Architecture Orienté Service Dynamique : D-SOA

Didier Parigot, Baptiste Boussemart

Architecture Orientée Services Dynamique : D-SOA

Didier Parigot*, Baptiste Boussemart*

*LogNet
INRIA Sophia Antipols - Méditerranée
@004,route des Lucioles BP 93 F-06902 Sophia-Antipolis cedex, France
Didier.Parigot@inria.fr,
http://www-sop.inria.fr/lognet

Résumé. Depuis quelques années, la notion d'Architecture Orientée Services (SOA) s'est rapidement répandue et a été largement acceptée par l'industrie du logiciel. L'aspect dynamique d'une SOA nous semble être l'un des points fondamentaux et cruciaux de cette approche.

Le concept d'une Architecture Orientée Services Dynamique D-SOA, se propose de fournir des moyens pour rendre l'architecture adaptable dynamiquement, en cours d'exécution, aux besoins des applications. Les entités qui collaborent pour une application donnée, ne sont pas forcément, connue ou prévisible statiquement. Une D-SOA doit pouvoir s'adapter dynamiquement à son environnement. Elle doit pouvoir supporter aisément des évolutions et des changements dans l'utilisation même de l'application. Elle doit permettre de connecter des entités sans qu'il soit nécessaire qu'elles se connaissent au préalable, qu'elles partagent une même interface de services.

A travers l'implémentation de D-SOA au-dessus de la plate-forme OSGi on montre d'une part, sa complémentarité par rapport à OSGi, puis d'autre part, sa simplicité de mise en œuvre. Cette immersion de D-SOA au-dessus OSGi permet d'envisager des utilisations pour un large public sur des petits supports matériels tels que des PDAs et Smartphones. Finalement, cette notion de SOA dynamique, nous semble être un élément important pour les architectures des applications pour l'Internet du Futur, surtout dans le contexte dynamique des réseaux sociaux et de l'Internet des Objets.

1 Introduction

Depuis quelques années, sur les problématiques autour de la conception de l'architecture logicielle, la notion d'Architecture Orientée Services [Anand et al. (2005)] (SOA) s'est largement imposée grâce en particulier à la montée en puissance de l'Internet. Maintenant, le concept de SOA recouvre des aspects et des domaines très variés qui dépassent d'une certaine manière largement le domaine initial qu'est l'architecture logicielle. Cette situation peut parfois induire une certaine confusion dans son interprétation. En effet l'un des points forts de cette approche a été de répondre à un fort besoin d'interaction entre les entités logicielles

qui composent une application finale. L'approche SOA propose un mécanisme d'assemblage souple à l'aide de services, indépendant d'une technologie particulière. Elle propose une architecture guidée par les besoins métiers (les services) de l'application. Elle propose une architecture qui s'adapte aux besoins de l'application. C'est ce point de vue de l'adaptation dynamique de SOA que traite cet article et que nous nommons D-SOA. Plus précisément, D-SOA est issue de l'architecture [Courbis et al. (2004)] de notre fabrique logicielle SmartTools [Parigot et Courbis (2005)] qui demandait une architecture fortement évolutive. Par la nature même de cette fabrique SmartTools, son architecture devait pouvoir accepter n'importe quelles entités en cours de fabrication.

Le concept de SOA dynamique se propose de fournir des moyens pour rendre l'architecture adaptable dynamiquement, en cours d'exécution, aux besoins des utilisateurs de l'application. L'assemblage des composants pour une application donnée, n'est pas forcément connu au démarrage de l'application. De plus, pour un composant donné, le protocole de communication et la localité (la machine qui propose ce composant) peuvent changer d'une invocation à l'autre. Plus précisément, les moyens de communication doivent rester ouverts à tout type de support de communication (protocole). On ne doit pas imposer une seule forme de communication. Cette généricité dans les moyens de communication doit être transparente à l'utilisateur. L'utilisation d'une ressource (sous forme d'un composant) qu'elle soit en répartie ou en locale, doit être spécifiée de la même manière dans le code source de l'application.

Les concepts importants de D-SOA sont les suivants :

- de concevoir par défaut une architecture en mode d'exécution répartie pour toutes ses entités ;
- de spécifier d'abord les besoins de l'application, à travers la conception de modèle simple, sans préjuger des choix techniques ultérieurs ;
- de pouvoir accepter de nouvelles entités (non connues) et d'accéder dynamiquement à ces nouveaux services.

L'article est organisé en 5 sections. La première section introduit les concepts de base de D-SOA qui sont d'une part, deux DSLs (*Domain-specific Language*) qui décrivent les services d'un composant (CDML) et les connexions entre composants (World) et d'autre part, les trois opérations élémentaires qui sont les opérations d'envoi et de réception de messages, puis l'opération de connexion entre deux composants. La deuxième section décrit plus précisément ces deux DSLs et ces trois opérations élémentaires. La quatrième section décrit, pour un mode d'exécution locale, le gestionnaire de composants (CM) qui organise l'architecture. Ce composant CM est l'élément principal de D-SOA. Une implémentation de D-SOA au-dessus de OSGi [The OSGi Alliance (2005)] est rapidement présentée. Puis la cinquième section, pour un mode d'exécution en répartie de D-SOA, décrit une organisation modulaire du CM. L'article se termine avec une section de comparaisons et une conclusion.

