Support langage et système pour l'administration autonome

Laurent Broto

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*
Discipline ou spécialité : *Informatique*

Présentée et soutenue par *M. Laurent Broto*
Le *30 septembre 2008*

Titre : *Support langage et système pour l'administration autonome*

JURY

F. BOYER, Maitre de Conférence a l'Université Joseph Fourier de Grenoble, Rapporteur
M. RIVEILL, Professeur a l'Ecole Polytechnique de l'Université de Nice, Rapporteur
M. DAYDE, Professeur a l'Institut National Polytechnique de Toulouse, Examinateur
N. DEPALMA, Maitre de Conférence a l'Institut National Polytechnique de Grenoble, Examinateur
D. HAGIMONT, Professeur a l'Institut National Polytechnique de Toulouse, Directeur de Thèse
JP. BAHSOUN, Professeur a l'Université Paul Sabatier de Toulouse, Directeur de Thèse

---

Ecole doctorale : *Ecole Doctorale Mathématiques Informatique Télécommunication de Toulouse*
Unité de recherche : *Institut de Recherche Informatique de Toulouse*
Directeurs de Thèse :
D. HAGIMONT, Professeur a l'Institut National Polytechnique de Toulouse
JP. BAHSOUN, Professeur a l'Université Paul Sabatier de Toulouse

A ma pib...

L'erreur est humaine mais un véritable
désastre nécessite un ordinateur.
Bill Gates, le 1er Janvier 1990

Je tiens à remercier

Monsieur Michel Daydé, Professeur à l'Institut National Polytechnique de Toulouse et Directeur adjoint de l'Institut de Recherche en Informatique de Toulouse qui m'a fait l'honneur de présider mon jury de thèse,

Madame Fabienne Boyer et Monsieur Michel Riveill qui ont accepté d'être les rapporteurs de mon travail,

Monsieur Jean-Paul Bashoun, Professeur à l'Université Paul Sabatier, et Monsieur Daniel Hagimont, Professeur à l'Institut National Polytechnique de Toulouse, qui m'ont encadré durant ces années de dur labeur,

Monsieur Zoubir Mammeri, Professeur à l'Université Paul Sabatier, pour la confiance qu'il m'a accordé en m'accueillant dans son équipe,

La société Newtech qui m'a accueilli durant les trois premières années de ma thèse.

Je tiens également à remercier toutes les personnes qui m'ont aidé, soutenu et encouragé.

Les environnements logiciels distribués sont de plus en plus complexes et difficiles à administrer car ils sont composés d'applications patrimoniales avec des interfaces d'administration spécifiques. De plus, les tâches d'administration menées par un humain sont sujettes à erreur et à un manque de réactivité. Ceci est particulièrement visible sur les architectures moyennement ou grandement distribuées. Pour résoudre ce problème, nous étudions la conception et l'implantation d'un système d'administration autonome. Le principe est d'encapsuler un morceau de logiciel patrimonial dans un composant puis d'administrer l'architecture patrimoniale comme une architecture à composant. Cependant, nous avons observé que les interfaces d'un modèle à composant sont de trop bas niveau et difficiles à utiliser. Nous introduisons donc des formalismes de haut niveau pour spécifier le déploiement et les politiques de reconfiguration. Cette thèse décrit ces contributions qui sont intégrées dans un prototype de système autonome baptisé TUNe.

Distributed software environments are increasingly complex and difficult to manage, as they integrate various legacy software with specific management interfaces. Moreover, the fact that management tasks are performed by humans leads to many configuration errors and low reactivity. This is particularly true in medium or large-scale distributed infrastructures. To address this issue, we explore the design and implementation of an autonomic management system. The main principle is to wrap legacy software pieces in components in order to administrate a software infrastructure as a component architecture. However, we observed that the interfaces of a component model are too low-level and difficult to use. Therefore, we introduced higher-level formalisms for the specification of deployment and management policies. This thesis describes these contributions which are integrated in a prototype autonomic system called TUNe.

Table des matières

Chapitre 1

Introduction

Les logiciels sont devenus de plus en plus complexes à administrer. Cette complexité est apparue avec la généralisation des environnements répartis, et plus particulièrement avec l'utilisation de grilles de calcul à grande échelle et d'applications multi-tiers.

Les applications multi-tiers permettent d'interconnecter des serveurs qui rendent des services différents. Ces serveurs sont des logiciels différents avec des procédures d'administration spécifiques, ce qui rend complexe l'administration de l'ensemble. Les grilles de calculs fournissent elles de grandes capacités de calcul et une haute scalabilité mais de part leur dispersion géographique, le grand nombre de logiciels gérés et leur hétérogénéité, l'administration est également très difficile.

Il n'est plus aujourd'hui raisonnable pour un administrateur humain d'administrer des logiciels multi-tiers réparti sur des centaines de noeuds. Cette administration serait trop coûteuse en terme humain (temps d'administration) mais aussi source d'erreurs et de manque de réactivité.

Une approche prometteuse pour résoudre ces problèmes est l'administration autonome proposée par IBM en 2003 [15]. Un logiciel d'administration autonome permet d'effectuer des tâches d'administration telles que le déploiement, la réparation ou encore l'optimisation en limitant les interventions humaines. Il fournit un support pour la supervision de l'environnement administré et permet de définir des réactions face à des événements comme des pannes ou des surcharges, afin de reconfigurer les applications administrées de façon autonome.

Dans cette approche, de nombreux travaux se sont appuyés sur un modèle à composants. Le principe général est de gérer les éléments administrés comme des composants logiciels, soit parce qu'ils ont été développés comme des composants, soit

en les encapsulant dans des composants[1]. Ainsi, l'environnement logiciel global peut être géré comme une architecture à composants. L'administrateur bénéficie alors des atouts du modèle à composants utilisé, l'encapsulation, les outils de déploiement et les interfaces de reconfiguration, afin d'implanter des procédures d'administration autonome.

Cependant, cette méthode engendre plusieurs problèmes :

- La construction des composants nécessite de connaître le modèle de programmation du système à composants utilisé.
- La construction de l'application globale, sous la forme d'une architecture logicielle à composants, nécessite de connaître le langage de description d'architecture du système à composants (généralement un ADL).
- La définition des politiques d'administration nécessite de connaître l'interface du système à composants permettant de reconfigurer l'architecture logicielle.

Ces différents modèles et langages à connaître sont une source de complexité et demandent un travail d'apprentissage important à l'administrateur qui n'est pas formé pour.

Nous proposons donc dans cette thèse une approche permettant d'automatiser les tâches d'administration en simplifiant au maximum la tâche pour l'administrateur.

Ce rapport de thèse est organisé de la façon suivante :

- **Chapitre 2**
  Nous étudions dans ce chapitre deux applications multi-tiers dont les tiers peuvent être répartis sur plusieurs machines appartenant à une grille de calcul. Ces deux applications ont été choisies car elles possèdent chacune un schéma de déploiement couramment utilisé en informatique, le schéma client-serveur. La première (J2EE) est une application utilisée dans des domaines grands public et industriels tandis que la seconde (Diet) est surtout cantonnée au monde de la recherche dans le domaine du calcul scientifique sur la grille.
  Nous soulignons pour chaque application les difficultés inhérentes à l'administration de ce type d'architecture et nous commençons une première classification des problèmes. Nous nous servons tout au long de cette étude de ces deux applications pour mettre en évidence les problèmes mais aussi comment notre approche traite ces problèmes.

- **Chapitre 3**
  Le chapitre 3 présente un état de l'art des différentes technologies à composants. Nous voyons pour chaque technologie à composants ce qu'elle peut

[1] On dit alors que l'on encapsule un logiciel patrimonial dans un composant. Un logiciel patrimonial est une application non prévue pour être administrée automatiquement. Dans la suite, nous nous intéressons surtout aux systèmes autonomes permettant d'aministrer des environnements logiciels patrimoniaux

apporter à l'administration autonome.

– **Chapitre 4**
Nous passons en revue dans le chapitre 4 différentes approches d'administration autonome. Pour chaque approche, nous nous intéressons à ce qu'elle permet de faire pour résoudre les problèmes de l'administration autonome mais aussi les problèmes qu'elle engendre. Nous concluons par une classification de ces approches.

– **Chapitre 5**
Une des approches présentées au chapitre précédent est celle du projet Jade, qui implante un système autonome en s'appuyant sur le modèle à composants Fractal. Pour illustrer la problématique que nous adressons, nous détaillons dans ce chapitre l'utilisation de Jade pour administrer l'application Diet et nous montrons les problèmes posés par cette approche, à savoir la complexité induite par le modèle à composants utilisé.

– **Chapitre 6**
Nous présentons dans ce chapitre notre contribution qui a pour objectif de fournir un système d'administration autonome sans tomber dans les écueils présentés précédemment.
Nous expliquons comment avec notre approche une application patrimoniale peut être administrée automatiquement et quel est le rôle de l'administrateur dans le processus de préparation de l'application patrimoniale et son rôle une fois que l'application a été démarrée.
Nous reprenons alors l'application utilisée au Chapitre 5 pour illustrer la problématique et nous l'administrons automatiquement avec notre système. Nous concluons alors sur ce que notre approche apporte.

– **Chapitre 7**
Nous voyons alors comment notre approche est implantée et nous décrivons son comportement du lancement de l'application patrimoniale jusqu'à son arrêt en passant par des phases de reconfiguration types telles que des réparations ou des optimisations.

– **Chapitre 8**
Nous présentons des résultats obtenus avec notre système sur différentes applications dans des cas de figure courants dans le cadre de l'administration. Nous mesurons des charges pour montrer que notre approche permet d'optimiser des applications patrimoniale et mesurons des temps pour montrer la réactivité.

– **Chapitre 9**
Nous concluons alors et donnons la liste des évolutions que nous envisageons en vue de rendre notre approche plus efficace et intuitive.

Chapitre 2

Cadre des travaux et problématique

2.1 Introduction

Les environnements informatiques d'aujourd'hui sont de plus en plus sophistiqués. On entends ici par environnement informatique l'ensemble des logiciels et des matériels sur lesquels les logiciels sont déployés. La complexité d'aujourd'hui est due aux propriétés des logiciels récents :

- les logiciels sont de plus en plus souvent multi-tiers, donc composés de plusieurs éléments rendant chacun un service précis. Ces différents tiers utilisent en général des méthodes de configuration différentes et des langages de configuration différents. Cela permet difficilement à une seule personne, l'administrateur, de connaître parfaitement les différents langages et les subtilités des configurations. Au mieux la configuration ne sera pas optimale, au pire, des erreurs de configuration peuvent apparaître lors du lancement de l'application,
- les logiciels sont aussi de plus en plus souvent déployés sur des ensembles de machines qui peuvent en compter plusieurs milliers. Lors de la phase de déploiement, chaque tiers devra donc être déployé sur un noeud[1] et configuré pour ce noeud. Les autres tiers devront être configurés pour leur noeud mais aussi en fonction des tiers avec lesquels ils vont communiquer. Ainsi, un serveur web aura besoin de connaître l'adresse de la base de données avec laquelle il va communiquer.

Cette sophistication complique considérablement les tâches d'administration qui sont multiples :

- l'exécution du logiciel. Cela correspond à son installation sur un noeud. Dans le cas de logiciel qui doit être installé sur plusieurs machines, on parlera de déploiement,

[1] on appelle noeud une machine appartenant à un ensemble de machine

- la maintenance du logiciel. Cela correspond à surveiller le logiciel déployé et palier aux pannes pouvant survenir (plantage du logiciel, plantage du système d'exploitation, ...),
- optimiser le logiciel. Cela correspond à rendre le logiciel plus efficace dans son environnement courant. Des optimisations peuvent être faites au moment du déploiement (comme par exemple déployer la version 64 bits du logiciel au lieu de la version 32 bits), mais aussi durant la vie du logiciel (comme par exemple ajouter des serveurs dans le cadre d'une application client serveur pour augmenter le nombre de clients pouvant se connecter simultanément aux serveurs).

L'ensemble des tâches de maintenance ou d'optimisation sera nommé *reconfiguration* car le logiciel réparé ou optimisé n'aura plus forcément les même propriétés que lors de la phase de déploiement.

La complexité croissante des logiciels rend l'administration difficile car
- lors du déploiement, les tiers doivent être configurés les uns par rapport aux autres et ils peuvent être très nombreux lorsqu'on utilise des tiers répliqués pour exploiter au maximum les ensembles de machines,
- lors des reconfiguration, tous les tiers qui sont reliés aux tiers reconfigurés doivent souvent être eux aussi reconfigurés. Cela implique de connaître parfaitement l'architecture de l'application à un moment donné ce qui peut être ardu après plusieurs reconfigurations successives.

Ces contraintes ne permettent généralement pas à un humain de pouvoir effectuer des déploiements optimums ou d'effectuer des reconfigurations valides (i.e reconfigurer tout ce qui doit l'être et de manière correcte) dans des temps limités.

Nous présentons dans la suite de ce chapitre deux applications multi-tiers différentes auxquelles nous nous référons tout au long de cette thèse. La première est une application multi-tiers grand public, l'ensemble web Apache, Tomcat et MySQL tandis que la deuxième est une application développée dans un cadre universitaire et est un ordonnanceur mathématique : Diet. Nous avons choisi ces applications car elles correspondent à des patrons généralement utilisés dans le cadre d'application multi-tiers : maillée dans le cas de l'ensemble Apache-Tomcat-MySQL et arborescente dans le cas de Diet.
Pour chaque application, nous montrons les problèmes de déploiement et de configuration puis les besoins de reconfiguration et les problèmes engendrés par cette dernière. Enfin, nous concluons en synthétisant les problèmes rencontrés et en ouvrant une discussion sur une automatisation de ces tâches.

2.2 L'application Apache-Tomcat-MySQL

Cette application permet de répondre aux requêtes de clients web grâce à des programme développés en Java, les servlets [17], qui peuvent faire appel à une base de données pour gérer des informations persistantes. Elle est composée de trois serveurs :

- Apache [2] qui est un serveur web pouvant être réparti sur plusieurs noeuds pour absorber la charge,
- Tomcat [25] qui est un conteneur de *servlets*. Une servlet est un programme Java qui s'exécute côté serveur et qui permet de répondre à des requêtes HTTP[2]. Tomcat peut aussi être déployé sur plusieurs machines,
- MySQL [19] qui est une base de données déployable aussi sur plusieurs machines grâce au middlware *C-JDBC* [Cecchet et al.] par exemple.

Le serveur Apache dialogue avec le serveur Tomcat, par le biais d'un intergiciel nommé *mod_jk* [18], qui lui même dialogue avec le serveur MySQL pour donner une architecture telle que celle présenté figure 2.1

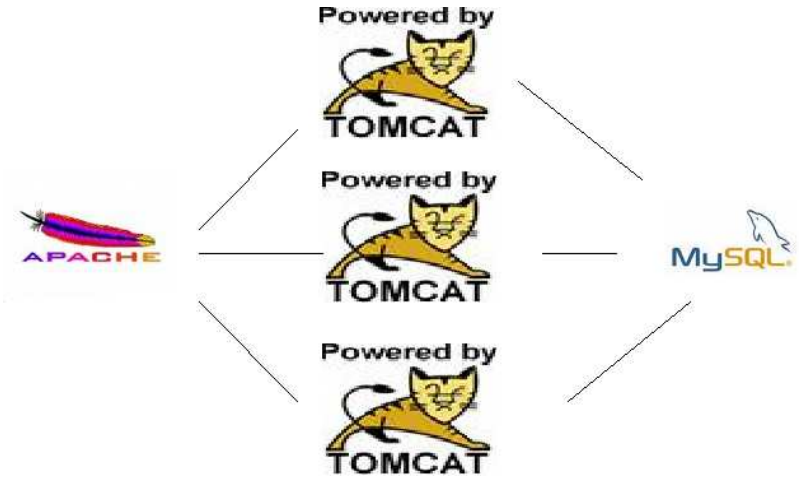

FIG. 2.1 – Une architecture Apache-Tomcat-MySQL

Dans cette architecture, le serveur Apache reçoit les requêtes web des différents clients qu'il transmet ensuite à un des serveur Tomcat. Ces derniers lors du traitement de la requête peuvent avoir besoin d'accéder à des données persistantes et peuvent alors accéder à la base de données MySQL.

2.2.1 Le déploiement et la configuration

Le déploiement de ces serveurs est aisé, il sont tous contenus dans un répertoire racine qu'il suffit de copier sur la machine hôte et ne dépendent que de librairies

[2]Hyper Text Transfert Protocol

standards, généralement déployées avec le système d'exploitation. La configuration est elle plus difficile car les fichiers de configurations sont dans des langage différents et demandent des informations différentes :

- Apache se configure en utilisant un fichier texte composé de paires *clé-valeur*. Les principales informations à donner sont les différents serveurs virtuels qui doivent contenir l'emplacement des fichiers des différents sites, les noms de sous-domaine ainsi que le port d'écoute,
- Tomcat se configure en utilisant un fichier XML[3] pour le serveur et de paires *clé-valeur* pour configurer les mod_jk pour le dialogue avec Apache. Le fichier XML contiendra entres autres l'adresse du serveur de base de données MySQL tandis que le fichier configurant les mod_jk contiendra entres autres l'adresse du serveur Apache,
- MySQL ne nécessite pas de configuration particulière dans le cadre d'une utilisation non clusterisée, mais nécessite de configurer des fichiers textes dans le cadre d'une utilisation clusterisée avec C-JDBC.

La configuration nécessite donc le paramétrage de divers fichiers dont les valeurs dépendent étroitement du type de machine sur lesquels les serveurs sont déployés (emplacement des fichiers web par exemple) ainsi que les adresses des machines pour configurer le dialogue entre les serveurs.

2.2.2 La reconfiguration

La reconfiguration de l'application peut être de trois types : reconfiguration sur panne, optimisation ou sécurisation. Dans le cas de la reconfiguration sur panne de noeud, un ensemble de serveurs doit être déplacé ce qui nécessite la reconfiguration des serveurs y afférents. Dans le cas d'une panne de serveur simple, aucune reconfiguration n'est requise mais il peut être judicieux d'arrêter les serveurs y afférents en vue de ne pas perdre de messages. Dans le cas de l'optimisation, comme dans le cas de la panne d'un noeud, les serveurs optimisés doivent être configurés et les serveurs y afférents doivent être eux aussi reconfigurés pour prendre en compte la présence de nouveaux serveurs. Enfin, dans le cas de la sécurisation, les machines émettant des requêtes invalides (comme par exemple un Apache qui envoi des requêtes directement à un MySQL) pourront être bannies.

2.3 L'application Diet

Diet [6; 7], développé à l'École Normale Supérieure de Lyon en France, est un ordonnanceur distribué. Il permet de répartir des opérations mathématiques sur plusieurs serveurs. Cette application est composée de quatre serveurs obligatoires :

[3] eXtensible Markup Language

– les MA (Master Agent), qui sont au sommet de la hiérarchie et qui distribuent les calculs aux différents LA,
– les LA (Local Agent), qui sont en dessous des MA et qui distribuent les calculs aux différents serveurs de calcul, les SeD,
– les SeD (Server Deamon) qui sont les serveurs permettant d'exécuter les calculs mathématiques proprement dit,
– OmniName qui est un serveur de nom Corba pour que les différents serveurs puisse se retrouver parmis les machines sur lesquels ils sont déployés.

Un exemple d'architecture Diet avec 6 MA, 4 LA et 3 SeD par LA mais sans OmniName est donné figure 2.2.

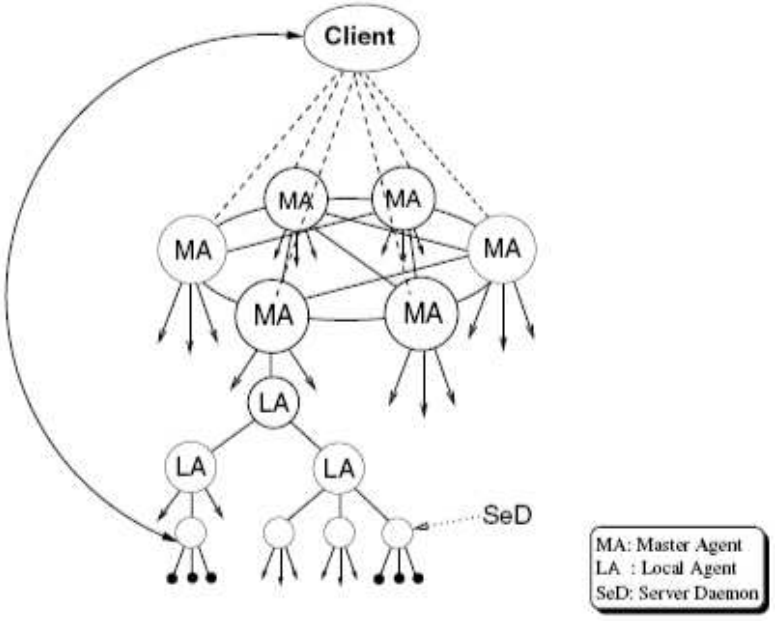

FIG. 2.2 – Une architecture Diet

Cette architecture arborescente permet un passage à l'échelle simple. En effet, si les MA envoient directement les requêtes aux SeD, ils risquent d'être saturés quand le nombre de SeD augmente. En déléguant aux LA et en permettant l'ajout de LA, on assure ainsi un passage à l'échelle simple pour ne pas que les MA soient saturés.

2.3.1 Le déploiement et la configuration

Le déploiement de Diet est aisé à l'image de celui de Apache-Tomcat-MySQL. Par contre, à la différence de l'application Apache-Tomcat-MySQL, la configuration de Diet est homogène, ce qui signifie que la syntaxe pour tous les type de serveurs est la même et les informations demandées sont semblables. La configuration est donc beaucoup plus aisée pour un humain, mais pose les mêmes problèmes à savoir connaître les paramètres des serveurs pour pouvoir configurer les serveurs y affé-

rents. Par contre, à la différence de l'application Apache-Tomcat-MySQL, le nombre de serveur déployés avec Diet est plus important. Il n'est pas rare de voir des applications Diet déployées avec plus de 400 serveurs. Le déploiement et la configuration, malgré son homogénéité, prennent donc du temps et sont aussi source d'erreurs de part le nombre de paramètres à changer d'un serveur à l'autre en fonction du noeud sur lequel ils sont déployés.

2.3.2 La reconfiguration

Pour effectuer des reconfigurations, l'administrateur doit manuellement redémarrer les serveurs en cas de pannes de ces derniers ou en cas de panne d'un noeud. De plus, l'application Diet a une charge qui peut varier énormément d'un moment à l'autre en fonction des requêtes que les utilisateurs font. Il est donc intéressant de pouvoir la reconfigurer rapidement pour s'adapter à la charge ou à l'absence de charge sans monopoliser inutilement des machines.

2.4 Synthèse

Nous avons présenté deux applications que nous utilisons tout au long de nos travaux. Nous avons pu voir que ces applications nécessitent d'être déployées et reconfigurées pour avoir des performances optimales. Nous avons aussi vu que la configuration, le déploiement et la reconfiguration peuvent être longs et ardus et sont donc source d'erreurs.
Il est donc intéressant d'imaginer un système de déploiement et de reconfiguration qui serait automatique. Des approches ont été développées pour Diet, avec GoDiet [10], mais ces approches sont généralement ad-hoc et non portables d'une application à l'autre.
Des outils de haut niveau pour administrer des applications ont été proposés. Ces outils peuvent être couplés à des outils permettant d'automatiser les tâches d'administration. Ces derniers, permettent de faire de *l'administration autonome*, sont basés sur des langages pour décrire le déploiement et définir des politiques de reconfiguration.

Nous présentons dans le chapitre suivant différentes approches d'administration autonome qui couvrent le domaine. Il en existe d'autres qui seront présentées en annexe A mais qui se classent parmi celles que nous détaillons ici.
Les approches d'administration autonome reposent sur un modèle à composants (défini section 3.1). L'utilisation d'un modèle à composants permet de faciliter l'administration des logiciels en les banalisant. En effet, si chaque composant encapsule un logiciel ou un morceau de logiciel et qu'il existe une logique pour administrer ces composants, il est alors aisé d'administrer n'importe quel logiciel. Le problème est alors déplacé dans les méthodes d'encapsulation.

Nous présentons donc dans le chapitre d'après les différentes approches à composants qui sont utilisées dans les approches d'administration autonome que nous détaillons. La majorité de ces approches à composants proposent un langage spécialisé pour le déploiement ce qui n'en fait pas un critère déterminant. Nous nous attachons donc à des critères déterminants que nous présentons section 3.1.4. Nous choisissons ces critères car ils facilitent l'implantation d'une approche d'administration autonome.
Nous présentons ensuite les approches d'administration autonome en décrivant les méthodes offertes à l'administrateur pour administrer son application (voir section 4.1.3).
Nous terminons enfin par un récapitulatif des différentes approches d'administration autonome présentées.

Chapitre 3

Approches à composants

3.1 Introduction

Les modèles à composants sont apparus pour palier aux problèmes rencontrés avec les modèles de programmation à objet. En effet, les modèles à objet permettent difficilement durant l'exécution de l'application de remplacer de manière simple une fonctionnalité par une autre. Les langages objets ont évolués vers cette possibilité en ajoutant le concept *d'interface comportementale* puis ensuite les possibilité d'introspection. Ces deux extensions permettent de remplacer une fonctionnalité par une autre (donc généralement une classe implantant une interface par une autre classe implantant la même interface) dans le cas où l'application a été développée pour.
Les modèles à composants vont plus loin en se positionnant au dessus du modèle à objet et en permettant explicitement de remplacer un composant par un autre ou de changer les liaisons entre les composants. Ainsi le programmeur n'a plus à se soucier des changements de fonctionnalités durant l'exécution, le modèle à composant propose des outils pour le faire.
Les modèles à composants fournissent donc un ensemble de services non fonctionnels pour permettre l'assemblage d'éléments basiques, les *composants*. Ces modèles permettent aux composants de passer des contrats entre eux, c'est à dire une définition des services fournis et des services requis. Les composants sont reliés entre eux par des *liaisons* pour former une architecture logicielle[1] interactions qui reste manipulable à l'exécution. Le logiciel permettant l'appel aux différentes fonctions pour créer des composants ou les relier est le *framework*.

[1] on entends par architecture logicielle la description de manière symbolique et schématique des différents composants d'un programme informatique et de leurs relations

3.1.1 Un Composant

Le composant est l'unité de structuration. En première approximation, il contient un morceau de code et des interfaces fonctionnelles et non fonctionnelles. Les interfaces non fonctionnelles, ou *contrôleurs*, permettent l'administration du composant (démarrage, arrêt, configuration, ...).
Les interfaces fonctionnelles permettent aux composants reliés de communiquer entre eux. L'ensemble de ces interfaces constitue la *membrane* du composant. Parmis ces interfaces, on en distingue deux types : les interfaces *clientes* qui émettent les requêtes et les interfaces *serveurs* qui répondent aux requêtes.
On distingue généralement deux types de composants : les composants *primitifs* qui encapsulent généralement un morceau de code permettant de rendre les services du contrat et les composants *composites* qui sont un assemblable d'autres composants. Enfin, les composants peuvent contenir des valeurs reflétant leur état qui sont les *attributs*.

3.1.2 Une liaison

Une liaison permet de relier deux composants entre eux. Elle peut être implantée différemment suivant les modèles et leurs spécificités. Elle peut par exemple être implantée sous forme de *référence* pour des composants écrits en Java et partageant le même espace d'adressage comme sous forme d'URL[2] dans un modèle composé de webservices distribués. On distingue plusieurs types de liaisons, des liaisons *primitives* qui relient directement deux composants et des liaisons *composites* qui peuvent effectuer des traitements sur les données échangées entre les composants.

3.1.3 Le framework

Le framework permet à l'utilisateur de construire, modifier et supprimer des composants ainsi que des liaisons. Il existe deux méthodes pour créer une application dans un modèle à composants :

- une méthode statique et descriptive qui permet de décrire l'architecture initiale dans un fichier. Le langage utilisé dans le fichier est un langage de description d'architecture *ADL*[3]. Ce langage permet de décrire les interfaces des composants, les composants à créer et la manière de relier les composants entre eux,
- une méthode dynamique par interfaces de programmation *API*[4]. Cette interface de programmation fournit des méthodes d'un langage pour permettre de créer des composants, de supprimer des composants et pour les relier entre eux.

