

Intégration négociée de services dans les systèmes distribués

Noha Ibrahim, Frédéric Le Mouël, Stéphane Frénot

INRIA Ares, laboratoire CITI, INSA Lyon
Bâtiment Léonard de Vinci, 20 avenue Albert Einstein
69621 Villeurbanne Cedex
{noha.ibrahim, frederic.le-mouel, stephane.frenot}@insa-lyon.fr

Résumé

Les fournisseurs d'accès à Internet ont de plus en plus tendance à fournir des ensembles de services (WiFi, voie sur IP, télévision sur IP). Les applications s'exécutant dans ces environnements dynamiques doivent pouvoir intégrer les services fournis par différents fournisseurs et présents partout dans l'environnement. Dans cet article, Nous proposons un système d'intégration de services. Ce système fournit un cadre de conception (*framework*) composé d'un ensemble d'interfaces - *Integrable*, *Negotiable*, *IntegrationLifeCycle* - et d'un ensemble d'outils (*toolkit*) implémentant ces interfaces. Ces outils offrent différentes techniques d'intégration (composition locale ou distante, tissage locale ou distant, déploiement), protocole de négociation par contrats et gestion du cycle de vie de l'intégration. Nous démontrons la faisabilité de notre système en implémentant un prototype basé sur la plateforme Java et la technologie OSGi¹.

Mots-clés : services, intégration, négociation, OSGi, systèmes distribués

1. Introduction

L'intégration de service se définit comme le problème de combiner différents services en un seul offrant une nouvelle fonctionnalité [21]. Cette problématique suscite de plus en plus d'intérêt dans plusieurs communautés et cela pour plusieurs raisons. Tout d'abord, le nombre de fournisseurs offrant des services nouveaux aux applications est en forte expansion [12]. Le développement de nombreux environnements distribués hautement dynamiques, comme les environnements pervasifs ou les grilles de calcul, modifie le comportement d'exécution des applications. L'intégration de service réduit le temps et l'effort des applications à chercher les services requis. En combinant les services, les applications utilisent ainsi de nouvelles fonctionnalités provenant de différents fournisseurs [21].

L'intégration de service est ainsi un défi, à cause de la diversité des services fournis dans les environnements distribués. L'intégration demande souvent la maîtrise de plusieurs technologies et techniques. Notre objectif est de fournir un système d'intégration de services adapté aux environnements distribués. Ce système fournit un cadre de conception (*framework*) composé d'un ensemble d'interfaces - *Integrable*, *Negotiable* *IntegrationLifeCycle* - et d'outils (*toolkit*) implémentant ces interfaces. Ces outils rendent possible l'application, de différentes techniques d'intégration (composition locale ou distante, tissage locale ou distant, déploiement), de protocole de négociation par contrats et de gestion du cycle de vie de l'intégration.

Actuellement, nombre de cadres de conception existent pour intégrer les services mais ces cadres sont destinés, très souvent, à un modèle précis de service et imposent une technique d'intégration [17] [14] [21]. Notre cadre de conception est indépendant des outils qui l'implémentent, et une application peut très bien utiliser notre cadre avec ses propres outils.

¹ Ce travail est partiellement financé par le projet européen IST Amigo-Ambient Intelligence for the Networked Home Environment [6].

Nous avons démontré la faisabilité de notre système en implémentant un prototype basé sur la plateforme Java et la technologie OSGi. Les outils que nous proposons sont spécifiques aux choix que nous avons fait. Le choix de Java a été motivé par sa portabilité et sa capacité à fournir une forte séparation entre les interfaces et leurs implémentations. Cette séparation reflète la séparation que nous avons faite entre notre cadre de conception et les outils l'implémentant. OSGi [1] a été choisi pour sa capacité d'Inversion de Control (IOC) [16]. OSGi simplifie le développement, le déploiement et la gestion des services en découplant les spécifications du service de son implémentation.

La force de notre système est dans son cadre de conception qui peut s'adapter aux plateformes existantes (.NET, OSGi, Fractal, J2EE, etc.). La limite de la version actuelle de notre système est dans ses outils ; les applications désirant utiliser ces outils doivent supporter la machine virtuelle Java et la plateforme OSGi.

La section 2 liste les propriétés de notre intégration de services. La section 3 présente l'architecture de notre système. La section 4 détaille l'implémentation de notre prototype. La Section 5 examine les travaux liés à la problématique d'intégration de services. La Section 6 conclue et présente les perspectives de ce travail.

2. Propriétés de l'Intégration

L'intégration que nous proposons possède trois propriétés [15]. Nous considérons ces propriétés comme essentielles pour les environnements distribués. D'autres propriétés peuvent être intéressantes comme l'interopérabilité ou l'adaptabilité, mais nous considérons qu'elles sont plus spécifiques à un certain domaine d'applications comme les environnements mobiles ou pervasifs. Suivant les futurs domaines d'application, ces propriétés seront étudiées ultérieurement.

