Université de Nice Sophia-Antipolis
École Doctorale STIC
Master recherche Réseaux et Systèmes Distribués

Rapport de Stage

Architecture Orientée Services Appliquée à la construction de RCPs réparties

Prépare par : Mohamed Ouazara

Responsable de stage : Didier Parigot

Stage effectue à : Projet SmartTools, INRIA Sophia-Antipolis
Du : 01 mars 2007 *au* : 30 juin 2007

Remerciements :

Je tiens tout d’abord à remercier Didier Parigot, mon encadrant, qui m’a guidé tout au long de mon stage. Il a su trouver des problèmes intéressants et me guider dans leurs études. Je le remercie aussi pour ses conseils et sa disponibilité tout au long de ce stage.

Une mention spéciale pour Fouad Allaoui, pour son aide et ses conseils durant toutes les étapes de mon travail.

Table des matières :

Table des illustrations :

1. Introduction

Les systèmes distribués sont une réalité depuis plusieurs années. En offrant la possibilité à des machines réparties sur un réseau d'exploiter des ressources mises à disposition par d'autres machines, les barrières de la localisation et de la disponibilité ont été franchies.
L'évolution des plates-formes à composants distribués : les EJB de Sun ou encore .NET de Microsoft, à donnée naissance à un nouveau mode de développement logiciel, appelé CBD pour *component-based development.* Le CBD soulève de nombreux défis, parmi lesquels la gestion de la distribution, la découverte des composants, l'exécution transactionnelle d'un système à composant, etc. Un problème majeur également soulevé, mais encore résolu que très partiellement est celui du déploiement de ces architectures à composants distribuées.
Le déploiement couvre toutes les activités et leurs interactions lors du cycle de vie d'une application. Parmi ces activités, citons l'installation, l'activation, la désinstallation, la mise à jour, etc.
Une application distribuée est constituée d'un ensemble de composants logiciels qui interagissent de façon complexe et sont, de fait, interdépendants. Pour fonctionner correctement, un composant nécessite d'autres composants de l'application, mais aussi parfois des composants particuliers comme une base de données. Il faut tenir compte de ces dépendances et les exploiter au mieux pour garantir une activation cohérente des divers composants de l'application. C'est ce que nous appelons respect de la logique applicative. Par ailleurs, le processus de déploiement doit tenir compte des contraintes physiques : l'application à déployer est distribuée, les sites du déploiement peuvent être temporairement déconnectés, ils sont sujets aux pannes, etc.

L'objectif de ce stage est de proposer une solution pour le déploiement d'applications réparties en s'appuyant sur l'architecture à composants de SmartTools [1], qui est une architecture orienté service instancier au-dessus de la plate-forme OSGi, en nous nous consacrons à la phase préalable du démarrage d'une application. Cette phase consiste à installer le code des composants, l'instancier, effectuer les liaisons entre les différents composants, et enfin les activer.
Pour prendre en compte la logique applicative, nous utilisons le fichier de description de lancement de SmartTools (***st.xml***). C'est un fichier au format XML qui décrit l'ordre dans lequel les composants vont être activés pour le déploiement d'une application répartie.
Pour prendre en compte les contraintes physiques du déploiement, nous proposons de considérer le processus de déploiement comme une application distribuée à part entière. Cette application utilise le mode d'exécution asynchrone et fiable de SmartTools, qui permet d'envisager le déploiement d'application réparties à grande échelle et facilite la gestion des pannes et des sites temporairement déconnectés.
Ce document est organisé de la façon suivante : dans la section 2, nous présentons le modèle de composants et l'architecture orienté service de SmartTools, dans la section 3 nous présentons quelques infrastructures logicielles pour applications distribuées et nous justifions notre choix de l'infrastructure R-OSGi [7]. La section 4 est consacrée à la description des solutions proposées pour le déploiement d'applications distribuées à base de composants SmartTools et de leur mise en œuvre. Enfin dans la section 5, nous concluons par une analyse du travail réalisé et nous évoquons les différentes perspectives de recherche.

2. SmartTools

SmartTools est un générateur d'environnements de développement basé sur les technologies objets et XML. Grâce à une technique de génération automatique à partir de spécifications, SmartTools permet de développer très rapidement des environnements spécialisés pour des langages de programmation ou pour des langages métiers. En particulier, certaines de ces spécifications (DTD, Schema) sont directement issues des technologies du W3C (World Wide Web Consortium), ce qui donne l'accès à un grand nombre de langages métiers. D'autre part, SmartTools s'appuie sur les technologies objets : implantation en Java, utilisation du patron visiteur, de la programmation par aspects, de la distribution des objets et composants. La combinaison de ces technologies permet de proposer, à moindre coût, une plate-forme de développement ouverte, interactive, uniforme et évolutive [1].
L'architecture de SmartTools est fondée sur l'approche à composants orientés services (SOA). Il possède son propre modèle de composants qui est facilement transposable vers d'autres technologies de composants.
Dans les deux parties qui suivent, nous allons présenter en détails le modèle de composants et l'architecture orientée services de SmartTools.

2.1 Modèle de composants de SmartTools

Dans SmartTools, on utilise un langage spécifique pour décrire les composants. Ce langage est nommé le langage CDML, il est basé sur XML. Chaque composant de SmartTools se modélise par un fichier CDML. On décrit grâce à ce langage les services fournis et les services requis par le composant. A partir de ce fichier de description, SmartTools utilise un générateur de composant pour générer le code des composants (le conteneur et la façade). Au niveau de l'implémentation, un composant SmartTools se compose de deux parties :

Le conteneur

Le conteneur est chargé d'interagir avec l'environnement d'exécution et de réaliser la communication entre les composants. Tous les échanges avec les autres composants sont réalisés dans cette partie. Le mode de communication entre les composants SmartTools est le mode asynchrone. Les composants communiquent entre eux en envoyant des messages.

