RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants
Le Croisic, France, 5 au 8 avril 2005

Implémentation de plates-formes dynamiques de services avec .NET

Clément Escoffier, Didier Donsez

Laboratoire LSR-IMAG,
220 rue de la Chimie
Domaine Universitaire, BP 53, 38041
Grenoble, Cedex 9, FRANCE
Clement.Escoffier, Didier.Donsez@imag.fr

Résumé
La demande afin de concevoir des applications dont l' architecture évolue dynamiquement en cours d'exécution est très en vogue aujourd' hui. Les plates-formes dynamiques de services (PDS) permettent de concevoir une application comme un ensemble de services fournis et requis qui peuvent apparaître et disparaître au cours de l' exécution de l' application. OSGi est le standard de fait des PDS pour des applications dont les services sont tous exécutés dans la même machine virtuelle Java. .NET étant le compétiteur de Java, il nous est apparu intéressant de proposer une PDS centralisée semblable à OSGi au dessus de la CLR .NET. Cet article propose 4 implémentations d' une telle PDS pour la CLR .NET.

Mots-clés : Programmation Orientée Service Dynamique, Chargeur de classes, CLR .NET, ROTOR, OSGi

---

1. Introduction

La demande afin de concevoir des applications dont l' architecture évolue dynamiquement en cours d' exécution est très en vogue aujourd' hui. Cette demande est principalement liée au besoin qu'ont certaines applications d' être sensible au contexte (context-aware) ou bien de pouvoir charger/mettre à jour/décharger certains constituants de l'application sans interrompre globalement son exécution. Ce besoin se retrouve dans des applications distribuées spontanées dans lesquelles les noeuds apparaissent et disparaissent (voir à tout jamais), mais également dans des applications centralisées dans lesquelles les constituants de l' application (qui apparaissent et disparaissent dynamiquement) s' exécutent sur la même machine et dans le même espace d' adressage de processus. Le système JINI est la figure emblématique des plates-formes dynamiques d' exécution pour les applications distribuées et spontanées. De son coté, la spécification OSGi devient peu à peu le standard de fait pour les plates-formes dynamiques centralisées.
Un premier point commun entre ces plates-formes est qu' ils suivent les principes de la programmation orientée service dynamique (POSD). La programmation orientée service (POS) [1], que les Web Services ont récemment popularisés, définit le service comme brique de base de construction des applications. Celui-ci est décrit syntaxiquement, sémantiquement et comportementalement par un contrat. L' architecture de l' application est décrite en termes de contrats fournis par des fournisseurs de services (service providers) et de contrats requis par les demandeurs de service (service requesters). En programmation orientée service dynamique (POSD) [2], les fournisseurs de services proposent ou retirent leurs services à tout moment en cours d' exécution de l' application. De leur coté, les demandeurs de services sont avertis de l' arrivée ou du retrait des services dont ils sont demandeurs. Nous appelons désormais les plates-formes mettant en oeuvre les principes de la POSD, plates-formes dynamiques de services (PDS).
Le second point commun entre les principales plates-formes existantes (JINI, OSGi) est qu' elles sont basées sur le langage Java. Avec ce langage, Sun a popularisé à la fois l' idée d' utiliser une machine virtuelle pour exécuter des programmes de manière sûre et portable, et l' usage du chargement de codes en cours d' applications. Cependant, son compétiteur Microsoft a proposé à son tour une autre machine virtuelle aux fonctionnalités semblables mais supportant plusieurs langages de programmation.

Il nous est donc apparu intéressant de vérifier si cette machine virtuelle, baptisée Common Language Runtime (CLR), pouvait être une bonne candidate pour développer des plates-formes dynamiques de services. Dans cet article, nous nous focaliserons uniquement sur la construction d' une tel PDS dans le contexte mono-machine avec la CLR. Nous nous concentrerons plus particulièrement sur les points durs de ces plate-formes qui sont le chargement et le déchargement des unités de déploiement (appelées assemblies dans la terminologie .Net) fournissant les services.
Cet article est organisé de la manière suivante. La section 2 présente les principes des architectures orientés services dynamiques et les plates-formes existantes. La section 3 est consacrée à la description du chargement de classes dans la plate-forme .NET. La section 4 décrit et compare les alternatives possibles de réalisation des plates-formes dynamiques de services avec .NET que nous avons implémentées. Le chapitre 5 conclut et présente les perspectives à ce travail.