[2] Uniform Resource Locator
[3] Architecture Description Language
[4] Application Programming Interface

Une application peut être créée par ADL puis reconfigurée grâce à l'API du framework.

3.1.4 Plan

Nous allons dans cette partie étudier quatre modèles à composants : C2, Weaves, SCA et Fractal.
Nous commençons par Fractal car c'est le modèle le plus générique des quatre qui regroupe les fonctionnalités dont nous avons besoin dans le cadre de l'administration autonome. Puis nous le comparons aux autres modèles.
Nous étudions en premier lieu le type de composant implanté par chaque modèle, le type de liaison proposé par chaque modèle et enfin la *dynamicité* de chaque modèle.

Dans l'étude des types de composants, nous nous intéressons à la possibilité de créer des composants composites et aux principales spécificités du modèle par rapport à notre modèle de référence Fractal. Les composants composites sont importants dans le cadre de l'administration autonomes de logiciels patrimoniaux car il permettent de refléter au mieux l'architecture de l'application administrée.
L'étude des types de liaison est intéressante pour savoir si le logiciel utilisant le modèle à composants est en mesure d'intercepter les messages en vue de pouvoir restaurer le système dans un état cohérent suite à une panne. En effet, lors d'une panne, il faut recréer un composant puis le mettre dans l'état que le composant en panne avait avant la panne. Une méthode qui permet de le faire est d'intercepter les messages que le composant reçoit puis les rejouer sur le composant réparé.
Enfin, la *dynamicité* est la capacité pour un modèle de pouvoir modifier l'architecture applicative durant l'exécution. Un modèle dynamique à notre sens devra comprendre les points suivants :

- possibilité d'ajouter ou d'enlever des composants durant l'exécution,
- possibilité de modifier les liaisons durant l'exécution,
- gestion du cycle de vie du composant. En effet, dans le cadre de la reconfiguration, les composants liés au composant à enlever ou au futur composant à ajouter doivent être arrêtés pour ne pas que les messages qu'ils émettent soient perdus.

Une dynamicité *complète* est une dynamicité qui répond à ces trois points et qui permettra les **reconfigurations** dans les approches d'administration autonome.

Nous terminons cette partie par une synthèse des différents modèles.

3.2 Fractal

Fractal [4] a été développé dans les laboratoires de France Telecom R&D et l'INRIA en 2004. Les spécifications principales de Fractal sont :

- permettre des composants composites en vue d'avoir une vue uniforme des applications quel que soit le niveau d'abstraction,
- introduire des composants partagés (sous-composants inclus dans plusieurs composants englobants) en vue de modéliser les ressources et leur partage tout en maintenant l'encapsulation dans des composants,
- fournir la possibilité d'introspection en vue de contrôler et surveiller l'exécution du système sous-jacent,
- permettre de la reconfiguration en vue de déployer et de reconfigurer dynamiquement un système.

Pour permettre aux développeurs d'étendre les mécanismes de reflexion[5], Fractal est extensible. Le modèle propose une large gamme de contrôles applicables aux composants, allant d'aucun contrôle (boîte noire d'objet classique) jusqu'à un contrôle total incluant la manipulation du contenu du composant, du cycle de vie, etc...

3.2.1 Composants

Comme vu dans les spécifications générales, Fractal permet de créer des composants composites. Pour que le composant englobant puisse communiquer avec les composants qu'il englobe, sa membrane, ensemble des interfaces fonctionnelles et non fonctionnelles, peut avoir des interfaces *internes* qui seront reliées aux interfaces fonctionnelles des composants englobés. Les interfaces internes et externes pouvant être reliées ensemble sont appelées interfaces *complémentaires* (voir figure 3.1).

De plus, une spécification originale de Fractal est qu'un composant peut être inclus dans plusieurs autres composants. Un composant de ce type est dit *partagé* entre les autres composants.
Prenons comme exemple un menu et une barre d'outils (voir figure 3.2) comportant un bouton *undo* qui correspond à l'item *undo* du menu, ce qui signifie que le bouton et l'item auront les mêmes réactions.
Il est naturel de représenter l'item du menu et le bouton par un même composant, lui même encapsulé dans deux composants qui sont la barre d'outil et le menu.
Sans le partage de composant, cette solution ne fonctionnerait pas et pour que le composant *undo* ait le même état dans le menu et dans la barre d'outil, il faudrait le mettre au même niveau que ces derniers composants. Avec le partage de composant, l'état du composant *undo* peut être partagé entre le menu et la barre d'outil tout en préservant l'encapsulation.

[5] nous entendons reflexion au sens classique des modèles à objets, à savoir introspection et intercession de structures à objets

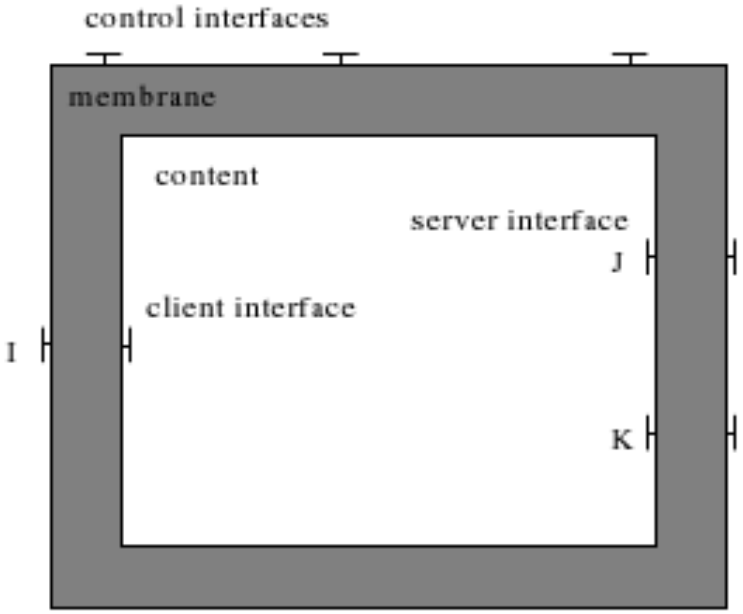

FIG. 3.1 – Un composant composite Fractal avec ses interfaces internes

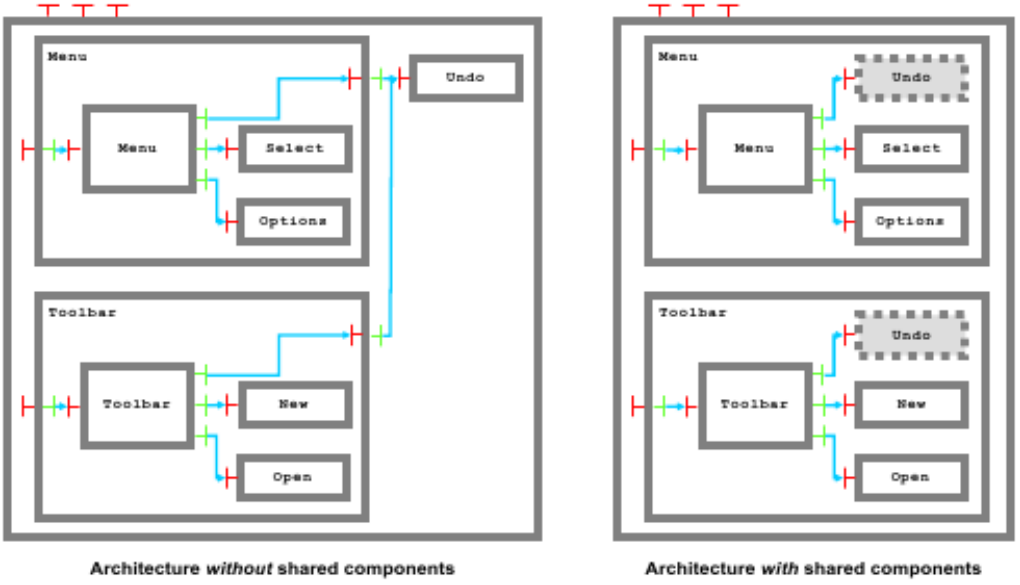

FIG. 3.2 – Composants partagés en Fractal

Le modèle Fractal n'impose pas de contrôleurs prédéterminés au niveau des composants. La spécification de Fractal identifie cependant différents types de membranes qui permettent différents niveaux de contrôle ou de réflexion.
Au plus bas niveau, un composant Fractal est une boîte noire qui ne permet pas de faire de la reflexion. Ces composants, appelés *composants de base*, sont comparables à des objets dans un langage orienté objet.
Au niveau intermédiaire de contrôle, un composant Fractal fournit l'interface **Com-**

ponent qui permet de réaliser de la réflexion sur ce composant, et donc de découvrir les interfaces, internes et externes, clientes et serveur, du composant. A ce niveau, le composant ne fournit pas de fonction de contrôle.
Au plus haut niveau, un composant Fractal peut permettre la manipulation de sa structure interne et fournir des interfaces de reflexion améliorées. Ainsi, la spécification Fractal fournit quelques exemples de contrôleurs utiles qui peuvent être combinés et étendus pour spécifier les composants :

- **le contrôleur d'attribut** : il permet de configurer les attributs. Un composant peut fournir une interface **AttributeController** pour fournir aux autres composants les méthodes *get* et *set* de ses attributs,
- **le contrôleur de liaison** : un composant fournit l'interface **BindingController** pour lier et délier ses interfaces,
- **le contrôleur de contenu** : un composant composite fournit l'interface **ContentController**, pour lister, ajouter ou supprimer des sous-composants,
- **le contrôleur de cycle de vie** : un composant fournit l'interface **LifeCycleController** pour permettre le contrôle explicite du cycle de vie pour faciliter les opérations de reconfiguration. Un contrôleur de cycle de vie basique fournit les opérations *start* et *stop* du composant.

3.2.2 Liaisons

La communications entre deux composants n'est possible que si leurs interfaces sont liées. Fractal supporte les deux types de liaisons : *primitives* et *composites*.
Une liaison primitive au sens Fractal, est une liaison entre une interface serveur et une interface cliente dans le même espace espace d'adressage. Les liaisons primitives sont donc implantées par des pointeurs ou des mécanismes propres aux langages telles que les *références Java*.
Une liaison composite au sens Fractal est un chemin de communication entre un nombre arbitraires de composants. Ces liaisons sont construites avec des liaisons primitives et des composants de liaisons tels que des *stubs*, *skeleton*, ... Une liaison est un composant fractal dont le but est de permettre la communication entre différents composants.
Dans les faits, seules les liaisons primitives sont fournies par le modèle, les liaisons composites devant être explicitement construites à l'aide de composants.
De plus, Fractal introduit une notion de contingence et de cardinalité. La contingence d'une interface indique s'il est garanti que les opérations fournies ou requises d'une interface sont présentes ou non à l'exécution :

- les opérations d'une interface obligatoire sont toujours présentes. Pour une interface client, cela signifie que l'interface doit être liée pour que le composant s'exécute,
- les opérations d'une interface optionelle ne sont pas nécessairement disponibles. Pour une interface serveur d'un composant composite, cela signifie que l'interface interne complémentaire n'est pas forcément liée à un sous-composant.

Pour une interface cliente, cela signifie que le composant peut s'exécuter sans que cette interface soit liée.

La cardinalité d'une interface de type T spécifie le nombre d'interfaces de type T que le composant peut avoir.
Une cardinalité singleton signifie que le composant doit avoir une, et seulement une, interface de type T.
Une cardinalité collection signifie que le composant peut avoir un nombre arbitraire d'interfaces du type T.
Enfin, grâce aux liaisons composites, un framework utilisant Fractal peut connaître le contenu des messages échangés grâce à des composants *intercepteurs*. Cette connaissance permet de rejouer les messages reçus lors d'une phase de réparation pour mettre le composant réparé dans l'état dans lequel il était avant la panne.

3.2.3 Dynamicité

La dynamicité dans Fractal est assurée par la possibilité d'ajouter ou de retirer des composants à une architecture déjà déployée, grâce à la reconfiguration des liaisons et à l'introspection des composants composites. Elle est complète dans ce modèle à composant car elle peut être couplée à une gestion du cycle de vie d'un composant.

3.3 C2

C2 [24] a été créé à l'université de Californie en 1995 pour construire des interfaces graphiques mais a été par la suite utilisé comme un modèle à composant général. Les architectures C2 sont donc vues comme un réseau de composants indépendants et concurrents reliés entre eux par des éléments de routage de messages, les *connecteurs*. Les interfaces graphiques pouvant être vues comme des architectures hiérarchiques, les réseaux de composants C2 sont eux aussi hiérarchiques. Ainsi, un composant ne peut communiquer qu'avec les composants en dessous de lui ou au dessus de lui dans la hiérarchie (voir figure 3.3).

De plus, le mode de communication est différent suivant que le composant communique avec les composants hiérarchiquement au dessus ou hiérarchiquement en dessous. Ainsi, un composant ne peut envoyer que des notifications asynchrones aux composants en dessous tandis qu'ils peuvent émettre des requêtes synchrones aux composants au dessus.

Enfin, les composants ne sont pas directement reliés entre eux mais sont reliés à des connecteurs qui leur permettent de communiquer.

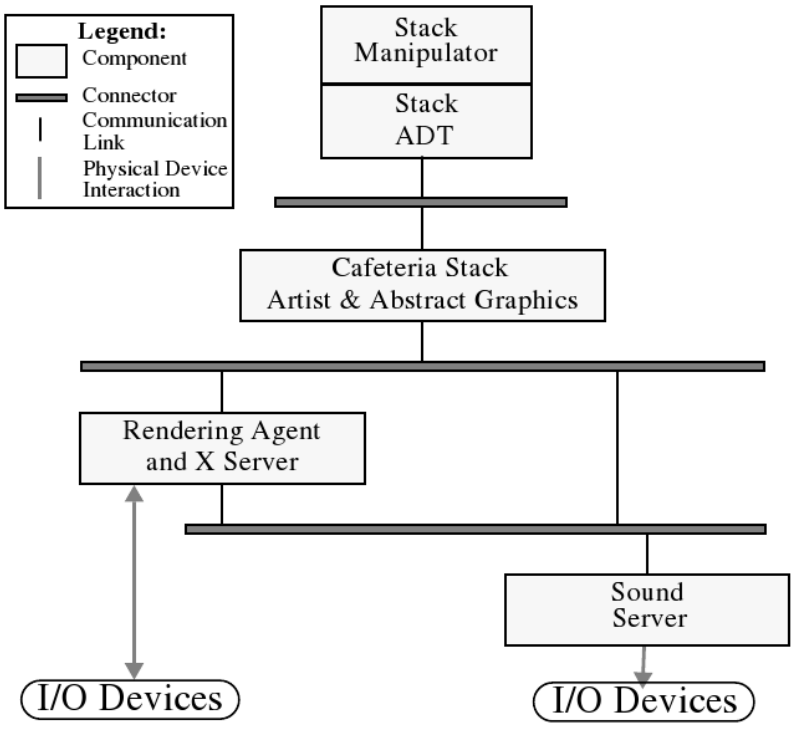

FIG. 3.3 – Système de manipulation audio-vidéo en C2

3.3.1 Composants

Dans ce modèle, les composants peuvent émettre et recevoir deux types de messages :

- les notifications (asynchrones) : une notification "descend" dans une architecture C2. Les notifications annoncent généralement des changements d'état interne du composant qui émet la notification. Les types des notifications qu'un composant peut émettre sont déterminés par sa membrane,
- les requêtes : une requête "remonte" dans l'architecture. Les requêtes sont des directives des composants du "dessous" demandant à un autre composant d'effectuer une action.

Enfin, une architecture C2 est *à plat*, ce qui signifie que rien n'est explicitement prévu pour encapsuler un composant ou un morceau d'architecture dans un autre composant. Il n'y a donc pas de composants composites en C2.

3.3.2 Liaisons

Dans C2, des connecteurs relient les composants entre eux. Cela s'apparente aux liaisons composites de Fractal. Un connecteur peut être connecté à n'importe quel nombre de composant aussi bien qu'à d'autres connecteurs. Un connecteur permet

de router et de *diffuser* des messages. Ils peuvent de plus filtrer les messages. Les connecteurs peuvent fournir plusieurs politiques de diffusion ou de filtrage telles que :

- **pas de filtrage** : chaque message est envoyé à tous les composants reliés au connecteur, vers le bas pour les notifications et vers le haut pour les requêtes,
- **filtrage de notification** : chaque notification est envoyée uniquement aux composants qui sont enregistrés pour cette notification,
- **prioritisation** : le connecteur définit une priorité sur les différents composants connectés, basée sur un ensemble de règle d'évaluation définies par l'architecte lors de la construction de l'architecture. Le connecteur envoie le message aux composants dans l'ordre déterminé par les priorités jusqu'à ce qu'une condition d'arrêt soit rencontrée,
- **ignorance de message** : le connecteur ignore tous les messages qui lui sont envoyé. Ce mode est utile pour isoler une partie de l'architecture ou pour ajouter de manière incrémentale des composants à une architecture existante. L'architecte peut connecter un composant sur le connecteur pour ensuite demander à ce dernier de relayer les messages en changeant sa politique de filtrage.

Le cycle de vie d'un composant est géré au niveau des connecteurs en permettant d'ignorer les messages qui lui sont envoyés. Les messages sont alors perdus dans ce cas sauf si le connecteur est modifié en vue de stocker les messages. Enfin, grâce aux connecteurs, un framework utilisant C2 peut connaître le contenu des messages échangés et, comme Fractal, assurer un rejeu des messages lors d'une phase de réparation.

3.3.3 Dynamicité

La dynamicité dans C2 est assurée par la possibilité d'ajouter ou de retirer des composants à une architecture déjà déployée et grâce à la reconfiguration des liaisons. Elle est complète dans ce modèle à composant car elle peut être couplée à une gestion du cycle de vie d'un composant qui permet d'isoler un ou des composants durant une reconfiguration.

3.4 Weaves

Weaves [11] est un modèle à composants développé par Gorlick et Razouk en 1991 permettant l'exécution concurrente de fragments de logiciels qui communiquent en passant des objets. Weaves se distingue surtout des autres modèles à composants par la richesse de ses outils qui permettent une observation continue du système et par ses possibilités de reconfiguration dynamique : il permet l'insertion automatique d'outils de monitoring[6], le suivi de l'exécution de l'application et il permet enfin de

[6] surveillance

reconfigurer l'application dynamiquement sans interrompre les flux de données.
Les composants Weaves consomment des objets en entrées et renvoient des objets en sortie. Comme en C2, ils ne sont pas directement reliés entre eux mais par l'intermédiaire de composants de liaisons, nommés *queues* et *ports* (voir figure 3.4). Enfin, comme en C2, ce type de liaison se rapproche des liaisons composites en Fractal.

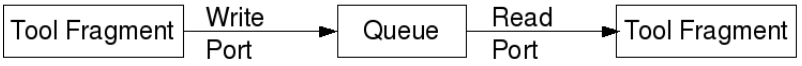

FIG. 3.4 – Une architecture Weaves triviale

3.4.1 Composants

A la différence de Fractal, dans Weaves, les composants se positionnent au niveau du *fragment de logiciel*. Un fragment de logiciel est un morceau d'application dont le grain est de l'ordre de la fonction et qui doit exécuter une seule opération bien définie. Le grain est donc beaucoup plus fin qu'en Fractal.
Un composant doit implanter un ensemble minimum de fonctions de contrôle correspondant aux fonctions du *contrôleur de cycle de vie* de Fractal :

- *start* qui démarre l'exécution du composant,
- *suspend* qui permet d'interrompre l'exécution du composant mais en le laissant dans un état connu qui permettra de le relancer,
- *resume* qui permet de reprendre l'exécution après un *suspend*,
- *sleep* qui permet d'interrompre l'exécution du composant pendant un certain temps,
- *abort* qui arrête immédiatement le composant mais dans un état inconnu.

On distingue deux types de composants :

- le premier type permet l'encapsulation d'un logiciel patrimonial. Weaves est donc le seul modèle à composant étudié qui permet nativement d'encapsuler un logiciel patrimonial,
- le deuxième type est prévu pour des routines écrites directement comme des composants Weaves.

Enfin, Weaves est comme C2, une architecture *à plat*. Les composants ne peuvent pas contenir explicitement d'autres composants Weaves ni un morceau d'architecture Weaves. Weaves ne permet donc pas la création de composant composite.

3.4.2 Liaisons

Dans Weaves, l'interconnexion entre composants est assurée par deux composants : les ports et les queues. Les ports permettent *l'emballage* et le *déballage* des

objets échangés par les composants dans des *enveloppes*. Ils permettent donc d'uniformiser le type d'objet qui transite entre les composants pour que les queues puissent agir de manière banalisée sur les objets qui transitent par elles.

Les queues sont des files finies d'enveloppes. Les enveloppes permettent de masquer le type de l'objet transmis. La figure 3.5 montre l'organisation des ports, queues et enveloppes.

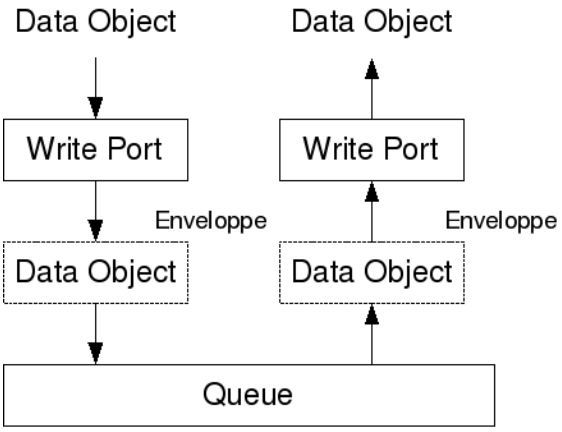

FIG. 3.5 – Organisation des ports, des queues et des enveloppes dans Weaves

Ce système de ports, queues et enveloppes augmente l'interopérabilité entre les différents composants, ces derniers n'ayant pas besoin d'échanger strictement le même type d'objet pour pouvoir communiquer entre eux.

Enfin, les ports permettent aussi d'assurer l'atomicité des transferts des objets. Ainsi, un objet envoyé à un composant ne peut se trouver qu'à un seul endroit parmis deux : soit dans une queue soit dans un composant émetteur ou récepteur. Cette atomicité permet la reconfiguration dynamique sans interrompre les flux de données.

3.4.3 Dynamicité

La dynamicité est assurée dans Weaves en permettant l'ajout ou le retrait de composants déjà déployés ainsi que le changement de liaisons. A l'image de C2, elle est aussi complète grâce au gestionnaire de cycle de vie associé à chaque composant et grâce à l'atomicité des queues. De plus, Weaves permet de connaître le contenu des messages échangés entre les composants au niveau des ports. Cela permet à un framework utilisant Weaves de restaurer un composant dans son *état complet*[7] avant sa panne.

[7] statefull

3.5 Service Component Architecture

Service Component Architecture *SCA* [22] est une spécification écrite par BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, and Sybase en 2005 qui décrit un modèle permettant de construire des applications qui utilisent une architecture orientée services. Le but de SCA est d'unifier les méthodes d'encapsulation et de communication dans les architectures orientées services en fournissant un modèle à composant.

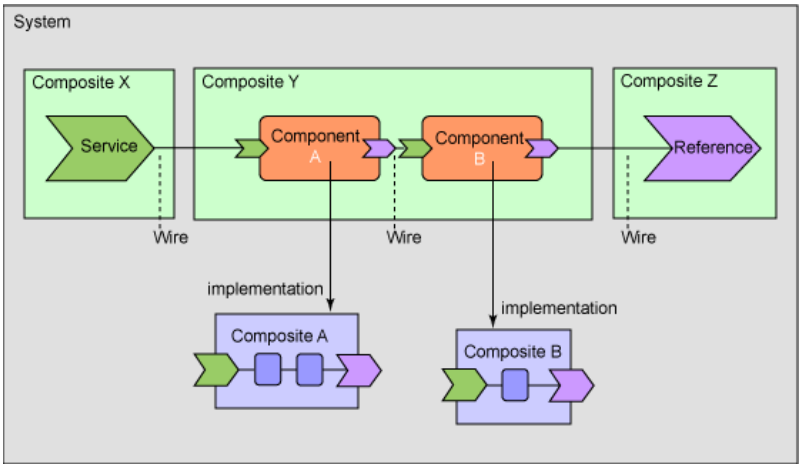

FIG. 3.6 – Une architecture SCA

Une application SCA est composée d'un ou plusieurs composants tel que montré figure 3.6. Ces composants peuvent être écrits dans différents langages (Java, C++, ...) et être répartis sur différents noeuds. Quel que soit le langage d'écriture du composant ou le degré de distribution de l'application, SCA permet de définir les composants et de décrire comment ils interagissent.

3.5.1 Composants

Pour permettre la séparation entre l'implantation et la fonction fournie, SCA définit une notion généralisée du composant. Il définit aussi la manière dont ces composants peuvent être combinés dans des composants composites. Une application complète peut être construite sous forme d'un composite ou combiner différents composites. Comme dans les autres approches, un composite SCA est décrit par un ADL.

Un composant proprement dit est un morceau de code encapsulé. La configuration, définie dans l'ADL, spécifie la manière dont le composant va interagir avec le monde extérieur. Un composant peut être implanté dans différentes technologies, mais quelle que soit la technologie retenue, il doit fournir un ensemble d'abstractions : *services* et *références* qui correspondent aux notions d'interfaces clientes et

serveurs, les *propriétés* qui correspondent aux attributs et enfin, les *liaisons*. Enfin, il n'est pas possible d'ajouter ou de retirer dynamiquement des composants dans une architecture SCA.

3.5.2 Liaisons

Une liaison spécifie la manière dont la communication doit être effectuée entre un composant SCA et une autre entité. A la différence de Fractal, SCA permet d'établir des liaisons vers des composants non SCA ou à l'extérieur des composants composites.
SCA supporte différents types de liaisons, mais un composant un SCA doit au moins fournir les liaison *SCA Services* qui peuvent être utilisés entre composants au sein d'un même composite et les liaisons *Web Services* décrites en WSDL qui permettent de lier un composant à un webservice. Enfin, les liaisons ne sont pas être modifiables lors de l'exécution de l'application.

3.5.3 Dynamicité

Le modèle ne permettant ni l'ajout ni le retrait dynamique des composants ni la modification des liaisons, et les composants ne possédant pas de contrôleur de cycle de vie, SCA n'est à notre sens pas dynamique.

3.6 Synthèse

Les principales spécifications des différents modèles que nous venons de présenter sont résumées dans le tableau 3.1.

Modèle	Composites	Interception [1]	Dynamicité
C2	non	oui	oui
Weaves	non	oui	oui
SCA	oui	non	non
Fractal	oui	oui	oui

[1] de messages entre les composants

TAB. 3.1 – Principales spécifications des approches à composant présentées

Ces différentes approches se ressemblent toutes quant à la définition des composants et des liaisons. Les seules différences se situent au niveau de la topologie des applications qui pourront être décrites.
La dynamicité du modèle nous intéresse car elle permet la reconfiguration des applications ce qui signifie que durant l'exécution des composants peuvent être ajoutés,

supprimés ou déplacés sur d'autres noeuds.
Les composants composites nous intéressent car ils permettent de refléter au mieux une application patrimoniale. Par exemple, une page Web pourra être encapsulée dans le composant du serveur Web.
Hormis Weaves qui possède des composants particuliers pour encapsuler du logiciel patrimonial, les autres approches permettent cette encapsulation par le biais de *wrappers* qui permettent de faire le lien entre un logiciel patrimonial et le code encapsulé dans le composant.

Au vu de ces éléments, Fractal semble être le modèle à composant le plus pertinent dans le cadre d'une utilisation pour de l'administration autonome.

Chapitre 4

Approches d'administration autonome

4.1 Introduction

L'auto administration est l'administration d'un logiciel par un autre logiciel. La majorité des logiciels actuels ne sont pas prévus pour s'auto administrer. Les applications d'auto administration devront donc, pour être utiles et efficaces, permettre d'administrer un logiciel *patrimonial*, soit, un logiciel non prévu pour s'administrer lui même.
On entend par auto administration ou administration autonome deux concepts : le déploiement et la reconfiguration. La reconfiguration inclut la réparation, l'optimisation ou n'importe quelle autre action modifiant l'architecture logicielle.