L'intégration vérifie les trois propriétés suivantes :

- Généricité : L'intégration doit pouvoir être utilisée par toutes sortes d'applications désirant intégrer des services dans ses propres services ou au contraire désirant charger ses services dans n'importe quel environnement.
- Cycle de vie : L'intégration peut avoir une durée de vie précise. Ce cycle de vie permet de décrire son existence et son évolution dans le temps. Un service résultant d'une intégration peut donc durer un certain moment et disparaître une fois son cycle atteint. Cette propriété permet aux applications intégrant les services pour un but, de les voir disparaître une fois le but atteint.
- Réversibilité : Comme un service intégré doit disparaître une fois son cycle atteint, l'intégration doit être réversible. Nous signifiions par réversible la faculté de revenir à l'état précédant l'intégration. Le service résultant de l'intégration disparaît et les services qui étaient intégrés sont toujours accessibles. Notre intégration de services vérifie une quatrième propriété non liée aux environnements distribués, l'efficacité. Nous proposons une négociation entre les services avant leur intégration, rendant l'intégration plus stable et efficace. Cette propriété est optionnelle. Les services peuvent ne pas négocier avant de s'intégrer. Les outils que nous avons développés offrent un protocole de négociation par contrats, mais d'autres outils peuvent implémenter notre cadre de conception sans proposer un protocole de négociation.

3. Architecture du cadre de conception de l'intégration

Cette section introduit l'architecture de notre cadre de conception (*framework*). Nous considérons que tous les services sont présents dans une couche middleware fournissant un environnement d'exécution. Nous y définissons un service spécial *IntegServ*. Ce service réside sur chaque machine désirant intégrer des services. L'intégration des services, qu'elle soit distante ou locale, est gérée par le service *IntegServ*. Nous allons tout d'abord définir notre modèle de service et ensuite détailler l'architecture du service *IntegServ*.

3.1. Modèle de Service

Notre modèle de service est constitué de trois parties (cf. figure 1) :

- Les interfaces : Une interface spécifie les méthodes qu'il est possible d'exécuter sur un service. Les interfaces d'un service sont publiques. Un service peut avoir deux types d'interfaces : les interfaces

fonctionnelles définissant les fonctionnalités rendues pas le service et les interfaces de gestion de l'intégration du service (par exemple, l'interface `Integrable` définit la manière d'intégrer le service).

- Les liens établis : Un service fournit des fonctionnalités et a également besoin d'autres pour pouvoir s'exécuter. Les liens établis (ou *bindings*) expriment ces dépendances construites statiquement au développement ou dynamiquement à l'exécution.
- Les objets fonctionnels : Ces objets réalisent les fonctionnalités attendues du service.

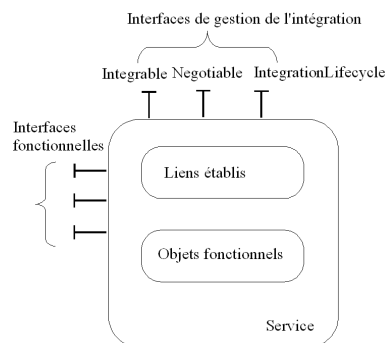


FIG. 1 – Modèle de service

Notre modèle de service est indépendant du choix de son implémentation et peut être appliqué aux EJBs [17], composants CORBA [26], composants Fractal [3], bundles/services OSGi [18] et Services Web (.NET) [9].

Les interfaces sont techniquement exprimées en utilisant un langage de programmation comme le langage Java (les interfaces java) ou en utilisant un langage de description d'interface (IDL, *Interface Description Language*), comme dans CORBA [26]. Les liens établis peuvent être exprimés par un langage de description d'architecture (*Architecture Description Language ADL*) [22]. L'implémentation des objets fonctionnels est dépendante du langage de programmation utilisé et résulte de l'instanciation des classes.

3.2. Architecture du service *IntegServ*

Plusieurs services s'exécutent sur différentes machines réparties dans un environnement distribué (cf. figure 2). Nous considérons deux types de services : les services basiques et les services intégrés. Un service qui veut pouvoir intégrer un autre service de l'environnement doit implémenter l'interface `Integrable`. Un service qui veut pouvoir négocier avec un autre service doit implémenter l'interface `Negotiable`. Tous les services intégrés implémentent l'interface `IntegrationLifeCycle`. Le service responsable de l'intégration dans l'environnement est le service *IntegServ*. Lorsqu'un appel à une méthode de l'interface `Integrable` d'un service, est effectué, cet appel est redirigé vers un *IntegServ* local (figure 2 étape 1) ou, si aucun ne s'exécute sur la machine, vers un *IntegServ* distant. Un service de découverte prend en charge la tâche de chercher et trouver un *IntegServ* dans l'environnement (figure 2 étape 2). Notre système ne fournit pas le service de découverte mais en utilise des existants comme le service de découverte de Jini [13] ou UPnP [5].

Le service *IntegServ* applique le modèle de réflexivité sur lui même. Il résulte lui même de l'intégration de trois services : le service technique d'intégration, le service de gestion de la négociation et le service de gestion du cycle de vie de l'intégration.

3.3. Service technique d'intégration

Le service technique d'intégration permet d'appliquer différentes méthodes d'intégration ciblant les parties internes du service (i.e. liens établis et objets fonctionnels) mais aussi les parties externes (i.e. interfaces). Ces différentes méthodes sont applicables l'une avant l'autre et peuvent être combinées.

