Typed Groups for the Grid

Laurent Baduel

▶ **To cite this version:**

Laurent Baduel. Typed Groups for the Grid. Modeling and Simulation. Université Nice Sophia Antipolis, 2005. English. NNT : . tel-00009757

HAL Id: tel-00009757

https://theses.hal.science/tel-00009757

Submitted on 13 Jul 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

École Doctorale
« Sciences et Technologies de l'Information et de la Communication »
de Nice - Sophia Antipolis

Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES

TYPED GROUPS FOR THE GRID

---

par

Laurent BADUEL

Thèse dirigée par **Françoise BAUDE** *et* **Denis CAROMEL**

au sein de l'équipe OASIS,

équipe commune de l'I.N.R.I.A. Sophia Antipolis et du laboratoire I3S

présentée et soutenue publiquement le 8 juillet 2005 à l'E.S.S.I. devant le jury composé de

Président du Jury	Johan MONTAGNAT	Laboratoire I3S, Nice - Sophia Antipolis
Rapporteurs	Henri BAL	Vrije Universiteit, Amsterdam
	André SCHIPER	École Polytechnique Fédérale de Lausanne
	El Ghazali TALBI	Laboratoire d'Informatique Fondamentale de Lille
Invité industriel	Emmanuel CECCHET	Consortium ObjectWeb
Directeur de thèse	Denis CAROMEL	Université de Nice - Sophia Antipolis, Institut Universitaire de France
Co-directrice de thèse	Françoise BAUDE	Université de Nice - Sophia Antipolis

To my parents,
— Laurent

Contents

List of Figures ix

Acknowledgements xi

I Résumé étendu en français ***(Extended french abstract)*** **xiii**

Introduction et objectifs **xv**
- 1.1 Contexte . . . xv
- 1.2 Besoins . . . xv
- 1.3 Organisation de la thèse . . . xvi

Résumé **xix**
- 2.1 État de l'art . . . xix
- 2.2 *ProActive* . . . xxi
- 2.3 Communication de groupe typé . . . xxiii
- 2.4 Implémentation et évaluation par micro-tests . . . xxvi
- 2.5 Une application test : Jem3D . . . xxviii
- 2.6 SPMD orienté-object . . . xxx
- 2.7 Travaux en cours et travaux collaboratifs . . . xxxii

Conclusion **xxxv**
- 3.1 Accomplissements . . . xxxv
- 3.2 Perspectives . . . xxxvi

II Thesis **1**

1 Introduction and objectives **3**
- 1.1 Context . . . 3
- 1.2 Needs . . . 3
- 1.3 Thesis organization . . . 4

2 Related work **7**
- 2.1 Group properties . . . 7
  - 2.1.1 Structure . . . 7
  - 2.1.2 Reliability and semantics . . . 8
  - 2.1.3 Dynamicity . . . 10
  - 2.1.4 Ordering . . . 10
  - 2.1.5 User interface . . . 11
- 2.2 Group toolkits . . . 12
  - 2.2.1 Internet multicast . . . 12
  - 2.2.2 Isis and Horus . . . 15
  - 2.2.3 Parallel Virtual Machine . . . 17
  - 2.2.4 Message Passing Interface . . . 18

2.2.5 Object Group Service: a CORBA Service . . . 19
2.2.6 JGroups . . . 20
2.2.7 Group Method Invocation . . . 21
2.3 Analysis of related work . . . 23
2.3.1 Drawbacks . . . 23
2.3.2 Proposal . . . 24

3 *ProActive* **27**
3.1 Programming model . . . 27
3.1.1 Distribution model . . . 27
3.1.2 Active objects . . . 28
3.1.3 Communication by messages . . . 29
3.1.4 Synchronization . . . 31
3.1.5 Service policy and control of the activity . . . 31
3.2 Environment and implementation . . . 32
3.2.1 Mapping active objects to JVMs: Nodes . . . 32
3.2.2 MOP: Meta-Objects Protocol . . . 34
3.2.3 Migration . . . 35

4 Typed group communication **39**
4.1 The typed group model . . . 39
4.1.1 Objectives . . . 39
4.1.2 Typed groups . . . 40
4.2 Application Programming Interface . . . 41
4.2.1 Group creation . . . 41
4.2.2 Group of `Objects`: a `Collection` and a `Map` . . . 43
4.2.3 The communication is a method call . . . 44
4.2.4 Group of *futures* . . . 45
4.2.5 Synchronization . . . 46
4.2.6 Broadcast vs. scatter . . . 47
4.2.7 Operation semantics on result group . . . 49
4.3 Advanced group features . . . 50
4.3.1 Errors and exceptions . . . 50
4.3.2 Hierarchical group . . . 53
4.3.3 Active group . . . 55
4.3.4 Dynamic dispatch group . . . 56

