RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Spécification de politiques d'administration autonome avec Tune

Laurent Broto[1], Patricia Stolf[2], Jean-Paul Bahsoun[1], Daniel Hagimont[3], Noel De Palma[4]

[1] Université Paul Sabatier, Toulouse, France
[2] Institut Universitaire de Formation des Maitres, Toulouse, France
[3] Institut National Polytechnique de Toulouse, France
[4] Institut National Polytechnique de Grenoble, France

IRIT - 118, route de Narbonne - 31062 Toulouse Cedex 9

{Laurent.Broto, Patricia.Stolf, Jean-Paul.Bahsoun, Daniel.Hagimont}@irit.fr
Noel.Depalma@inrialpes.fr

Résumé
Les infrastructures logicielles réparties sont de plus en plus complexes et difficiles à administrer, du fait qu'elles intègrent des logiciels très diversifiés ayant des interfaces d'administration très spécifiques. De plus, l'administration étant réalisée par des humains, elle est pénalisée par de nombreuses erreurs de configuration et une réactivité très faible. En réponse à ce problème, nous explorons la conception et l'implantation d'un système d'administration autonome. Les logiciels à administrer sont encapsulés dans des composants afin d'administrer une infrastructure logicielle comme une architecture à composants. Cependant, nous avons observé que les interfaces d'un modèle à composants sont de trop bas niveau d'abstraction et difficiles à utiliser. En conséquence, nous introduisons un formalisme de plus haut niveau permettant la spécification de politiques de déploiement et de gestion autonome. Cet article présente les outils de spécification de politiques qui sont fournis par le système d'administration autonome Tune.

Mots-clés : Administration, systèmes autonomes, architectures logicielles.

---

1. Introduction

Les environnements informatiques d'aujourd'hui sont de plus en plus sophistiqués. Ils intègrent de nombreux logiciels complexes qui coopèrent dans le cadre d'une infrastructure logicielle, potentiellement à grande échelle. Ces logiciels se caractérisent par une grande hétérogénéité, en particulier en ce qui concerne leurs fonctions d'administration qui sont le plus souvent propriétaires.
La conséquence est que l'administration de ces infrastructures logicielles (installation, configuration, réparation, optimisation ...) est une tâche très complexe, source d'erreurs, et consommatrice en ressources humaines.
Une approche très prometteuse consiste à implanter un logiciel d'administration autonome. Ce logiciel fournit un support pour le déploiement et la configuration des applications dans un environnement réparti. Il fournit également 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.
De nombreux travaux se sont appuyés sur un modèle à composants pour concevoir un système autonome [4,5,7]. Le principe général est d'encaspuler les éléments administrés (des logiciels patrimoniaux) dans des composants et d'administrer l'environnement logiciel comme une architecture à composants. L'administrateur bénéficie alors des atouts de ce modèle à composants, l'encapsulation, les outils de déploiement et les interfaces de reconfiguration, afin d'implanter des procédures d'administration autonome.

Dans un précédent projet (Jade [5]), nous avons conçu et implanté un tel système d'administration autonome en s'appuyant sur le modèle à composants Fractal [2]. Dans Jade, un administrateur peut encapsuler des logiciels patrimoniaux dans des composants Fractal, décrire un environnement logiciel à déployer en utilisant l'ADL Fractal (*Architecture Description Language*) et implanter des procédures de reconfiguration (des *gestionnaires autonomes*) en utilisant les interfaces de reconfiguration de Fractal.
Cependant, nos expérimentations avec Jade ont montré que les interfaces d'un modèle à composants (comme Fractal) sont de trop bas niveau et difficiles à utiliser. Pour implanter les composants qui encapsulent les logiciels patrimoniaux, décrire des architectures à déployer et implanter des programmes de reconfiguration autonome, l'administrateur doit maîtriser un nouvel intergiciel, le modèle à composants Fractal dans le cas de Jade.
Tune est une évolution de Jade dont le but est de fournir un formalisme de plus haut niveau pour toutes les précédentes tâches (encapsulation des logiciels, déploiement, reconfiguration). La motivation principale est de cacher les détails du modèle à composants sur lequel on s'appuie et de fournir une interface de spécification beaucoup plus intuitive pour l'encapsulation des logiciels, le déploiement et la reconfiguration.
La suite de l'article est structurée comme suit. La section 2 présente le contexte de nos travaux et nos motivations. La section 3 décrit nos contributions. Après un bref état de l'art (section 4), nous concluons en section 5.

2. Contexte

Dans cette section, nous présentons deux cas d'application que nous utilisons pour illustrer et valider nos contributions. Nous présentons ensuite plus en détail ce en quoi consiste un système d'administration autonome s'appuyant sur un modèle à composants.

