêmes outils (JonasAdmin dans le cas de JOnAS) compatibles avec le standard permettent d'explorer JonasALaCarte selon l'arborescence du modèle JSR77.

Le standard JSR88 définit la méthode de déploiement des applications J2EE. D'une part, il spécifie le format des paquetages (modules) et des descripteurs de déploiement des applications

J2EE. D'autre part, il définit les interfaces que doit implanter le serveur J2EE pour déployer les applications. Le but est de permettre l'interopérabilité entre les applications éventuellement développées par des tierces personnes et les serveurs d'applications provenant de fournisseurs différents. Dans cette thèse, nous nous sommes concentrés sur l'empaquetage et le déploiement du serveur. Nous avons maintenu l'empaquetage des applications et le format des descripteurs de déploiement comme définis par JSR88. De plus, comme nous avons procédé d'une manière incrémentale, nous avons également gardé les interfaces de déploiement des applications au niveau du serveur. Les mêmes applications déployées dans JOnAS peuvent être déployées dans JonasALaCarte en utilisant des outils compatibles avec JSR 88 (e.g. ceux intégrés dans Eclipse).

8.1.2 Pertinence du redémarrage partiel

La politique du redémarrage partiel (*micro-reboot*) consiste à redémarrer uniquement la partie fautive d'un système pour réparer la panne. Nous avons utilisé cette technique pour réparer la panne d'un service dans JonasALaCarte. Dans cette section, nous discutons de la pertinence de cette technique. Nous souhaitons savoir si (1) elle permet de réparer les pannes et si (2) elle est plus efficace que le redémarrage de tout le serveur.

La politique du redémarrage partiel pour la réparation des pannes logicielles est une technique utilisée dans JAGR [CKZ+03]. JAGR est une extension du serveur JBoss intégrant un système d'administration dont le but est de réparer les pannes intervenant au niveau applicatif. JAGR offre les sondes nécessaires à la détection des pannes des EJB et implante les interfaces nécessaires au redémarrage des EJB fautifs. Des entités de gestion analysent les exceptions remontées et décident, en corrélant les exceptions entre elles, de l'occurence d'une panne et des EJB fautifs. Ces EJB sont alors redémarrés. JAGR montre que cette politique est intéressante puisque le serveur retrouve son fonctionnement « normal » au bout d'un temps raisonnable qui est de l'ordre de la seconde.

Nous avons essayé d'appliquer cette approche pour réparer la panne au niveau de l'intergiciel. En effet, lorsqu'une panne est détectée, au lieu de redémarrer tout le serveur comme c'est le cas dans JOnAS actuellement, nous redémarrons uniquement le service fautif. Néanmoins, notre implantation actuelle des sondes est primitive. En effet, pour détecter la panne du service Web, nous avons implanté une sonde sous forme de Servlet acquittant les requêtes clientes. Nous ne considérons que les pannes franches ; si la Servlet ne répond pas alors le service Web est considéré en panne. Nous nous sommes permis de faire cette hypothèse car la Servlet est « minimale ». Si elle ne répond pas, alors il est fort probable que le problème vienne du conteneur Web qui se charge de la communication. Néanmoins, il est possible que la faute soit liée à d'autres facteurs comme la panne du réseau ou la saturation temporaire de la machine hébergeant la Servlet. La faute est simulée dans notre expérience par un simple arrêt de la Servlet. Il est donc normal que le service retrouve son fonctionnement normal après son redémarrage. Nous avons besoin de tester d'autres types de fautes « réelles ». Il est probable qu'elles laisseront le service dans un état instable même après son redémarrage ou que la faute sera propagée au niveau d'autres services. Pour ce faire, nous pensons qu'il est nécessaire d'intégrer des injecteurs et des détecteurs de pannes plus évolués, comme ceux de JAGR, analysant plus finement les exceptions. Les injecteurs et les détecteurs de pannes sont facilement intégrables dans JonasALaCarte sous forme de composants Fractal. En effet, les architectures du système administré et du système d'administration sont suffiamment modulaires pour intégrer différents algorithmes et politiques d'administration.

Sur le plan des performances, le redémarrage d'un service Web fautif est plus intéressant que celui du serveur. Deux facteurs expliquent ce gain de performances. Tout d'abord, le temps du redémarrage du service, incluant le rétablissement des liaisons et la récupération de l'état des

applications, est approximativement de 3 secondes. Le redémarrage de tout JOnAS nécessite approximativement 40 secondes. D'autre part, les services non impactés par la faute maintiennent leurs états et leurs données. Notons que la panne du service est cachée aux utilisateurs Web puisque nous sauvegardons les requêtes entrantes et l'état des applications Web en cours d'exécution.

8.1.3 Comparaison avec les travaux de l'état de l'art

Plusieurs travaux sont proches de ceux décrits dans cette thèse. Nous les classons en quatre catégories : les systèmes d'administration fondés sur l'architecture, la gestion des reconfigurations pour les composants, l'administration des serveurs J2EE et l'administration des systèmes distribués d'une manière générale.

Systèmes d'administration fondés sur l'architecture

Notons tout d'abord que, contrairement aux systèmes étudiés dans l'état de l'art, notre système d'adminisration ne cible pas une fonction particulière de l'administration, mais couvre à la fois la configuration, le déploiement et la reconfiguration d'implantation et de structure. De plus, il traite le cycle de vie des composants de l'empaquetage à l'exécution contrairement aux autres systèmes qui se focalisent soit sur la gestion du cycle de vie des paquets soit sur les composants à l'exécution. Dans la suite de cette section, nous faisons une comparaison détaillée de notre système avec les trois systèmes présentés dans le chapitre 2 : SmartFrog, Nix et Rainbow.

Nous avons vu que SmartFrog 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 complexes. Dans notre travail, l'ADL Fractal permet de décrire des architectures distribuées et d'automatiser le déploiement. Néanmoins, nous ne disposons pas, pour le moment, d'outils pour la description de *workflow*. Les étapes du processus de déploiement sont actuellement séquentielles. En revanche, l'ADL Fractal étant extensible, il est possible de définir des *worflows* pour le déploiement. Par ailleurs, contrairement à notre travail, SmartFrog ne gère pas le dépôt des paquetages ; les paquetages sont supposés être installés sur les machines cibles. De plus, SmartFrog ne gère pas les cycle de vie des paquetages installés. Par conséquent, les opérations de reconfiguration d'implantation sont limitées.

