Objets actifs mobiles et communicants

Françoise Baude – Denis Caromel – Fabrice Huet – Julien Vayssière

Equipe Oasis, INRIA - CNRS - I3S
2004 route des lucioles
06902 Sophia Antipolis
Prénom.Nom@inria.fr

RÉSUMÉ. *Nous présentons dans cet article une bibliothèque Java, nommée ProActive, conçue et implémentée afin d'appliquer au problème des agents mobiles sur Internet un modèle à objets actifs. Nous montrons comment un système d'agents mobiles peut tirer avantage des objets actifs, et du mode de communication asynchrone avec futurs (promesses de résultat). Notre objectif est ici de démontrer que l'on peut construire une bibliothèque à objets mobiles communicants qui soit à la fois simple et transparente à utiliser, sûre de par sa sémantique, flexible et ouverte de par l'utilisation de la réflexion, mais également performante.*

ABSTRACT. *This article presents the design and implementation of a Java library aimed at applying an active object model to the problem of mobile agents on the Internet. We show how mobile agent systems can benefit from active objects and from the asynchronous communication mode with future objects. Our goal is to demonstrate that it is possible to build a library for providing communicating mobile objects whose use is simple and transparent, whose semantics are reliable, flexible and open thanks to the use of reflexion, and which is efficient, too.*

MOTS-CLÉS : *Migration, asynchronisme, objets actifs, réflexion, méta-objets.*

KEYWORDS: *Migration, asynchronism, active objects, reflection, meta-objects.*

1. Introduction

Les modèles à *objets actifs* proviennent de l'unification des notions d'objet et d'activité, de la même manière que le concept d'objet lui-même provient de l'unification des notions de structure de données et de procédure. Un objet actif se conforme donc à la règle des trois unités : unité de données (objets), unité de code (classes) et unité d'activité (threads). Le concept d'*agent mobile* résulte de la rencontre entre les techniques de programmation distribuée à objets et les techniques de code mobile. Dans son acception la plus courante, la notion d'agent mobile désigne une entité logicielle autonome qui se déplace de machine en machine afin de remplir une tâche.

Même si diverses plates-formes d'agents mobiles ont été proposées (dans le cadre du langage Java, Aglets [VEN 97] et Voyager [VOY 99] pour les plus célèbres), aucune n'intègre de façon réellement transparente les aspects objets actifs communicants d'une part, mobilité d'autre part. Notre objectif dans cet article est tout d'abord de démontrer que ces deux aspects peuvent être unifiés dans un cadre *flexible* offrant une grande *transparence* à l'utilisateur. Un des points clés est la présence d'une communication systématiquement asynchrone avec futurs qui réduit le couplage entre les entités actives mobiles. Une partie importante de la flexibilité (par exemple transformer dynamiquement un objet reçu par télé-chargement de code, sans le source, en un objet actif mobile) est obtenu par l'utilisation de *techniques réflexives* [MAE 88]. Nous nous attacherons à montrer comment elles permettent d'obtenir un système flexible, ouvert et extensible. D'autre part, la flexibilité ne doit pas nuire à la correction de tels logiciels. Aussi, nous avons pour objectif de garantir l'acheminement et la transmission FIFO des messages, pour les requêtes mais également pour les réponses, et cela y compris en présence de migration des émetteurs et des récepteurs. Enfin, obtenir toutes ces caractéristiques n'a que peu d'intérêt si les performances en deviennent inacceptables. Aussi, nous allons étudier les performances obtenues avec deux techniques répandues de localisation des agents mobiles : *répéteurs* (forwarder) et *serveur centralisé*.

En résumé, notre objectif est ici de démontrer que l'on peut construire une bibliothèque à objets mobiles communicants qui soit simple et transparente à utiliser, sûre de par sa sémantique, flexible et ouverte de par l'utilisation de la réflexion, mais également performante.

Cet article s'articule de la manière suivante. La partie 2 présente le modèle à objets actifs et la bibliothèque ProActive sur lesquels nous nous appuyons. Quelques propriétés importantes pour la migration sont détaillées, et les grandes lignes de l'implémentation sont exposées. La partie 3 présente le modèle de migration et les primitives de programmation, ainsi que des abstractions pour la migration construites au-dessus de ces primitives. Outre quelques performances de base, la partie 4 présente en détails deux solutions possibles pour localiser les objets actifs mobiles et des mesures de performances les comparant. Nous présentons les caractéristiques techniques de diverses bibliothèques assez semblables à la nôtre dans la partie 5 et la partie 6 dresse le bilan de ces travaux et donne quelques pistes de recherches futures.

2. ProActive : objets actifs asynchrones communicants

Dans un souci de portabilité, la bibliothèque ProActive s'interdit toute modification du langage Java et de sa machine virtuelle (JVM, pour *Java Virtual Machine*). Nous sommes donc en présence d'une simple bibliothèque, qui utilise les outils standards (`javac` et JVM). Pour des raisons de déverminage des programmes (problème exacerbé en l'occurrence car ils sont répartis et parallèles), ProActive s'interdit également la modification du code source écrit par l'utilisateur (pas de pré-traitement). La seule technique utilisée consiste, outre l'utilisation de techniques de réflexion permettant de manipuler les événements de l'environnement d'exécution (appels de méthodes distantes par exemple), à générer (éventuellement dynamiquement) du code complémentaire à celui de l'utilisateur ; technique qui s'apparente à la notion de bibliothèque *générative* ou *active* [VEL 98, CZA 00].

De plus, la bibliothèque est elle-même écrite entièrement en Java, ce qui autorise l'utilisation de n'importe quelle plate-forme Java et des outils associés, en particulier le fonctionnement dans un environnement ouvert et dynamique avec télé-chargement dynamique de code ; nous utilisons le mini-serveur `http` de la plate-forme Java pour transférer dynamiquement le bytecode nécessaire.

2.1. *Modèle de base*

ProActive offre un modèle de programmation parallèle et répartie que l'on peut qualifier de "*à objets actifs*" [BRI 96] (par opposition à des approches où les activités sont orthogonales aux objets), et de "*hétérogène*" dans le sens où tous les objets ne sont pas actifs. Un des objectifs est de faciliter la programmation répartie, en particulier par le biais de la réutilisation [CAR 93].

En résumé, le modèle de ProActive présente les caractéristiques suivantes :

– des objets actifs et accessibles à distance,

– la séquentialité des activités (processus purement séquentiels),

– une communication par appel de méthode standard,

– des appels systématiquement asynchrones vers les objets actifs,

– un mécanisme d'attente par nécessité (futur transparent),

– les continuations automatiques (un mécanisme transparent de délégation),

– l'absence d'objets partagés,

– une programmation des activités qui est centralisée et explicite par défaut,

– du polymorphisme entre objets standards et objets actifs distants.

Un *appel asynchrone* permet de ne pas se bloquer lors d'une communication. Un *futur* est alors un objet résultat d'un appel asynchrone : initialement il n'a pas encore de valeur. D'une façon duale, l'*attente par nécessité* permet de se bloquer automatiquement si l'on tente d'utiliser le résultat non encore revenu d'un appel asynchrone. Un futur

a) Class-based :

```
class pA extends A implements Active {}
...
A a = (A) ProActive.newActive ("pA", params, Node);
```

b) Instanciation-based :

```
A a = (A) ProActive.newActive ("A", params, Node);
```

c) Object-based :

```
a = (A) ProActive.turnActive (a, node);
```

Figure 1. *Création des objects actifs : Class-, Instanciation-, Object-based*

est *transparent* dans le sens où le programmeur n'a pas besoin d'ajouter du code pour obtenir ce type de synchronisation.

Par rapport à RMI [1], ProActive se caractérise par : (1) la création d'activités à distance (et non pas seulement localement dans la JVM du serveur), (2) des objets distants générés à partir de classes ou d'objets (et non pas uniquement d'interfaces), (3) les appels asynchrones et les futurs transparents, (4) un contrôle fin de l'activité des objets distants, (5) et bien sûr la migration que nous détaillerons largement ultérieurement. Nous allons maintenant présenter succinctement les principales caractéristiques de ProActive – pour plus de détails voir [CAR 98].

2.2. *Création des objects actifs*

La bibliothèque a pour objectif de permettre la création d'un object actif distant le plus simplement possible, à partir de code (donc d'une classe) initialement local et séquentiel. Un objet Java standard, créé par une instruction :

```
A a = new A ("foo", 7);
```

peut être transformé en un objet actif distant de trois manières (Exemple 1).