2.1. Applications

Notre principale cible applicative est l'administration de serveurs répartis sur une grappe de machines ou une infrastructure de type grille. Nous donnons deux exemples de telles organisations.

2.1.1. Serveur J2EE en grappe

Les spécifications de la plate-forme Java 2 Platform, Enterprise Edition (J2EE) définissent un modèle pour le développement d'applications Web [8] selon une architecture multi-tiers. Ces applications sont alors composées d'un serveur Web (par exemple Apache), un serveur d'application (ex. Tomcat) et un serveur de base de données (ex. MySQL). Une requête HTTP au serveur Web référence soit un contenu statique qui est retourné directement au client par le serveur Web, soit un document qui doit être généré dynamiquement, la requête étant alors transmise au serveur d'application. Quand le serveur d'application reçoit une requête, il déclenche l'exécution de traitements matérialisés par des Servlets ou des EJB, qui peuvent interroger la base de données stockant les données persistentes. Le résultats de ces traitements est la génération d'une page Web qui est retournée au client. Dans ce contexte, la croissance exponentielle du nombre d'usagers de l'Internet a fait naître le besoin de concevoir des applications Web passant à l'échelle et hautement disponibles. Pour passer à l'échelle et assurer la disponibilité, une approche très utilisée consiste à dupliquer les serveurs composant une architecture J2EE sur une grappe de machines.
Dans cette approche (Figure 1), un composant particulier que nous appelons un répartiteur, est installé devant chaque groupe de serveur dupliqué et répartit la charge de travail entre les duplicas. Différents algorithmes de répartition de la charge peuvent être implantés par le répartiteur, par exemple, une répartition aléatoire (*random*) ou tourniquet (*round-robin*).

2.1.2. Ordonnanceur réparti sur une grille

Les grilles de calcul permettent le partage, l'aggrégation et la sélection de ressources, dynamiquement en fonction de leur disponibilité, capacité, coût, et en fonction des besoins des applications. Diet [3] est un exemple d'intergiciel dont le but est de répartir la charge de calcul sur une grille. Diet est construit sur un ensemble d'outils permettant de localiser un serveur de calcul approprié, en fonction de la nature du calcul demandé par le client, de la localisation des données nécessaires au calcul (qui peuvent être le résultat d'un calcul précédent), et de la disponibilité des ressources de calcul dans la grille. Le but de

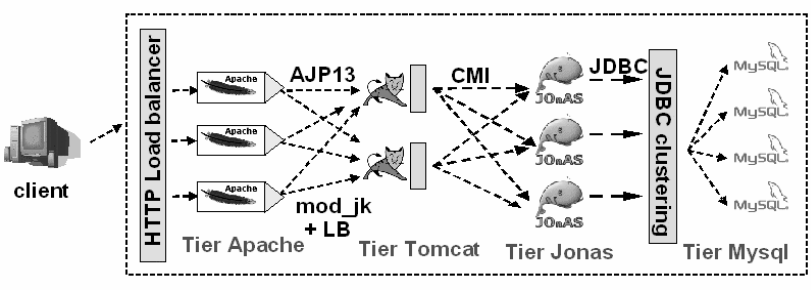

FIG. 1 – Serveur J2EE en grappe

Diet est de fournir un accès transparent à une grande quantité de ressources de calcul dans une grille. Comme illustré sur la Figure 2, Diet repose principalement sur les composants suivants. Un client est une application qui utilise Diet pour résoudre un problème (un calcul). Un MA (*Master Agent*) reçoit une requête de calcul de la part d'un client. Le MA choisit le meilleur serveur pouvant effectuer ce calcul et retourne la référence de ce serveur au client. Le client peut alors envoyer sa requête de calcul au serveur sélectionné pour qu'il effectue le calcul. Les LAs (*Local Agents*) sont chargés de faire remonter l'information de supervision des serveurs de calcul vers le MA. Les LAs ne prennent pas de décisions d'ordonnancement, ils permettent d'éviter la surcharge du MA lorsque l'on gère des insfrastructures de grande taille. Les SeDs (*Server Daemons*) sont les serveurs de calculs ; ils déclarent à leur LA père les types de calculs qu'ils peuvent effectuer et ils fournissent aux clients une interface permettant de soumettre les calculs à effectuer.

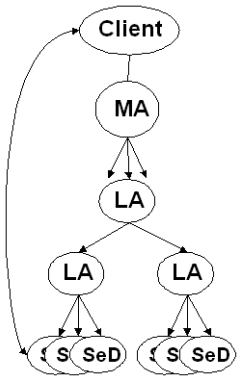

FIG. 2 – L'ordonnanceur réparti Diet

Dans ces deux exemples d'application, le déploiement des serveurs dans une grappe de machines ou une grille est très complexe et demande beaucoup d'expertise. De nombreux fichiers de configuration doivent être édités et configurés de façon cohérente. Egalement, les pannes ou les surcharges (lorsque le degré de duplication est trop faible) doivent être traitées par des administrateurs humains.