2. Programmation Orientée Service Dynamique

Cette section décrit les principes de la programmation orientée service dynamique, liste les principales applications et présente deux grands représentants des plates-formes dynamiques de services (PDS) : JINI et OSGi

2.1. Principes

La programmation orientée service (POS), que les Web Services ont récemment popularisés, définit le service comme brique de base de construction des applications. Un service est décrit syntaxiquement, sémantiquement et comportementalement par un contrat. L'architecture de l'application est décrite en terme de contrats fournis par des fournisseurs de services (service providers) et de contrats requis par les demandeurs de service (service requesters). Les services fournis sont enregistrés dans un registre de services (aussi appelé courtier) par leurs fournisseurs. Un demandeur interroge le registre qui lui retourne la liste des services respectant le contrat requis.
La programmation orientée service dynamique (POSD) s'intéresse à considérer que les services peuvent apparaître et disparaître à tout moment au cours de l'exécution de l'application. En programmation orientée service dynamique (POSD) [3][4], les fournisseurs de services proposent ou retirent leurs services à tout moment en cours d'exécution de l'application. De leur coté, les demandeurs de services sont notifiés de l'arrivée ou du retrait des services dont ils sont demandeurs.
Nous appelons désormais les plates-formes mettant en œuvre les principes de la POSD, plates-formes dynamiques de services (PDS). Ces plates-formes sont principalement constituées d'un registre de services permettant l'enregistrement et le courtage de services, un système de notification prévenant des chargements dans l'annuaire et d'un mécanisme de déploiement et de chargement des classes implémentant le contrat ou fournissant un point d'accès à ce contrat. Une PDS peut éventuellement permettre la mise à jour partielle de la définition d'un contrat ou bien celle des classes d'implémentation, sans que la PDS n'interrompt globalement son service.

2.2. Applications

Les applications qui requièrent une PDS sont multiples. On peut citer les
– Passerelle de services
– Serveurs d' applications
– Applications à plugins (Eclipse 3.0)
Les passerelles de services sont des passerelles accessibles par Internet (ou par un autre réseau) sur lesquelles des prestataires de services déploient leurs applications. L'utilisation d'une PDS permet l'installation dynamique de nouvelles fonctionnalités, sans arrêts de la passerelle. Celles-ci permettent par exemple d'effectuer un diagnostic de pannes, ou offrent un nouveau service à l'utilisateur. L'utilisation de plate-forme de services dans une voiture permettrait de vendre au client des services autour de son véhicule comme l'appel d'urgence, l'aide à la navigation, la localisation de services (hôtels, pompes à essence, sites touristiques ...). La plate-forme peut aussi servir à diagnostiquer les pannes à distance.
Les PDS peuvent aussi être utilisées dans le cadre de serveurs d'applications, afin de pouvoir installer, mettre à jour ou retirer des applications sur le serveur sans pour autant perturber le fonctionnement des autres applications.

Les applications à plugins peuvent utiliser une PDS pour la gestion de leurs plugins. En effet, cela permet d'installer ou de retirer des plugins sans redémarrer toute l'application. L'environnement de développement Eclipse utilise depuis sa version 3, une plate-forme de services OSGi pour sa gestion de plugins.

2.3. Plates-Formes

JINI [5] est la figure emblématique des plates-formes dynamiques de services pour les applications distribuées et spontanées. Dans JINI, un service est décrit par une interface Java qualifiée d'un ensemble de propriétés. Le courtage de services repose sur un registre totalement distribué (mis à jour par tous les fournisseurs de service par une diffusion de messages d'annonce UDP Multicast). Le courtage retourne la liste des services potentiellement disponibles jusqu'à l'expiration de leurs baux (lease). Un service est rendu par une entité distante et il est accessible via un proxy qui peut être un simple stub RMI ou un stub « intelligent » (par exemple : doué de fonctions de cache d'état, utilisant un protocole non TCP, ...). Ce proxy est une classe Java qui est chargée depuis une URL quelconque.

OSGi (Open Service Gateway Initiative) [6] est une spécification du consortium industriel OSGi pour la définition d'une plate-forme dynamique de services centralisée en Java. Bien que la cible initiale était les passerelles domotiques/immotiques, cette spécification a été adoptée largement pour les passerelles industrielles, ainsi que pour les passerelles véhiculaires (automotic). La prochaine spécification (Release 4) devrait couvrir également les besoins de fabricants de téléphones mobiles et des applications à plugins. Dans OSGi, les services sont fournis par des unités de déploiement appelés bundles qui sont installés, démarrés, mis à jour, arrêtés ou bien retirés à tout moment. Ces bundles et les services qu'ils fournissent sont partagés dans l'espace de références de la même machine. Un service est une interface Java qualifiée par des propriétés. Le fournisseur enregistre son service dans un registre de services local à la plate-forme OSGi. Le demandeur interroge le registre au moyen d'une expression vérifiant les propriétés des services enregistrés. Quand un service doit être retiré (par exemple : arrêt d'un bundle ou condition non remplie pour assurer le service), tous les demandeurs sont notifiés de manière synchrone avant que le service soit réellement retiré. Ceci est rendu possible par la centralisation des services sur la même machine.

Il existe d'autres PDS. Nous mentionnerons OpenWings [7] qui est une plate-forme dynamique de services à la fois centralisée et distribuée basés sur le langage Java. Les connections entre services requis et services fournis peuvent être de type synchrone et asynchrone. Pour les connections entre deux machines distribuées, les connections masquent jusqu'au déploiement les intergiciels utilisés qui peuvent être JINI, RMI, IIOP, SOAP/HTTP en synchrone et JMS, SOAP/JAXM en asynchrone.

