

PROJET RNTL ARCAD
D2.2 : validation du mécanisme
d'extension de la plate-forme choisie dans
le sous-projet 2
T0+24 : document interne

Coordonnateur : Denis Caromel
Univ. de Nice Sophia Antipolis – INRIA
2004 Rt. des Lucioles, BP 93
F-06902 Sophia Antipolis Cedex

avec la participation de Mireille Blay-Fornarino, David Emsellem
Daniel Hagimont, Thomas Ledoux, Michel Riveill et J. Vayssiere

Octobre 2002

Résumé

Ce document a pour objectif de valider les mécanismes d'extension de la plate-forme choisie dans le sous-projet 1 en fonction des propriétés non-fonctionnelles définies dans le document D2.1.

1 Introduction

Les propriétés non-fonctionnelles appliquées aux composants, ou les expérimentations menées sont les suivantes :

- migration, communication asynchrone, et synchronisation par futurs,
- interactions (voir Sous-projet 3),
- prise en compte du contexte d'exécution (bande passante, etc.),
- structuration en composants d'un middleware,
- gestion de données dupliquées.

Globalement, nous avons choisi les mécanismes d'extension suivants : approches à base d'interception (objets d'interposition, interactions), approches à base de tissage de code, approches par composants.

Les plate-formes ou frameworks sur lesquels nous avons mené nos expérimentations sont :

- JOnAS (serveur EJB du consortium ObjectWeb),
- ProActive (bibliothèque pour la programmation répartie et mobile, également membre du consortium ObjectWeb)
- Fractal, un modèle générique de composant, et Julia, une implémentation de ce modèle (consortium ObjectWeb),
- Interactions, bibliothèque pour la programmation des coordinations entre objets,
- OpenCCM, implémentation ObjectWeb du modèle de composant Corba.

Nous allons maintenant détailler les différentes expérimentations réalisées.

2 JOnAS – ProActive

2.1 Objectifs et cadre

Le travail proposé consiste à utiliser ProActive [1, 2, 3] pour ajouter aux EJBs les aspects communication asynchrone et mobilité [4].

Les études et réalisations ont été faites dans le cadre de JOnAS (versions de l'année 2001), qui implémentaient alors la norme EJB 1.1.

Nous avons étudié la possibilité de rendre actifs et migrables les beans (session et entity) ainsi que l'EJB Object. En effet, ce sont les éléments principaux mis en jeu lors de l'utilisation des beans. Nous allons voir comment nous avons réalisé ces modifications, et quelles en sont les conséquences sur JOnAS.

2.2 Réalisation

Voici les modifications apportées à JOnAS afin d'ajouter cette fonctionnalité pour les *sessions beans*.

Propriété *migrable* dans le fichier de déploiement `jonas-ejbjar.xml`

Il fallait tout d'abord que le déployeur de bean puisse spécifier si oui ou non les instances des beans pouvaient être migrables ou non. Pour cela, il fallait rajouter la propriété *migrable* dans un des fichiers de déploiement. Selon la norme des EJB, le fichier de `ejb-jar.xml` est standardisé. Pour rester dans la norme, nous ne pouvions donc pas le modifier. Par conséquent, nous avons ajouté un tag spécifique dans le fichier `jonas-ejb-jar.xml` (voir la figure 1).

```

<jonas-ejb-jar>
  <jonas-session>
    <ejb-name>Op</ejb-name>
    <jndi-name>OpHome</jndi-name>
    <migrable value="true">true</migrable>
  </jonas-session>
</jonas-ejb-jar>

```

FIG. 1 – Exemple de fichier de déploiement d'un bean migrable

Modification de GenIC La propriété *migrable* doit maintenant être prise en compte par JOnAS.

GenIC est un outil fourni avec JOnAS. Il analyse les fichiers XML et les classes Java fournies par le développeur permettant de créer les classes d'interposition. Nous devons donc prendre en compte le nouveau tag *migrable* et adapter les classes générées.

Il faut notamment pouvoir connaître la propriété de migration au moment de la création du bean, d'où la nécessité de rajouter la méthode `isBeanMigrable()` dans le Home correspondant au bean. Le choix du Home s'impose ici par le fait qu'il se comporte comme la fabrique, et que par conséquent, c'est lui qui possède les informations relatives au bean qu'il instancie. Maintenant que nous avons toutes les informations requises pour la migration, nous pouvons créer le bean comme une instance d'objet actif.

Modification du processus de création d'un bean Le bean session est créé lors de l'instanciation du contexte. Notre philosophie étant de rendre transparent pour JOnAS la gestion d'un bean actif ou non, nous avons décidé de rendre actif le bean au moment même de sa création. La figure 2 présente cette création, et en particulier l'appel à l'API `ProActive` lorsque le bean est détecté comme devant être migrable.

De cette manière, tous les appels vers le bean deviennent des appels vers le bean rendu actif, passant par le mécanisme mis en oeuvre par `ProActive`. Cela permet d'ajouter à un session bean JOnAS la propriété non-fonctionnelle de migration et d'asynchronisme.

```

SessionBean sb = (SessionBean)c.newInstance();
//ProActive Migration Begin
if (home.isBeanMigrable()) {
    //Give the bean migrable properties
    sb = (SessionBean)
        ProActive.turnActive(sb,EJBServiceImpl.getNode());
}
//ProActive Migration End

```

FIG. 2 – Création d'un bean session

2.3 Apports conceptuels

La migration et l'asynchronisme des beans sessions fonctionnent correctement, si on respecte quelques contraintes.

En effet, de nombreuses classes référencées par le bean ne sont pas sérialisables, ce qui n'est pas concevable avec la migration. Nous avons donc du nous "séparer" de certaines d'entre elles (suppression de références vers des objets non sérialisables), ce qui n'a pas eu d'incidence notable sur le fonctionnement du bean proprement dit. De la même manière, nous n'avons pas pu rendre totalement transparent l'ajout de cette fonctionnalité pour le développeur de beans : il doit encore étendre l'interface `moveable`, et par là même, définir la méthode `migrateTo()`. Mais cette contrainte a depuis disparu avec les nouvelles versions de ProActive.

Nous avons ainsi réussi à faire migrer un bean `Session Stateful Synchronized` (l'exemple fourni avec JOnAS), qui correspond au bean session le plus complexe à mettre en oeuvre. Cela confirme la validité de cette approche.

