RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

Communication de groupe typé pour objets répartis

Laurent Baduel, Denis Caromel

Equipe OASIS : INRIA - I3S - CNRS
INRIA Sophia-Antipolis, 2004 route des Lucioles,
BP 93 - 06902 Sophia-Antipolis Cedex France
prénom.nom@sophia.inria.fr

Résumé
La communication de groupe est un dispositif crucial pour le calcul haute performance notamment sur les grilles de calculs. Tandis que les bibliothèques issues des travaux antérieurs imposaient des contraintes spécifiques aux programmeurs (l'utilisation d'interfaces consacrées), notre approche se veut plus flexible.
Cet article propose un modèle où, étant donnée une classe Java, les communications de groupes sont lancées en utilisant les méthodes publiques cette classe. En conservant la notation pointée, les communications de groupe demeurent typées. En outre, des *groupes résultats* sont automatiquement construits pour manipuler le résultat d'opérations collectives, fournissant ainsi une manière élégante et efficace de programmer des opérations de rassemblement. Cette flexibilité permet la manipulation de résultats ayant la forme de groupes d'objets, et l'envoi de différents paramètres à différents membres d'un groupe (distribution des données). Des groupes hiérarchiques peuvent être construits et déployés facilement : il s'agit d'un mécanisme important pour l'utilisation de plusieurs grappes dans le calcul sur grilles. Le défi était de fournir une gestion simple, efficace, et dynamique de groupes d'objets distribués sur la grille. Des mesures de performances démontrent la viabilité de l'approche.

Mots-clés : Communication de groupe, Distribution, Java

---

1. Introduction

La programmation d'applications à hautes performances nécessite la définition et la coordination de plusieurs activités parallèles. Une librairie pour la programmation parallèle se doit de fournir, non seulement une communication point à point, mais également des primitives de communication au sein de groupes d'activités. Dans le monde Java, RMI, le mécanisme standard de communication point à point, est approprié aux interactions de type client/serveur. Dans un contexte de calculs à hautes performances, des communications asynchrones et collectives doivent être accessibles au programmeur, ainsi le seul usage de RMI n'est pas suffisant.
Nous avons développé *ProActive*, une librairie 100% Java pour la programmation parallèle, distribuée et concurrente. *ProActive* fournit de façon transparente un service d'invocation de méthodes à distance vers des objets actifs distribués, des communications asynchrones avec *futurs* et des mécanismes de synchronisation de haut-niveau tels que l'*attente par nécessité*.
Cet article présente la conception d'un mécanisme d'invocation de méthode vers un groupe d'objets actifs distribués et quelques principes de son implémentation au sein de la libraire *ProActive*. Les approches existantes pour le calcul parallèle et distribué en Java incluent l'utilisation de structures de programmation dédiées au parallélisme, comme les collections parallèles et distribuées [7, 9], ou l'emploi de librairies de type MPI dans un style de programmation SPMD [12]. Notre approche se veut plus générale car elle permet d'établir des modèles alternatifs de programmation parallèle tout en pouvant fournir la communication de groupes à des applications réparties déjà existantes.
Dans la section 2, nous présenterons les fonctionalités de la librairie *ProActive* sur lesquelles se base la communication de groupe. Dans la section 3, nous poursuivrons en exposant les principes de notre

communication de groupes et en présentant les méthodes de création et d'utilisation de cette API. La section 4 insistera sur des mécanismes d'optimisation et mettra en valeur des mesures d'efficacité. Nous apporterons notre conclusion dans la dernière section en donnant nos perspectives de recherche.

2. Cadre et précédents travaux

2.1. Distribution dans *ProActive*

La librairie *ProActive* repose sur les APIs standards de Java (Java RMI, l'API de réflexion,...). Aucune modification de l'environnement d'exécution n'est requise, ni aucun préprocesseur ou compilateur spécial. Une machine virtuelle Java standard suffit à utiliser la librairie. Le modèle de distribution de *ProActive* est parti d'un effort de simplification et d'un souci de réutilisation de code d'applications dans des systèmes à objets [5, 6], en respectant une sémantique précise [1].