Nix se focalise sur l'installation des paquetages et la gestion de versions de code. Nous nous sommes inspirés de Nix pour la gestion des paquetages, notamment le mécanisme de ramasse-miettes. Néanmoins, Nix est limité à un environnement centralisé et spécifique (Linux). Notre système de déploiement permet de gèrer le cycle de vie du code de l'empaquetage à l'exécution dans des environnements distribués. De plus, les abstractions de déploiement que nous avons définies sont génériques et peuvent être implantées dans d'autres langages que Java. Comparés à l'ADL Fractal, les outils proposés par Nix sont très complexes (pour un administrateur). En effet, le langage de description proposé n'est pas typé et son utilisation est réduite à des scripts Shell. Par ailleurs, Nix ne se base pas sur un modèle de composants évolué permettant de rendre l'architecture du système administré explicite à l'exécution. Il n'est donc pas conçu pour la reconfiguration de structure à l'exécution.

Rainbow partage un des objectifs de notre système d'administration, celui d'automatiser la reconfiguration à l'exécution. Il se base, comme dans JonasALaCarte, sur une représentation du système administré. Cette représentation utilise un modèle de composants hiérarchique. Dans JonasALaCarte, la carte représentant le système administré reproduit exactement l'architecture et la configuration de ce système. En effet, dans notre approche, le système d'administration

et la carte sont construits à l'aide du même modèle de composants. Dans JonasALaCarte, l'administrateur a une vue qui reflète l'architecture réelle du système administré, tandis que dans Rainbow une couche de correspondance est interposée pour représenter le système administré sous forme de composants. Dans JonasALaCarte, la génération de la carte ainsi que les outils permettant la correspondance avec le système administré sont indépendantes de la sémantique de l'application. En effet, la carte reproduit la création des composants et les contrôleurs et pas les interfaces fonctionnelles. Dans Rainbow, la création des composants correspondant aux éléments du système d'application nécessite une analyse du code fonctionnel. Par ailleurs, dans la carte, nous représentons également les composants du système d'administration lui-même qui est formé de composants Fractal. Ainsi, contrairement à Rainbow, le système d'administration est administrable en bénéficiant du même modèle de composants et des outils de déploiement et de reconfiguration. En effet, les actuateurs, les capteurs et les gestionnaires du système d'administration sont des composants Fractal configurables, déployables et reconfigurables de la même manière que les systèmes administrés.

Gestion des reconfigurations d'implantation pour les composants

Parmi les modèles de composants traitant les reconfigurations d'implantation, citons JPloy [LvdH04] et SOFA [HT03]. Ces deux modèles utilisent des manipulations de bytecode pour garantir que les différentes versions d'une même classe sont chargées avec des noms différents. Ces noms sont générés à partir de fichiers de descriptions (comparables à des descriptions ADL) qui spécifient les versions des classes utilisées. L'avantage de cette approche est qu'elle n'engendre aucun surcoût à l'exécution. En revanche, elle rend impossible l'utilisation de certaines méthodes du langage Java. Il n'est, par exemple, pas possible d'utiliser certaines méthodes de construction de classes par réflexion (e.g. `Class.forName(String name)`).

Deux autres travaux de gestion de reconfiguration d'implantation ont été menés pour le modèle Fractal. Dans [KLQS04], une gestion modulaire de chargement de classes des composants Fractal est assurée grâce à l'intégration du *module loader* de l'implantation Oscar. Néanmoins, cette solution ne traite pas la gestion des paquetages et le déploiement distribué. Dans [CDD04], FROGI propose un déploiement de composants Fractal sur OSGi. Pour ce faire, les composants Fractal sont encapsulés dans des *bundles* et sont déployés utilisant OSGi. Néanmoins, la communication entre deux composants chargés par deux *class loaders* passe par une communication entre les services OSGi et n'utilise pas les liaisons Fractal. Dans notre solution, l'infrastructure de déploiement est complètement cachée à l'application Fractal lors de l'exécution et ne casse pas le modèle, notamment les liaisons Fractal.

Administration de serveurs J2EE

L'administration des serveurs d'applications J2EE (JOnAS, Geronimo et JBoss) est basée sur JMX pour l'instrumentation du serveur. Cependant, 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 qui ne permet pas d'exprimer des architetcures explicites entre les MBeans. 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.

Par ailleurs, plusieurs travaux ont pour but d'étendre les serveurs d'applications existants pour introduire des gestions des pannes et des performances. Nous avons déjà présenté JAGR qui est une extension de JBoss pour réparer les pannes des applications J2EE s'exécutant dans le serveur. Par ailleurs, un travail similaire au notre sur JOnAS utilise la plateforme OSGi [onD]

pour le déploiement et la mise à jour de code des services. Pour ce faire, les services de JOnAS sont encapsulés dans des *bundles* séparés et les dépendances entre eux sont exprimées dans des fichiers *manifest*. Comme dans notre travail, la plateforme OSGi se charge de résoudre les dépendances et facilite la mise à jour de code. Néanmoins, la solution malgré ses avantages, a les mêmes limitations que OSGi. Elle n'offre pas de moyens pour décrire l'architecture et la configuration distribuée du système. Elle ne permet également pas d'automatiser le déploiement distribué à partir d'une description de l'architecture. Nous réalisons dans JonasALaCarte cet objectif à l'aide de fichiers ADL. Par ailleurs, dans JonasALaCarte, les dépendances entre services (modélisées sous forme de liaisons Fractal) sont facilement adaptables et peuvent facilement être réparties sur des machines distribuées (à l'aide de Fractal RMI). De plus, JonasALaCarte est basé sur un modèle de composants modulaires qui facilite la mise en place des opérations de reconfiguration de structure et l'adaptation du système aux pannes. En résumé, nous montrons dans cette thèse que l'association de Fractal et d'OSGi permet de couvrir les besoins de reconfiguration d'implantation et de structure.

Par ailleurs, une équipe de recherche de Sun s'intéresse au passage à l'échelle des applications J2EE [JDJ+06]. Ce travail part du constat que la machine virtuelle Java limite les performances des applications J2EE. En effet, la JVM ne propose pas de moyens pour contrôler les ressources utilisées comme le CPU. De plus, les mécanismes de ramasse-miettes peuvent être longs et limiter les performances. Déployer les serveurs sur des grappes et dédier les serveurs à une seule application J2EE ne résoud pas forcément le problème de performances induit par la JVM. La solution proposée consiste à modifier la machine virtuelle Java pour améliorer l'*isolation* des ressources utilisées par les applications. Dans notre travail, nous isolons l'intergiciel dans des paquets et des composants différents pour faciliter le déploiement et la mise à jour du code. Nous pensons que l'isolation des services dans des unités de déploiement séparées améliore la gestion des ressources puisque nous pouvons adopter un déploiement incrémentale et à la demande. Néanmoins, nous n'avons pas étudié de prêt l'aspect gestion de ressources, notre priorité étant l'amélioration du temps de la configuration et du déploiement, la mise à jour de code et la gestion des pannes.