3. Chargement de classes en Java et en .NET

Dans cette section, nous rappelons brièvement le conditionnement des applications Java et le mécanisme de chargement des classes, puis nous détaillerons plus particulièrement ces deux points dans .Net qui est au c$\frac{1}{2}$ur de notre problématique.

3.1. Chargement de classes en Java

Java a popularisé l'idée d'utiliser une machine virtuelle pour exécuter des programmes de manière sûre et sans « problèmes » d'hétérogénéité d'architectures de machines. Java a également popularisé l'usage du chargement de code en cours d'applications. La machine virtuelle Java [8] interprète du code portable (ByteCode) délivré sous forme de classes. De manière basique, la machine virtuelle charge les classes depuis un des répertoires locaux (listés dans la variable d'environnement CLASSPATH). Cependant, il est néanmoins recommandé de livrer les classes en les conditionnant dans un fichier JAR (Java Archive). Ce fichier contient un manifeste donnant un certain nombre d'informations sur l'application tels que la classe servant le point d'entrée, la liste des dépendances avec d'autres fichiers JAR, etc.

La machine virtuelle Java charge les classes de l'application et de l'environnement d'exécution au fur et à mesure qu'elle rencontre une référence d'importation dans la classe en cours de chargement [9][10]. Le chargement est en fait délégué à un chargeur de classes (java.lang.ClassLoader) et plus exactement à une hiérarchie de chargeurs. Chaque chargeur demande de charger la classe à son supérieur hiérarchique et si le supérieur échoue alors il tente de récupérer le fichier classe depuis les répertoires et fichiers JAR (locaux ou distants) qu'il connaît. Ce mécanisme est appelé délégation. Le chargeur de base peut

être personnalisé afin de créer des politiques de chargement personnalisées (J2EE, RMI, JMX, ANT, J2ME/CDC/Personal Basic Profile, JITS [11]). Les classes sont déchargées par le ramasse-miette (garbage collector) quand celles-ci ne sont plus utilisées. Il en va de même pour un chargeur de classes.

La politique de chargement de classes d' OSGi est implémentée par un chargeur de classe (BundleClassLoader) qui utilise une délégation vers ses « voisins » pour résoudre le chargement des classes présentes dans des packages importés et/ou exportés. Un package (importé ou exporté) correspond en général à un contrat entre un bundle demandeur et un bundle fournisseur. Le BundleClassLoader masque à ses voisins les classes qui ne sont pas dans packages exportés ou importés [1]. Cette fonctionnalité permet d'avoir un schéma privé de nommage de classes évitant des conflits de chargement entre bundles. Ces classes sont généralement les classes d'implémentation des services. OSCAR [13][14], qui est une implémentation libre de la spécification R3 d'OSGi, utilise un chargeur de classes flexible et configurable, dirigé par plusieurs politiques de chargement, de résolution des conflits de noms et de version : le ModuleLoader [15].

3.2. Chargement de classes en .NET

.Net est la solution que Microsoft a créé pour concurrencer Sun et son langage Java. Bien que .Net soit globalement proche de Java, il existe quelques différences importantes. Tout d'abord, la plate forme .Net [16] supporte plusieurs langages (CSharp, JSharp, VB.Net ...) alors que la machine virtuelle Java, ne prend en compte qu'un seul langage (Java). Tous ces langages s'exécutent sur l'environnement d'exécution de .Net appelé CLR (Common Langage Runtime). La CLR utilise un langage portable au format MSIL (MicroSoft Intermediate Language), tout comme le bytecode de la machine virtuelle Java. Le MSIL des classes est la plupart du temps compilé en code natif de la machine cible au moment du chargement grâce au compilateur JIT (Just In Time). Contrairement à une JVM, la CLR conserve le code natif compilé pour les chargements ultérieurs dans un cache global à la machine, le Global Assembly Cache (GAC).

L'implémentation principale est la plate-forme .Net de Microsoft, uniquement disponible sur les systèmes d'exploitation Windows. Il existe cependant d'autres implémentations de .Net comme Mono qui est une version libre fonctionnant sous Windows mais aussi sous Linux. Microsoft a aussi partagé une partie des sources de sa plate-forme .Net : Shared Sources CLI (appelée ROTOR). Cette dernière est une machine virtuelle .Net très proche de la CLR commerciale.

Pour s'exécuter, l'application est conditionnée dans un emballage de classes appelé Assembly. Il n'est pas possible de charger une classe indépendante comme le fait Java. Il est néanmoins possible de charger dynamiquement des assemblies. Une assembly ressemble à un fichier Jar. Son manifeste qui est au format XML contient entre autre des informations tels que le numéro de version, une langue (appelée culture) ainsi qu'une clé de hachage (permettant d'identifier le fichier de manière unique). Cette dernière information définit un ensemble d' assemblies nommées « Strong Name Assemblies » [17].