Une application distribuée et/ou concurrente construite avec *ProActive* est composée d'entités de grain moyen appelées *objets actifs*. Chaque objet actif possède une activité propre et la capacité de décider dans quel ordre servir les appels de méthode qu'il reçoit et stocke dans une file d'attente de requêtes. Les appels de méthode envoyés à un objet actif sont rendus asynchrones [1] avec génération d'objets *futurs* transparents qui sont soumis à des mécanismes de synchronisation tels que l'*attente par nécessité* [5]. Au début de chaque appel distant asynchrone, un “rendez-vous” se produit pour s'assurer que la requête de l'appelant se place dans la file d'attente de l'objet actif appelé.

ProActive fournit la capacité de *créer à distance des objets actifs*. Pour cela, il faut être en mesure d'apporter quelques nouveaux services, notament l'identification de la machine virtuelle Java (JVM). *ProActive* définit des objets dont le rôle est de recueillir plusieurs objets actifs dans une entité logique : ce sont les *nœuds*. Les nœuds procurent une abstraction pour la localisation physique d'un ensemble d'objets actifs. Pour appeler et manipuler les nœuds, un nom symbolique leur est associé. Ce nom est l'URL indiquant leur localisation.

Par exemple : `rmi://lo.inria.fr/Node1`.

La création d'un objet actif se fait en spécifiant le nœud sur lequel il sera positionné :

```
A a = (A) ProActive.newActive("A", params, "rmi://lo.inria.fr/Node1");
```

Afin d'aider la phase de déploiement des objets actifs d'une application, le concept de *nœuds virtuels* comme entités pour placer les objets actifs a été présenté dans [4]. Ces nœuds virtuels sont décrits extérieurement par des descripteurs XML qui sont lus à l'exécution et servent à instancier des nœuds pour y placer des objets actifs.

2.2. Travaux précédents

Des travaux antérieurs ont été réalisés sur le sujet des communications de groupe : des plateformes (Isis, Horus [13]), en passant par les systèmes d'exploitation distribués tels que Amoeba [8], System V et jusqu'aux API de bas (MPI) et haut niveau (JavaGroups [3]).

L'étude présentée dans [10] est la plus proche de la notre : les objectifs et l'approche sont similaires. Le but de ce mécanisme de communication de groupes est de généraliser tous les genres d'invocation de méthode entre objets (point à point ou collective, synchrone ou asynchrone, locale ou distante). De nombreux modes de communication sont disponibles et doivent être explicitement choisis par le programmeur, ce qui exige un certain effort de sa part.

La différence principale entre le mécanisme de communication de groupe que nous présentons ici et d'autres systèmes, est qu'il s'agit d'un mécanisme additionnel intégré autour d'un cadre basé sur des communications point à point. Ainsi, les programmeurs peuvent bénéficier de plusieurs modèles de communication d'une manière flexible et transparente.

Notre mécanisme fournit une communication de *groupes typés*, dans le sens où seules des méthodes définies sur des classes ou des interfaces mises en application par des membres du groupe peuvent être appelées. Grâce à la réflexion et au protocole à méta-objet (MOP) mis en œuvre dans *ProActive*, il ne nous est pas nécessaire de passer la signature de la méthode comme paramètre d'une instruction de communication de groupe.

[1] sauf cas particuliers où ils restent synchrones

3. Communication de groupe typé

3.1. Principes

Notre système de communication de groupe repose sur le mécanisme élémentaire d'invocation distante et asynchrone de méthodes. Comme l'ensemble de la librairie, ce mécanisme est mis en application en utilisant une version standard de Java. Le mécanisme de groupe est indépendant de la plateforme. Il doit être considéré comme une réplique de plusieurs invocations à distance de méthode vers des objets actifs. Naturellement, le but est d'incorporer quelques optimisations à l'exécution, de façon à réaliser de meilleures exécutions qu'un accomplissement séquentiel de n appels de méthode à distance. De cette façon, notre mécanisme est la généralisation du mécanisme d'appel de méthode asynchrone sur des objets distants.