La façade

C'est la partie métier du composant où tous les services du composant sont implémentés. Les services du composant SmartTools sont décris dans le fichier **cdml**. Dans SmartTools, il y a deux types de services: ***les services d'entrées*** (***Les inputs***: sont des services fournis par le composant) et **les *services de sorties*** (***Les outputs***: sont des services requis, l'implémentation de ces services est réalisée par les autres composants). Au niveau de l'implémentation d'un composant SmartTools, la façade est un élément qui est accessible à partir du conteneur. Une contrainte dans l'implémentation des composants SmartTools est que la façade soit indépendante de son conteneur, c'est à dire qu'il n'y a aucune référence de la façade vers le conteneur. Cette contrainte rend la partie métier (la façade) indépendante de toutes les technologies sur lesquelles on implémente le composant (le conteneur). Donc pour établir la communication entre la façade et son conteneur, dans SmartTools, on utilise le patron de conception "listener". Pour plus de détails sur cette architecture, le lecteur pourra se reporter à [1].

La figure 1 montre un exemple de la description de composant **Graph** (visualisation d'un graphe) et la figure 2 est sa représentation graphique.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" type="graph" extends="abstractContainer">

    <containerclass name="GraphContainer"/>
    <facadeclass name="GraphFacade"/>
    <dependance name="koala-graphics" jar="koala-graphics.jar"/>
    <attribute name="nodeType" javatype="java.lang.String"/>

   <input name="addComponent" method="addNode">
      <parameter name="nodeName" javatype="java.lang.String"/>
      <parameter name="nodeColor" javatype="java.lang.String"/>
   </input>

   <input name="addEdge" method="addEdge">
      <parameter name="srcNodeName" javatype="java.lang.String"/>
      <parameter name="destNodeName" javatype="java.lang.String"/>
   </input>

</component>
```

Figure 1: Description du composant Graph

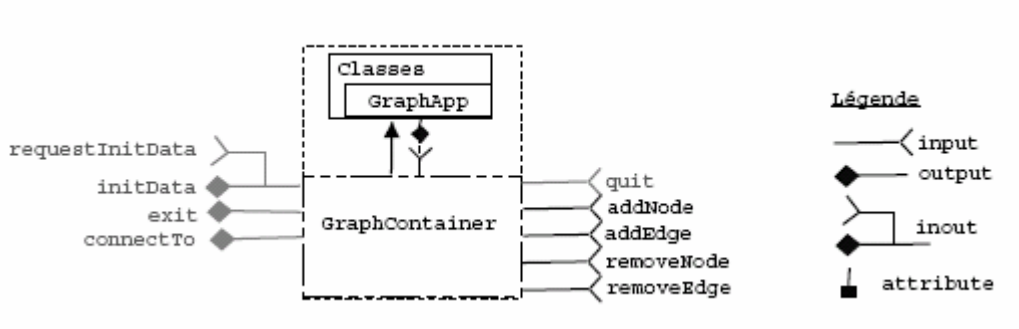

Figure 2: Modèle graphique du composant Graph [1]

2.2 Architecture orientée services de SmartTools

Dans SmartTools, les fonctionnalités de composants sont exposées sous forme des services (les services d'entrées et les services de sorties). Pour gérer les services, SmartTools a introduit une architecture orientée service sur laquelle toutes les échanges entre des composants sont réalisés. L'élément principale de cette architecture est un composant spécial appelé le Gestionnaire de Composants (Component Manager ou CM). Le CM est chargé de gérer le lancement, la mise à jour et l'arrêt des composants. Il lie les composants entre eux à l'aide du service ***connectTo*** qui permet à un composant de demander à être connecté avec un autre composant. Après que les connexions sont établies, les composants communiquent directement entre eux en envoyant des messages.

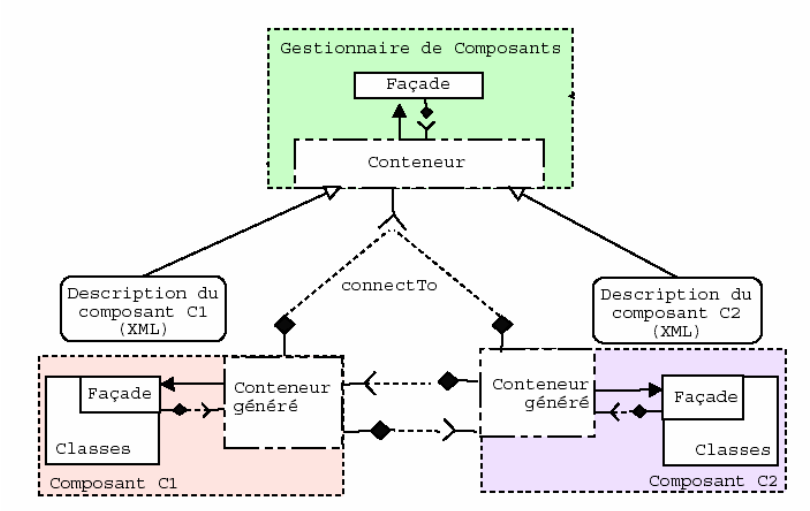

Figure 3: Fonctionnement du Component Manager [1]

Pour construire une application, le CM interprète les actions d'un fichier de description de lancement (***st.xml***). Les principales actions définies dans ce fichier sont :

- chargement d'un type de composant (***load_component***);
- création d'une instance (***start_component***);
- connexion entre deux composants (***connectTo***).

Par exemple, avec le fichier de lancement de la figure 4, le CM charge deux types de composant : db (composant base de donnée) et userDb (utilisateur de la base de donnée) et établit une connexion entre lui-même et une instance de chacun de ces deux composants (cette opération fait appel implicitement à la commande ***start_component***), puis connecte les deux instances crées (***connectTo***) pour qu'elles puissent communiquer directement entre eux.
La commande ***connectTo*** établie automatiquement la connexion dans les deux sens, autrement dit les services d'entrés (les inputs) du premier composant sont relié aux services de sorties (outputs) du deuxième, et inversement. Pour établir ces connections le CM utilise le fichier CDML de chaque composant.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<world
        repository="./bundle"
        debug="ON">

        <load_component jar="db.jar" url="file:./bundle/db.jar" name="db" />
        <load_component jar="userDb.jar" url="file:./bundle/userDb.jar" name="userDb" />

        <connectTo id_src="ComponentsManager" type_dest="db" id_dest="db_1" />
        <connectTo id_src="ComponentsManager" type_dest="userDb" id_dest="userDb_1"/>

        <connectTo id_src="db_1" type_dest="userDb" id_dest="userDb_1" />
</world>
```

Figure 4: Exemple de descriptif de lancement.