Administration des systèmes distribués

Il existe plusieurs travaux autour des grappes et des environnements distribués à grande échelle. La majorité [ADZ00, AFF+01, FCC+03, CIG+03] traite plutôt l'aspect gestion des ressources. Dans cette thèse, nous avons considéré les grappes comme exemple d'environnement distribué où les machines sont homogènes et dédiées à l'hébergement et à l'exécution des serveurs J2EE. Par conséquent, le problème d'allocation des ressources ne s'est pas vraiment posé. Néanmoins, dans un travail en cours, nous considérons des environnements plus hétérogènes et à plus grande échelle, comme les grilles. Pour ce faire, nous avons étendu l'ADL pour exprimer les contraintes et les ressources désirées. L'adoption des services comme unités de déploiement permet d'exprimer des contraintes de ressources à grain relativement fin. Néanmoins, le déploiement et la reconfiguration des serveurs J2EE à l'échelle des grilles reste encore un sujet de recherche ouvert.

8.2 Aspects quantitatifs

Dans cette section, nous commençons par présenter une étude quantitative du travail de réingénierie de JOnAS du point de vue du développeur JOnAS et du point de vue de l'administrateur. Ensuite, nous montrons le surcoût sur les performances du serveur.

8.2.1 Point de vue du développeur JOnAS

Le travail de réingénierie dans JOnAS (« fractalisation » et empaquetage des services) a un coût très raisonable pour la taille du serveur (400 000 lignes de code) pour un ingénieur connaissant Fractal et le code de JOnAS. En effet, nous avons effectué la réingénierie de JOnAS sur deux versions différentes. La modification de JOnAS a nécessité trois mois pour la première version et seulement un mois pour la deuxième avec une meilleure connaissance du code de JOnAS et de l'implantation Julia de Fractal.

La plus grande difficulté de notre travail de réingénierie a été dans la recherche des dépendances entre les services de JOnAS, afin de remplacer les références statiques par des liaisons Fractal. En effet, cette recherche de dépendances a été effectuée manuellement. Dans la version actuelle de JonasALaCarte, nous avons traité les services les plus utilisés de JOnAS (les services EJB, Web, Ear, JMX, Security, Transaction, Registry et BD), ainsi que les entités d'administration se chargeant de la configuration et de la gestion du cycle de vie des services. Il reste à encapsuler les service JMS, Web services et Discovery, ce qui représente 10% de l'effort dépensé. Le service EJB a nécessité le plus de travail car c'est le service le plus complexe. Par ailleurs, il a de nombreuses dépendances avec d'autres parties de JOnAS. Pour encapsuler et configurer les services, nous avons ajouté approximativement 3300 lignes de code dans JOnAS. Le code représente onze classes Java correspondant aux composants de gestion et de configuration et 8 classes qui correspondent aux adaptateurs Fractal des services.

8.2.2 Point de vue de l'administrateur

Grâce à notre système d'administration, nous avons réduit d'un facteur significatif le temps passé par l'administrateur à configurer et déployer une grappe J2EE. En effet, nous avons estimé le temps passé dans la configuration et le déploiement d'une grappe contenant 2 duplicas d'un service Web et 2 duplicas du service EJB suite à plusieurs expériences menées par les membres du projet Sardes. Ce temps s'élève à quelques heures (le temps nécessaire pour écrire les fichiers ADL) au lieu d'un temps de l'ordre de la semaine comme c'est le cas actuellement pour les grappes JOnAS.

Ce gain est attribué à deux propriétés : l'unification de la configuration et l'automatisation du déploiement. En effet, la configuration de JonasALaCarte consiste à écrire les fichiers ADL de chaque service, ainsi que les fichiers décrivant l'architecture à déployer. Contrairement à JOnAS où les fichiers de configuration ont des syntaxes différentes, les fichiers de configuration sont uniformes et nécessitent la connaissance d'une seule syntaxe, celle de l'ADL Fractal. Partant de la description de l'architecture du système, l'installation et l'instanciation des composants sont effectués d'une manière automatique.

Notons qu'une grande partie du temps de l'administrateur est passée dans la vérification de la validité d'une configuration. Ce travail est effectué manuellement puisque nous ne disposons pas d'outils de vérification sémantique comme celui présenté dans [BLQ+05]. Dans le cas de JOnAS, la vérification de configuration comprend, par exemple, la vérification de la co-localisation de certains services (comme le service Web et le service Security), de la cohérence des paramètres de configuration et de l'existance de certains services (e.g. le service JMX doit être sur chaque machine).

8.2.3 Évaluation des performances

Nous présentons, dans cette section, une évaluation des performances de JonasALaCarte comparées à celles de JOnAS. La figure 8.1 présente l'environnement expérimental que nous

avons utilisé pour effectuer cette évaluation.

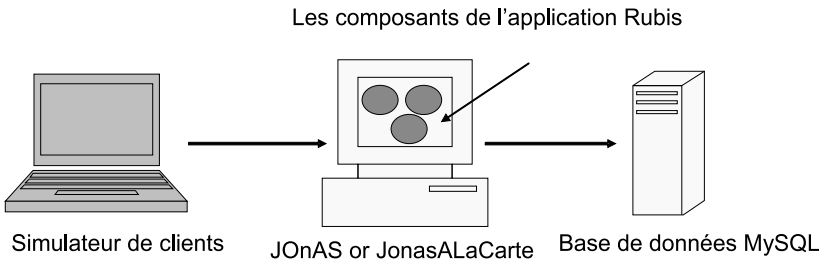

FIG. 8.1 – *Environnement expérimental pour l'évaluation de performances.*

Nous avons utilisé Rubis [CMZ02] qui est un *benchmark* pour les applications J2EE implantant un site de commerce éléctronique de type eBay. Rubis traite 26 types de requêtes Web comme l'enregistrement de nouveaux utilisateurs et l'achat ou la vente d'articles. Le traitement de ces requêtes est effectué grâce à un ensemble de Servlets et d'EJB avec et sans états ainsi que des EJB entités pour la persistance des données. Nous avons utilisé la version de Rubis qui fait intervenir les différents types de composants EJB et qui implante le patron le plus utilisé dans les applications J2EE (le patron *session-façade*). Rubis offre également un générateur de requêtes clientes simulant un ensemble d'utilisateurs Internet.

Le *benchmark* donne trois types d'informations : la charge CPU, la consommation mémoire et le nombre de requêtes clientes par seconde.

Environment matériel

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). La considération de la configuration centralisée est suffisante pour comparer les performances des deux serveurs. En effet, la distribution de JonasALaCarte sur des machines différentes ne modifie pas les communications par rapport à un serveur JOnAS distribué. Voici les caractéristiques des machines utilisées dans les expériences.

- 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 RAM.

