École d'hiver GRID 2002:

Calcul Distribué, Méta-Computing, Globalisation des Ressources

Comité d'organisation :

- **Responsable :** Emmanuel JEANNOT, LORIA, Nancy
- Jens GUSTEDT, LORIA, Nancy
- Jean-Louis PAZAT, IRISA-INSA, Rennes
- Stéphane VIALLE, SUPELEC, Metz

Chapitre 1

Communication de groupes typés dans *ProActive*

Laurent Baduel, Françoise Baude & Denis Caromel (INRIA I3S)

1.1 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 de groupes d'activités.

Dans le monde Java, RMI [9], 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, l'usage simple de RMI n'est pas suffisant.

Nous avons développé une librairie 100% Java, *ProActive*, pour la programmation parallèle, distribuée et concurrente. *ProActive* fournit un service d'invocation de méthodes à distance vers des objets actifs distribués de façon transparente, des communications asynchrones avec *futurs* et des mécanismes de synchronisation haut-niveau tels que l'*attente par nécessité*.

Ce chapitre présente la conception d'un mécanisme d'invocation de méthode vers un groupe d'objets et 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 consacrées, comme les collections parallèles et distribuées [5], ou l'exécution de librairie de type MPI dans un style style de programmation SPMD [7].

Notre approche se veut plus générale car elle permet d'établir des modèles al-

ternatifs de programmation parallèle tout en pouvant fournir la communication de groupes à des applications réparties déjà existantes. Ceci permet une réutilisation du code.

1.2 Cadre et précédents travaux

1.2.1 Distribution dans *ProActive*

ProActive est bâti sur les APIs standards de Java[1] et ne nécessite aucune modification de l'environnement d'exécution n'est requise, ni aucun préprocesseur ou compilateur spécial, ni aucune machine virtuelle Java modifiée. Le modèle de distribution et d'activité de *ProActive* est parti d'un effort de simplification et de réutilisation du code d'applications distribuées dans des systèmes à objets [3, 4], en respectant une sémantique précise [1].

1.2.1.a Le modèle de base

Une application distribuée et/ou concurrente construite avec *ProActive* est composée d'entités de grain moyen appelés *objets actifs*. Chaque objet actif a son propre *thread* (ou fil d'activité) qui possède 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[2] avec génération d'objets *futurs* transparents qui sont soumis à des mécanismes de synchronisation tels que l'*attente par nécessité*[3]. Au début de chaque appel distant asynchrone, un court "rendez vous" se produit pour s'assurer que l'appelant se place dans le contexte de l'appelé. C'est à dire que l'appel se bloque le temps que la requête se place dans la file d'attente de l'objet actif distant.

1.2.1.b Lier les objets actifs aux JVMs : les nœuds

Un autre service fournit par *ProActive* (absent de l'API RMI) est la capacité de *créer à distance des objets actifs distants*. Pour cela, il faut être capable d'identifier la JVM et fournir quelques nouveaux services. Les *nœuds* sont des objets définis dans *ProActive* dont le but est de recueillir plusieurs objets actifs dans une entité logique. Ils fournissent une abstraction pour la localisation physique d'un ensemble d'objets actifs. A tout moment, une JVM accueille un ou plusieurs nœuds. La manière traditionnelle d'appeler et de manipuler les nœuds est de les associer à un nom symbolique. Celui-ci est l'URL donnant leur localisation.
Par exemple : `rmi://lo.inria.fr/Node1`.

[1] Java RMI [9], l'API de réflexion [8],...
[2] sauf cas particuliers où ils restent synchrones

Un objet actif est créé 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 composants de ProActive, le concept de nœuds virtuels comme entités pour tracer les objets actifs a été présenté dans [2]. Ces nœuds virtuels sont décrits extérieurement par les descripteurs XML qui sont alors lus à l'exécution par le runtime.

1.2.2 Travaux précédents

Le travail prśenté dans [6] est proche du notre : les objectifs et l'approche sont similaires. Le but du mécanisme de communication de groupes de [6] est de généraliser tous les genres de communication (point à point ou collective, synchrone ou asynchrone, locale ou distante). de nombreux modes de communication sont disponibles, cela exige un certain effort du programmeur afin de choisir le mode de communication désiré.