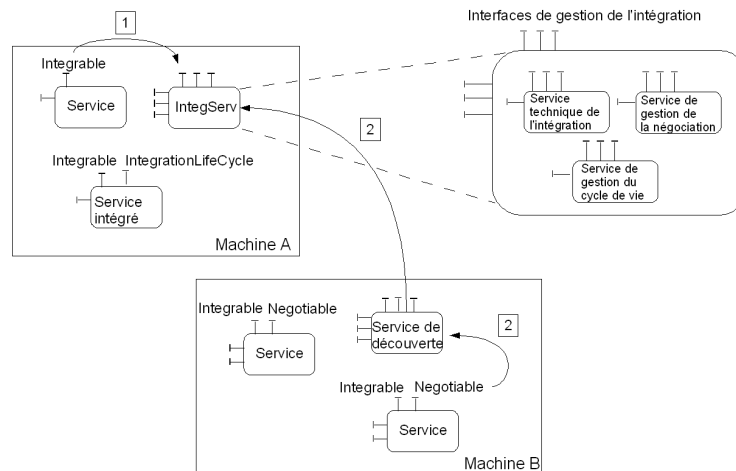


FIG. 2 – Architecture du service *IntegServ*

Elles ont toutes pour cible un ensemble de services. Le service technique d'intégration fournit trois techniques d'intégration :

- La composition de services est une technique d'intégration des services ciblant les interfaces. Le service *IntegServ* définit deux moyens pour connecter les services à travers leurs interfaces. La première est de republier toutes les interfaces des services à intégrer. Le service composé fournit ainsi toutes les interfaces des services participant à l'intégration (cf. figure 3). Le deuxième moyen est de fournir une seule interface qui redirige vers toutes les autres interfaces des services. Cette méthode est utilisée si les sorties d'une interface sont compatibles avec les entrées d'une autre, formant ainsi une chaîne d'interfaces compatibles (cf. figure 4). Bien sûr une combinaison de ses deux méthodes est possible.
- Le tissage des services permet d'entrelacer le code d'un objet d'un service, dans le code des objets d'un autre service. Dans cette technique d'intégration, seuls les objets fonctionnels des services sont concernés (cf. figure 5). Le service tissé publie les mêmes interfaces fonctionnelles qu'il possédait avant le tissage, mais la fonctionnalité des méthodes de l'interface a bien changé.
- Le déploiement des services permet la distribution des services dans l'environnement en les téléchargeant sur différentes machines. Les liens établis du service sont alors concernés. Une fois un service déployé sur une machine, de nouveaux liens sont créés et les anciens détruits (cf. figure 6).

Rien n'empêche d'ajouter d'autres techniques d'intégration au service technique d'intégration. La seule condition est de respecter le modèle de service défini.

3.4. Service de gestion de la négociation

Nous définissons la négociation comme le processus d'arriver à un accord entre deux services sur les termes et conditions de l'intégration. Nous distinguons trois parties dans le processus de négociation : les contrats, les stratégies et les phases de négociation.

Un Contrat est composé de trois parties essentielles [10] :

- La description du contrat. Un Contrat [4] synthétise les termes et conditions de la négociation dans des clauses.
- Les clauses du contrat. Les clauses contiennent des éléments mesurables et quantifiables, comme la qualité de service QoS ou la mémoire. Nous enrichissons ces clauses avec des paramètres de l'intégration comme la durée de l'intégration.
- Les pénalités à appliquer en cas de non respect du contrat.

Notre cadre de conception définit deux types de stratégies : les stratégies services et les stratégies machines. La stratégie service prend en compte le service en question et analyse les contrats qu'il possède déjà. Cette stratégie se trouve dans le service. La stratégie machine se place à un niveau plus globale et analyse les contrats de tous les services présents sur une machine. Ces deux stratégies sont complémen-