La disponibilité d'un tel mécanisme de communication de groupes simplifie la programmation des applications en regroupant les activités semblables fonctionnant en parallèle. En effet, du point de vue de la programmation, utiliser un groupe d'objets du même type, appelé *groupe typé*, prend exactement la même forme que l'utilisation d'un simple objet de ce type. Ceci est possible grâce à des techniques de réification : la classe d'un objet que nous voulons rendre actif et accessible à distance est étendue au moment de l'exécution, et les appels de méthode sont réifiés.

D'une manière transparente, les appels de méthode dirigés vers un objet actif sont exécutés au travers d'un stub[2] qui est d'un type compatible avec l'objet original. Le rôle du stub est de contrôler l'appel en lui appliquant la sémantique exigée : s'il s'agit d'un appel vers un objet actif distant simple, alors l'invocation à distance asynchrone standard est appliquée ; si l'appel est dirigé vers un groupe d'objets, alors la sémantique des communications de groupes est appliquée comme nous le verrons dans le reste de cette section.

3.2. Création d'un groupe

Les groupes sont créés en utilisant la méthode statique :

```
      ProActiveGroup.newGroup("ClassName", paramètres[], nœuds[]);
```

La superclasse commune à tous les membres du groupe doit être indiquée à la création du groupe, et lui donne ainsi un type minimal. Les groupes peuvent être créés vides, puis remplis par des objets actifs déjà existants. Des groupes non-vides peuvent aussi être construits en utilisant deux paramètres supplémentaires : une liste de paramètres requis pour la construction des membres du groupes et la liste des nœuds où ils seront créés. Le n-ième objet actif est créé avec les n-ièmes paramètres sur le n-ième nœud. Dans ce cas, le groupe est créé et les objects actifs sont construits puis immédiatement inclus dans le groupe. Prenons le cas d'une classe standard Java :

```
class A {
   public A()
   public void foo () {...}
   public V bar (B b) {...}
}
```

Voici un exemple de la création de deux groupes :

```
      // Pré-construction de paramètres pour la création d'un groupe
Object[][] params = { {...} , {...} , ... };

      // Nœuds sur lesquels seront créés les objets actifs
Node[] nodes = { ... , ... , ... };

      // Création 1:
      // Création d'un groupe vide de type "A"
A ag = (A) ProActiveGroup.newGroup("A");
```

[2] représentant local d'un objet distant

```
     // Création 2:
     // Un groupe de type "A" et ses membres sont créés en même temps
A ag2 = (A) ProActiveGroup.newGroup("A", params, nodes);
```

Des éléments ne peuvent être inclus dans un groupe que si leur type est compatible avec la classe spécifiée à la création du groupe. Par exemple, un objet de classe B (B étendant A) peut être inclus dans le groupe. Cependant, étant basées sur le type de A, seules les méthodes définies dans la classe A peuvent être appelées sur le groupe, mais notons que la redéfinition de méthode va fonctionner normalement.

La limitation principale de la construction de groupe est que la classe indiquée au groupe doit être *réifiable*, selon les contraintes imposées par le protocole à méta-objet de *ProActive* : le type ne doit pas être un type primitif (`int`, `double`, `boolean`,...), ni une classe `final`. Dans ces cas, le MOP ne peut pas créer de groupe d'objet.