4.1.1 Le déploiement

Le déploiement permet d'installer sur différents noeuds le logiciel patrimonial. Il comporte la phase de déploiement proprement dite, c'est à dire la copie des binaires et des librairies sur les noeuds distants et la configuration du logiciel déployé. En effet, suivant le type de noeud, voire l'adresse du noeud ou même ses spécificités (système de fichiers répartis, multiprocesseurs, ...) les logiciels devront être configurés différemment. L'approche d'auto administration doit donc prendre en compte les spécificités du noeud et configurer les logiciels pour qu'ils puissent exploiter de manière cohérente le noeud sur lequel ils ont été déployés.

4.1.2 La reconfiguration

La reconfiguration inclut toutes les actions qui modifient l'architecture logicielle. Les principales sont la réparation et l'optimisation.

La réparation vise à remettre en ordre de marche un logiciel ou un morceau de logiciel défaillant alors que l'optimisation permet à l'application déployée de s'adapter à l'environnement dans lequel elle est déployée et qui peut varier au cours du temps.

La reconfiguration est rendue difficile par le nombre de noeuds sur lequel le logiciel est déployé. En effet, il est difficile pour l'administrateur de surveiller en permanence l'état des logiciels déployés pour les réparer le cas échéant ou de surveiller les modifications des différents environnements pour optimiser le logiciel sur des centaines de noeuds. En plus de cette phase de surveillance, l'administrateur doit aussi exécuter des actions en vue de réparer ou d'adapter le logiciel à son nouvel environnement.

Une autre difficulté lors de la reconfiguration sont les *reconfigurations en cascade*. En effet, si un noeud tombe en panne, les logiciels présents sur ce noeud seront redéployés sur un autre noeud. Il faut donc que tous les logiciels afférents aux logiciels redéployés soient à leur tour reconfigurés pour prendre en compte la modification de l'architecture.

Enfin, à la suite d'une reconfiguration, il faut vérifier que le logiciel déployé reste cohérent. En effet, dans une architecture Apache-Tomcat-MySQL par exemple, il faudra en permanence une instance de chaque serveur. Il ne faut pas par exemple que suite à une reconfiguration, un serveur soit détruit et que son nombre d'instance soit nul.

L'approche d'auto administration devra donc surveiller le logiciel, entreprendre des actions pour le reconfigurer et enfin vérifier la cohérence de l'architecture suite à la reconfiguration. Ceci est généralement fait grâce à une boucle *Boucles sondes-actuateurs-manager*.

Enfin, généralement, la reconfiguration est implantée par une boucle **sondes-actuateurs-manager** où les sondes déclenchent une notification, le manager décide de l'action à mener en fonctions de politiques et les actuateurs procèdent aux modifications.
Une telle boucle est détaillée dans l'approche Jade (voir section 4.2.2).

4.1.3 Plan

Nous présentons dans cette section quatre approches permettant l'auto administration de logiciel : Jade, FDF, ArchStudio et enfin les travaux de O. Kephart et son

équipe.
Les approches d'administration autonome se ressemblant toutes, nous avons choisi de présenter ici des approches qui ne fournissent pas forcément toutes une auto administration complète (déploiement et reconfiguration) mais qui mettent l'accent sur un point particulier de l'auto administration.
Ainsi Jade et les travaux de O. Kephart font de l'auto administration complète mais de manière différente tandis que ArchStudio propose des techniques de reconfiguration et FDF des techniques de déploiement.
Nous commençons par Jade qui nous paraît être l'approche répondant le plus aux besoins de l'auto administration et nous comparons les autres approches à Jade.
Pour chacune de ses approches, nous étudions :

- le modèle à composants utilisé et la gestion des logiciels patrimoniaux. La gestion des logiciels patrimoniaux est importante car elle permets aux administrateurs de faire de l'auto-administration de logiciels non conçus pour,
- le déploiement et la manière dont l'architecture logicielle est décrite. La manière dont l'architecture logicielle est décrite est importante car si elle n'est pas ergonomique elle risque de rebuter l'administrateur et lui faire perdre du temps lors de l'apprentissage. De plus, si elle est trop compliquée, elle sera source d'erreurs,
- la reconfiguration et les techniques mises en oeuvre pour garder l'architecture logicielle patrimoniale cohérente. La conservation de l'architecture logicielle dans un etat cohérent[1] est important car dans le cas contraire, les logiciels patrimoniaux risquent de ne pas fonctionner correctement voire de ne pas fonctionner du tout. De plus, dans cette section, nous nous intéressons aussi à la manière dont les politiques de reconfiguration sont décrites et implantées. Comme pour la description de l'architecture logicielle, l'ergonomie de la description des politiques de reconfiguration est importante pour éviter a l'administateur de perdre du temps et de faire des erreurs.

Nous terminons cette partie par une synthèse des différents approches.

4.2 Jade

Jade [3] est une approche d'auto administration basée sur Fractal développée à l'INRIA à Grenoble en France en 2004 qui permet de déployer des applications patrimoniales et de les reconfigurer.
L'unité d'administration dans Jade est le composant. Chaque composant encapsule une partie de l'application administrée grâce à des wrappers. Les wrappers sont des composants Fractal développés en Java qui doivent implanter les interfaces de la membrane des composants Fractal. Parmi ces interfaces, une interface fonctionnelle permet d'exécuter des actions simples sur les logiciels déployés telles que le démar-

[1] on entends par cohérence de l'architecture logicielle la conservation de cette dernière dans un etat permettant aux logiciels déployés de pouvoir fonctionner.

rage, l'arrêt ou la configuration. Les wrappers traduisent donc les appels de Jade en appels système permettant d'effectuer ces actions sur les parties du logiciel déployé. Les wrappers sont la partie la plus sensible de cette approche d'auto administration car ils manipulent directement la couche patrimoniale mais aussi la partie la plus difficile à développer pour l'utilisateur car ils demandent des compétences Fractal pour implanter les interfaces non fonctionnelles et des compétences Java pour implanter le code de démarrage, configuration, ...

4.2.1 Le déploiement

Jade utilise l'ADL Fractal pour décrire l'application ainsi que le mécanisme de déploiement intégré à Fractal pour déployer les wrappers sur les noeuds.
Fractal permet de déployer des composants sur des machines distantes en utilisant un mécanisme d'appel à distance proche de RMI[2] nommé Fractal RMI.
En déployant ainsi les composants, Jade déploie donc les wrappers qui pourront agir localement sur les logiciels patrimoniaux. Les mécanismes Fractal standards de navigation dans l'architecture logicielle permettent aux différents wrappers de se configurer en fonction des logiciels afférents.
Ainsi dans le cadre d'une architecture trois tiers J2EE, le wrapper d'un serveur Tomcat pourra demander l'adresse IP[3] du serveur Apache auquel il est relié, si le serveur Apache définit une propriété comportant sa propre adresse.

4.2.2 La reconfiguration

La reconfiguration est basée sur la notion de noeud. Un noeud est un composant composite Fractal qui contient les wrappers des logiciels patrimoniaux. Ainsi, toute action de réparation ou d'optimisation se fait au niveau du noeud. La structure générale de la reconfiguration est une boucle décrite dans la figure 4.1.
Les différents éléments constituant cette boucle sont :

- sondes (sensors) : pour reconfigurer, Jade doit observer les évènements suivants : changement d'état interne des composants (pour la réparation), taux d'utilisation des ressources sur le noeud (pour l'optimisation),
- actuateurs (actuators) : ils permettent à Jade d'exécuter des actions de reconfiguration sur les composants administrés. Ces actions sont l'arrêt et le redémarrage du composant ou l'ajout et son retrait,
- manager : il implante la partie analyse et décision de la boucle de réparation. A partir des informations renvoyées par les sondes, il peut piloter les actuateurs pour reconfigurer l'application.

L'état de l'application est connu grâce à un service appelé *System Representation*

[2]Remote Method Invocation - Appel de méthode distante
[3]Internet Protocol

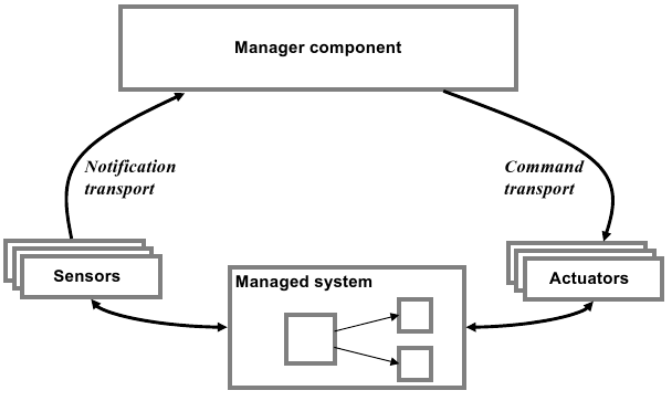

FIG. 4.1 – La boucle de contrôle de réparation

qui possède la même structure que la représentation à composant de l'application déployée mais sur un noeud considéré comme fiable. En effet, Jade répartit les wrappers et donc les composants Fractal sur différents noeuds. L'architecture logicielle est donc sensible à une panne de noeud ce qui signifie que si un noeud tombe en panne, l'état du noeud qui comprend les logiciels déployés sur ce noeud et leur état est perdu. Le System Representation déployé sur un noeud fiable permet donc de palier à ce problème.

Un exemple de politique implantée dans le manager de réparation pourra être :
- sur la réception d'une notification de panne d'un composant J2EE, exécuter les commandes suivantes : retirer les liaisons du composant en panne puis démarrer un nouveau composant en remplacement de l'ancien et enfin rétablir les liaisons,
- sur la réception d'une notification de panne de noeud, exécuter les commandes suivantes : retirer les liaisons de tous les composants présents sur le noeud en panne, demander un nouveau noeud à l'allocateur de noeud puis installer la même configuration de composant sur le nouveau noeud et enfin rétablir toutes les liaisons.

Les politiques de reconfiguration sont écrites en Java et implantées en Java au niveau du manager. Elles ne sont pas reconfigurables sans recompiler Jade.
Enfin, la cohérence de l'architecture n'est pas vérifiée lors d'une reconfiguration.

4.3 ArchStudio

Cette approche développée à l'université de Californie en 1998 [20], est basée sur le modèle à composant C2. Elle permet l'auto administration d'application écrites en C2. Elle ne gère donc pas les logiciels patrimoniaux. Elle est intéressante dans

le cadre de notre étude sur l'auto administration car les auteurs ont fait un travail important sur les politiques de reconfiguration et de maintien de cohérence.

4.3.1 Le déploiement

ArchStudio utilise le mécanisme de déploiement fourni par le framework *Java-C2*, une implantation de C2 en Java. L'application est décrite dans un fichier ADL. Ce fichier contient les descriptions des liaisons entre les composants et les connecteurs ainsi que les morceaux de code contenus dans les composants. On ne parle pas ici de wrappers comme dans Jade car l'application est déjà écrite en C2.

4.3.2 La reconfiguration

La reconfiguration en C2 est surtout basée sur l'optimisation de l'application et entres autres son dimensionnement dynamique. Les auteurs de cette approche insistent sur le fait qu'effectuer des changements à une architecture logicielle n'est raisonnable que dans le cas où il existe une méthode de vérification pour s'assurer de la correction des modifications apportées en vue de ne pas laisser l'application reconfigurée dans un état instable. Ils proposent donc un ensemble de règles que les approches d'auto administration doivent respecter pour que les reconfigurations aboutissent toujours à une application cohérente.
Ainsi, la reconfiguration se fait au moyen d'une boucle sondes-actuateurs-manager qui à la différence de la boucle de Jade intègre une notion de contre réaction permettant de vérifier la validité des changements.
La reconfiguration avec ArchStudio se fait en définissant des modifications d'architecture grâce aux outils *Argo* et *ArgoShell*. Ces outils génèrent un fichier de modification d'architecture qui peut alors être pris en compte par le système grâce à un outil appelé *Extension Wizard*. Les politiques de reconfiguration sont donc modifiables durant l'exécution de l'application.

4.4 FDF

FDF [9], pour Fractal Deployment Framework, développée au LIFL à Lille en France en 2005, est une approche qui est spécialisée dans le déploiement d'application patrimoniales et qui repose comme Jade sur le modèle à composant Fractal.
A la différence de Jade, dans FDF, tout est réifié en composant[4], ce qui signifie que le grain est beaucoup plus fin.
Par exemple, le protocole de copie du logiciel patrimonial sera un composant Fractal.

[4]everything is reified as components

FDF fournit une librairie de composants de déploiement et permet aux développeurs d'écrire leurs wrappers, toujours sous forme de composants.
Ainsi, la représentation d'une application avec FDF se fera en assemblant ces différents composants de bas niveau et ces wrappers dans un composite.

4.4.1 Le déploiement

L'idée de FDF est de fournir une librairie de composants permettant de construire une solution de déploiement adaptée à chaque application patrimoniale et son environnement.

Tout ces composants, participant au déploiement, sont unifiés en composants Fractal. Ainsi le protocole de copie, par exemple, est vu comme un logiciel patrimonial. Les composants participant au déploiement pour FDF sont :
- les noeuds physique sur lesquels les morceaux d'application seront déployés,
- les protocoles de prise de contrôle à distante tels que SSH[5], Telnet, ...,
- les protocoles de transfert de fichiers tels que FTP[6], SCP[7], ...,
- les *shells* tels que SH, CSH, Windows Command Shell, ...,
- les variables d'environnements des shells,
- les middlewares tels que le JRE[8], CORBA Naming Service, ...,
- les serveurs d'applications tels que OpenCCM[9], ...

Ces composants, fournis par FDF, sont des wrappers de logiciels patrimoniaux et doivent donc implanter l'interface métier de la membrane qui inclut les méthodes *install(), start(), stop(), uninstall()*. Comme dans Jade, cette interface permettra à FDF d'orchestrer le déploiement.

FDF fournit un outil graphique, *FDF Explorer*, qui permet de combiner les composants pour obtenir un ensemble qui représente l'application à déployer. Cet ensemble de composants est placé dans un composant composite Fractal qu'il faut alors démarrer grâce au contrôleur de cycle de vie de Fractal.
Les liaisons entre les composants représentent les dépendances entre composants et donnent un ordre de déploiement.

Par exemple, un composant *shell* qui représente un shell tel que SH permet grâce à son interface serveur Fractal d'exécuter des commandes. Pour ce faire, il aura besoin d'un protocole de prise de contrôle à distance pour envoyer les commandes à la machine distante. Il sera donc lié au sens des liaison Fractal à ce composant. Ce dernier composant aura besoin de connaître des informations tels que le port, l'utilisateur,

[5] Secure SHell
[6] File Transfert Protocol
[7] Secure CoPy
[8] Java Runtime Environment
[9] Open Corba Component Model

.... Il sera donc lui même relié à d'autre composants qui permettront de lui fournir ces informations.

4.4.2 La reconfiguration

Les dernières orientations de ce projet proposent d'étendre FDF pour permettre le déclenchement de règles de reconfiguration afin d'effectuer des reconfigurations dynamiques. Ces travaux n'ont toutefois pas encore abouti.

4.5 Approche de O. Kephart

O. Kephart et son équipe à IBM [12] proposent une approche originale qui s'appuie sur des *composants autonomes* et un système qui s'auto administre grâce à ces différents composants autonomes. Le modèle à composant utilisé n'est pas fixé mais ils préconisent d'utiliser des modèles à composant orientés webservices tel que SCA. Les seules applications patrimoniales gérées doivent être écrites dans un modèle à composant orienté webservice.
A la différence des autres approches, il n'existe pas de gestion centralisée qui comporte les boucles de reconfiguration ou de réparation, mais une intelligence décentralisée au niveau des composants autonomes.
Le concept clé de l'approche est donc le composant autonome. Un composant autonome est un composant qui doit lui-même déterminer son comportement en fonction de règles et interagir lui-même avec d'autres composants pour fournir ou consommer des services informatiques.
Deux types de composants coexistent : des composants qui encapsulent les services des applications patrimoniales et des composants de plus haut niveau dotés de pouvoir d'administration tels que des gestionnaires de charge ou des allocateurs de ressources. Ces composants de plus haut niveau pourront aussi permettre d'aider les autres composants à effectuer leur tâche en se positionnant comme répertoire de règle, sentinelle, allocateur ou registres par exemple.

4.5.1 Le composant autonome

Un composant autonome doit pouvoir se gérer tout seul, ce qui signifie qu'il doit être en mesure de :

- s'auto configurer,
- s'auto réparer en cas de panne,
- s'auto optimiser,
- s'auto sonder.

En d'autre terme les différents problèmes doivent être traités localement et non pas auprès d'un niveau supérieur comme dans les autres approches présentées. Deux composants sont dit liés si ils se sont mis d'accord sur le service que l'un fournira à l'autre. Enfin, il existe des composants dont le comportement est prédéfini comme par exemple le **registre** qui permet aux composants de se découvrir entre eux, se lier entre eux et exposer les services qu'ils proposent.

4.5.2 Les règles

Les règles permettent au système d'atteindre un but. On distingue trois types de règles :

- les règles d'auto administration qui s'appliquent au niveau du composant. Elles sont généralement simples et de la forme **si-alors-sinon**. Ainsi une règle pourra être : "IF (ResponseTime) > 2 sec) THEN (Increase CPU share by 5%)",
- les règles de *but*, de niveau intermédiaire qui s'appliquent à des composants liés, qui peuvent être par exemple : "ResponseTime must not exceed 2s",
- les règles *utilitaires*, de haut niveau, qui sont des fonctions d'évaluation permettant de chiffrer la pertinence de l'état atteint par le système par rapport à ses spécifications. Ce sont les contraintes du système.

Les composants autonomes qui utilisent les règles de but ou les règles utilitaires doivent être suffisamment évolués pour les traduire en actions en vue d'atteindre la spécification attendue et décrite par la règle.

4.5.3 Le déploiement

Le déploiement peut être assuré par les mécanismes du modèle à composant sous-jacent. La configuration des relations peut donc être faite lors du développement de l'application.
Cependant, l'équipe a exploré une autre approche nommée *l'auto assemblage guidé par le but*[10]. Ils ont monté que le système était plus robuste avec ce type de configuration car les décisions de liaison sont prises au niveau local et sont donc très dépendantes de l'environnement auquel elle peuvent alors s'adapter automatiquement. La configuration se fait donc suivant les phases suivantes :

- une règle définissant le système est donnée. Cette règle pourra être : "transforme toi en serveur d'application". En plus de cette règle, une méthode est fournie pour contacter le composant particulier *registre*,
- lors de l'initialisation, chaque composant contacte le registre pour trouver les composants susceptibles de fournir les services dont il a besoin. Suite à ce contact au registre, les composants se contactent entre eux pour se lier,

[10] goal-driven-self-assembly

– une fois que tous les composants ont atteint leurs buts, le système est dit *assemblé*.

4.5.4 La reconfiguration

Deux aspects de la reconfiguration sont gérés par l'approche de Kephart : la réparation et l'optimisation.
Pour la réparation, la détection de panne se fait au niveau de chaque composant qui doit surveiller ses entrées. Si un composant détecte que le composant auquel il est lié n'assure plus le contrat passé entre eux, il peut alors terminer la relation avec ce composant et en créer une nouvelle avec un autre composant. Dans le cas ou le service est *sans-état*[11], cela peut se faire sans problème particulier en contactant le registre et en répétant les phases d'initialisation.
Dans le cas d'un service *plein-état*[12], les auteurs précisent juste que les entrées du composant doivent être réactives à une panne et que des approches ont été proposées dans le cadre des *webservices distribués vaguement couplés*[13].
Ils proposent cependant une méthode qui consiste à se lier à deux dupliquas du composant et d'effectuer les mêmes opérations sur chacun d'eux pour maintenir leur état interne en cohérence. En cas de problème sur un composant, il peut être détruit et un nouveau pourra être créé en récupérant l'état interne de celui qui n'a pas eu de panne.
Cette approche ressemble au System Representation utilisé dans les autre approches d'auto administration.

Concernant l'optimisation du système, elle est liée à l'optimisation de chaque composant. Cependant, un comportement correct au niveau local ne donne pas forcément un comportement correct au niveau du système. De plus, des composants pourront souvent entrer en conflit pour se lier à une ressource.

Pour permettre de régler les conflits de ressources, ils s'appuient sur le patron du *marché*[14]. Les composants seront classés en deux parties, les *acheteurs* et les *vendeurs*. Chaque acheteur devra fixer le prix du service dont il a besoin et chaque vendeur le prix du service qu'il fournit. Ainsi, les composants ne pourront se lier à une ressource que lorsque le prix de la ressource correspondra au prix qu'il est prêt à y mettre.

Ils ne précisent pas de quelle manière l'architecture de l'application peut rester cohérente. Il est tout de même raisonnable de penser que des règles de but pourront permettre à l'architecture de rester cohérente.

[11] stateless
[12] statefull
[13] loosely-coupled distributed web services
[14] market-like mechanism

4.6 Synthèse

Les principales caractéristiques des différentes approches que nous venons de présenter sont résumées dans le tableau 4.1.

Modèle	**M. à C.** [1]	**Patri.** [2]	**Archi.** [3]	**Politiques** [4]	**Cohérence** [5]
Jade	Fractal	oui	ADL	Programmation	non
FDF	Fractal	oui	Outil dédié	Sans objet	Sans objet
ArchStudio	C2	non	ADL	Outil dédié	oui
Kephart	SCA [6]	non	Automatique	Automatique	Automatique

[1] modèle à composant utilisé
[2] gestion des applications patrimoniales
[3] méthode de description de l'architecture
[4] description et mise en oeuvre des politiques
[5] vérification de la cohérence de l'application après reconfiguration
[6] en fait, tout modèle basé sur des web services

TAB. 4.1 – Principales caractéristiques des approches présentées

On remarque que le système le plus abouti dans l'approche d'auto administration centralisée est Jade. Cependant, en plus de ne pas gérer la cohérence suite à une reconfiguration, il impose un développement en Java des politiques de reconfiguration et une description de l'application en ADL.
FDF fournit lui un outil pour décrire l'application mais ne permet pas de reconfiguration tandis que ArchStudio fournit des outils pour décrire les politiques de reconfigurations mais pas d'outils pour décrire l'application et ne permet pas la gestion des applications patrimoniales.
Enfin, l'approche de O. Kephart semble prometteuse mais se situe à cheval entre l'intelligence artificielle et l'auto administration. Elle ne semble pas assez mûre pour permettre son utilisation dans un milieu autre que la recherche.

Chapitre 5

Orientations générales

En vue de de souligner les difficultés techniques que rencontre un administrateur pour configurer un logiciel de gestion autonome et l'application gérée, nous décrivons dans ce chapitre la mise en oeuvre de Jade pour administrer l'application Diet. Nous présentons tout d'abord les différents serveurs de Diet et les fichiers de configuration, nous expliquons les wrappers qui permettent à Jade d'administrer Diet puis les opérations à mener permettant à Jade de déployer et reconfigurer Diet. Nous présentons ensuite les différentes contributions que nous apportons dans cette thèse pour simplifier l'administration et résoudre les problèmes posés par l'administration autonome de logiciel patrimonial.

5.1 Les serveurs de Diet

Comme vu lors de la problématique, Diet est composé de trois types de serveurs : les MA, les LA et les SeD. D'un point de vue patrimonial, ces trois serveurs se présentent sous forme de deux binaires :

- *dietAgent* qui se comporte comme un LA ou un MA en fonction de sa configuration,
- *server* qui est le SeD.

Étudions maintenant les fichiers de configuration de chacun de ces serveurs. Chacun d'entre eux possède deux fichiers de configuration : un fichier pour configurer le serveur lui même et un fichier pour permettre au serveur de retrouver le serveur de nom Corba.

Dans le cadre de la configuration dans Diet, les LA et les MA sont configurés par un fichier qui contient, *entres autres*, les options suivantes :

- *agentType* pour définir si le binaire se comporte comme un MA ou comme un LA,
- *parentName* pour définir le nom du LA ou MA parent dans le serveur de nom Corba,
- *name pour* définir le nom du serveur dans un serveur de nom pour Corba,

Les SeD sont eux configurés par un fichier qui comporte les options suivantes :

- *parentName* pour définir le nom du LA ou MA parent dans le serveur de nom Corba,
- *name* pour pour définir le nom du serveur de calcul dans un serveur de nom pour Corba.

Et enfin, chacun d'entre eux est accompagné du fichier de configuration pour Corba qui contient l'option suivante :

- *InitRef :NameService=corbaname : :adresse* pour spécifier l'adresse du serveur de nom Corba pour retrouver les autres serveurs.

Comme souligné dans la problématique, on se rend compte que les fichiers de configuration sont interdépendants. Ainsi, chaque serveur Diet doit connaître l'adresse courante du serveur de nom Corba et chaque SeD doit connaître le nom de son père. Ce qui ne paraît pas grand chose à petite échelle se révèle problématique à grande échelle. En effet, dans le cas d'une panne du noeud qui contient le serveur de nom Corba, tous les serveurs Diet devront être reconfigurés. De même, dans le cas d'une reconfiguration où un ensemble de SeD doit changer de père, tous les SeD concernés devront être reconfigurés. Enfin, dans le cas d'une reconfiguration ajoutant des serveurs, un nom différent doit automatiquement être attribué au serveur.

Ensuite, chacun de ces serveurs doit être exécuté d'une manière différente. Le binaire des LA et des MA est lancé en passant en paramètre le nom du fichier de configuration pour Diet tandis que le nom du fichier de configuration pour Corba est contenu dans une variable d'environnement. Le binaire des SeD est lui lancé avec le nom de son fichier de configuration et l'ensemble des calculs qu'il est en mesure d'effectuer. Le nom du fichier de configuration pour Corba est lui aussi passé dans une variable d'environnement. Le nom des fichiers peut rester le même d'un noeud à l'autre mais par contre l'emplacement de ces fichiers peut varier d'un noeud à l'autre en fonction du répertoire de déploiement sur le noeud. L'application de gestion autonome doit donc aussi prendre en compte le répertoire de déploiement en vue de pouvoir exécuter les serveurs.

Pour conclure cette partie sur les serveurs de Diet, les figures 5.1 et 5.2 montrent les fichiers de configuration que l'application de gestion autonome doit générer pour un LA.

```
name = LA1
parentName = MA1
agentType = DIET_LOCAL_AGENT
```

FIG. 5.1 – Fichier de configuration Diet pour un LA

```
InitRef = NameService=corbaname::localhost
```

FIG. 5.2 – Fichier de configuration Omni pour un LA

5.2 Administrer Diet par Jade

Nous suivons dans cette section les quatre étapes requises pour permettre l'administration de Diet par Jade. Nous voyons d'abord les wrappers permettant le contrôle de chaque morceau de l'application patrimoniale, puis le composant composite dans lequel ces wrappers sont insérés pour décrire l'architecture de l'application patrimoniale, puis un composant particulier permettant le déploiement de l'application et enfin un composant permettant la reconfiguration.

