RenPar'17 / SympA'2006 / CFSE'5 / JC'2006
Canet en Roussillon, 4 au 6 octobre 2006

Distribution transparente et dynamique de code pour applications Java

Nicolas Geoffray, Gaël Thomas, Bertil Folliot

Laboratoire d'Informatique de Paris 6
8 rue du Capitaine Scott
75015 PARIS - France
prenom.nom@lip6.fr

Résumé
La délégation d'exécution de code est un mécanisme prometteur pour les systèmes nomades et la répartition de charge. Les systèmes nomades pourraient profiter des ressources voisines pour leur envoyer du code à exécuter, et les serveurs Web pourraient répartir leur charge durant un pic de charge. Ce papier présente une infrastructure capable de distribuer une application Java existante entre plusieurs machines et de manière transparente. Le code source des applications ne nécessite pas de modification : nous utilisons le tissage dynamique d'aspects pour analyser (monitorage) et distribuer l'application pendant l'exécution. L'infrastructure est composée d'un environnement d'éxécution extensible et adaptable et de JnJVM, une machine virtuelle Java flexible étendue avec un tisseur d'aspect dynamique. Notre système est chargé pendant l'exécution lorsque l'application le nécessite, sans la redémarrer. Une première évaluation montre que notre système augmente le nombre de transactions par secondes d'un serveur Web de 73%.

Mots-clés : Distribution transparente, Machine virtuelle, Tissage dynamique d'aspects, Répartition de charge

1. Introduction

Certains systèmes s'exécutent dans des environnements dont il est difficile de prédire l'évolution. Par exemple un serveur Web est sujet à des périodes courtes et aléatoires de rafales de requêtes (l'effet *Hot spot of the week* [24]), et des périodes où les requêtes arrivent par nombre limité. Il est difficile de dimensionner un tel système pour garantir une exécution sans fautes y compris pendant les périodes de forte charge. Un autre exemple concerne les systèmes nomades, qui peuvent se trouver par moments dans des lieux où de fortes ressources de calcul sont disponibles, puis dans des lieux sans aucune autre ressource.
Pour répondre à ces situations non prédictibles, nous proposons d'injecter à chaud un système de délégation (ou migration) d'exécution dans l'environnement d'exécution lorsque les ressources locales sont surchargées et des ressources extérieures deviennent disponibles. La délégation d'exécution offre de meilleures performances et disponibilité qu'une exécution uniquement locale. Le système de délégation est enlevé lorsqu'il n'est plus nécessaire. Les principaux avantages de notre solution sont :

- L'environnement d'exécution charge le système de délégation de code dynamiquement : l'application ne doit pas être redémarrée pour en bénéficier.
- Le code de notre solution est indépendant de l'application. Une application avec une nette séparation entre objets dépendants et indépendants de la plate-forme peut bénéficier de notre système.
- Les politiques de migration peuvent être adaptées dynamiquement : par exemple quel code déléguer et sous quelles contraintes (mémoire ou calculs). Notre système est développé au-dessus d'un environnement d'exécution qui permet une adaptation dynamique.
- Les ressources externes ne sont pas dédiées à une application : elles peuvent être partagées entre plusieurs applications.

– Les ressources externes n'ont pas à installer une machine virtuelle spécifique : notre système envoie du code à des environnements standard.

La migration de code s'effectue au-dessus de JnJVM [29] une machine virtuelle Java (JVM) adaptable. JnJVM est enrichie avec un tisseur dynamique d'aspects [19] qui permet de modifier l'application pendant son exécution. Le tisseur d'aspects nous permet d'ajouter du monitorage et la distribution de code dans l'application. Nous utilisons JnJVM pour ces travaux car (i) elle s'exécute dans le même environnement que le système de migration de code (ii) son tisseur dynamique d'aspects est efficace et fonctionnel.

La Section 2 décrit l'environnement d'exécution du système qui permet la modification dynamique de l'application ainsi que l'adaptation des politiques de migration. La Section 3 décrit ce qu'il est possible de déléguer dans un programme Java. La Section 4 présente notre système de délégation d'exécution. Dans la Section 5 nous présentons une évaluation du système. La Section 6 décrit les travaux similaires. La Section 7 présente des perspectives et conclut.

2. L'environnement d'exécution

Le système de migration de code s'appuie sur trois mécanismes différents : chargement dynamique de code, adaptation de l'environnement d'exécution et adaptation de l'application. La micro machine virtuelle (MVM) offre les deux premiers mécanismes et JnJVM offre le dernier. Cette section décrit ces deux environnements.

2.1. La micro machine virtuelle

La MVM est un compilateur et un tisseur de code dynamique. Elle compile des composants, dont l'architecture permet de les adapter pendant l'exécution. Un composant contient des champs et des méthodes. Deux mécanismes sont offerts par l'architecture composants : le remplacement d'un composant par un autre et la modification de l'implantation d'une méthode d'un composant par une autre. Le tisseur dynamique de code permet d'insérer les nouvelles implantations. Le compilateur de la MVM est décrit dans [21]. La MVM est une extension du système décrit dans [22]. La MVM est aussi l'environnement d'exécution de plusieurs projets, par exemple un cache Web [18] ou un ORB (Object Request Broker) flexible [17].

Dans ce projet, la MVM est utilisée pour charger dynamiquement le code du système de migration, et adapter les politiques du système pendant l'exécution.