5 Implementation and micro-benchmarks **61**
5.1 Implementation details . . . 61
5.1.1 Motivations . . . 61
5.1.2 A proxy for the group . . . 62
5.2 Features . . . 63
5.2.1 Thread pool . . . 63
5.2.2 Factorization of common operations . . . 65
5.3 Micro-benchmarks . . . 66
5.3.1 In a cluster context . . . 66
5.3.2 In a Grid context . . . 69
5.4 Matrix multiplication . . . 71

6 Applicative benchmark: Jem3D **75**
6.1 Basic architecture of Jem3D . . . 76
6.1.1 Geometry definition . . . 76
6.1.2 Application aspects . . . 77
6.1.3 Overall skeleton and control . . . 78
6.2 Design of the parallel and distributed version of Jem3D . . . 79
6.2.1 Basic ideas and principles . . . 79
6.2.2 Partitioning, local and remote objects . . . 80

6.3 A group communication model to enhance performances . . . 81
6.4 Benchmarking . . . 82
6.4.1 Benchmarks on cluster . . . 82
6.4.2 Benchmarks on an Intranet heterogeneous cluster . . . 85
6.4.3 Benchmarks on a grid using a fast RMI protocol . . . 87

7 Object-Oriented SPMD **93**
7.1 Context and related works . . . 93
7.1.1 SPMD programming . . . 94
7.1.2 SPMD programming with an object-oriented flavor . . . 94
7.1.3 Our SPMD programming approach . . . 95
7.2 Object-Oriented SPMD . . . 96
7.2.1 Design and principles . . . 96
7.2.2 Requirements . . . 96
7.2.3 Main principles of OO SPMD . . . 97
7.2.4 Topologies . . . 98
7.2.5 Synchronization barriers . . . 100
7.2.6 Extensibility and reactivity . . . 101
7.3 Example and benchmarks . . . 102
7.3.1 MPI Jacobi . . . 103
7.3.2 OO SPMD Jacobi . . . 104
7.3.3 Benchmarks . . . 105
7.4 Comparison with the MPI API . . . 106

8 Ongoing and collaborative work **111**
8.1 Group behavior component . . . 111
8.2 Using IP multicast . . . 114
8.3 Components . . . 116
8.4 Peer-to-peer computing . . . 119

9 Conclusion **123**
9.1 Achievements . . . 123
9.2 Perspectives . . . 124

Bibliography **127**

Abstract & *Résumé* **136**

List of Figures

2.1 Group structures . . . 8
2.2 FIFO message ordering . . . 10
2.3 Causal message ordering . . . 11
2.4 Total message ordering . . . 11
2.5 RMTP Network Architecture . . . 15
2.6 Horus layers . . . 17
2.7 The Object Group Service . . . 19
2.8 JGroups architecture . . . 21
2.9 RepMI . . . 22
2.10 The integration approach . . . 24
2.11 The service approach . . . 24
2.12 The interception approach . . . 25

3.1 Seamless parallelization and distribution with active objects . . . 28
3.2 Execution of an asynchronous and remote method call . . . 30
3.3 Base-level and meta-level of an active object . . . 34
3.4 Migration and tensioning . . . 36

4.1 *Typed group* and *Group* representations . . . 43
4.2 One-way method call on a group . . . 45
4.3 Method call on a group, with results . . . 46
4.4 Scattered parameters . . . 48
4.5 Exception mechanism of an asynchronous method call on group . . . 52
4.6 Exception mechanism of a one-way method call on group . . . 53
4.7 The `add` method . . . 54
4.8 The `addMerge` method . . . 54
4.9 Hierarchical groups . . . 54
4.10 Hierarchical and active groups . . . 57
4.11 Differences between basic groups and dynamic dispatch groups: behavior and usage 58

5.1 Adaptative thread pool . . . 64
5.2 Factorization of common operations . . . 66
5.3 Method call on cluster . . . 67
5.4 Method call on cluster depending on group size . . . 68
5.5 Speedup on cluster . . . 68
5.6 The map of current *Grid'5000* . . . 69
5.7 Method call on Grid . . . 70
5.8 Method call on Grid with a hierarchical group . . . 72
5.9 *Broadcast-Broadcast* matrices multiplication performances . . . 73