La différence principale entre le mécanisme de communication de groupe que nous présentons ici et d'autres systèmes[3], 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 en même temps de tous les genres de modèle de communication d'une manière flexible et sans travail suplémentaire. Voici un exemple de code de [6] :

```
class SumImpl extends GroupMember implements Sum...;
     // Création d'un groupe nommé "Name" (N est le nombre de membre)
Group.create("Name", N);

     // Création de chaque membre du groupe
SumImpl sum = new SumImpl();

     // Attachement de cet objet en tant que membre du groupe
     // join bloque jusqu'à ce que N membres aient rejoint le groupe
Group.join("Name", sum);

     // On accède au groupe via un stub
Sum stub = (Sum) sum.createGroupStub();

     // On choisit le mode de communication pour
     // chaque appel de méthode dans le programme
Group.setInvoke(stub, "void add(double v)", Group.GROUP);
```

[3]ISIS, MPI, Horus, ...

```
    // On déclenche un appel de méthode vers chaque membre du groupe
    // Chaque membre ajoute 42.0 à sa propre valeur.
stub.add(42.0);
```

Dans notre approche, grâce à la réflexion et au protocole à méta-objet, il n'est pas nécessaire de passer la signature de la méthode comme paramètre d'une instruction de communication de groupe. Comme dans [6], notre mécanisme fournit une *communication de groupes typés*, dans le sens où seulement des méthodes définies sur des classes ou des interfaces mises en application par des membres du groupe peuvent être appelées.

1.3 Communication de groupes typés

1.3.1 Principes

Notre système de communication de groupes est établi sur le mécanisme élémentaire de *ProActive* pour l'invocation à distance de méthodes asynchrones avec génération automatique de groupe de futurs pour rassembler les réponses.

Ce mécanisme est mis en application en utilisant un Java standard, avec RMI. Le mécanisme de groupe est indépendant de la plateforme et 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 une généralisation du mécanisme d'appel de méthode de *ProActive*, établi sur RMI. Mais rien n'empêche d'employer d'autres couches de transport à l'avenir.

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 actifs du même type, plus tard appelé *groupe typé*, prend exactement la même forme que l'utilisation d'un simple objet actif de ce type. Ceci est possible grâce au fait que la bibliothèque *ProActive* est construite sur des techniques de réification : la classe d'un objet que nous voulons rendre actif et accessible à distance, est reifiée au niveau Méta au moment de l'exécution.

D'une manière transparente, les appels de méthode dirigés vers un objet actif sont exécutés au travers d'un stub[4] qui est d'un type compatible avec l'objet original. Le rôle du stub est de contrôler l'appel comme une entité de première classe et de lui appliquer la sémantique exigée : si c'est un appel vers un objet actif distant simple, alors l'invocation à distance asynchrone standard de *ProActive* est appliquée ; si

[4]représentant local d'un objet distant

l'appel est dirigé vers un groupe d'objets, alors la sémantique des communications de groupes est appliquée.

Cette sémantique est décrite dans le reste de cette section.

1.3.2 Création d'un groupe

Les groupes sont créés en utilisant la méthode statique :
`ProActiveGroup.newActiveGroup("ClassName",...)`.

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 paramétres 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 B bar (...) {...}
}
```

Voici un exemple de créations de groupe :

```
    // Pré-construction de paramètres pour la création d'un groupe
Object[][] params = {{...} , {...} , ... };
    // Nœuds sur lesquels seront créés les objets actifs
Node[] nodes = { ... , ..., ... };

    // Solution 1:
    // Création d'un groupe vide de type "A"
A ag1 = (A) ProActiveGroup.newActiveGroup("A");

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

Des éléments ne peuvent être inclus dans un groupe que si leur type égalent ou étendent 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.

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.

1.3.3 Représentations et manipulation de groupes

La représentation typée des groupes que nous avons présenté dans la sous-section précédente 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. Afin de contrôler un groupe, c'est cette deuxième représentation qui doit être employée. 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 parmi deux possibilités : la première aurait consisté à employer des méthodes statiques de la classe `ProActiveGroup` afin de contrôler les groupes, laissant les groupes à juste une seule représentation. La seconde, que nous avons choisi, consiste en l'association de deux représentations complémentaires d'un groupe, l'une pour l'usage fonctionnel et l'autre pour la gestion. Au niveau de l'implémentation, nous avons fait attention à conserver une cohérence forte entre les deux représentations d'un même groupe. Les modifications exécutées sous une forme sont instantanément reportées sous l'autre forme. Pour passer d'une forme à l'autre nous avons définis deux méthodes : 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.