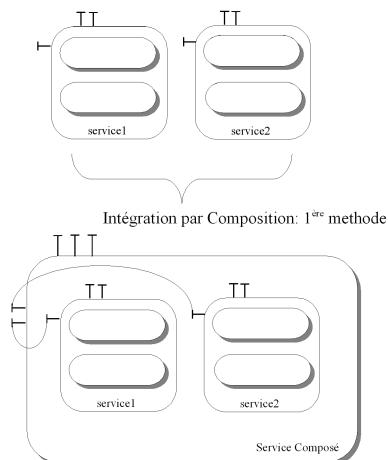


FIG. 3 – Composition de services : première méthode

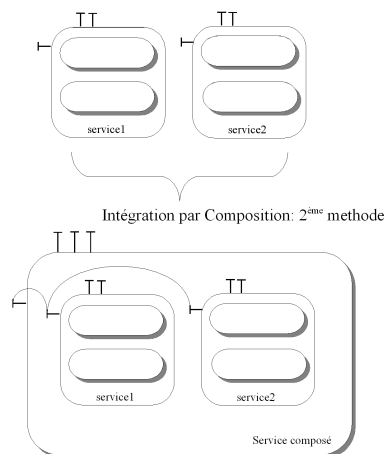


FIG. 4 – Composition de services : deuxième méthode

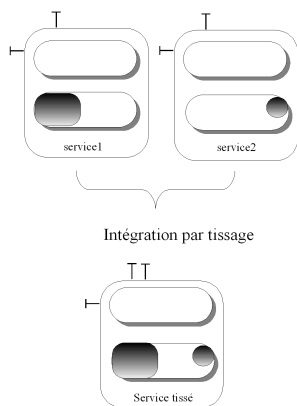


FIG. 5 – Tissage de services

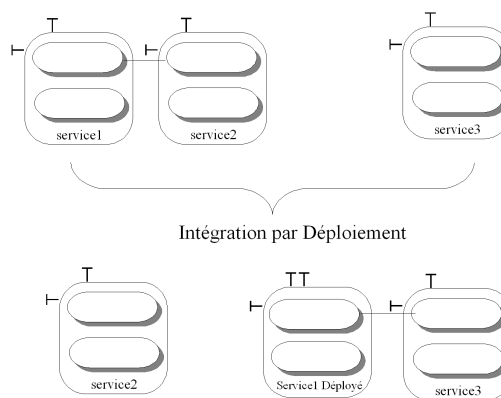


FIG. 6 – Déploiement de services

taires. En effet, il ne suffit pas de savoir si un service est capable ou non de contracter une ressource, il faut être sûr que cette ressource est suffisante et non utilisée par un autre service de la même machine. La stratégie machine est gérée par le service de gestion de la négociation. Si par exemple la ressource à contracter est la mémoire, la stratégie service va prendre en compte seulement la mémoire consommée par le service alors que la stratégie machine va prendre en compte la mémoire globale restante sur la machine.

La négociation est divisée en trois phases : l'initialisation, la négociation active et la destruction.

- **L'initialisation** : Les occurrences sont uniques et se produisent dans cet ordre précis :
 - proposer un contrat au service que l'on désire intégrer.
 - analyser le contrat.
 - accepter/refuser le contrat. Le contrat est signé si la négociation aboutie sinon une autre proposition devra être faite par le contractant (retour à l'étape 1).
- **La négociation active** : les occurrences sont multiples :
 - vérifier la durée de vie du contrat et notifier tout changement.
 - renégocier le contrat.
- **Destruction** : Les occurrences sont uniques :
 - invalider le contrat.

3.5. Service de gestion du cycle de vie de l'intégration

L'intégration peut avoir un cycle de vie défini et géré par le service de gestion du cycle de vie. Une fois ce cycle atteint le service intégré doit disparaître laissant place aux services initiaux. Tous les services intégrés implémentent l'interface *IntegrationLifeCycle*.

Le cycle de vie des services intégrés est composé de trois parties : l'initialisation, le service intégré actif et la destruction.

- **L'initialisation** : La liste des phases se produisent dans cet ordre précis et se produisent une seule fois durant le cycle de vie des services intégrés. Nous indiquons à côté de chaque phase l'interface requise.
 - activer la notification des événements [*IntegrationLifeCycle*].
 - vérifier que le service accepte de négocier [*Integrable*].
 - négocier si le service implémente *Negotiable* [*Negotiable*] sinon passer étape 4.
 - intégrer les services [*Integrable*].
 - initialiser le service intégré [*IntegrationLifeCycle*].
 - exécuter le service intégré [*IntegrationLifeCycle*].
- **Service intégré actif** : Les occurrences sont multiples, Nous indiquons à côté de chaque phase l'interface requise. :
 - suspendre le service intégré [*IntegrationLifeCycle*].
 - vérifier la durée du cycle de vie [*IntegrationLifeCycle*].
 - re-exécuter le service intégré [*IntegrationLifeCycle*].
- **Destruction** : La liste des phases se produisent dans cet ordre précis et se produisent une seule fois durant le cycle de vie des services intégrés. Nous indiquons à côté de chaque phase l'interface requise.
 - arrêter le service intégré [*IntegrationLifeCycle*].
 - invalider le contrat associé à cette intégration [*Negotiable*].
 - désintégrer le service intégré [*Integrable*].

4. Premier prototype

Dans cette section, nous présentons le développement de notre cadre de conception et l'implémentation de notre prototype sous Java et OSGi. Dans cet article, nous détaillons l'implémentation des trois techniques d'intégration proposées par notre service *IntegServ*. La négociation et le cycle de vie feront l'objet d'un autre article.

4.1. Cadre de conception

L'interface *Integrable* fournit quatre méthodes permettant la gestion de l'intégration de services : la méthode *integrate* (avec ou sans durée), la méthode *getIntegratedServices*, la méthode *disIntegrate* et la méthode *isNegotiable* (vérifie si le service implémente l'interface *Negotiable*). Si l'intégration/la désintégration n'est pas possible, une exception est levée (*IntegrationException* ou *UnIntegrationException*). Le paquetage *IntegServ* (cf. figure 7) contient trois paquetages correspondant aux trois services composant *IntegServ*. Le paquetage *ServiceTechnique* contient trois classes correspondant aux trois techniques d'intégration implantant l'interface *Integrable*. La classe *ByComposition* implémente la méthode *integrate* de l'interface *Integrable* en utilisant la technique de composition. La classe *ByWeaving* implémente la méthode *integrate* de l'interface *Integrable* en utilisant la technique de tissage. Enfin, la classe *ByDeployment* implémente la méthode *integrate* de l'interface *Integrable* en utilisant la technique de déploiement.