6.1 Definition of an element and a control volume in 2D and 3D . . . 77
6.2 Definition of a facet in 2D and 3D . . . 78
6.3 Overall application skeleton . . . 79
6.4 Architecture of the sequential version of Jem3D . . . 80
6.5 Architecture of the distributed version of Jem3D . . . 81

6.6 Average duration of 100 iterations 83
6.7 Speedup 84
6.8 Intranet computing 86
6.9 Grid computing 88
6.10 Rendering of a Jem3D computation 90
6.11 JECS: a Java Environment for Computational Steering and Visualization 90
6.12 A more complex and irregular mesh 91

7.1 An SPMD group 97
7.2 Topologies 98
7.3 Topologies classes 99
7.4 Data distribution schemes 103
7.5 Distributed algorithm 103
7.6 Performances of C/MPI and Java/OO SPMD versions 106
7.7 OO SPMD scalability in a peer-to-peer experiment 107
7.8 MPI to Java translations 109

8.1 The communicator component 115
8.2 Group RMI vs Group IP Multicast 117
8.3 The three types of grid components 118
8.4 Redistribution from M components to N components 119
8.5 A peer-to-peer infrastructure 120
8.6 An heart beat sent as a group communication 121

Acknowledgements

During the course of my thesis work, there were many people who were fundamental in helping me. Without their guidance, help, and patience, I would have never been able to accomplish the work of this thesis. I would like to take this opportunity to acknowledge them for their help, either technical or moral, or both.

First of all, I would like to thank *Denis Caromel*, my supervisor. He has constantly encouraged and motivated me. I thank him for the trust he placed on me since the very beginning of our fecund collaboration. I admire his vast knowledge and skill in many areas of computer science. I am grateful for all those opportunities, to make interesting research, to publish, to travel, to teach, etc. he offered me.

I would like to express my gratitude to my second supervisor, *Françoise Baude*, whose expertise, understanding, patience, and permanent availability added considerably to my graduate experience. I appreciate her precious assistance in writing reports and the infinite kindness she attests everyday to all her students.

My greetings go also to *Henri Bal*, *André Schiper*, and *El Ghazali Talbi* who honored me by accepting to be reviewers for my thesis. I thank *Emmanuel Cecchet* for taken part of the jury, and *Johan Montagnat* for presided at it.

I am sure that the first reviews of my thesis were an horrible, boring, and displeasing work. So I am very grateful to *Françoise B.*, *Denis C.*, *Alexandre D.*, and *Fabrice H.* who did it with courage and with great care.

I really enjoyed the time I was PhD student. I known what was responsible for that: it was the wonderful atmosphere that was in the OASIS research group, and more generally in the INRIA lab at Sophia Antipolis. I met here people from many countries who generously shared their work and life experiences with me. I thank all the OASIS members and my friends at INRIA for those four years in their company.

A special acknowledgement goes to *Arnaud Contes*, my office-mate. I know him for a very long time now. Our friendship started when we both began an internship in the OASIS team, I hope it will last for many years again. We immersed ourselves together in the *ProActive* library and the distributed computing world. We had shared happiness, irritation, frustration, success, and many pleasant moments. *Arnaud*, my words for you are: *"Courage, you are the next!"*

I would like to salute other PhD students with who I shared this adventure. Older ones gave me advices; younger ones gave me fresh views on my work. Thank you *Fabrice H.* ("tonton looz"), *Julien V.* (master of the MOPs), *Carine C.* (I'm happy you feel well in your new position in England), *Ludovic H.* the untiring sportsman, *Rabéa B.* (congratulations for your position), *Felipe L.* the Mexican expert of French spoonerism, "papi" *Rémi C.*, the "chi-Chilean" clan: *Tomás B.* and *Javier B.*, *Christian B.* my sport partner, *Olivier N.* the new World Company agent, *Laurent Q.* my homonym partner, *Alexandre G.* the "celtic", *Christian D.* my "punk" friend, *Matthieu M.* the Britain engineer-student, *Alexandre D.* the Apple fashion victim, and those I may have forgotten.

I thank the engineers of the team that frequently helped me. Thank *Lionel M.*, *Romain Q.*, and *Igor "the kangaroo" R.*.