2.2. JnJVM

JnJVM est une JVM qui suit la spécification J2SE 1.2 à 1.4 [8]. JnJVM utilise les classes de base de Gnu Classpath. Elle est construite au-dessus de la MVM : elle étend la MVM avec des composants spécifiques pour une JVM comme un compilateur de bytecode (JIT) ou un chargeur de classes Java. JnJVM est équivalente en performance à des JVMs libres comme Kaffe version 1.1.6 et est trois fois plus lente que les JVMs industrielles comme IBM version 1.4.2. Les tests ont été effectués avec le JavaGrande Forum Benchmark Section 2 sur un PowerPC et un Pentium. Nous pensons qu'un travail d'ingénierie permettrait d'avoir de meilleurs résultats car le compilateur et le ramasse-miettes de JnJVM ne sont pas optimisés. JnJVM prend une place mémoire de 20 Mo et est constituée de 10000 lignes de code pour la MVM.

Tissage dynamique d'aspects dans JnJVM

Un aspect est une association entre un code existant et un nouveau code. Tisser un aspect est l'action de modifier le code existant pour utiliser le nouveau code. Dans ces travaux, nous utilisons le tissage dynamique d'aspects : les aspects sont tissés pendant l'exécution. La programmation orientée aspects introduit aussi les notions de `coupe` et de `point de jonction`. Un point de jonction est un point d'exécution du programme (par exemple un accès à un champ d'un objet ou l'appel d'une méthode) et une coupe est un ensemble de points de jonction. La MVM, et donc JnJVM, laisse des points de jonction dans tout le programme : il est ainsi possible de modifier tout appel de méthode, ou tout accès à un champ. Les deux aspects présentés dans ces travaux concernent les appels de méthodes.

Le tisseur dynamique d'aspect de JnJVM est présenté dans [19]. Dans son implantation, le tissage dynamique d'aspect sur une méthode n'affecte que les prochains appels sur cette méthode. Le tisseur

n'effectue pas de remplacement de la pile d'exécution.

3. Migration de code pour Java

3.1. Discussion sur la migration d'objets de classes ou de méthodes

Java manipule différentes entités : des classes, des objets, des méthodes et des threads. Migrer un objet signifie envoyer l'objet vers une machine externe, migrer une classe signifie migrer toutes les instances de cette classe, et migrer une méthode signifie que son exécution (et toutes les méthodes qu'elle appelle) est migrée. Ces entités sont toutes des candidates pour notre système de migration de code, chacune ayant ses avantages et inconvénients décrits dans la suite de cette section. La migration d'objets et de classes est surtout pour gagner de la mémoire, tandis que la migration de méthode permet d'optimiser les calculs. Finalement, la migration de threads est plus complexe à implanter. Elle nécessite d'avoir des machines virtuelles spécifiques des deux cotés (machine locale et machines externes), car la JVM standard interdit de capturer et restaurer une pile d'exécution.

Migration d'objets

Le choix des objets à migrer peut être fait soit en analysant le bytecode ou le source d'une application soit pendant l'exécution. Dans le premier cas, il existe des algorithmes [26] pour déterminer les relations entre les objets de l'application et ainsi connaître quels objets sont utilisés intensément. Le code de l'application est alors modifié pour rendre les objets choisis distants. Dans le second cas, l'environnement d'exécution analyse l'utilisation des objets de l'application pendant l'exécution et récupère des informations sur ces objets. Avec ces informations, un graphe est créé. Les arcs entre les objets représentent leurs interactions, et le poids d'un objet représente les autres informations, par exemple la taille de l'objet. Un algorithme de partition de graphe est alors appliqué pour choisir les objets les plus intéressants pour la migration. Le défaut de l'analyse statique par rapport à l'analyse dynamique est que les informations collectées ne sont pas aussi précises, et les interactions entre objets peuvent être trompeuses, par exemple lorsque les objets migrés sont utilisés intensément pendant le démarrage, mais peu après.

Migration de classes

Migrer une classe demande à la JVM de trouver toutes les instances de cette classe, et de les envoyer à une ressource externe. Choisir une classe pour la migration peut être fait statiquement ou dynamiquement. De même que pour la migration d'objets, un graphe est calculé selon les interactions entre classes, et le partitionnement de ce graphe permet de choisir les classes les plus intéressantes pour la migration.

Migration de méthodes

La migration de méthodes est plus simple à implanter que les deux migrations précédentes. Une analyse statique permet de calculer un graphe d'appels de méthodes. Une analyse dynamique récupère des informations sur l'utilisation des méthodes comme le nombre de fois où la méthode est appelée, son temps d'exécution, ou les données échangées (les objets en arguments et l'objet retour). Comparée à la migration de classes, l'analyse est plus difficile car un programme possède plus de méthodes que de classes. La migration de méthodes est surtout utilisée pour les performances, alors que la migration de classes est utilisée pour sauver de la mémoire ; les systèmes nomades utilisent généralement un interpréteur de bytecode, car une méthode compilée occupe de la place mémoire. La migration d'une méthode empêche des communications importantes entre les ressources externes et la JVM locale : l'exécution d'une méthode à distance n'effectue pas des allers-retours entre JVMs au contraire de la migration de classes ou d'objets : avec la migration d'objets ou de classes, les méthodes appelées au sein d'un objet migré sont renvoyées vers la JVM locale.

3.2. Première implantation : migration de méthodes

Dans notre prototype, nous avons implanté la migration de méthodes. Nous travaillons sur l'intégration d'analyseurs statiques pour objets [26], ou classes [9] pour permettre au système un choix plus fin sur ce qu'il peut migrer. Des analyseurs statiques existent pour des langages sûrs [11] et pourraient être transposés au monde Java pour nos travaux.

La migration de méthodes nécessite de connaître quelles méthodes de l'application peuvent effecti-

vement être migrées. Les méthodes possibles ne doivent pas manipuler des objets dépendants de la machine locale (une socket par exemple), et ne doivent pas appeler des méthodes qui manipulent ces objets. Cette analyse est actuellement manuelle, mais nous travaillons sur un outil pour la rendre automatique. Les informations données pour une méthode candidate pour la migration sont : (i) les classes qu'elle utilise pendant son exécution, c'est-à-dire les classes des méthodes qu'elle appelle et les classes de ses arguments, (ii) les objets statiques qu'elle utilise (et ceux qui sont utilisés par les méthodes qu'elle appelle) et (iii) quels arguments et objets statiques elle modifie (voir Section 4.4.2).