Les wrappers Ce sont des composants Fractal qui seront administrés par Jade. Leur code est en Java tandis que leur interface est en ADL Fractal. Ils doivent implanter trois interfaces : *Resource*, *BindingController* et enfin *AttributeController*. L'interface *AttributeController* permet d'implanter le contrôleur d'attribut Fractal. Grâce à elle, les autres composants peuvent récupérer les attributs du composant wrappé. Le SeD pourra par exemple connaître le nom de son père en interrogeant le contrôleur d'attribut du LA. Les méthodes à implanter sont de type *getter et setter* pour être génériques. Ainsi, le wrapper du LA doit implanter, entre autre, les méthodes présentées figure 5.3 pour le contrôleur d'attributs.

```
public String getDirLocal() {
  return dirLocal;
}

public void setDirLocal(String dl) {
  dirLocal = dl;
}

public void setName(String n) {
  name = n;
}

public String getName() {
  return name;
}
...
```

FIG. 5.3 – Quelques méthodes du contrôleur d'attribut pour un LA

L'interface *BindingController* permet de gérer les liaisons entre composants. Ainsi

lorsque des SeD doivent changer de LA, toutes les liaisons entre les SeD déplacés et le LA d'origine sont cassées et de nouvelles liaisons sont créées vers le LA de destination. Le wrapper du LA doit donc implanter, entres autres, les méthodes présentées figure 5.4 pour le contrôleur de liaison.

```
public void bindFc(final String itfName, final Object itfValue) {
  if (itfName.startsWith(LA_PREFIX)) {
    ma.put(itfName,itfValue);
  }
}

public void unbindFc(final String itfName) {
  if (itfName.startsWith(LA_PREFIX)) {
    ma.remove(itfName);
  }
}
...
```

FIG. 5.4 – Quelques méthodes du contrôleur de liaison pour un LA

L'interface *Resource* permet la gestion du logiciel patrimonial. Elle permet la configuration, le démarrage et d'autres méthodes de gestion patrimoniale de ce dernier. Le wrapper du LA doit donc implanter, entre autre, les méthodes présentées figure 5.5 pour le contrôleur de logiciel patrimonial. Cette interface est la plus difficile à implanter, nous ne la commentons pas car elle est *ad-hoc* mais en donnons un extrait pour illustrer la difficulté d'implantation.

```
public void configure() throws JadeException {
  Logger.println("mkdir " + dirLocal);
  ExecutableCmd.syncExec("mkdir " + dirLocal, null);
  updateConfigFile();

  Enumeration en = la.elements();
  Object da;
  while (en.hasMoreElements()){
    da = en.nextElement();
    ((Resource) da).configure();
  }

  Logger.println(DebugResources.on,"LocalAgent.configure");
}

private void updateConfigFile() throws JadeException {

  try {
    PrintWriter p = new PrintWriter(dirLocal + "/"+this.name+".cfg");
    p.println("agentType=DIET_LOCAL_AGENT");
    p.println("parentName="+parentName);
    p.println("name="+name);
    p.close();
    } catch(Exception e) {
      throw new JadeException("Cannot write la config file");
  }
}


public void start() throws JadeException {
  Logger.println(DebugResources.on,"LocalAgent started");
  String command;
  this.stop();
  if(outputLog.equals("on")) {
    command = new String(dirInstall + "/bin/starter "+ dirInstall + "/bin/dietAgent
        true "+dirLocal+"/StdOut.log "+dirLocal+"/StdErr.log  dietAgent "+dirLocal
        +"/"+ this.name+".cfg -NULL "+envPath+" -NULL");
  } else {
    command = new String(dirInstall + "/bin/starter "+dirInstall + "/bin/dietAgent
        false  dietAgent "+dirLocal+"/"+ this.name+".cfg -NULL "+envPath+" -NULL");
  }
  agent_pid = ExecutableCmd.asyncExec(command, null);
  try {
    Thread.sleep(2000);
  } catch (Exception e) {
    throw new JadeException ("LocalAgentImpl: error in start" + e + "\nWhen
        executing : "+command);
  }
  Enumeration en = la.elements();
  Resource da;
  while (en.hasMoreElements()){
    da = (Resource)en.nextElement();
    da.start();
  }
}

...
```

FIG. 5.5 – Quelques méthodes du contrôleur de logiciel patrimonial pour un LA

Enfin, la figure 5.6 montre le fichier ADL représentant le composant LA. C'est dans ce fichier que l'interface du composant est définie.

```
<?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/basic.dtd">


<definition name="fr.jade.resources.lagent.LocalAgentType">
  <interface name="sed" role="client" signature="fr.jade.resources.resource.
      Resource" cardinality="collection" />
  <interface name="la" role="server" signature="fr.jade.resources.resource.Resource
      " cardinality="singleton"/>
  <content class="fr.jade.resources.lagent.LocalAgentImpl"/>
  <attributes signature="fr.jade.meta.api.control.GenericAttributeController"/>
  <controller desc="parametricprimitive"/>
</definition>
```

FIG. 5.6 – Le composant qui permet la gestion du LA en Fractal

Le composant qui décrit l'architecture globale de l'application Ce composant est un composant composite qui contient les composants utilisés pour le déploiement. Il permet aussi d'affecter des valeurs aux attributs des composants développés pour chaque partie de l'application. Un extrait est donné figure 5.7

```
<?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="fr.jade.test.diet.StartDietType">

  <interface name="service" role="server" signature="fr.jade.service.Service"/>


  <component name="la_0" definition="fr.jade.resources.lagent.LocalAgentType">
    <attributes signature="fr.jade.meta.api.control.GenericAttributeController">
      <attribute name="dirLocal" value="/tmp/vittoz_diet_local/la_0"/>
      <attribute name="name" value="LA_0"/>
      <attribute name="parentName" value="MA_0"/>
    </attributes>
    <virtual-node name="node1"/>
  </component>

...

</definition>
```

FIG. 5.7 – Une partie du fichier ADL qui représente Diet pour Jade

Cet ADL permet d'ajouter un LA nommé *la_0* dans l'application patrimoniale.

Le composant de déploiement Ce composant permet d'appeler les méthodes de démarrage des composants contenus dans le composant composite décrivant l'application patrimoniale. Il détermine les dépendances dans l'ordre de démarrage des serveurs et il est appelé implicitement au démarrage de Jade et explicitement lors des reconfigurations.

La reconfiguration La reconfiguration dans Jade est aussi faite dans un composant, le *RepairManager*. L'administrateur doit implanter des méthodes qui permettent à Jade d'appliquer des politiques de reconfiguration lors de certains évènements. Ces méthodes permettent entre autre de réparer un composant.

5.3 Synthèse

Cet exemple nous permet de prendre conscience de plusieurs défauts inhérents aux applications telles que Jade qui se configurent en ADL et (ou) en code :

- généralement, l'administrateur se retrouve en position de développeur pour ecrire les wrappers. Le développement est souvent hors de compétence des administrateurs, le code des wrappers ne sera donc généralement pas bien écrit et sera donc source d'erreurs. Jade demande donc aux acteurs de sortir de leur rôle,
- Jade n'est pas utilisable sans connaître les langages Java et l'ADL de Fractal. L'acteur qui écrit les wrappers devra donc apprendre ces deux langages. Ceci est une perte de temps et est également source d'erreurs,
- l'utilisation de l'ADL pour décrire l'application ne permet pas une description en intension. Si 1000 SeD doivent être déployés initialement, le fichier du composant qui décrit l'application sera très long et donc lui aussi source d'erreurs. De plus les fichiers ADL ne sont pas modulaires et sont donc difficilement réutilisables,
- les politiques de déploiement des logiciels patrimoniaux sur des noeuds particuliers font généralement parti du modèle a composant sous-jacent. Il est donc difficile de les modifier et de pouvoir ainsi de pouvoir spécialiser des noeuds pour un logiciel ou une partie de logiciel en particulier,
- enfin, l'écriture de code pour décrire les politiques de reconfiguration ou pour permettre des actions sur la couche patrimoniale rends la réutilisation difficile car le code écrit est trop axé sur l'application patrimoniale à déployer.

5.4 Nos propositions

Pour parvenir à éliminer ces problèmes, nous avons tout d'abord découpé notre application de gestion autonome en trois niveaux d'utilisation distincts permettant à tous les acteurs travaillant sur une application patrimoniale de rester dans leur domaine de compétence lors du passage à l'application patrimoniale auto gérée. Nous reposons sur une architecture à composant car nous avons pu voir lors de l'étude des approches existantes qu'une architecture à composant est pertinente dans le cadre de la gestion autonome mais nous avons éliminé l'ADL et les API. Nous proposons un contrat que nous voyons section 6.1 pour faciliter le passage d'une application patrimoniale à cette même application gérée par notre approche. Enfin,

pour personnaliser certains comportements, nous sommes obligés de revenir à un langage de programmation traditionnel. Cependant, nous fournissons avec TUNe une librairie couvrant la majorité des besoins et si l'utilisateur doit quand même écrire du code, ce code est restreint à quelques méthodes de quelques lignes. Une connaissance approfondie du langage de programmation n'est donc pas requise et la taille du code produit permet d'écrire rapidement un code juste, vérifiable et réutilisable.

Chapitre 6

Description de TUNe

Nous présentons dans ce chapitre les spécifications de notre approche TUNe, son utilisation générale et son fonctionnement. Une fois ces bases posées, nous décrivons en détail les différents diagrammes et le langage d'encapsulation permettant de répondre aux spécifications. Nous voyons ensuite le cas particulier des sondes qui peuvent être déployées pour reconfigurer l'application patrimoniale et les différentes manière de lancer TUNe. Nous concluons enfin par une comparaison avec notre approche de référence Jade.

6.1 Spécifications

TUNe a été développé pour répondre aux besoins suivants :
- déployer une application patrimoniale sur différents noeuds,
- reconfigurer cette application,
- terminer l'exécution de cette application.

Pour que les utilisateurs sans connaissances systèmes, composants, middlewares ou autres puisse l'utiliser, nous avons posé les contraintes suivantes :
- masquer le niveau à composant,
- utilisation de langage connus et si possible graphiques,
- découpage d'utilisation en plusieurs niveaux pour que chaque utilisateur puisse paramétrer une partie de l'application patrimoniale,
- contrôle personnalisé des composants,
- fourniture d'une librairie standard compatible avec la majorité des applications pour la majorité des utilisateurs.

En respectant ces contraintes et ces spécifications, nous cherchons à avoir un système de gestion autonome répondant à tous les besoins exprimés dans l'état de l'art sans tomber dans les défauts de ces même approches.

6.2 Organisation générale

Nous distinguons deux niveaux d'exécution de TUNe : le niveau patrimonial et le niveau *System Representation* (SR) pour l'architecture générale et quatre niveaux d'utilisation de TUNe : le niveau coeur, le niveau développeur, le niveau utilisateur lui même découpé en deux parties : l'encapsulation de composants et l'écriture de politiques de reconfiguration. Nous présentons dans cette section l'utilisation générale de TUNe, la position de chaque niveau d'utilisation par rapport au fonctionnement de TUNe puis enfin le découpage en couche à l'exécution.

6.2.1 Utilisation générale de TUNe

Pour déployer l'application sur différents noeuds et la reconfigurer sans que l'utilisateur ne doive connaître le modèle à composant sous-jacent, nous utilisons le langage UML[1] [13]. Nous nous servons du diagramme de classe pour le déploiement et la représentation de la grille et du diagramme état-transition pour les politiques de reconfiguration. L'utilisation du diagramme de classe permet de définir une application en *intension* en faisant juste figurer le nombre initial de composants et le schéma utilisé pour les relier entre eux. L'utilisation du diagramme état-transition permet de définir un flot de données de manière intuitive avec des *ronds* et des *flèches*.

Dans le diagramme de classe, chaque classe représente une classe de composant. Les associations entre les classes représentent les liaisons entre les composants. Enfin, les cardinalités permettent de contraindre le schéma de déploiement. Les associations et les cardinalités permettent de définir le *pattern* de l'application. Le diagramme de classe de la figure 6.1[2] permet de déployer Diet avec 1 MA, 2 LA et 8 SeD. Ce diagramme de classe UML est le diagramme de déploiement sur lequel nous revenons plus en détail section 6.3.1. Ce diagramme de déploiement permet de décrire le découpage de l'application. C'est à son niveau que l'administrateur fait apparaître les *morceaux* d'application qui sont déployés.

[1] Unified Modeling Language

[2] les figures présentées dans ce chapitre permettent juste d'illustrer l'utilisation générale de TUNe. Elles sont expliquées en détails dans les chapitres suivants

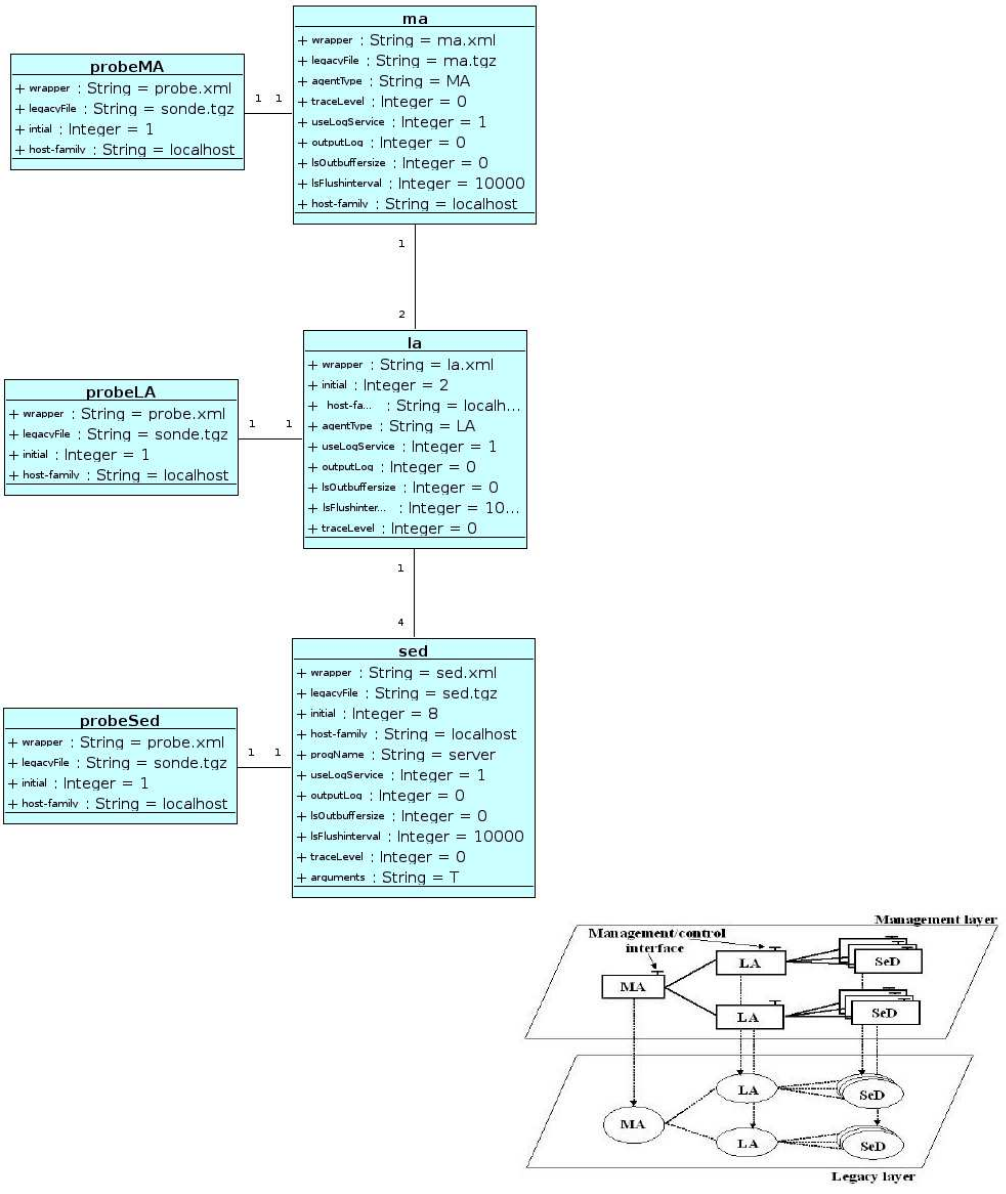

FIG. 6.1 – Le diagramme de classe pour déployer Diet

Pour exécuter le déploiement, un autre diagramme de classe UML doit être donné : le diagramme de *noeuds*. Ce diagramme permet de décrire la grille sur laquelle l'application patrimoniale va être déployée et la manière dont les composants de l'application vont être distribués sur chaque noeud. Chaque classe représente une famille de noeuds qui partagent des attributs et une politique d'allocation de noeud. La figure 6.2 représente une grille composée d'un seul noeud.

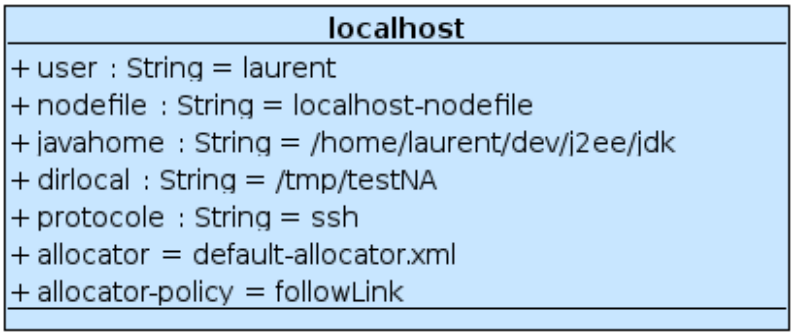

FIG. 6.2 – Une grille très simple !

Dans les diagrammes état-transition, chaque état représente une action à effectuer. Les liaisons entre les états représentent un enchaînement d'actions. Les diagrammes présentés figure 6.3 permettent le lancement de Diet et la réparation d'un LA. Ces

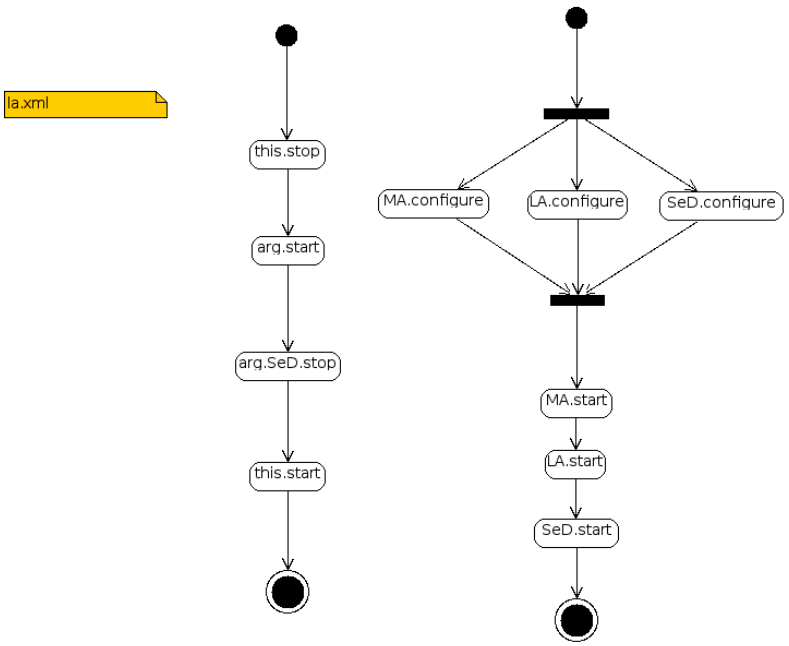

FIG. 6.3 – Deux diagrammes de reconfiguration de Diet

diagrammes état-transition sont nommés diagrammes de reconfiguration. Nous revenons plus en détail sur les diagrammes de reconfiguration section 6.3.3.

Comme nous pouvons le constater sur un diagramme de reconfiguration, des actions sont appelées, telles que *SeD.configure* ou *SeD.start*. TUNe ne pouvant savoir comment configurer ou démarrer un SeD, ces actions sont décrites dans un fichier WDL *Wrapper Description Language*. Ce fichier WDL contient un langage dérivé de XML sur lequel nous revenons section 6.4.1 et qui permet de décrire les types d'actions pouvant être effectuées sur chaque composant décrit dans le diagramme de déploiement. La figure 6.4 montre les actions pouvant être effectuées sur un SeD.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='sed'>

    <method name="start" key="extension.GenericStart" method="start_with_pid_linux" >
        <param value="$dirLocal/$progName $dirLocal/$srname-cfg $arguments"/>
        <param value="LD_LIBRARY_PATH=$dirLocal"/>
        <param value="OMNIORB_CONFIG=$dirLocal/$omni.srname-cfg" />
    </method>

    <method name="configure" key="extension.GenericConfigurePlainText" method="configure">
        <param value="$dirLocal/$srname-cfg"/>
        <param value=" = "/>
        <param value="traceLevel:$traceLevel" />
        <param value="parentName:$la.srname"/>
        <param value="name:$srname"/>
        <param value="useLogService:$useLogService"/>
        <param value="lsOutbuffersize:$lsOutbuffersize"/>
        <param value="lsFlushinterval:$lsFlushinterval"/>
    </method>

    <method name="stop" key="extension.GenericStop" method="stop_with_pid_linux" >
        <param value="$PID"/>
    </method>