3.3. Représentations et manipulation de groupes

La représentation typée des groupes correspond à la vue fonctionnelle des groupes d'objets. Afin de fournir une gestion dynamique des groupes, une deuxième (et complémentaire) représentation d'un groupe a été conçue. Cette seconde représentation suit un modèle plus standard : l'interface `Group` étend l'interface `Collection` de Java qui fournit des méthodes telles que `add`, `remove`, `size`, ... Cette gestion de groupes comporte une sémantique simple et classique (ajouter dans le groupe, enlever le n-ième élément, ...) qui fournit une propriété de rang des éléments au sein d'un groupe. Les méthodes de gestion d'un groupe ne sont pas disponibles dans la *représentation typée* mais dans la *représentation de groupe*. La double représentation est un choix de conception. Nous avons choisi l'association de deux représentations complémentaires, l'une pour l'usage fonctionnel et l'autre pour la gestion dynamique. Au niveau de l'implémentation, nous avons pris soin de maintenir une cohérence forte entre les deux représentations d'un même groupe. Les modifications faites sous une forme sont instantanément reportées sous l'autre forme. Pour alterner d'une forme à l'autre nous avons défini deux méthodes (voir figure 1) : la méthode statique `getGroup` de la classe `ProActiveGroup` retourne la représentation de groupe à partir d'une représentation typée. La méthode `getGroupByType` définie dans l'interface `Group` fournit l'opération inverse.

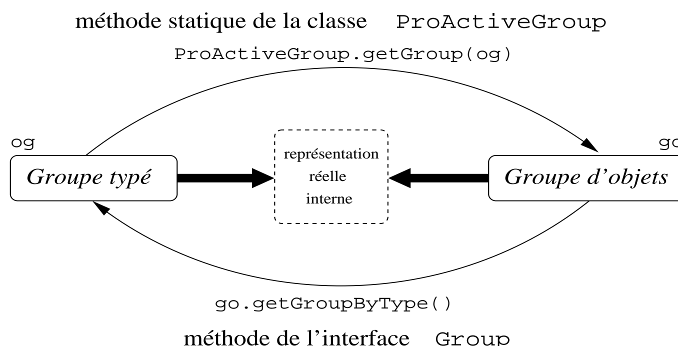

FIG. 1 – *double représentation des groupes*

Voici un exemple de l'emploi de chaque représentation d'un groupe :

```
     // Création d'un objet Java standard et de deux objets actifs
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b  = (B) ProActive.newActive("B", paramsB[], node);
     // Notons que B etend A
```

```
     // Pour la gestion d'un groupe, on obtient la représentation
     // de groupe à partir d'un groupe typé
Group gA = ProActiveGroup.getGroup(ag);

     // On ajoute les objets au groupe
gA.add(a1);
gA.add(a2);
gA.add(b);

     // Une nouvelle référence vers le groupe typé peut être obtenue
A ag_new = (A) gA.getGroupByType();
```

Notons que les groupes ne contiennent pas nécessairement que des objets actifs, mais peuvent contenir des objets standard de Java et des groupes typés (d'où l'obtention de groupes hiérarchiques). La seule restriction est qu'ils soient de classe compatible avec la classe du groupe. La section suivante examinera les implications de tels groupes hétérogènes dans la gestion des communications vers les éléments du groupe.

3.4. Communication de groupe

L'invocation d'une méthode sur un groupe a une syntaxe identique à une invocation de méthode sur un objet Java :

```
     // Une communication de groupe
ag.foo();
```