2.2. Administration autonome basée-composants

Un système autonome fondé sur un modèle à composants vise à fournir une vision uniforme d'un environnement logiciel composé de différents types de logiciels. Chaque logiciel administré est encapsulé dans un composant et l'environnement logiciel est abstrait sous la forme d'une architecture à composants. Ainsi, le déploiement, la configuration et la reconfiguration de l'environnement logiciel sont réali-

sés en utilisant les outils associés à ce modèle à composants. Cette solution a été retenue dans différents projets de recherche, en particulier les projets Jade et Tune.
Le modèle à composants que nous avons utilisé pour Tune est le modèle à composants Fractal [2]. Un composant Fractal est une entité à l'exécution qui est encapsulée et possède une ou plusieurs interfaces (point d'interaction avec le composant définissant un ensemble de méthodes). Les interfaces peuvent être de deux types : les interfaces serveurs correspondent aux points d'interaction permettant de recevoir des appels de méthodes, les interfaces clientes correspondent aux points d'interaction permettant au composant de réaliser des appels sortants. La signature d'une interface peut être décrite sous la forme d'une interface Java standard. Les composants peuvent être assemblés pour former une architecture à composants, en reliant des interfaces clientes à des interfaces serveurs. Différents types de liaisons existent, en particulier des liaisons locales analogues à des références Java, et des liaisons distantes analogues à des références RMI. Un langage de description d'architecture (ADL) permet de décrire une architecture et un interprète de ce langage permet de déployer une architecture décrite avec l'ADL. Enfin, Fractal fournit une interface de contrôle très complète, permettant d'introspecter (observer) et reconfigurer une architecture déployée.
Tout logiciel géré par Tune est encapsulé dans un composant Fractal (que nous appelons *Wrapper*), qui fournit une interface permettant son administration locale. Ainsi, le modèle à composant Fractal est utilisé pour implanter une couche d'administration (Figure 3) au dessus de la couche patrimoniale (composée des logiciels administrés). Dans cette couche, les composants fournissent une interface d'administration pour les logiciels encapsulés dont le comportement est spécifique au logiciel (défini par le wrapper associé). Cette interface permet ainsi de contrôler l'état du composant de manière homogène, en évitant des interfaces de configuration complexes et propriétaires. Enfin, les interfaces de contrôle de Fractal permettent d'administrer les attributs des composants et de relier ces derniers entre eux.

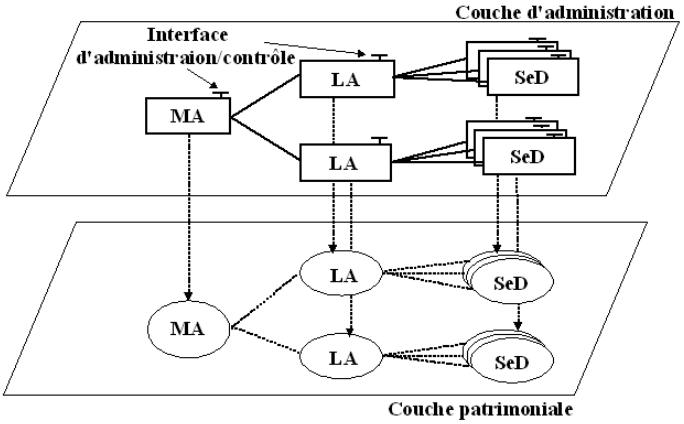

FIG. 3 – Couche d'administration dans Tune

Nous pouvons alors distinguer deux rôles importantes :
– le rôle des interfaces d'administration et de contrôle qui permettent de gérer la configuration et la liaison des composants. Elles incluent des méthodes de navigation dans la couche d'administration à composant mais aussi des méthodes de modification en vue d'implanter la reconfiguration,
– le rôle des wrappers qui permettent de refléter les changements de la couche d'administration au niveau de la couche patrimoniale. L'implantation d'un wrapper peut faire appel au système de navigation de la couche d'administration pour accéder aux attributs des composants et ainsi pouvoir écrire des fichiers de configuration. Par exemple, la configuration du serveur Web Apache nécessite de connaître le nom et l'adresse de la machine exécutant le serveur Tomcat, ce qui nécessite de naviguer dans l'architecture.