2.4 Limites et perspectives

Pour profiter de l'asynchronisme des beans au niveau du client, il est également nécessaire de rendre l'EJBObject actif. En effet, selon le principe des EJB, le client n'a pas la main directement sur le bean, mais sur un objet RMI qui va déléguer au bean (avec des traitements avant et après la délégation).

De la même manière que pour le session bean, il est nécessaire de modifier le générateur de classes d'interposition (GenIC), et la génération de l'objet `home`. Cependant, l'architecture de JOnAS à l'époque de nos tests ne permet pas d'utiliser nos modifications directement. En effet, l'implémentation de l'objet Remote et le stub RMI associé ne sont pas directement compatibles entre eux.

Pour régler ce problème, une autre solution, plus radicale, serait de complètement cacher RMI en utilisant que des objets actifs dans JOnAS. Concrètement, cette opération serait rendue possible en n'utilisant plus la fonctionnalité RMI `exportObject`, qui sert à instancier le stub et le skeleton prenant en charge la communication réseau, ainsi que la création des différents threads de l'objet. A la place, on pourrait utiliser *turnActive* de ProActive qui rend les mêmes services mais met en place en plus les différents mécanismes nécessaires au bon fonctionnement de la migration et de l'asynchronisme.

Les modifications nécessaires à la migration/asynchronisme d'un *bean entity* sont sensiblement les mêmes que pour un bean session (modification et prise en compte du fichier de déploiement, modification de la création du bean). Cependant, il faut dans ce cas respecter la notion de transaction pour rester dans le modèle EJB. Cette notion est pour l'instant incompatible, ou tout du moins très difficile à mettre en oeuvre, avec la notion d'asynchronisme de ProActive. La prise en compte de transactions JOTM d'ObjectWeb dans le cadre de ProActive devrait rendre possible cette réalisation.

3 JOnAS – Interactions

3.1 Objectifs

L'objectif de notre travail est de proposer un modèle de composants interagissants [5] qui facilite l'adaptation et l'assemblage de composants (métiers et techniques). Dans ce cadre nous proposons un service d'interactions qui permet de définir et d'utiliser dynamiquement des interactions pour assembler des composants. En terme d'implémentation, nous avons étendu le modèle de composants de JOnAS (EJB) pour intégrer le support des interactions et fournir le service JOnAS correspondant au serveur d'interactions.

3.2 Cadre

Le modèle proposé est le résultat d'une réflexion générale sur l'assemblage et l'adaptation de composants. Dans le cadre du projet RNTL ARCAD un effort significatif a été fourni pour proposer un environnement de programmation (Noah) s'intégrant à la plate-forme JOnAS pour expérimenter, valider et évaluer le modèle proposé.

3.3 Réalisation

Interaction logicielle et composants EJB Une présentation de la solution proposée à base d'interaction logicielle est disponible dans les livrables ARCAD : D3.1 : document "spécification d'un modèle d'interaction" (juin 2002) ; D3.2 : spécification de mise en oeuvre (septembre 2002) ; D3.3 : démonstration de test mettant en évidence différents protocoles d'interaction (septembre 2002). Dans ce livrable nous nous attachons à décrire les différentes étapes de l'implémentation du service d'interactions dans la plateforme JOnAS.

Pose et destruction d'interactions En cours d'exécution, le programmeur peut lier ou délier les instances de composants entre eux en utilisant les schémas qui sont enregistrés auprès du serveur. Un schéma décrit une règle de réécriture qui permet de réécrire un message notifiant comme une suite d'action. Pour lier différents composants, le programmeur s'adresse au serveur d'interactions qui mémorise la nouvelle interaction, renvoie son nom et demande à chacun des composants liés par l'interaction et définissant des messages notifiant de fusionner les nouvelles règles qui le concernent. La prise en compte de ces modifications a lieu à compter du prochain appel.

Le retrait d'interactions consiste à demander au serveur d'interactions la destruction d'une interaction. Chacun des composants déliés comportant un message notifiant est prévenu afin de retirer la ou les règle(s) qui le concerne(nt) et de recalculer les règles résultantes.

Actuellement dans nos mises en oeuvre chaque composant intègre la pose d'interactions "dès qu'il le peut", c'est-à-dire dès que le message de pose d'interaction lui parvient. Comme les composants sur lesquels nous travaillons sont jusqu'à présent synchronisés, la pose ne peut donc pas intervenir en cours d'exécution d'une opération du composant. Ainsi nous n'avons pas statué dans le cas général sur les conditions préalables à la pose des interactions.

Rendre un composant interagissant Seules les instances de composants "interagissants" peuvent être liées par des interactions comportant des règles où leurs messages apparaissent comme notifiants. Tout composant peut cependant apparaître dans la partie action d'une règle.

- Un composant interagissant doit pouvoir remplir les tâches suivantes :
- fusionner ou (dé-fusionner) dynamiquement des règles d'interactions,
 - donner la main à un contrôle local lors de la réception de messages "notifiants",

- adresser des messages directement aux composants liés sans passer par le serveur d'interactions.

Exécution Lorsqu'un composant interagissant reçoit un message qui se révèle être notifiant, la règle d'interaction qui lui est associée, est évaluée localement. Les appels entre les composants lors de cette évaluation sont alors directs et ne passent pas par le serveur d'interactions.

Navigation et Analyse du graphe d'interactions Via le serveur d'interactions, et les objets qu'il mémorise, il est possible de naviguer dans le graphe d'interactions et de l'analyser pour déterminer des propriétés telles que l'absence de cycle ou de points de non-déterminisme. L'analyse du graphe est basée sur une représentation XML du graphe d'interactions. Le dépliage de ce graphe nous permet d'obtenir le graphe de propagation correspondant à un modèle de message donné (destinataire et signature de l'appel donnés, paramètres non instanciés). La construction et l'analyse de ce dernier graphe permettent de détecter des cycles, des attentes cycliques et des points de non-déterminisme (une même instance de composant peut recevoir plusieurs messages dans un ordre indéterminé). Ce travail est en cours et ne sera pas discuté davantage dans ce livrable, il rejoint les travaux menés sur les vérifications et simulation d'architectures logicielles [6].