Environnement logiciel

Les machines exécutent la version 2.4.26 du noyau Linux. Nous avons utilisé la version 4.4.0 de JOnAS, la version 4.1.3 de MySQL, la version 1.4.2 de RUBiS et la machine virtuelle Java JDK 1.4.1_01.

Résultats de l'évaluation

Comparé à JOnAS, JonasALaCarte ne présente pas de surcoût significatif en termes de consommation CPU, de mémoire et de nombre de requêtes par seconde. La figure 8.2 montre la consommation CPU de JonasALaCarte et la figure 8.3 présente celle de JOnAS. Notons que le *benchmark* distingue la consommation CPU du serveur HTTP (la courbe *frontend*), du conteneur Web (la courbe Servlets), du conteneur EJB (la courbe EJB) et de la base de données

(la courbe Database). Ne disposant pas de serveur HTTP (Apache par exemple) dans notre expériementation, la courbe associée est nulle. Les courbes associées aux conteneurs Web et EJB sont confondues car nous exécutons ces deux conteneurs au sein du même serveur sur une même machine. La courbe de consommation CPU de la base de données montre une consommation relativement faible qui ne dépasse pas les 10%. Ceci montre que le goulot d'étranglement n'est pas la base de données et que les mesures de la consommation CPU au niveau des serveurs J2EE sont significatives.

Nous constatons que la moyenne de la consommation CPU des deux serveurs est autour de 70%. Néanmoins, un ralentissement (80 secondes à la place de 40) est observé dans le démarrage de JonasALaCarte suite au changement des mécanismes de chargement de classes qui est plus lent avec l'intégration de OSGi. En revanche, le temps du démarrage d'un serveur est négligeable devant la durée de l'exécution du serveur une fois démarré.

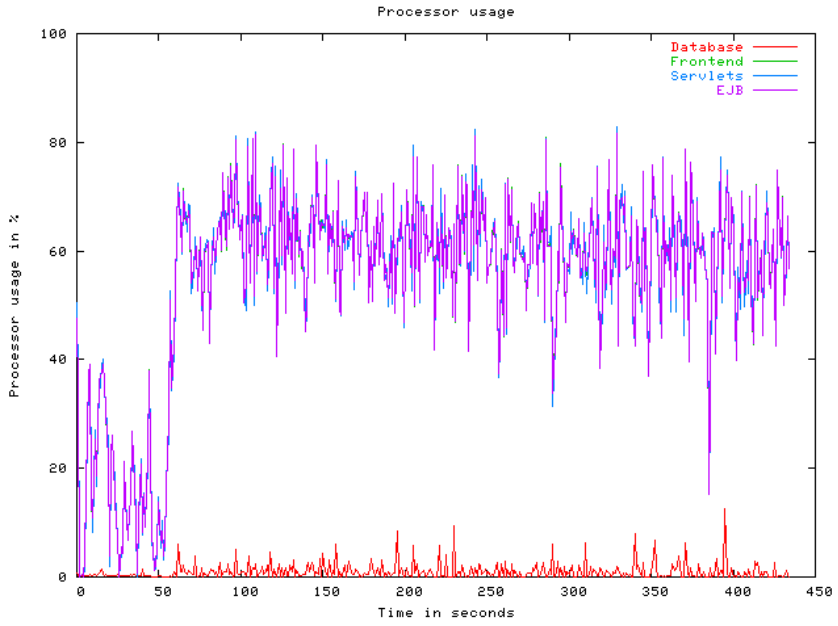

FIG. 8.2 – *Charge CPU de JonasALaCarte.*

Concernant la mémoire consommée, une moyenne calculée suite à 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 réingénierie. Néanmoins, pour un serveur d'applications généralement hébergé sur des machines puissantes, ce surcoût n'est pas significatif.

Enfin, nous avons enregistré le même nombre de requêtes par seconde (9 requêtes/s pour les 2 serveurs).

En conclusion, le travail de réingénierie qui est effectué à gros-grain (à la granularité des services) dans JOnAS induit un surcoût négligeable sur les performances du serveur. En effet, le nombre d'objets créés suite à la « fractalisation » est négligeable devant le nombre d'objets du serveur.

8.3 Synthèse

Nous avons présenté dans ce chapitre une étude qualitative et quantitative de notre contribution de thèse. Pour les aspects qualitatifs, nous avons montré que la « fractalisation » de

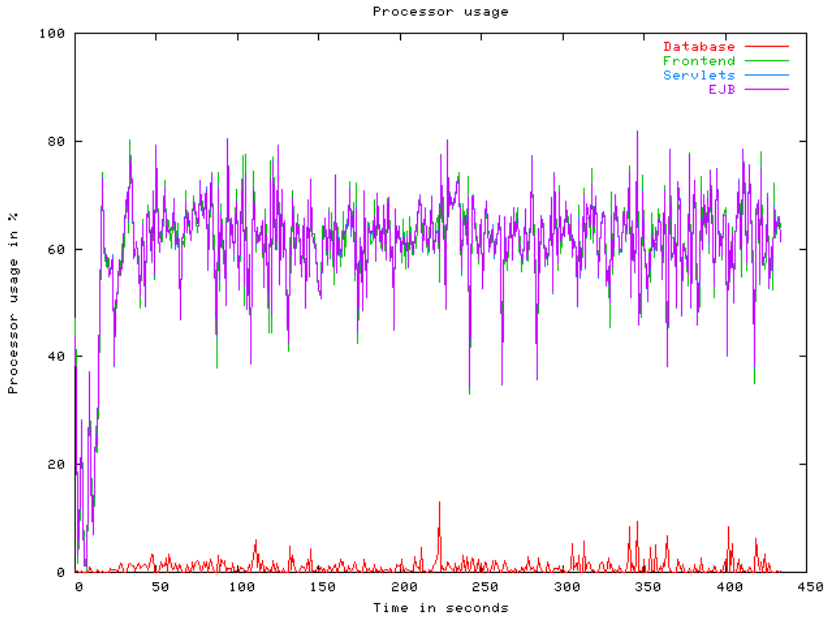

FIG. 8.3 – *Charge CPU de JOnAS.*

JOnAS ne viole pas les standards de l'administration J2EE. De plus, nous avons discuté de la pertinence du redémarrage partiel pour réparer les pannes du serveur. Enfin, une comparaison de notre travail avec les travaux de l'état de l'art montre que l'administration fondée sur l'architecture est pertinente pour automatiser le déploiement, la configuration et la tolérance des pannes. Néanmoins, nous devons enrichir notre travail pour traiter la gestion des ressources.

Par ailleurs, l'étude quantitative montre que le temps nécessaire au travail de réingénierie est raisonable pour la taille d'un serveur d'applications comme JOnAS. De plus, l'administration fondée sur l'architecture apporte un gain immédiat en termes de temps de configuration et de déploiement et ne dégrade pas les performances du serveur.