2.3. Motivations

L'administration autonome à base de composants s'est affirmée comme étant une approche pertinente. Les expériences que nous avons menées avec Jade [5] pour administrer une grappe J2EE ou une infrastructure Diet sur la grille ont validé ce choix d'implantation. Mais lorsque Jade a été utilisé par des utilisateurs extérieurs à notre groupe de recherche, nous avons remarqué que :

– l'écriture de wrapper est difficile. Le développeur doit avoir une bonne connaissance du modèle à composants utilisé (dans le cas de Jade, Fractal).
– le déploiement n'est pas aisé. Les ADL sont généralement riches et nécessitent eux aussi une connaissance approfondie du modèle à composants sous-jacent. De plus, ces langages ne permettant pas une représentation compacte de l'architecture, les fichiers écrits doivent spécifier tous les composants à déployer ainsi que les noeuds associés. Déployer un millier de composants nécessite une spécification ADL de plusieurs milliers de lignes.
– les services de reconfiguration sont difficiles à implanter car ils doivent utiliser les interfaces de la couche d'administration. Bien qu'elles soient, comme nous l'avons vu, homogènes, elles nécessitent elles aussi une bonne connaissance du modèle à composants.

Ces différentes observations nous ont amené à la conclusion qu'il était nécessaire de fournir un niveau d'interfaçage plus élevé pour décrire l'encapsulation des logiciels patrimoniaux dans les composants, le déploiement de l'application à large échelle ainsi que les règles de reconfiguration.
Nous présentons notre approche dans la section suivante.

3. L'interface d'administration de Tune

Comme mentionné précemment, notre but est de fournir une interface de haut niveau en vue de décrire l'application à encapsuler, déployer et reconfigurer.
Une vue générale de l'interface d'administration de Tune est proposée à la Figure 4.

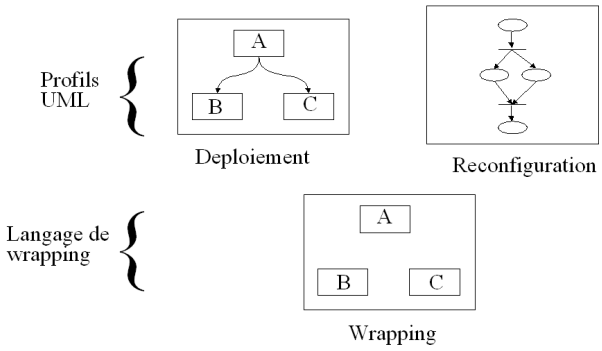

FIG. 4 – L'interface d'administration de Tune

Dans nos expériences précédentes avec le système Jade, nous sommes arrivés à la conclusion que l'interêt d'utiliser Fractal est significatif pour un système d'administration autonome mais que les interfaces fournies par Fractal sont de trop bas niveau. Plus précisément, dans Jade, l'administrateur doit :

– écrire les wrappers avec l'interface de programmation du modèle à composant Fractal. Notre approche pour résoudre ce problème consiste à introduire un Langage de Description de Wrapper (WDL) permettant de décrire le comportement des wrappers. A l'exécution, une description WDL est interprêtée par un wrapper générique (un composant Fractal). Ainsi, l'administrateur n'a plus à écrire de composant Fractal,
– spécifier le déploiement en utilisant le langage de description d'architecture de Fractal. Notre approche introduit un profil UML pour décrire graphiquement le déploiement. Le premier avantage est qu'UML est plus intuitif que le langage de description d'architecture de Fractal et ne nécessite pas de

connaître le modèle à composant sous-jacent. Le deuxième avantage est qu'en utilisant ce schéma de déploiement, l'administrateur peut décrire son application en ïntentionpplutôt qu'en extension comme il devait le faire avec l'ADL Fractal. Ceci est particulièrement intéressant dans le cadre de déploiements importants tels que Diet où des milliers de serveurs doivent être déployés,
– spécifier les règles de reconfiguration qui doivent être écrites en Java et utiliser les API Fractal. Notre approche introduit un profil UML pour spécifier les règles de reconfiguration grâce à un diagramme d'états. Ces diagrammes sont utilisés pour définir le *workflow* des opérations qui doivent être exécutées pour reconfigurer l'environnement administré. Un des avantages principal de cette approche, en plus de sa simplicité, est qu'elle ne peut produire après une reconfiguration qu'une architecture cohérente avec le schéma abstrait spécifié pour le déploiement.

Nous détaillons dans les sections suivantes ces trois aspects.

3.1. Un profil UML pour les schémas de déploiement

Le profil UML que nous avons introduit pour spécifier les diagrammes de dépoiement est illustré dans la Figure 5 où un schéma de déploiement est donné pour l'application Diet.

Un schéma de déploiement décrit l'organisation générale du logiciel à déployer. Lors du déploiement, ce schéma est interprêté pour déployer une architecture à composants.

Chaque élément (boîte) correspond à un logiciel qui peut être instancié en plusieurs réplicas. Un lien entre deux éléments génère une liaison entre les composants instanciés depuis ces deux éléments. Chaque liaison est bidirectionnelle (implantée par deux liaisons en sens opposé) et permet la navigation entre les composants.