Bien sûr, un appel de ce type a une sémantique différente : l'appel de méthode est rendu asynchrone et est propagé vers tous les membres du groupe. Un appel de méthode sur un groupe est un appel de méthode sur chaque membre du groupe. Ainsi, si un membre est un objet actif, la sémantique de communication de *ProActive* sera utilisée, s'il s'agit d'un objet Java, la sémantique sera celle d'un appel de méthode classique.
Par défaut, les paramètres de la méthode invoquée sont diffusés à tous les membres du groupe (*broadcast*). Il est également possible, grâce à des méthodes statiques, de changer le comportement des groupes pour que les paramètres soient distribués selon les membres (*scatter*) et non plus diffusés : pour distribuer les données à travers une communication de groupe, il suffit de rassembler ces données au sein d'un groupe et de le passer en paramètre à un appel de méthode.
Voici un exemple :

```
     // Création du groupe de paramètres
B bg = (B) ProActiveGroup.newGroup("B", {{...},{...},...} , {...,...,...});

     // bg est envoyé à tous les membres de ag
ag.bar(bg);

     // Changement du mode de communication de ag (distribution)
ProActiveGroup.setScatterGroup(bg);

     // Chaque membre de bg est envoyé à un membre de ag
ag.bar(bg);
```

3.5. Un groupe comme résultat d'une communication de groupe

La particularité de notre mécanisme de communication est que le résultat de la communication d'un groupe typé est un groupe typé. Ce groupe résultat est construit dynamiquement et de façon transparente au moment de l'invocation de la méthode, avec un futur pour chaque réponse attendue. Le groupe

résultat est mis à jour au fur et à mesure que les réponses arrivent dans le contexte de l'appelant. Toutefois, il peut être instantanément utilisé pour lancer un appel de méthode sachant que le mécanisme d'*attente par nécessité* entre en jeu : si tous les résultats ne sont pas encore arrivés, l'appel de méthode se fera automatiquement au moment de leurs retours.
Dans le code ci-dessous, un nouvel appel de méthode de `f1()` est automatiquement déclenché dès qu'une réponse de l'appel `vg = ag.bar(...)` reviendra dans le groupe `vg` :

```
     // Un appel de méthode qui renvoie un résultat
V vg = ag.bar(...);

     // vg est un groupe typé de type "V"
     // L'opération suivante est aussi une communication de groupe sur
     // les résultats de l'appel précédent.
vg.f1();
```

Comme le montre la figure 2, le placement des éléments dans le groupe est une propriété conservée à travers un appel de méthode : le résultat de l'appel de méthode appliquée au n-ième membre d'un groupe est stocké à la n-ième place dans le groupe résultat.

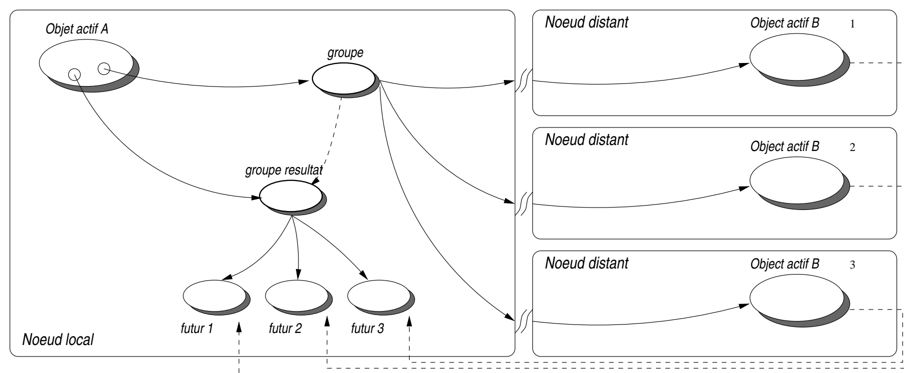

FIG. 2 – *communication de groupe avec groupe résultat*

Des groupes dont le type est basé sur des classes finales ou des types primitifs ne peuvent pas être construits. Par conséquent, la construction dynamique d'un groupe de résultats est également limitée : seules les méthodes dont le type de retour est, soit vide, soit un type réifiable au sens du MOP de *ProActive* peuvent être invoquées sur un groupe d'objets ; autrement, elles lèveront une exception au moment de l'exécution parce que la construction transparente d'un groupe de futurs de types non-réifiable a échoué.