</wrapper>
```

FIG. 6.4 – Le fichier WDL d'un SeD

De plus, la politique d'allocation de noeud, qui correspond au choix d'un noeud parmis un ensemble, est propre à chaque composant noeud. Les différentes politiques sont elles aussi décrites dans un fichier WDL sur lequel nous revenons section 6.4.3. Au lancement de TUNe, les diagrammes de déploiement et de noeud sont interprêtés puis l'application est déployée. Ce déploiement correspond à la copie des fichiers d'archive de l'application patrimoniale sur les noeuds désignés puis à leur décompression. Une fois l'application déployée, les diagramme de reconfiguration sont exécutés à la demande lors d'évènements, les notification, envoyés à TUNe.

Avec ces différents diagrammes et ces fichiers WDL, TUNe peut administrer n'importe quelle application patrimoniale comme J2EE ou Diet ou autres en respectant le contrat défini section 6.1. Ces applications peuvent être déployées, démarrées et reconfigurées juste avec ces diagrammes. La prise en main est rapide, il faut deux heures à un débutant pour commencer à administrer Diet avec TUNe.

6.2.2 Les niveaux d'utilisation

Quatre niveaux d'utilisation sont définis pour permettre aux utilisateurs de TUNe de trouver le niveau qui correspond le mieux à leur spécialité. Les applications patrimoniales peuvent donc être *TUNifiées* par plusieurs utilisateurs, chacun dans leur domaine de compétence. Les quatre niveaux sont :

- le niveau *coeur*. Ce niveau correspond au coeur de TUNe, la couche qui reçoit les notifications, interprête les diagrammes de déploiement et de noeud, applique les différentes politiques de reconfiguration et maintient à jour l'architecture à composant. Ce niveau est développé au sein du laboratoire et nécessite des compétences qui ne sont pas requises pour utiliser TUNe. C'est l'application TUNe !
- le niveau *développeur*. Comme nous pouvons le remarquer sur les diagrammes de reconfiguration figure 6.3 et dans le fichier WDL figure 6.4, des actions sont appelées sur les composants. Ces actions doivent être codées dans un langage de programmation pour pouvoir au plus bas niveau contrôler les composants. Nous avons choisi de coder les actions en Java. En vue de respecter la contrainte de la fourniture d'une librairie couvrant la majorité des besoins, l'équipe de développement de TUNe fournit des actions génériques déjà codées pour démarrer, arrêter, configurer la majorité des composants. Cependant ces actions peuvent ne pas suffire et les utilisateurs au niveau développeur pourront coder les leurs. Il en est de même pour les politiques d'allocation de noeuds,
- le niveau *utilisateur*. Ce niveau, qui corresponds à la *TUNification* proprement dite est découpé en deux sous niveaux :
  - les utilisateurs qui définissent les politiques de déploiement et de reconfiguration. Ces utilisateurs définiront les diagramme de noeud, déploiement et reconfiguration,
  - les utilisateurs qui vont encapsuler les composants. Ces utilisateurs, qui pourront aussi travailler au niveau développeur ou avec des collaborateurs de ce niveau, vont fournir les fichier WDL permettant à TUNe de contrôler les composants patrimoniaux.

6.2.3 Les niveaux d'exécution

Le niveau patrimonial

Le niveau patrimonial est le niveau où s'exécute l'application patrimoniale. Il est réparti sur plusieurs noeuds. Lors du déploiement de l'application patrimoniale, TUNe va en première approximation déposer sur chaque noeud un morceau de l'application, le configurer puis l'exécuter, suivant un diagramme de reconfiguration, le *startchart*. Une fois ces différentes étapes effectuées, le niveau patrimonial correspond pour l'application à l'environnement qu'elle aurait eu si elle n'avait pas été déployée par TUNe. Les échanges de messages au niveau patrimonial sont transparents pour

TUNe ce qui signifie que le comportement général de l'application n'est absolument pas impacté par TUNe. L'application s'exécute donc comme si elle était déployée manuellement, TUNe ne peut en première approche qu'agir sur chaque partie de l'application.
Ce niveau contient aussi des parties du coeur de TUNe qui vont pouvoir contrôler les différentes parties du logiciel patrimonial. Ce code de TUNe, nommé *effecteurs* va exécuter les actions codées en Java et définies dans le WDL. Les actions codées en Java au niveau développeur s'exécuteront donc à ce niveau. Ce niveau est donc majoritairement paramétré par le diagramme de noeud (pour connaître les noeuds utilisés) et par les fichiers WDL (pour connaître le code des actions à effectuer sur les différentes parties de l'application patrimoniale).

Le niveau System Representation (SR)

Ce niveau est le niveau qui contient la représentation à composant de l'application. Lors de l'exécution de TUNe et de l'interprétation des diagrammes de déploiement et de noeud, deux architectures Fractal sont créées : une qui représente l'application telle que décrite dans le diagramme de déploiement et l'autre qui représente la grille telle que décrite par le diagramme de noeud. Ces deux architectures peuvent être modifiée durant l'exécution de diagrammes de reconfiguration. Chaque modification de l'architecture à composant donne lieu à des modifications au niveau patrimonial de manière injective. Ainsi, le niveau SR peut être modifié par les diagrammes de reconfiguration et ces modifications sont ensuite reflétées au niveau patrimonial. L'inverse n'est pas possible, le niveau patrimonial ne pouvant pas s'auto-modifier. Ceci est une hypothèse de travail qui peut bien sûr être reconsidérée. Ces modifications à sens unique permettent de s'assurer que le niveau SR représente vraiment l'architecture de l'application patrimoniale. Ceci est rendu possible par l'architecture de chaque composant Fractal qui communique avec un effecteur de la couche patrimoniale. Ainsi chaque composant Fractal de la couche SR possède son effecteur dans la couche patrimoniale. Le niveau patrimonial peut par contre envoyer des *notifications* au niveau SR pour déclencher l'exécution de diagramme de reconfiguration. Ce niveau est donc majoritairement paramétré par le diagramme de déploiement (pour connaître le découpage de l'application et pour s'assurer de la cohérence suite à des reconfiguration).
L'interprétation du diagramme de déploiement figure 6.1 et du diagramme de noeud figure 6.2 génère le SR de la figure 6.5 où on peut constater que chaque classe du diagramme de déploiement génère un ou plusieurs composant Fractal. De même, ces interprétations génèrent la couche patrimoniale toujours de la même figure où on peut constater que chaque composant Fractal pilote un composant patrimonial par le biais de son effecteur et que toute la couche patrimoniale est déployée sur le même noeud.

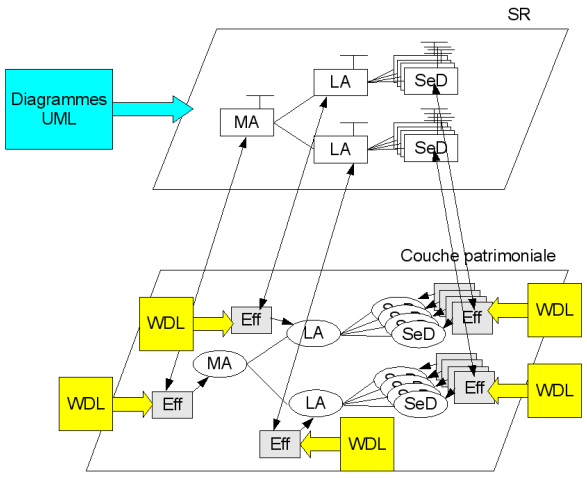

FIG. 6.5 – Le résultat de l'interprétation d'un diagramme de noeud et de déploiement

6.3 Les diagrammes

Comme précédemment vu, nous définissons deux types de diagrammes au niveau utilisateur : les diagrammes issus des diagrammes de classe UML qui permettent de décrire l'architecture de l'application et de la grille et les diagrammes issus des diagrammes de reconfiguration UML qui permettent de décrire les phases de reconfiguration. Nous étudions dans la suite de cette section chaque type de diagramme en détail et leur impact sur la couche SR (et donc sur l'architecture à composant) et patrimoniale.

6.3.1 Le diagramme de déploiement

Ce diagramme est issu du diagramme de classe UML. Nous n'avons pas utilisé de diagramme de composant car nous ne pouvons pas faire figurer de cardinalité sur ce dernier diagramme. Nous nous servons de 3 concepts UML dans ce diagramme : les classes, les associations et les cardinalités.

Les classes

Chaque classe représente une partie de l'application patrimoniale à administrer. Au niveau SR, chaque instance de classe sera représentée par un composant Fractal et au niveau patrimonial, chaque instance de classe représentera un tier de l'application. Dans chaque classe des attributs sont définis. Une partie des attributs est prédéfinie

et l'utilisateur doit en donner les valeurs, une autre partie est aussi prédéfinie mais TUNe affecte les valeurs et enfin une partie est à la discrétion de l'utilisateur. Les attributs prédéfinis à remplir par l'utilisateur sont les suivants :

- *wrapper* [OBLIGATOIRE]. Cet attribut doit avoir comme valeur par défaut le nom du fichier WDL qui définit les actions applicables à ce type de composant,
- *legacyFile* [OPTIONNEL]. Cet attribut doit avoir comme valeur par défaut le nom du fichier *tgz*[3] qui contient l'application patrimoniale à déployer pour ce type de composant,
- *host-family* [OBLIGATOIRE]. Cet attribut permet de déclarer sur quelle famille de noeud les composants de ce type doivent être déployés. Ces noms correspondent aux noms des classes du diagramme de noeud,
- *initial* [OPTIONNEL]. Cet attribut permet de spécifier combien de composant de ce type doivent être initialement déployés. Si il n'est pas défini, TUNe en déploiera un par défaut.

Les attributs prédéfinis et remplis par TUNe sont les suivants :

- *dirLocal*. Cet attribut contient le répertoire de déploiement du logiciel patrimonial sur son noeud de déploiement. Nous voyons dans la partie implantation que cet attribut est une recopie de l'attribut *dirLocal* du noeud sur lequel le logiciel patrimonial est déployé,
- *nodeName*. Cet attribut contient le nom du noeud sur lequel le logiciel patrimonial a été déployé. Le noeud est alloué par un allocateur de noeud (voir section 6.3.2)
- *tubeAddr*. Cet attribut contient le nom d'un tube nommé qui permet d'envoyer des notifications à TUNe (voir section 6.5.1),
- *srname*. Cet attribut contient le nom unique du composant au sein du SR.

D'autres attributs peuvent être définis par l'utilisateur pour pouvoir paramétrer plus finement les composants. Ces attributs peuvent être lus et écrits depuis les diagrammes de reconfiguration. Lors de la phase de création de l'architecture Fractal, ces attributs deviennent des attributs Fractal lisibles par le contrôleur d'attribut *AttributeController*.

Les associations

Les associations UML entre les classes sont implantées au niveau du SR par des liaisons Fractal qui peuvent être parcourues grâce aux contrôleurs de liaison *BindingController*. Pour chaque association, deux liaisons sont crées entre les composants Fractal comme décrit figure 6.6.

Le diagramme UML en haut de la figure donne l'architecture Fractal présentée en bas de la figure. On voit que deux liaisons relient C1 et C2. Cette double liaison créée en Fractal permet la navigation de C1 vers C2 et de C2 vers C1. Ces associations

[3] TAR Gunzip, un format d'archivage compressé

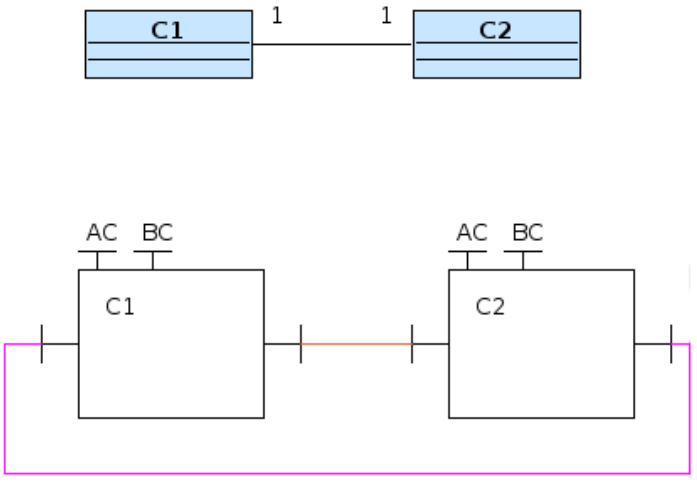

FIG. 6.6 – Une association UML transformée en double liaison Fractal

permettent de naviguer dans l'architecture Fractal et permettent d'aller récupérer des valeurs d'attributs d'un composant depuis un autre composant. Ainsi si les classes Apache et Tomcat sont reliées, un composant Apache sera relié à un composant Tomcat (en fonction des cardinalités) et chacun des deux composants pourra par exemple connaître l'adresse du noeud de l'autre grâce à l'attribut *nodeName*. Ce parcours est rendu possible par la syntaxe du WDL que nous décrivons section 6.4.1. Les associations peuvent être nommées pour faciliter la navigation dans le WDL (voir section 6.4.1).

Les cardinalités

Les cardinalités permettent à TUNe de maintenir cohérente l'architecture lors du déploiement et lors des reconfigurations avec le diagramme de déploiement. Utilisées avec l'attribut *initial*, elles permettent de valider au déploiement la cohérence de l'architecture. Utilisées avec le nombre courant de composant, elles permettent de valider les architectures issues des reconfigurations. Pour déterminer l'utilité de chaque cardinalité, nous nous appuyons sur la figure 6.7.

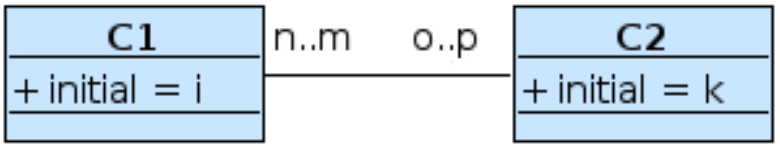

FIG. 6.7 – Deux composants et leurs cardinalités associées

Ainsi, suivant ce diagramme de déploiement :

- i composants C1 et k composants C2 seront déployés initialement,
- il faudra au minimum n composant(s) C2 reliés à C1,
- il faudra au maximum m composant(s) C2 reliés à C1,
- il faudra au minimum o composant(s) C1 reliés à C2,
- il faudra au maximum p composant(s) C1 reliés à C2.

Dans le cas où une seule cardinalité figure sur le diagramme pour un composant (comme à la figure 6.1), le nombre minimum et maximum de composant reliés sont les mêmes et sont égaux à la cardinalité spécifiée. La figure 6.8 sera donc un schéma de déploiement invalide. En effet, on déploie initialement 2 composants C2 et 1

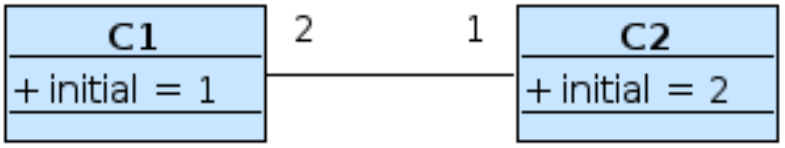

FIG. 6.8 – Schéma de déploiement incohérent

composant C1. Mais chaque composant C2 doit être relié à au moins deux composant C1 et au plus deux composants C1. Le déploiement ne peut pas se faire. La figure 6.9 permet elle de faire un déploiement valide.

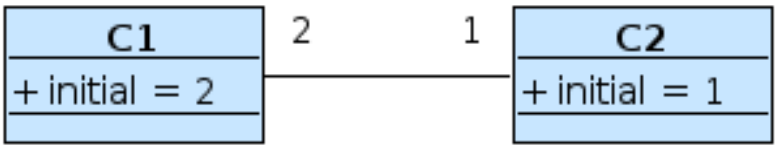

FIG. 6.9 – Schéma de déploiement cohérent

6.3.2 Le diagramme de noeud

Ce diagramme est lui aussi issu du diagramme de classe UML. Nous nous servons de la notion de classes et d'associations, par contre, nous ne nous servons pas de la notion de cardinalité. Les classes permettent de regrouper un ensemble de noeuds qui partagent les mêmes propriétés, représentées sous forme d'attributs, tandis que les associations permettent de naviguer d'un noeud à l'autre. Les cardinalités ne sont pas utilisées car elle permettent de contraindre le schéma lors de reconfiguration. Hors les noeuds ne peuvent pas être reconfigurés, en effet, TUNe ne peut pas ajouter un noeud à la grille !

Les classes

Chaque classe représente un ensemble de noeuds. Au niveau SR, chaque instance de classe sera représentée par un composant Fractal. Au niveau patrimonial, chaque instance de classe représente un ensemble de noeud partageant les mêmes propriétés et la même politique d'allocation. Dans chaque classe, des attributs sont prédéfinis, affectables par l'utilisateur ou par TUNe. L'utilisateur peut aussi rajouter ses propres attributs. Ces attributs représentent les propriétés de l'ensemble des noeuds de cette classe. Les attributs prédéfinis à remplir par l'utilisateur sont les suivants :

- *user* [OPTIONNEL]. Cet attribut contiendra le nom de l'utilisateur à utiliser pour se connecter au noeud distant. Si il n'est pas défini, l'utilisateur courant est utilisé,
- *nodefile* [OBLIGATOIRE]. Cet attribut contient le nom d'un fichier qui contient l'ensemble des noms des noeuds qui peuvent être utilisés par cette classe de noeuds,
- *dirLocal* [OBLIGATOIRE]. Cet attribut contient le nom du répertoire où le déploiement s'effectuera sur le noeud distant. Il est recopié dans l'attribut (affecté par TUNe) *dirLocal* dans les composants Fractal de l'application déployée,
- *javahome* [OBLIGATOIRE]. Cet attribut contient l'emplacement sur le noeud distant de la machine virtuelle Java pour pouvoir exécuter les effecteurs de TUNe,
- *protocole* [OPTIONNEL]. Cet attribut peut prendre comme valeur *ssh* ou *oarsh*[4] qui sera le protocole de copie du logiciel patrimonial et d'exécution à distance des effecteurs sur le noeud distant. Si il n'est pas présent, le protocole *ssh* est choisi par défaut. Le type de protocole est pour le moment uniquement extensible par les développeurs de niveau *coeur*,
- *allocator* [OBLIGATOIRE]. Cet attribut donne le nom d'un fichier WDL dans lequel un ensemble de politiques d'allocation sont décrites,
- *allocator-policy* [OBLIGATOIRE]. Cet attribut donne le nom de la politique d'allocation courante parmi celle décrites dans le fichier WDL défini par l'attribut *allocator*.

Le nom de chaque classe est le nom qui est repris dans l'attribut *host-family* du diagramme de déploiement. Nous verrons section 6.4.3 comment définir des politiques d'allocation. Lors de la phase de création de l'architecture Fractal, ces attributs deviennent des attributs Fractal lisibles par le contrôleur d'attribut *AttributeController*. Les attributs des noeuds sont utilisables depuis le WDL par les actions du logiciel patrimonial. Nous y revenons lors de la description du WDL, section 6.4.1.

[4] protocole de connexion utilisé sur Grid 5000, proche de ssh

Les associations

Les associations UML dans le diagramme de noeud n'ont pas de représentation au niveau patrimonial. Elle sont juste présentes pour facilité la navigation par le WDL entre les différents noeuds pour récupérer des attributs.

6.3.3 Les diagrammes de reconfiguration

Les diagrammes de reconfiguration sont issus du diagramme état-transition d'UML. Ils permettent de spécifier une suite d'actions à effectuer, l'ordre de ces actions et le niveau de parallélisation souhaité entre elles. Des états particuliers sont utilisés pour paralléliser, attendre, débuter et finir le diagramme de reconfiguration. De plus, des mots clés sont définis pour connaître le composant cible sur lesquels des méthodes de reconfiguration vont s'appliquer et une navigation au sein du SR est possible. La navigation se fait par une *notation pointée*[5], en partant d'un composant *source* qui est à l'origine de la notification (voir section 6.5.1) et en pointant un composant *cible*. Les commentaires UML sont utilisés pour associer chaque diagramme de reconfiguration à une notification particulière émise par la couche patrimoniale. Enfin, des diagrammes spécialisés doivent obligatoirement être fournis par l'utilisateur pour le démarrage et l'arrêt de l'application patrimoniale.

L'état initial

L'état initial UML est le premier état dans le diagramme de reconfiguration. Aucune action n'est associée à cet état, c'est le point d'entrée du diagramme.

L'état final

L'état final UML est le dernier état dans le diagramme de reconfiguration. Lors de l'exécution d'un diagramme de reconfiguration, les actions entre l'état initial et l'état final sont exécutées sous forme de transaction. Lors de l'exécution de l'état final, la cohérence de l'architecture par rapport au diagramme de déploiement est vérifiée. Si l'architecture est cohérente, le SR est modifié puis les actions sont exécutées par les effecteurs au niveau patrimonial. Dans le cas où l'architecture issue de l'exécution du diagramme n'est pas cohérente, une exception est levée, les actions ne sont pas effectuées par les effecteurs et le SR est laissé dans l'état dans lequel il était lors de l'état initial.

[5] on utilisera indifféremment le terme chaîne pointée ou notation pointée

Les états *fork* et *join*

Ces états définis eux aussi par UML permettent de séparer (*fork*) ou de réunir (*join*) des flots d'exécution lors de l'exécution du diagramme de reconfiguration. Des actions peuvent ainsi être menées en parallèle en vue de gagner du temps.

Les états d'action

Tous les états qui ne sont pas initiaux, finaux, fork ou join sont considérés comme des états permettant d'effectuer des actions sur le SR et sur le logiciel patrimonial. Chaque action s'exécute sur un composant *cible*. Au sein d'un même flot d'exécution, les états d'action sont exécutés séquentiellement.

Les mots clés Dans le langage utilisé pour exprimer des actions, des mots clés sont définis pour repérer le composant *cible* sur lequel les actions vont être menées :

- *this*. Le composant *cible* est le composant *source*. Ce mot clé n'est pas obligatoire, le composant *cible* par défaut est *this*,
- *arg*. Le composant *cible* est le composant passé en paramètre de la notification,
- *un nom de classe du diagramme de déploiement*. Les actions s'appliqueront à tous les composants de la classe dont le nom est donné,
- *une variable affectée dans le diagramme de reconfiguration*. Le composant *cible* est alors celui référencé par la variable (voir section 6.3.3).

La navigation Les actions ne doivent pas nécessairement être exécutés sur le composant *source* de la notification (*this*, *arg*, ...). Ainsi une notation pointée permet d'écrire des *chaînes pointées* qui permettent de naviguer dans le SR pour désigner un composant *cible* sur lequel des actions de reconfiguration seront appliquées. Le dernier nom de la chaîne pointée correspond à l'action définie dans le WDL du composant *cible*. La navigation n'est possible qu'entre types de composants reliés par une association dans le diagramme de déploiement. Pour plus de détails concernant la navigation, se reporter à la section 6.4.1.

L'affectation Une syntaxe particulière (une chaîne contenant un égal) permet d'affecter une valeur a un attribut dans le composant *cible*. Un exemple est donné figure 6.10 ou figure 6.11. Si la variable affectée n'est pas définie dans la classe au niveau du diagramme de déploiement, elle est automatiquement créée.

Ajout et retrait de composants Une syntaxe particulière (chaîne finissant par "++" ou "- -") permet d'exécuter une action de dimensionnement qui correspond à l'ajout ou au retrait d'un composant du SR et dans la couche patrimoniale. Une chaîne finissant par "++" va permettre d'ajouter un composant au SR et déployer le logiciel patrimonial associé sur un noeud. A l'inverse, une chaîne finissant par "- -" va retirer le logiciel patrimonial du noeud où il se trouve et retirer le composant qui le pilote du SR. Le composant ajouté ou retiré est le composant *cible*. Une action d'ajout renvoie le nouveau composant créé (pour par exemple le configurer puis le démarrer) tandis qu'une action de retrait renvoie le type du composant retiré (pour par exemple en créer un autre du même type mais sur un autre noeud). L'utilisation des "++" ou "- -" pour ajouter ou retirer des composants permet de préserver la cohérence de l'application si les arités sont respectées.

Exemples Déroulons le diagramme de droite sur la figure 6.3 : les actions suivantes sont effectuées :

- sur l'état *fork*, création de trois flots d'exécution,
- exécution parallèle des méthodes *configure* sur tous les MA, tous les LA et tous les SeD,
- sur l'état *join*, attente de la fin d'exécution des méthodes *configure*
- puis exécution successive des méthodes *start* des MA, LA et SeD.

Le diagramme de gauche de cette même figure permet l'exécution des actions suivantes lors de la réception de la notification nommée *la.xml* :

1. appel de la méthode *stop* sur le composant qui a émis la notification,
2. appel de la méthode *start* sur le composant passé en paramètre de la notification,
3. appel de la méthode *stop* sur tous les SeD reliés au composant passé en argument,
4. appel de la méthode *start* sur le composant qui a émis la notification.

Concrètement au niveau patrimonial, les actions suivantes seront effectuées :

1. arrêt de la sonde qui a détecté la panne du LA,
2. redémarrage du LA qui était en panne,
3. arrêt des SeD qui sont reliés au LA qui vient de redémarrer. Cette action permet un redémarrage des SeD pour qu'ils puissent se réenregistrer auprès du LA qui vient de redémarrer,
4. redémarrage de la sonde associée au LA réparé.

Le diagramme de la figure 6.10 permet d'affecter la variable PID du composant *source* (omission du mot clé *this*) par la valeur de l'argument passé à la notification lors de la réception d'une notification nommée *setPID*. On utilise ce diagramme lors du démarrage de logiciel patrimonial pour pouvoir conserver l'identificateur du processus pour arrêter le logiciel en cas de besoin.

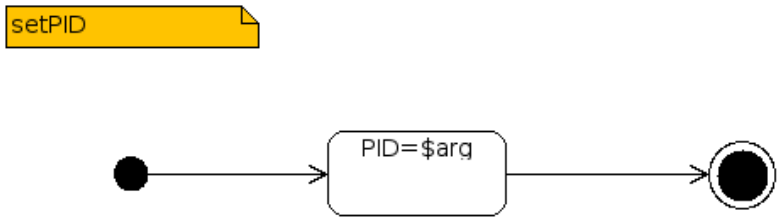

FIG. 6.10 – Affectation d'une variable dans un composant

Enfin, la figure 6.11 permet de migrer un composant et sa sonde. On entends par migration arrêter un composant sur un noeud puis le redémarrer sur un autre noeud. Cela peut être utile en cas de surcharge du noeud initial. Ce diagramme fait appel à toutes les actions possibles sur un diagramme de reconfiguration. Nous devons donc modifier la politique d'allocation pour permettre au composant *source* d'être déployé sur un autre noeud. Le nouveau noeud est passé en paramètre de la notification. Ensuite, nous revenons à une politique d'allocation qui permet de déployer deux composants sur un même noeud. En effet, le composant qui sonde doit être sur le même noeud que le composant sondé. Nous illustrons de plus dans cette figure l'utilisation de variable et le changement de politique d'allocation d'une classe de noeud. Le composant *source* est un composant relié par la liaison *probed* au composant qui

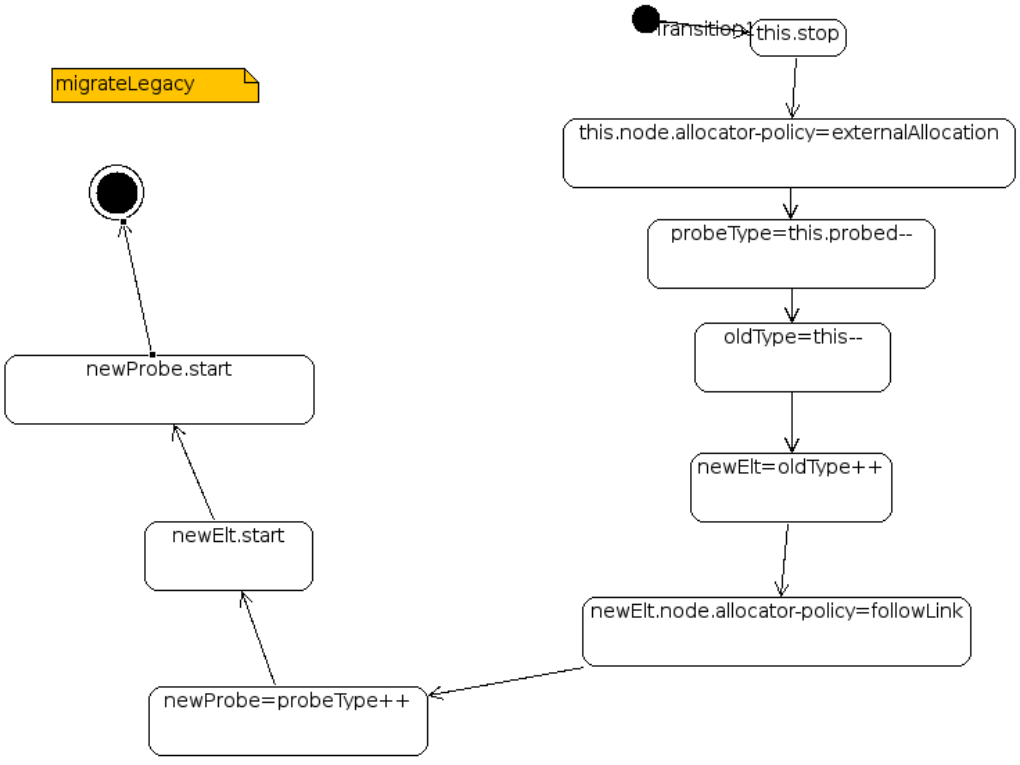

FIG. 6.11 – Migration d'un composant et de sa sonde

le sonde. Les actions suivantes sont réalisées :

- arrêt du composant à migrer,
- changement de la politique d'allocation de la famille de noeud auquel le composant est relié pour passer à une politique nommée *externalAllocation* qui permet de forcer l'allocation sur un noeud grâce à l'argument de la notification (voir section 6.5.1),
- retrait de la sonde et sauvegarde de son type dans la variable *probeType*,
- retrait du composant à migrer et sauvegarde de son type dans la variable *oldType*,
- création d'un composant du type de *oldType*, donc du type du composant à migrer et sauvegarde de la référence à ce composant dans la variable *newElt*,
- changement de la politique d'allocation de la famille de noeud auquel le composant est relié pour passer à une politique nommée *followLink* qui permet de déployer deux composant reliés par un certain type de liaison sur le même noeud,
- création d'un composant du type de *probeType*, donc du type de la sonde et sauvegarde de la référence à ce composant dans la variable *newProbe*,
- démarrage du composant référencé par *newElt*, donc démarrage du composant migré,
- démarrage du composant référencé par *newProbe*, donc démarrage de la sonde migrée.

Les diagrammes de reconfiguration particuliers

Pour finaliser le déploiement d'une application, TUNe exécute un diagramme obligatoire nommé *startchart*. Ce diagramme permet la configuration et le lancement des tiers patrimoniaux. De même, pour terminer l'exécution d'une application, TUNe exécute un diagramme obligatoire nommé *stopchart*. Ce diagramme est exécuté juste avant le retrait des logiciels patrimoniaux des noeuds distants. Ces diagrammes de reconfiguration sont les seuls à ne pas nécessiter la présence du commentaire UML indiquant le nom de la notification qui permettra leur appel. Un exemple de *startchart* est celui de droite sur la figure 6.3. Un exemple de *stopchart* est donné figure 6.12.

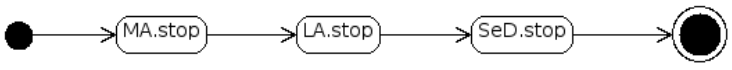

FIG. 6.12 – Diagramme de reconfiguration permettant l'arrêt de l'application : le stopchart

Ce diagramme permet d'arrêter tous les composants *MA*, *LA* et *SeD*.

6.4 L'encapsulation de logiciel patrimonial

L'encapsulation de logiciel patrimonial est réalisée au niveau développeur et utilisateur de TUNe. Elle est en deux parties : une partie code Java qui implante des méthodes appelées par les effecteurs au niveau patrimonial et une partie langage WDL qui permet de spécifier la manière dont les effecteurs doivent appeler ces méthodes. Le schéma général est que le WDL permet de récupérer des attributs dans la couche SR pour ensuite les passer comme argument aux fonctions Java appelées par les effecteurs.

6.4.1 La partie spécification en WDL

Le WDL[6] permet de décrire l'encapsulation des logiciels patrimoniaux. On entends par encapsulation :
- définir l'interface de contrôle du logiciel patrimonial,
- faire un lien entre cette interface de contrôle et des méthodes Java,
- permettre le passage de paramètre à ces méthodes Java. Les valeurs passées aux méthodes Java peuvent être récupérées depuis le SR grâce aux chaînes pointées.

C'est un langage qui possède une syntaxe XML et qui utilise seulement trois balises :
- *wrapper* avec l'attribut *name* qui définit le nom du wrapper,
- *method* avec les attributs :
  - *name* qui sera le nom donné dans les états d'action des diagrammes de reconfiguration,
  - *key* qui est le nom de la classe qui possède la méthode,
  - *method* qui est le nom de la méthode Java dans la classe *key*.
- *param* qui sont les paramètres à passer à la méthode.

L'intérêt du WDL est d'utiliser la notation pointée dans la balise *param* en vue de naviguer dans le SR comme dans les diagrammes de reconfiguration et passer des arguments aux méthodes Java. Ainsi, dans la balise *param*, une chaîne
- qui commence par $ signifie qu'un attribut doit être récupéré dans le SR. Tous les attributs prédéfinis ou définis par l'utilisateur peuvent être utilisés. Cette chaîne aura comme composant *source* le composant sur lequel la méthode doit s'appliquer,
- qui ne commence par par $ sera passée telle quelle à la méthode Java.

[6] Wrapping Description Language

Pour naviguer, 3 types de déréférencements peuvent être utilisés :
- déréférencement par nom. Ainsi, $C1.attribut ira chercher l'attribut *attribut* dans le composant C1 relié au composant courant. Si plusieurs composants C1 sont reliés, les attributs *attributs* seront récupérés et séparés par des ";",
- déréférencement par nom de liaison. Lorsque qu'une liaison entre deux composants est nommée, le nom de la liaison peut être utilisé en lieu et place du nom du composant dans le déréférencement par nom. Cela permet d'utiliser le même fichier WDL pour plusieurs classes de composants différentes. Par exemple, dans le cadre de sonde, un même fichier WDL pourra sonder plusieurs types de composant sans modifier le WDL (voir l'exemple de sonde section 6.5.2),
- pas de déréférencement. L'attribut est récupéré dans le composant courant.

Le mot clé *node* est ajouté à ce langage WDL pour pouvoir déréférencer le noeud sur lequel le composant est déployé (pour un exemple, voir le code présenté 6.17). Ainsi le fichier WDL présenté figure 6.4 définit 3 méthodes appelables depuis les diagrammes de reconfiguration : *start*, *stop* et *configuration*. Intéressons nous à la méthode *configure*. Elle est définie dans la classe *extension.GenericConfigurePlainText* et se nomme *configure* dans cette classe. Les paramètres suivants lui seront passés :
- *$dirLocal/$srname-cfg* : un chemin composé du répertoire d'installation du logiciel patrimonial puis d'un fichier qui porte le nom du composant dans le SR suivi de la chaîne "*-cfg*". sera passé en premier paramètre. Par exemple, ce pourrait être : */tmp/deploiement/sed_0-cfg*,
- = : le caractère égal sera passé en deuxième paramètre,
- *tracelevel :$tracelevel* : la chaîne composée de la chaîne "*tracelevel :*" puis de la valeur de l'attribut défini par l'utilisateur *tracelevel* sera passée en troisième argument. Par exemple, ce pourrait être *tracelevel :0* en suivant le diagramme de déploiement figure 6.1,
- *parentName :$la.srname* : la chaîne composée de la chaîne "*parentName :*" puis la valeur de l'attribut srname du ou des LA qui sont reliés à ce SeD sera passée en quatrième argument. Par exemple, ce pourrait être *parentName :la_0*. On a utilisé ici la navigation dans l'architecture,
- ...

6.4.2 La partie méthodes en Java

Les méthodes Java appelées par le fichier WDL sont des méthodes Java standards répondant tout de même à certaines règles dans leurs arguments. En effet, TUNe ne sait pas à priori combien d'arguments vont être passés. Ainsi, les méthodes Java utilisables pour la configuration et définies dans le WDL doivent avoir un des trois profils suivants :
- `public void methode(String argumentUnique)`

- `public void methode(String premierArgument, String[] argumentsSuivants)`
- `public void methode(String[] tousLesArguments)`

Le reste de la méthode est du code Java standard qui peut faire appel à toutes les classes standards J2SE. La figure 6.13 présente le code utilisé pour arrêter un composant :

```
public void stop_with_pid_linux(String cmd)
{
  System.out.println("GenericStop: stopping "+cmd+" ( BashOnly )");
  try
  {
    Runtime.getRuntime().exec("kill -9 "+cmd);
  }
  catch(Exception e)
  {
    e.printStackTrace();
  }
}
```

FIG. 6.13 – Méthode générique d'arrêt forcé d'un composant

Une partie du fichier WDL associé pourrait être la suivante celle présentée figure 6.14.