3. Infrastructures logicielles pour applications distribuées :

Dans cette section, nous allons passer en revu quelques infrastructures logicielles pour applications distribuées, en particulier les applications basées sur le standard OSGi, vu que notre outil SmartTools et basé sur ce standard.

3.1 Jini

3.1.1 Objectif :

Jini est une technologie développée et distribuée par Sun Microsystems. Elle offre une infrastructure logicielle permettant à des objets Java (services) de ce découvrir et de s'utiliser de façon spontanée.

3.1.2 Concepts et fonctionnement :

Un réseau Jini appelé communauté est constitué des entités suivantes :

- Proxy : c'est un objet Java sérialisable, qui assure la connexion au service est les invocations de méthodes (peut être un stub RMI)
- Service lookup (LUS), les services enregistre leurs Proxy auprès des LUS
- Service : il prépare un Proxy d'accès au service, recherche un LUS et enregistre le service auprès du LUS
- Client : interroge tous les LUS voisins pour obtenir la liste des Proxy, invoque les méthodes sur un Proxy

Les concepts de base de JINI sont :

- ***La découverte*** : c'est un processus par lequel un client trouve un service lookup pour enregistrer ou demander des services, il y a plusieurs protocoles pour ça :
  - Unicat Discovery Protocol : utilisé quand le client connaît le port du LUS et l'adresse IP de la machine quelle l'héberge, dans ce cas l'objet Proxy représentant le LUS est transféré vers le client.
  - Multicast Request Protocol : utilisé lorsque le client veut connaître les LUS voisins, la découverte peut être restreinte à des groupes.
  - Multicast Announcement Protocol : utilisé par le LUS pour annoncer périodiquement sa présence aux clients, chaque annonce contient l'adresse IP et le port du lookup.

- ***L'enregistrement d'un service (join):***
  Lorsqu'un client reçoit le Proxy du lookup sur lequel il souhaite s'enregistrer, il le fait via la méthode « register() » de l'interface « ServiceRegistrar », De cette façon le service client a la capacité de transmettre au lookup son propre proxy.

- ***La recherche d'un service (lookup) :***
  Le recherche d'un service se déroule de la même façon que dans l'enregistrement, en invoquant la méthode « lookup() » sur le Proxy du LUS que nous avant reçu lors de la phase de découverte.

La spécification de Jini n'impose aucun protocole pour la communication entre les services, malgré que l'implémentation de Sun utilise RMI pour contacter le service lookup.

Dans la version 3 (R3) d'OSGi, Jini a été invoquée dans la section « recommended », où ils ont définie l'export des services OSGi vers Jini est inversement, par contre dans la R4 cette recommandation a été supprimée pour les raisons suivantes :

- Aucun des membres de l'alliance OSGi n'a montré un intérêt pour cette recommandation
- Les implications de sécurité qui sont provoquées par Jini n'étaient pas compatibles avec le modèle de sécurité fort d'OSGi
- Les interactions complexes entre les chargeurs de classe ont rendu l'utilisation de Jini encombrante

Etc.…

L'inconvénient de Jini est quelle a besoin d'un LUS, sorte d'annuaire centralisé et aussi elle est assez lourd à mettre en place.

3.2 UPnP (Universal Plug and Play)

3.2.1 Objectif :

L'architecture UPnP a été créée par un consortium d'industriels désirant définir un standard ouvert basé sur IP, permettant de mettre en réseau de manière simple des équipements domestiques et professionnels communicants.
Les éléments de bases d'un réseau UPnP sont : les équipements (Devices), les services et les points de contrôle (Control Point, CP).

Le rôle d'un point de contrôle est:

- Agir sur les équipements
- Réagir au changement d'état des équipements

Le rôle d'un équipement est:

- Fournir des services
- Notifie ces changements d'état

3.2.2 Concepts et fonctionnement :

Le principe de fonctionnement de UPnP est extrêmement simple, il se résume dans les six étapes suivantes :

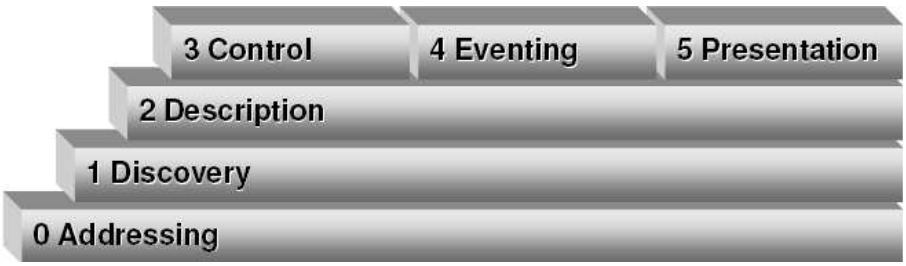

Figure 5: Fonctionnement UPnP [6]

0 – Chaque équipements et points de contrôle obtiennent des adresses pour participer au réseau on utilisant un DHCP ou AutoIP.

1 – Cette phase permet de mettre en relation tous les acteurs du réseau, elle utilise le protocole SSDP (Simple Service Discovery Protocol). Lorsque un équipement est connecté SSDP lui permit de présenter aux points de contrôle sa structure et ces services, alors que pour un point de contrôle qui vient de s'ajouter SSDP lui permit de découvrir tous les équipements et services présents, ou juste un ensemble qui l'intéresse.
L'équipement utilise la méthode « Notify » pour s'annoncer, tandis que le point de contrôle utilise la méthode « M-Search » pour ces recherches.
Les messages d'annonce et de recherche sont des datagrammes qui utilisent une variante multicast de http sur UDP.