3.6. Synchronisations sur des groupes résultats

Pour profiter du modèle d'appel distant et asynchrone de *ProActive*, quelques nouveaux mécanismes de synchronisation ont été ajoutés. Des méthodes définies sur l'interface `Group` permettent d'exécuter diverses formes de synchronisation (`waitOne`, `waitN`, `waitAll`, `waitTheNth`, `waitAndGet`, ...).
Par exemple :

```
     // Une méthode appelée sur un groupe typé
V vg = ag.bar(...);

     // On accède à la représentation Group de vg
Group gV = ProActiveGroup.getGroup(vg);
```

```
     // Pour attendre et retourner le premier membre de vg
V v = (V) gV.waitAndGetOne();

     // Pour attendre que tous les membres de vg soient arrivés
gV.waitAll();
```

4. Mécanismes d'optimisation

Une implémentation naïve apportait déjà des gains de performances [2]. Un appel de méthode sur un groupe de n objets est plus rapide que le contact des n objets de façon individuelle. Cette première amélioration provient de l'économie de plusieurs réifications d'appels de méthode. Cette opération du méta-niveau construit un objet représentant l'appel de méthode. Lors d'un appel de groupe un seul objet de ce type est construit. Les mesures de performances présentées ont été réalisés sur un réseau local à 100Mb/s, avec 8 stations de travail Pentium III 1Ghz sous Linux avec une JVM 1.4 de Sun.

4.1. Multithreading

L'utilisation de plusieurs fils d'execution (*threads*) permet l'envoi simultané des messages vers chaque destinataire. Les temps des rendez-vous RMI sont ainsi recouverts et non pas cumulés comme cela aurait été le cas si les appels avaient été successifs. Pour conserver la sémantique de *ProActive* une barrière de synchronisation assure que toutes les requêtes ont été transmises aux objets distants et placées dans leur file d'attente avant de passer à l'instruction suivant une communication de groupe.
La figure 3.(a) présente le temps nécessaire pour atteindre des objets distants en fonction du nombre de ces objets. La courbe en trait continu exprime le temps mis par des appels standard point à point. La courbe en pointillés expose les temps mis par un appel de groupe. Plus le nombre d'objets à atteindre augmente plus l'utilisation des groupes est efficace (jusqu'à un facteur 4,2 pour 1000 objets).

4.2. Sérialisation unique

Le protocole RMI se charge de transmettre les paramètres de l'appel à tous les membres en les sérialisant puis en les transmettant sur le réseau. La sérialisation est un processus particulièrement lent de Java [11]. Dans le cas d'une diffusion des mêmes paramètres à tous les objets (*broadcast*), ces paramètres seront sérialisés par chaque thread. Pour éviter ce gaspillage de ressources, une sérialisation unique des paramètres de l'appel est faite par le mécanisme de communication de groupes avant que les appels ne soient délégués à RMI.
La figure 3.(b) présente le temps nécessaire pour atteindre 24 objets distants en fonction de la taille des paramètres de l'appel de méthode. Ces paramètres sont exprimés en nombre d'objet de la classe `Object`. Le procédé devient de plus en plus efficace à mesure que la taille des données à transmettre s'accroit.

5. Conclusion et perspectives

La communication de groupe est un dispositif crucial pour le calcul à hautes performances et le calcul de grille, pour lesquels MPI est généralement le seul modèle largement utilisé de coordination.
L'implémentation courante de notre dispositif optimise la latence du réseau en recouvrant les communications point à point. Nous avons noté que l'implémentation de la diffusion, de la distribution et des opérations de rassemblement pourrait tirer profit d'une structuration et d'une localisation spécifique des données et des activités. Le méta-niveau de *ProActive* est construit de façon à adapter dynamiquement les requêtes et les réponses d'une communication de groupe en fonction de l'hétérogénéité des membres.
Notons aussi que l'exécution d'une communication de groupe sur des groupes hiérarchiques tire naturellement profit de la structure hiérarchique fondamentale du réseau sous-jacent. Si un membre est lui-même un groupe, par exemple un groupe représentant un ensemble d'objets actifs présents sur une grappe d'ordinateurs, seul un appel de méthode sera acheminé vers la grappe puis sera propagé au sein des membres.
ProActive bien que fortement couplé à RMI peut être capable de s'abstraire du protocole de transfert de