Une application .Net s'exécute dans un Domaine d' Application. Il s'agit d'un environnement d'exécution isolé. On peut le comparer à un processus virtuel. Il peut y avoir plusieurs domaines d'applications en même temps dans une machine virtuelle (donc dans un processus), mais ils sont complètement séparés. La communication de 2 domaines d'applications doit utiliser un canal de communication interprocessus, généralement .Net Remoting (équivalent sur .Net de Java RMI).

Les assemblies sont chargées dans les domaines d'application. Elles ne sont pas disponibles dans les autres domaines. Sur la figure1, l' assembly Asm1 n'est pas disponible dans le domaine d'application AppDomain 2. Par défaut, chaque application .Net s'exécute dans un domaine portant le nom de l'application. Mais une application peut créer d'autres domaines.

Les domaines d'applications gèrent le chargement des assemblies. Lorsque l'application a besoin d'une classe, le principe de chargement est le suivant : si une classe est déjà dans une assembly du domaine, elle est forcément réutilisée sinon l' environnement d'exécution recherche l' assembly contenant la classe puis la charge, la classe contenue étant ensuite utilisée. Il n'est pas possible de recharger une assembly plus récente lorsque qu'une autre version est déjà chargée dans le domaine. Par contre, un domaine d'application peut être détruit. Les assemblies sont alors déchargées. Il s'agit de la seule façon de décharger du code sur la plate-forme .Net. Les classes n'étant pas considérées comme des objets « normaux », le ramasse miettes ne les collecte pas.

[1] La spécification 3 d'OSGi prévoit qu'une seule version d'un package peut être chargée dans le framework OSGi à la fois. Cette limitation devrait être levée dans la future version de la spécification OSGi [12].

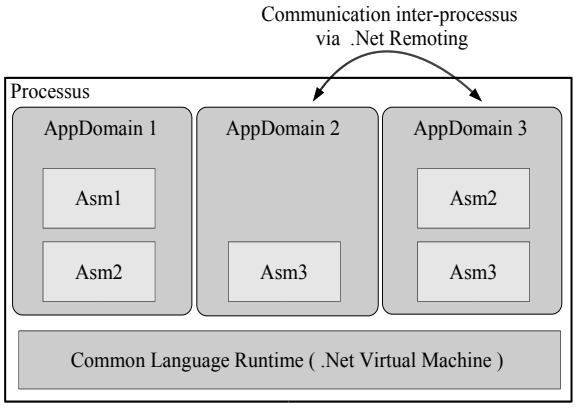

FIG. 1 – Les domaines d'applications sur .Net

La plate-forme .Net n'a pas de Classpath comme Java. L'environnement d'exécution recherche les assemblies dans un répertoire spécial appelé Global Assembly Cache (GAC) stockant les assemblies dites « Strong Name ». La machine virtuelle scanne aussi d'autres localisations spécifiées dans le fichier de configuration de l'application. Ce fichier XML, permet de décrire précisément la politique de recherche de l'application en spécifiant les localisations scannées. Par contre, pour des raisons de sécurité, il est statique, c'est à dire qu'il ne peut pas être modifié en cours d'exécution.

4. Alternatives de plates-formes dynamiques de services en .NET

Notre objectif est d'étudier la possibilité de créer une plate-forme de services sur .Net. Pour cela, 4 alternatives ont été étudiées. Nous nous sommes particulièrement intéressés aux 6 fonctionnalités suivantes que nous retiendrons pour comparer nos 4 propositions :
– l'apparition dynamique des services
– le chargement dynamique du code
– le déchargement dynamique partiel
– la visibilité du code chargé (c'est à dire la portée des classes),
– l'invocation de services
– le comportement des attributs statiques
– la modification de la CLR
L'apparition dynamique des services est une fonctionnalité primordiale dans une plate-forme de services. Les services apparaissant en cours d'exécution doivent être pris en compte. Pour cela, la capacité de pouvoir charger et décharger du code dynamiquement est importante. C'est une condition nécessaire pour implémenter une plate-forme de services. Il doit être possible d'installer un service sur la plate-forme, de le prendre en compte puis de le désinstaller sans redémarrer la plate-forme. Les spécifications OSGi proposent un système de visibilité de classes différent du système par défaut de Java. Un des enjeux des solutions proposées est de pouvoir adapter le comportement du chargement des classes de .Net afin de proposer une politique répondant aux spécifications OSGi. De plus OSGi permet le partage des attributs statiques entre les services, ainsi que l'invocation des services hébergés sur la plate-forme.
Le dernier critère retenu concerne l'utilisation de la machine virtuelle .Net « officielle ». En effet, il est possible grâce à ROTOR de modifier la machine .Net afin de changer son comportement. Pour la dernière proposition (4.4), nous avons utilisé une CLR modifiée.