Du plus statique au plus dynamique, **a) Class-based** passe par la création d'une nouvelle classe qui implémente l'interface-marqueur `Active`[2] et hérite (éventuellement) d'une classe existante. Ce cas permettra d'ajouter des méthodes spécifiques aux aspects répartis, et également de doter l'objet actif d'une activité propre, en place de l'activité FIFO réalisée par défaut (voir plus loin section 2.4). Le second paramètre de cet appel statique (`params`) correspond aux paramètres effectifs à passer au constructeur lors de la création distante. Dans notre exemple, il peut être défini par :

1. Remote Method Invocation, bibliothèque standard de Java
2. Interface marqueur pure, `Active` ne comporte aucune routine et n'impose donc aucune contrainte d'implémentation au programmeur.

```
Object[] params = {"foo", new Integer (7)};
```

Afin d'éviter cette syntaxe, que nous admettons volontiers peu élégante, un motif de conception de type *factory* peut être implémenté dans la classe `pA` (voir [CAR 98]). Le dernier paramètre (`Node`) spécifie la machine virtuelle où doit être placé l'objet actif (c'est habituellement une URL de la forme `//lo.inria.fr/VM1`). Dans le cas d'un node `null`, la création se fait dans la JVM en cours. Un autre objet actif passé en paramètre à la place du paramètre node (`newActive` est surchargée) permet la co-allocation (même machine virtuelle) avec un objet actif déjà existant :

```
A a0= (A) ProActive.newActive("pA",params,null); // JVM en cours
A a1= (A) ProActive.newActive("pA",params,"//lo.inria.fr/VM1");
A a2= (A) ProActive.newActive("pA",params,a1);   // co-allocation
```

La seconde primitive de création, **b) Instanciation-based**, permet de prendre une classe existante standard (qui n'implémente pas l'interface `Active`) et de l'*instancier* directement (sans la modifier) en un objet actif distant. Les deux paramètres (`params` et `Node`) ont bien sûr le même rôle que précédemment.

Enfin, le dernier type de création, **c) Object-based**, est encore plus dynamique puisqu'il prend un objet déjà créé, et en fait un objet actif accessible à distance. La sémantique est la suivante. Tout d'abord, si le code nécessaire à un accès distant sur ce type d'objet n'existe pas encore, il est généré dynamiquement. Si `node` est la JVM en cours, les éléments nécessaires à la création d'un objet actif sont ajoutés à l'objet en question (une activité propre, une file d'attente, etc.). Si `node` n'est pas la JVM en cours, une copie de l'objet en question est transmise à la JVM `node`, et les éléments mentionnés ci-dessus sont ajoutés dans la JVM de cette copie. Dans tous les cas, l'objet original passif reste en place. Cette technique permet entre autre la répartition de code pour lequel on ne dispose pas du source.

2.3. *Asynchronisme, polymorphisme et futurs*

Les trois primitives de création des objets actifs retournent un objet qui est *polymorphiquement compatible* avec le type d'origine. Cela permet d'affecter l'objet distant retourné dans une entité du type de création. Ainsi, il sera possible de réutiliser dans un cadre réparti du code prévu initialement pour fonctionner avec des objets locaux et non-actifs. Le programme de l'exemple 2 met en œuvre ce type de réutilisation.

Tout d'abord, selon le principe bien connu de la programmation répartie à objets, les invocations de méthodes jouent également le rôle d'IPC[3], et la cible d'un appel distant est elle-même un objet. La classe standard `A` présente deux méthodes publiques `foo` et `bar`. Celles-ci vont pouvoir être utilisées à distance, de plus les appels

3. Inter-Process Communication

```
class A {
  public foo (...) ...;
  public B bar (...) ...;
  ...
}
class Client {
  static void useA (A a) {
    a.foo (...);       // (1)
    B v = a.bar (...); // (2)
    ...
    v.f1(...);         // (3)
}
main {
  A a1 = new A();
  Client.useA (a1);  // dans useA: appels standards
  A a2 = (A) ProActive.newActive("A",null,''//lo.inria.fr/VM1'');
  Client.useA (a2);  // Polymorphisme:
}                    // dans useA: appels distants et asynchrones
```

Figure 2. *Asynchronisme, polymorphisme et futurs*

sont systématiquement asynchrones — le client n'est pas bloqué, et une attente automatique (*wait-by-necessity*) se produit lorsque le résultat d'un appel asynchrone non encore retourné est utilisé. Sur notre exemple, le second appel "`Client.useA (a2)`" correspond à du polymorphisme entre objet local et objet distant, les points (1) et (2) donnent alors lieu à des appels asynchrones, et le point (3) à une attente par nécessité[4].

Dans le cadre de programmes gérant explicitement la répartition et les synchronisations inter-objets, il est possible d'agir directement sur un futur par deux primitives :

```
ProActive. waitFor (Object obj);   // Attente explicite d'un futur
if ProActive. isAwaited (Object obj) ... // Test l'état d'un futur
```

La seconde primitive permet, suite à un ensemble d'appels asynchrones, de traiter le premier résultat revenu, ou encore de gérer une activité locale en attente d'un résultat.

Comme pour Java RMI et la plupart des systèmes à objets répartis, les paramètres des appels distants sont transmis par *copie profonde*, sauf bien sûr les objets distants eux mêmes qui sont transmis par référence (ce qui confère au système une topologie dynamique). D'autre part, et cette fois-ci contrairement à RMI, on obtient une struc-

4. La philosophie d'implémentation (pas de modification du code source) impose quelques restrictions qui se veulent simples à comprendre et à vérifier : classes actives non `final`, existence d'un constructeur sans arguments, pour obtenir un appel asynchrone le résultat ne doit être ni de type primitif, ni `final`, ni pouvoir retourner une exception déclarée.

```
class BoundedBuffer extends FixedBuffer implements Active {
  live (Body myBody) {
    while (true) {
      if      (this.isFull())   myBody.serveOldest ("get");
      else if (this.isEmpty())  myBody.serveOldest ("put");
      else myBody.serveOldest();
      myBody.waitForNewRequest ();
} } }
```

Figure 3. *Programmation explicite du contrôle*

turation forte en *sous-systèmes* (un objet actif et tous ses objets passifs) sans aucun objet partagé (objet standard accessible par plus d'un objet actif). Cette propriété résulte d'un passage par copie profonde qui est effectif dès l'appel du constructeur, ce qui n'est pas le cas pour Java RMI.

2.4. *Activités, contrôle explicite et abstractions*

Les appels distants étant asynchrones, ils sont mémorisés sous la forme de *requêtes* dans une file d'attente du côté de l'appelé. Ultérieurement, nous dirons que la requête est *servie* par l'objet actif. Jusqu'à présent, nous avons créé un objet actif sans préciser son activité, et par défaut un service FIFO des requêtes était alors réalisé : exécution des requêtes dans l'ordre d'arrivée. Dans ce sens, les objets actifs sont bien des processus purement séquentiels.

La création d'objets actifs de type **a) Class-based** permet de donner à un objet un comportement spécifique[5] : activité propre (qui ne consiste pas à servir des méthodes publiques), activité mixte (services et activité propre), services purs mais de type non-FIFO.

L'exemple 3 présente un simple tampon borné. La routine `live` permet à l'utilisateur de spécifier, sous la forme d'une thread explicite, l'activité de l'objet. Lorsque cette routine existe, elle remplace totalement le comportement FIFO fourni par défaut. Le paramètre `myBody` donne accès à tout un ensemble de routines de services permettant de sélectionner la requête à servir. Dans ce cas, nous utilisons `serveOldest`, mais toute une bibliothèque est accessible au programmeur (Exemple 4). On y trouve des services *bloquants*, *non-bloquants*, *temporisés*, etc. mais également des itérateurs sur la file d'attente permettant d'étendre la bibliothèque si nécessaire. Notons `waitForNewRequest` qui permet de programmer une attente non-active. Nous sommes ici en présence d'une programmation de l'activité qui est *explicite*, avec service lui aussi explicite, sans non-déterminisme sur l'ordre des services ; dans l'exemple 3 si le tampon n'est ni plein ni vide, on impose de servir la requête la plus ancienne. Ce type

5. Un objet actif se situe toujours dans un continuum : *Acteur pur – Agent – Serveur pur.*

```
void serveOldest (); // Sert la plus ancienne requête, bloque
void serveOldest (String s); // la plus ancienne sur s, bloque
void serveOldest (String s1, String s2);//plus ancienne sur s1, s2
...
void serveOldestWithoutBlocking (); // service non bloquant
...
void serveMostRecentFlush (String s); // sert plus ancienne,
...                                   // et supprime les autres
void serveOldestTimed (int t); // bloquant pour au plus t ms
...
void waitForNewRequest (); // Attente non active d'une requête
...
```

Figure 4. *Routines de service*

```
class BoundedBuffer extends FixedBuffer implements ImplicitActive {
  live (ImplicitBody myBody) {
    myBody.forbid ("put", "isFull");
    myBody.forbid ("get", "isEmpty");
} } }
```

Figure 5. *Programmation implicite et déclarative du contrôle*

de programmation est fort utile si l'on souhaite avoir un contrôle fin sur l'activité de l'objet, mais dans certains cas on souhaite programmer à un niveau d'abstraction plus élevé, d'une façon plus déclarative, voire non-derministe. Ce type de programmation *implicite* peut être réalisé en construisant ce que nous appelons des *abstractions de contrôle*. L'exemple 5 montre l'utilisation d'une telle abstraction, programmable par l'utilisateur sous la forme d'une autre interface marqueur (*ImplicitActive*) qui permet simplement d'associer une routine de blocage à une routine publique. Nous sommes ici en présence d'une programmation du contrôle que l'on peut qualifier d'implicite, déclarative, à service lui aussi implicite, et non-déterministe (si le tampon n'est ni vide, ni plein, on ne précise pas quelle méthode servir).