Voici un exemple de quand et comment employer 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(ag1);

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

    // L'ajout de membres prend effet immédiatement sur la forme
    // typée du groupe, une méthode peut être invoquée et
    // atteindra tous les membres du groupe.
ag1.foo();

    // Une nouvelle référence vers le groupe typé peut être obtenue
A ag1new = (A) gA.getGroupByType();
```

Notons que les groupes ne contiennent pas nécessairement que des objets actifs, mais peuvent contenir des objets standard de Java. La seule restriction est qu'ils soient de classe compatible avec la classe du groupe. Nous verrons en 1.3.4 les implications de tels groupes hétérogènes dans la gestion des communications vers les éléments du groupe.

1.3.4 Communication de groupes

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

```
Object[][] constructorArray = {{...},{...},...};
Node[] nodes = {...,...,... };
A ag1 = (A) ProActiveGroup.newActiveGroup("A", constructorArray, nodes);
...
ag1.foo(...); // Une communication de groupe
```

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 en utilisant plusieurs threads. Comme dans le modèle de base de *ProActive*, un appel de méthode vers un groupe est non-bloquant et un futur transparent est créé pour collecter les résultats. 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.

Les paramètres de la méthode invoquée sont diffusés à tous les membres du groupe. Il est également possible, grâce à des méthodes statiques de changer le comportement des groupes pour que les paramètres soient distribués selon les membres et non plus diffusés.

1.3.5 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 résultat est construit dynamiquement et de façon transparente au moment de l'invocation de la méthode, avec un futur pour chaque réponse attendue. Ce résultat est mis à jour au fur et à mesure que les réponses arrivent dans le contexte de l'appelant. Toutefois, le groupe résultat peut être instantanément utilisé pour lancer un appel de méthode, le mécanisme d'*attente par nécessité* entre en jeu : si tous les résultats ne sont pas encore arrivés, l'appel de méthode se fera au fur et à mesure de leur retour.

Dans le code ci-dessous, un nouvel appel de méthode de `f1()` est automatiquement déclenché dès qu'une réponse de l'appel `vg=ag1.bar()` reviendra dans le groupe `vg` :

```
     // Un appel de méthode qui renvoie un résultat
V vg = ag1.bar();
     // vg est un groupe type de type "V"
     // L'opération suivante est aussi une communication de groupe sur
     // les résultats de l'appel précédant.
vg.f1();
```

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.

Comme expliqué dans 1.3.2, des groupes dont le type est basé sur des classes finales ou des types primitifs ne peuvent pas être construits. Ainsi, la construction dynamique d'un groupe de résultats est également limitée. En conséquence, seules les méthodes dont le type de retour est, soit vide, soit un type réifiable dans le sens du protocole à méta-objet 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-reifiable a échoué.

Pour profiter du modèle d'appel distant et asynchrone de *ProActive*, quelques nouveaux mécanismes de synchronisation ont été ajoutés. Des méthodes statiques définies dans la classe `ProActiveGroup` permettent d'exécuter diverses formes de

synchronisation (`waitOne`, `waitN`, `waitAll`, `waitTheNth`, `waitAndGet`, ...).
Par exemple :

```
    // Une méthode appelée sur un groupe typé
V vg = ag1.bar();

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

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

1.4 Conclusion et perspectives

La communication de groupes est un dispositif crucial pour le calcul à hautes performances et le calcul de grille, pour lesquels MPI est généralement le seul modèle disponible de coordination.

L'implémentation courante de notre dispositif optimise seulement 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 spécifique des données et des activités. Comme le méta-niveau de *ProActive* est structuré de façon à adapter dynamiquement les requêtes et les réponses en fonction de l'appelant et de l'appelé, nous bénéficions de cette adaptabilité dans les communications de groupe ; et cela même au sein d'un unique groupe, selon l'hétérogénéité de ses membres.

Une autre alternative que nous sommes en train de considérer, particulièrement valable pour des réseaux de postes de travail, est l'utilisation d'une couche de transport multicast. Notons 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 propagé au sein des membres de la grappe.

Alors que les appels de méthode vers des groupes sont actuellement mis en application par des appels RMI vers chaque membre du groupe, nous travaillons actuellement à une autre optimisation qui devrait s'avérer efficace dans le contexte de Java : une sérialisation unique des paramètres de la méthode pour une diffusion vers tous les membres du groupe.

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] 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. To appear.

[3] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[4] 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.

[5] 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)*.

[6] J. Maassen, T. Kielmann, and H. Bal. Generalizing Java RMI to support efficient group Communication. In *ACM Java Grande Conference*, 2000.

[7] A. Nelisse, T. Kielmann, H. Bal, and J. Maassen. Object-based Collective Communication in Java. In *Joint ACM Java Grande - ISCOPE 2001 Conference*.

[8] Sun Microsystems. Java Core Reflection, 1998. http://java.sun.com/products/jdk/1.2/docs/guide/reflection.

[9] Sun Microsystems. Java Remote Method Invocation Specification, October 1998. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.