2 – Cette phase permet aux points de contrôle d'avoir une description plus détaillée sur l'équipement à partir de l'URL qui a été fournie lors de la phase de découverte, la description est écrite en XML et contient des informations telles que : la marque, le modèle de l'appareil et le numéro de série. On y trouve également la liste des services proposés par l'équipement.

3 – Un point de contrôle peut invoquer une action d'un équipement, pour cela il utilise le protocole SOAP.

4 – Un point de contrôle peut s'abonner pour qu'il soit notifié de la modification des variables d'état d'un équipement (suite à une action ou un changement interne), la notification se fait par l'envoi d'un « event messages » codés en XML et formatés selon l'architecture GENA (General Event Notification Architecture).

5 – Un navigateur examine un équipement via une IHM HTML

L'UPnP est passé du statut recommandation dans la R3 d'OSGi a celui de spécification dans la R4, le chapitre UPnP Device Driver traite la manière de développer des périphériques UPnP et des points de contrôle UPnP au dessus une plateforme OSGi. Ce chapitre décrit d'une part l'API org.osgi.service.upnp dont l'interface principale UPnPDevice représente un périphérique, et d'autre part un élément de la passerelle.
L'UPnP Base Driver, qui assure le pont entre les points de contrôle UPnP et les périphériques UPnP présents sur le réseau, et les bundles hébergés par la plate-forme.

La spécification a comme ambition de permettre les équivalences suivantes :

- Un Bundle OSGi peut jouer le rôle de Control Point.
- Un Service OSGi peut être exporté sur le réseau UPnP en tant que Device.
- Les événements UPnP peuvent être reçus et transmis entre Bundles, Devices et Control Points.

L'inconvénient de UPnP est que les services doivent être des services UPnP, les paramètres des méthodes se restreint aux types supportés par UPnP (pas d'objet Java compliqué).

3.3 R-OSGi

3.3.1 Objectif :

R-OSGi est un projet développé au sein du département informatique de l'*ETH* de Zurich [http://r-osgi.sourceforge.net], et qui vise à fournir une infrastructure logicielle pour l'accès aux services OSGi distribués.

3.3.2 Concepts et fonctionnement :

Dans R-OSGi on retrouve le principe du fournisseur et consommateur d'un service, le fournisseur enregistre le service avec un flag d'accès distant, alors que le consommateur enregistre un écouteur sur ce service, ou bien se connecte explicitement au fournisseur s'il connaît son adresse IP.

- ***Découverte de service*** : R-OSGi utilise SLP (Service Location Protocol) pour la découverte des services, l'avantage de ce protocole c'est qu'il peut fonctionner en deux modes : en utilisant un composant centrale DA (Directory Agent) comme annuaire de service ou bien en utilisant la convergence multicast. Aussi la notion de service pour SLP est proche de celle utilisé dans OSGi, un service SLP est décrit par un *serviceURL* constitué du type de service et une adresse URL.

- ***Génération du Proxy*** : une fois un service est découvert, l'application établie un canal de communication avec le pair distant, et décide de récupérer le service qui répond à ces besoins, dans ce cas un bundle Proxy est généré, en utilisant la librairie ASM [7].

- ***Invocation des méthodes*** : R-OSGi utilise sont propre protocole orienté message (Message Oriented Middleware) pour la communication entre bundles, c'est la communication en mode asynchrone.

R-OSGi offre deux politiques pour la publication d'un service à distance :

- *Service_proxy* : pour créer dynamiquement un Proxy du coté client
- *Transfer_bundle* : pour le transfère du le bundle dans le pair distant, c'est-à-dire créer une copie du bundle contenant le service et l'installer de l'autre coté.

3.4 Notre choix

Notre choix c'est porté sur cette technologie R-OSGi, pour les avantages qu'elle présente :

- Utilise le protocole SLP pour la découverte des services, ce qui va nous permettre d'avoir le choix entre utiliser le mode d'annuaire pour l'enregistrement et la découverte des services ou bien le mode de la convergence multicast.
- La notion de service utilisé dans SLP est proche de celle utilisé dans OSGi, et par la suite de celle utilisé dans SmartTools.
- Utilise le protocole orienté message [7] pour l'invocation des méthodes et la communication entre composants. C'est une approche semblable à celle utilisée dans SmartTools, ce qui va nous permettre de préserver l'aspect de la communication asynchrone.
- Sa taille, pour faire fonctionner R-OSGi nous avons besoin d'activer deux bundles seulement.

4. SmartTools dans un environnement distribué

4.1 Objectif :

Cette section a pour objectif de définir et de mettre en œuvre une architecture qui permettra le déploiement d'applications distribuées à base de composants SmartTools, pour cela on va s'appuyer sur l'infrastructure R-OSGi pour les raisons invoquées dans la section précédente.

4.2 Démarche :

Pour atteindre notre objectif nous avons décomposé le travail en plusieurs phases. Chacune d'entre elle respecte un certain nombre d'hypothèses. À la fin de chaque phase nous présenterons l'ensemble des problèmes de la solution proposée, et les améliorations qu'on peut apporter pour la phase suivante.
Pour toutes les phases on se limite à des applications reparties sur un réseau local, et qui s'exécutent sur une plate-forme stable (les composants restent actifs et ne disparaissent pas).
Notre solution doit respecter les contraintes suivantes de SmartTools :

- privilégier le mode asynchrone pour l'échange des messages entre composants,
- les données doivent être transmises en format XML pour les types non primitif,

On doit aussi veiller à ce que le fonctionnement en répartie n'affecte pas le bon fonctionnement en mode local.

Pour les quatre premières phases, nous avons utilisé Oscar (implémentation open source de la spécification OSGi) comme framework pour l'exécution de nos applications, alors que pour la cinquième phase nous avons utilisé Equinox le noyau OSGi d'Eclipse.