4. Système de migration

L'environnement d'exécution, la MVM, est enrichi avec un système qui migre l'exécution ou la mémoire vers des ressources externes. Dans cette section, nous présentons comment l'environnement est modifié pour analyser l'exécution et la distribuer. La Figure 1 donne une vue d'ensemble du système.
Le système de migration étend la MVM. Ses principales activités sont :
– Analyser (monitorage) dynamiquement l'application pendant son exécution. Les résultats permettent de savoir ce qui est le plus intéressant à migrer.
– Distribuer les entités Java. L'environnement doit distribuer l'application de manière transparente.
– Administrer les ressources externes. Celles-ci peuvent être ajoutées et supprimées dynamiquement. Le système doit prendre en compte toute modification de son environnement extérieur.
– Adapter la distribution grâce à un agent autonome. L'agent analyse les résultats du monitorage et décide de quelles méthodes migrer et sur quelles ressources.
Le système ne nécessite ni une modification de l'application, ni l'exécution de machines virtuelles Java spécifiques sur les ressources externes. Le code envoyé peut être exécuté par toute JVM, afin de pouvoir utiliser un large champ de ressources. Le système suit les étapes suivantes :

1. L'analyse statique du bytecode de l'application donne ce qui peut être migré au sein de l'application,
2. Un agent est lancé dans le système : il tisse un aspect de monitorage dans l'application et décide quand la migration doit être faite
3. Les ressources externes s'enregistrent auprès du système,
4. En utilisant les données collectées par le monitorage des méthodes Java données à l'étape 1, l'agent tisse l'aspect distribution au sein de l'application. L'aspect distribution envoie les méthodes aux ressources externes.
5. Lorsque la migration n'est plus nécessaire, l'aspect distribution est enlevé.

Notre implantation est composé de 2000 lignes pour le gestionnaire de ressources, l'agent de distribution et la gestion de la cohérence mémoire (voir Paragraphe 4.4.2). Les deux aspects distribution et monitorage sont implantés en 300 lignes.

4.1. Administration
Deux entrées au sein du système sont fournies pour l'administrer. La première cible les systèmes nomades. La seconde les serveurs Webs. Elles permettent (i) d'administrer les choix de migration, et (ii) d'enregistrer les ressources externes. L'utilisation d'un système de découvertes de services comme Jini [12] est en projet. Les ressources externes peuvent être des grilles, des serveurs ou des ordinateurs personnels.

Administration pour les systèmes nomades

Pour les systèmes nomades, le système de migration est enrichi d'une socket de communication. Des instructions peuvent être envoyées au travers de cette socket : par exemple un administrateur peut envoyer une nouvelle liste de méthodes candidates pour la migration, ou forcer la migration d'une méthode.

Administration pour serveurs Web

Nous utilisons des servlets pour administrer un serveur Web enrichi de notre système de migration. Une servlet est un objet Java qui reçoit des requêtes et renvoie des réponses. Elle fait partie de la spécification

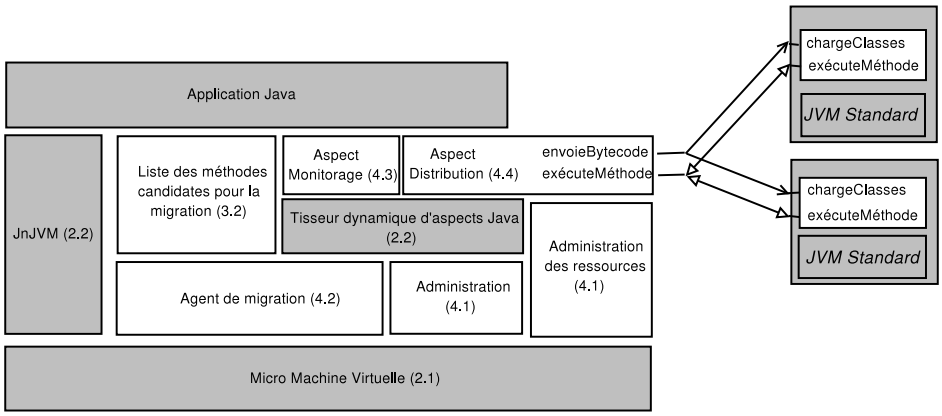


FIG. 1 – Architecture du système de migration

J2EE. Dans un serveur Web, elle crée dynamiquement des pages Web selon le contenu des requêtes. Une servlet est exécutée au sein d'un conteneur de servlets. L'évaluation de notre système utilise Tomcat, un conteneur de servlets. Ainsi, nous avons préféré utiliser une approche servlet plutôt qu'une interface graphique. La servlet a trois fonctionnalités : (i) communiquer avec le système de migration, en récupérant des instructions données par un administrateur au travers d'une page Web, (ii) imprimer sur une page Web l'état des méthodes (local ou distant) et leurs statistiques d'utilisation et (iii) recevoir les requêtes d'enregistrement des ressources externes.

4.2. Agent de migration

La servlet et la socket de communication sont des entrées qui permettent à un utilisateur d'administrer le système de migration. L'utilisateur peut adapter le système selon l'environnement courant : charge de la machine locale, ressources externes disponibles, utilisation mémoire, etc. Cette tache n'est pas acceptable dans le long terme car elle nécessite une présence humaine permanente. Nous avons donc enrichi le système de migration avec un agent autonome qui effectue le travail d'adaptation.