Par contre, cet article ne présente pas dans tous ces détails le modèle de composants, issue de SmartTools. En effet, les articles [Parigot et Courbis (2005); Courbis et al. (2004)] décrivent plus précisément les motivations d'un tel modèle de composants pour cette fabrique logicielle SmartTools. De plus, les aspects techniques de l'intégration de D-SOA au-dessus d'OSGi ne sont pas traités dans l'article car cela demanderait d'entrer dans les détails de la technologie OSGi.

2 Principe de base d'une SOA Dynamique

Dans cette section nous présentons les concepts de base d'une SOA dynamique. D-SOA est basé à la fois sur un développement dirigé par les modèles, le concept de publication et utilisation de services (*Publish/Subscribe* [Eugster et al. (2003)]) et d'une connexion dynamique entre composants. D-SOA se décompose en une phase de génération bâtie autour d'un générateur de composants (*Component Generator* CG) et d'une partie d'exécution bâtie autour d'un gestionnaire de composants (*Component Manager*, CM). En terme de développement dirigé par des modèles, la démarche s'appuie sur deux DSLs.

Le premier DSL dénommé *Component Description Meta Language* CDML, décrit les services d'un composant. Le générateur de composants (CG) utilise ce fichier pour produire le code non-fonctionnel des composants (génération des conteneurs). Un CDML indique les services fournis (mot clef `input`) ou requis (mots clef `output`) pour un composant. Plus précisément, un CDML définit un composant qui requiert et fournit ces services. Lors de l'exécution, plusieurs instances de ce composant pourront être créées.

Comme exemples de bases, nous utiliserons les deux composants `cmp1` et `cmp2` decrits par les CDML des figures 1 et 2. La figure 4 montre une exécution possible avec deux instances (`cmp2-1` et `cmp2-2`) du composant `cmp2` et une seule instance du composant `cmp1`.

La génération du conteneur s'appuie sur un modèle de communication asynchrone. Les instances de composants communiquent entre elles à l'aide de messages asynchrones. Cette communication s'effectue à l'aide d'une méthode d'envoi (`envoi`) de messages et d'une méthode de réception (`réception`) de messages. Cette méthode `envoi` transforme l'appel d'une méthode en un message asynchrone. La méthode de réception `réception` transforme un message asynchrone en l'appel à la méthode qui réalise le service associé.

Le deuxième DSL dénommé World, décrit la topologie de l'application, à l'aide d'une opération de connexion entre deux instances de composant, dénoté `connectTo`. La topologie d'une application correspond aux instances de composants présentes au démarrage et aux connexions entre ces instances. Au lancement d'une application, le CM est lancé puis il lit un fichier World pour instancier la topologie de l'application. Par exemple, la topologie de la figure 4 correspond au fichier World de la figure 3. Pour effectuer l'opération de connexion entre deux instances (`connectTo`), le CM utilise le descriptif CDML de chaque instance de composant pour associer les services requis par l'une des instances avec les services fournis par l'autre instance et réciproquement. En cours d'exécution, une instance peut dynamiquement demander à être connectée à une autre instance. Pour cela elle utilise ce service `connectTo` du CM. Par exemple, l'instance `A` envoie le message (`connectTo A B`) au CM pour établir une nouvelle connexion avec l'instance `B`. Si cette instance `B` n'existe pas, le CM créerait dynamiquement cette instance avant d'établir la connexion avec `A` .

2.1 Description des langages domaine-spécifiques de D-SOA

2.1.1 Description des services d'un composant : CDML

Dans D-SOA, un service (requis ou fournis) correspond à une seule méthode dans le code métier d'un composant. Le fichier CDML décrit les services fournis (`input`) et les services requis (`output`). Par exemple, le CDML de la figure 1 qui décrit le composant

```
<component name="cmp1" extends="abstractContainer" ns="cmp1">
<containerclass name="Cmp1Container" />
<facadeclass name="Cmp1AppFacade" userclassname="Cmp1App" />
<output name="updateLabel" method="updateLabel">
<arg name="message" javatype="java.lang.String" />
</output>
</component>
```

FIG. 1 – *Définition de* `cmp1` *en* CDML.

```
<component name="cmp2" extends="abstractContainer" ns="cmp2">
<containerclass name="Cmp2Container"/>
<facadeclass name="Cmp2AppFacade" userclassname="Cmp2App"/>
<input name="updateLabel"  method="updateLabel">
<arg name="message" javatype="java.lang.String"/>
</input>
</component>
```

FIG. 2 – *Defintion de* `cmp2` *en* CDML.

(`name="cmp1"`) définit le service requis `updateLabel`. Symétriquement le CDML de la figure 2 pour le composant (`cmp2`) définit le service fournis `updateLabel`.