4.1. Un seul domaine d'application pour toutes les assemblies

Dans cette première alternative, nous nous sommes principalement intéressé à l'apparition dynamique de services, et à la représentation d'un service. Pour cela, nous avons décidé de considérer les services

comme les bundles d' OSGi [18]. Cette première approche propose d'empaqueter les services dans des assemblies, comme OSGi empaquette ses services dans des fichiers Jar spéciaux nommés bundles. En effet .Net permet le chargement dynamique des assemblies à l'intérieur d'un domaine d'application, tout comme OSGi des bundles. La figure 2 illustre cette architecture.

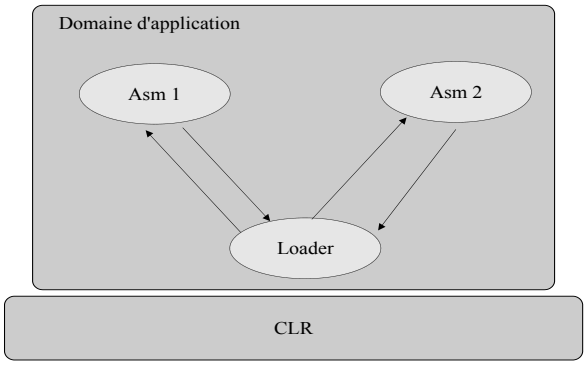

FIG. 2 – architecture et chemin de recherche dans la solution à un seul domaine d'application

Cette approche propose de créer une assembly particulière (Loader), gérant le chargement des services (en chargeant les assemblies contenant ces services). Ceci permet l'apparition dynamique des services. De plus l'accès aux services est rapide vu que l'invocation est intra-processus.
Mais cette solution ne résout pas de nombreux problèmes. Tout d'abord, les assemblies chargées ne sont pas déchargeables. En effet, .Net ne permet pas le déchargement des assemblies individuellement. Pour décharger les assemblies, il faudrait décharger le domaine d'application en entier (ce qui est équivalent à éteindre la plate-forme) puis la redémarrer sans le service supprimé (tout en gérant les dépendances). Ce problème est une limitation non négligeable. De plus, cette approche mais elle rencontre aussi un autre problème. Les classes chargées dans le domaine sont disponibles à toutes les autres classes du domaine. C'est à dire, lorsque Asm2 est chargée, les classes de Asm1 peuvent utiliser les classes de Asm2. Ce comportement permet de ne charger qu'une seule fois la classe et d'être sûr que toutes les autres classes du domaine l'utilisent. Ce comportement se rapproche du système de chargement de classes spécifié dans OSGi, et permet aussi de s'assurer de l'unicité des attributs statiques. En contre-partie, il n'est pas possible de modifier ce comportement. Or OSGi précise qu'il est possible de ne pas « publier » une classe (c'est à dire, interdire l'accès à cette classe à d'autres bundles).
Afin de résoudre le problème de déchargement de code, une deuxième approche utilisant les méthodes de déchargement de code de la plate-forme .Net a été réalisée. Le problème de partage de classes est étudiée dans l'alternative 4.3 et 4.4.

4.2. Un bundle par domaine d'application

La deuxième alternative étudiée se base sur les domaines d'applications de .Net. Chaque service est toujours empaqueté dans une assembly, mais celle-ci est chargée à l'intérieur d'un domaine d'application. La plate-forme de services s'exécute dans un autre domaine. Les figures 3 et 4 illustrent cette architecture.
La plate-forme de services gère le chargement des assemblies contenant les services et le déchargement des services. Pour cela, la plate-forme crée de nouveaux domaines d'applications pour héberger le nouveau service, elle y charge ensuite l' assembly du service. Lorsque le service disparaît, la plate-forme détruit le domaine l'hébergeant, déchargeant par la même occasion l' assembly du service.
De plus pour résoudre le problème de visibilité des classes, la plate-forme doit aussi gérer l'ensemble des ressources disponibles dans chacun des domaines. Lorsqu'un service cherche à charger une ressource, il délègue à la plate-forme cette recherche (1). La plate-forme interroge les autres services (2).