Mise en oeuvre Il existe à ce jour, une mise en oeuvre du service d'interactions, qui permet de faire interagir des composants java locaux, RMI et/ou composants EJBs. Le service d'interactions implémente le modèle décrit précédemment. Il comprend principalement :

- des entités interagissantes sur lesquelles des interactions peuvent être posées ou retirées ;
- un serveur d'interactions qui permet d'enregistrer les schémas d'interactions et de poser et détruire des interactions.

La mise en oeuvre du serveur d'interactions ne présentant pas de problème particulier, nous présenterons essentiellement les aspects liés aux entités interagissantes et à l'environnement d'exécution.

Entités interagissantes Seules les instances de composants " interagissants " peuvent être liées par des interactions comportant des règles où leurs messages apparaissent comme notifiants. Tout composant peut cependant apparaître dans la partie action d'une règle. Les composants doivent donc

subir des transformations de code pour supporter les interactions, en particulier le contrôle de la réception de message et l'ajout des fonctionnalités d'attachement et de détachement des règles d'interactions. Pour rendre une classe Java interagissante, il suffit d'appliquer l'outil fourni avec le service d'interaction, " GENINT ", qui modifie directement son byte-code. Le byte-code de l'objet est transformé par méta programmation en utilisant BCEL [7]; un mécanisme de capture de message (wrapper) est mis en place pour déléguer quand cela est nécessaire l'exécution du message à l'interprétation d'un arbre de règles [8, 9].

Pour rendre un composant EJB interagissant, l'utilisateur doit simplement le spécifier dans le fichier de déploiement dont la grammaire XML a été étendue. Pour JOnAS [OBJ], le développeur insère la ligne suivante : `<jonas-interaction value="true"></jonas-interaction>`

Dans le cas d'un composant EJB, les interactions sont prises en charge par les objets d'interposition générés, au même titre que les autres services standards. Dans JOnAS, ils sont obtenus par l'outil de génération de code GenIC que nous avons modifié pour qu'il appelle l'utilitaire " GENINT " si le composant doit être interagissant. Le code généré dépend des informations décrites dans le fichier de déploiement. Le programmeur écrit donc ses beans indifféremment du fait qu'ils supportent ou non les interactions. L'intégration aurait également pu se faire par modification du bean au moyen de la méta programmation au détriment de la composition du service d'interactions avec les autres services standards puisque la logique de composition de services est située au niveau des objets d'interposition JOnAS.

Surcoût d'une réception de message dans le modèle EJB.

Pour tous les composants interagissants, le surcoût lié au mécanisme d'interaction, dans le cas d'une interaction vide est celui d'un test. Dans le cas où l'interaction est non vide, le message lui-même est exécuté en utilisant la méthode d'invocation dynamique de l'API de réflexion de Java.

Interopérabilité et interactions Un des défis auquel nous avons été confronté était de permettre à des composants situés sur des plates-formes différentes d'interagir. Les objets interagissants peuvent être des objets locaux, des objets RMI, des beans. Il s'agit de pouvoir les manipuler de la même façon tant au niveau du serveur que dans les communications directes inter-composants. Pour cela, il faut donc à la fois les désigner de façon uniforme et également minimiser les différences entre les différentes implémentations. En effet, le parsing, la gestion de l'arbre, la fusion sont autant d'éléments communs à toutes les versions réalisées jusqu'à présent. Pour cela, nous avons

encapsulé les références dans des objets " interactingObject " qui gèrent les envois de message et présentent une même interface. Ces manipulations sont bien sûr transparentes du point de vue de l'utilisateur. Une modification du protocole de communication pour passer à RMI-IIOP devrait nous permettre de proposer l'interopérabilité avec les objets C++ que nous avons initialement dans le monde CORBA [8].

Environnement de programmation Outre les outils de génération de code présentés au paragraphe précédent, plusieurs outils et interfaces graphiques ont été définis pour aider le programmeur à définir des schémas d'interactions et à monitorer son application. Ainsi la pose d'interactions peut aussi être effectuée par une interface graphique. A partir de la liste des schémas d'interactions enregistrés, l'utilisateur sélectionne le schéma d'interaction à instancier (security par exemple). Une fenêtre rappelle alors le type des composants à connecter (Object et SecurityService) et les instances de composants interagissant existants de ce type afin de poser une interaction. L'utilisateur peut également visualiser de façon progressive le réseau d'interactions ou visualiser le graphe de propagation pour un message donné. Afin de faciliter la tâche du programmeur, il est également possible de visualiser la règle de réécriture associée à un composant résultat de l'ensemble des règles qui le lie. D'autres outils plus à destination des chercheurs autour de la composition sont également en cours de développement pour étudier le résultat des fusions et aider à valider la commutativité et l'associativité lors de l'introduction de nouveaux opérateurs.

3.4 Apports conceptuels

L'apport conceptuel de RAINBOW se situe essentiellement sur l'expression d'interactions logicielles permettant de décrire comment les composants " interagissent " entre eux et ce de façon dynamique sans intervenir directement dans les composants.

Interaction logicielle et Service d'interactions Pour autoriser une adaptabilité dynamique des applications, nous proposons de considérer les interactions comme des entités de première classe. Cela nous conduit à définir un modèle d'interaction qui présente les propriétés suivantes.

- Un schéma d'interaction spécifie les dépendances comportementales entre les composants qui se trouveront liés ; les interactions instances de ce schéma maintiennent la cohérence des composants liés,

- la définition d'un schéma d'interaction ne dépend pas des implémentations ; une interaction peut lier des objets ou composants définis sur différentes plates-formes,
- un schéma d'interaction est décrit en se basant sur l'interface des composants, il peut être utilisé pour lier un ensemble d'instances de composants,
- un schéma d'interaction et une interaction peuvent être créés et détruits en cours d'exécution,
- une interaction ne peut pas contrôler des propriétés n'appartenant pas à l'interface du composant ; l'interface d'un composant lié par des interactions n'est pas modifiée même si son comportement est modifié par l'interaction ,
- la gestion des interactions sur un composant est basée sur une composition qui respecte les propriétés de commutativité et d'associativité.