A research group would not be completed without a dark army of interns. Even if they did not stay for a long time with us, they often did a very valuable work. I would like to salute *Santosh A.* (Ma poule, I promise I will visit you in Italy soon), *Guillaume C.* the gifted geek, the D.L.P. team (*Benjamin "the squirrel" B.*, *Sébastien "duffman" C.*, and *Katia P.*), *Nicolas G.* the little electromagnetic genius, *Christian L.* the Austrian "topgnu", *Christophe M.* the rocker of diamond, *Jonathan S.* the German uber-coder, the "marseillais": *Patrice F.* and *Olivier C.*, and *Damien P.* (you work too much).

I would like to thank my friends for their presence, especially when I returned frustrated from the work (it was too often). They bore my mood swings with patience and kindness. Thank you *Ludovic "casimir" L.*, *Fabian B.* the rich merchant banker, *Virginie L.* "la ch'tite", *Alexandre G.* the eternal student, *Stéphane V.* (stop trying to convince me to skydive, I should accept), *Stéphane C.* said "Yeyette", *Thomas S.* "grandes oreilles" and the pretty *Maïa S.*.

I thank *Claire Senica*, our project assistant at INRIA, whose efficiency frequently helped me to solve administrative annoyances. I thank her for her attention to our PhD students' concerns. Thank also to *Patricia Lachaume*, our project assistant at I3S, for the same reasons.

I will never forget my teaching experience. I am convinced that students learnt me as much as I learnt them. For all those questions that made me doubt, and for the efforts of organization and clarification I had to do, I thank them. I am grateful to the professors and professor assistants with who I taught in the University of Nice - Sophia Antipolis.

Of course, I would like to thank my family for the support they provided me through my entire life. I must acknowledge my parents, *Irène* and *Pierre*, without whose love, encouragement, and assistance, I would not have finished this thesis.

My last words are for *Isabelle Attali*, the project leader of the OASIS team, who tragically died with her two young sons *Ugo* and *Tom* in the Tsunami, Sri Lanka, December 2004. I thank you for the great opportunities you gave me and for the constant interest you attest in our works. Your generosity, your dynamism, and your kindness were the sun of OASIS.

This thesis is also dedicated to you.

Part I

Résumé étendu en français

(Extended french abstract)

Introduction et objectifs

1.1 Contexte

De nombreuses applications destinées aux grilles de calculs telles que les simulations numériques, l'acquisition et l'analyse de données, intègrent des calculs intensifs et la gestion d'énormes quantités de données qui doivent être transférées et traitées sur de multiples machines de façon à améliorer les performances globales de traitement.

Ces dernières années, beaucoup d'intergiciels et d'outils ont été développés pour les grilles (Globus [GLO], Legion [NAT 02], Unicore [UNI], Condor-G [FRE 01], HiMM [SAN 03], ...). Généralement ces intergiciels adoptent des mécanismes fiables de communications point-à-point. Cependant les applications destinées aux grilles peuvent souvent tirer profit de schémas de communication de type un-vers-plusieurs ou plusieurs-vers-plusieurs [JEA 03, MAI 02].

Fournir un intergiciel pour le calcul sur grilles incorporant une implémentation efficace de l'abstraction des groupes au niveau de la programmation peut aider au développement de logiciels et réduire le coût des communications à grandes et même petites échelles. Les performances d'un programme s'exécutant sur des machines parallèles à mémoires non-partagées sont grandement dépendantes de l'efficacité des communications inter-processus. Les environnements de programmation parallèle n'offrent souvent qu'un support médiocre pour des modèles de communication haut niveau. Cette thèse propose un modèle de communication de group à haut niveau pour ces architectures.

Selon l'*Object Group design pattern* [MAF 96], un groupe est le représentant local d'un ensemble d'objets distribués sur des machines interconnectées et à qui on peut assigner l'exécution d'une tâche. Le modèle de l'«objet group» spécifie que lorsqu'une méthode est invoquée sur un groupe, l'environnement d'exécution envoie une requête d'invocation de la méthode sur les membres du groupe, attend une ou plusieurs réponses des membres selon une politique définie, et retourne le(s) résultat(s) au client. Ces groupes sont habituellement dynamiques, c'est-à-dire que l'ensemble constitué par les membres peut changé à l'exécution.

1.2 Besoins

Depuis quelques années, l'intérêt porté au langage Java pour construire des applications de calculs à hautes performances n'a cessé de croître. Java fournit un modèle de programmation orienté-objet avec support de la concurrence, ramasse-miettes, et sécurité. Java est également capable d'exprimer le multitâche et l'invocation distante de méthode (RMI : *Remote Method Invocation*) [SUN98], une version orienté-objet de l'appel distant de procédure (RPC : *Remote Procedure Call* (RPC) [BIR 84]).