Un élément inclut un ensemble d'attributs de configuration pour le logiciel (tous de type chaîne de caractère). La majorité de ces attributs sont propres au logiciel mais quelques-uns sont définis par Tune et sont utilisés lors du déploiement :
– **wrapper** qui donne le nom du fichier contenant la description WDL de ce composant,
– **legacyFile** qui donne le nom du fichier d'archive comprenant les binaires du logiciel à déployer,
– **hostFamily** qui donne une indication concernant l'allocation dynamique de la machine sur laquelle ce logiciel doit être déployé,
– **initial** qui donne le nombre initial d'instance de ce composant à déployer.

Le schéma de la figure 5 décrit l'architecture d'une application Diet où un MA, deux LA et 10 SeDs (5 par LA) doivent être déployés. Une sonde est liée à chaque logiciel pour le monitorer et lever une notification en cas de défaillance de ce dernier.

On peut remarquer qu'une cardinalité est associée à chaque liaison. Soient A(n) et B(m) deux éléments liés où n est la valeur de l'attribut initial du composant A et m celle du composant B. La sémantique est alors la suivante :
– A(n) 1-1 B(m) : chaque composant A doit être lié à un composant B et chaque composant B à un composant A. Ainsi, il doit y avoir autant de composants A que de composants B et donc m=n. Cette cardinalité est par exemple utilisée pour associer une sonde à chaque logiciel.
– A(n) 1-u B(m) : chaque composant A doit être lié à u composants B et chaque composant B doit être lié à un seul composant A. Ainsi, nous devons avoir n*u composants B et m=n*u. Cette cardinalité est par exemple utilisée pour faire des déploiements en arbre, comme pour Diet.
– A(n) t-u B(m) : ce cas est une généralisation des deux cas précédents. Chaque composant A doit être lié à u composants B et chaque composant B doit être lié à t composants A. La cardinalité doit respecter la règle suivante : m=n*u/t avec m>=u et n>=t. Cette cardinalité est utilisée par exemple pour déployer des grappes J2EE (Figure 1).

Le schéma de déploiement présenté dans la Figure 5 déploie l'application Diet illustrée dans la Figure 3.

3.2. Un langage de description de Wrapper

Lors du déploiement, le schéma précédent est parsé et pour chaque élément, un certain nombre de composants Fractal sont créés. Ces composants Fractal implantent les wrappers pour les logiciels patrimoniaux déployés et permettent de les contrôler. Chaque composant créé est une instance d'un wrapper générique qui est un interpréteur de spécifications WDL.

Une spécification WDL définit un ensemble de méthodes qui pourront être appelées lors des opérations de configuration ou de reconfiguration de l'application déployée. L'enchainement des appels à

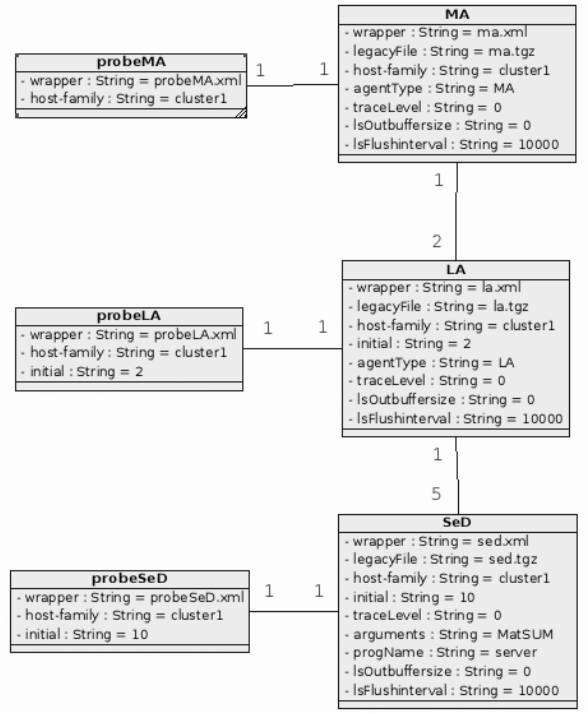

FIG. 5 – Schéma de déploiement pour Diet

ces méthodes lors des opérations de (re)configuration est défini grâce à une interface présentée dans la section 3.3.

Généralement, une spécification WDL définit les méthodes *start* et *stop* pour contrôler l'activité du logiciel déployé ainsi qu'une méthode *configure* pour refléter au niveau patrimonial l'état du niveau d'administration (en se servant des attributs définis dans le diagramme de déploiement). D'autres méthodes spécifiques au logiciel déployé peuvent être définies et doivent être écrites en Java.