Ce type de construction incrémentale d'abstraction pour le contrôle dénote l'aspect *ouvert* ("Open Implementation" [KIC 96]) de la bibliothèque ProActive. Cet aspect va se révéler particulièrement important pour la construction d'un service de migration puissant et flexible (cf. section 3.2).

2.5. *Propriétés*

Pour conclure cette présentation rapide, attardons nous sur quelques propriétés qui vont jouer un rôle important pour la migration.

Tout d'abord, bien que la sémantique de la communication soit asynchrone, nous suivons un protocole qui impose un rendez-vous entre l'appelant et l'appelé pour transmettre la requête. Cette stratégie apporte plusieurs propriétés importantes :

– après une instruction de communication, l'appelant est certain que la requête est bien dans le contexte de l'appelé (assure une sémantique "*at most one*"),

– les requêtes ne peuvent pas se doubler ; l'ordre d'émission est identique à l'ordre de réception

– propriétés identiques pour les retours de résultat qui suivent le même protocole.

D'autre part, l'attente par nécessité d'une façon intrinsèque apporte une synchronisation fortement dirigée par les données. Par analogie à un modèle "*data-flow*" dans un cadre fonctionnel, nous pourrions parler d'"*object-flow*". Un objet actif se bloque principalement en attente du retour d'un appel asynchrone qui se trouve être représenté par un objet. De plus, dynamiquement nous connaissons dans ce cas l'objet actif qui peut le débloquer. Ceci permet par exemple de détecter dynamiquement un grand nombre de blocages.

Si notre infrastructure permet de répartir du code pour lequel on ne dispose pas du source, certains éléments de celui-ci peuvent engendrer quelques difficultés. Tout d'abord les threads Java classiques, sans être foncièrement incompatibles avec les objets actifs leur font perdre leur caractère purement *séquentiel*. Sans être rédhibitoire, transformer un objet multi-threadé en un objet actif nécessitera une attention particulière. L'utilisation d'objets distants *RMI* nécessite également un traitement particulier. Cependant, des objets actifs et des objets *RMI* peuvent très bien collaborer dans la même application ; nous avons largement expérimenté cette situation. Le danger majeur réside dans le caractère synchrone des objets *RMI* qui peuvent, si l'on n'y prend garde, créer facilement des inter-blocages.

2.6. *Implémentation : réflexion et méta objets pour la répartition*

2.6.1. *Éléments réfléchis et protocole*

Une des décisions les plus importantes qui furent prises durant la phase de conception de ProActive fut de faire reposer la partie qui traite de répartition et de synchronisation sur un *protocole à méta-objets* (MOP, Meta-Object Protocol) [KIC 91] (Ce dernier nous a permis d'établir une séparation claire entre, d'un côté, l'interception de certains événements qui nous intéressent et, de l'autre côté, la manière dont ces événements sont traités afin de leur ajouter une sémantique supplémentaire.

Le but d'un protocole à méta-objets est d'assurer le lien entre le *niveau de base* et le *niveau méta* [MAE 88], c'est-à-dire de permettre à des éléments de l'environnement d'exécution qui ne sont pas représentés par des objets dans le cadre d'un langage de programmation donné d'être représentés par des objets au niveau méta[6]. On parle à ce propos de *réifier* [FER 89] des éléments du niveau de base pour pouvoir les manipuler comme des objets au niveau méta. Dans notre cas, nous nous sommes interessés à la réification des appels de constructeurs et des appels de méthode d'instances ; les appels de méthodes statiques ne peuvent être interceptés. Nous avons pour cela developpé à la fois un mécanisme d'interception de ces événements et un ensemble de classes Java permettant de représenter ces événements par des objets.

Une des caractéristiques principales de notre protocole à méta-objets est de permettre au programmeur de travailler *objet par objet*. On peut ainsi choisir de réifier les appels de méthode ou de constructeur vers une certaine instance d'une classe – on parle alors, par abus de langage, d'un *objet réifié*, tandis que pour d'autres instances de la même classe rien ne sera changé par rapport au cas normal. Lorsqu'un objet est réifié, chaque appel de méthode qui lui est envoyé est tout d'abord converti en un objet, qui est ensuite passé à une instance d'une classe qui implémente la sémantique supplémentaire que l'on désire ajouter à l'appel de méthode vers cet objet. Par exemple, on peut décider que, pour chaque appel de méthode intercepté, l'on affiche le nom de la méthode appelée à des fins de mise au point du programme, on pourrait également utiliser cette technique pour implémenter des objets actifs persistants, ou encore ajouter de la sécurité.

Notre bibliothèque permet tout aussi bien de réifier des objets au moment de leur création que de réifier des objet déjà existants. Pour créer un objet réifié de la classe `A` qui ait pour méta-objet une instance de la classe `MetaA`, nous utilisons le fragment de code suivant dans lequel nous avons volontairement omis le code relatif au traitement des exceptions pour des raisons de concision :

```
Object[] params = ...;
Object[] paramsMeta = ...;
A a = (A) MOP.newInstance ("A", params, "MetaA", paramsMeta);
```

Le tableau d'objets `params` contient les paramètres de l'appel au constructeur de l'objet réifié, tandis que `paramsMeta` contient les paramètres du constructeur du méta-objet. Comme il n'est pas toujours possible d'avoir accès au code de création d'un objet, par exemple dans le cas d'objets instanciés par des bibliothèques distribuées uniquement sous forme de bytecode, notre protocole à méta-objets propose également une primitive qui permet de réifier des objets déjà existants :

```
Object[] paramsMeta = ...;
A a1 = new A (... parameters...);
A a2 = (A) MOP.turnReified (a, "MetaA", paramsMeta);
```

Bien évidemment, dans ce cas precis, l'appel au constructeur de `A` n'est pas réifié, seulement les appels de méthode vers l'objet réifié le seront, à condition qu'ils soient

6. En toute rigueur, nous nous plaçons ici dans le cadre des *runtime MOPs* [CAR 01] : un sous-ensemble des protocoles à méta-objets qui ne s'intéressent qu'aux entités existant à l'exécution.

```
public class Debug implements MetaObject {
  public Object reify (MethodCall c) {
    System.out.println (c.getMethodName());
    return c.execute ();
} }
```

Figure 6. *MetaObjet*

effectués par l'intermédiaire de la référence réifiée (`a2` dans l'exemple ci-dessus) et non pas par la référence directe à l'objet réifié (`a1` dans l'exemple). Dans les deux cas, l'objet résultat renvoyé par l'appel qui crée l'objet réifié (soit `newInstance`, soit `turnReified`) est compatible au niveau du type avec l'objet réifié. C'est cette compatibilité qui est obtenue par une génération de code. Techniquement, le MOP génère dynamiquement une classe Java qui dérive de la classe de l'utilisateur et permet d'intercepter les appels afin de les réifier. Grâce à cette propriété de polymorphisme entre une référence vers un objet et une référence vers le même objet réifié, il est possible de réutiliser du code écrit pour des objets normaux avec des objets réifiés, par exemple des objets distants.

L'écriture d'une classe qui implémente un comportement de niveau méta se fait de manière simple en implémentant l'interface `MetaObject`. Par exemple, si le comportement désiré est d'afficher le nom de chaque méthode appelée, on peut utiliser le méta objet de l'exemple 6. Il est ainsi possible d'écrire des classes de méta-niveau implémentant toutes sortes de sémantiques, la seule contrainte étant que cette sémantique puisse s'exprimer en termes d'appels de méthode et s'implémenter en interceptant les appels au niveau de l'appelant. Dans le cadre de ProActive, nous utilisons principalement le protocole à méta-objets pour quatre choses. Tout d'abord pour conférer de manière transparente un caractère distant aux objets. Ensuite, pour transformer des appels synchrones en appels asynchrones avec futurs et pour pouvoir exprimer facilement des politiques de synchronisation en manipulant les appels de méthode comme des objets. Enfin, pour permettre l'implémentation d'un mécanisme de migration faible transparent. Mais d'autres utilisations sont envisageables (persistance, sécurité, etc.). Le coût de la réification d'un appel dans notre implémentation est environ 0.02 ms sur un PIII 800Mhz. Dans le cas d'un appel local entre deux objets actifs (sans communication réseau), ce temps est de 0.15 ms. Ainsi, dans le cadre d'une application distribuée sur Internet cela reste négligeable par rapport à la latence réseau (actuellement 200ms) [7], et tout à fait acceptable sur un réseau local.