L'interface *Negotiable* fournit quatre méthodes : la méthode *accept* (permet à un service d'accepter ou de refuser un contrat), la méthode *propose* (permet à un service d'initier une négociation en proposant à un autre service un contrat), la méthode *isAlive* (vérifie la durée du contrat et notifie quand il s'achève) et la méthode *invalidate* (invalider un contrat). L'exception *NegotiationException* est levée si pour une raison donnée la négociation ne peut avoir lieu.

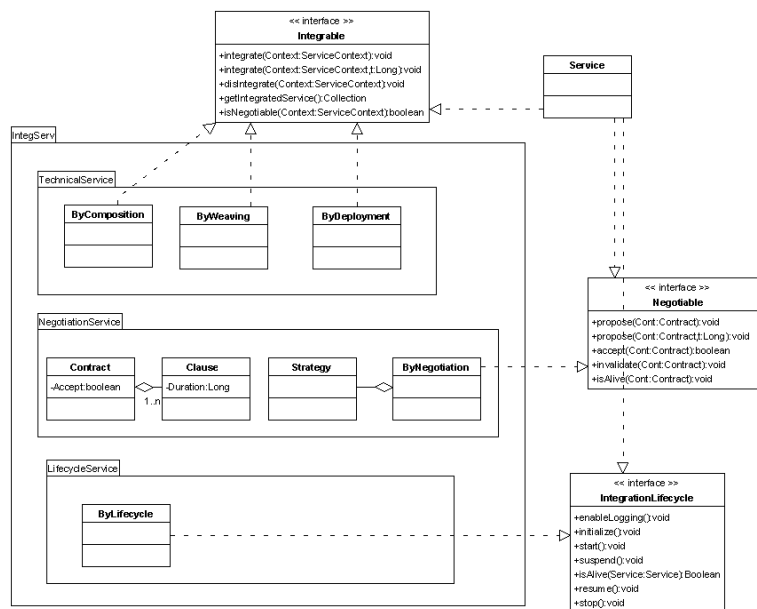


FIG. 7 – Diagramme de classe UML du service *IntegServ*

L'interface `IntegrationLifecycle` fournit plusieurs méthodes : `enableLogging`, `initialize`, `start`, `suspend`, `isAlive`, `resume` et `stop` correspondant aux étapes du cycle de vie. L'exception `LifecycleException` est levée si le service n'est pas disponible.

4.2. Implémentation Java, OSGi

OSGi (*Open Services Gateway initiative*) [1] est une spécification standardisée pour le déploiement de services. Elle est initialement et principalement appliquée aux passerelles installées entre un réseau extérieur tel qu'Internet et un réseau local tel qu'un réseau domestique. Cette spécification permet de favoriser le déploiement des services Java qui sont téléchargés dynamiquement sur les passerelles concernées et qui sont accessibles par tous les appareils du réseau interne connectés à cette passerelle. Nous avons implémenté notre cadre de conception sur la plateforme Oscar [7], une implémentation open-source d'OSGi. Nous avons enrichi Oscar avec notre service *IntegServ*. Nous appliquons notre modèle de service aux composants de déploiement d'OSGi (*bundles*). Les interfaces du modèle correspondent aux interfaces Java. Les objets fonctionnels sont les objets Java instanciés et exécutés avec un `Activator`. Les liens établis sont représentés par le fichier `manifest.mf`. Un service est fourni par un bundle. Un bundle peut également fournir plusieurs services.

4.2.1. Scénario d'utilisation

Dans cet article, nous avons choisi de détailler le code de nos trois techniques : déploiement des services par RMI, la composition des services par imbrication des entrées/sorties (deuxième méthode) et le tissage des services. Nous définissons le scénario suivant illustrant nos différentes techniques. Soient deux services A et B s'exécutant sur deux machines différentes (cf. figure 8). La fonctionnalité du service B est rendu par un appel à la méthode `methodB` et le service A par un appel à la méthode A. Nous supposons que les sorties de la méthode `methodA` sont compatibles avec les paramètres d'entrées de la méthode B. Nous détaillerons les différentes techniques d'intégration en intégrant les deux services A et B.

Le Service B décide d'intégrer le service A (figure 8 étape 1). Le service *IntegServ* présent sur la machine B, arrête le service A pour le télécharger sur la machine B (figure 8 étape 2 & 3), et l'exécuter (figure 8 étape 4). Une fois déployé, *IntegServ* le compose (ou le tisse) avec le service B. Un nouveau service C apparaît ainsi sur la machine B (figure 8 étape 5 & 6) proposant une combinaison des deux services A et B.