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

Au niveau de la programmation, les groupes peuvent faciliter le développement de logiciels puisqu'ils simplifient l'implémentation de modèle de programmation. Au niveau des communications, les groupes peuvent réduire le coût des communications plus plusieurs raisons. D'abord la distribution du même message à un ensemble de receveurs peut bénéficier de l'abstraction des groupes puisque des optimisations spécifiques peuvent être appliqués même si la couche de transport est basée sur un modèle point-à-point. Par exemple, le transfert des objets sur le réseau nécessite une sérialisation avant l'envoi. La sérialisation étant un processus significativement lent, envoyer la même copie sérialisée de l'objet aux membres du groupes améliore grandement le temps totale de la communication de groupe. Ensuite, le mécanisme haut niveau de communication de groupe peut aussi être implémenté sur un protocole de transport de type un-vers-plusieurs.

1.3 Organisation de la thèse

Le but de notre recherche est de définir un mécanisme efficace et élégant de communication de groupe dédié particulièrement au calcul sur Grille. Ce mécanisme doit être basé sur un modèle qui s'intègre parfaitement dans le modèle orienté-objet de Java. Ce travail s'applique à étendre le mécanisme de RMI en fournissant la possibilité d'exprimer également des communications multipoints de façon à améliorer les performances des applications et à réduire la complexité de programmation de ces applications réparties. Plus généralement, nous visons à fournir un modèle qui aide à la définition et à la coordination d'activités distribuées.

Le document est organisé comme suit :

- Le chapitre 2 (résumé dans la section 2.1, page xix) donne une vue d'ensemble des travaux relatifs aux mécanismes de communication de groupe dans différents domaines. J'en identifie les principales caractéristiques selon les domaines d'applications et présente un état de l'art des outils les plus remarquables. Une discussion sur les fonctionnalités requises conclue le chapitre.

- Dans le chapitre 3 (résumé dans la section 2.2, page xxi), je présente l'intergiciel *ProActive*. J'introduis le modèle de programmation et donne une description de ses fonctionnalités pour le calcul parallèle et distribué. Cette présentation décrit les principaux éléments de l'interface de programmation d'application (API).

- Le chapitre 4 (résumé dans la section 2.3, page xxiii) introduit un model orienté-objet pour un mécanisme de communication de groupe haut-niveau. Je décris l'approche typée par la présentation de l'API ajoutée dans l'intergiciel *ProActive*. De cette façon, nous observerons les fonctionnalités de ce mécanisme.

- Le chapitre 5 (résumé dans la section 2.4, page xxvi) donne des détails sur l'implémentation. Il présente les points principaux d'optimisation su système. Des évaluations de performances sur cluster et grilles concluent le chapitre.

- Dans le chapitre 6 (résumé dans la section 2.5, page xxviii) je présente une application numérique, nommée Jem3D, dont l'implémentation est basée sur le mécanisme des com-

munications de groupe pour l'échange intensif des données et la synchronisation des processus. Les performances de l'application, ainsi que son passage à l'échelle, sont présentés sur plusieurs plateformes.

- Le chapitre 7 (résumé dans la section 2.6, page xxx) introduit le modèle de programmation SPMD orienté-objet. Ce chapitre décrit les concepts existants de la programmation SPMD et présente notre approche objet. Le code et les performances sont analysés en utilisant un programme de typique : les itérations de Jacobi.

- Dans le chapitre 8 (résumé dans la section 2.7, page xxxii), je présente les plus récentes fonctionnalités liées au mécanisme des communications de groupe qui sont encore en cours de développement. Il s'agit du composant de comportement, de l'utilisation d'IP multicast, de l'implémentation des composants Fractal, et du calcul pair-à-pair.

- Enfin, je conclue en dressant le bilan des accomplissements de cette thèse et en présentant les perspectives.

Résumé

2.1 État de l'art

Par opposition à la communication point-à-point, la communication multipoint implique deux processus ou plus. Un tel type de communication est employé dans une panoplie croissante d'applications à large échelle. La diffusion consiste à envoyer les mêmes données répliquées à plusieurs récepteurs. Beaucoup d'applications réparties exigent la livraison des messages et des services d'adhésion pour un groupe de processus.