Afin de supporter ces différentes propriétés, nous avons défini un service d'interactions qui présente les composantes suivantes :

- un serveur d'interactions qui permet de définir dynamiquement de nouveaux "schémas d'interactions " qui constituent ainsi une bibliothèque, de lier et de délier des composants par la création et la destruction d'interactions, de naviguer dans le graphe d'interactions,
- des composants " interagissants " ; c'est-à-dire des objets ou composants qui ont été préparés à interagir ; leur comportement peut être modifié dynamiquement.

Une présentation plus détaillée des interactions est disponible dans les livrables ARCAD : D3.1 : document "spécification d'un modèle d'interaction"

3.5 Limites et perspectives

Le modèle proposé permet l'assemblage dynamique de composants à l'aide de règles de réécritures qui peuvent être manipulées comme des objets de première classe. Nous avons utilisé cette approche dans plusieurs occasions : assemblage de bases de connaissances, assemblage de composants métiers, adaptation d'une application en fonction de l'environnement.

Les perspectives en ce qui concerne la diffusion de Noah sont essentiellement des perspectives de consolidation et d'évolution de l'environnement logiciel fourni à partir du retour des utilisateurs. En terme de modélisation de la solution actuelle, nous travaillons actuellement sur la sûreté du service d'interactions proposés (accepter ou refuser des poses d'interactions selon l'utilisateur, etc.)

4 EJB adaptables dans JOnAS

4.1 Objectifs et cadre

La prise en compte des soucis de performance et de qualité de service pour la construction des applications réparties large échelle constitue l'un des principaux enjeux des plates-formes middleware à composants. On est en droit d'attendre qu'une application répartie ait conscience de son environnement d'exécution, de ses fluctuations (bande passante réseau, capacités mémoire, capacités de communication, etc.) pour finalement s'adapter dynamiquement aux évolutions de cet environnement et ainsi fournir la meilleure qualité de service possible.

Dans le cadre du projet ARCAD, notre équipe s'intéresse à la construction des architectures adaptables et plus particulièrement à la mise en oeuvre d'une infrastructure pour middleware capable d'adapter dynamiquement une application à son contexte d'exécution. Dans un premier temps, nos travaux ont permis de faire la proposition d'une telle infrastructure dans le cadre d'un système réflexif en Java [10]. Plus récemment, nous nous sommes intéressés à l'utilisation et à la transposition de cette infrastructure dans un contexte EJB dans le cadre de la plate-forme JOnAS [11].

4.2 Réalisation

Motivation. L'un des intérêts de la plate-forme EJB est qu'elle supporte le paradigme de séparation des préoccupations (*separation of concerns*). En effet, grâce à un descripteur de déploiement qui va déclarer les besoins en services middleware et la présence du container qui va injecter le code correspondant, le développement EJB permet de séparer le code métier du code des services non fonctionnels (persistance, transactions, sécurité).

Cependant, la configuration entre les composants EJB et les services middleware est seulement supportée au déploiement (via le descripteur de déploiement) et n'adresse donc pas l'évolution du contexte d'exécution. Notre but est donc de transposer notre infrastructure adaptable aux EJB pour reconfigurer dynamiquement les associations composants EJB - services middleware.

Description de l'infrastructure adaptable initiale. L'objectif de notre infrastructure est d'adresser la reconfiguration dynamique des connexions entre composants métiers et services non fonctionnels dans des environnements d'exécution changeants, non prévisibles a priori [10]. Cette reconfigu-

ration est effectuée par un *moteur d'adaptation* lorsqu'il détecte - via un *framework d'observation* - des changements significatifs dans l'environnement d'exécution. Des stratégies d'adaptation, qui expriment quand et comment une application particulière doit être adaptée (i.e. modification des associations composants fonctionnels - services non fonctionnels), sont externalisées et exprimées dans un (ou plusieurs) langages spécialement conçu(s). Ces *politiques d'adaptation* sont ensuite interprétées dynamiquement par le moteur d'adaptation. Enfin, précisons que pour notifier le moteur d'adaptation de tout changement, le framework d'observation est à la fois à l'écoute d'événements exogènes via des sondes logicielles (mesure de l'utilisation CPU, de la bande passante réseau, etc.) et d'événements endogènes (introspection de valeurs de l'application elle-même).

Il existe actuellement deux types de politiques d'adaptation :

- **les politiques systèmes.** Elles consistent en un ensemble de règles de la forme *condition* -> *action* où la condition est liée au contexte d'exécution et l'action réside en l'attachement/détachement d'un service non fonctionnel. Ces politiques sont relativement indépendantes des applications métiers.
- **les politiques applicatives** Elles définissent des groupes de composants métiers selon leurs caractéristiques run-time et lient les politiques systèmes à ces groupes.

Les deux types de politiques d'adaptation¹ sont présentés dans un exemple ci-dessous.

```
<system-policy name="tracer">
  <rule>
    <when>
      <less-than>
        <property-value name="/system/network.bandwidth"/>
        <number value="40000"/>
      </less-than>
    </when>
    <ensure>
      <attached service="logservice" role="main"/>
    </ensure>
  </rule>
</system-policy>

<application-policy>
  <group name="logged-components">
```

¹écrites en XML dans la première version de l'infrastructure, en Scheme depuis.

```

<select from="all">
  <or>
    <equals>
      <property-value name="className"/>
      <string value="Account"/>
    </equals>
    <equals>
      <property-value name="className"/>
      <string value="AccountWithInterests"/>
    </equals>
  </or>
</select>
<bind policy="tracer"/>
</group>
</application-policy>

```

La politique système "tracer" consiste en une règle qui assure l'attachement d'un service "logservice" - à un groupe spécifique décrit dans la politique applicative - si et seulement si la bande passante du réseau est inférieure à 40000 bps. Cette politique peut être réutilisée dans différentes applications. La politique applicative définit un groupe "logged-components" qui contient toutes les instances des classes `Account` et `AccountWithInterests`, et lie ce groupe à la politique système "tracer".

Dans notre infrastructure initiale, la séparation des préoccupations est réalisée par le MOP RAM [12]. Chaque métaobjet implémente un service non fonctionnel particulier alors que les objets de base implémentent les composants métiers. Le couplage lâche entre le framework d'observation, le moteur d'adaptation, les composants métiers et les services non fonctionnels permet d'envisager de réutiliser notre moteur avec un MOP différent de RAM ou encore avec un serveur EJB puisque ce dernier supporte le paradigme de séparation des préoccupations.