Dans un premier temps, nous avons travaillé avec deux composants de SmartTools très simple :

- Rcmp1 : qui envoie un message updateLabel vers l'autre composant.
- Rcmp2 : qui reçoit un message updateLabel et affichera la valeur (chaîne de caractère) dans une interface graphique.

4.3 Phase 1 : solution basique

Sous un ensemble d'hypothèses fortes, cette phase avait pour objectif de proposer une première solution de base pour l'ajout du caractère distribué aux composants SmartTools via la technologie R-OSGi. Des modifications (corrections) ont étaient apportées à cette dernière pour que l'on puisse avancer dans notre travail. En effet c'est une technologie assez récente et qui n'est pas encore testée sur toutes les implémentations OSGi.

4.3.1 Hypothèses :

- Chaque composant est instancié avec un nom sur une seule machine à la fois.
- Le CM sur une machine n'est responsable que de la connexion dans un seul sens.
- On établi des connexions qu'entre instance de composant et on traitera la connexion entre type de composant par la suite.
- l'instance source de la commande *connectTo* doit toujours exister sur la machine en local.
- On écoute tous les services des machines à distance, sans aucun mécanisme de filtrage.

4.3.2 Spécification et mise en œuvre :

A chaque fois qu'un composant est activé, il publie deux types de service OSGi :

- ***ContainerService*** (pour un type de composant) qui permet :
  - la création d'une instance de ce type de composant, seul le CM local utilise cette fonctionnalité.
  - La récupération du fichier descriptif, sous forme d'un objet Java pour le CM local et sous forme d'un fichier XML pour les CM à distance.

- ***ContainerProxy*** (pour chaque instance d'un type de composant) qui permet de rendre le conteneur du composant visible à distance, via le Proxy qui sera généré par le framework R-OSGi (fonctionnalité offerte par R-OSGi) qui a besoin de ce connecté à distance à ce composant.

Le composant manager publie un service ***DiscoveryListener***, pour l'écoute de ces deux interfaces (services) ***ContainerService, et ContainerProxy.***

Etant donné deux machines M1 et M2 avec les deux fichiers de lancement ***st.xml*** décrite dans la figure 1 et 2 respectivement. Le CM de la machine M1 active le composant rcmp1 et crée une instance rcmp1-1, alors que le CM de la machine M2 active le composant rcmp2, crée une instance rcmp2-1 et fait appel à une nouvelle commande ***connectToRemote*** qui accepte deux paramètres. Le premier paramètre est l'instance locale et le deuxième est l'instance à distance.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<world
      repository="./bundle"
      debug="ON">

      <load_component jar="rcmp1.jar" url="file:./bundle/rcmp1.jar"
            name="rcmp1" />

      <connectTo id_src="ComponentsManager" type_dest="rcmp1"
            id_dest="rcmp1-1" />
</world>
```

Figure 6: Fichier de lancement de la machine M1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<world
      repository="./bundle"
      debug="ON">

      <load_component jar="rcmp2.jar" url="file:./bundle/rcmp2.jar"
            name="rcmp2" />

      <connectTo id_src="ComponentsManager" type_dest="rcmp2"
            id_dest="rcmp2-1" />

      <connectToRemote id_src= 'rcmp2-1' type_dest='rcmp1' id_dest='rcmp1-1' />
</world>
```

Figure 7: Fichier de lancement de la machine M2

Une fois le CM de la machine M2 est notifié de la découverte des deux services ***ContainerService et ContainerProxy,*** il crée le Proxy qui correspond à chaque service. Puis récupère le fichier descriptif CDML en format XML, pour la liaison entre l'instance locale et le Proxy de l'instance distante. La commande ***connectToRemote,*** n'effectue la connexion que dans un seul sens, c'est l'instance locale qui interroge l'instance distante via son proxy.

La figure 8 représente la communication entre les deux composants rcmp1 et rcmp2 via le Proxy qui a été crée sur la machine M2.

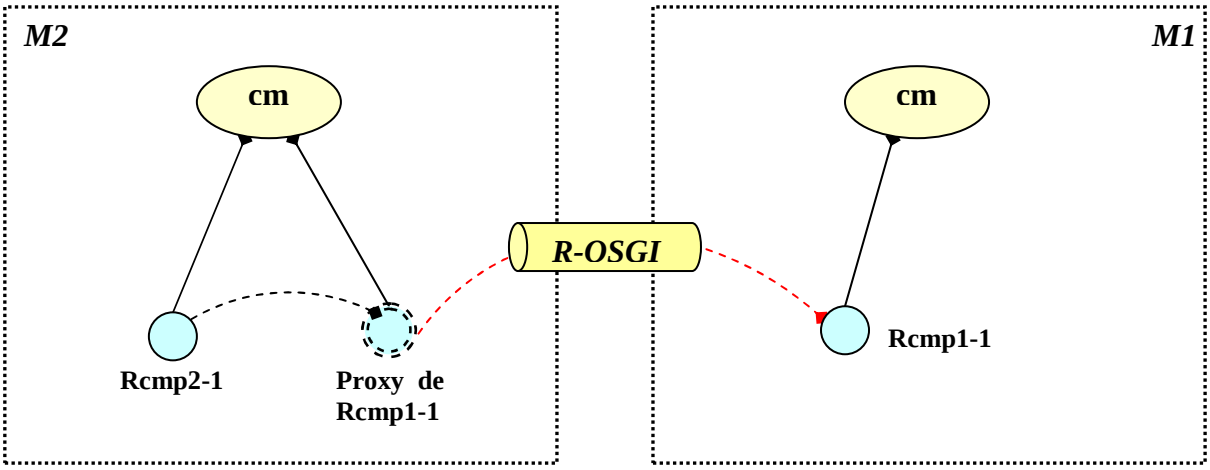

Figure 8: Communication entre deux composants à distance

4.3.3 Evaluation :

Bien que la communication entre les deux composants fonctionne, cette solution reste une solution basique et présente un inconvénient majeur : sur chaque machine, on se retrouve avec un ensemble de bundle Proxy, installé dans le framework et qui ne serve à rien. Ceci est dû au faite que l'interface ***ContainerProxy*** est générique, valable pour tous les composants. Il faut donc générer le Proxy pour avoir accès aux informations sur le type du composant et le nom de l'instance. Ce problème est dû aussi au faite que nos composants managers écoutent tous les services sans filtrage.