L'agent est composé de capteurs et d'états, et est exécuté en tant que thread MVM. L'implantation actuelle de l'agent permet de répartir la charge d'une application exécutée sur Linux. L'agent sonde la charge du système d'exploitation (il lit le fichier `/proc/loadavg`), et commute entre deux états : *chargé* et *non chargé*. L'état chargé migre les méthodes dont l'exécution serait plus intéressante si elle était distante. Le choix se fait en comparant le nombre d'octets envoyés avec la qualité du réseau, et en comparant le temps d'exécution de la méthode avec le temps d'exécution de l'aspect de distribution. L'état non chargé change l'exécution des méthodes migrées pour qu'elle devienne locale, c'est-à-dire supprime l'aspect distribution.

Parce qu'il est implanté avec la MVM, l'agent peut être modifié dynamiquement en lui ajoutant de nouveaux capteurs ou de nouveaux états. Ainsi la sélection des méthodes à migrer peut être adaptée dynamiquement. Par exemple, si la mémoire est une contrainte, la lecture du fichier `/proc/meminfo` peut être ajoutée, ainsi qu'un nouvel état *mémoire contrainte*. Cet état permettrait de passer d'une politique de migration de méthodes à une politique de migration d'objets ou de classes. Nous prendrons en considération les problèmes d'espace mémoire lorsque le système pourra migrer des classes ou des objets.

4.3. L'aspect monitorage

Le système utilise le tissage dynamique d'aspects Java pour introduire du monitorage dans les méthodes données par l'analyse statique. L'aspect monitorage récupère trois informations. La première est le temps d'exécution moyen afin de connaître les méthodes les plus coûteuses. La seconde est le nombre de fois où une méthode est appelée, afin de connaître les méthodes populaires de l'application. La troisième est la taille mémoire des objets utilisés par la méthode (les arguments, les objets statiques, et l'objet retour) pour connaître le coût réseau de la migration.

L'agent utilise ces informations pour choisir quelles méthodes sont les plus intéressantes pour la migration. Un administrateur ou l'agent peut modifier l'aspect de monitorage pour qu'il récupère d'autres informations.
Notre prototype récupère les informations sur l'exécution des méthodes et sur les caractéristiques des ressources externes. La prise en compte de la qualité réseau est une amélioration en cours.

4.4. L'aspect distribution

L'exécution de méthodes à distance rend l'exécution de l'application distribuée, où les ressources externes offrent leur temps processeur, et où le serveur leur envoie du code.
Nous utilisons le canevas RMI de distribution pour des raisons de facilité. Chaque ressource externe exécute un objet RMI qui exporte deux méthodes : la première, `receiveFiles` reçoit le bytecode Java des méthodes migrées, et la seconde `remoteMethodInvocation` provoque l'exécution d'une méthode.

4.4.1. Algorithme de l'aspect distribution

Le serveur qui exécute l'application possède une référence pour chaque ressource externe. Une référence est une souche avec laquelle le système peut appeler les deux méthodes RMI. Lorsque le système décide de migrer une méthode, il tisse l'aspect distribution sur la méthode qui se charge de :

1. Récupérer les objets statiques utilisés par la méthode,
2. Placer les arguments dans un tableau,
3. Choisir une ressource externe en récupérant sa souche,
4. Envoyer le bytecode de la méthode (appel à `receiveFiles`) si la ressource externe ne le possède pas,
5. Invoquer la méthode avec `remoteMethodInvocation`,
6. Attendre la fin de l'exécution distante,
7. Récupérer un tableau d'objets pour mettre à jour les objets modifiés (voir Paragraphe 4.4.2),
8. Retourner le résultat de la méthode.

L'appel est synchrone : lorsqu'un thread fait un appel à la méthode `RemoteMethodInvocation`, il doit attendre le résultat.
Lorsque le système décide que la migration de la méthode n'est plus nécessaire, il lui enlève l'aspect distribution.

4.4.2. Difficultés d'une distribution transparente

Parce que nous distribuons de manière transparente une application, nous devons résoudre les problèmes inhérents d'une distribution d'exécution. Nous discutons à présent comment notre système gère certains de ces problèmes comme la sémantique d'appels à distance, la cohérence mémoire ou la tolérance aux fautes.

Sémantique d'appels à distance

Des canevas de distribution comme RMI ont une sémantique d'appels par copie : une copie des arguments est envoyée et les modifications effectuées à distance sont perdues lorsque l'exécution distante de la méthode est terminée. Cette sémantique n'est pas possible avec notre système car nous effectuons une distribution transparente, et les appels distants doivent avoir le même effet sur les objets qu'un appel local. Nous utilisons une sémantique de copie-restauration pour obtenir cette transparence.
L'algorithme de copie-restauration est coûteux : pour un objet `Obj`, l'appel distant doit renvoyer le nouveau `Obj` modifié, et tous les objets que l'ancien `Obj` référence. L'application qui a fait un appel distant reçoit tous ces objets et doit comparer tous les objets locaux référencés par `Obj` pour les comparer avec les nouvelles références.
Une optimisation de cette sémantique est d'effectuer une analyse de l'application afin de connaître quels objets sont modifiés par l'appel distant. Ces objets seront accompagnés d'une sémantique de copie-restauration alors que les autres objets auront une sémantique uniquement par copie. C'est la raison pour laquelle nous récupérons lors de l'analyse statique les paramètres et objets statiques modifiés par la méthode.

Une implantation non standard de l'appel par copie-restauration existe pour Java, appelée NRMI [31]. Nous n'avons pas utilisé cette implantation car elle modifie les classes standard de Java, et JnJVM utilise Gnu Classpath. De plus, dans NRMI, le code de l'application doit être modifié pour spécifier quels objets ont une sémantique de copie-restauration. Comme nous n'effectuons pas un appel direct à la méthode migrée, mais utilisons la méthode `remoteMethodInvocation` d'un objet RMI qui connaît quelle méthode doit être exécutée, nous pouvons implanter la sémantique de copie-restauration hors du canevas RMI. La sémantique est ajoutée dans l'aspect distribution [1]. L'implantation de l'objet RMI renvoie au serveur un tableau de tous les objets modifiés par la méthode.