Application à JOnAS (1/2) : vers un serveur EJB adaptable. Dans l'architecture des EJB, un container EJB joue le rôle d'un objet d'interposition entre les composants métiers et les services middleware. Il est en charge d'injecter de façon transparente le code des services non fonctionnels (persistance, transactions, sécurité) déclarés par le descripteur de déploiement. Aussi, il nous a semblé naturel d'étendre le serveur EJB au niveau du container.

Cependant, les containers sont générés statiquement en utilisant les informations fournies par les descripteurs de déploiement, ce qui veut dire qu'il est impossible de modifier dynamiquement les associations composants EJB

- services middleware sans modifier les containers eux-mêmes. Grâce à l'outil GenIC (*Generate Interposition Classes*), proposé en open source par JOnAS et permettant la génération du code du container, nous avons pu étendre l'architecture des EJB.

Dans le but de respecter le plus possible la spécification des containers EJB, nous avons modifié GenIC pour créer une indirection à partir de l'*EJB Object* (réification du container) vers un autre objet appelé *DynamicComposite*. DynamicComposite est responsable de la composition dynamique des services middleware. Ainsi, un EJB Object délègue les messages reçus à un objet DynamicComposite qui va attacher/détacher dynamiquement (cf. ci-après) les services middleware associés aux composants EJB avant ou après l'envoi des messages à ceux-ci.

Enfin, d'un point de vue performance, pour ne pas pénaliser les composants EJB qui n'ont pas besoin d'adaptation, nous avons introduit un nouveau *tag* dans le fichier `jonas-ejb-jar.xml`. Rappelons que ce dernier est lu par GenIC pour générer le code du container correspondant. Grâce au tag `<DynamicComposite>true or false</DynamicComposite>`, le développeur peut identifier explicitement les composants EJB qui doivent être adaptables ou non.

Application à JOnAS (2/2) : un service d'adaptation dynamique.

L'indirection proposée par DynamicComposite n'est qu'une première étape : il faut maintenant intégrer les composants réutilisables de notre infrastructure adaptable à savoir le framework d'observation et le moteur d'adaptation², puis relier l'ensemble.

JOnAS est défini de façon relativement modulaire pour autoriser l'ajout d'un nouveau service en plus de ceux prévus par la spécification. Aussi, nous avons créé un nouveau service appelé *DynamicAdaptation* qui joue le rôle d'un méta-service en encapsulant le framework d'observation et le moteur d'adaptation. Au démarrage du serveur JOnAS, tous les services sont initialisés. L'initialisation de DynamicAdaptation consiste en charger les politiques d'adaptation et démarrer le moteur d'adaptation.

Enfin, l'utilisation du pattern *Adapter* nous a permis de relier le service d'adaptation, DynamicComposite et les services qu'il compose. Pour cela, nous avons encore transformé GenIC pour modifier les constructeurs des containers EJB de façon à instancier un objet DynamicComposite et l'enregistrer dans le moteur d'adaptation via le service DynamicAdaptation.

²les métaobjets RAM et les objets Java étant remplacés respectivement par les services non fonctionnels et les composants EJB.

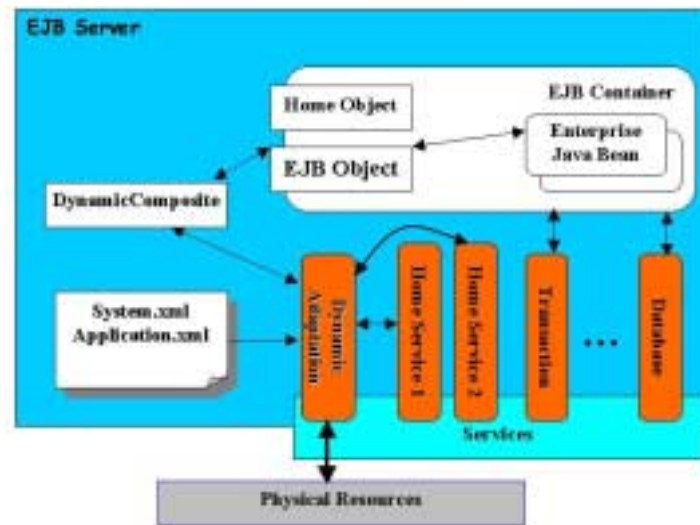


FIG. 3 – Architecture EJB adaptable

Utilisation. Du point de vue de l'utilisateur, le prototype s'utilise exactement comme dans un développement EJB normal avec le serveur JOnAS : il développe ses EJB, écrit ses descripteurs de déploiement. Si il désire avoir des capacités d'adaptation dynamique sur certains beans, il ajoute simplement le *tag* `<DynamicComposite>` dans `jonas-ejb-jar.xml` et utilise comme d'habitude l'outil GenIC. A ce moment là, même si une politique d'adaptation n'est pas définie, le programme s'exécutera normalement (avec des performances moins bonnes dues à l'indirection). Il ne reste plus qu'à écrire ou à réutiliser des politiques d'adaptation !

4.3 Apports conceptuels

Ce travail nous a permis "d'ouvrir" le serveur d'EJB JOnAS pour reconfigurer dynamiquement les connexions entre composants métiers et services non fonctionnels, en appliquant des politiques d'adaptation elles-mêmes reconfigurables. L'infrastructure proposée permet aux applications EJB d'avoir conscience des variations du contexte d'exécution et de s'adapter à tout changement significatif pendant l'exécution.

En conclusion, l'architecture proposée permet de prendre en compte les soucis de performance et de qualité de service pour la construction des applications réparties large échelle.

4.4 Limites et perspectives

Dans le but de valider rapidement notre prototype, nous avons utilisé nos propres services non fonctionnels (par exemple gestion de log) plutôt que les services JOnAS prédéfinis (persistance, transactions, sécurité). Pour aller plus loin dans la validation de la transposition de notre infrastructure dans un contexte EJB, il faudrait remplacer nos services "maison" par les services EJB prédéfinis.