Les motivations principales de l'introduction de ce WDL sont :

– masquer la complexité du modèle à composants sous-jacent (dans Tune, Fractal),
– la majorité des besoins des utilisateurs sont couverts par un ensemble réduit de méthodes qui peuvent être réutilisées.

La figure 6 montre un exemple de spécification WDL qui encapsule un serveur de calcul Diet (SeD). On retrouve dans cet exemple la définition des méthodes *start* et *stop* qui sont appelées pour lancer ou arrêter le SeD. Une méthode de configuration est aussi définie, permettant de refléter l'état du composant SeD (ses attributs de configuration) dans le fichier de configuration du SeD.

L'implantation de ces méthodes est commune à la majorité des logiciels patrimoniaux et a été utilisée pour la plupart des logiciels que nous avons encapsulés (LA et MA pour Diet mais aussi, Apache, Tomcat et MySQL pour J2EE).

Une définition de méthode inclut la description des paramètres à passer à la méthode quand cette dernière est appelée. Ces paramètres peuvent être des chaînes de caractères constantes, des valeurs d'attribut ou une combinaison des deux. Tous les attributs définis dans le schéma de déploiement peuvent être utilisés comme paramètre de méthode. Cependant, d'autres attributs auto-gérés par Tune peuvent aussi être utilisés :

– *dirLocal* qui est le répertoire où le logiciel est déployé sur le noeud distant,
– *compName* qui est un nom unique associé à chaque instance de composant,
– *PID* qui est l'identifiant du processus dans la couche patrimoniale.

Dans la figure 6, la méthode *start* prend en paramètre la commande shell permettant de démarrer le

serveur ainsi que les variables d'environnement qui doivent être associées au processus créé :
– *$dirLocal/$progName* est le nom du binaire à exécuter,
– *$dirLocal/$compName-cfg* est le nom du fichier de configuration passé en paramètre au binaire lancé et qui est généré par la méthode *configure*,
– *$arguments* est un paramètre à passer au binaire (propre à Diet),
– *LD_LIBRARY_PATH=$dirLocal* est une variable d'environnement à passer au shell.
La méthode *configure* est implantée par la classe Java *ConfigurePlainText*. Cette méthode génère un fichier de configuration composé de paires < attribut-valeur > :
– *$dirLocal/$compName-cfg* est le nom du fichier de configuration à générer,
– = est le séparateur entre chaque attribut et valeur dans le fichier généré,
– *<param value="A :V"/>* demande l'insertion de la paire dans le fichier de configuration.
Il est intéressant dans certains cas de naviguer dans l'architecture déployée pour configurer le logiciel. Voici deux exemples liés à Diet :
– le fichier de configuration du LA doit contenir le nom du serveur déployé. Ce nom est récupéré dans l'attribut *$compName* du composant LA.
– le fichier de configuration d'un SeD doit indiquer le nom de son LA père. Comme le SeD est relié au LA par une liaison au niveau du diagramme de déploiement (et donc au niveau de la couche d'administration), le nom du LA pourra être récupéré dans l'attribut *$LA.compName*.

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<wrapper name='sed'>
    <method name="start" key="appli.wrapper.util.GenericStart"
                         method="start_with_pid_linux" >
         <param value="$dirLocal/$progName $dirLocal/$compName-cfg $arguments"/>
         <param value="LD_LIBRARY_PATH=$dirLocal"/>
    </method>

    <method name="configure" key="appli.wrapper.util.ConfigurePlainText"
                             method="configure">
         <param value="$dirLocal/$compName-cfg"/>
         <param value=" = "/>
         <param value="traceLevel:$traceLevel" />
         <param value="parentName:$LA.compName"/>
         <param value="name:$compName"/>
         <param value="lsOutbuffersize:$lsOutbuffersize"/>
         <param value="lsFlushinterval:$lsFlushinterval"/>
    </method>

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

FIG. 6 – Une spécification WDL

3.3. Un profil UML pour des procédures de (re)configuration

Les reconfigurations sont déclenchées par des notifications. Une notification peut être générée par un composant spécifique de monitoring (les sondes dans le schéma de déploiement) ou par un logiciel patrimonial qui possèderait ses propres fonctions de monitoring.
A chaque fois qu'un composant wrapper est instancié, un tube de communication (tel que les pipes Unix) est créé. Ce tube peut être utilisé par les sondes ou les logiciels patrimoniaux pour générer des notifications. Ces notifications possèdent une syntaxe particulière permettant le passage de paramètre.