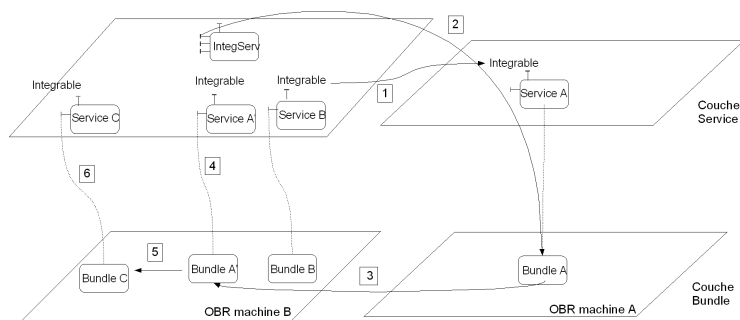


FIG. 8 – Intégration des services A et B

4.2.2. L'appel d'intégration

Le service B appelle la méthode `integrate` de l'interface `Integrable` avec comme paramètre le contexte distant contenant la machine A :

```
serviceB.integrate(context);
```

Listing 1 – L'appel à la méthode d'intégration

Le paramètre `context` de la méthode `integrate` est de type `ServiceContext` qui hérite de la classe `BundleContext` d'OSGi. Le `ServiceContext` contient une liste de services disponibles dans le contexte. Ce paramètre peut être de classe `ConcretContext` quand le contexte est local, c.a.d que le service à intégrer est sur la même machine. Il peut aussi être de classe `ConcretContextSerializable` quand le contexte est distant, c.a.d que le service à intégrer est sur une autre machine réseau.

4.2.3. Technique de déploiement

Les services sont techniquement fournis par des bundles. Sur chaque machine, nous utilisons un répertoire décentralisé et distribué d'Oscar (*Oscar Bundle Repository OBR*) où tous les bundles sont stockés. Le service `IntegServ` s'exécutant sur la machine B analyse l'emplacement du service A et décide de le télécharger de sa machine et de l'exécuter localement avant de l'intégrer avec le service B. L'implémentation de la méthode `integrate` (cf. Listing 2) correspond à celle fournie par la classe `ByDeployment` (cf. figure 7). Le service A est alors arrêté et le bundle A est sérialisé et envoyé par flux (cf. Listing 3).

```
ByDeployment mtc = new ByDeployment();
String location = "C:/OBR/bundleA.jar";
byte[] filedata = mtc.downloadBundle(location);
File file = new File(location);
BufferedOutputStream output = new
BufferedOutputStream(new FileOutputStream(file.getName()));
output.write(filedata,0,filedata.length);
output.flush();
output.close();
```

Listing 2 – Intégration par déploiement

```
File file = new File(fileName);
byte buffer[] = new byte[(int)file.length()];
BufferedInputStream input = new BufferedInputStream(new FileInputStream(fileName));
input.read(buffer,0,buffer.length);
input.close();
return(buffer);
```

Listing 3 – code de la méthode `downloadBundle`

4.2.4. La Composition par redirection

Une fois téléchargé, le bundle A est installé et exécuté. Le service A est ainsi disponible sur la machine B. Une composition locale (cf. figure 9) fournie par la classe `ByComposition` (cf. figure 7) peut avoir

lieu. Un nouveau bundle est ainsi créé offrant un service C accessible par la méthode `methodC()` et redirigeant vers les deux services A et B (s'ils sont disponibles). La méthode C lance la méthode A suivi de la méthode B. La classe `ByComposition` (cf. figure 7) implémente ici la composition par redirection (cf. figure 4) parce que nous considérons ici des services dont les interfaces sont compatibles.

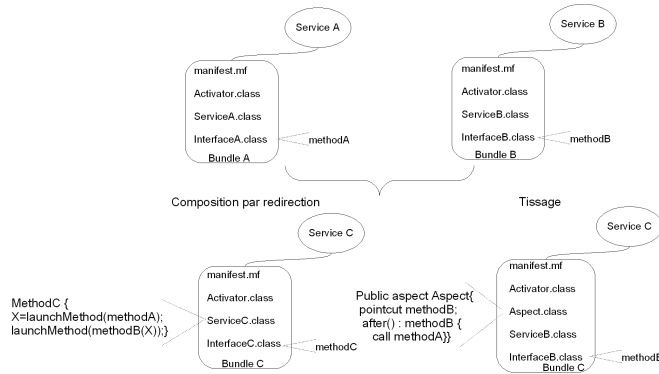


FIG. 9 – Composition par redirection et tissage de services

Le code de la méthode `launchMethod` est donné en Listing 4.

```
Object launchMethod(Object o, Object[] args,
    String nomMethode) throws Exception {
    Class[] paramTypes = null;
    if (args != null) {
        paramTypes = new Class[args.length];
        for (int i=0; i<args.length; ++i)
            {paramTypes[i] = args[i].getClass();}
    }
    Method m = o.getClass().getMethod(nomMethode, paramTypes);
    return m.invoke(o, args);
}
```

Listing 4 – code de la méthode `launchMethod`

4.2.5. Technique de tissage

Le service `IntegServ` peut appliquer une autre technique d'intégration pour intégrer les deux services A et B, le tissage (cf. figure 9). Nous distinguons deux différentes méthodes pour le faire (cf. figure 10). La première consiste à tisser dans le code du service B l'appel à la méthode A du service A. La deuxième consiste à tisser le code de la méthode A. Dans la deuxième technique, le service tissé peut ainsi exister même si le service A devient indisponible, alors que dans la première technique si le service A disparaît le service tissé ne peut plus fonctionner correctement.

Techniquement le service B est arrêté, `IntegServ` extrait les classes du bundle B. Pour la première technique, `IntegServ` tisse l'aspect et la classe Service B. Pour l'instant, nous avons utilisé `AspectJ` [14] qui agit sur les classes. Pour cela, nous lui fournissons un aspect en code source et après compilation nous exécutons le tissage avec le service B (cf. figure 9). Nous sommes en train de tester `JAC` [19] pour son orientation plus objets que classes. Nous allons essayer de voir si nous pouvons utiliser la deuxième méthode de tissage (cf. figure 10) avec `JAC`. Le code de l'aspect tissé sur le service B est donné dans le listing 5.