Cependant, est-ce possible d'aller plus loin sans "casser" le modèle et l'architecture des EJB ? En effet, les services EJB ne sont pas assez modulaires pour être (re)composer aisément dans un contexte d'adaptation. Ensuite, différents types de beans sont définis par la spécification des EJB (*entity vs session*) et correspondent à des choix prédéfinis, ad-hoc de la configuration des propriétés non fonctionnelles (persistance). Ainsi, au moment du codage des objets métiers, le développeur fait le choix de façon irréversible sur l'utilisation ou non d'un service non fonctionnel. En conclusion, il semble difficile d'aller plus loin sans "casser" le modèle des EJB.

5 Jonathan – Fractal

5.1 Objectifs et Cadre

L'objectif de cette expérimentation est tout d'abord, utilisant le modèle Fractal [13, 14], de rendre encore plus flexible et configurable l'ORB Jonathan [15]. On va chercher par exemple à rendre adaptables les attributs et les liaisons des "composants" préexistants de l'ORB.

Mais ces travaux réalisent également une opération quelque peu inverse : étendre la plate-forme Julia, qui est l'implémentation de référence du modèle Fractal, en vue de gérer via l'ORB Jonathan les composants répartis prévus par le modèle.

5.2 Réalisation

Les résultats sont peu nombreux pour le moment, étant donné que cette expérimentation vient de commencer. En fait nous avons juste commencé à "fractaliser" Jonathan, dans le but de pouvoir réifier les liaisons réparties sous forme de composants composites Fractal. Pour cela, nous avons commencé à convertir les "composants" de Jonathan en composants Fractal, à dégager une architecture hiérarchique pour ces composants, et à diminuer la granularité de ces composants (le but étant d'atteindre le niveau des objets session, i.e., des éléments de pile de protocoles). En pratique, ce travail

a principalement consisté à convertir les "composants" Jonathan, du type suivant :

```
public class IImpl implements I {
    private J j;
    public IImpl (Object[] usedComponents) {
        j = (J)usedComponents[0];
    }
    // ...
}
public class IImplFactory extends GenericFactory {
    Object[] getUsedComponents (Context ctxt) {
        return new Object[] { ctxt.getValue("j") };
    }
    Object newInstance(Context ctxt, Object[] usedComponents) {
        return new IImpl(usedComponents);
    }
}
```

en composants Fractal reconfigurables du type :

```
public class IImpl implements I, UserBindingController {
    private J j;
    public IImpl () { }
    public void getFcBindings (String itfName) {
        if (itfName.equals("j")) return j;
    }
    public void addFcBinding (String itfName, Object o) {
        if (itfName.equals("j")) j = (J)o;
    }
    public void removeFcBinding (String itfName, Object o) {
        if (itfName.equals("j")) j = null;
    }
    // ...
}
```

5.3 Apports

La "fractalisation" de Jonathan a permis de mettre en évidence l'architecture hiérarchique de ce logiciel. Elle a permis également de rendre reconfigurables les attributs et les liaisons des "composants" de Jonathan (par exemple pour changer dynamiquement la taille d'un cache, ou pour remplacer complètement un composant de gestion de cache par un autre).

5.4 Limites et perspectives

Le travail qui reste à faire consiste à poursuivre la fractalisation de Jonathan, à un grain plus fin. Il faudra alors voir si on peut effectivement réifier les liaisons réparties sous forme de composant composites reconfigurables, à quel coût, et voir si c'est utile. Il faudra ensuite réaliser l'intégration avec Julia et, auparavant, résoudre un certain nombre de problèmes plus théoriques liés à la répartition (concernant notamment le canevas de nommage et de liaison, le passage de références réparties, ...).

6 OpenCCM – Réplication

6.1 Objectifs

La structuration des applications réparties sous la forme d'architectures à composants offre de nombreux avantages. Elle permet la réutilisation de composants logiciels et elle encourage la séparation (comme dans la programmation par aspects) entre le code métier des composants et le code de gestion des services systèmes utilisés.

Un de ces aspects, qui est particulièrement important dans le contexte des applications réparties, est la *duplication de composants*. La duplication de composants peut être utilisée dans différents buts : la gestion de cache pour les performances, la tolérance aux pannes ou la gestion d'utilisateurs mobiles.

Notre objectif est de montrer que la duplication de composants peut être gérée comme une propriété non-fonctionnelle, séparément du code métier de l'application. Il devient alors possible d'associer différentes stratégies de gestion de la duplication pour une même application, en fonction du contexte d'exécution de l'application.

6.2 Cadre

Ces travaux font partie d'une réflexion globale dans le projet Sardes de l'INRIA-Rhône-Alpes sur la gestion d'aspects non-fonctionnels dans les systèmes à composants.

Différentes propriétés non-fonctionnelles sont abordées, en particulier celles liées à la tolérance aux pannes, à la sécurité et à la gestion de la mobilité. Nous nous intéressons ici à la gestion de la duplication.

Les expérimentations que nous menons sur ce sujet reposent sur l'utilisation de logiciels libres diffusés dans le cadre du consortium ObjectWeb. En particulier, dans l'expérimentation ici présentée, nous nous appuyons sur l'implantation OpenCCM du modèle à composant de Corba.

6.3 Réalisation

La duplication couvre deux points essentiels : la gestion de la duplication et la gestion de la cohérence. La gestion de la duplication considère la stratégie et les mécanismes de création et de placement des copies sur les différentes machines. La gestion de la cohérence considère les relations entre ces copies. Afin de supporter différents scénarios de duplication, les gestions de la duplication et de la cohérence doivent permettre de spécifier :

- La configuration de la duplication (quoi, quand, où). La configuration de la duplication doit permettre le contrôle des entités qui sont dupliquées (quoi), c'est à dire les composants qui sont dupliqués. Elle doit également permettre de contrôler le moment auquel la duplication doit être effectuée (quand). Enfin, elle doit permettre le contrôle du placement de ces duplicas dans un système réparti (où). En résumé, il doit être possible de configurer une stratégie de duplication.
- La configuration de la cohérence (comment). Etant donné qu'aucun protocole de gestion de la cohérence ne répond à tous les besoins des applications et des environnements d'exécution, la configuration de la cohérence doit permettre d'adapter le protocole qui assure la cohérence entre les copies des composants.