On notera que l'utilisation de tubes permet à n'importe quel logiciel écrit dans n'importe quel langage de générer des notifications.
Une notification envoyée dans le tube associé au wrapper est transmise au noeud d'administration où un programme de reconfiguration peut alors être exécuté (dans notre prototype actuel, le code d'administration qui permet d'initier le déploiement et la reconfiguration est exécuté sur un noeud d'administration alors que le logiciel administré est lui distribué sur des noeuds distants). Une notification est définie par un type, le nom du composant qui l'a générée et un argument (tous de type chaîne de caractère).
Pour définir les réactions aux évènements, nous avons introduit un profil UML qui permet de spécifier les reconfigurations comme des diagrammes d'état. Ces diagrammes définissent les opérations (et leur enchainement) qui doivent être appliquées en réaction à une notification. Une opération dans un diagramme d'état peut être l'affectation d'un ou plusieurs attributs de composants à une valeur, ou l'appel d'une méthode ou d'un ensemble de méthodes sur des composants. Pour désigner le(s) composant(s) sur lequel(lesquels) les opérations doivent être appliquées, la syntaxe des opérations dans le diagramme d'état permet de naviguer dans l'architecture, à l'image de ce que le WDL permet de faire.
Considérons par exemple le diagramme de gauche de la figure 7, qui est la réaction à la notification correspondant à l'arrêt inopiné d'un LA dans Diet. La notification (*fixLA*) est générée par la sonde *probeLA* ; la variable *this* est donc le nom de l'instance de composant *probeLA* (*compName*) qui a levé la notification. Ainsi :
– *this.stop* appelle la méthode *stop* de la sonde pour éviter la génération d'autres notifications.
– *this.LA.start* appelle la méthode *start* sur le LA qui est relié à la sonde qui a levé la notification. C'est la réparation proprement dite du LA.
– *this.LA.SeD.stop* appelle la méthode *stop* des SeDs reliés au LA fautif. Ceci est nécessaire parce que dans Diet, le redémarrage d'un LA nécessite le redémarrage de tous ses SeDs fils pour qu'ils se réinitialisent et se reconnectent à leur LA père. Les sondes des SeDs vont donc détecter la panne des SeD et les redémarrer.
– *this.start* redémarre la sonde liée au LA.
Il est à noter que les opérations des diagrammes d'état utilisent les entités définies dans le schéma de déploiement (LA, SeD ...), mais elles s'appliquent à l'exécution sur les instances de composants déployées.
Nous étendons actuellement Tune pour permettre le redéploiement de logiciels (changement de noeud ou ajout d'instances de composants), tout en garantissant que l'architecture déployée reste conforme au schéma de déploiement initialement décrit.
Un diagramme similaire est utilisé pour démarrer l'architecture Diet, comme illustré Figure 7 (à droite). Dans ce diagramme, lorsqu'une expression commence par le nom d'un composant du schéma de déploiement (MA, LA ...), toutes les instances de ce composant sont considérées. Ainsi, une même opération peut être appliquée à plusieurs composants dans un seul état. Un tel diagramme de démarrage permet de s'assurer que (1) les fichiers de configuration sont générés avant (2) le démarrage des serveurs, qui se fait dans un ordre particulier (ici, MA, LA puis SeD). Chaque sonde est démarrée après le démarrage du serveur qu'elle surveille.

4. Travaux voisins

L'administration autonome est une approche séduisante qui veut simplifier la tâche difficile qu'est l'administration de systèmes en construisant des systèmes auto-réparables, auto-optimisables et auto-configurables [6].
Les solutions d'administration pour les systèmes patrimoniaux sont généralement proposées sous forme ad-hoc et sont liées à une implantation particulière du système (par exemple [9] pour l'auto-optimisation dans des grappes de serveurs Web). Cela réduit malheureusement les possibilités de réutilisation et nécessite de réimplanter les procédures d'auto administration pour chaque contexte particulier. De plus, l'architecture des systèmes administrés est souvent complexe (par exemple les architectures multi-tiers) et demande des connaissances avancées pour leur administration.
L'utilisation d'un modèle à composants pour administrer les architectures patrimoniales a été proposé dans plusieurs projets [1,4,5,7] et a été reconnue comme étant une approche pertinente, mais dans la majorité des cas, les procédures autonomes doivent être programmées en utilisant l'API du modèle à