Comme le modèle de communication de D-SOA est basé sur l'envoi de messages asynchrones, les services ne retournent pas de résultat. Chaque service est défini comme le profil d'une méthode Java sans type de retour. Le générateur CG prend en entrée un fichier CDML et génère le conteneur et la façade associés à ce composant. Dans un CDML, on précise la classe Java pour le conteneur généré (`containerclass`) et de même pour la façade générée (`facadeclass`) avec la possibilité d'étendre une façade déjà écrite (`userclassname`) par le développeur. Le CG utilise une notion simple d'héritage simple (`extends`) qui permet de définir un CDML par extension d'un autre CDML. En effet, les services de base associés au cycle de vie d'un composant doivent être toujours présents. Ils sont décrits par le CDML `abstractContainer` commun à tout composant. Ce CDML définit le service `connectTo` requis par tout composant qui est le service de base de D-SOA. Ce service est fourni par le CM.

Dans D-SOA, un message correspond à un service et un service correspond à une méthode. Le rôle du conteneur est de transformer un message entrant (`réception`) qui correspond à un service fournis, par un appel vers la méthode définie dans la façade du composant, et inversement de transformer un appel à une méthode qui correspondent à un service requis, en un message sortant (`envoi`) vers le composant fournissant ce service. Ces transformations font que la communication entre les instances de composant s'effectue toujours à travers ces deux méthodes génériques : `envoi` et `réception`. Cette propriété de généricité permet une plus grande souplesse pour la gestion des connexions entre instances de composant. Ces deux méthodes `envoi` et `réception` sont expliquées plus en détails dans la section 2.2.

2.1.2 Description de la topologie d'une application : World

Le fichier World (voir la figure 3), lu et exécuté par le CM, lors du lancement de l'application décrit l'ensemble des connexions entre les instances de composants. Une instance de composant est identifiée par le couple : (nom du composant et nom de l'instance). Par exemple,

```
<world>
<connectTo id_src="ComponentsManager" type_dest="cmp1" id_dest="cmp1" />
<connectTo type_src="cmp1" id_src="cmp1" type_dest="cmp2" id_dest="cmp2-1" />
<connectTo type_src="cmp1" id_src="cmp1" type_dest="cmp2" id_dest="cmp2-2" />
</world>
```

FIG. 3 – *Exemple d'un fichier en* World.

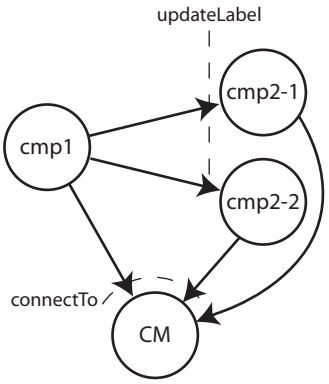

FIG. 4 – *Schema des instances de composant pour le fichier* World *de la figure 3.*

l'instance (`cmp2, cmp2-2`) de la figure 3 correspond à une instance `cmp2-2` du composant `cmp2`. L'opération de connexion `connectTo` prend comme argument l'instance source (`type_src id_src`) et l'instance de destination (`type_dest id_dest`). Lorsqu'une instance est crée par le CM, elle est par défaut connectée au CM. En cours d'une exécution, une instance de composant peut demander dynamiquement de nouvelles connexions au CM à l'aide du service `connectTo`. Le rôle du CM est de fournir aux instances ce service de connexion.

2.2 Les opérations de base de D-SOA

D-SOA est basée principalement sur les trois opérations suivantes : l'opération de connexion , l'envoi de message et la réception d'un message.

2.2.1 Opération de connexion entre deux instances de composant

L'opération (`connectTo A B`) effectue la connexion entre les deux instances `A` et `B`. Le principe de base de cette opération est d'associer par nom, les services fournis (`input`) par la première instance `A` avec les services requis (`output`) par la seconde instance `B` et inversement, les services fournis de `B` avec les services requis de `A`. Cette association s'effectue simplement par association des noms des services. Par exemple dans la figure 5, le service étoile requis de `B` est associé au service étoile fournis de `A` et inversement, le service triangle requis de `A` est associé au service triangle fournis par `B`. La figure 5 montre la connexion entre les instances `A` et `B`, elle se déroule en 4 étapes suivantes :