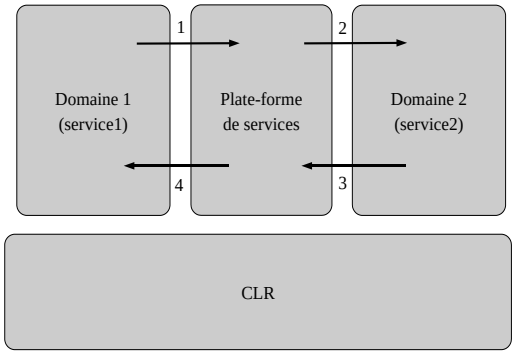

FIG. 3 – Chemin de recherche

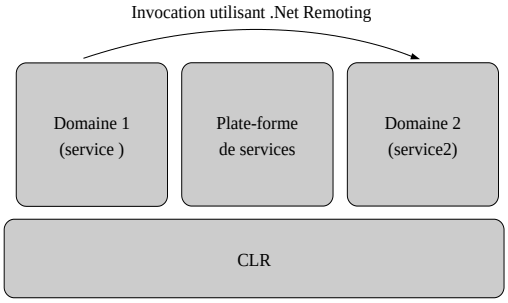

FIG. 4 – Invocation de services

Si la ressource est déjà chargée dans un autre domaine (3), la plate-forme récupère cette ressource et la transmet au service demandeur (4)
Malgré ces 2 avantages, cette solution possède un inconvénient indéniable. Comme nous avons vu auparavant, les domaines d'application sont des environnements isolés d'exécution que l'on peut assimiler à des processus virtuels. Leur isolement oblige l'utilisation d'un système de communication interprocessus pour les faire communiquer entre eux. La technologie couramment utilisée est .Net Remoting. Malheureusement, cette communication nécessite des indirections coûteuses. Il faut tout d'abord sérialiser l'objet échangé puis le déssérialiser. De plus la latence du réseau diminue encore l'efficacité de la communication. Cette perte de temps, ne permet pas de concevoir une plate-forme de services de manière acceptable. Néanmoins ce type de communication par RPC locaux a déjà été choisie pour la communication entre Xlets s'exécutant sur une plate-forme J2ME/CDC/PBP [19] (utilisé par exemple dans Java TV [20]).
Un autre aspect qu'impose cette alternative concerne la non cohérence des attributs statiques. Ceux ci ne traversent pas les frontières des domaines. En effet, lorsqu'une classe est chargée dans 2 domaines différents, les attributs statiques sont recopiés dans chaque domaine. Il faudrait mettre en place un dispositif

garantissant la cohérence de ces éléments.
L'alternative suivante se concentre sur le partage de code entre les services, afin de simplifier le chemin de recherche lorsqu'une ressource est demandée par un service.

4.3. Utilisation du Shared Domain

La troisième alternative utilise ROTOR, l'environnement .Net de Microsoft distribué sous la licence Shared Sources [21]. En effet, l'étude de cette plate-forme a montré un aspect intéressant de .Net : le Shared Domain. Ce domaine particulier n'est pas réellement un domaine d'application, car il ne s'agit pas d'un environnement d'exécution. Par contre, il est possible de charger des assemblies à l'intérieur de ce domaine. Ces assemblies appelées, Neutral Domain Assemblies, sont « partagées » entre tous les autres domaines. En effet, les assemblies présentes dans le Shared Domain sont compilées une seule fois puis recopiées dans chaque domaine d'application les utilisant. L' assembly recopiée est exécutée. La figure 5, montre le fonctionnement de ce domaine : les assemblies Asm1 et Asm2 sont partagées ; l'Asm2 est recopié dans les domaines B et C. Par défaut, l' assembly mscorlib, contenant les types de bases, est chargée dans le Shared Domain, car elle est utilisée par toutes les applications .Net. Le Shared Domain sert donc a partager du code entre tous les domaines d'applications afin d'améliorer les performances (les assemblies ne sont compilées qu'une fois pour tous les domaines). ROTOR, tout comme la plate-forme .Net de Microsoft dans sa version 1.1, propose 3 politiques de chargement d' assemblies dans le Shared Domain :

– aucune assembly autre que mscorlib (c'est le comportement par défaut)
– seulement les assemblies dites Strong Names
– toutes les assemblies

La troisième approche de plate-formes de services reprend l'alternative deuxième et utilise le Shared Domain pour partager des assemblies entre les différents bundles et la plate-forme. Bien que cela ne résolve pas le problème de performance lors de l'invocation d'un service, le partage de code est dorénavant beaucoup plus rapide. En revanche, il n'est pas possible de décharger une assembly du Shared Domain. De plus, les assemblies présentes, sont accessibles à tous les domaines d'applications, il n'est pas possible de modifier cette portée. Cette alternative ne résout pas non plus le problème des éléments statiques (recopiés dans chaque domaine d'application).

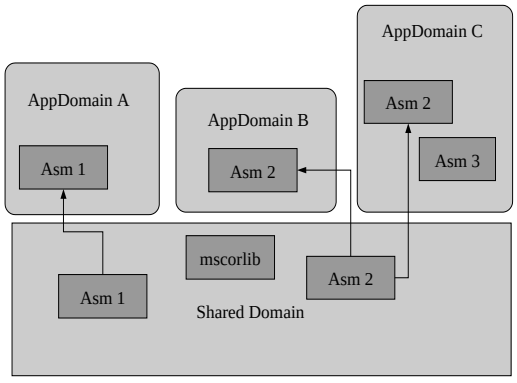

FIG. 5 – Utilisation du Shared Domain

Pour implémenter cette solution, nous avons forcé le chargement de certaines assemblies dans le Shared Domain (nom fort, et « heavy trusted »), de plus ROTOR a été configuré pour utiliser la 2° politique de partage. Il est néanmoins possible d'exécuter cette alternative sur la machine .Net officielle, en effectuant les même réglage. Mais, Microsoft a en projet de rendre transparente l'utilisation du Shared Domain : le

CLR décidera si l' assembly doit être chargée dans le Shared Domain ou non.