2.6.2. *Méta-objets pour la répartition*

Un objet actif n'est pas une entité monolithique. Il est au contraire composé de plusieurs méta-objets gérant chacun un des *aspects non-fonctionnels* liés à la répartition.

7. Étant donné la vitesse de la lumière, ce temps ne peut pas étre inférieur à 20 ms pour une liaison Boston–Nice par exemple.

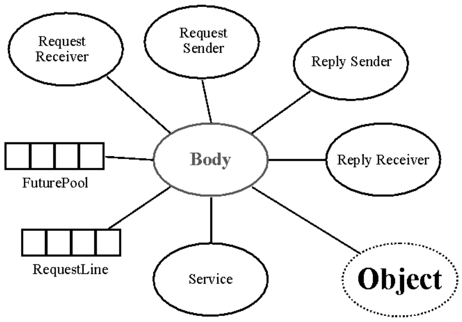

Figure 7. *Classes constitutives d'un objet actif asynchrone*

Nous avons identifié sept fonctionnalités principales (figure 7) : l'envoi et la réception des requêtes, l'envoi et la réception des réponses, les requêtes et futurs en attente, le service (l'activité de l'objet actif). Ces méta-objets sont différents de ceux proposés par CodA [MCA 95] car nous ne nous attachons pas à réifier les mêmes choses ; par exemple CodA propose de réifier des éléments comme la liaison dynamique («*method lookup*»), ou la gestion de l'état de l'objet.

Grâce à cette architecture, il est possible de construire des objets actifs modulables où chaque fonctionnalité peut être ajustée indépendamment des autres. Enfin, en l'absence d'objets partagés, ces sept méta-objets, l'objet actif lui-même et tous ses objets passifs vont constituer l'*unité de migration* de ces objets communicants.

3. Migration

3.1. *Modèle de base et primitive*

La conception de la migration dans ProActive a été guidée par les fonctionnalités souhaitées, mais également par l'environnement d'exécution choisi (Java et la JVM).

3.1.1. *Migration Faible*

Il est actuellement quasiment[8] impossible d'implémenter de façon complètement portable de la *migration forte* en Java de par l'absence de trois fonctionnalités : interruption préemptive, lecture et sérialisation du thread, écriture (reconstruction) du thread sur la JVM cible. D'autre part, nous avons ajouté trois contraintes supplémentaires : (1) par souci de portabilité nous nous sommes interdit toute modification de

8. Nous nous entourons ici de précautions car différentes techniques sont en cours de développement (voir par exemple [PEI 97, BOU 00, FUN 98, WAL 00]), et surtout des changements dans le standard de la JVM pourraient dans l'avenir le permettre.

la machine virtuelle, (2) comme nous voulons pouvoir travailler à l'exécution nous ne voulions pas utiliser des outils tels qu'un préprocesseur de code source, (3) finalement, en raison des problèmes de sécurité nous nous sommes également interdit de modifier le bytecode des classes au chargement par classloader spécialisé.

Nous avons donc implémenté la migration en nous basant uniquement sur les techniques fournies par notre protocole à méta-objets, à savoir ajout dynamique de code à l'exécution et interception des appels par polymorphisme. Nous implémentons donc une *migration faible* : un objet actif devra décider de lui-même de migrer — de façon *pro-active*. Il ne sera pas possible de l'extérieur d'un objet de déclencher une migration préemptive sans un acte volontaire de l'objet actif. Nous allons maintenant détailler le fonctionnement des primitives de migration.

3.1.2. *Primitive*

Il existe en fait une seule primitive implémentée dans la bibliothèque ProActive pour fournir la migration des objets actifs communicants : `migrateTo()` . Cette primitive est surchargée en deux versions décrites dans la table 1.

	Permet la migration vers
`static void migrateTo(URL)`	un site référencé par une URL.
`static void migrateTo(Objet)`	le site supposé d'un autre objet actif.

Tableau 1. *Primitives de migration (méthodes statiques)*

Un objet voulant migrer appelle lui-même la méthode statique `migrateTo()` qui se charge de toutes les opérations nécessaires à la migration, notamment la sérialisation de tout le sous-système et sa reconstruction à l'arrivée sur le site distant. Il est possible soit de migrer vers un *nœud* distant que l'on désigne explicitement, soit de migrer pour rejoindre un autre agent, sans que l'on sache explicitement sur quel nœud il se trouve. Dans ce dernier cas, il n'est toutefois pas possible de garantir que, une fois la migration effectuée, les deux objets mobiles se trouveront sur le même nœud : il est possible que l'agent que l'on cherche à rejoindre migre lui aussi de site en site.

Pour utiliser cette primitive, un objet actif peut implémenter une méthode publique qui appellera la méthode statique *ProActive.migrateTo()*. Ainsi, la demande de migration pourra venir d'un objet tiers. Cependant, la décision de migrer viendra *in fine* de l'objet mobile lui-même : lors du service d'une requête portant sur une telle méthode publique. A noter que du fait de la migration faible, tout code situé après *ProActive.migrateTo()* ne sera pas exécuté. L'exemple 8 illustre l'utilisation des méthodes de migration dans un objet actif.

```
public class SimpleAgent implements Active, Serializable {
  public void moveToHost(String t) {
      ProActive.migrateTo(t);
  }
  public void joinFriend(Object friend) {
      ProActive.migrateTo(friend);
  }
  public ReturnType foo(CallType p) {
      ...
} }
```

Figure 8. *SimpleAgent : exemple d'un objet actif mobile*

3.2. *Abstractions pour la mobilité*

Le modèle et les primitives introduits précédemment permettent de construire des objets actifs mobiles en gérant explicitement les migrations. Mais il est également possible de construire, au dessus des primitives de base, des méthodes et des concepts de plus haut niveau afin de gérer la migration à différents niveaux d'abstraction. Cela permettra entre autres de faciliter la spécification de comportements autonomes.

Dans la suite de cette section, nous allons présenter trois types d'abstractions réutilisables : l'exécution automatique de méthodes lors du départ ou de l'arrivée sur un nouveau site, le suivi d'un *itinéraire* pré-établi, et un agent mobile générique.

3.2.1. *Exécution automatique de méthodes sur départ et arrivée*

La migration d'un objet actif mobile comporte deux phases principales correspondant au déroulement nominal de la migration, ce sont le *départ* d'un site et *l'arrivée* sur le site de destination. Une troisième phase existe en présence d'une levée d'*exception*. ProActive permet de construire une abstraction qui va associer à chacune de ces phases une méthode à exécuter. Cette association se fait en appelant une des méthodes de la table 2 avec comme paramètre le nom de la méthode à exécuter.

	Méthode à exécuter
`static void onDeparture(String s)`	au départ
`static void onArrival(String s)`	à l'arrivée
`static void onException(String s)`	lors d'une exception

Tableau 2. *API d'exécution automatique sur départ et arrivée*

Des contraintes ont été imposées sur les méthodes susceptibles d'être exécutées automatiquement. Elles ne doivent prendre aucun paramètre et ne doivent retourner aucun résultat. Cependant il est possible d'avoir des fonctions complexes en utilisant

`static void add(URL, String method)`	ajoute un site à visiter
`static void travel()`	démarre un itinéraire
`static void requestFirst(Boolean)`	service des requêtes avant migration
`static void itineraryStop()`	arrête un itinéraire
`static void itineraryResume()`	reprend l'exécution d'un itinéraire
`static void itineraryRestart()`	recommence l'itinéraire

Tableau 3. *API d'utilisation d'un itinéraire*

les attributs de l'objet. Une utilisation typique de ces méthodes est la suppression et la reconstruction automatique d'éléments non sérialisables qui ne peuvent donc pas être déplacés avec l'objet, tels qu'une interface graphique (exemple 9).

3.2.2. *Itinéraire*

L'autonomie d'un agent mobile provient notamment de sa capacité à aller de site en site sans intervention extérieure. Dans le cadre de notre bibliothèque, un itinéraire est formé d'un ensemble de paires *destination-méthode* représentant les sites à visiter et les actions à effectuer à l'arrivée sur chacun d'entre eux. Les destinations sont définies soit par reférence directe, soit par leur nom symbolique, auquel cas les mécanismes de nommage de RMI sont utilisés[9]. Une méthode est désignée par son nom, et est exécutée par réflexion.