La configuration de la duplication Dans CCM, le modèle de déploiement permet la description d'une architecture initiale d'application. Cette description définit quels composants doivent être déployés, sur quels sites ils doivent l'être et comment les composants déployés doivent être interconnectés. Même si à l'heure actuelle, CCM ne considère pas la reconfiguration d'une architecture à composant (modification dynamique de l'architecture, correspondant au "quand"), le modèle de déploiement est l'endroit le plus indiqué pour décrire une stratégie de duplication.

Puisque la duplication consiste à créer et interconnecter des copies de composants dans l'architecture de l'application, la configuration de la duplication peut être naturellement spécifiée dans le modèle de déploiement. Le modèle de déploiement étendu avec la duplication devra permettre de décrire l'ensemble des composants dupliqués, le moment auquel la duplication doit prendre effet et le placement voulu pour chaque copie. La définition d'une stratégie de création dynamique de copies pourra s'appuyer sur des services de reconfiguration au niveau du modèle de déploiement de CCM lorsqu'ils seront définis.

En conséquence, la configuration de la duplication dans OpenCCM est ajoutée dans les programmes de déploiement. Comme ces programmes contrô-

lent la création des instances de composants et leur interconnexion, ils sont également responsables de la création des copies de composant et des interconnexions avec les autres composants. Par exemple dans une stratégie de gestion de cache, le programme de déploiement doit connecter un composant client à une copie locale d'un composant serveur, en remplacement de la copie distante. Ces copies du serveur devant être maintenues cohérentes, elles sont reliées par une connexion appelée "connexion de cohérence" que nous décrivons par la suite.

La configuration de la cohérence La gestion de la cohérence dans un système à objets dupliqués repose en général sur des *objets d'interception* qui déclenchent des actions de gestion de la cohérence lorsque les objets sont appelés. L'utilisation de mécanismes d'interception dans un système à composants permet d'intégrer la gestion de la cohérence sans modifier le code métier des composants. Les actions de gestion de la cohérence prennent la forme de traitements avant et après les appels de méthodes.

Les protocoles de cohérence définissent les relations entre les copies et fournissent les traitements pour maintenir la validité de ces relations. Logiquement, ces traitements nécessitent l'existence de connexions entre les copies afin de propager les actions de cohérence. Il s'agit des "connexions de cohérence" mentionnées ci-dessus, qui sont établies lors du déploiement. Dans l'exemple de la stratégie de gestion de cache, les actions de cohérence peuvent consister à invalider ou mettre à jour des copies et elles sont implantées avec des connexions de cohérence.

La plupart des protocoles de gestion de cohérence nécessitent d'accéder aux données internes des composants (pour copier et mettre à jour leur état). Dans l'exemple de la gestion de cache, les données des composants doivent être copiées lorsqu'un composant est mis en cache ou mis à jour. L'accès aux données internes des composants peut être basé sur des mécanismes de capture et restauration globales des composants, ou sur des fonctions d'accès et de modification de l'état spécifiques aux applications. L'implémentation de la gestion de la cohérence peut s'appuyer sur ces primitives sans connaître les détails d'implémentation du composant.

Les mécanismes d'interception et les connexions de cohérence forment la base à l'implantation d'une gestion de la cohérence. L'implantation dépend du protocole de cohérence considéré pour une application donnée.

Prototype L'implantation de cette gestion non-fonctionnelle de la duplication de composants prend place comme une application d'OpenCCM. Il

s'agit à l'heure actuelle d'un *patron de conception* visant à intégrer la gestion de la duplication dans les programmes de déploiement et la gestion de la cohérence dans des intercepteurs. Cependant, il est envisagé d'automatiser cette tâche en implantant un configurateur de duplication qui modifie les programmes de déploiement et *génère des intercepteurs* en fonction d'une spécification de duplication.

Ce patron a été validé avec une application de gestion de réservation d'équipements collectifs utilisée à l'INRIA Rhône-Alpes. Pour cette application, nous avons implanté deux stratégies de gestion de la duplication. La première vise à gérer des composants dupliqués permettant un accès plus rapide aux données (système de mise en cache). Le deuxième implante un schéma de duplication pour la gestion du mode déconnecté. Les données peuvent être dupliquées sur une machine mobile, permettant ainsi de consulter et réserver des équipements en mode déconnecté. La reconnexion de la machine mobile resynchronise des données modifiées et notifie l'utilisateur en cas de conflit de réservation. Ces deux stratégies ont été implantées sans modification du code métier de l'application originelle.

6.4 Apports conceptuels

Ces travaux ont permis de montrer qu'il était possible de gérer la duplication de composants comme une propriété non-fonctionnelle dans un système à composants. Cette gestion est non-fonctionnelle, car elle ne nécessite aucune modification du code métier de l'application. Il est alors possible d'associer à une même application différentes politiques de gestion de la duplication et de la cohérence.

Cette gestion non-fonctionnelle prend place au niveau des programmes de déploiement et au niveau de l'interception des appels de méthodes entre les composants. Cette intégration paraît d'autant plus judicieuse que les programmes de déploiement et les interceptions sont des caractéristiques que l'on retrouve dans la plupart des systèmes à composants.

6.5 Limites et perspectives

La perspective principale ouverte par ces travaux est la réalisation d'outils de plus haut niveau permettant d'automatiser la génération de schémas de duplication. Ces outils devraient permettre de générer des composants duplicables et de modifier les programmes de déploiement pour implanter une stratégie de duplication.

Une expérimentation a été menée avec le modèle à composants CCM,

mais il serait intéressant de l'appliquer à d'autres modèles comme COM ou EJB. D'autres travaux sont actuellement en cours avec d'autres aspects, notamment la persistance, la mobilité et la sécurité.

7 Conclusion

En utilisant les techniques d'objets d'interposition, le tissage de code, et les composants, nous avons successivement mené les cinq expérimentations suivantes.

Tout d'abord les trois premières études portent toutes sur les beans de JOnAS : ajout de la mobilité par la bibliothèque ProActive, ajouts centralisés et dynamiques de relations d'interactions, adaptation aux variations de l'environnement (bande passante, etc). Ces trois expérimentations utilisent à la base une technique d'interception des requêtes envoyées à un bean. La quatrième expérience vise, en utilisant un modèle à composant (Fractal), à structurer en composant l'ORB Jonathan, ceci afin de rendre ce dernier plus facilement configurable et adaptable — par exemple au niveau des protocoles utilisés. Enfin, la cinquième et dernière expérimentation montre que l'on peut gérer la réplication et les protocoles de cohérence associée comme une propriété non-fonctionnelle. L'expérimentation porte sur OpenCCM, utilisant dans cette première version un simple patron de conception, mais avec en perspective l'utilisation d'objets d'interceptions.