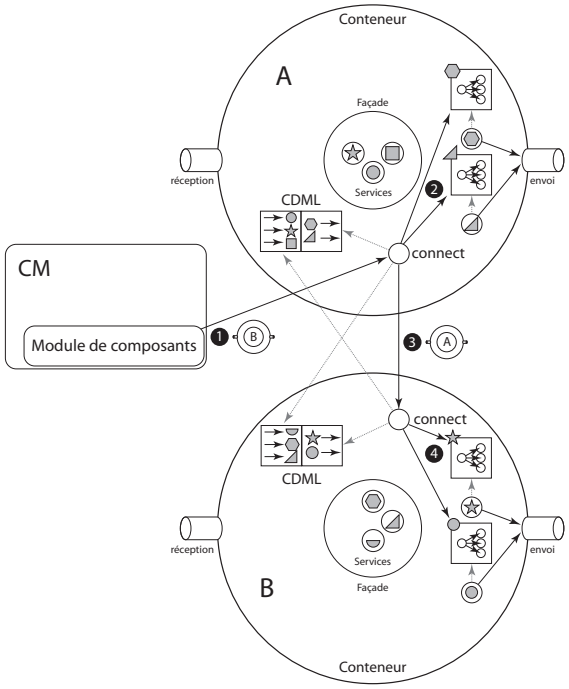

FIG. 5 – *Operation de connexion entre instances de composant.*

1. La méthode `connect` est appelée par le module de composants sur le conteneur `A` (source) avec comme argument le conteneur `B` (destination)

2. Ajout du conteneur `B` aux services de sorties de `A` qui sont respectivement identiques aux services d'entrées de `B`.

3. Pour la connexion au sens inverse, le conteneur `A` invoque la methode `connect` sur le conteneur `B` en passant comme argument lui-même, donc le conteneur `A`.

4. Ajout du conteneur `A` aux services de sorties de `B` qui sont respectivement identiques aux services d'entrées de `A`.

L'opération d'association des services proprement dite (la méthode `connect` de la figure 5) est effectuée localement sur chaque instance à l'aide des deux fichiers de composant CDML associés à chaque instance.

Cette opération de connexion impose que l'instance source (`A`) soit active sinon l'opération échoue. Par contre pour l'instance de destination (`B`), si elle n'existe pas, le CM va s'il connait le composant associé, créée l'instance de destination. La section 3.1 donnera plus d'explications sur le rôle du CM, en particulier sur la création d'instance. Après que le CM ait établi une connexion entre deux instances, les instances communiquent directement (en Pair à Pair) entre elles par messages asynchrones, sans passer par le gestionnaire.

Message
nom du message (String) nom de l'instance de l'expéditeur (String) nom du composant de l'expéditeur (String) nom de l'instance du destinataire (String) nom du composant du destinataire (String) données (PropertyMap = Map<String,Object>)
... constructeur et accesseurs

FIG. 6 – *Le format des messages.*

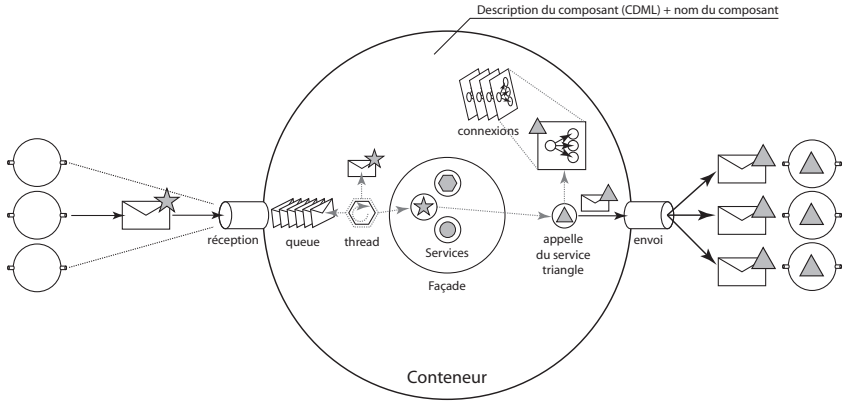

FIG. 7 – *Architecture d'un conteneur.*

2.2.2 Opération d'envoi et de réception des messages

Tout appel de méthode qui correspond à un service requis est transformé en un appel à la méthode d'envoi (`envoi`) avec comme argument un objet de classe `Message`. Cet objet message (voir la figure 6) représente une forme sérialisée de l'appel de la méthode. La méthode `envoi` appelle la méthode `réception` sur toutes les instances de composants connectées à ce service. Cette association est établie lors de la connexion par la méthode `connect` de la figure 5. La méthode `réception` ajoute le message reçu à une queue. Enfin, le conteneur possède une boucle de traitement des messages, exécuté dans un processus léger. La figure 7 montre le fonctionnement d'un conteneur, dans le cas, d'une réception d'un message (étoile) et de l'envoi d'un message (triangle) vers trois instances.

3 D-SOA en mode d'exécution locale

Cette section détaille le rôle et l'architecture du CM pour une exécution en mode locale, sur une seule machine virtuelle Java (JVM). Le CM est le composant central de D-SOA. Dans la section 4 on expliquera l'architecture du CM en mode réparti.

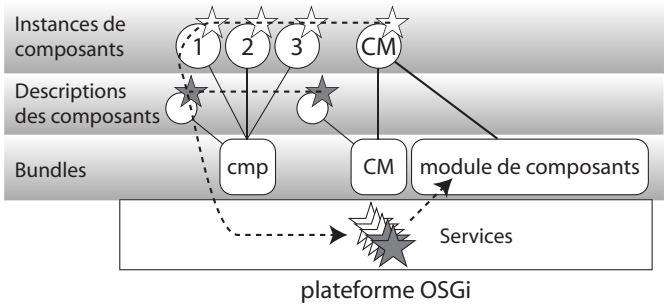

FIG. 8 – *Organisation de* D-SOA *au-dessus d'*OSGi.