Cette abstraction associe à chaque objet actif un itinéraire courant qui est suivi de manière séquentielle, et contrôlé par les méthodes de la table 3. Un objet mobile démarre un itinéraire en appelant la méthode `travel()`. Il est possible d'appeler la méthode `requestFirst()` pour spécifier le moment où doit intervenir le traitement des requêtes. L'agent peut ainsi suivre un itinéraire qui sera prioritaire sur le traitement des requêtes (elles seront ignorées jusqu'à la dernière destination) ou bien traiter toutes celles en attente avant de poursuivre son parcours. Il est possible de contrôler directement et dynamiquement un itinéraire afin de construire un comportement adapté à une application particulière (voir en particulier les méthodes `Stop` et `Resume` de la table 3).

3.2.3. *Agent mobile générique*

Beaucoup d'applications à base d'agents mobiles présentent les mêmes caractéristiques : un ou plusieurs agents se déplacent de site en site, effectuent sur chaque site des opérations et partagent leurs résultats. Nous allons montrer comment il est possible d'implémenter la migration et le partage d'information en utilisant ProActive.

L'activité de l'objet, présenté dans l'exemple 10, se décompose en plusieurs opérations. L'agent sert d'abord toutes les requêtes en attente, puis il effectue une opé-

9. Ce qui revient à associer à chaque site une URL du type rmi ://nomdusite/nomdelobjet

```
public void deleteSwingInterface() { ... }
public void rebuildSwingInterface()  { ... }
public void migrateTo(String dest) {
    ...
    ProActive.migrateTo(dest);
}
public void live(Body b) {
  ...
  b.itinerary.add("//tuba.inria.fr/Node1","rebuildSwingInterface");
  b.itinerary.add("//oasis.inria.fr/Node2","rebuildSwingInterface");
  ProActive.onDeparture("deleteSwingInterface");
  ProActive.requestFirst(true);
  ...
  ProActive.travel();
}
```

Figure 9. *Exécution automatique de méthodes et itinéraires*

ration (par exemple une recherche d'information). L'information est ensuite partagée avec les autres objets mobiles. Le code nécessaire pour cette tâche est extrêmement simple puisqu'il consiste à appeler une méthode sur chacun des autres objets, le moteur d'exécution ProActive se chargeant de transformer l'appel local en appel distant quelque soit le site sur lequel se trouve le destinataire.

Il suffit de redéfinir la méthode *performOperation()* afin de programmer les fonctions précises de l'agent. Il est aussi possible d'avoir une politique de sélection de site (méthode *getNextDestination()*) totalement aléatoire ou bien dépendante des informations déjà recueillies.

La migration étant indépendante de la capacité à communiquer, une propriété fondamentale consiste à gérer de manière transparente l'envoi et la réception des requêtes et des futurs, quel que soit l'état de l'objet mobile. Cette fonctionnalité importante, traditionnellement appelé la *localisation* des agents mobiles, sera largement évoquée dans une section spécifique (voir 4.1). Mais il existe deux autres propriétés plus spécifiques à ProActive que nous allons maintenant détailler.

3.2.4. *Migration et rendez-vous*

Sans migration, ProActive garantit un certain nombre de propriétés, en particulier par l'utilisation d'une phase de rendez-vous pour la transmission des messages (cf. section 2.5). Maintenir un tel rendez-vous en présence de migration est un choix relativement coûteux car cela implique de propager ce rendez-vous sur plusieurs machines virtuelles, quellle que soit la technique de localisation utilisée. Néanmoins, nous avons opté pour cette solution afin de privilégier la correction par rapport aux performances. Ainsi, même en présence d'une migration de la source ou de la cible d'une commu-

```
public class GenericAgent implements Active, Serializable {
...
 public void shareInformation() {
    int max = agentList.size();
    for (int j=0;j<max; j++) {
     ((GenericAgent) agentList.elementAt(j)).informationFound(...);
 }  }
 public void live(Body myBody) {
    while (...) {
      this.serveAllPendingRequests();
      this.performOperation();
      this.shareInformation();
      ProActive.migrateTo(getNextDestination());
} } }
```

Figure 10. *Modèle générique d'activité d'un agent mobile*

nication (voire des deux simultanément), la bibliothèque garantit une sémantique "*at most one*", et des messages qui ne peuvent se doubler point à point.

3.2.5. *Communications locales et migration*

Il existe une optimisation des systèmes répartis fort utile : deux entités se trouvant dans le même espace d'adressage doivent pouvoir communiquer directement sans passer par la pile des protocoles réseaux, tout en conservant une sémantique identique (en particulier en ce qui concerne les copies profondes d'objets). Dans un système avec migration, la réalisation de cette caractéristique est bien plus délicate car la propriété "*dans le même espace d'adressage*" n'est plus statiquement décidable (à l'allocation des objets actifs), mais varie dynamiquement en fonction des migrations.

Nous avons implémenté cette optimisation dans ProActive, tout en prenant soin de maintenir la sémantique de passage des paramètres. Ainsi, si un objet mobile arrive sur un site, il communique directement avec les autres objets présents sur ce site sans passer par le réseau. De manière symétrique, un objet présent sur un site communique directement avec les objets récemment arrivés. Dans tous les cas, l'absence d'objets passifs partagés est maintenue.

4. Etude de performance et localisation

Les mesures de performances que nous avons effectuées ont fait apparaître un temps de migration (hors mise à jour nécessaire à la localisation) d'environ 75 ms sur un réseau local (Pentium III à 800 Mhz, Ethernet 100 Mb). Ce chiffre donne une indication de l'ordre de grandeur du coût de la migration dans notre système, notamment par rapport au coût d'une communication ProActive dans la même configuration (inter-machine sur LAN) : environ 5 ms. Le lecteur pourra se reporter à [ISM 99] pour une étude exhaustive du coût de la réalisation d'une tâche donnée par migration vs. par communication. Toujours dans cette configuration, nous avons mesuré la migration d'un agent du logiciel commercial Voyager [VOY 99] : environ 50 ms. Notre objectif n'étant pas la performance pure, mais avant tout la transparence, la flexibilité et la correction, ces chiffres nous semblent démontrer que le surcoût introduit reste largement acceptable. Mais dans le cadre d'un système alliant étroitement mobilité et communication, il nous semble important d'analyser un autre aspect peu abordé dans la littérature : le *coût de la communication en présence de migration*. C'est ce que nous allons étudier dans le reste de cette section.

4.1. *Techniques de localisation*

Afin de pouvoir assurer la communication en présence de migration, il est nécessaire de pouvoir *localiser* après un nombre quelconque de migrations un objet actif.

La localisation d'objets mobiles n'est pas un problème nouveau. Il a été étudié par exemple dans le cadre des systèmes d'exploitation distribués [NUT 94, TAC 96]. Il y a cependant une différence lorsqu'on parle de localisation d'objets mobiles ou d'agents mobiles. Ceux-ci doivent pouvoir s'exécuter dans des environnement hétérogènes, aussi bien au niveau logiciel que physique. Dans de telles conditions, il est difficile voire impossible d'avoir un service sous-jacent au niveau du système d'exploitation ou encore une coopération entre les différents sites à même de recevoir un objet mobile. Nous nous plaçons donc dans un cadre très général où la localisation est assurée explicitement par la couche applicative.

4.1.1. *Répéteur réparti*

Une première approche pour la localisation de l'objet actif mobile auquel on doit délivrer un message est basée sur la remarque simple suivante : une source n'a pas besoin de connaître la localisation exacte d'un agent, il lui suffit d'avoir l'assurance que tout message qu'elle envoie à cet agent est correctement acheminé à destination. Ce premier mécanisme est basé sur l'utilisation de *répéteurs*. Dans la littérature, cette notion est connue indifféremment sous les noms de *répartiteur* ou *transmetteur* (en anglais *forwarder*) [MAI 92]. Le principe en est simple : l'agent qui quitte un site laisse derrière lui un objet répéteur qui sera chargé de faire suivre les messages vers son nouveau site d'exécution présumé. Ce nouveau site peut lui-même abriter un répéteur, et ainsi de suite pour chaque migration. On obtient ainsi une chaîne de répéteurs qui a pour dernier maillon l'objet actif mobile. Le message sera donc transmis de site en site jusqu'à l'agent. Ce mécanisme est utilisé dans [VOY 99].

L'appelant a ainsi l'impression de contacter l'agent directement sans passer par des répéteurs. Si ceux-ci ont la même interface que l'agent, alors la transparence sera assurée. Très peu de modifications sont nécessaires du côté de l'appelant pour pouvoir profiter de ce mécanisme. Afin de maintenir le rendez-vous, un appel à travers des répéteurs présente une phase synchrone, c'est à dire que l'appelant reste bloqué jusqu'à ce que l'objet actif appelé ait reçu le message.