4.4 Phase 2 : ajout d'un mécanisme de filtrage

Dans cette phase, nous allons implémenter un mécanisme de filtrage par nom d'instance, ce qui va nous permettre de relâcher la première hypothèse de la phase 1. On va considérer q'un composant peut être instancier plusieurs fois, et sur plusieurs machines à la fois.
Un mécanisme de filtrage est prévu dans R-OSGi, via le champ ***FILTER*** de l'interface ***DiscoveryListener,*** mais jusqu'à la version actuelle (*0.6.1*) il n'est pas encore implémenté. Ce qui nous a amené à ajouter cette fonctionnalité au code de R-OSGi. Pour cela nous avons ajouté à l'URL du service (***ContainerProxy***) diffusé des informations supplémentaire (comme le nom de l'instance du composant).

4.4.1 Hypothèses :

- Une machine n'est responsable que de la connexion dans un seul sens.
- Connexion entre instance de composant et non pas par types de composant, l'instance source existe (activer) toujours en local.

4.4.2 Spécifications et mise en œuvre:

Chaque composant va publier le service ***ContainerService*** avec une propriété qui indique le type du composant ainsi que le nom de l'instance. De son coté le composant manager distant va effectuer un filtrage sur les URLs qu'il reçoit. Autrement dit, il ne doit faire appel au processus de création du Proxy que pour les URLs qui porte la propriété qui l'intéresse.
Pour le service ***ContainerProxy***, il sera publié avec la propriété du type du composant seulement, puisque toutes les instances du même type de composant ont le même fichier de description. Par suite le fichier CDML d'une instance sera utilisé pour toute les autres instances du même type de composant. On n'a donc pas besoin de récupérer à chaque fois.

4.4.3 Evaluation :

Sous les hypothèses ci-dessus, cette solution reste la plus optimale, seul les Proxy relatifs aux instances qui nous intéressent, seront crées. Malgré ça, un autre problème persiste c'est dans le cas ou deux machines possèdent deux instances avec le même nom, la communication ne sera faite qu'avec l'instance la première découverte. Ceci est du au faite que le composant manager utilise le nom de l'instance comme clé d'enregistrement dans l'annuaire des composants démarrés.

4.5 Phase 3 : connexion par type de composant

Dans cette phase, on va essayer d'ajouter un mécanisme qui permettra à l'utilisateur de choisir entre se connecter à toutes les instances d'un type de composant donné (indépendamment du nom de l'instance), ou de se connecter à une seule instance (la première découverte) de ce type de composant.

4.5.1 Hypothèses :

- l'instance source existe toujours et en local.
- Le type de composant destination est connu par au moins un composant manager distant.

4.5.2 Spécifications et mise en œuvre :

Il s'est avérai que pour un type d'application donné, on a besoin de ce connecter à l'ensemble des instances d'un type de composant, ou bien au contraire se connecter à une et une seule instance d'un type de composant, peu importe son nom et sa localisation. Or avec les mécanismes que nous possédons, nous sommes toujours obligés d'indiquer le nom de l'instance destinataire. Pour cela on a essayé d'introduire de nouvelles notations (attributs) dans l'instruction ***connectToRemote***:

```
<connectToRemote id_src= 'rcmp2-1' type_dest='rcmp1' id_dest='*' />
```

id_dest='*': indique au composant manager, que nous souhaitons nous connecter à toutes les instances du même type de composant **'rcmp1'**, toutes les instances actives et celles qui seront crées par la suite.

```
<connectToRemote id_src= 'rcmp2-1' type_dest='rcmp1' id_dest='!' />
```

id_dest=’ !’: indique au composant manager, que nous souhaitons nous connecter à une et une seul instance du type de composant **’rcmp1’**, peu importe le nom et la localisation de l’instance. Autrement dit on se connecte à la première instance découverte.

4.5.3 Evaluation :

Les notations introduites lors de cette phase, nous ont permis de traiter tous les cas possible de connexion à une instance d’un type de composant donné. Il reste à résoudre le problème quand aucune instance n’existe, où le type du composant n’est connu par aucun des composants managers distants. C’est l’objectif de la phase suivante.

4.6 Phase 4 : communication entre composants managers

Pour les phases précédentes, nous avons supposé que le type de composant qu’on veut utiliser à distance est connu par au moins une machine. C’est à dire que le bundle correspondant est déjà activé. Or cette hypothèse n’est pas toujours valable. Chaque machine possède plusieurs composants, et n’active que ceux dont elle a besoin pour ses propres applications.
Le premier objectif de cette phase est d’introduire la possibilité qu’un CM peut demander aux autres machines de lui activer un type de composant donné, si elles le possèdent dans leur base des composants. Le deuxième objectif, c’est de permettre à un composant manager de forcer la connexion dans l’autre sens si l’application en a besoin. Pour réaliser ceci, une communication entre composants managers c’est avéré indispensable.

Nous avons opté pour une connectivité forte entre les composants managers. C’est à dire dès qu’une machine intègre le réseau, son CM essaye d’établir une connexion avec tout les CMs des machines déjà présentes sur le réseau. La connexion entre deux composants mangers est faite dans les deux sens, vu que chaque CM une fois qu’il détecte le service ***ContainerProxy*** avec le paramètre ‘composant manager’ comme type de composant, il crée automatiquement le Proxy correspondant.
Un CM, pour SmartTools est un composant comme un autre (un fichier CDML lui est associé), on peut lui appliquer les même fonctionnalités (un ***connectToRemote*** entre deux CMs est possible sans aucune modification).