```
<method name="stop" key="extension.GenericStop" method="stop_with_pid_linux" >
  <param value="$PID"/>
</method>
```

FIG. 6.14 – Balise WDL générique d'arrêt d'un composant

Ainsi, lors de l'appel à *this.stop* dans un diagramme de reconfiguration, le PID du composant va être récupéré au niveau du SR, passé en paramètre de la méthode *stop_with_pid_linux* pour que cette dernière exécute la commande système *kill* pour tuer le processus.
Plusieurs méthodes Java sont fournies ainsi que des WDL d'exemple pour répondre à la majorité des besoins. Ces méthodes sont les suivantes :

- modification ou création d'un fichier de configuration sous forme de paire clé-valeur, utilisée notamment pour la configuration de Diet ou d'Apache,
- modification ou création d'un fichier de configuration en XML, utilisé notamment pour la configuration de Tomcat,
- démarrage d'un processus Linux en attendant le PID ou un fichier annonçant le lancement du processus, utilisée dans tous les cas étudiés dans cette thèse,
- arrêt d'un processus Linux.

Si cette libraire ne suffit pas, les utilisateurs de niveau développeur peuvent en rajouter.

6.4.3 Cas particulier du WDL pour les politiques d'allocation de noeud

Pour chaque classe de noeud, l'utilisateur doit donner un fichier WDL qui comprend la description des politiques d'allocation pouvant être utilisées par le noeud. Ce sont comme pour les composants applicatifs des méthodes Java, mais appelées implicitement par TUNe lors du choix d'un noeud pour déployer un composant. Le fonctionnement est le même que pour le WDL des composants applicatifs et le mot clé *todeploy* a été rajouté pour pouvoir accéder aux attributs du logiciel patrimonial à déployer. Un exemple de WDL de noeud est donné figure 6.15.

```
<wrapper name='default-allocator'>

    <method name="getNodeRR" key="extension.DefaultAllocator" method="round_robin" >
        <param value="$todeploy.srname"/>
    </method>

    <method name="followLink" key="extension.DefaultAllocator" method="followLink">
        <param value="$todeploy.srname"/>
        <param value="$todeploy.probed.srname"/>
    </method>

    <method name="externalAllocation" key="extension.DefaultAllocator" method="external">
        <param value="$todeploy.arg"/>
        <param value="$todeploy.srname"/>
    </method>

</wrapper>
```

FIG. 6.15 – Fichier WDL pour décrire les politiques d'allocation d'un noeud

Dans ce fichier, 3 politiques d'allocation de noeud sont définies : *getNodeRR*, *followlink* et *external*. Ce sont les trois politiques fournies en standard avec TUNe. Elles permettent respectivement d'allouer des noeuds en *Round Robin*, d'allouer des noeuds de manière collocalisée en suivant les liaisons et enfin d'allouer un noeud précis passé en paramètre par le diagramme de reconfiguration qui fera appel à cette politique. La politique *getNodeRR* fera appel à la méthode *round_robin* de la classe *extension.DefaultAllocator* avec comme argument le srname du composant à déployer.
Les méthodes Java doivent avoir le même profil que les méthodes pour les composants applicatifs mais un *Vector* doit être le premier argument de la méthode. Ce vecteur contient la liste des noeuds physique pour cette classe de noeuds. La méthode *round_robin* est présentée figure 6.16 :

L'attribut de classe *__wv* est automatiquement attribué par TUNe et permet à l'utilisateur de stocker des variables sous forme *clé-valeur* pour s'en resservir lors d'un appel suivant à cette méthode.

```
public class DefaultAllocator
{

  public WrapperVariable __wv;

  public Node round_robin(Vector nodes, String component)
  {
    Integer node_number=(Integer)__wv.getVar("node-number");
    if(node_number==null)
      node_number=new Integer(0);

    Node node_to_return=(Node)nodes.get(node_number.intValue());

    __wv.setVar("node-number", new Integer((node_number.intValue()+1)%nodes.size())
        );
    __wv.setVar(component, node_to_return);

    return node_to_return;
  }
}
```

FIG. 6.16 – Code de la méthode permettant l'allocation des noeuds en round robin

6.5 Sondes et notifications

Pour pouvoir reconfigurer l'application patrimoniale, nous avons vu qu'il existe des diagrammes de reconfiguration. Ces derniers sont appelés lors de l'émission d'une notification par un composant applicatif qui a le rôle de sonde. Dans notre approche, un composant de sondage est un composant banal au même titre que les composants qu'il surveille. Au niveau développeur, des sondes peuvent être écrites. Nous proposons sinon en jeu de sonde permettant de surveiller un composant local ou un composant à distance.

6.5.1 Les notifications

Une notification est une chaîne de caractère écrite dans un tube nommé local créé et géré par TUNe (voir section 7.5). Ce tube nommé se trouve dans l'attribut *tubeAddr* que TUNe ajoute à chaque composant déployé. Une notification comporte 3 champs séparés par des ";" :

- le nom de la notification qui sert à déclencher les diagrammes de reconfiguration,
- la source de la notification qui est soit un nom de composant dans le SR soit le mot clé *this*. Dans ce dernier cas, TUNe analyse la provenance de la notification et remplace *this* par le composant *source* de la notification,
- un argument qui peut être un nom de composant pour être utilisé ou une chaîne pointée pour désigner un composant ou enfin une chaîne de caractère à affecter à un attribut. Cette argument est utilisable dans les diagrammes de reconfiguration grâce au mot clé *arg*.

Une même notification peut déclencher l'exécution de plusieurs diagrammes de reconfiguration dans un ordre non déterminé.

6.5.2 Les sondes

Les sondes sont des logiciels patrimoniaux qui surveillent d'autres logiciels patrimoniaux. Elles doivent donc figurer dans le diagramme de déploiement. Dans le diagramme présenté figure 6.1, les sondes sont les classes nommées *probeMA, probeLA et ProbeSeD*. Elles sont généralement reliées au(x) composant(s) sondé(s) pour pouvoir récupérer des attributs permettant le sondage ou l'envoi de notifications. Ainsi une sonde que nous fournissons ne fait que lire la table des processus et recherche le PID du processus sondé à l'intérieur. Le WDL de lancement de la sonde est présenté figure 6.17 :

```
<method name="start" key="extension.GenericStart" method="start_with_pid_linux" >
  <param value="$node.javahome/bin/java -cp $dirLocal DistributedProbe '$probed.PID
      ' '$tubeAddr' '$probed.srname' la.xml '$probed.nodeName'"/>
</method>
```

FIG. 6.17 – Lancement d'une sonde

En cas de panne, si la sonde surveille un LA par exemple, elle enverra donc la notification *la.xml ;this ;la_0* dans le tube nommé *tubeAddr*. Cela pourra déclencher un diagramme de réparation tel que celui à gauche dans la figure 6.3.

6.6 Exécution de TUNe et récupération des traces

Pour exécuter une application TUNifiée, l'administrateur dispose de plusieurs méthodes. Deux grandes famille de méthodes se distinguent : un TUNe qui gère une application ou un TUNe qui gère plusieurs applications. Nous détaillons les différentes méthodes de ces deux familles. Nous résumons ensuite les ordres pouvant être donnés à TUNe lorsqu'il est utilisé en mode interactif et nous terminons par la gestion des traces.

Un TUNe par application Depuis la console de contrôle, l'administrateur passe en ligne de commande à TUNe le nom du fichier contenant les diagrammes UML. Ce mode d'exécution est simple et est souvent utilisé à des fins de test. C'est un mode non interactif.

Un TUNe pour plusieurs applications Depuis la console de contrôle, l'administrateur passe en ligne de commande à TUNe la méthode pour lancer des applications. Trois méthodes sont disponibles :

- *interactif* : TUNe affiche alors une invite où l'utilisateur peut donner des ordres à TUNe permettant la gestion des applications. Ces ordres sont décrit ci-après,
- *telnet* : TUNe se comporte comme un serveur telnet. Ce mode permet de contrôler le TUNe et les applications exécutées à distance. L'invite présentée est la même que dans le mode interactif et les ordres sont aussi les mêmes,
- *pipe* : TUNe rends immédiatement la fin à la console de contrôle après avoir créé un tube nommé. Ce mode permet de contrôler TUNe depuis des *scripts*[7]. L'administrateur ou le script contrôle alors les applications en écrivant des ordres dans le tube nommé. Les ordres sont les mêmes que dans le mode interactif.

Ces modes de lancement sont des modes interactifs.

Les ordres de contrôle Dans les modes interactifs, les ordres pouvant être donné à TUNe sont les suivants :

- *deploy* suivi du nom du fichier contenant les diagrammes UML pour lancer l'application patrimoniale. Cette instruction renvoi un numéro d'identification de l'application,
- *undeploy* suivi du numéro d'identification de l'application pour arrêter l'application patrimoniale,
- *list* pour obtenir la liste de toutes les applications lancées,
- *quit* pour arrêter toutes les applications patrimoniales en cours et quitter TUNe.

Les traces et les erreurs Les traces des logiciels patrimoniaux sont remontées à la machine sur laquelle TUNe s'exécute. Les traces sont classées en 3 catégories :

- traces obligatoires : exceptions, déploiement d'une application et arrêt d'une application,
- traces optionnelles : création des architectures Fractal, création des liaisons entre les composants, réception des notifications, appel de méthode, ...,
- traces distantes : les traces des effecteurs et des logiciels patrimoniaux.

En vue de ne pas surcharger le réseau, le mode *silent* permet de n'afficher que les traces obligatoires, le mode *-v* permet d'afficher les traces obligatoires et les traces optionnelles et enfin le mode *-vv* permet d'afficher toutes les traces.

[7] fichiers de traitement par lot

6.7 Conclusion

Comparons la mise en oeuvre de Diet avec Jade et TUNe dans le tableau 6.1 :

Approche	**Déploiement**	**Flot déploiement**	**Politiques** [1]	**Effecteurs** [2]
Jade	code ADL	code Java + API Fractal	code Java + API Fractal	code Java + API Fractal
TUNe	graphique	graphique + navigation	graphique + navigation	code Java fourni + WDL

[1] Description des politiques de reconfiguration et application de ces dernières
[2] Code pour effectuer les actions sur le logiciel patrimonial

TAB. 6.1 – Comparaison synthétique de Jade et TUNe

A la vue du tableau précédent, nous pouvons constater que nos objectifs ont été atteints :

- problème de connaissance par l'administrateur des ADL des approches à composant : l'architecture à composant est masquée,
- problème de connaissance par l'administrateur des langages d'écriture des politiques : l'écriture des politiques est graphique, la réutilisation en est améliorée,
- problème de connaissance par l'administrateur des langages d'encapsulation : notre langage WDL est très simple et l'administrateur peut ne pas écrire une seule ligne de code en utilisant la librairie de méthodes que nous fournissons. Dans les cas particuliers, les développeurs s'en occupent à leur niveau.

Nous avons donc une approche de gestion autonome qui permet au plus grand nombre de rendre leurs application auto-gérables avec un minimum de connaissance.

Chapitre 7

Implantation

Nous présentons dans cette section les choix architecturaux et les choix d'implantation retenus pour notre approche TUNe. Nous allons suivre le déroulement d'une application *TUNifiée*, du lancement de TUNe à l'arrêt de TUNe en passant par le déploiement de l'application, son exécution sur les noeuds distants, sa reconfiguration et sa terminaison. Nous suivons l'application Diet présentée dans les sections précédentes avec le diagramme de déploiement présenté figure 6.1, le diagramme de démarrage présenté figure 6.3 et le diagramme de terminaison. Cette application se déploie sur le noeud local avec le diagramme de grille présenté 6.2.

7.1 Lancement de TUNe

Comme expliqué au chapitre précédent, une application peut être lancée de quatre manières différentes :

- un TUNe par application : le nom du fichier uml qui contient les diagrammes de description de l'application sont passés en ligne de commande à TUNe,
- un TUNe qui gère plusieurs applications : le nom des applications peut être passé grâce à un serveur telnet embarqué dans TUNe, de manière non interactive par un tube nommé ou de manière interactive en console. Ces derniers modes d'exécution sont rendus possible grâce à une surcouche implantant l'interprêteur, le serveur telnet ou lisant le tube nommé qui va ensuite exécuter les applications comme dans le premier cas (une application par TUNe) mais en lançant les classes principales de TUNe sur la même machine virtuelle.

Lors du lancement de l'application, le fichier UML est parsé grâce à un parseur XML SAX[1]. Ce parseur permet d'extraire les diagrammes contenus dans le fichier UML pour générer l'architecture Fractal dans le cas du diagramme de déploiement et de noeud ou générer des graphes dans le cas de diagrammes état-transition.

[1] Simple API for XML

7.1.1 Génération de l'architecture Fractal

La partie applicative Pour que l'utilisateur n'ait pas à connaître les API ou l'ADL Fractal, TUNe génère automatiquement l'architecture Fractal depuis les diagrammes UML. Il existe donc des règles de passage d'un modèle à l'autre :

- une classe donnera un ensemble de composants Fractal,
- une association donnera un ensemble de liaisons Fractal entre des interfaces clientes et serveurs,
- un attribut de classe donnera un attribut Fractal par composant.

Chaque composant Fractal est créé avec un contrôleur d'attribut *générique* qui permet de récupérer n'importe quel attribut grâce à un *get* ou un *set*, un contrôleur de liaison et un contrôleur de cycle de vie.

Au niveau des interfaces métiers, une interface métier cliente est créée pour chaque association UML. L'interface cliente porte le nom de la classe vers lequel elle pointe. De même, si c'est une liaison nommée, une autre interface cliente est créée et porte le nom de la liaison (voir figure 7.1). Dans cette figure, le composant C1 a donc deux interfaces, *C2* et *link* pour naviguer indifféremment avec le nom du composant pointé ou le nom de la liaison (voir section 7.4.5). De plus, une interface *Node* permet l'interconnexion de chaque composant applicatif avec le composant représentant la classe de noeud sur lequel cette partie d'application est déployée.

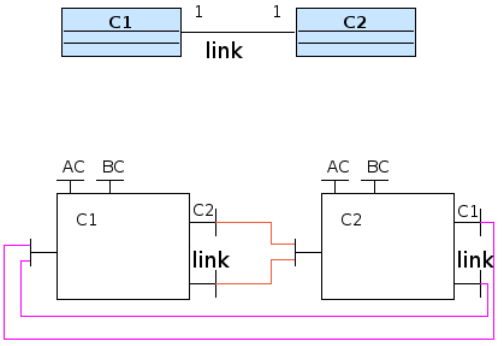

FIG. 7.1 – Les interfaces clients et serveurs de deux composants liés. L'interface node n'est pas représentée

Les interfaces serveurs implantent l'interface Java *WrapperItf* qui comprend entres autres la méthode *meth* (voir figure 7.2) qui permet l'exécution de méthode du WDL sur laquelle nous revenons section 7.4.4 :

```
public interface WrapperItf extends Remote
{
    public void meth(String className, String methName, Object[] args) throws
        RemoteException;
}
```

FIG. 7.2 – Interface cliente et serveur des composants Fractal

L'ensemble de ces composants et leur liaisons sont placés dans un composant composite nommé *applicatif*.

La partie noeud Les règles de passage pour ce diagramme sont plus simples car chaque classe UML donne un seul composant Fractal. A ce jour, les composants Fractal représentant les noeuds ne sont pas reliés entre eux. L'ensemble de ces composants sont placés dans un composant composite nommé *cluster*. Nous verrons dans les perspectives qu'il est possible d'avoir une description plus détaillée du matériel.

Le partage des composants Pour coller au mieux à la représentation réelle de l'application, nous avons séparé les composants applicatifs des composants noeuds en les mettant dans des composites différents. Il est cependant intéressant de les relier pour que les composants applicatifs puissent avoir accès aux propriétés des noeuds sur lesquels ils se trouvent et vice-versa. Nous nous servons donc du partage de composant Fractal qui nous permet de relier les composants applicatifs aux composants noeuds au sein du super-composant *application* qui regroupe les composites *applicatif* et *noeud* en fonction de l'attribut *hostFamily* de chaque composant *applicatif*.

Les classes internes aux composants (*content class*)

Pour les composants applicatifs La classe interne au composant applicatif est la classe *GenericWrapper*. En plus d'implanter le contrôleur d'attribut (par une *hashtable*) et le contrôleur de liaison (toujours avec une *hashtable*), elle implante le contrôleur de cycle de vie et elle est en charge de la communication avec le monde patrimonial. Elle fonctionne avec une autre classe, décrite section 7.4.4, la classe *RemoteWrapper* qui sert d'*effecteur* et communiquent entre elles par RMI. La classe GenericWrapper doit résoudre les arguments des méthodes décrites dans le WDL pour transmettre le nom de la méthode à exécuter avec ses arguments à la classe *RemoteWrapper*. Pour résoudre les arguments, elle fait appel à la classe *AttributeUtil* qui implante la fonction de navigation, détaillée section 7.4.5. Lors de la création

du composant, une instance de *GenericWrapper* est créée et la nouvelle instance va lire le fichier WDL associé au composant pour stocker tous les noms de méthodes appelables pour ce type de composant, les nom des méthodes Java qui les implantent et enfin les arguments (sous forme de chaîne pointée non déréférencée) à passer à la méthode Java.

Pour les composants noeuds La classe interne au composant noeud est la classe *GenericNode*. Elle implante elle aussi le contrôleur d'attribut et le contrôleur de liaison. Elle est en charge de l'appel à la politique d'allocation de noeuds, de préparation de noeuds (copie de l'environnement d'exécution nécessaire au fonctionnement de TUNe) et du lancement à distance des *RemoteWrapper*. Pour chaque composant noeud créé, une instance de cette classe est créée. Lors de sa construction elle lit le fichier contentant la liste des noeuds et pour chaque noeud crée une instance de la classe *Node* en lui passant entre autre les informations suivantes :

- répertoire d'installation distant,
- utilisateur et mot de passe distant,
- répertoire d'installation de la JVM.

Cette structure de donnée contient aussi un booléen permettant de savoir si l'environnement d'exécution de TUNe a déjà été déployé ou non sur ce noeud. Il est initialisé à *false*. L'ensemble des noeuds créés est ajouté à une liste propre à cette instance de GenericNode. Enfin, elle interprête le fichier WDL passé en paramètre par l'attribut *allocator* du diagramme de classe et stocke de la même manière que la classe *GenericWrapper* les différentes méthodes appelables avec leurs paramètres.

7.1.2 Génération des graphes de reconfiguration

Les diagrammes état-transition sont représentés dans TUNe comme des graphes. Chaque état est représenté par une instance de la classe *OneChartState* (voir section 7.4) et chaque transition est représenté par une référence vers l'instance de *OneChartState* qui est l'état suivant. Pour simplifier la programmation, chaque instance de classe possède aussi une ou des références vers le ou les état(s) précédent(s). Pour chaque état d'un diagramme état transition, une instance de la classe *OneChartState* est créée. Cette instance permet d'exécuter le diagramme et de reconfigurer le SR. Lors de l'analyse du fichier XML, chaque instance est créé avec les attributs suivants :

- type de l'état : *initial, final, fork, join ou action*,
- dans le cas du type action, la chaîne pointée représentant le contenu de l'état.

A la fin de l'analyse du diagramme de reconfiguration, pour chaque ligne dans le commentaire du diagramme état transition (qui correspond aux noms des notifications qui vont pouvoir déclencher la reconfiguration), une entrée dans une *hashtable* de l'application nommée *hashtable de reconfiguration* est créée avec comme clé le

nom de la notification et comme valeur la référence vers l'état initial du graphe de reconfiguration associé.
Une fois tous les diagrammes analysés, TUNe passe à la phase de déploiement automatique.

7.2 Déploiement des logiciels patrimoniaux

La méthode *startFc* du composant composite applicatif est appelée. Cela a comme effet d'appeler toutes les méthodes *startFc* des composants applicatifs. La méthode *startFc* est implantée dans la classe GenericWrapper. Cette méthode va exécuter les actions suivantes :

- demande à l'allocateur de noeud courant pour cette famille de noeud de choisir un noeud physique (voir ci-dessous pour l'allocation d'un noeud) par l'interface Fractal cliente Node,
- copie par la commande *scp* du fichier archive contenu dans l'attribut *wrapper* de la classe uml,
- décompression de l'archive sur le noeud distant par *ssh*,
- demande de création d'un *RemoteWrapper* sur le noeud distant (voir section 7.2 pour cette création),
- création d'une liaison RMI entre le *GenericWrapper* du composant courant et le *RemoteWrapper* nouvellement créé,
- requêtes *ping*[2] sur ce *RemoteWrapper* jusqu'à obtenir une réponse signifiant que la création s'est bien déroulée.

Ces créations se font en parallèle car les méthode *startFc* sont appelées en parallèle par Fractal. Une fois que toutes les méthodes *startFc* sont terminées, le diagramme de reconfiguration nommé *startchart* est alors exécuté.

Allocation des noeuds Lors de l'appel à la méthode *getNode* de l'interface Fractal Node, la classe GenericNode fait appel à une méthode d'allocation écrite en Java. Cette méthode d'allocation est trouvée dans le fichier WDL associé au noeud et en fonction de la politique courante d'allocation donnée dans l'attribut *allocator-policy*. Cette méthode Java, en fonction de la politique d'allocation qu'elle implante et de la liste de noeuds du *GenericNode* associé, renvoie alors un noeud de la liste de noeud au *GenericNode*. Ce dernier initialise alors le noeud en appelant la méthode *deploy()* de l'instance *Node*. Si le noeud a déjà été initialisé, rien ne se passe. Dans le cas contraire, les actions suivantes sont exécutées :

- création par *ssh* du répertoire d'installation distant,
- copie par *scp* des binaires de TUNe,
- exécution par *ssh* de la classe *RemoteLauncher*,

[2] envoi d'un paquet de données sur le réseau et attente de l'accusé de réception

– création d'une liaison RMI entre le *GenericNode* courant et le *RemoteLauncher* nouvellement créé,
– *ping* de l'instance de classe *RemoteLauncher* jusqu'à obtenir une réponse,
– passage du booléen du déploiement du noeud à *true*.

L'instance de la classe *RemoteLauncher* créée permet juste de créer des instances de *RemoteWrapper* sur la même JVM pour économiser les ressources du noeud.

7.3 Synthèse de l'analyse des diagrammes et du déploiement

Une fois les diagrammes analysés, les logiciels patrimoniaux déployés et avant l'exécution du *startchart* qui va réellement lancer l'exécution de l'application patrimoniale, l'architecture de TUNe est celle décrite dans la figure 7.3. Les seules variables d'ajustement seront les logiciels patrimoniaux déployés sur les noeuds et les noeuds alloués. Sur cette figure, au niveau du SR dans le *TUNe runtime*, deux

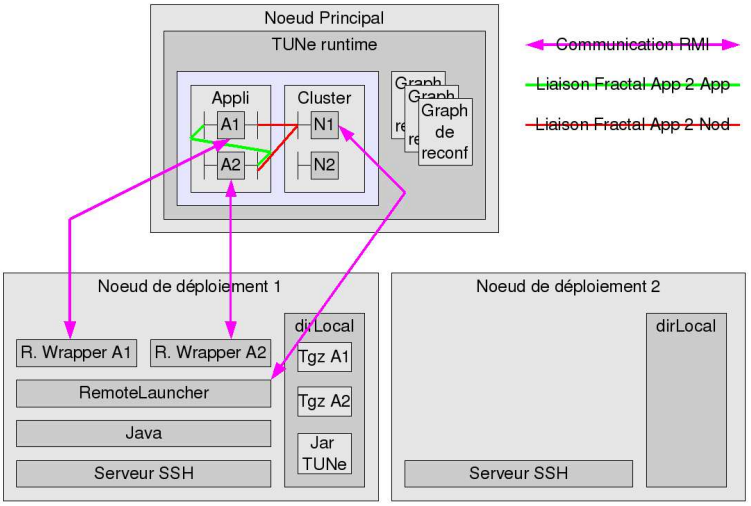

FIG. 7.3 – L'architecture de TUNe après le déploiement

GenericNode, N1 et N2, on été créés mais un seul est utilisé et deux composants *Applicatifs*, A1 et A2 sont créés et reliés entre eux.
On peut aussi voir les communications RMI entre les *GenericWrapper* et les *RemoteLauncher* représentée par les flèches allant du noeud principal au noeud de déploiement.
Enfin, on peut voir sur cette figure deux noeuds de déploiement, *Noeud de déploiement 1* et *Noeud de déploiement 2* où seul le noeud 1 a servi pour le déploiement.

Sur le noeud ayant servi pour le déploiement, on peut voir la couche applicative, du *Serveur SSH* jusqu'au *RemoteWrapper*. Sur le deuxième noeud, le dirLocal est déjà créé en dehors de toute action de TUNe. Il peut s'agir d'un répertoire système tel que */tmp*.

7.4 Démarrage effectif de l'application et la reconfiguration

Une fois le logiciel patrimonial déployé sur les noeuds distants et les *RemoteWrapper* lancés, les méthodes définies dans les fichiers WDL et appelées depuis les diagrammes de reconfiguration peuvent être appliquées sur l'application patrimoniale. Le *startchart*, un diagramme de reconfiguration obligatoire, permet à l'utilisateur de TUNe de gérer lui même le cycle de lancement de l'application TUNifiée. Généralement, ce diagramme fera appel à l'ensemble des composants d'une même famille pour les configurer puis les démarrer. Nous allons présenter ici la manière dont un diagramme de reconfiguration et sa représentation interne sous forme de graphe fonctionne. Les instances des classes *OneChartState* permettent de contrôler le flot de contrôle d'exécution et d'appeler les méthodes contenue dans le nom des états. Pour exécuter le startchart, TUNe recherche dans la hashtable de reconfiguration la clé *startchart*. Cette recherche permet de trouver l'état initial de ce diagramme. Avant l'exécution du startchart, une phase de validation du SR est lancée (voir section 7.4.3). Après l'exécution du startchart, l'application patrimoniale est lancée et TUNe se met en attente de notifications provenant du monde patrimonial.

7.4.1 Le contrôle du flot d'exécution

Comme dit plus haut, une instance de la classe *OneChartState* peut être de type *action*, *initial*, *terminale*, *fork*, *join*. Lors de l'exécution du diagramme de reconfiguration, la méthode *startRunner* de l'instance représentant l'état initial est exécutée. Dans le cas de l'état initial, un thread est créé et la méthode *startRunner* de l'état suivant est appelé. Si l'état suivant est de type :

- *fork* : autant de thread que d'états suivants sont créés. Les références des threads sont stockées dans l'instance de la classe de type *Fork* et les méthodes *startRunner* des instances suivantes sont appelées,
- *join* : les références des threads contenues dans les instances *Fork* précédentes sont récupérées. Les threads sont alors tous attendus avant d'appeler la méthode *startRunner* de l'instance suivante,
- *action* : une méthode sur le logiciel patrimonial est exécutée (voir section 7.4.4),
- *final* : dans le cadre du déroulement du diagramme rien n'est effectué.

Le flot d'exécution est donc contrôlé par le type affecté à l'instance de classe *OneChartState* et par les références sur les états suivants et précédents.

7.4.2 La reconfiguration

Un diagramme de reconfiguration est parcouru au plus trois fois pour une même reconfiguration.

La première passe Elle permet de détecter si un état modifiant la structure du SR est présent dans le diagramme. Un tel état sera un état de type "++" ou "- -" et sera appelé *état de modification structurelle*. Les autres états d'action ne sont pas exécutés lors de cette première passe qui s'avère donc très rapide. Si aucun état de reconfiguration n'est trouvé, TUNe passe directement à la *passe d'action*. Sinon TUNe exécute d'abord la *passe de validation*.

La passe de validation Lors de cette passe, le SR est clôné lors de l'appel à la méthode *startRunner* de l'état initial. Les seuls états interprêtés seront les état de modification structurelle. Nous détaillons ici l'ajout de composant, mais des actions similaires sont menées lors du retrait d'un composant. Lors du passage sur un de ces états, les actions suivantes sont entreprises :

- recherche du type de composant à créer (voir section 7.4.5),
- création du composant applicatif Fractal dans le SR clôné,
- création des liaisons vers les autres composants attachés au composant nouvellement créé en essayant de respecter les règles du diagramme de déploiement,
- création de la liaison vers le *GenericNode* associé.

A la fin de l'exécution du diagramme, lors du passage sur l'état *terminal*, une validation du SR clôné et modifié est engagée (voir section 7.4.3). Si le SR est valide, le SR nouvellement clôné remplace le SR d'origine et la passe d'action est exécutée. Si le SR est invalide, les SR ne sont pas échangés, une exception est levée et la passe d'action n'est pas exécutée. Dans le meilleur des cas (SR valide), cette passe ne modifie que la couche SR de TUNe.

La passe d'action Cette passe ne modifie que la couche patrimoniale de TUNe. Lors de la rencontre d'un état de modification structurelle **et** si le diagramme est un diagramme modifiant la structure du SR, la méthode *startFc* des composants créés lors de la passe de validation est appelée. Cela aura pour effet de déployer les nouveaux composants au niveau patrimonial comme lors du déploiement initial. De plus, les variables affectées dans le diagramme de reconfiguration sont remplacées par les références aux nouveaux composants (en vue de pouvoir appeler des méthodes sur ces derniers par des appels du type *nouveauComposant.configure* par exemple,

voir section 6.3.3). Dans le cas d'un état d'action qui ne touche pas à la structure du SR, les actions suivantes sont menées :

- recherche du composant cible (voir section 7.4.5),
- appel de la méthode *meth* du GenericWrapper sur le composant trouvé lors de l'étape ci-dessus. Cette méthode va interagir avec le RemoteWrapper pour apporter les modifications à couche patrimoniale (voir section 7.4.4).

Synthétiquement, la même démarche est entreprise lors des reconfigurations que lors du déploiement initial : création du SR puis lancement du logiciel patrimonial lors de l'initialisation de l'application, modification du SR puis lancement des nouveaux composants lors d'une phase de modification structurelle pour une reconfiguration.

7.4.3 La validation du SR

Le SR est validé après la phase de déploiement et avant l'exécution du startchart (au cas où des incohérences seraient présentes dans le diagramme de déploiement, voir section 6.3.1) ou à chaque fin d'exécution de diagramme de reconfiguration. Le but de cette phase est de valider la cohérence du SR avec le diagramme de déploiement donné par l'utilisateur. Elle se passe en deux phases :

- recherche d'un des composants dans le composant composite applicatif,
- recherche de liaisons associées à ce composant dans **le diagramme de déploiement**. Ceci permet de connaître les cardinalités,
- comptabilisation des liaisons entrantes sur l'interface serveur et sortantes sur les interfaces clientes,
- vérification que le nombre de liaison compté est en accord avec le nombre de liaisons maximum et minimum.

Cette étape est faite pour tous les composants du composant composite applicatif. Dans le cas où un composant enfreint les règles édictées dans le diagramme de déploiement, si c'est lors de l'exécution du startchart, une exception est levée et l'application arrêtée. Dans le cadre d'un diagramme de reconfiguration, l'erreur est signalée au runtime de TUNe qui prendra les décisions adéquates (non échange des SR, exception et abandon de la reconfiguration courante).

7.4.4 La communication entre le SR et le monde patrimonial

Cette communication est rendue possible par la paire *GenericWrapper* et *RemoteWrapper* qui communiquent par RMI avec l'interface *WrapperItf*. Lors de l'appel à la méthode *meth* (figure 7.2) du *GenericWrapper*, le nom de la classe, le nom de la méthode et les chaînes d'arguments du WDL sont passées au GenericWrapper. Le GenericWrapper résout alors les arguments (variables commençant par $ dans les

chaînes d'argument) grâce à la classe *AttributeUtil* (voir section 7.4.5).
Ainsi, la chaîne :