3.1 Le gestionnaire de composant en mode d'exécution locale

Les rôles du CM sont :

- de créer la topologie initiale définie par le fichier World, lu au lancement ;
- de connaître les composants disponibles et leurs fichiers CDML associés ;
- de connaître l'ensemble des instances créées ;
- de créer une instance de composant, lorsque c'est nécessaire ;
- d'établir une connexion entre deux instances de composant.

Le CM doit maintenir une table des composants disponibles, et une table des instances actives (créées). Lors d'une demande de connexion, si l'instance de destination (`id_dest`) n'est pas dans la table des instances créées et que le composant de destination (`type_dest`) est dans la table des composants disponibles, alors le CM créé cette instance de destination sinon la demande échoue.

3.2 D-SOA au-dessus de la plate-forme OSGi

En termes d'implémentation (voir le site gforge [gforge]), D-SOA s'exécute au-dessus de la plateforme OSGi. Plus précisément, chaque composant de D-SOA correspond à un module OSGi (*bundle*) pour la gestion du cycle de vie. Pour une exécution de D-SOA, une plateforme OSGi sera lancée avec le CM (bundle) démarré par défaut. Dans ce contexte, deux services OSGi sont utilisés et publiés. Le premier `ContainerService` permet lors du démarrage d'un composant (bundle), de publier le CDML associé au CM qui ajoute ce dernier à sa table des composants disponibles. Le deuxième `ContainerProxy` permet lors de la création d'une instance de composant de publier au CM, le conteneur de cette instance qui est ajouté à la table des instances créées. La figure 8 montre cette architecture au-dessus d'OSGi et la publication des deux services OSGi.

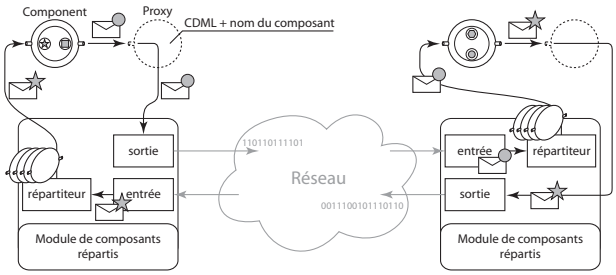

FIG. 9 – *Execution en répartie de* D-SOA.

4 D-SOA en mode réparti

Dans cette section, une organisation modulaire du CM est présentée. Cette organisation permet d'étendre facilement le CM en fonction de la nature de l'exécution de l'application, en locale ou en répartie. Plus précisément, le mode local correspond à une exécution sur une seule JVM, et le mode réparti correspond à une exécution sur plusieurs JVM. Mais surtout cette organisation du CM en modules permet de s'abstraire du mode d'exécution pour l'opération de connexion. C'est le CM qui va déterminer si l'instance de destination est accessible en local ou en réparti.

4.1 Le CM en mode réparti

Double connexions Le CM, sur chaque JVM, est responsable de la création des instances créées localement. Une demande de connexion à distance (l'instance destinataire est présent sur une machine distante) est sous le contrôle des deux CM (de l'instance source et de l'instance destination). Dans la section 2.2.1, on a montré qu'une opération de connexion s'effectue en quatre étapes. Les deux dernières étapes ne peuvent s'effectuer que sur la machine à distance, par le CM de destination. Il faut donc que le CM source envoit au CM destinataire une opération de connexion dans le sens inverse pour établir la connexion dans les deux sens. Pour cela les deux CM sont connectés comme deux instances de composant avec le service `connectTo` à la fois fournis et requis.

Double création des *proxies* L'exécution en réseau de notre architecture demande un mécanisme de création d'objets d'interposition (*proxy*). En effet lorsqu'une instance demande à être en connexion avec une instance distante, alors un *proxy* doit être créé en local pour la communication de la source vers la destination. Comme énoncé dans le paragraphe précédent, pour la communication dans l'autre sens, un proxy doit être aussi créé sur la machine distante pour la communication de la destination vers la source. Dans la sous-section 4.3 le module responsable de la création effective de ces *proxies* sera précisé et la structure de ces *proxies* sera détaillée. La figure 9 donne le schéma de cette communication à distance.

Recherche et Création d'une instance de composant en mode réparti En mode réparti, pour savoir si une instance est déjà créée, le CM ne doit pas se contenter de rechercher uniquement en local. Si l'instance n'existe pas en local alors le CM doit aussi demander à l'ensemble des CMs connectés à lui même. Le CM doit donc connaître en plus des instances locales, les instances créées à distance (instances réparties).