Cette approche permet à chaque machine dans le réseau de disposer de toutes les ressources (composants) des autres machines, comme si elles étaient des ressources locales, et ceci par une simple communication entre les CMs.

La figure 9 représente la communication entre composants managers. Sur chaque machine en trouve une base des composants disponibles, et chaque CM dispose d’un annuaire des bundles activés.

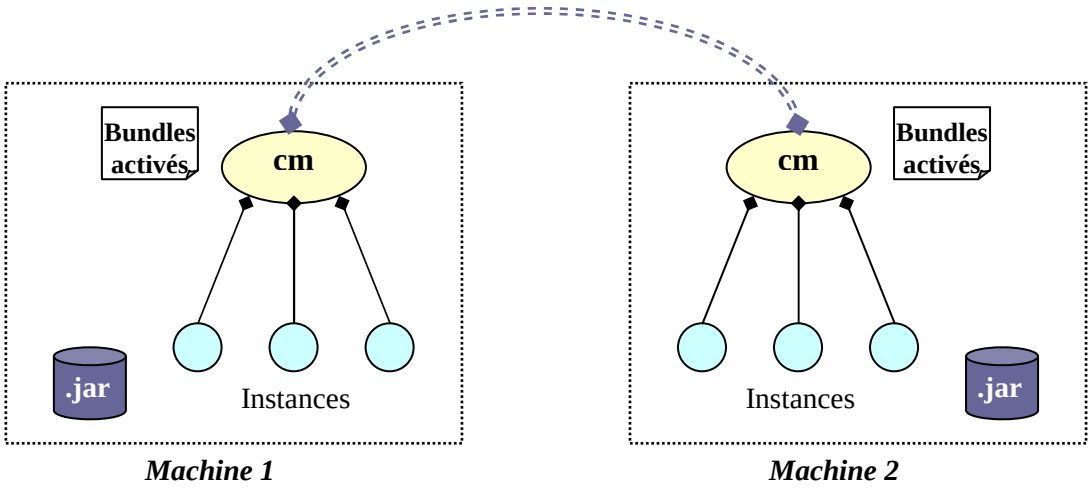

Figure 9: Communication entre CMs

4.6.1 Hypothèses :

- l'instance source existe toujours et en local
- les composants managers sont fortement connectés.

4.6.2 Spécifications et mise en œuvre :

Comme nous l'avons déjà évoqué auparavant, la commande ***connectToRemote*** ne permet que la connexion dans un seul sens, car le Proxy installé dans notre framework local, est un composant passif. Il ne fait que transmettre les appels de fonction au composant distant. Or dans certaines applications, un composant peut avoir besoin d'une connexion dans les deux sens avec le composant à distance. Ce qui revient à dire que nous avons besoin qu'une commande ***connectToRemote*** soit exécutée depuis la machine distante, avec comme ***id_dest*** notre instance locale.
Pour répondre à ce besoin, nous avons défini un nouvel argument pour la commande ***connectToRemote***. L'attribut '***dc***' qui prend deux états '*ON*' où '*OFF*', indique au composant manger qu'on a besoin d'une double connexion ou non.

```
<connectToRemote id_src= 'rcmp2-1' type_dest='rcmp1' id_dest='rcmp1-1' dc='ON' />
```

Une fois que le composant manager a récupéré le Proxy de l'instance à distance, et si l'argument ***dc*** de la commande ***connectToRemote*** est positionné à '*ON*', il doit invoquer la méthode ***connectToRemote*** sur le CM distant (responsable de l'instance distante), en lui passant comme paramètres : **id_src** correspondant au nom de l'instance distante et **id_dest** correspondant au nom de l'instance locale, concaténer avec l'adresse IP de la machine locale. Ceci pour éviter tout conflit avec d'autre instance qui porte le même nom sur d'autre machines.

Pour permettre l'appel de la méthode ***connectToRemote*** entre composants managers, cette dernière a était ajouter au fichier descriptif (***cm.cdml***) pour le type de composant « composant manager ». C'est à dire comme service fourni et requit au sens SmartTools du terme.

La figure 10 représente un extrait du fichier descriptif du composant manager. Nous avons ajouté la fonction ***connectToRemote*** comme service fourni (output) et requit (input).

```
<output name="connectToRemote" method="connectToRemote"
            doc="connectToRemote" >
   <arg name="id_src" doc="" javatype="java.lang.String"/>
   <arg name="type_dest" doc="" javatype="java.lang.String"/>
   <arg name="id_dest" doc="" javatype="java.lang.String"/>
   <arg name="dc" doc="" javatype="java.lang.String"/>
   <arg name="actions" doc="" javatype="java.util.HashMap"/>
</output>
```

```
<input doc="connectToRemote" method="connectToRemote"
                name="connectToRemote">
   <attribute doc="" javatype="java.lang.String" name="id_src"
      ns="fr.smarttools.core.component"/>
   ...
</input>
```

Figure 10: Extrait du fichier cm.cdml

Le deuxième objectif de cette phase, c'était de permettre à un composant manager de demander aux autres CM avec lesquels il est connecté, de lui crée une instance d'un type de composant donné, si ces CMs connaissent ce type de composant. C'est à dire s'il apparaît dans leurs annuaires des composants démarrés, sinon ils vérifient dans leur ***repository*** (base des composants) s'ils ont un composant de ce type là, puis le démarre.
La demande de la création de l'instance auprès des autres composants managers est effectuée par l'envoi d'un message ***startComponent*** (avec comme paramètres le type du composant et le nom de l'instance) à tout les CMs connectés. Puis ceux qui connaissent ce type de composant vont répondre (en activant ce type de composant).

Pour garder l'aspect de la communication asynchrone de SmartTools (sans attendre le résultat), un composant manager qui n'arrive pas à trouver une instance ou un type de composant donné, il envoi un message (***startComponent***) aux autres CMs connectés pour demander la création de cette instance ou le démarrage du composant désiré. Puis il enregistre la commande courrante (***connectToRemote***) dans une liste (***nextWord***) des commandes non encore résolue (en attente), puis continue l'exécution des autres commandes.
Les commandes de la liste ***nextWord*** sont réexécuté à chaque fois qu'on est notifié de la connexion d'un nouveau composant manager, ou bien la création d'une nouvelle instance. La liste est mise à jour au fur et à mesure de la disponibilité du type de composant désiré.