Cohérence mémoire

Lorsqu'une ressource externe exécute une méthode, les objets envoyés comme arguments sont dupliqués. Il peut donc y avoir deux copies d'un même objet. Pour une application mono-threadée, l'exécution est arrêtée localement, et attend le résultat de l'appel distant. Aucune modification des objets ne peut être effectuée localement, et uniquement l'appel distant peut les modifier. En revanche, si l'application est multi-threadée, les objets envoyés sont manipulés localement et à distance. Ce problème ne concerne pas le système de migration, mais le programmeur : si les objets ne sont pas protégés dans le programme avec le mot clé Java `synchronized`, alors l'exécution d'une méthode à distance ne change pas la sémantique de l'application.
Les méthodes synchronisées peuvent être migrées, mais les appels `synchronized` au sein d'une méthode migrée demandent une attention particulière. Comme l'appel synchronisé est effectué sur une JVM distante de la JVM locale, il doit être envoyé à la JVM locale. Ce problème est complexe : une première solution serait de modifier la sémantique des appels synchronisés sur les JVMs externes pour informer la JVM locale, mais cette solution n'est pas satisfaisante car elle nécessite d'exécuter sur les ressources externes des JVMs modifiées. Nous réfléchissons actuellement sur tisser dynamiquement un aspect sur les méthodes qui font des appels synchronisés : cet aspect ajouterait des appels distants pour informer la JVM locale d'un appel synchronisé.

Tolérance aux fautes

La migration de méthodes permet à notre système d'être partiellement tolérant aux fautes : si une ressource externe qui exécute une méthode tombe en panne (détecté par timeout), la méthode peut être exécutée sur une autre ressource, car toutes les modifications faites sur les objets par la ressource tombée en panne ne sont pas enregistrées au niveau de la JVM locale. Cette approche n'est cependant pas possible pour des applications qui interagissent avec des entités externes, comme des bases de données. Pour ce type d'application, l'utilisation de points de retour sera nécessaire. De plus, lorsque le système pourra migrer des objets et des classes, nous devrons répliquer les objets distribués, pour garantir une tolérance aux fautes.

5. Évaluation

Nous comparons JnJVM sans migration et JnJVM avec migration pour une application Web. La comparaison avec des JVMs industrielles n'est pas encore appropriée : JnJVM et Gnu Classpath n'ont pas encore leur maturité, et nécessitent des efforts d'ingénierie (voir Section 2.2).
L'application Web est une implantation libre de TPC-W. [2] L'application est un magasin interactif implanté avec des servlets, et qui communique avec une base de données MySQL. Le conteneur de servlets utilisé est Tomcat version 5.0.28. Le serveur Web est un PowerBook G4 1 GHz avec 1 Go de mémoire.
L'application comporte 29 classes, dont 15 servlets, 49 méthodes, et 5409 lignes de code. Une analyse manuelle du code permet de trouver 21 méthodes candidates pour la migration (les 28 autres méthodes utilisent des objets spécifiques aux servlets qui manipulent des objets dépendants du système d'exploitation).
Tomcat est lancé avec JnJVM, qui s'exécute au-dessus de la MVM, sans le système de migration. Après le démarrage de l'application, nous chargeons et installons au sein de la MVM le système de migration,

[1] Il est utilisé si nécessaire, car certains canevas peuvent déjà fournir une sémantique de copie-restauration
[2] http ://jmob.objectweb.org/tpcw.html

et lui donnons la liste des méthodes candidates pour la migration récupérées par l'analyse statique. L'agent de migration démarre et tisse l'aspect monitorage sur ces méthodes. Les ressources externes s'enregistrent dynamiquement auprès du système. Pour la première évaluation, les ressources sont un biprocesseur G5 2 GHz et un Pentium P4 3 GHz. Pour la seconde évaluation, une grille de machines de 20 biprocesseurs Xeon 2 GHz s'enregistre auprès du système de migration. Les JVMs exécutées sur ces machines sont des Sun JVM 1.4 pour Pentium et Xeon et IBM JVM 1.4 pour PowerPC. Elles exécutent toutes l'objet RMI présenté dans la Section 4.4.

Nous utilisons l'utilitaire `Siege`[3] pour simuler des utilisateurs qui envoient des requêtes à l'application TPC-W. Les informations récupérées par `Siege` sont : temps d'exécution, données transférées, temps de réponse moyen, taux de transactions, nombre moyen de connexions simultanées et la disponibilité (le nombre de requêtes satisfaites par le serveur).

Pour cette expérience, `Siege` simule de 10 à 400 utilisateurs simultanés. La première configuration est JnJVM sans migration, la seconde est JnJVM avec migration et deux ressources, et la troisième est JnJVM avec migration et une grille de machines. La Figure 2(a) compare le nombre de transactions par secondes. Elle montre que le nombre de transactions par seconde est augmenté de 17% avec deux ressources, et 73% avec la grille. Ces résultats ne dépendent pas du nombre de connexions. La Figure 2(b) compare le temps de réponse moyen du serveur. Lorsque 300 utilisateurs envoient des requêtes simultanément, le temps de réponse moyen est amélioré de 10% avec deux ressources et 42% avec la grille. La Figure 2(c) compare la disponibilité de chaque configuration. La disponibilité du serveur sans migration diminue à partir de 290 utilisateurs connectés simultanément, alors qu'il diminue à partir de 300 utilisateurs avec deux ressources et 360 utilisateurs avec la grille.