Chapitre 9

Conclusion

Dans ce chapitre qui constitue la conclusion de notre travail, nous commençons par rappeler la problématique de la thèse ; nous dressons ensuite un bilan des principaux apports de nos travaux et, enfin, nous en présentons les perspectives.

9.1 Rappel de la problématique

L'administration d'un système comprend sa configuration, son déploiement, son observation et son adaptation à l'exécution. Les opérations d'adaptation comprennent la réparation des pannes, la gestion des variations de performances, la gestion des ressources et la gestion de la sécurité.

La mise en œuvre des fonctions d'administration dans les systèmes modernes est une tâche difficile pour plusieurs raisons. Tout d'abord, les systèmes à administrer sont de plus en plus complexes. Leurs architectures ne sont en général pas suffisamment modulaires pour effectuer des opérations de reconfiguration à l'exécution. Par ailleurs, la distribution à grande échelle rend l'accès aux systèmes administrés plus difficile. Par conséquent, les systèmes d'administration sont aujourd'hui des systèmes complexes qui ont besoin d'être à leur tour administrés. En effet, ils ont besoin d'être configurés, déployés et adaptés suivant les événements extérieurs (comme les pannes) et suivant l'évolution des politiques d'administration. Enfin, avec le coût croissant de l'administration et la complexité de sa mise en œuvre, il est nécessaire d'automatiser autant que possible les différentes fonctions de l'administration.

L'administration des couches basses (système d'exploitation, réseaux) a suscité un grand intérêt dans les communautés académiques et industrielles. Les travaux effectués ont mené à la création d'un certain nombre de standards qui sont largement adoptés. En revanche, l'administration des intergiciels n'a fait l'objet que de très peu de travaux. C'est le sujet d'étude que nous avons abordé dans le cadre de cette thèse.

9.2 Principaux apports

Dans cette thèse, nous avons identifié deux besoins pour l'administration de systèmes distribués. Le premier besoin est que le système à administrer soit *administrable*, c'est à dire facilement configurable, déployable et adaptable à l'exécution. Le second besoin est de construire des *systèmes d'administration* automatisant certaines fonctions, telles le déploiement, la mise à jour de

code et la réparation des pannes. Nous avons apporté des contributions pour couvrir ces deux besoins. Nous résumons ces contributions dans les deux sections suivantes.

9.2.1 Construction de systèmes administrables

Un prérequis pour qu'un système soit administrable est que son architecture soit modulaire et explicite. Pour ce faire, nous avons décidé d'utiliser le modèle de composants Fractal qui permet de construire des systèmes dans lesquels l'architecture est explicite lors de la construction et de l'exécution du système. Nous avons proposé une démarche permettant de rendre administrable un logiciel patrimonial. Cette démarche consiste à identifier les différentes parties du système et de leurs dépendances, à identifier les paramètres de configuration et à empaqueter les composants. Nous avons illustré cette démarche à travers la réingénierie de JOnAS, le serveur d'application J2EE développé au sein du consortium ObjectWeb.

Le choix de J2EE a été motivé par le fait que les serveurs d'applications présentent une architecture très complexe formée de plusieurs étages. Les étages sont architecturés sous forme de *services* offrant des propriétés dites non-fonctionnelles aux applications. Le code de JOnAS contient approximativement 400 000 lignes. Nous l'avons rendu administrable. Le prototype résultant est appelé JonasALaCarte. Dans JonasALaCarte, les différents services du serveur J2EE sont encapsulés sous forme de composants Fractal et leurs dépendances sont exposées sous forme de liaisons Fractal. De plus, les services sont archivés dans des paquetages différents et les dépendances entre les paquetages sont explicitement exprimées afin d'en tenir compte lors du déploiement.

Nous avons montré que le travail de réingénierie demande un temps raisonable (il faut approximativement un mois pour « fractaliser » les 400 000 lignes de code de JOnAS pour un ingénieur ayant une bonne connaissance de ce code). Par ailleurs, nous avons montré que JonasALaCarte respecte les différents standards implantés par JOnAS. Enfin, des expériences réalisées sur une grappe de machines ont prouvé que le travail de réingénierie ne dégrade pas significativement les performances du serveur initial.

9.2.2 Construction de systèmes d'administration

Au sein de cette thèse, nous avons proposé un système d'administration *fondé sur l'architecture*, c'est à dire un système d'administration utilisant des éléments d'architecture du système administré pour le configurer, le déployer et le reconfigurer. Ce système d'administration est formé d'un ensemble de composants Fractal structurés sous forme de boucles de commande. Des composants *capteurs* permettent de superviser le système administré. Ils envoient des notifications à des composants *gestionnaires* lors de l'occurrence d'événements de types pannes ou dégradation de performances. Les composants gestionnaires implantent les politiques d'administration. Ils analysent les événements reçus, déroulent un protocole défini et envoient des commandes pour adapter le système administré. Ces commandes sont exécutées à l'aide de composants *actionneurs*. Par ailleurs, nous avons introduit au système d'administration *la carte* qui est une représentation causalement connectée du système administré. Les gestionnaires utilisent la carte pour récupérer l'état courant du système administré à l'exécution ainsi que sa configuration et son architecture initiale, par exemple en cas de panne.

Le système d'administration contient un protocole de déploiement reposant sur l'utilisation d'un ADL. L'architecture du système à déployer est décrite dans des fichiers ADL sous forme d'assemblages de composants dont les dépendances sont explicitées à l'aide de compositions et de liaisons Fractal. Le protocole de déploiement automatise l'installation et l'instanciation des composants sur les machines cibles. Il est développé sous forme d'extension du système de

déploiement de Fractal. Cette extension consiste à intégrer la plateforme OSGi qui offre des propriétés évoluées d'empaquetage et de chargement de classes.

Le système d'administration contient également des mécanismes de reconfiguration de structure et d'implantation. Les reconfigurations de structure permettent de modifier à l'exécution l'architecture du système administré ; les reconfigurations d'implantation permettent de mettre à jour dynamiquement le code des parties du système. La mise en œuvre du système de reconfiguration d'implantation est rendue possible par l'utilisation de la notion d'empaquetage (et de gestion de versions associée) définie dans le protocole de déploiement.

Le système d'administration proposé a été utilisé dans JonasALaCarte. Nous avons décrit son utilisation pour automatiser le déploiement du serveur sur une grappe de machines. Nous avons également illustré le protocole de mise à jour dynamique de code via la mise à jour d'un service Web. Enfin, nous avons présenté un scénario d'auto-réparation du serveur suite à une panne logicielle et matérielle pour illustrer les mécanismes d'administration autonome mis en œuvre dans JonasALaCarte. Nous avons montré que la politique du redémarrage partiel (*micro-reboot*) était pertinente pour la réparation des pannes dans les serveurs J2EE.