4.6.3 Evaluation :

A l'issu de cette phase, on peut dire qu'on a élaboré une vraie architecture pour l'interaction entre composants SmartTools en répartie, en respectant les caractéristiques spécifiques de SmartTools et les contraintes de conception qu'on a posé au début.
C'est sur cette architecture qu'on va se baser pour aborder la phase suivante, à savoir la construction des RCPs réparties.

4.7 Phase 5 : construction des RCPs réparties

L'architecture orienté service de SmartTools a était intégré au-dessus de la plateforme Eclipse. L'objectif de cette phase était de tester la solution de déploiement de composants distribués proposée précédemment, dans le contexte Eclipse (avec le noyau Equinox). Et aussi de voir si le passage à Equinox n'introduisait pas de nouvelles contraintes.

4.7.1 Spécifications et mise en œuvre :

Nous avons essayé de faire communiquer à distance des RCPs à base de composants (plugins) SmartTools. Notre démarche était de créer une RCP basique en utilisant les deux composants rcmp1 et rcmp2 présentés précédemment.
Le long de cette phase nous avons utilisé Equinox (noyau d'Eclipse) comme framework pour l'exécution de nos RCPs.
Le seul problème que nous avons rencontré, était que R-OSGi ne supporte pas l'instruction ***Require-Bundle (particulier à Equinox)*** dans le fichier ***manifest*** de nos plugins (bundles), donc on était obligé de la remplacer par l'instruction équivalente ***Import-Package***.

4.7.2 Evaluation :

Cette phase nous a permis de valider notre architecture distribuée, pour des applications construites par assemblage de plugins SmartTools, au-dessus des RCPs d'Eclipse.

5. Conclusion :

L'objectif de ce travail était de proposer une solution pour le déploiement d'applications distribuées à base de composant SmartTools, nous avons choisi l'infrastructure logicielle R-OSGi comme support de distribution. L'avantage de cette technologie est qu'elle permet l'exécution repartie de bundle (composant) conforme au standard OSGi. Comme notre architecture SmartTools est basée sur ce standard, il était naturel d'utiliser cette technologie. Cette bibliothèque nous a permis d'atteindre notre objectif avec peu de modifications et un effort de développement minimal.

L'architecture de déploiement proposée présente plusieurs caractéristiques intéressantes. D'une part, il n'y a pas de contrôle centralisé du déploiement vu que nous avons opté pour une recherche de services par diffusion au lieu d'utiliser un annuaire centralisé pour la publication et la recherche de services. Cela permet d'éviter la création de goulot d'étranglement et facilite le passage à l'échelle de l'approche. Par ailleurs, l'exécution asynchrone des différentes activités garantit une activation au plus tôt de l'ensemble des composants déployés.

La solution proposée présente quelques limites :

- Nous n'avons pas prévu un mécanisme de gestion de pannes. Ce mécanisme n'est pas difficile à mettre en œuvre car une fois qu'un service est perdu à cause d'une panne (réseau ou machine) l'application est notifié (service OSGi de base). Il ne reste donc qu'à déclencher le processus de recherche d'un autre service similaire et de détruire les Proxy relatifs au service perdu.
- Nous n'avons pas spécifié un espace de nommage pour les applications (définition des réseaux de machine). On écoute tous les services au lieu des services d'une application donnée.
- Nous avons opté pour une connexion forte entre les composants managers, ce qui engendre une grande quantité de messages échangés inutiles.
- Le mécanisme de filtrage que nous avons ajouté à R-OSGi est très basique, nous attendons que R-OSGi intègre un mécanisme plus puissant dans sa prochaine version.

Nous envisageons comme perspectives de notre travail, l'étude des problèmes suivants :

- R-OSGi et SmartTools utilise le même traitement pour l'échange de message (le codage des appels de méthodes) entre composants, donc il faudrait réfléchir à réduire le code pour éviter de faire la même chose deux fois à chaque appel.
- Adapter notre architecture pour arriver à déployer des applications distribuées dans des environnements partiellement connectés et sur des équipements hétérogènes.
- Aborder l'aspect sécurité des communications entre composants.
- Mettre en place une stratégie de recherche et de sélection des ressources plus riches.
- Mettre en place des mécanismes d'équilibrage de charge entre les différents nœuds du réseau.

Nous pensons que le travail effectué durant ce stage, donne des éléments de base pour aborder l'étude de l'ensemble des problèmes restants et de généraliser assez facilement notre approche.

Bibliographie :

[1] Carine Courbis, Pascal Degenne, Alexandre Fau and Didier Parigot. « Un modèle abstrait de composants adaptables » *revue TSI, Composants et adaptabilité, 23(2), 2004*

[2] Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot and Joseph Variamparambil. « Un modèle de composants pour l'atelier de développement SmartTools » *In* Systèmes à composants adaptables et extensibles, *Octobre 2002*

[3] Vincent Hourdin, Stéphane Lavirotte, Jean-Yves Tigli. « Comparaison des systèmes de services pour dispositifs » *rapport de recherche ISRN I3S/RR-2006-25-FR*

[4] Didier Donsez. « La plate-forme dynamique de services OSGi », *Intergiciel et Construction d'Applications Réparties, version du 08 janvier 2007*

[5] Didier Donsez. « Jini », Université Joseph Fourier IMA –IMAG/LSR/ADELE

[6] Didier Donsez. « Mise en oeuvre d'UPnP avec OSGI », *Université Joseph Fourier IMA –IMAG/LSR/ADELE*

[7] Jan S. Rellermeyer, Gustavo Alonso. « Services Everywhere: OSGi in Distributed Environments » *Department of Computer Science ETH Zurich*

[8] http://r-osgi.sourceforge.net/