Ces résultats montrent que le système de migration de code est performant. Il faut cependant noter que l'augmentation de ressources ne provoque pas une augmentation linéaire des performances. Le serveur Web est toujours un goulot d'étranglement car il doit distribuer les requêtes entre les ressources externes.

6. Travaux existants

Notre système de migration distribue sur des ressources externes une application Java développée comme centralisée. Trois approches dans l'existant permettent d'obtenir cette distribution : l'exécution de l'application sur une grille de machines dédiées, distribution statique et automatique du programme et délégation transparente d'exécution.

6.1. Exécution distribuée sur une grille de machines

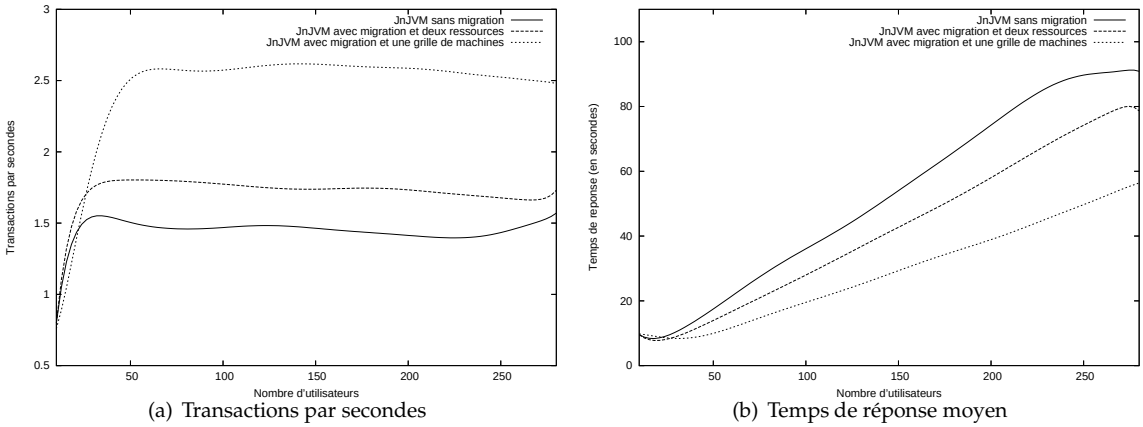
La distribution d'un programme sur une grille peut être implantée avec la migration de thread, une mémoire partagée répartie, ou une machine virtuelle répartie.

Trois approches sont possibles pour la migration de threads : modification du source de l'application [7], modification du bytecode de l'application [32, 23], ou utiliser une machine virtuelle étendue (en l'occurrence celle de Sun) comme Moba [25] ou le système de Bouchenak *et al* [3]. La modification de bytecode ou de source ajoute des instructions au sein de l'application qui permettent de migrer un thread pendant son exécution. Ces instructions sont soit des gestions d'exceptions, ou des tests, et réduisent donc les performances de l'application lorsque aucune migration n'est effectuée. Une machine virtuelle étendue nécessite d'exécuter cette JVM sur chaque site. Notre solution envoie du code exécutable par des JVMs standard. De plus la migration de threads est complexe à implanter par rapport à la migration de méthodes.

La mémoire partagée répartie (MPR) est utilisée pour allouer et manipuler des objets. Les systèmes existants sont Java/DSM [34], MultiJav [4], Javanaise [10] ou Jessica [14]. L'application s'exécute au-dessus d'une grille de machines, où chaque noeud participe à la MPR, et par conséquent exécute une JVM modifiée. L'ajout et la suppression de noeuds ne sont pas considérés dans ces systèmes, ce qui n'est pas satisfaisant pour notre solution car les noeuds doivent pouvoir aller et venir dynamiquement.

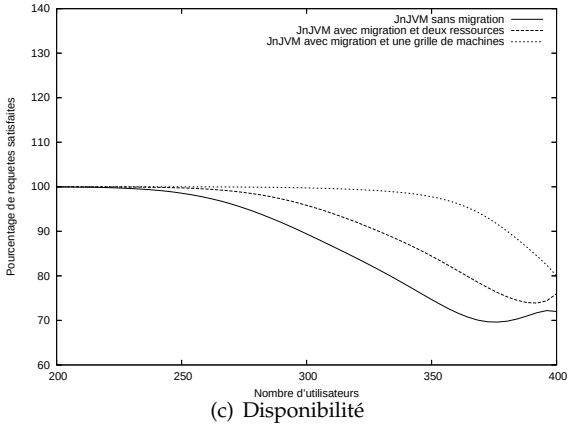
Enfin, une machine virtuelle distribuée, comme cJVM [1] ou JESSICA2 [35] exécute une application au-dessus d'une grille de machines où chaque noeud participe à l'exécution de la JVM. Ces JVMs offrent en général une migration de threads pour la répartition de charge dynamique. Comme pour les machines virtuelles réparties, supprimer ou ajouter un noeud pendant l'exécution n'est pas considéré.

[3] http ://www.joedog.org/siege



(a) Transactions par secondes

(b) Temps de réponse moyen



(c) Disponibilité

FIG. 2 – Évaluation du système de migration

6.2. Distribution automatique

JavaParty [20] étend le langage Java avec un nouveau mot clé `remote`, qu'un développeur peut utiliser pour les classes dont les instances peuvent être migrées. Un pré-processeur est appliqué sur le code de l'application pour écrire le code distribuée. L'exécution de l'application se fait sur une JVM standard. Un registre qui permet de localiser les objets. Notre solution ne modifie pas le langage Java. De plus dans JavaParty, la distribution est choisie de manière statique, alors qu'elle est choisie de manière statique et dynamique dans notre système. Comme JavaParty, nous pouvons contrôler la distribution des objets en spécifiant quelles entités du programme peuvent être migrées. Remote Objects in Java [16] introduit un nouveau mot-clé `remotenew` pour créer des objets distants. Le nouveau mot clé est traduit en un nouvel opcode, que seule une JVM dédiée peut exécuter.