Pour une meilleure modularité et gestion des informations, le CM va déléguer la gestion des tables (composants et instances) à des modules de composants particuliers qui prendront en charge, les tables de composants et d'instances pour chaque mode de communication. Pour chaque mode d'exécution et pour chaque protocole de communication, un module de composants particulier est défini. A l'exécution, un ou plusieurs modules peuvent être connectés au CM. Un module de composants gère sa table des composants disponibles et sa table des instances de composants créées indépendamment des autres modules de composants. Le module de composants pour le mode local sera toujours démarré avant le CM. Puis en fonction du protocole de communication choisi, le module de composants associé sera démarré. La sous-section 4.3 décrit plus précisément ces modules de composants.

4.2 Stratégie de choix d'un module de composants

Le e CM définit la stratégie de choix du module de composants qui va réaliser la connexion effective. Par exemple une stratégie pourra privilégier d'abord les connexions locales par rapport à des connexions réparties. La structure du CM permet d'instancier différentes stratégies, en utilisant le patron de conception *Command* (voir la figure 10. La demande de connexion aux modules de composants s'effectue en deux phases. Dans la première phase, le CM interroge l'ensemble des modules de composants actifs, sur la présence ou non de l'instance de destination. Chaque module de composants répond (de manière asynchrone) au CM. Lorsque le CM est en possession de toutes les réponses (même négatives), alors dans la seconde phase, il choisi en fonction de sa stratégie, le module de composants qui se charge de la connexion effective. Si dans la première phase, il n'y a aucune réponse positive, la requête de connexion est mise en attente jusqu'à un événement (message), tel qu'un composant a été démarré ou découvert. Finalement, le CM doit gérer une liste de requêtes non encore satisfaites. Comme les modules de composants associés au CMs sont en communication, les modules de composants peuvent s'avertir entre eux du démarrage de nouvelles instances. La figure 10 donne le schéma de l'architecture du CM et d'une stratégie possible.

4.3 Les modules de composants

Le rôle de ces modules de composants, en plus de la gestion des tables (composants et instances) est de mettre en place en fonction du protocole de communication, la connexion entre deux instances à travers le réseau. Le module de composants, lors d'une connexion en répartie, va créer le *proxy* dédié au protocole du module. Dans D-SOA, les *proxies* ne font que redéfinir les deux méthodes génériques de communication `envoi` et `réception`. Par exemple la méthode `envoi` est redéfinie pour écrire dans le port ouvert associé au protocole. Pour la méthode `réception`, cela dépend fortement du protocole, et sera donc explicitée plus tard pour chaque protocole. Par contre, une implémentation d'un *proxy* ne dépend pas du composant en connexion. En effet la communication entre instances s'effectue toujours à travers ces deux méthodes génériques pour tous types de connexion. Avec cette propriété de

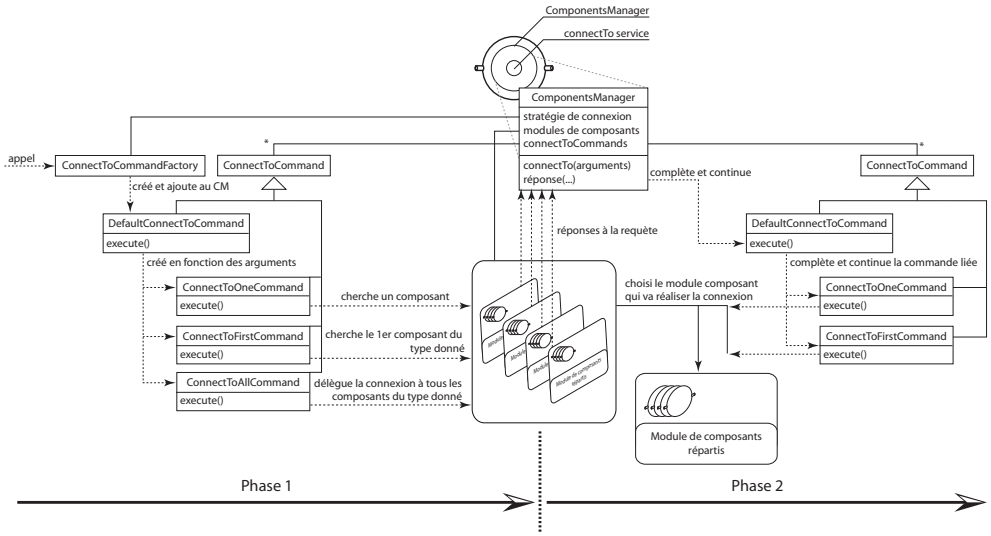

FIG. 10 – *Architecture et stratégie du gestionnaire de composant.*

généricité, il n'y a pas de génération de *proxy*, mais juste la création d'un objet d'une classe *proxy*, avec une implémentation différente pour chaque protocole.