Globalement, nous pouvons dire que les techniques utilisées permettent bien d'adapter et d'étendre les plates-formes; toutes ces expérimentations ont été globalement très positives. Il est à noter l'apparition d'un point de focus important et naturel : l'endroit d'une plate-forme où sont inter-connectés, appelés, les composants techniques implémentant les services fonctionnels. Ce point de focus est GenIC dans le cas de JOnAS. En terme réflexif, cela correspond à une sorte de compositeur de meta-objets. Techniquement, il n'est pas très difficile d'ajouter de nouveaux services techniques en ajoutant de nouveaux "meta-objets".

En perspective, il faut cependant noter une conclusion importante qui rejaillit de plusieurs expérimentations. Si être capable en pratique d'ajouter de nouveaux services techniques est une première étape, elle doit se poursuivre avec un aspect sans doute encore plus délicat :

la capacité des services techniques à se composer.

En effet, la cohérence de la composition n'est pas toujours évidente, et quelques fois il faudrait revoir complètement l'implémentation, voire la sémantique des services techniques existants. En d'autres termes, faire des ser-

vices techniques des "boites noires" appelées composants n'est pas toujours suffisant pour les rendre complètement composables.

Une perspective prometteuse nous semble résider dans la capacité de définir une *sémantique comportementale* pour les composants, les caractérisant ainsi plus finement que ce qui est fait actuellement avec de simples interfaces. Cela devrait permettre de progresser dans la capacité à composer correctement les services techniques, et donc à rendre adaptables et extensibles les plates-formes middlewares.

Références

- [1] D. Caromel, W. Klauser, J. Vayssiere, Towards Seamless Computing and Metacomputing in Java, pp. 1043–1061 in *Concurrency Practice and Experience*, September–November 1998, 10(11–13), Editor Geoffrey C. Fox, Published by Wiley & Sons, Ltd.
- [2] F. Baude, D. Caromel, F. Huet, L. Mestre, J. Vayssiere, Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications, pp. 93–102, in *HPDC-11*, Edinburgh, Scotland, July 2002.
- [3] Laurent Baduel, Françoise Baude, Denis Caromel, Efficient, Flexible and Typed Group Communications for Java Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, November 3–5, 2002.
- [4] Laurent Vaills, Alexandre Guyot, Nicolas Guillier, ProActive-EJB, Rapport de projet ESSI, 2000–2001.
- [5] Mireille Blay-Fornarino, David Ensellem, Audrey Ocello Anne-Marie, Pinna-Dery, Michel Riveill, Jérémy Fierstone, Olivier Nano, Gilles Chabert, Un service d'interactions : principes et implémentation, Journées "Systèmes à composants adaptables et extensibles", 17 et 18 octobre 2002, Grenoble
- [6] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, W. Mann, Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4) :336–355, April 1995.
- [7] The Byte Code Engineering Library. <http://jakarta.apache.org/bcel/>, and Justice : An implementation of a free class file verifier for Java.
- [8] L. Berger, Évaluation Dynamique des Messages dans un Environnement Compilé et Distribué : Application aux Interactions entre Objets Distants, Colloque Langages et Modèles à Objets (LMO'99), Villefranche-sur-Mer (France). ISBN 2-7462-0008-2, 131–146, 27–29 janvier, 1999.

- [9] M. Bartorello, H. Maguin, M. Blay-Fornarino, A.-M. Dery, M. Riveill, Adjonction de services au sein d'un serveur EJB, Journées composants : flexibilité du système au langage, Besançon, 25 et 26 octobre, 2001
- [10] Pierre-Charles David, Thomas Ledoux : An Infrastructure for Adaptable Middleware, Springer-Verlag, LNCS 2519, DOA'02, Irvine, California, USA, October 2002.
- [11] Zahi Jarir, Pierre-Charles David, Thomas Ledoux : Dynamic Adaptability of Services in Enterprise JavaBeans Architecture, ECOOP'02, Workshop on "Component-Oriented Programming", Malaga, Spain, June 2002.
- [12] N. M. N Bouraqadi-Saâdani, T. Ledoux, M. Südholt : A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation, Invited presentation at the International Workshop on "Experiences with Reflective Systems" (held in conjunction with Reflection 2001).
- [13] Eric Bruneton, Thierry Coupaye, Jean Bernard Stefani, "Recursive and Dynamic Software Composition with Sharing", Workshop on Component Oriented Programming, 2002, Malaga, Spain.
- [14] Thierry Coupaye, Romain Lenglet, Mikaël Beauvois, Eric Bruneton, Pascal Déchamboux, "Composants et composition dans l'architecture des systèmes répartis", Journées Composants, 2001, Besançon, France.
- [15] Bruno Dumant, François Horn, Frédéric Dang Tran, Jean-Bernard Stefani, "Jonathan : an Open Distributed Processing environment in Java", Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer-Verlag, pp. 175–190, 1998.
- [16] V. Marangozova, D. Hagimont, Adaptation d'une application répartie pour la disponibilité : expérience et évaluation, 2ième Conférence Française sur les Systèmes d'Exploitation (CFSE-2), Chapitre Français de l'ACM-SIGOPS, Paris, Avril 2001.
- [17] V. Marangozova, D. Hagimont, Availability through Adaptation : a Distributed Application Experiment and Evaluation, European Research Seminar on Advances in Distributed Systems (ERSADS'2001), Bologne, Mai 2001.
- [18] V. Marangozova, Patrons de conception pour la duplication de composants, Journées francophones des jeunes chercheurs en systèmes d'exploitation, Hammamet, Avril 2002.

- [19] V. Marangozova, D. Hagimont, An infrastructure for CORBA component replication, 1st IFIP/ACM Working Conference on Component Deployment, Berlin, Juin 2002.
- [20] V. Marangozova, D. Hagimont, Non-functional Replication Management in the Corba Component Model, 8th International IFIP/ACM Conference on Object-Oriented Information Systems, Montpellier, Septembre 2002.