Une évalution du système d'administration a montré qu'il apportait un gain significatif en termes de temps de configuration et de déploiement sur les grappes. Enfin, une comparaison avec d'autres systèmes d'administration fondés sur l'architecture a montré les avantages de notre système d'administration. D'abord, notre système couvre plusieurs fonctions de l'administration (la configuration, le déploiement et la reconfiguration) alors que les systèmes comparables sont généralement restreints à une seule fonction. De plus, le système de déploiement que nous avons proposé traite le cycle de vie des composants de l'empaquetage à l'exécution alors que les systèmes de déploiement que nous connaissons se concentrent exclusivement sur l'empaquetage ou sur la gestion des composants à l'exécution. Enfin, le système d'administration que nous proposons est *administrable*. En effet, comme il est formé de composants Fractal, il est configurable, déployable et adaptable au même titre que le système administré.

9.3 Perspectives

Nous envisageons de continuer notre travail selon les axes suivants.

9.3.1 Application de l'approche à un grain plus fin

Nous pensons que notre travail effectué au niveau des intergiciels est transférable au niveau applicatif. Dans le cadre des applications J2EE, nous pensons qu'il serait intéressant de « fractaliser » le service EJB. Cette solution présenterait plusieurs avantages. D'abord, elle permettrait de disposer d'une vue explicite des applications et de les modifier dynamiquement suivant les besoins. D'autre part, elle permettrait d'exprimer les dépendances entre les applications elles-mêmes et de partager les ressources entre elles. Les applications J2EE ne seraient plus alors des boîtes noires indépendantes comme c'est le cas actuellement. Enfin, disposer des mêmes unités de configuration, de déploiement et de reconfiguration, que sont les composants Fractal, permettrait d'exprimer les dépendances entre les applications et les services du serveur. Ceci serait utile pour une meilleure gestion des ressources. En effet, il deviendrait possible d'exprimer dans les fichiers ADL les services dont une application a besoin et de ne déployer ces services qu'à la demande.

Néanmoins, la « fractalisation » à grain fin nécessite une étude des performances. Il est probable qu'un classement des applications sera nécessaire pour identifier les types d'applications pour lesquelles la « fractalisation » ne dégrade pas les performances. De plus, il faut que la « fractalisation » soit cachée au développeur d'applications J2EE qui doit respecter la syntaxe

spécifiée par les standards J2EE. Ceci nécessite la création d'outils convertissant en composants Fractal le code écrit suivant les standards J2EE.

9.3.2 Implantation d'autres politiques d'administration

Dans cette thèse, nous nous sommes concentrés sur les mécanismes automatisant l'adaptation du système administré à l'exécution. Nous avons testé la politique du redémarrage partiel pour la réparation des pannes. Nous envisageons d'étudier d'autres types de politiques.

En particulier, nous souhaitons travailler sur les aspects gestion de ressources pour une exploitation optimale des ressources et pour améliorer les performances. En effet, les serveurs d'applications J2EE sont aujourd'hui consacrés à une seule application à la fois. Dans certains cas, même une exploitation d'une grappe J2EE ne suffit pas à supporter les pics de charge. Nous pensons qu'il serait intéressant de mener une étude plus rigoureuse de la gestion des ressources. Pour ce faire, il est nécessaire que le déploiement soit dirigé par les ressources disponibles. Nous n'avons pas rencontré ce besoin pour le déploiement dans une grappe où les machines cibles sont généralement homogènes et où les ressources sont suffisamment disponibles aux serveurs et aux applications J2EE. Néanmoins, dans d'autres environnements de type grille, les machines ont des configurations hétérogènes et partagent leurs ressources entre plusieurs applications.

Nous avons commencé par étendre l'ADL Fractal [1] pour exprimer, d'une part, les contraintes en ressources des composants à déployer et, d'autre part, les ressources disponibles. Néanmoins, outre la prise en compte des ressources disponibles, il est nécessaire d'implanter les algorithmes nécessaires au redimensionnement du serveur et des applications pour les adapter à la charge que subit le système. Nous pensons que les mécanismes que nous avons implantés dans JonasA-LaCarte sont suffisants pour effectuer des modifications de l'architecture des serveurs J2EE.

9.3.3 Administration à grande échelle

Nous sommes intéressés à étendre notre travail sur l'administration des serveurs J2EE à un environnement de plus grande échelle comme les grilles. Comparées aux applications scientifiques de type applications de calculs parallèles, les applications J2EE sont interactives, l'architecture des serveurs J2EE est complexe et les dépendances entre les parties du système sont plus nombreuses et plus importantes. Par conséquent, l'administration des applications J2EE dans des environnements grilles présente aujourd'hui un véritable défi. En effet, en plus des problèmes liés à la sécurité, les systèmes d'administration doivent assurer la continuité du service des applications Internet et la cohérence entre les différentes parties du système. Plusieurs travaux de recherche se focalisent aujourd'hui sur la définition de modèles et d'architectures permettant d'exploiter au mieux les ressources des grilles. Néanmoins, l'administration de ces systèmes très largement distribués sollicite actuellement moins d'attention.

Nous avons commencé [2] à traiter l'aspect déploiement des serveurs J2EE à grande échelle. Pour ce faire, nous proposons de structurer les machines cibles suivant une topologie de réseau *overlay* pair-à-pair (ou *P2P*). L'adoption de réseaux *P2P* nous offre des facilités de routage et une gestion décentralisée des pannes. Nous pensons que l'association de notre approche de l'administration fondée sur l'architecture à une structuration *P2P* permettrait de faciliter la gestion des serveurs J2EE à grande échelle.

1. Ce travail est réalisé avec Didier Hoareau et Yves Mahéo du laboratoire Valoria de l'Université de Bretagne Sud.

2. Ce travail est réalisé avec Fabienne Boyer et Jean-Bernard Stefani du projet Sardes, dans le cadre du co-encadrement du master de Deniz Cokuslu.

Bibliographie

[ACL04] T. Abdellatif, E. Cecchet, and R. Lachaize, *Evaluation of a group communication middleware for clustered J2EE application servers*, IEEE International Symposium on Distributed Objects and Applications, October 2004.

[ACN02] J. Aldrich, C. Chambers, and D. Notkin, *Architectural Reasoning in ArchJava*, Proceedings 16th European Conference on Object-Oriented Programming (ECOOP) (Malaga, Spain), 2002.

[AD06] T. Abdellatif and A. Danes, *A simple approach to autonomic J2EE servers*, International Conference on Self-Organization and Autonomic Systems in Computing and Communications (SOAS 2006) (Erfurt, Germany), 2006.