Le principal désavantage des répéteurs vient de leur prolifération. Des sites qui voient passer beaucoup d'agents risquent de voir une part non négligeable de leurs ressources occupées par ceux-ci. Un autre désavantage est la fragilité : la perte d'un seul répéteur brise la chaîne et provoque la perte du lien vers l'objet actif mobile. Il est possible de remédier à ces difficultés en mettant à jour dynamiquement les références vers la nouvelle position d'un agent (figure 11, étape 4) afin de réduire la longueur des chaînes de répéteurs ; les communications sont ainsi plus fiables et plus efficaces. Cette optimisation importante, appelée quelquefois «*tensioning* ou *shortcuting*», est implémentée dans notre bibliothèque : dès qu'une communication se produit, la référence (le proxy) est remis à jour si nécessaire.

4.1.2. *Serveur centralisé*

Une solution évidente pour localiser un agent est d'utiliser un *serveur centralisé de localisation*. Celui-ci maintient une base de données des localisations des agents dont il a la charge (figure 12A). La mise à jour de la localisation d'un objet mobile ne doit se faire qu'une fois la migration achevée afin d'assurer la consistance des informations contenues dans le serveur (figure 12B). Du point de vue de l'appelant, un objet mobile peut être dans 3 états : *joignable*, *en cours de migration*, *migré*. Le premier cas est trivial, la source contacte directement l'agent. Le dernier cas se règle grâce au serveur qui permet de se ramener au premier. Un objet en cours de migration ne peut pas accepter de requêtes pendant un delai plus ou moins long. Cependant, il n'a pas encore migré et n'est donc pas joignable sur le site distant. Pour un même appel de méthode, l'agent a pu successivement passer dans ces différents états.

Nous avons implémenté une solution simple consistant à bloquer la source au niveau de l'objet mobile jusqu'à ce que la migration soit finie.

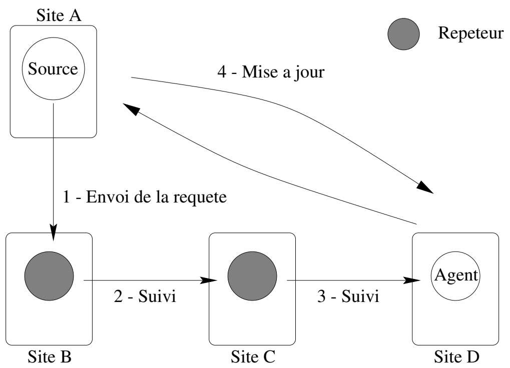

Figure 11. *Localisation avec répéteurs*

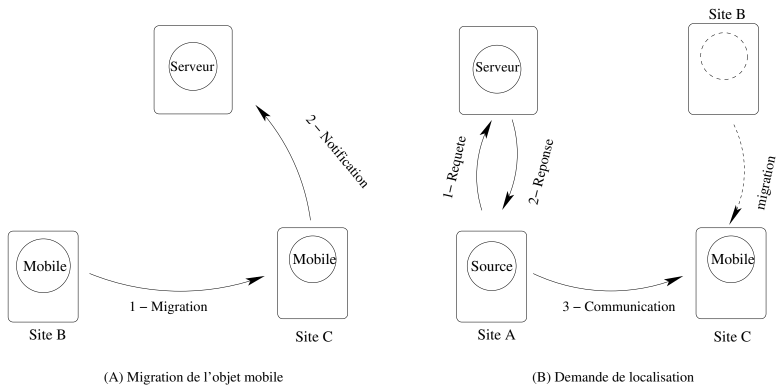

Figure 12. *Opérations de localisation avec serveur centralisé*

4.2. *Etude des performances*

Nous allons dans cette partie étudier les performances d'une application utilisant les deux mécanismes de localisation décrits précédemment. Nous allons considérer une application simple composée de deux objets actifs. Le premier ne bouge pas et joue le rôle de source qui envoie des appels de méthode vers un second objet actif qui lui migre de site en site. La source contacte l'agent avec une fréquence λ, i.e toutes les $1/\lambda$ secondes elle essaie de contacter l'agent. De même, l'agent migre avec une fréquence ν et donc attend en moyenne $1/\nu$ secondes avant de migrer. Nous considérerons que tous les sites joignables par l'agent sont à égale distance les uns des autres (i.e les temps de communications sont tous égaux).

4.2.1. *Expérimentations*

Nos expérimentations ont été effectuées sur un réseau local (LAN) et sur un réseau régional (MAN, Medium Area Network). Les mesures ont été effectuées en fixant alternativement les valeurs λ ou ν, et en faisant varier l'autre. Les valeurs ont été choisies pour refléter différents types de liens entre la source et l'objet mobile. Un faible λ est caractéristique d'une application avec une source relativement silencieuse et un agent relativement indépendant, tandis qu'un λ élevé décrit une application où la source envoie beaucoup d'informations à l'agent. Avec un ν faible, nous considérons les systèmes où l'agent passe beaucoup de temps sur chaque site, alors qu'un ν élevé concerne les applications extrêmement mobiles. Nous donnons dans le tableau 4 des exemples d'applications suivant les couples de paramètres.

Suite à l'ensemble de nos mesures expérimentales, les moyennes des paramètres recueillies pour un LAN et un MAN sont présentées dans la table 5 ; chaque valeur représente la moyenne de 40 heures d'expérimentation.

Caractéristiques	*Exemple d'application*
λ faible, ν faible	Administration Réseau
λ faible, ν élevé	Commerce electronique, voire Data Mining
λ élevé, ν faible	Multimedia Mobile [HAL 97]

Tableau 4. *Exemple d'applications*

	LAN	*MAN*
Temps de communication	15.43 ms	62.19 ms
Temps de service	26.16 ms	84.90 ms
Temps de migration - Serveur	508.92 ms	1238.14 ms
Temps de migration - Forwarder	370.22 ms	1071.04 ms

Tableau 5. *Caractéristiques des expérimentations*

La durée de migration est dépendante du système de localisation choisi car avec le serveur elle inclut le temps nécessaire pour mettre à jour la localisation. Dans le cas du serveur de localisation, nous avons indiqué le temps de service moyen pour traiter un message de l'agent ou de la source. Ce temps est plus élevé dans le cas du MAN car nous mesurons aussi le temps de communication d'une réponse éventuelle à une demande de localisation. Dans le cas des répéteurs, le temps de migration n'inclut pas le "tensioning" car celui-ci n'est effectué que lorsqu'une communication a lieu.

4.2.2. *Résultats et analyse*

Les résultats obtenus sont rassemblés dans la figure 13. La courbe de temps dans le cas du serveur centralisé est relativement linéaire (figures 13 A, E, G). Dans pratiquement toutes nos expérimentations, le mécanisme à base de serveur centralisé s'est avéré plus efficace que les répéteurs, excepté dans le cas du réseau régional avec $\nu = 1$ (figure 13 B). Dans ce cas, les répéteurs bénéficient du fait que la source essaie de contacter directement l'objet mobile et que celui-ci se déplace faiblement. Pour un ν fixé (figures A,B,C,D) le temps de communication décroît légèrement quand λ augmente. Plus la source essaie de contacter l'agent, plus la chaîne des répéteurs est courte car elle est mise à jour après chaque appel de méthode. Cela diminue donc le temps moyen pour joindre l'agent. Dans le cas du serveur, un λ élevé diminue la probabilité de contacter l'agent alors qu'il est en train de migrer, éliminant ainsi de longues attentes. Cependant, c'est la localisation par répéteurs qui bénéficie le plus d'une source très active, dépassant le serveur dans une configuration (réseau régional, figure B), et approchant son temps même pour un objet très mobile (réseau local, figure C).

Si λ est fixé, la situation est beaucoup plus claire. La localisation par répéteur a des résultats catastrophiques. Le serveur montre sa faible sensibilité aux conditions expérimentales, la variance de ses résultats étant très faible comparée à celle des répéteurs.