Doorastha [5] permet à un programmeur d'annoter son code Java avec de nouveaux mot clés afin de spécifier quels objets de l'application peuvent être distribués et comment ils sont transmis aux méthodes. Le canevas Doorastha compile le programme et choisit où placer les objets. Le placement peut être décidé statiquement ou dynamiquement. Pangaea [27] utilise un algorithme de graphe d'objets [26] qui analyse le code source de l'application pour trouver les objets qui interagissent le plus. En utilisant ce graphe, Pangaea crée un code distribué pour exécuter l'application sur plusieurs machines. Le canevas de distribution peut être RMI, CORBA ou Doorastha. Le placement des objets et le nombre de machines sont décidés statiquement. J-Orchestra [30] et Addistant [28] modifient le bytecode Java de l'application pour créer un code distribué, selon des informations données par le programmeur. Ces informations spécifient quelles classes doivent avoir leurs instances distribuées. Dans J-Orchestra, ces classes sont transformées en classes RMI. Addistant utilise un système maître-proxy, où les proxies d'un objet communiquent avec l'objet maître. Pangaea, J-Orchestra et Addistant modifient l'application avant son exécution. L'adaptation dynamique n'y est pas considérée, et les objets qui n'étaient pas choisis comme migrés statiquement ne peuvent pas l'être dynamiquement. De plus, ces systèmes n'analysent pas l'application pendant l'exécution, et ne peuvent pas adapter leurs stratégies de distribution.

6.3. Délégation transparente de code

Lattanzi *et al.* [13] améliorent les performances d'un FPGA avec un interpréteur Java en déléguant l'exécution de méthodes populaires d'une application Java à un coprocesseur. La JVM analyse chaque méthode pour trouver quelles méthodes devraient être déléguées. Les méthodes choisies sont compilées et migrées vers le coprocesseur. Cette solution utilise une mémoire partagée et ne doit donc pas faire face aux problèmes de distribution sur un réseau de machines.

Azul Systems [2] font de la répartition de charge pour les applications J2EE. Les applications délèguent l'exécution de réponses Web vers une grille dédiée, composée de matériels dédiés et d'une JVM dédiée. La technologie est récente et spécifique aux applications Web. Notre solution permet de distribuer n'importe quelle application Java.

Diaconescu *et al.* [6] proposent une infrastructure similaire à la notre pour distribuer une application Java. Le système effectue une analyse de l'application inspirée de Pangaea [26] pour créer un graphe d'objets. Ce graphe est analysé pour générer un programme distribué. La machine virtuelle Joeq [33] est utilisée pour compiler l'application originale en une application distribuée. Les communications entre la JVM locale et les ressources externes se font avec MPI (Message Passing Interface). Pendant l'exécution, le système analyse la qualité de la répartition, mais l'utilisateur ne peut pas l'adapter. Notre système s'exécute au-dessus d'un environnement flexible, la MVM, qui facilite l'adaptation des politiques de distribution.

AIDE [15] étend la ChaiVM, une JVM dédiée pour les systèmes embarqués et analyse les interactions entre les classes de l'application pendant l'exécution. Un algorithme de partition de graphe utilise cette analyse pour trouver la meilleure partition. Une partition est un ensemble de classes et le choix se fait selon les communications entre les classes. Les partitions peuvent être envoyées vers des ressources externes. Les ressources externes exécutent une JVM spécifique, alors que notre système envoie du code exécutable sur des JVMs standard.

7. Conclusion

Nous avons présenté un environnement d'exécution qui permet de déléguer des parties de l'exécution d'une application sur des ressources externes. Cet environnement effectue du monitorage sur l'exécution de l'application. Il analyse les résultats pour choisir comment la distribuer.
Nous avons appliqué le tissage dynamique d'aspects pour intégrer monitorage et distribution dans des applications existantes. L'association de ce mécanisme avec un agent de distribution et une plateforme d'exécution adaptable donne un système autonome extensible pour la répartition de charge ou la récupération d'espace mémoire. L'évaluation de notre système montre que la disponibilité et la rapidité d'un serveur Web sont nettement améliorées.
Les applications dont les liens entre les objets indépendants et dépendants du système d'exploitation sont nettement séparés ont plus de chance d'utiliser efficacement notre systême, car la plupart de ses méthodes pourront être migrées.
Les travaux futurs pour améliorer notre prototype incluent :
– Le développement d'un analyseur de méthodes. Pour le moment, l'analyse est manuelle, ce qui n'est pas concevable pour de grandes applications. Cet outil facilitera l'utilisation de notre système.
– La possibilité de migrer des objets et des classes pour permettre de prendre en compte les problèmes d'espace mémoire.
– L'intégration de différentes analyses statiques. L'analyse d'objets au travers d'un graphe de communications [26], ou l'analyse d'interactions de classes [9] donneront de plus amples informations sur ce qu'il est intéressant de migrer.
– L'utilisation de canevas de distribution autres que RMI, qui pourraient donner de meilleures performances.
– Le développement d'un langage pour l'implantation d'agents. Il peut y avoir plusieurs agents différents, selon le domaine où s'exécute notre système.
– L'évaluation de notre système sur plusieurs domaines d'applications. Nous avons évalué le système pour la répartition de charge. La prochaine évaluation concernera la migration avec contrainte mémoire.