[ADZ00] Mohit Aron, Peter Druschel, and Willy Zwaenepoel, *Cluster reserves: a mechanism for resource management in cluster-based network servers*, Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Santa Clara, California), June 2000, pp. 90–101.

[AFF+01] 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*, Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management, May 2001.

[AG03] T. Corbi A. Ganek, *The Downing of the Autonomic Computing Era*, IBM Systems Journal - Special issue on autonomic computing (2003), 5–18.

[AKS05] T. Abdellatif, J. Kornaś, and J-B. Stefani, *J2EE Packaging, Deployment and Reconfiguration Using a General Component Model*, 3rd International Working Conference on Component Deployment (CD 2005) (Grenoble, France), November 2005.

[aok05] *Aokell web site*, 2005, http://www.lifl.fr/seinturi/aokell/javadoc/overview-summary.html.

[Apa] Apache, http://www.apache.org.

[ASCN03] J. Aldrich, V. Sazawal, C. Chambers, and David Notkin, *Language Support for Connector Abstractions*, Proceedings 17th European Conference on Object-Oriented Programming (ECOOP), 2003.

[BBH+05] 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*, 24th IEEE Symposium on Reliable Distributed Systems (SRDS) (Orlando, Florida, USA), October 2005.

[BCS] E. Bruneton, T. Coupaye, and J.-B. Stefani, *The Fractal Component Model Specification - ObjectWeb Consortium*, http://www.objectweb.org, June, 2005.

[BLQ+05] P. Bidinger, M. Leclercq, V. Quéma, A. Schmitt, and J.-B. Stefani, *Dream Types – A Domain Specific Type System for Component-Based Message-Oriented Midd-*

leware, 4th Workshop on Specification and Verification of Component-Based Systems (SAVCBS'05), in association with ESEC/FSE'05 (Lisbon, Portugal), September 2005.

[CBCP01] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, *An Efficient Component Model for the Construction of Adaptive Middleware*, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01) (Heidelberg, Germany), November 2001, pp. 160–178.

[CDD04] H. Cervantes, M. Desertot, and D. Donsez, *Frogi : Déploiement de composants fractal sur osgi*, Proceedings of the 1ère Conférence Française sur le Déploiement et la (Re)Configuration de Logiciels (DECOR'2004) (Grenoble, France), October 2004.

[CHG+04] S.W. Cheng, A.C. Huang, D. G., B. Schmerl, and P. Steenkiste, *Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure*, Computer, vol. 37, no. 10, pp. 46-54, 2004.

[CIG+03] J.S. Chase, D.E. Irwin, L.E. Grit, Justin D. Moore, and S.E. Sprenkle, *Dynamic virtual clusters in a grid site manager*, Proceedings 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03) (Seattle, Washington), June 2003.

[CKZ+03] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox", *JAGR: An Autonomous Self-Recovering Application Server*, Proc. 5th International Workshop on Active Middleware Services, 2003.

[CLQ05] E. Cecchet, O. Layaïda, and V. Quéma, *LeWYS: un Canevas Logiciel à Composants pour Construire des Applications de Supervision*, Actes des Journées sur les Systèmes à Composants Adaptables et Extensibles (Le Croisic, France), 6–8 avril 2005.

[CMZ02] E. Cecchet, J. Marguerite, and W. Zwaenepoel, *Performance and Scalability of EJB Applications*, Proceedings of OOPSLA'02 (Seattle, WA, USA), November 2002.

[cor02] *Corba/iiop specification*, OMG TC Document formal/02-06-01, 2002.

[DJa04] E. Dolstra, M. De Jonge, and E. Visser (2004a), *Nix: A safe and policy-free system for software deployment.*, In Damon L. editor 18th Large Installation System Administration Conference (LISA' 04), pages 79-92, Atlanta, Georgia, USA. USENIX, 2004.

[Dol05] E. Dolstra, *Efficient upgrading in a purely functional component deployment model.*, In Eighth International SIGSOFT Symposium on Component-based Software Engineering (CBSE 2005), volume 3489 of Lecture Notes in Computer Science, pages 219-234, St. Louis, Missouri, USA. Springer-Verlag, 2005.

[DvdHT02] E. Dashofy, A. van der Hoek, and R. Taylor, *Towards Architecture-Based Self-Healing Systems*, Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02), ACM, 2002.

[DVdJb04] E. Dolstra, E. Visser, and M. de Jonge (2004b), *Imposing a memory management discipline on software deployment.*, In Estubier and J. and Rosenblum D. editors 26th International Conference on Software Engineering (ICSE'04), pages 583-592, Edinburgh, Scotland. IEEE Computer Society., 2004.

[edga] *A Distributed Infrastructure for e-Business.*, http://www.akamai.com, 2002.

[edgb] *Edge Side Includes (ESI).*, http://www.esi.org/index.html, 2002.

[edgc] *WebSphere Edge Server.*, http://www-3.ibm.com/software/webservers/edgeserver, 2002.

[Ent02] Enterprise JavaBeansTM Specification, Version 2.1, August 2002, Sun Microsystems, `http://java.sun.com/products/ejb/`.

[Exe04] F. Exertier, *J2EE deployment: the JOnAS case study*, 2004, pp. 27–36.

[FCC+03] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, *SHARP: an architecture for secure resource peering*, Proceedings of the nineteenth ACM Symposium on Operating Systems Principles (SOSP'03), ACM Press, 2003, pp. 133–148.

[FR03] M. Fleury and F. Reverbel, *The jboss extensible server*, International Middleware Conference (2003).

[fraa] *Fractal ADL*, http://fractal.objectweb.org/tutorials/adl/.

[frab] *Fractal Explorer*, http://fractal.objectweb.org.

[fra05] *FracTalk web site*, 2005, http://csl.ensm-douai.fr/FracTalk.

[FSLM02] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller, *THINK: A Software Framework for Component-based Operating System Kernels*, Proceedings of the USENIX Annual Technical Conference, 2002.

[GGL+] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft, *SmartFrog: Configuration and Automatic Ignition of Distributed Applications*, HP OVUA, 2003.

[Gro02] Object Management Group, *CORBA Components*, formal/02-06-65, June 2002.

[Hal04] R. Hall, *A Policy-Driven Class Loader to support Deployment in Extensible Frameworks*, Second International Working Conference on Component Deployment, CD 2004 (Edinburg, UK), May 2004, pp. 81–96.

[HC04] R. S. Hall and H. Cervantes, *An OSGi Implementation and Experience Report*, Consumer Communications and Networking Conference (CCNC'2004) (Las Vegas, Nevada, USA), 2004.