```
$node.javahome/bin/java -cp $dirLocal DistributedProbe '$probed.PID' '$tubeAddr' '$
    probed.srname' la.xml '$probed.nodeName'
```

peut être transformée en :

```
/usr/local/jdk1.6/bin/java -cp /tmp DistributedProbe '1234' '/tmp/probe_0' 'la_0'
    la.xml 'localhost'
```

Les arguments résolus, la méthode *meth* du *RemoteWrapper* peut être appelée avec le nom de la classe, le nom de la méthode et les arguments résolus.

Le *RemoteWrapper* Cette classe implante elle aussi l'interface *WrapperItf*. Dans le *RemoteWrapper*, l'appel de méthode est effectif avec les arguments reçus dans le paramètre *args*. Les actions suivantes sont effectuées :

- recherche par introspection de la classe passée dans l'argument *className*,
- recherche par introspection de la méthode passée dans l'argument *methName*,
- création par introspection d'une instance de la classe *className*,
- appel par introspection de la methode *methName* de la classe *className* avec les arguments *args*.

Les modifications du logiciel patrimonial sont donc effectuées par les *RemoteWrapper* qui sont les effecteurs de TUNe.

7.4.5 La résolution des chaînes pointées

Nous avons vu dans la partie *contribution* que des chaînes pointées peuvent être passées au niveau des notifications, dans le nom des états des diagrammes de reconfiguration et enfin dans les balises *value* du WDL.
Nous retrouvons ces chaînes lors de la résolution des arguments dans le *GenericWrapper*, dans la détermination du composant cible dans les états *OneChartState* et dans la classe *NotifyClass* que nous verrons section 7.5.
Ces chaînes sont composées de mots séparés par des espaces. Un mot peut commencer par un $, ce qui signifie que c'est une variable à résoudre. De plus, ces chaînes peuvent contenir des '.' pour naviguer dans le SR.
La résolution de ces chaînes pointées est implantée dans la classe *AttributeUtil*. Cette dernière permet de :

- *findComponent* : trouver un composant (détermination du composant cible d'une notification ou appel de méthode dans un diagramme de reconfiguration). Cette méthode est appelée uniquement avec une chaîne pointée,
- *solveVar* : lire un attribut (résolution des arguments pour appel de méthodes pour les effecteurs par exemple). Cette méthode est appelée avec une chaîne pointée qui peut comprendre plusieurs $,

– *solveVar avec un paramètre supplémentaire* : écrire un attribut (cas particulier de l'affectation dans les diagrammes de reconfiguration). Cette méthode est appelée avec une chaîne pointée qui contient un '='.

findComponent : recherche d'un composant

Cette méthode est appelée avec la référence d'un composant (qui est le composant source, le composant à partir duquel la navigation dans le SR va débuter) et une chaîne pointée qui permet d'aller trouver un autre composant dans le SR. Elle procède ainsi :

1. affectation du composant courant avec le composant passé en paramètre
2. suppression du $ en tête de chaîne
3. extraction de la chaîne de caractère entre le début de la chaîne et le premier point. On nommera cette chaîne le composant cible
4. si le composant cible est une chaîne de caractère vide, on a terminé, renvoyer le composant courant
5. sinon
   (a) si le composant cible est nommé *this*, passage à l'étape 6
   (b) sinon
      i. à l'aide du contrôleur d'attribut du composant courant demander le composant lié à l'interface qui porte le nom du composant cible. La navigation avec les liaisons est possible car on a vu que les interfaces clientes portent le nom du composant lié dans le diagramme de déploiement ou le nom de la liaison de ce même diagramme
      ii. si le composant existe, affecter le composant courant avec la référence du composant trouvé
      iii. sinon renvoyer null
   (c) fin si
6. supprimer de la chaîne initial le nom du composant cible, rajouter un $ devant et se rappeler avec cette nouvelle chaîne pointée et la référence au composant courant
7. fin si

solveVar : déférencement des variables dans une chaîne

Dans le cas d'un attribut à lire ou écrire :

1. décomposer la chaîne pointée en mots
2. pour tous les mots
3. si le mot ne commence pas par un $, passer au mot suivant
4. si le mot commence par un $, extraire le dernier mot de la chaîne pointée et le stocker dans une variable nommée *attribut*
5. chercher le composant avec *findComponent*
6. si le composant existe, à l'aide de son contrôleur d'attribut, lire la valeur de l'attribut et remplacer toute la chaîne pointée commençant par un $ par la valeur de l'attribut dans la chaîne d'origine
7. sinon remplacer toute la chaîne pointée commençant par un $ par la chaîne *null*

solveVar : affectation d'un attribut

Cette méthode fonctionne comme *solveVar* pour remplacer des variables dans une chaîne mais elle coupe la chaîne en deux de part et d'autre du '=', résout la partie gauche et se sert du contrôleur d'attribut pour écrire l'attribut avec la partie droite.

7.5 Attente des notifications pour reconfigurer

Une fois l'application déployée et le startchart exécuté, TUNe se met en attente de notification du monde patrimonial pour reconfigurer ou réparer l'application déployée. Ces notifications proviennent de tube nommé[3] que chaque *RemoteWrapper* crée à son lancement. Lors du lancement du *RemoteWrapper*, un tube nommé et un thread sont créés. Le thread est une boucle infinie qui lit le tube nommé. Pour émettre une notification, il suffit alors à une application du monde patrimonial telle qu'une sonde d'écrire dans le tube nommé une chaîne de caractères divisée en trois champs et séparés par des ';' :

- *name* : nom de la notification,
- *component* : provenance de la notification. Ce champs doit être le nom au niveau du SR du composant générant la modification. Il faut donc penser lors de l'écriture des WDL encapsulant des logiciels pouvant émettre des notifications à passer en paramètre l'attribut *srname*. Ce champs permettra de déterminer le composant source du diagramme de reconfiguration associé à la notification. Le mot clé *this* est autorisé et TUNe déterminera le composant à l'origine de la notification grâce au tube nommé utilisé pour envoyer la notification,
- *arg* : un argument qui est une chaîne pointée.

Lors de la réception d'une notification, le *RemoteWrapper* appelle par RMI le *GenericWrapper* avec la notification brute (sans résoudre les chaînes pointées), en séparant juste les champs. L'appel RMI appelle la méthode *notify* d'une instance de la classe *NotifyClass*, classe associée au *GenericWrapper*. Cette méthode permet de trouver le composant qui sera le composant source de la reconfiguration grâce au champs *component*, puis de résoudre la chaîne pointée de l'argument *arg* grâce à la classe *AttributeUtil* et enfin de chercher tous les graphes de reconfiguration associés à cette notification grâce à la hashtable de reconfiguration et de les exécuter.
Ce cycle est résumé par la figure 7.4. On s'aperçoit avec cette figure que TUNe implante aussi une boucle *sondes-actuateurs-manager* où :

- les sondes sont des applications patrimoniales particulières qui peuvent envoyer des notifications,
- les actuateurs sont les *RemoteWrapper* qui exécutent des méthodes du WDL,
- le manager est composé de l'ensemble des diagrammes de reconfiguration.

A la différence des autres approches, dans TUNe, tout est configurable par l'utilisateur.

[3] pipe dans le monde UNIX

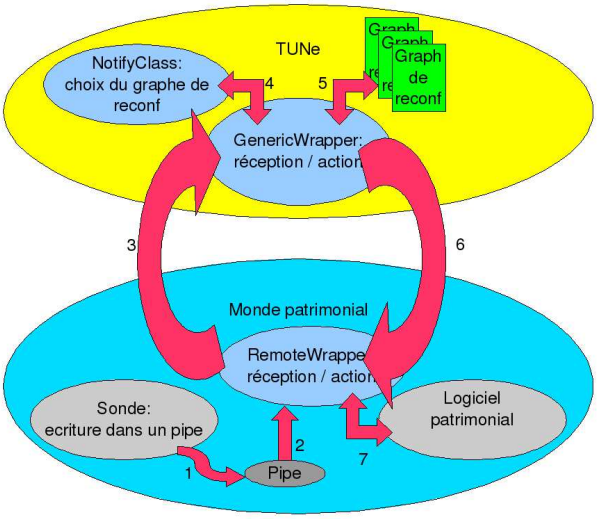

FIG. 7.4 – De l'envoi d'une notification (1) jusqu'à la modification du logiciel patrimonial (7)

7.6 Terminaison d'une application et terminaison de TUNe

Pour arrêter une application, le diagramme *stopchart* est appelé. Ce diagramme de terminaison est un diagramme qui s'exécute dans le même contexte que le start-chart. Cela permet à l'utilisateur de choisir l'ordre d'arrêt de son application patrimoniale. Tous les noeuds où cette application était déployée sont alors recherchés. Si aucune autre application s'exécute sur le noeud **avec les mêmes fichiers d'archive**, ces derniers sont supprimés. C'est la phase de *dé-déploiement*. Si aucune autre application ne s'exécute sur le noeud, les fichiers binaires de TUNe sont à leur tour supprimés et le noeud devient marqué *non déployé*.
L'arrêt de TUNe correspond à l'arrêt de toutes les applications s'exécutant sur le TUNe arrêté. Les binaires de TUNe sont donc supprimés au fur et à mesure que les applications sont arrêtées. Lorsque TUNe est enlevé de tous les noeuds, l'exécution de TUNe peut alors rendre la main au système d'exploitation qui l'a démarré.

Chapitre 8

Validation

Nous présentons dans cette section les résultats que nous avons obtenus pour trois logiciels patrimoniaux TUNifiés. Nous avons TUNifié Diet, qui nous a servi d'exemple durant cette étude mais aussi l'ensemble Apache-Tomcat-MySQL et Xen. Pour chaque logiciel, nous avons étudié la réparation avec TUNe ainsi que le dimensionnement dynamique en fonction de la charge des noeuds distants. Dans les sections suivantes, nous voyons en détail les logiciels TUNifiés, les méthodes de test et enfin les résultats sous forme de courbes.

8.1 Diet

Nous présentons dans cette section les résultats que nous obtenons en auto administrant l'application patrimoniale Diet. Nous présentons tout d'abord les temps de déploiements et de configuration obtenus pour l'application entière puis nous montrons l'impact d'une réparation d'un SeD sur la charge du noeud accueillant le SeD et enfin l'impact d'un dimensionnement sur la charge du noeud surchargé.

8.1.1 Déploiement et configuration

Les temps de déploiement présentés dans le tableau 8.1 correspondent au temps de déploiement requis pour :

- interprêter les diagrammes,
- construire l'architecture à composants,
- allouer et préparer les noeuds pour recevoir TUNe,
- déployer TUNe et les archives des logiciels patrimoniaux,
- exécuter le *startchart*, ce qui correspond à la configuration du logiciel patrimonial.

L'application patrimoniale est composée d'un MA et sa sonde, un LA et sa sonde et enfin de 120 SeD. Les SeD sont reliés par groupe de 40 à une sonde, ce qui fait 4 sondes pour les 120 SeD. Dans le cadre de cette expérience, nous faisons varier le nombre de noeud en utilisant la politique d'allocation en Round Roubin.

Nombre de noeuds	**Temps de déploiement (s)**
20	116,75
40	155,12
60	187,25
80	209,38
100	257
110	270,96

TAB. 8.1 – Temps de déploiement en seconde en fonction du nombre de noeuds réservés au déploiement

8.1.2 Réparation

Pour cette expérience, nous avons lancés 1 MA, 1 LA, 2 SeD tous sur des noeuds différents. Chaque SeD est accompagné sur son noeud d'une sonde de réparation. Sur 12 autres noeuds nous avons déployés 12 clients qui émettent chacun 100 000 requêtes durant le temps de l'expérience. Nous arrêtons un SeD durant l'expérience et observons la charge processeur des noeuds comportant les SeD. La figure 8.1 présente ces charges en fonction du temps en seconde.

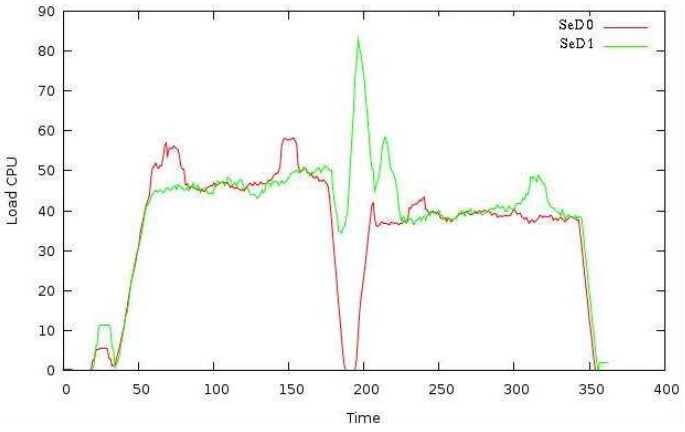

FIG. 8.1 – Charge des noeuds comportant chacun un SeD lors d'une panne en fonction du temps en seconde

A t=190, nous arrêtons un SeD, SeD0. La charge monte à t=200 sur le noeud hébergeant SeD1, le temps que le LA redirige les requêtes sur le SeD fonctionnant encore. Dix secondes plus tard, à t=210, la charge sur SeD0 remonte. Le LA mettant 10s en moyenne pour détecter la panne d'un SeD et pour rediriger les requêtes, on peut déduire que TUNe a relancé immédiatement le SeD en panne.

8.1.3 Redimensionnement

Dans le cadre de cette expérience, nous avons repris le déploiement initial de l'expérience de réparation (section 8.1.2) mais en lançant uniquement un seul SeD. Les sondes déployées avec les SeD permettent cette fois-ci de lancer un SeD supplémentaire quand le noeud où la sonde est déployée atteint la charge de 70%. La courbe figure 8.2 présente la charge processeur du noeud hébergeant le SeD initialement déployé et du noeud allant recevoir le SeD lors du redimensionnement.

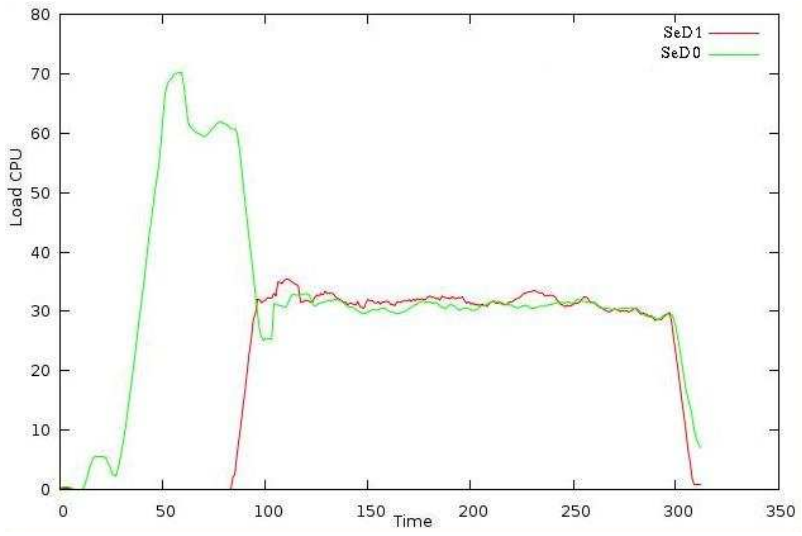

FIG. 8.2 – Charge des noeuds comportant chacun un SeD lors d'un redimensionnement en fonction du temps en seconde

A t=60, la charge du noeud comportant SeD0 atteint 70%. La sonde du SeD0 demande alors un redimensionnement. A t=80, le second SeD, SeD1 commence à monter en charge. Sachant qu'il faut au LA 10s pour rediriger les requêtes, on peut déduire que TUNe a mis 10s pour recevoir la notification, la traiter, déployer les binaires sur un nouveau noeud et enfin configurer le SeD et le LA.

8.2 Apache-Tomcat-MySQL

Une architecture Apache-Tomcat-MySQL est composée de trois tiers chacun relié par un middleware permettant d'assurer la communication entre eux. Dans le cadre de cette étude, nous avons retenu *mod_jk* pour la communication entre Apache et Tomcat et *c-jdbc* pour la communication entre Tomcat et MySQL. L'application permettant l'évaluation est RUBiS [1], une application de *e-commerce* généralement utilisée pour tester cet ensemble 3-tiers. Des clients envoyant des requêtes simulent une navigation et des transactions de clients réels qui sont envoyés à l'Apache qui répond directement aux clients pour les pages statiques alors qu'il passe les requêtes aux Tomcat pour les pages dynamiques. Les Tomcat traitent les requêtes en faisant appel le cas échéants aux MySQL à leur disposition. Nous étudions pour cette application le cas de la panne d'un Tomcat, de la surcharge d'un noeud comportant un Tomcat ou de la sous-charge d'un noeud comportant deux tomcats. Ces expériences ont été réalisées sur Grid 5000 avec l'allocateur de noeuds OAR.

8.2.1 Panne d'un Tomcat

Pour ce cas de figure, nous utilisons 1 Apache, 2 Tomcat et 3 MySQL. A un instant donné, un opérateur arrête brutalement un Tomcat par le biais d'un *kill -9*. Cette dernière instruction permet sous Linux d'arrêter un processus sans lui laisser le temps d'enregistrer sont état ou même de prévenir les processus auquel il est rattaché qu'il va s'arrêter. Ce comportement nous convient pour cette expérience car c'est celui qui est tenu lors d'une panne brutale réelle. Une sonde est reliée à chaque tiers patrimonial et teste à intervalle régulier la présence du PID*Process IDentifier* de chaque tier auquel elle est reliée. Lors de la non détection d'un PID, une notification est envoyée à TUNe pour que le tiers en panne soit réparé et l'architecture remise en forme. La courbe 8.3 montre la charge des deux noeuds.

L'opérateur arrête le processus tomcat_1 à la 19ème seconde. La charge du noeud tomcat_1 chute alors pour devenir quasi nulle. A la seconde 23, la charge augmente sur le noeud tomcat_2. Le délai de 4s est le délai normal pour que les mod_jk prennent en compte la panne : les requêtes d'Apache basculent alors toutes vers le tomcat_2. La charge du noeud tomcat_2 augmente alors jusqu'à presque doubler. A la seconde 37, TUNe répare le tomcat_1. La charge des noeuds met alors environ 4s pour s'équilibrer et devenir comme avant la panne. TUNe a donc détecté la panne du tomcat_1 arrêté par l'utilisateur et l'a automatiquement réparé. Couplé au middleware mod_jk, l'interruption de service aura été quasi nulle et le noeud tomcat_2 aura été surchargé juste quelques secondes.

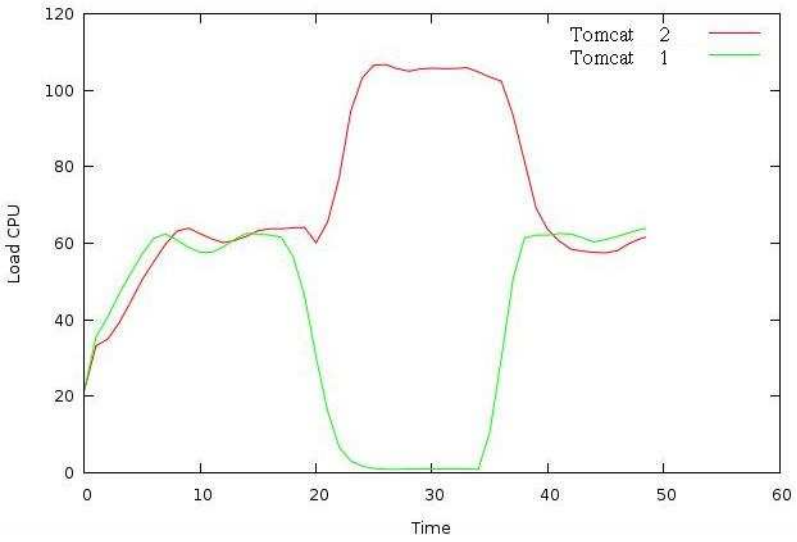

FIG. 8.3 – Charge des noeuds comportant un Tomcat lors d'une panne en fonction du temps en secondes

8.2.2 Surcharge d'un noeud

Dans ce cas de figure, nous utilisons la même architecture que lors de la panne d'un Tomcat section 8.2.1 mais avec un seul Tomcat lors du déploiement. Une sonde reliée à ce Tomcat permet de sonder la charge du noeud comportant ce Tomcat et d'envoyer une notification à TUNe lorsque la charge dépasse 35% (cette charge est faible mais il faudrait trop de bases de données et de clients pour pouvoir faire la simulation avec une charge plus élevée). TUNe va alors déployer un autre Tomcat sur un autre noeud pour soulager le premier noeud. La simulation se fait toujours avec RUBiS et le nombre de requêtes est progressivement augmenté sur l'application en faisant varier le nombre de clients. La courbe 8.4 montre la charge du noeud comportant le Tomcat initialement déployé.

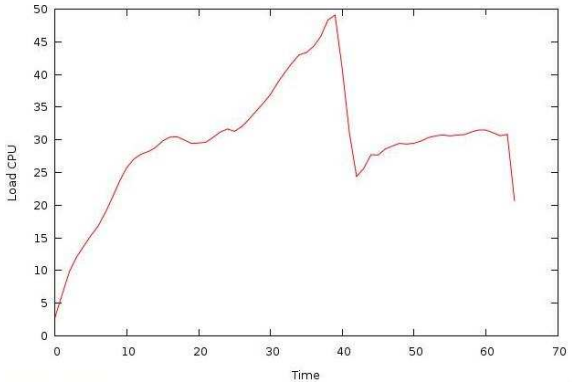

FIG. 8.4 – Charge du noeud comportant le Tomcat déployé initialement

On peut voir sur cette courbe que la charge monte progressivement jusqu'à atteindre les 35%. Cette valeur dépassée, la notification de surcharge est envoyée à TUNe qui va déployer un autre Tomcat sur un autre noeud. La charge du noeud chute alors brutalement pour remonter et se stabiliser. Cette expérience montre que TUNe a détecté la surcharge et a pu prendre les moyens approprié pour la faire cesser.

8.2.3 Sous-charge d'un noeud

Ce cas de figure peut se retrouver à la suite du précédent concernant la surcharge section 8.2.2. En effet, la surcharge peut être due dans le monde réel à un pic de requêtes temporaire. Une fois ce pic passé le Tomcat déployé en plus devient inutile et peut être supprimé. C'est ce que nous avons simulé dans ce troisième cas de figure. L'architecture utilisée est la même initialement que lors de la panne du Tomcat section 8.2.1 mais avec en plus une sonde de sous charge. Cette sonde va envoyer une notification dans le cas où la charge du noeud comportant le tomcat sondé passe en dessous de 25% pour que TUNe supprime le Tomcat. En vue de pouvoir laisser s'établir une charge supérieure à 25%, la sonde possède un chien de garde qui lui interdit d'envoyer des notifications avant 5 secondes de fonctionnement. Nous avons déployé les clients avant de déployer l'application trois tiers et en nombre suffisant pour que les noeuds soient chacun chargés à 30%. Nous faisons ensuite descendre la charge en supprimant petit à petit des clients. Les deux Tomcats étant sondés simultanément et la charge équitablement répartie par les mod_jk sur les deux Tomcats les deux noeuds devraient passer en même temps en dessous de 20% de charge. Nous pouvons alors vérifier le maintien de la cohérence de l'architecture par TUNe qui doit interdire d'enlever un deuxième Tomcat. La courbe 8.5 présente la charge des deux noeuds comportant chacun un Tomcat.

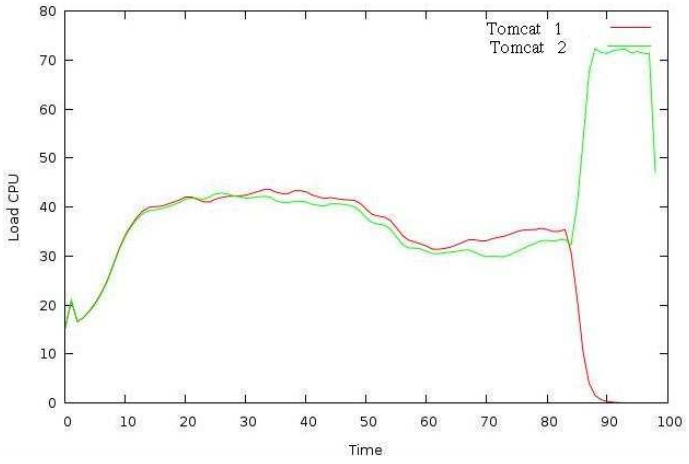

FIG. 8.5 – Charge des noeuds comportant les Tomcats déployés initialement

Sur cette courbe, on peut voir les charges s'équilibrer aux alentours de 30% ce qui était l'objectif. A partir de la seconde 37, l'opérateur diminue le nombre de clients envoyant des requêtes ce qui a pour effet de diminuer la charge des noeuds. A la seconde 45, la charge des deux noeuds passe en dessous de 25%. Deux notifications sont alors envoyées à TUNe : les notifications de sous charge du noeuds tomcat_1 et tomcat_2. La première arrivée est celle de tomcat_1. TUNe arrête alors le processus tomcat_1 ce qui a pour effet de faire chuter la charge du noeud tomcat_1. La notification du tomcat_2 est quant à elle refusée par TUNe pour maintenir l'application cohérente avec le diagramme de déploiement. Le tomcat_2 n'est donc pas arrêté et la charge du noeud tomcat_2 augmente pour absorber la charge des deux tomcats et se stabiliser à 50%. TUNe a donc ici répondu à une notification, fait du dimensionnement dynamique et maintenu cohérente l'application.

8.3 Xen

Xen [27], développé par l'université de Cambridge aux Royaume Uni permet de faire fonctionner plusieurs systèmes d'exploitation *invités* (ou *domU*) virtuels sur un même noeud physique, *hôte* (ou *dom0*). Les systèmes d'exploitation invités partagent les ressources physiques du noeud hôte tout en étant complètement isolés les uns des autres. Les systèmes de virtualisation sont de plus en plus utilisés de nos jours depuis l'apparition des instructions de virtualisation au sein des nouveaux micro processeurs qui permettent une augmentation de la vitesse d'exécution des systèmes hôtes. Xen fonctionne directement au dessus du matériel tel que montré dans la figure 8.6.

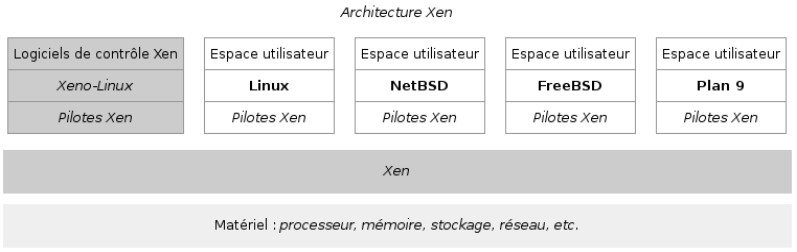

FIG. 8.6 – Une architecture Xen exécutant 4 systèmes d'exploitation sur un même noeud hôte.

La mise dans un processus Xen d'un système d'exploitation complet permet de migrer facilement d'un noeud à l'autre le système d'exploitation invité. De plus, Xen possède des fonctionnalités permettant de ne pas interrompre les liaisons réseaux lors de la migration du système d'exploitation invité. Ces fonctionnalités permettent un large éventail d'applications incluant :