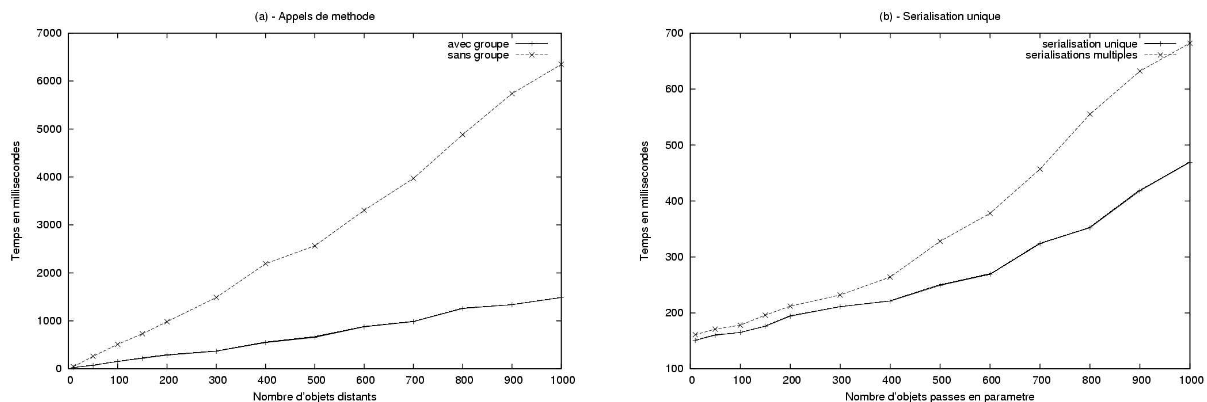

FIG. 3 – *multithreading et sérialisation unique*

données. Une autre alternative que nous sommes en train de considérer, particulièrement valable pour des réseaux locaux de station de travail, est l'utilisation d'une couche de transport multicast.

Bibliographie

1. I. Attali, D. Caromel, and R. Guider. A Step Towards Automatic Distribution of Java Programs. In *FMOODS 2000, Stanford University, September 6-8, 2000*, pages 141–161. Kluwer Academic.
2. L. Baduel, F. Baude, and D. Caromel. Efficient, Flexible and Typed Group Communication in Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, 2002.
3. B. Ban. Design and implementation of a reliable group communication toolkit for java. *Cornwell University*, september 1998.
4. F. Baude, D. Caromel, F. Huet, L. Mestre, and J. Vayssière. Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. In *11th IEEE International Symposium on High Performance Distributed Computing*, 2002.
5. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.
6. D. Caromel, F. Belloncle, and Y. Roudier. The C++// Language. In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.
7. V. Felea and B. Toursel. Methodology for Java Distributed and Parallel Programming Using Distributed Collections. In *Int. Workshop on Java for Parallel and Distributed Computing (IPDPS 2002)*.
8. M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep. Group communication in amoeba and its applications. *Distributed Systems Engineering*, 1(1):48–58, 1993.
9. P. Launay and J.L. Pazat. The Do! project: distributed programming using Java.
10. J. Maassen, T. Kielmann, and H. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming.
11. J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient java RMI for parallel programming. *Programming Languages and Systems*, 23(6):747–775, 2001.
12. A. Nelisse, T. Kielmann, H. Bal, and J. Maassen. Object-based Collective Communication in Java. In *Joint ACM Java Grande - ISCOPE 2001 Conference*.
13. R. Van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system. *CACM, 39(4)*, april 1996.

Ce travail est soutenu par le projet ACI GRID RMI.