Les modules de composants pour UDP **et** TCP Pour tous les CM connectés, un seul port de communication est ouvert avec le protocole UDP. Donc il est nécessaire que les messages lus sur cet unique port soient redirigés vers l'instance de destination. Cela est effectué par un répartiteur de message qui trouve le nom de l'instance de destination dans le message. Le module `SimpleModuleUDP` instancie ce répartiteur UDP et l'associe au port UDP. Par contre pour le protocole TCP, pour chaque CM connecté, le serveur TCP va ouvrir une *socket*. Un répartiteur reste donc nécessaire pour rediriger les messages envoyés par le même CM (même JVM). Ainsi pour chaque protocole, une implémentation différente de la classe *proxy* existe et prendre en compte l'appel au répartiteur spécifique qui est géré par le module de composants en question.

Module de composants au-dessus d'un mécanisme de tuyaux virtuels : `VirtualPipes`
Un mécanisme de tuyaux virtuels qui gère plusieurs tuyaux de communication dans un seul tuyau physique (une *socket* TCP par exemple) permet d'associer plusieurs tuyaux virtuels par connexion physique (TCP). Ce mécanisme est similaire à celui introduit dans la librairie JXTA [Wilson (2002)], mais avec une notion de numéros de tuyau virtuel local au tuyau physique, et non un numéro unique à tout le réseau entier, permette une gestion plus souple des communications entre instances de composant.

4.4 D-SOA comme modèle d'exécution pour un *Overlay Network*

L'objectif d'un *Overlay Network* est d'organiser un ensemble d'agents (ordinateurs, terminaux etc...) accessibles au-dessus d'un réseau physique, afin que ces agents puissent partager (mutualiser) leurs services entre eux. Un *Overlay Network* doit fournir un ensemble de fonctionnalités qui sont pour un agent :

1. s'inscrire ou quitter le réseau ;
2. publier ses services et rechercher des services dans le réseau ;
3. exécuter les services en mode mutualisé.

Dans [Liquori et Cosnard (2007)] les auteurs proposent un protocole, dénommé Arigatoni, pour réaliser ces fonctionnalités, en particulier pour les points 1 et 2. Un premier prototype PiNet est en cours de développement dans notre équipe de recherche. Un module de composants connecté à PiNet permet d'offrir un modèle d'exécution à Arigatoni, mais cela permet de résoudre pour D-SOA les problématiques d'organisation, de gestion et de recherche d'instances de composants sur un réseau à grande échelle.

5 Comparaisons

OSGi : La phase de génération du CG apporte un support plus simple pour des services asynchrones que les solutions proposées par OSGi [The OSGi Alliance (2005)]. Puis le CM et le processus de connexion des instances fournissent de faits, une gestion automatique des notifications des services, qui est normalement du ressort du développeur. Par contre, la technologie OSGi apporte à D-SOA un excellent support logiciel pour le cycle de vie des composants et la gestion dynamique des modules.

Web Services : Dans [Variamparambil (2002)], à l'aide d'une transformation simple d'un CDML vers une description des services Web (WSDL) [Kopecký (2007)] équivalent, on a montré qu'un composant de D-SOA pouvait être facilement transformer en un *Web Services*. Le processus de transformation des appels de méthode en des messages de D-SOA permet d'être conforme au protocole de transport SOAP [SOAP (2000)] des *Web Services*. Par contre, la transformation d'un ensemble de composants de D-SOA en un ensemble de *Web Services* équivalents, est plus difficile, sauf dans le cas d'une topologie connue d'avance. En effet, la technologie des *Web Services*, impose pour la connexion entre des *Web Services* qu'ils connaissent mutuellement leur interface Java des services. Puis, par rapport à OSGi cette technologie des *Web Services* n'apporte peu de support techniques en terme de composant (cycle de vie).

R-OSGi : Pour une première expérience de notre D-SOA en mode réparti, nous avons utilisé avec succès l'approche R-OSGi [Rellermeyer et al. (2007)]. Mais l'inconvénient de cette utilisation est qu'il y avait des traitements effectués deux fois, comme par exemple, les transformations des appels de méthodes en des messages. De plus, la génération des *proxies* de R-OSGi en cours d'execution nous semblait bien compliquée pour notre contexte de communication générique. D-SOA est une version plus adaptée et légère que R-OSGi pour une exécution répartie.

RMI Sun Microsystems (1996) : Comme notre modèle de communication est à base de messages asynchrones, l'utilisation directe de la technologie RMI (appel synchrone), est d'un support peu adéquat à D-SOA.

Fractal et *Service Component Architecture* (SCA) : Par rapport à la plateforme Fractal [Coupaye et Stefani (2006)], le CM joue certainement un rôle assez similaire au concept de membrane qui est l'un des éléments fondateurs de cette approche, vis-à-vis de sa colonie d'instances de composants. Par contre les aspects de connexions dynamiques semblaient ne pas être une préoccupation initialement prévue par Fractal. Les avantages de l'utilisation de cette technologie Fractal pour D-SOA ne sont pas aussi clair que l'apport de la technologie OSGi. Dans D-SOA il y a une séparation nette entre la définition des services d'un composant (le fichier CDML) et l'utilisation des services (le fichierWorld) alors que dans les approches SCA et Fractal il n'y a pas cette séparation. Cette séparation permet d'accepter des topologies dynamiques. Comme une implémentation de SCA s'appuyant sur la technologie OSGi est en cours d'implémentation dans le projet Eclipse *Swordfish SOA Runtime Framework Project*, D-SOA, bâtie aussi sur OSGi, pourra facilement profiter de cet effort de développement.