4.4. Utilisation des chargeurs de classes des Assemblies

L'étude de ROTOR a permis de découvrir d'autres aspects intéressants de la plate-forme .Net : chaque assembly contient un chargeur de classes (figure 6). Cette dernière alternative s'intéresse directement à la politique de chargement de classes de .Net. En effet, comme nous l'avons exposé auparavant, la plate-forme de services OSGi propose un système de chargement de classes permettant à 2 bundles de partager des paquetages. Cette approche, est une tentative de modification de la politique de chargement de classes de .Net. Les chargeurs de classes, contenus dans chaque assembly, a pour but de gérer les classes de l' assembly en les transformant en objets EEClass (Executive Environnement Class), il ne s'agit pas d'objet .Net, car ils ne sont pas manipulables à partir de la plate-forme .Net.

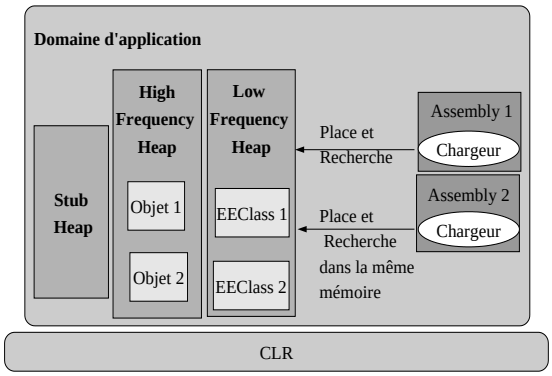

FIG. 6 – Chargeur de classes interne des assemblies

Chaque EEClass possède les méta-données d'une classe (tableau des méthodes ...), et est liée au code de la classe. Le chargeur de classes de l' assembly charge les classes en provenance de son assembly et place chacune d'entre elle dans une mémoire nommée Low Frequency Heap appartenant au domaine d'application. Cette mémoire n'est pas explorée par le ramasse miettes de la machine virtuelle. En effet, le ramasse miettes de .Net ne scrute que le High Frequency Heap, situé lui aussi dans le domaine d'application, contenant tous les objets créés à partir de .Net. Les classes étant « normalement » des objets à longue durée de vie, elles ne sont pas examinées. Cela implique que dès qu'une classe est chargée, elle n'est déchargée que lorsque le domaine d'application est détruit.

Le fait que les classes soient stockées dans une mémoire du domaine d'application explique pourquoi toutes les classes chargées sont disponibles à toutes les autres classes du domaine. Lorsqu'un type (classe, structure, ou type primitif) est demandé, le Low Frequency Heap est examiné et un pointeur sur le type demandé est renvoyé.

Une modification du classloader interne des assemblies a été réalisée afin de modifier le chargement des classes de .Net. L'idée était de permettre uniquement le chargement et l'utilisation des classes contenues dans l' assembly. Il n'était donc pas possible de charger des classes provenant d'autres assemblies (sauf pour les types de base). Pour cela, une nouvelle classe en C++ a été crée étendant la classe « classloader » de Rotor. De plus la méthode CSharp de chargement dynamique (appDomain.load()) a été modifiée. Pour finir, le constructeur de la classe « assembly » a été modifié pour créer des assemblies utilisant le nouveau chargeur.

Ce prototype ne fonctionne pas de façon optimale, renvoyant fréquemment des erreurs concernant le Low Frequency Heap. Néanmoins, nous avons vu qu'il était possible de modifier .Net afin d'adapter le chargement des classes.

4.5. Comparaison

Nous pouvons comparer nos 4 alternatives dans le tableau suivant selon les critères présentés en début de section :

Aspect étudié	*1 seul domaine d'application*	*Multi domaine d'application*	*Utilisation du Shared Domain*	*Modification de la CLR*	*OSGi (JVM)*
Apparition dynamique des services	Oui	Oui	Oui	Oui	Oui
Chargement dynamique	Oui	Oui	Oui	Oui	Oui
Déchargement dynamique	Non	Oui	Non	Non	Oui
Invocation sans proxy	Oui	Non	Non	Oui	Oui
Comportement des attributs statiques	Attributs statiques uniques	1 copie par domaine	1 copie par domaine	Attributs statiques uniques	Attributs statiques uniques
Modification de la CLR	Non	Non	Non	Oui : politique de chargement des classes	JVM standard depuis 1.1

TAB. 1 – Chargeur de classes interne des assemblies

5. Conclusion et Perspectives

Les plates-formes dynamiques de services permettent la construction d'applications dont l'architecture évolue au cours de l'exécution. Cette propriété est principalement liée au besoin de certaines applications d'être sensible au contexte (context-aware) ou bien de pouvoir charger/mettre à jour/décharger certains constituants de l'application sans interrompre globalement l'exécution de l'application. Les principales PDS sont actuellement basées sur la plate-forme Java (machine virtuelle + environnement d'exécution). JINI est la figure emblématique des PDS distribuées pour les applications spontanées sur un réseau de machines « ambientes ». OSGi est une PDS centralisée dans laquelle les services se partagent la même machine virtuelle. Un des points complexes pour la réalisation d'une PDS centralisée est le chargement, le partage et le déchargement des classes (c.a.d interface ou class) représentant les contrats et les implémentations.