Bibliographie

1. Y. Aridor, M. Factor, and A. Teperman. cJVM : A Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing*, pages 4–11, Fukushima, Japan, September 1999.
2. Azul systems. http ://www.azulsystems.com.
3. S. Bouchenak and D. Hagimont. Zero Overhead Java Thread Migration. Technical Report 0261, INRIA, May 2002.
4. X. Chen and V. H. Allan. MultiJav : A Distributed Shared Memory System Based on Multiple Java Virtual Machines. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 91–98, Las Vegas, USA, July 1998.
5. M. Dahm. Doorastha : A Step Towards Distribution Transparency, 2000.
6. R. E. Diaconescu, L. Wang, Z. Mouri, and M. Chu. A Compiler and Runtime Infrastructure for Automatic Program Distribution. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 52–67, Washington, USA, 2005.
7. S. Funfrocken. Transparent Migration of Java-Based Mobile Agents. In *Proceedings of the Mobile Agents Conference*, pages 26–37, 1998.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification (2nd Edition)*. Addison Wesley Longman, Inc., 2000.
9. X. Gu. Adaptive Offloading Inference for Delivering Applications in Pervasive Computing Environments. In *In Proceedings of the IEEE Pervasive Computing and Communication*, 2003.
10. D. Hagimont and D. Louvegnies. Javanaise : Distributed Shared Objects for Internet Cooperative Applications. In *Proceedings of the Middleware Conference*, The Lake District, England, 1998.
11. G. Haïk, J.P. Briot, and C. Queinnec. Automatic Introduction of Mobility for Standard-Based Frameworks. In *Proceedings of the Distributed Objects and Applications Conference*, pages 813–827, November 2005.

12. Jini. http ://www.jini.org.
13. E. Lattanzi, A. Gayasen, M. Kandemir, V. Narayanan, L. Benini, and A. Bogliolo. Improving Java Performance by Dynamic Method Migration on FPGAs. In *Proceedings of the IEEE Reconfigurable Architecture Workshop*, April 2004.
14. M. Ma, C. Wang, and F. Lau. JESSICA : Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10) :1194–1222, 2000.
15. A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, T. Giuli, and X. Gu. Towards a Distributed Platform for Resource-Constrained Devices. Technical report, Hewlett-Packard, 2002.
16. N. Nagaratnam and A. Srinivasan. Remote Objects in Java. In *Proceedings of the IASTED International Conference on Networks*, 1996.
17. F. Ogel, B. Folliot, and G. Thomas. A Step Toward Ubiquitous Computing : An Efficient Flexible Micro-ORB. In *Proceedings of the ACM SIGOPS European Workshop*, pages 176–181, Leuven, Belgium, sept. 2004.
18. F. Ogel, S. Patarin, I. Piumarta, and B. Folliot. C/SPAN : A Self-Adapting Web Proxy Cache. In *Proceedings of the Autonomic Computing Workshop of the International Workshop on Active Middleware Services*, page 178, June 2003.
19. F. Ogel, G. Thomas, and B. Folliot. Support Efficient Dynamic Aspects Through Reflection and Dynamic Compilation. In *Proceedings of the Symposium on Applied Computing*, Santa Fe, USA, March 2005.
20. M. Philippsen and M. Zenger. JavaParty : Transparent Remote Objects in Java. *Concurrency : Practice and Experience*, 9(11) :1225–1242, November 1997.
21. I. Piumarta. The Virtual Processor : Fast, Architecture-Neutral Dynamic Code Generation. In *Proceedings of the Virtual Machine Research and Technology Symposium*, pages 97–110, San Jose, USA, 2004.
22. I. Piumarta, F. Ogel, and B. Folliot. YNVM : Dynamic Compilation in Support of Software Evolution. In *Proceedings of the OOPSLA Engineering Complex Object Oriented System for Evolution Workshop*, Tampa Bay, USA, October 2001.
23. T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *Proceedings of ASA/MA*, pages 16–28, 2000.
24. M. Seltzer. The World Wide Web : Issues and Challenges. Presented at IBM Almaden, 7 1996.
25. K. Shudo and Y. Muraoka. Asynchronous Migration of Execution Context in Java Virtual Machines. *Future Generation Computer Systems*, 18(2) :225–233, Oct. 2001.
26. A. Spiegel. Object Graph Analysis. Technical Report B-99-11, Freie Universitat Berlin, 1999.
27. A. Spiegel. PANGAEA : An Automatic Distribution Front-End for Java. In *Proceedings of the IPPS/SPDP Workshop*, pages 93–99, 1999.
28. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. *Lecture Notes in Computer Science*, 2072, 2001.
29. G. Thomas, F. Ogel, A. Galland, B. Folliot, and I. Piumarta. Building a Flexible Java Runtime upon a Flexible Compiler. In Jean-Jacques Vandewalle David Simplot-Ryl and Gilles Grimaud, editors, *Special Issue on 'System & Networking for Smart Objects' of IASTED International Journal on Computers and Applications*, volume 27, pages 28–47. ACTA Press, 2005.
30. E. Tilevich and Y. Smaragdakis. J-Orchestra : Automatic Java Application Partitioning. In *Proceedings of the European Conference on Object-Oriented Programming*, Malaga, Spain, June 2002.
31. E. Tilevich and Y. Smaragdakis. NRMI : Natural and Efficient Middleware. In *Proceedings of the International Conference on Distributed Computing Systems*, 2003.
32. E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings og the ASA/MA Conference*, pages 29–43, 2000.
33. J. Whaley. Joeq : A Virtual Machine and Compiler Infrastructure. In *Proceedings of the ACM SIGPLAN Conference*, 2003.
34. W. Yu and A. Cox. Java/DSM : A Platform for Heterogeneous Computing. *Concurrency : Practice and Experience*, 9(11) :1213–1224, 1997.
35. W. Zhu, C. Wang, and F. Lau. JESSICA2 : A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *Proceedings of the International Conference on Cluster Computing*, Chicago, USA, September 2002.