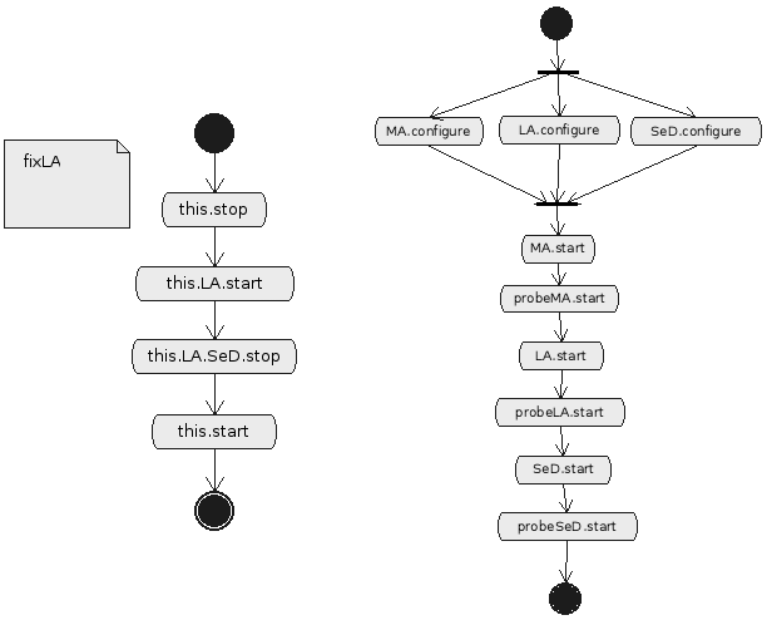

FIG. 7 – Diagrammes d'état pour la réparation et le démarrage

composants sous-jacent qui est de trop bas niveau et source d'erreur.
Dans cet article, nous avons proposé une interface de haut niveau qui est composée :
– d'un langage/framework pour décrire les wrappers,
– d'un profil UML pour spécifier les schémas de déploiement,
– d'un profil UML pour spéficier les reconfigurations comme des diagrammes d'état.
L'interêt principal de cette approche est de fournir une interface de plus haut niveau à l'administrateur du logiciel.

5. Conclusions

Les applications distribuées sont de plus en plus complexes et difficiles à administrer et leur administration coûte énormément de ressources humaines. Pour palier à ce problème, plusieurs projets de recherche proposent d'implanter l'administration comme un logiciel et d'utiliser un modèle à composants pour bénéficier des facilités d'introspection et de reconfiguration inhérentes à ces logiciels.
Bien que l'administration autonome basée sur des architectures à composants s'est avérée très pertinente, nous avons observé que les interfaces d'un modèle à composants sont de trop bas niveau et difficiles à utiliser. Afin d'encapsuler les logiciels patrimoniaux dans des composants, de décrire des architectures à déployer et d'implanter des programmes de reconfiguration, l'administrateur de l'application doit encore apprendre un nouveau système à composants avec ses API complexes ou ses langages spécifiques.
Dans cet article, nous proposons un niveau d'abstraction plus élevé pour décrire l'encapsulation des logiciels dans des composants, le déploiement de l'architecture et les politiques de reconfiguration à appliquer automatiquement. Cette interface d'administration est principalement basée sur des profils UML pour la description des schémas de déploiement et la description des diagrammes d'état pour la reconfiguration. Un outil de description de wrapper est aussi introduit pour masquer les détails du modèle à composants sous-jacent.
Nous étendons actuellement notre prototype pour permettre plusieurs types de reconfiguration (nous nous limitons actuellement à des invocations de méthodes sur les wrappers). Notamment, nous fournissons des diagrammes d'état pour permettre le redéploiement, c'est à dire changer un logiciel de noeud et ajouter des instances de composant) tout en restant conforme au schéma de déploiement spécifié.

Bibliographie

1. Blair (G.) et al. – Reflection, self-awareness and sef-healing in OpenORB. – Proceedings of the 1st Workshop on Self-Healing Systems, WOSS 2002. ACM, 2002.
2. Bruneton (E.) et al. – The Fractal Component Model and its Support in Java. – Software - Practice and Experience (SP&E), special issue on "Experiences with Auto-adaptive and Reconfigurable Systems", 36(11-12) :1257-1284, September 2006.
3. Combes (P.) et al. – A Scalable Approach to Network Enabled Servers. – 7th Asian Computing Science Conference, January 2002.
4. Garlan (D.) et al. – Rainbow : Architecture-based self adaptation with reusable Infrastructure. – IEEE Computer, 37(10), 2004.
5. Hagimont (D.) et al. – Autonomic Management of Clustered Applications. – IEEE International Conference on Cluster Computing, Barcelona September 25th-28th, 2006.
6. Kephart (J.O.) et Chess (D.M.). – The Vision of Autonomic Computing. – IEEE Computer Magazine, 36(1), 2003.
7. Oriezy (P.) et al. – An Architecture-Based Approach to Self-Adaptive Software. – IEEE Intelligent Systems 14(3), 1999.
8. Sun Microsystems. – Java 2 Platform Enterprise Edition (J2EE). – http ://java.sun.com/j2ee/
9. Urgaonkar (B.) et al. – Dynamic Provisiong of Multi-Tier Internet Applications. – 2nd International Conference on Autonomic Computing (ICAC-2005), Seattle, WA, June 2005.

Les travaux présentés dans cet article ont bénéficié du support de l'ANR au travers des projet Selfware (ANR-05-RNTL-01803), Scorware(ANR-06-TLOG-017) et Lego (ANR-CICG05-11).