[HHW99] R. Hall, D. Heimbigner, and A. Wolf, *A Cooperative Approach to Support Software Deployment Using the Software Dock*, Proceedings of the 21st International Conference on Software Engineering (ICSE'99) (Los Angeles, CA), May 1999, pp. 174–183.

[HT03] P. Hnetynka and P. Tuma, *Fighting Class Name Clashes in Java Component Systems*, Proceedings of the Joint Modular Language Conference (JMLC'2003) (Klagenfurt, Austria), 2003.

[HW04] J. Hillman and I. Warren, *An Open Framework for Dynamic Reconfiguration*, 26th International Conference on Software Engineering, 2004, pp. 594–603.

[JDJ+06] M. Jordan, L. Dayns, M. Jarzab, C. Bryce, and G. Czajkowski, *Scaling j2ee application servers with the multi-tasking virtual machine*, Softw. Pract. Exper. **36** (2006), no. 6, 557–580.

[jmx04] *Java Management Extensions (JMX) specification version 1.2*, Mars 2004, Sun Microsystems, http://java.sun.com/products/JavaManagement/.

[jon] *Java Open Application Server (JOnAS)*, http://jonas.objectweb.org.

[JPMF] J.-Ph Hubaux J.-Ph. Martin-Flatin, S. Znaty, *A survey of distributed enterprise network and system management paradigms*, Journal of Network and Systems Management **7**, no. 1.

[jsra] *J2EE Management Specification (JSR 77)*, http://jcp.org/jsr/detail/77.jsp.

[jsrb] *JEEE Deployment Specification (JSR88)*, http://jcp.org/jsr/detail/88.jsp.

[jsrc] *JMX remote API 1.0 Specification, version 1.0, 2003 October (JSR160)*, http://jcp.org/en/jsr/detail?id=160.

[jul02] *Julia: Fractal Composition Framework Reference Implementation*, 2002, Objectweb, http://www.objectweb.org/fractal/.

[Kep05] J. Kephart, *Research Challenges of Autonomic Computing*, Proceedings of the 27th international conference on Software Engineering (ICSE 2005) (New York, USA), 2005, pp. 15–22.

[KLQS04] J. Kornaś, M. Leclercq, V. Quéma, and J.-B. Stefani, *Support pour la reconfiguration d'implantation dans les applications à composants java*, Proceedings of the 1ère Conférence Française sur le Déploiement et la (Re)Configuration de Logiciels (DECOR'2004) (Grenoble, France), October 2004.

[LF] J. Lindfors and M. Fleury, *JMX-Managing J2EE with Java Management Extensions*, Sams, The JBoss Group, 2005.

[LH05] O. Layaïda and D. Hagimont, *Designing Self-adaptive Multimedia Applications Through Hierarchical Reconfiguration*, The 5th IFIP Conference on Distributed Applications and Interoperable Systems (Athens, Greece), Lecture Notes in Computer Science, vol. 3543, Springer, June 2005, pp. 95–107.

[LvdH04] C. Luer and A. van der Hoek, *JPloy: User-Centric Deployment Support in a Component Platform*, Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004) (Edinburg, Scotland), 2004.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, *Specifying Distributed Software Architectures*, 5th European Software Engineering Conference (Sitges, Spain), September 1995, pp. 137–153.

[MFH01] S. McDirmid, M. Flatt, and W.C. Hsieh, *Jiazzi: New-age components for old-fashioned Java*, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), ACM Press, 2001.

[MM00] R. Marvie and P. Merle, *Vers un modèle de composants pour cesure, le corba component model*, Tech. Report Projet RNRT 98, LIFL, November 2000.

[Mul04] A. Mulder, *Apache Geronimo Development and Deployment*, Pearson Education, Inc, 2004.

[obj] *ObjectWeb Consortium*, http://www.objectweb.org.

[Oga97] K. Ogata, *Modern Control Engineering, 3rd edition*, Prentice-Hall, 1997.

[OGP03] D. Oppenheimer, A. Ganapathi, and D.A. Patterson, *Why do Internet services fail, and what can be done about it?*, 4th Symposium on Internet Technologies and Systems (USITS 2003), 2003.

[omg] *OMG. The Object Management Group*, http://www.omg.org.

[onD] *M. Desertot. JOnAS 5 and dynamic services with OSGi. ObjectWeb Architecture Meeting, Juin 2005*, www.objectweb.org/phorum/.

[Ope03] Open Services Gateway Initiative, OSGi service gateway specification, Release 3, April 2003, `http://www.osgi.org`.

[PBJ97] F. Plasil, D. Balek, and R. Janecek, *Dynamic Component Updating in Java/CORBA Environment*, Tech. Report 97/10, Department of Software Engineering, Charles University, Prague, 1997.

[PBJ98] ———, *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Proceedings of International Conference on Configurable Distributed Systems (ICCDS'1998) (Annapolis, Maryland, USA), 1998.

[Pro] JOnAS Project, *Java Open Application Server (JOnAS): A J2EE Platform*, http://jonas.objectweb.org/current/doc/JOnASWP.html.

[pro05] *Proactive web site*, 2005, http://www-sop.inria.fr/oasis/ProActive.

[SAH+03] C.A.N. Soules, J. Appavoo, K. Hui, D. Da Silva, G.R. Ganger, O. Krieger, M. Stumm, R.W. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, *System Support for Online Reconfiguration*, Proceedings Usenix Annual Technical Conference (San Antonio, Texas), 2003.

[SB98] L. Sheng and G. Bracha, *Dynamic class loading in the java virtual machine*, Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'1998) (Vancouver, Canada), 1998.

[SPMF] J. Sch´onw´alder, A. Pras, and J.-Ph. Martin-Flatin, *On the future of internet management technologies*, IEEE Cmmunications Magazine **41**, no. 10.

[ST94] M. Sloman and K. Twilde, *Domains: A Framework for structuring Management Policy*, Addison-Wesley ed., ch. 16, pp. 433–453, in Morris Sloman, editor, Network and Distributed Management Systems, 1994.

[Sun] Sun, *Java 2 platform, enterprise edition (j2ee)*, http://java.sun.com/j2ee/.

[Szy02] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2002.

[vdHHW98] A. van der Hoek, D. Heimbigner, and A. Wolf, *Software Architecture, Configuration Management, and Configurable Distributed Systems: A Ménage a Trois*, Tech. Report CU-CS-849-98, Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, January 1998.

[vRBV03] R. van Renesse, K. Birman, and W. Vogels, *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*, ACM Transactions on Computer Systems **21** (2003), no. 2.

[WHW+04] S. White, J. Hanon, I. Whalley, D. Chess, and J. Kephart, *An Architectural Approach to Autonomic Computing*, Proceedings IEEE Int. Conference on Autonomic Computing (ICAC 2004), 2004.