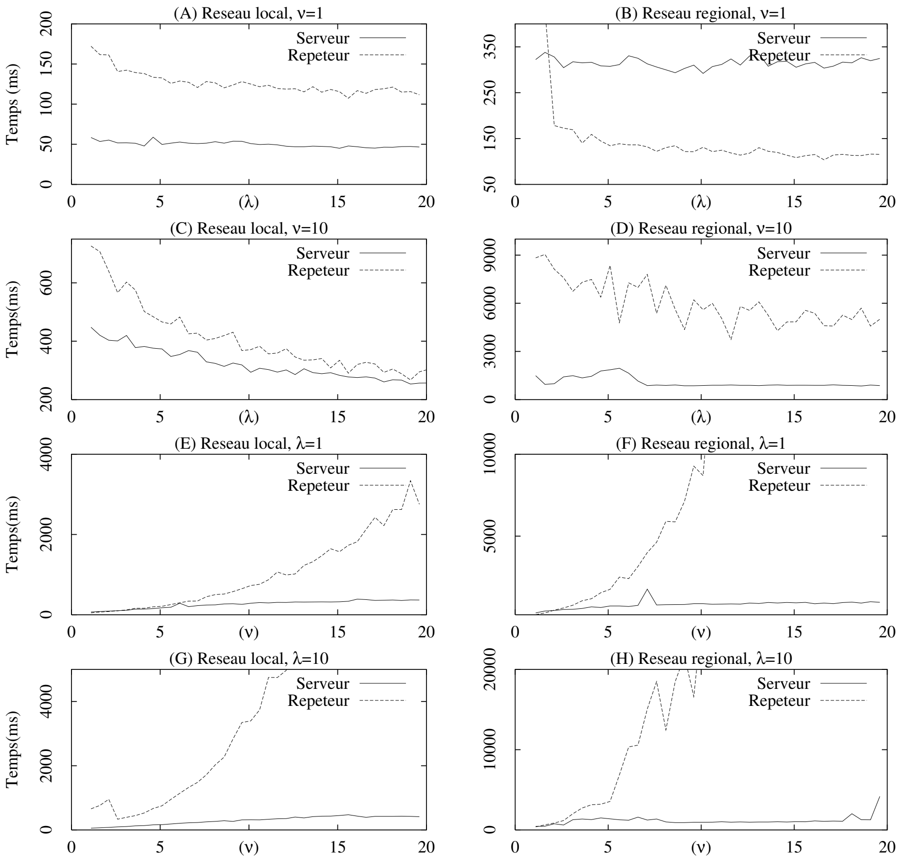

Figure 13. *Comparaison des temps de communication en présence de migrations*

Fréquences		LAN	MAN
Migrations peu fréquentes,	ν=1	Serveur	Répéteurs
Migrations fréquentes,	ν=10	Serveur ou Répéteurs	Serveur
Communications peu fréquentes,	λ=1	Serveur	Serveur
Communications fréquentes,	λ=10	Serveur	Serveur

Tableau 6. *Synthèse : localisation optimale en fonction du réseau et de l'application*

Au vu des résultats des expérimentations, nous pouvons dégager le mécanisme de localisation optimal en fonction du réseau et de l'application (table 6). Le mécanisme des répéteurs, bien que souvent utilisé [VEN 97, VOY 99], ne semble pas le plus efficace : c'est le mécanisme de localisation le mieux adapté uniquement avec des migrations peu fréquentes et sur un réseau régional. Si, comme dans nos expérimentations le serveur n'est jamais surchargé (car nous n'avons considéré qu'une seule source et qu'un seul objet mobile), alors un serveur centralisé est plus efficace dans tous les autres cas. Nous pouvons nous attendre à des temps plus élevés dans le cas plus général de n sources et m agents.

5. Travaux connexes

Nous n'étudierons dans cette partie que des bibliothèques permettant de programmer des objets mobiles en Java, ne modifiant pas la JVM. Les plus célèbres sont Aglets [VEN 97] et Voyager [VOY 99]. Le lecteur est invité à se reporter, par exemple, à [KIN 97] et [FUG 98] pour des présentations générales des plateformes à objets et agents mobiles.

5.1. *Migration forte ou faible*

Du fait de l'utilisation de Java et de l'impossibilité d'accéder en lecture/écriture à la pile d'exécution dans le but d'avoir une photo exacte du contexte d'exécution, aucun des systèmes à objets mobiles reposant uniquement sur des bibliothèques ne met en œuvre de la migration dite « forte ». Au contraire, il s'agit de migration dite « faible ». Cependant, on va voir que deux types de solution sont envisagés.

(1) On ne réalise la migration qu'en des points de l'exécution où la sauvegarde du contexte de l'objet est simple à réaliser ou implicite. Par exemple, l'exécution de la primitive `MoveTo` de Voyager est bloquante tant qu'il y a des threads actives attachées à l'objet. Dans ProActive, un unique point de ce type existe et se situe entre le service de deux requêtes, puisque l'ordre de migration se présente lui-même comme une requête.

(2) On autorise la migration à condition que l'objet soit passé à un point d'enregistrement de son contexte. Lorsque l'objet a migré, son exécution reprend depuis le point d'enregistrement le plus récent. Cela peut avoir pour effet d'engendrer un *retour en arrière* dans son exécution et donc d'éventuelles incohérences entre l'état à partir duquel l'objet reprend son exécution et l'état dans lequel il a éventuellement «laissé» d'autres objets. Selon les systèmes, de tels points d'enregistrement sont implicitement ou explicitement placés par le programmeur. Dans Ajents [IZA 99] par exemple, sur demande du programmeur, chaque appel d'une méthode distante commence par l'enregistrement du contexte de l'objet : ainsi, on sait exactement par un mécanisme de réification, quelle est la méthode invoquée et ses paramètres. Si une demande de migration de l'objet appelant est générée pendant l'exécution de la méthode distante, alors cette exécution est suspendue, et le dernier point d'enregistrement servira à reprendre l'exécution juste au moment de l'invocation. On voit que l'on réexécutera donc l'appel distant de méthode. Aucune solution générale, simple et efficace ne semble possible pour résoudre les incohérences induites par cette solution.

5.2. *Communication entre objets mobiles*

5.2.1. *Forme de la communication*

Dans les différents systèmes étudiés, l'interaction entre objets se réalise par le biais d'appel de méthode (comme dans Ajents ou Voyager) ou parfois aussi par échange de message (comme dans Voyager ou Aglets). En l'absence d'un système de réflexion tel que celui utilisé dans ProActive (ou aussi dans Voyager, afin de permettre de l'invocation de méthode Java standard même si l'objet est distant), le programmeur doit récupérer explicitement un proxy vers l'objet afin de pouvoir interagir avec lui.

5.2.2. *Mode synchrone ou asynchrone*

En général, l'interaction est par défaut de type *synchrone* (dans Ajents, invocation par `Ajents.rmi`, `Sync.invoke` ou appel de méthode Java standard dans Voyager, `sendMessage` dans Agents) mais il peut exister une variante *asynchrone* (invocation par `Ajents.armi`, `Future.invoke` dans Voyager, ou `sendFutureMessage` dans Aglets). En fait, aucune des bibliothèques couramment utilisées ne propose une seule et unique primitive d'interaction qui unifierait ces deux modes. En revanche, le principe d'attente par nécessité mis en œuvre dans ProActive permet naturellement de reproduire une communication synchrone à base d'une communication asynchrone.

La différenciation entre les diverses communications vers l'objet récepteur nécessite souvent d'être explicitée par le programmeur : par exemple, dans Aglets, une chaîne de caractères sert à typer le message, et son décodage au sein d'une méthode `handleMessage` permet de décider de son traitement. Tout ceci est évidemment inutile lorsqu'un mécanisme de réflexion est mis en œuvre

5.2.3. *Localisation*

Afin de pouvoir communiquer avec un objet mobile distant, il est nécessaire de pouvoir l'atteindre où qu'il soit. La présence en général d'un proxy permet de masquer au programmeur cette gestion. Selon les systèmes, la localisation de l'objet se fait par des répéteurs (comme dans Voyager ou Aglets) ou par serveur (comme dans Ajents, où plus précisément chaque site maintient la liste des objets qu'il héberge ou hébergeait en indiquant leur nouvelle localisation). [ROT 00] mentionne une autre stratégie qui consiste à lancer une recherche de l'agent à travers le réseau. Bien sûr, cela ne peut s'appliquer que si les frontières du réseau sont fixées. La décision d'implanter une stratégie à base de répéteur ou de serveur dépend, comme nous l'avons étudié, de divers facteurs telles la fréquence de contact d'un agent et la fréquence de migration. Après une étude très brève, [ROT 00] donne quelques arguments en faveur de la stratégie de localisation à base de serveur.

5.2.4. *Description d'itinéraire et d'événements liés à la mobilité*

Spécialement dans les systèmes dédiés aux applications à base d'agents mobiles, par opposition à ceux plutôt dédiés au calcul réparti, un mécanisme d'itinéraire permet de gérer les différents sites à visiter : Agents et Voyager ainsi que ProActive le proposent. À cette notion d'itinéraire est couramment adjointe celle d'événement de mobilité correspondant au départ d'un site ou à l'arrivée sur un site.

6. Conclusion

Nous avons présenté comment un modèle à objets communicants pouvait offrir un service de migration simple et flexible. Ces travaux montrent en particulier qu'une communication asynchrone avec futur est particulièrement adéquate : la migration est transparente aux appels entrants mais également sortants (réception/émission de requêtes et de réponses). Cette transparence autorise la construction d'abstractions de plus haut niveau pour la mobilité.