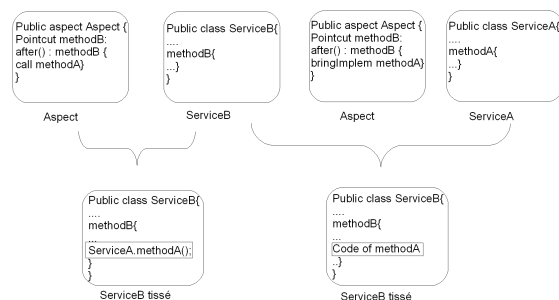


FIG. 10 – Deux méthodes différentes de tissage

```

public aspect Aspect {
    pointcut callMethodB() : call (public static * ServiceB.methodB(..));
    after(ServiceContext context) : callMethodB() {callMethodA(context);}
    void callMethodA(ServiceContext context) {
        try {
            ServiceReference[] refs = context.getServiceReferences(
                InterfaceA.class.getName(), "(service=serviceA)");
            if (refs != null) {
                InterfaceA comp = (InterfaceA ) context.getService(refs[0]);
                comp.methodA();
            } catch (InvalidSyntaxException ex){}
        }
    }
}

```

Listing 5 – Aspect code source compilé par AspectJ

4.2.6. La réversibilité de l'intégration

Le service B appelle la méthode `disIntegrate` de l'interface `Integrable` avec comme paramètre le contexte distant contenant la machine A :

```
serviceB.disIntegrate(context);
```

Listing 6 – L'appel à la méthode de désintégration

Cet appel a pour effet d'arrêter l'exécution du service C et de défaire tous les liens le concernant. `IntegServ` vérifie avant que le service n'est pas utilisé par un autre. S'il est utilisé, la désintégration est reportée le temps d'avertir tous les services utilisant le service C de sa prochaine disparition.

Rien n'empêche d'enrichir notre prototype sous OSGi. La procédure est simple. Il suffit d'ajouter des implémentations de nouvelles techniques dans le package `IntegServ`. Par exemple, ajouter une classe à ce paquetage qui implémente différemment la méthode `integrate` de l'interface `Integrable`. Rien n'empêche non plus d'utiliser d'autres outils basés sur une autre technologie qu'OSGi et qui respecte le modèle de service et le cadre de conception.

5. Etat de l'art

Trois domaines importants de la programmation par objets se penchent sur la problématique de l'intégration : la programmation orientée composants (*Component-Based Software Engineering CBSE*) [8], la programmation orientée aspect (*Aspect-Oriented Programming AOP*) [11] et la programmation orientée services (*Service-Oriented Programming SOP*) [24].

Les différents types de modèles à base de composants comme les EJBs [17], le modèle à composant de CORBA [26], Fractal [3] permettent l'intégration de composants. L'intégration de composants dans ces différents modèles est souvent réduite au déploiement de composants et/ou à la paramétrisation de certains attributs de ces composants.

Fractal [3] [2] d'ObjectWeb est un modèle à composants pour la conception et l'implantation d'applications et de systèmes reconfigurables. Il est principalement utilisé pour construire des intergiciels, où les problèmes de reconfiguration et de déploiement sont cruciaux. Pour Fractal, comme pour les autres modèles cités, un composant est en première approximation un objet, isolé du monde extérieur par des interfaces. On distingue ici les interfaces serveur (les services rendus par le composant), les interfaces client (les services requis) et les interfaces de gestion et configuration. Une interface client et une interface serveur peuvent communiquer lorsqu'elles sont liées (opération de *binding*). Pour ce faire, chaque interface est identifiée par un nom, et la recherche d'interfaces est déléguée à un service d'annuaire.

La programmation orientée aspect permet d'implanter les préoccupations transverses (les aspects) indépendamment les unes des autres et de les combiner ultérieurement (le tissage) pour produire l'application finale. AspectJ [14], FAC [20] et JAC [19] sont des modèles à base d'aspect appliquant le tissage d'aspect comme méthode d'intégration. FAC est une extension du modèle à composant Fractal supportant la programmation orientée aspect. JAC (*Java Aspect Component*) propose un cadre de conception pour la programmation aspect sous Java. JAC permet à des applications, d'adapter dynamiquement les aspects avant de les appliquer aux applications durant leur exécution.

Dans la terminologie de la programmation orientée services, l'intégration de service est souvent réduite à une composition de service. Actuellement, ces recherches visent à développer une architecture qui permet la composition de service en utilisant un raisonnement logique offert par les langages de description de service comme DAML [23], UDDI et WSDL [25]. Les servicesWebs [9] fournissent une solution pour la composition et l'assemblage des composants web en se basant sur des protocoles XML. SWORD [21] est un outil de développement pour composer les servicesWebs. Il ne se base pas sur les langages de description usuels comme WSDL ou DAML-S mais définit un plan de génération basé sur les règles en utilisant un modèle entité-relation.