Dans cet article, nous nous sommes intéressés à présenter des propositions pour implémenter des plates-formes dynamiques de services centralisées au dessus de Microsoft .NET le compétiteur de la plate-forme Java. Nous avons implémenté et comparé ces solutions selon les critères suivants qui sont disponibles avec OSGi : l'apparition dynamique des services, le chargement dynamique du code, le déchargement dynamique partiel, la visibilité du code chargé (c'est à dire la portée des classes), l'invocation de services, le comportement des attributs statiques. De plus, une des solutions oblige à modifier la CLR.

La conclusion de ce travail est que pour l'instant, la plate-forme .NET ne permet pas de réaliser une PDS centralisée, optimisée comme celle d' OSGi au dessus de Java. Pour l'instant, Java remporte le match

grâce à OSGi. Cependant, au vue de l'engouement pour OSGi dans des domaines générant une forte valeur ajoutée, Microsoft pourrait bien retailler sa CLR en vue d'obtenir des chargeurs de classes en adéquation avec les besoins des PDS centralisés comme OSGi.

6. Bibliographie

1. Bieber, G., Carpenter, J., Introduction to Service-Oriented Programming, OpenWings whitepaper, Septembre 2001, http ://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf
2. H. Cervantes, « Vers un modèle à composants orienté services pour supporter la disponibilité dynamique », Thèse de Doctorat d'Informatique de l'Université Joseph Fourier, Mars 2004, http ://www-adele.imag.fr/Les.Publications/reports/PHD2004Cer.pdf.
3. H. Cervantes, R. Hall, « A Framework for Constructing Adaptive Component-based Applications : Concepts and Experiences », Actes de 7th Symposium on Component-Based Software Engineering (CBSE), Mai 2004, Edinburgh, Ecosse, http ://www-adele.imag.fr/Les.Publications/intConferences/CBSE2004Cer.pdf
4. R. Hall, H. Cervantes, « Gravity : Supporting Dynamically Available Services in Client-Side Applications », Actes de ESEC/FSE, Septembre 2003, Helsinki, Finlande, http ://www-adele.imag.fr/Les.Publications/intConferences/ESEC2003Hal.pdf
5. Ken Arnold, Bryan O'Sullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, Bryan O'Sullivan , « The Jini Specification », The Jini Technology Series, 1 edition, Juin 1999, Addison-Wesley Pub Co ; ISBN : 0201616343
6. OSGi Consortium, The Open Services Gateway Service-Platform Release 3, IOS Press , Mars 2003, ISBN 1 58603 311 5
7. OpenWings,http ://www.openwings.org
8. Venners, Bill, « Inside the Java Virtual Machine », McGraw-Hill Companies , 1997, ISBN 0-07-135093-4
9. S. Liang, G. Bracha, « Dynamic class loading in the Java Virtual Machine ». Actes de ACM Symp. on Object-Oriented Programming : Systems, Languages and Applications 1998, volume 33(10) of Sigplan Notices, pages 36–44. ACM Press, Octobre 1998
10. Halloway, Stuart Dabbs, « Component Development for the Java Platform », Addison-Wesley Pub Co , 2001, ISBN 0201753065
11. Bizzotto, G, « JITS, Java in the Small », Rapport de DEA, LIFL, Université Lille 1, Juin 2002, http ://www.lifl.fr
12. R. Hall, "Modularity for Java and How OSGi Can Help" présentation invitée à la conférence DECOR04, Octobre 2004, http ://hal.ccsd.cnrs.fr/ccsd-00003299
13. R. Hall and H. Cervantes, « An OSGi Implementation and Experience Report », Actes de IEEE Consumer Communications and Networking Conference (CCNC), Janvier 2004, Las Vegas, USA, http ://www-adele.imag.fr/Les.Publications/BD/CCNC2004Hal.html
14. OSCAR : An OSGi framework implementation http ://oscar.objectweb.org/
15. Hall, Richard S - 2004 - A policy-driven Class Loader to Support Deployment in Extensible Frameworks Component Deployment : Second International Working Conference , 3083 / 2004
16. Thai, Thuan L. and Lam, Hoang, « .NET Framework Essentials », O'Reilly, 2003, ISBN 0-596-00505-9
17. MSDN - Microsoft Developer Network- http ://msdn.microsoft.com
18. Ntuba, Jemea, Desing and Implementation of an OSGi Service Architecture for the .Net Platform, Free University of Berlin Faculty of Mathematics and Computer Science, 2004
19. Courtney, J., « Personal Basis Profile Specification », Sun Microsystem , 2001
20. Calder, B., « Java TV specification 1.0 », Sun Microsystem , 2000
21. Stutz, David and Neward, Ted and Shilling, Geoff, « Shared Source CLI Essentials », O'Reilly , 2003, ISBN 0-596-00351-X