Bus d'intégration ou *Enterprise Service Bus* (ESB) : Par rapport aux technologies de type bus d'intégration, D-SOA cherche au contraire à rendre chaque composant autonome. En effet les composants connectés à des bus d'intégration perdent leurs autonomies et ne peuvent plus communiquer vers l'extérieur qu'à travers la technologie de bus choisie. Par contre, un module de composants spécifique pourrait facilement gérer la communication avec un bus d'intégration.

6 Conclusion

Dans cet article nous avons présenté une SOA dynamique qui, d'une part, s'appui en terme de développement sur une plateforme standard du domaine, OSGi et d'autre part, propose des mécanismes d'extension en terme de services, d'ajout de connexions et de support de communication ce qui donnent à cette architecture des capacités d'extension et d'adaptabilité. De plus ces mécanismes sont complémentaires aux opérations de base d'une SOA standard. L'organisation du CM avec les modules de composants et le couplage avec un *Overlay Network* sont des moyens simples d'extension de D-SOA. L'utilisation de Java et d'OSGi comme support pour notre implémentation, permet de profiter pleinement des avancées dans ce domaine. Puis, comme la mise en œuvre de D-SOA est relativement simple (petite taille du code résultant), on peut envisager un déploiement sur de petits supports informatiques comme des téléphones portables. Il suffit qu'une JVM soit disponible sur les supports ciblés. En termes de perspectives, il faudra approfondir d'une part, notre mécanisme de tuyaux virtuels et d'autre part, la connexion avec un *Overlay Network*.

Références

Anand, S., S. Padmanabhuni, et J. Ganesh (2005). Perspectives on service-oriented architecture. In *ICWS*. IEEE Computer Society.

Coupaye, T. et J.-B. Stefani (2006). Fractal component-based software engineering. In M. Südholt et C. Consel (Eds.), *ECOOP Workshops*, Volume 4379 of *Lecture Notes in Computer Science*, pp. 117–129. Springer.

Courbis, C., P. Degenne, A. Fau, et D. Parigot (2004). Un modèle abstrait de composants adaptables. *revue TSI, Composants et adaptabilité 23*(2).

Eugster, Felber, Guerraoui, et Kermarrec (2003). The many faces of publish/subscribe. *CSURV : Computing Surveys 35*.

gforge. Smarttools gforge. http ://gforge.inria.fr/projects/smarttools/.

Kopecký, J. (2007). Web services description language (WSDL) version 2.0 : RDF mapping. World Wide Web Consortium, Note NOTE-wsdl20-rdf-20070626.

Liquori, L. et M. Cosnard (2007). Logical Networks : Towards Foundations for Programmable Overlay Networks and Overlay Computing Systems. In *TGC, Trustworthy Global Computing*, Lecture Notes in Computer Science. Springer.

Parigot, D. et C. Courbis (2005). Domain-driven development : the smarttools software factory. Technical Report RR-5588, INRIA, Sophia Antipolis.

Rellermeyer, J. S., G. Alonso, et T. Roscoe (2007). Building, deploying, and monitoring distributed applications with eclipse and R-OSGI. In L.-T. Cheng, A. Orso, et M. P. Robillard (Eds.), *ETX*, pp. 50–54. ACM.

SOAP (2000). Simple Object Access Protocol (SOAP) 1.1. http ://www.w3.org/TR/SOAP/.

Sun Microsystems (1996). *Java Remote Method Invocation Specification*. Sun Microsystems.

The OSGi Alliance (2005). OSGi service platform — core specification. Release 4.

Variamparambil, J. G. (2002). Enabling smarttools components with component technologies :webservices, corba and ejbs. Technical report, INRIA.

Wilson, B. J. (2002). *JXTA*. New Riders.

Summary

In recent years, the concept of Service Oriented Architecture (SOA) has spread rapidly and has been widely accepted by the software industry. The dynamic aspect of a SOA seems to be one of the fundamental and critical aspects of this approach.

The concept of a Dynamics Service-Oriented Architecture (D-SOA), is to provide ways to build adaptable architectures to the needs of applications. The entities, that collaborate for a given application, are not necessarily known statically. A D-SOA must be able to dynamically adapt to its environments and (its) various uses. A D-SOA must be able to connect to entities that are not necessarily known beforehand.

The implementation of D-SOA above the OSGi platform shows that it is complementarity to the OSGi approach, and it is simple to implement. This immersion of D-SOA above OSGi allows to build applications for a large audience and executables on small computer such as PDAs and smartphones. Finally, the notion of dynamic SOA seems to be important to the architectures of applications for the Internet of the Future, especially in the dynamic context of Social Networks and the Internet Objects.