Cette mobilité est du type *migration faible*. L'équilibrage de charge est peut-être le seul domaine d'application exclu de ce fait ; néanmoins, si la granularité des services est suffisamment petite, l'équilibrage peut s'effectuer entre chacun d'eux. Si dans le futur la plate-forme Java autorise une migration forte *portable*, ce service peut venir naturellement s'intégrer au modèle par une primitive appelable à distance sur un objet actif. Toujours dans le domaine des limitations, notons que pour être transformée en objet actif migrable, la classe de l'objet doit posséder deux propriétés qui se veulent simples à comprendre et à vérifier : avoir un constructeur sans arguments, et ne pas être `final`.

Outre l'aspect flexible et transparent, l'aspect dynamique est également important : il est possible de transformer un objet reçu sans code source en un objet actif mobile, avec télé-chargement dynamique de bytecode si nécessaire. Si le système privilégie une sémantique la plus fiable possible pour les communications ("*at most one*", et FIFO point à point) par une technique de rendez-vous, cela ne devait pas être au prix de performances inacceptables. Celles-ci peuvent se résumer en quelques chiffres de base : 0.02 ms pour un appel réifié, 0.15 ms pour un appel ProActive intra-VM, 5 ms pour le même appel inter-machine sur LAN, 75 ms pour une migration sur LAN (pour le système Voyager qui présente moins de fonctionnalités et une sémantique moins forte la migration coûte 50 ms sur LAN). Mais dans le cadre d'un système alliant étroitement mobilité et communication, il nous a semblé important d'analyser un autre aspect peu abordé dans le passé : le coût de la communication en présence de migration. Nos expérimentations démontrent que ce n'est pas pas forcément le mécanisme de localisation le plus utilisé qui est le plus performant. Également à noter, la variance du temps de communication est très importante dans certains cas : de quelques ms à plusieurs secondes (voir par exemple la table 5, figures G et H). En perspective, il nous semblerait donc intéressant de concevoir un système *adaptatif* qui utiliserait le système de localisation le plus adapté en fonction des paramètres dynamiques du réseau et de l'application.

7. Bibliographie

[BOU 00] BOUCHENAK S., HAGIMONT D., « "Pickling threads state in the Java system" », *Proc. TOOLS Europe, Mont Saint-Michel / Saint-Malo, June 5-8*, 2000.

[BRI 96] BRIOT J.-P., GUERRAOUI R., « Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances », *Technique et Science Informatiques (TSI)*, vol. 15, n° 6, 1996, p. 765–800, Editions Hermès, Paris.

[CAR 93] CAROMEL D., « Toward a Method of Object-Oriented Concurrent Programming », *Communications of the ACM*, vol. 36, n° 9, 1993, p. 90–102.

[CAR 98] CAROMEL D., KLAUSER W., VAYSSIERE J., « Towards Seamless Computing and Metacomputing in Java », *Concurrency Practice and Experience*, vol. 10, 1998.

[CAR 01] CAROMEL D., VAYSSIÈRE J., « Reflections on MOPs, Components, and Java Security », *Proceedings of ECOOP 2001*, vol. 2072 de *LNCS*, Springer-Verlag, June 2001, p. 256–274.

[CZA 00] CZARNECKI K., EISENECKER U., GLÜCK R., VANDEVOORDE D., VELDHUIZEN T., « Generative programming and active libraries (extended abstract) », *Generic Programming. Proceedings*, vol. 1766 de *LNCS*, Springer-Verlag, 2000, p. 25–39.

[FER 89] FERBER J., « Computational Reflection in Class Based Object Oriented Languages », *Proc. of the OOPSLA-89 : Conference on Object-Oriented Programming : Systems, Languages and Applications*, 1989, p. 317-326.

[FUG 98] FUGGETTA A., PICCO G., VIGNA G., « Understanding Code Mobility », *IEEE Trans. of Software Engineering*, vol. 24, n° 5, 1998, p. 342–361.

[FUN 98] FUNFROCKEN S., « Transparent Migration of Java-Based Mobile Agents », *Workshop on Mobile Agents*, 1998, p. 26-37, LNCS, No. 1477, Springer-Verlag.

[HAL 97] HALLS D., « Applying Mobile Code to Distributed Systems », PhD thesis, Computer Laboratory, University of Cambridge, jun 1997.

[ISM 99] ISMAIL L., HAGIMONT D., « A performance Evaluation of the Mobile Agent Paradigm », *Proc. OOPSLA'99*, 1999.

[IZA 99] IZATT M., CHAN P., « Ajents : Towards an Environment for Parallel, Distributed and Mobile Java Applications », *Proc. of the 1999 Java Grande Conference*, ACM, 1999.

[KIC 91] KICZALES G., DES RIVIERES J., *The art of the metaobject protocol*, MIT Press, Cambridge, MA, USA, 1991.

[KIC 96] KICZALES G., « Beyond the Black Box : Open Implementations », *IEEE Software*, vol. 13, n° 1, 1996, p. 8–11.

[KIN 97] KINIRY J., SIMMERMANN D., « A hands-on look at Java Mobile Agents », *IEEE Internet Computing*, vol. 1, n° 4, 1997, p. 49–52.

[MAE 88] MAES P., NARDI D., « Meta-Level Architectures and Reflection », 1988, editors, Collection of papers, North-Holland.

[MAI 92] MAISONNEUVE J., SHAPIRO M., COLLET P., « Implementing References as Chains of Links », *3d Int. Workshop on Object-Orientation in Operating Systems*, 1992.

[MCA 95] MCAFFER J., « Meta Level Programming with CodA », *Proceedings of ECOOP'95, Aarhus, Denmark*, LNCS 952, p. 190–214, Springer-Verlag, Berlin, 1995.

[NUT 94] NUTTALL M., « Survey of Systems Providing Process or Object Migration », rapport n° 94/10, may 1994, Imperial College London, DOC.

[PEI 97] PEINE H., STOLPMANN T., « The Architecture of the Ara Platform for Mobile Agents », *First International Workshop on Mobile Agents (MA'97)*, LNCS, Springer-Verlag, Berlin Germany, 1997, p. 50–61.

[ROT 00] ROTH V., « Scalable and Secure Global Name Services for Mobile Agents », *6th ECOOP Workshop on Mobile Object Systems : Operating System Support, Security and Programming Languages*, vol. 1964 de *LNCS*, Springer Verlag, 2000.

[TAC 96] TACONET C., BERNARD G., « Object location in wide area networks for microkernel based operating systems », *Proceedings of the ECOOP'96 II Workshop on Mobility and Replication*, july 1996.

[VEL 98] VELDHUIZEN T. L., GANNON D., « Active Libraries : Rethinking the roles of compilers and libraries », *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, 1998.

[VEN 97] VENNERS B., « Under the Hood : The architecture of aglets », *JavaWorld : IDG's magazine for the Java community*, vol. 2, n° 4, 1997.

[VOY 99] VOYAGER, « ObjectSpace, Inc. », 1999, www.objectspace.com.

[WAL 00] WALSH T., NIXON P., DOBSON S., « As strong as possible mobility : an architecture for stateful object migration on the Internet », rapport n° TCD-CS-2000-11, 2000, Department of Computer Science, Trinity College Dublin.

Article reçu le 5 février 2001.
Version révisée le 4 octobre 2001.
Rédacteur responsable : JEAN-PAUL ARCANGELI

Françoise Baude *est maître de conférences à l'Université de Nice - Sophia Antipolis depuis 1993. Après l'obtention d'un Doctorat de l'Université de Paris-Sud (Décembre 1991), elle a effectué un séjour post-doctoral à l'Université de Warwick, Grande-Bretagne, puis à Queen's University, Kingston, Canada. Ses activités de recherche concernent les langages et environnements de programmation parallèle et répartie.*

Denis Caromel *est titulaire d'un Doctorat de l'Université de Nancy I (Février 1991), et d'une Habilitation à Diriger des Recherches de l'Université de Nice – Sophia Antipolis (Novembre 1996). Il est professeur dans cette même université depuis 1998, et membre de l'Institut universitaire de France. Il a séjourné deux ans en Californie (1988 à 1989). Il concentre ses recherches sur les modèles formels et les outils pour une programmation répartie simple, sûre et efficace.*

Fabrice Huet *est étudiant en thèse à l'Université de Nice - Sophia Antipolis depuis novembre 1999. Ses thèmes de recherche incluent le code et les agents mobiles, les systèmes répartis et la méta-programmation.*

Julien Vayssière *est ingénieur Supélec et actuellement étudiant en thèse à l'Université de Nice - Sophia Antipolis depuis novembre 1999. Ses thèmes de recherche incluent les protocoles à méta objets, en particulier leur sécurité, ainsi que les systèmes répartis à objets.*