6. Conclusion et travaux futurs

Dans cet article, nous avons présenté un système d'intégration de services. Notre système fournit un cadre de conception avec un ensemble d'interfaces - *Integrable*, *Negotiable*, *IntegrationLifeCycle* - et un ensemble d'outils implémentant ces interfaces. Notre cadre de conception est simple et s'applique à un modèle générique de service. Son utilisation est également simple, les services invoquent seulement la méthode *integrate* de l'interface *Integrable* sans avoir besoin de connaître les mécanismes impliqués.

Pour démontrer la faisabilité de notre système, nous avons implanté l'ensemble d'outils correspondant à notre cadre de conception en utilisant les technologies Java et OSGi. Nous avons enrichi la plateforme OSGi avec notre service *IntegServ*. Dans cet article, nous détaillons uniquement les différentes techniques de l'intégration : déploiement, la composition par redirection et le tissage.

Cette implantation constitue également la limite de la version actuelle de notre système : les applications désirant utiliser ces outils doivent supporter la machine virtuelle de Java et la plateforme OSGi. Rien n'empêche toutefois l'application de notre modèle de service et notre cadre d'intégration à d'autres technologies.

Dans le futur, nous visons l'ajout d'une description sémantique aux services afin d'enrichir les possibilités de négociation et d'intégration de services, en permettant par exemple la transformation d'interfaces entre services sémantiquement compatibles.

Bibliographie

1. OSGI Alliance. OSGi Service Platform, Core Specification Release 4. Draft, 07 2005.
2. E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002*, June 2002. Malaga, Spain.
3. Eric Bruneton. *Developing with Fractal*. The ObjectWeb Consortium, France Telecom (R&D), March 2004. version 1.0.3.
4. David Portabella Clotet, Vincenzo Pallotta, and Martin Rajman. Systematic definition and assent to eContracts for Web Services. In *Workshop on Contract Architectures and Languages (CoALa 2005)*, in

conjunction with the 9th IEEE International Enterprise Computing Conference (EDOC 2005), September 2005. Enschede, The Netherlands.

5. Microsoft Corporation. Understanding UPnP : A white paper. Technical report, UPnP Forum, 2000.
6. Nikolaos Georgantas, editor. *Detailed Design of the Amigo Middleware Core : Service Specification, Interoperable Middleware Core*, Deliverable D3.1b, IST Amigo project, 2005.
7. Richard S. Hall. Oscar an OSGI framework implementation. Technical report, Objectweb organisation, 2005.
8. George T. Heineman and William T. Councill. *Component-Based Software Engineering : Putting the Pieces Together*. Addison-Wesley, June 2001.
9. Will Iverson. *Real Web services*. O'Reilly, October 2004.
10. Hai Jin and Hao Wu. Semantic-enabled Specification for Web Services Agreement. *International Journal of Web Services Practices*, Vol.1(No.1-2 pp. 13-20), 2005.
11. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, June 1997.
12. Trung Nguyen Kien, Abdelkarim Erradi, and Piyush Maheshwari. WSMB : a middleware for enhanced web services interoperability. In *Interop-ESA'05, First International Conference on Interoperability of Enterprise Software and Applications*, 2005. Geneva, Switzerland.
13. S. Ilango Kumaran. *JINI Technology An Overview*. Prentice Hall PTR, 2002.
14. Ramnivas Laddad. *AspectJ in Action : practical Aspect-Oriented Programing*. Manning publications, July 2003.
15. Frédéric Le Mouël, Françoise André, and Maria-Teresa Segarra. AeDn : An Adaptive Framework for Dynamic Distribution over Mobile Environments. *Annales des Télécommunications*, 57(11-12) :1124–1148, November-December 2002.
16. Berin Loritsch. Developing With Apache Avalon. Technical report, Apache Software Foundation, 2001.
17. Richard Monson-Haefel. *Entreprise JavaBeans*. O'Reilly & Associates, March 2000.
18. OSGIalliance. About the OSGI service platform. Technical report, OSGI alliance, July 2004. revision 3.0.
19. Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. JAC : An Aspect-based Distributed Dynamic Framework. *Software Practise and Experience (SPE)*, 34(12) :1119–1148, 2004.
20. Nicolas Pessemier, Lionel Seinturier, and Laurence Duchien. Components, ADL and AOP : Towards a common approach. In *Workshop ECOOP Reflection, AOP and Meta-Data for software Evolution (RAM-SE04)*, June 2004.
21. S. R. Ponnekanti and A. Fox. SWORD : A Developer Toolkit for Web Service Composition. In *11th World Wide Web Conference*, 2002. Honolulu, USA.
22. M. W. Rennie and V. B. Mistic. Towards a Service-Based Architecture Description Language. Tr 04/08, University of Maniuba, August 2004.
23. Mithun Sheshagiri, Marie des Jardins, and Timothy Finin. A planner for composing services described in DAML-S. In *International Conference on Automated Planning and Scheduling (ICAPS) 2003 Workshop on planning for web services*, July 2003.
24. Munindar Singh and Michael N. Huhns. *Service-Oriented Computing*. Wiley, December 2005.
25. Aaron E. Walsh. *UDDI, SOAP and WSDL : the Web Services specification Reference book*. Pearson Education, April 2002.
26. Ron Zahavi. *Entreprise Application Integration with Corba Component and Web-Based solutions*. John Wiley & sons, November 1999.