– la mutualisation des ressources et donc baisse des coûts liés à l'achat ou location des noeuds physiques et à l'énergie consommée par ces noeuds,

– le déploiement d'une même image d'un système d'exploitation invité sur plusieurs noeuds aux caractéristiques différentes (architecture, système d'exploitation, ...),
– la portage d'anciennes applications sur des machines récentes sans réécriture,
– ...

Nous nous sommes intéressés à la mutualisation des ressources dans le cadre d'un centre d'hébergement Web. Grâce à la virtualisation, il est facile de créer des serveurs dédiés (tournant sur un système d'exploitation virtuel) en ne réservant que les noeuds physiques nécessaires à une bonne performance de l'ensemble des serveurs dédiés.
Ainsi, en couplant la migration permise par Xen (et le fait qu'on ne perde pas de requêtes lors de ces migrations) et l'auto administration permise par TUNe (avec différentes applications patrimoniales), nous pouvons automatiquement allouer des noeuds lorsque la charge du système hôte devient trop importante, migrer les serveurs dédiés sur les nouveaux noeuds et rendre les serveurs alloués lorsque la charge diminue, le tout de manière transparente et pour différents types de serveurs.
Nous présentons dans la suite de cette section un scénario de migration de systèmes virtuels dans le cadre de la mutualisation d'hébergement.
Ces expériences ont été réalisées sur Grid 5000 en utilisant le logiciel Kadeploy qui permet de lancer un noeud avec une image de système d'exploitation.

Pour cette démonstration nous proposons l'architecture suivante :
– deux noeuds physiques dom0,
– une architecture composée uniquement de clients et de Tomcat,
– un Tomcat par domU,
– TUNe qui est exécuté sur un noeud tiers ne comportant pas Xen nommé TUNE node.

Ainsi après déploiement de deux Tomcats sur deux domU, l'architecture sera celle montée figure 8.7.

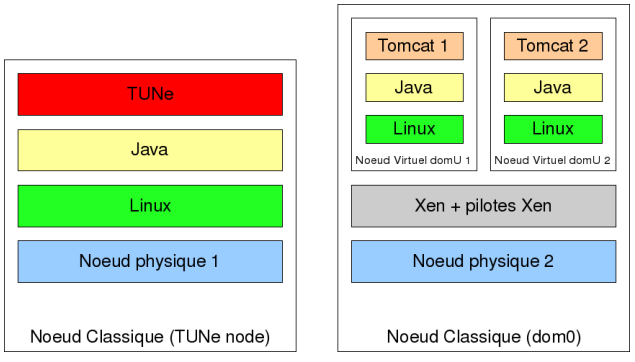

FIG. 8.7 – Une architecture après déploiement comportant un TUNe et deux Tomcat sur des systèmes virtuels

Étudions maintenant le scénario de migration mis en oeuvre et les résultats.

8.3.1 Migration de processus grâce à Xen et TUNe

TUNe a été configuré pour réaliser le scénario suivant :

1. à t=400, un chien de garde sur le TUNe node lance une notification pour que TUNe déploie 2 domU sur le premier dom0,
2. à t=750, un chien de garde sur le TUNe node lance une notification pour déployer un Tomcat sur chaque domU,
3. de t=1100 à t=1500, augmentation du nombre de clients,
4. de t=1500 à 3500, maintien de la charge,
5. de t=3500 à 5000, diminution de la charge en diminuant le nombre de client.

Chaque dom0 possède une sonde de charge déployée par TUNe qui provoque une allocation de noeud puis la migration d'un domU lorsque la charge dépasse 45% et qui provoque une migration de domU puis une libération du noeud lorsque la charge est en dessous de 10%. Après déploiement, l'architecture est donc celle montrée figure 8.7. La courbe 8.8 présente la charge appliquée aux Tomcat en fonction du temps. La charge est proportionnelle au nombre de client.

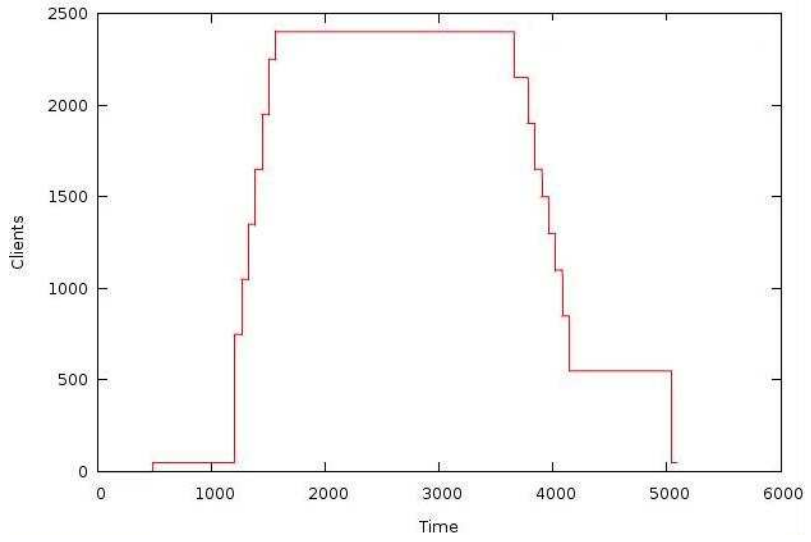

FIG. 8.8 – La charge appliquée aux Tomcat par unité de temps

Comme prévu par le scénario, à t=1100, la charge commence à augmenter.

La courbe 8.9 donne la charge du premier dom0, celui où sont lancés initialement les domU et les Tomcats.

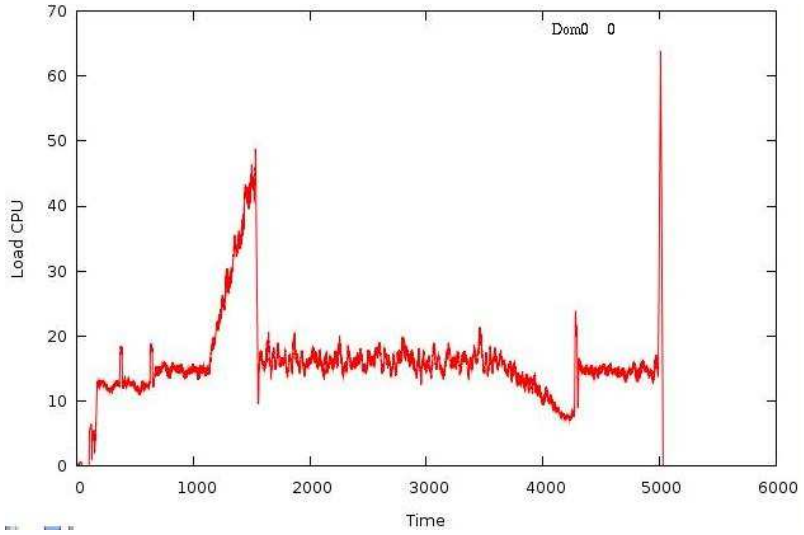

FIG. 8.9 – La charge du dom0 utilisé pour le déploiement initial

Les deux pics visibles à t=400 et t=750 correspondent respectivement au lancement des domU et des Tomcat. On peut donc constater que la charge monte à partir de t=1100 pour chuter brutalement à t=1500. En effet, la charge a dépassé 45% et la migration d'un domU a été réalisée. On peut constater cette migration sur la courbe montrant la charge du deuxième dom0 figure 8.10.

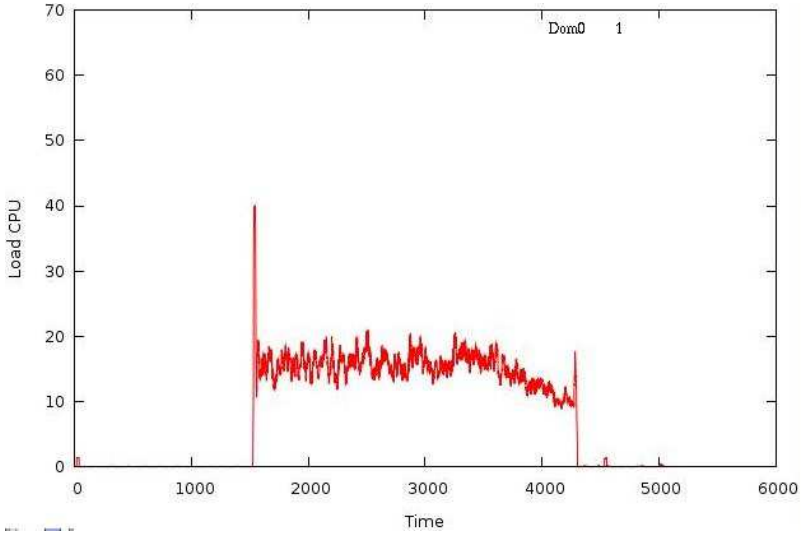

FIG. 8.10 – La charge du dom0 utilisé pour la migration

A t=1500, la charge du deuxième domU augmente brutalement à cause de la création du domU qui migre et le lancement de Tomcat. La charge se stabilise alors sur le premier dom0 et le deuxième dom0 jusqu'à t=3500, instant où le nombre de

clients diminue. La charge sur les deux dom0 diminue donc jusqu'à t=4300 où la charge sur le deuxième dom0 passe en dessous de 10%. Une migration inverse est alors effectuée ce qui fait remonter la charge du premier dom0. La charge du premier dom0 se stabilise alors jusqu'à t=5000 qui est la fin de l'expérience.

8.3.2 Conclusion

L'approche des machines virtuelles Xen couplée à TUNe est pertinente. Elle permet de réaliser de la mutualisation d'hébergement sans intervention humaine. Nous avons prolongé cette expérience par une autre expérience non présentée dans le cadre de ces travaux et qui permet de réduire la consommation énergétique d'un parc de machine.

Chapitre 9

Conclusion et perspectives

9.1 Conclusion

L'objectif de TUNe est la conception d'une approche d'administration autonome permettant le déploiement et la reconfiguration d'applications patrimoniales. Après l'étude des problèmes engendré par les autres approches d'administration autonome et en particulier Jade, nous avons pris le parti de créer un système d'administration autonome qui demande l'apprentissage du moins possible de nouveaux concepts à l'administrateur.

Un des principaux résultats de TUNe est de permettre à l'aide d'un langage universel, UML, de définir des politiques de déploiement et de reconfiguration. Nous réutilisons le diagramme de classe pour définir le déploiement et les diagrammes état transition pour définir les politiques de reconfigurations. Ces politiques s'appliquent au niveau des composants applicatifs et sont automatiquement répercutées dans le monde patrimonial grâce au langage de description de wrappers.

Le langage de description de wrapper, simple à appréhender, est basé sur des balises XML. Son principal rôle est de faire le lien entre les méthodes appelées dans le diagramme de reconfiguration et des méthodes Java dans le monde patrimonial qui permettent la reconfiguration à proprement parler. Le langage permet de se déplacer dans l'architecture à composants grâce à une notation pointée et ainsi permet d'utiliser des attributs des composants (qui sont associés aux logiciels administrés).

Les méthodes Java de reconfiguration doivent répondre à une interface simple et résoudre un problème simple pour être le plus réutilisables possibles. TUNe embarque en standard une librairie de méthodes Java génériques permettant les principales actions à effectuer sur le logiciel patrimonial, à savoir la configuration, le démarrage et l'arrêt.

Le deuxième résultat intéressant de TUNe est sa capacité à garder une architecture

cohérente. En effet, grâce au diagramme de déploiement et au langage de reconfiguration ne permettant de ne lier que les composants prévus pour être liés, l'architecture obtenue après une reconfiguration ne peux être invalide. L'administrateur peut aussi ajouter des gardes-fous (sous forme de cardinalités) dans le diagramme de déploiement pour contrôler le nombre de composants applicatifs pouvant être ajoutés ou enlevés.

9.2 Perspectives

TUNe est une plateforme récente qui demande des évolutions. A ce jour, nous envisageons trois séries d'évolution : créer un méta-modèle pour TUNe, enrichir les diagrammes de description de politiques d'administration et enfin permettre à TUNe l'interception de messages au sein du logiciel patrimonial déployé. Nous étudions dans cette section les travaux en cours et les pistes de chacun de ces points.

9.2.1 La méta modélisation

TUNe utilise des diagrammes UML pour représenter une grille ou une application. Ces diagrammes UML sont des diagrammes existants et prévus principalement pour de la modélisation d'application écrites en langage objet. Nous utilisons des outils d'édition de diagrammes UML pour la définition de ces diagrammes.

Cependant, les diagrammes d'UML ont parfois été détournés de leur usage premier pour être utilisés pour la définition de politiques d'administration dans TUNe. En d'autres termes, ils ne sont pas utilisés exactement pour ce qu'ils ont été définis. En définitive, il apparaît plus judicieux de considérer que les diagrammes permettant la définition de politiques d'administration dans TUNe sont des langages dédiés (des *Domain Specific Languages* ou DSL) à l'administration d'infrastructures logicielles. Il est alors possible de définir les méta-modèles de ces DSL, ce qui permet de définir précisément et formellement la syntaxe et la sémantique de ces langages. De plus, la définition de ces méta-modèles permet de construire des outils d'édition spécialisés (par exemple avec Eclipse/EMF/GMF) grâce auxquels nous pourrons commencer la validation des diagrammes avant même le lancement de TUNe. Une proposition de méta-modèle a déjà été publiée dans l'article suivant [5].

9.2.2 Autres diagrammes

Les langages de définition des politiques d'administration de TUNe (syntaxes graphiques sous forme de diagrammes ou textuelle) sont amenés à évoluer en fonction des besoins d'administration rencontrés.

A ce jour, il n'est pas possible de créer une hiérarchisation des noeuds composant le diagramme de noeud. Il pourrait être intéressant de prévoir une sorte d'héritage où un noeud peut hériter des propriétés d'un noeud abstrait. De plus, il pourrait être intéressant de représenter les liaisons entre les noeuds pour représenter la topologie du réseau.

Egalement, dans le cadre d'une administration d'infrastructures intergicielles comme J2EE, il pourrait être intéressant d'envisager l'introduction de diagrammes de déploiement d'applications au dessus des intergiciels. Ainsi nous pourrions avoir un diagramme de déploiement de servlets/beans au dessus d'un diagramme de déploiement de Tomcat/Jonas.

Enfin, nous nous sommes récemment intéressés au déploiement de TUNe. Dans sa version actuelle, TUNe administre une infrastructure logicielle répartie depuis un site centralisé. Il est cependant intéressant de répartir cette administration en déployant des serveur TUNe en différents points névralgiques de l'infrastructure. Le déploiement de TUNe est effectué par TUNe et spécifié sous la forme d'un diagramme particulier.

Le problème qui se pose alors est la liaison des différents diagrammes. Actuellement, pour lier le diagramme de déploiement et le diagramme de noeuds, nous utilisons l'attribut *host-family* appartenant à chaque composant logiciel. Cette solution est difficilement généralisable pour définir la liaison entre des diagrammes quelconques. Nous nous orientons vers une méthode plus générales pour lier des diagrammes, à savoir l'utilisation d'un langage de liaison de diagrammes.

9.2.3 Interception de messages

Comme spécifié dans la partie contribution, TUNe n'intervient pas au niveau de l'exécution de l'application patrimoniale (il n'est pas intrusif). TUNe n'a pas connaissance des messages échangés entre les différents tiers d'une application patrimoniale. La conséquence est notamment que la réparation redémarre les serveurs ayant fait l'objet d'une panne, mais les traitements en cours sont perdus.

Pour réaliser des réparations sans perte de traitements, il est nécessaire de connaître l'état du logiciel patrimonial à un instant t. Une des méthodes pour connaître l'état d'un logiciel patrimonial est d'intercepter les messages échangés, ce qui permet de rejouer les messages afin de ne pas perdre les traitements en cours au moment de la panne.

Annexe A

Autres approches d'administration autonome

Nous passons en revue dans cette annexe d'autres projets proposant des approches d'administration autonome. Ces projets rejoignent ceux que nous avons présentés en détails dans l'état de l'art ou reprennent des cas d'utilisation de notre approche TUNe. Nous présentons chaque projet et mettons en avant les critères qui les rapprochent d'un de nos projets de référence ou de TUNe.

A.1 Mécanismes de coordination décentralisés guidés par des patrons

T. De Wolf and T. Holvoet proposent à l' Université de Leuven en Belgique un système d'administration autonome guidé par les patrons [26]. Ces travaux sont semblables à ceux de l'équipe de O. Kephart présentés section 4.5 dans le sens qu'ils proposent des composants autonomes. Mais ils ont constaté que généralement ces composants autonomes sont écrits de manière ad-hoc pour une application et qu'il n'existe pas de processus permettant leur écriture et leur réutilisation. Ils se basent donc sur des processus de développement tels que le *Processus Unifié* [23] et proposent des patrons d'écriture de ces composants.

A.2 Reconfiguration dynamique pour des applications autonomes

Piotr Kaminski et son équipe de l'Université de Victoria se sont posés la question de l'évolution d'un logiciel durant son exécution [14]. C'est un mode de reconfigu-

ration où l'optimisation est réalisée en changeant la version d'un composant. Ils proposent donc une méthode pour pouvoir automatiquement mettre à jour un composant durant la phase de vie du logiciel. Cette approche repose sur un modèle à composants orienté Web Services tel que SCA présenté section 3.5. Leur idée est d'ajouter automatiquement des composants lors de l'évolution d'un composant pour se conformer à l'interface serveur de l'ancien composant et pour s'adapter à l'interface serveur du nouveau composant. Les composants sont donc reliés par ce que nous avons appelé liaisons composites, et un ou plusieurs composants adaptant les interfaces implantent la liaison composite tel que décrit figure A.1.

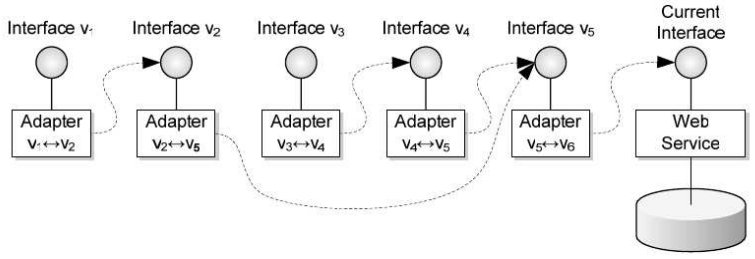

FIG. A.1 – Une chaîne pour adapter les version d'un composant

A.3 Les environnements virtualisés autonomes

Daniel Menasce et ses étudiants de l'Université Georges Mason ont proposé des approches de systèmes autonomes permettant de déployer (organiser) et optimiser des systèmes informatiques [16]. Leurs techniques sont principalement basées sur des politiques à base d'équations mathématiques. Ils proposent dans leurs travaux d'optimiser un système de virtualisation en allouant dynamiquement les ressources du CPU hôte aux différents systèmes invités. Pour ce faire, ils proposent de définir une fonction mathématique qui spécifie la politique d'administration autonome utilisée.

A.4 L'administration autonome de réseau par apprentissage

Michael L. Littman et son équipe de l'Université de Rutgers proposent des systèmes qui sondent eux mêmes leur état et qui s'optimisent de manière continuelle une fois déployés [8]. Ils proposent comme cas d'étude la réparation d'un réseau, l'optimisation d'un réseau et l'amélioration d'un filtre de spam[1]. Ils proposent un environnement où l'administrateur introduit un ensemble de sondes, un ensemble

[1] courrier électronique non désiré

d'action à effectuer, sans spécifier de politique, et un but, ici le temps de reconfiguration qui doit être le plus faible possible. Leur environnement apprend automatiquement les meilleures politiques à appliquer en fonction des informations envoyées par les sondes, des actions à effectuer et du but. Ce projet se situe entre les projets basés sur une boucle *sondes-actuateurs-manager* et les projets à comportement émergent comme celui de O. Kephart. L'administration autonome ne repose pas sur des composants autonomes mais sur des composants normaux dont le manager possède lui un comportement émergent.

A.5 AutoMate

M. Parashar et son équipe de l'Université de Rutgers proposent un système à base de composants autonomes [21]. Comme dans le projet de O. Kephart présenté dans notre état de l'art, la composition de composant (le déploiement), la réparation et l'optimisation se font de manière autonome au niveau du composant. Pour permettre cela, ils proposent d'introduire un système de *clé* qui permet à deux composants de savoir si ils peuvent se lier.

Bibliographie

[1] Amza, C., Cecchet, E., Chanda, A., Cox, A. L., Elnikety, S., Gil, R., Marguerite, J., Rajamani, K., and Zwaenepoel, W. (2002). Specification and implementation of dynamic web site benchmarks. In *EEE 5th Annual Workshop on Workload Characterization (WWC-5). Austin, TX.*

[2] Apache (200x). Apache http server project. http://httpd.apache.org/.

[3] Bouchenak, S., Boyer, F., Krakowiak, S., Hagimont, D., Mos, A., Jean-Bernard, S., de Palma, N., and Quema, V. (2005b). Architecture-based autonomous repair management : An application to j2ee clusters. In *SRDS '05 : Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 13–24, Washington, DC, USA. IEEE Computer Society.

[4] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java. In *Software - Practice and Experience, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems", 36(11-12) :1257-1284.*

[Cecchet et al.] Cecchet, E., Marguerite, J., and Zwaenepoel, W. C-jdbc : Flexible database clustering middleware. In *USENIX Annual Technical Conference, Freenix track. Boston, MA.*

[5] Combemale, B., Broto, L., Tchana, A. B., and Hagimont, D. (2008). Metamodeling Autonomic System Management Policies – Ongoing Works. In *IEEE International Workshop on Model-Driven Development of Autonomic Systems , Turku, Finland, 28/07/08-01/08/08*, http ://www.ieee.org/. IEEE.

[6] DIET (200x). The distributed interactive engineering toolbox. http://graal.ens-lyon.fr/~diet/.

[7] Eddy, C. and Frédéric, D. (2006). Diet : A scalable toolbox to build network enabled servers on the grid. volume 20, pages 335–352.

[8] Fenson, E. and Howard, R. (2004). Reinforcement learning for autonomic network repair. In *ICAC '04 : Proceedings of the First International Conference on Autonomic Computing*, pages 284–285, Washington, DC, USA. IEEE Computer Society.

[9] Flissi, A. and Merle, P. (2006). A generic deployment framework for grid computing and distributed applications. In *On the Move to Meaningful Internet Systems 2006 : CoopIS, DOA, GADA, and ODBASE.*

[10] GO-DIET (200x). An application tool to launch a diet platform. http://graal.ens-lyon.fr/~diet/godiet.html.

[11] Gorlick, M. M. and Razouk, R. R. (1991). Using weaves for software construction and analysis. In *ICSE '91 : Proceedings of the 13th international conference on Software engineering.*

[12] Hanson, J. E., Whalley, I., Chess, D. M., and Kephart, J. O. (2004). An architectural approach to autonomic computing. In *ICAC '04 : Proceedings of the First International Conference on Autonomic Computing*, pages 2–9, Washington, DC, USA. IEEE Computer Society.

[13] Jézéquel, J.-M., Hußmann, H., and Cook, S., editors (2002). *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *Lecture Notes in Computer Science*. Springer. 3540442545.

[14] Kaminski, P., Müller, H., and Litoiu, M. (2006). A design for adaptive web service evolution. In *SEAMS '06 : Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 86–92, New York, NY, USA. ACM.

[15] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. In *IEEE Computer Magazine, 36-1.*

[16] Menasce, D. A. and Bennani, M. N. (2006). Autonomic virtualized environments. In *ICAS '06 : Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 28, Washington, DC, USA. IEEE Computer Society.

[17] Microsystems, S. (200x). Java 2 platform enterprise edition (j2ee). http://java.sun.com/j2ee/.

[18] ModJK (200x). The apache tomcat connector. http://tomcat.apache.org/connectors-doc/.

[19] MySQL (200x). Mysql mysql web site. http://www.mysql.com/.

[20] Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA. IEEE Computer Society.

[21] Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G., and Hariri, S. (2006). Automate : Enabling autonomic applications on the grid. *Cluster Computing*, 9(2) :161–174.

[22] SCA (200x). Sca projet home. http://osoa.org/display/Main/Home.

[23] Scott and Kendal (2002). The unified process explained.

[24] Taylor, R. N., Medvidovic, N., Anderson, K. M., E. James Whitehead, J., and Robbins, J. E. (1995). A component- and message-based architectural style for gui software. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering.*

[25] Tomcat (200x). The apache software foundation apache tomcat. http://tomcat.apache.org/.

[26] Wold, T. D. and Holvoet, T. (2007). Design patterns for decentralised coordination in self-organising emergent systems. In *Engineering Self-Organising Systems*.

[27] Xen (200x). Home of the xen hypervisor. http://xen.org/.

Liste des tableaux

Table des figures