Propriétés d'un groupe

Selon leurs structures, les groupes peuvent supporter un large spectre d'applications. La structure d'un groupe est choisie selon les besoins de l'application. Pour chaque besoin de programmation d'une application correspond une structure de groupe plus appropriée. Généralement quatre structures de groupe sont proposées : les *groupes de pairs*, les *groupes clients-serveur*, les *groupes de diffusion*, et les *groupes hiérarchiques*.

Un groupe peut être *ouvert* ou *fermé*. Dans un groupe ouvert tout objet ou processus peut envoyer des messages aux membres du groupe. Dans un groupe fermé seuls les membres du groupe peuvent envoyer des messages. Un groupe est *dynamique* si des membres peuvent être ajoutés ou supprimé pendant l'exécution. Un groupe non dynamique est dit *statique*. Un groupe est dit *égalitaire* si tous les membres du groupe ont une même activité et aucun d'entre eux n'est responsable de services supplémentaires.

La *sémantique de livraison* définit si une communication de groupe réussit à communiquer un message au groupe. Habituellement, il y a cinq possibilités pour cette sémantique : le *zéro livraison*, la *livraison unique*, la *n livraison*, la *livraison par quorum*, et la *livraison atomique* (ou *totale*). La *sémantique de réponse* définit le nombre de réponses attendues pour considérer réussite la réponse d'une communication de groupe. On identifie cinq sémantiques : le *zéro réponse*, la *réponse unique*, la *n réponse*, la *réponse par quorum*, et la *réponse totale*.

Systèmes de communications de groupe

Nous avons identifié quatre communautés qui étudient les communications de groupe ; chacune pour ses propres intérêts qui peuvent différer ou être tout à fait similaires aux autres. D'abord, la communauté Internet se concentre sur les aspects réseaux et protocole. Ensuite, la communauté des systèmes d'exploitation s'intéresse aux logiciels d'exploitation répartis. Puis, la communauté d'algorithme distribué est impliquée dans la conception d'application tolérante aux pannes. Pour finir, la communauté du parallélisme est intéressée par les plateformes d'exécution pour des applications parallèles.

Dans cette section certains des systèmes de communications de groupe les plus significatifs présentés. Ils sont conçus soit pour l'Internet, soit pour les systèmes d'exploitation répartis, soit pour des systèmes tels que UNIX (en tant qu'outils), soit pour des applications tolérantes aux pannes, soit pour des environnements d'applications distribuées. Le plus éloigné de nos travaux est présenté d'abord, menant aux projets qui sont les plus proches de nos recherches.

- Le besoin de communication de groupe dans le monde d'Internet diffère du besoin dans le monde applicatif. IP Multicast est la base de la plupart des autres protocoles. Il fournit un dispositif de base pour l'émission de messages. MTP, XTP, RAMPE et RMTP sont d'autres protocoles qui représentent l'ensemble des principales solutions.
- Isis et Horus sont deux projets développés à l'université de Cornell pour construire des applications réparties, tolérantes aux pannes. Leurs modèles de programmation sont basés sur la *synchronisation virtuelle* des processus. Les sémantiques de communication sont définit par l'utilisation d'un ensemble spécifiques de primitives.
- PVM (*Parallel Virtual Machine*) et MPI (*Message Passing Interface*) sont des bibliothèques pour la conception d'applications distribuées. Leurs modèles fournissent une abstraction de la plateforme d'exécution de manière. Ces bibliothèques proposent des mécanismes pour réaliser des opérations collectives (communications et synchronisation) impliquant plusieurs processus.
- Enfin, plus proches de nos considérations, des outils tels que l'OGS (*Object Group Service*) pour CORBA, et JGroups et GMI (*Group Method Invocation*) pour Java, introduisent des mécanismes de communication de groupe dans des langages à objets. Contrairement à JGroups, OGS et GMI tentent de tirer partie du style de programmation orienté-objet pour réaliser des communications de groupe.

Proposition

Notre but est de libérer le programmeur de l'implémentation d'un code de communication complexe nécessaire pour la communication de groupe. Nous voulons réaliser ceci en permettant au programmeur de se focaliser sur l'application elle-même. Les communications de groupe doivent être exprimées en utilisant des invocations de méthode distantes juste comme le RMI exprime les communications point-à-point. L'intégration dans les langages orientés-objet devient naturelle.

Cependant nous ne voulons pas forcer le programmeur à implémenter ou prolonger des interfaces et des classes spécifiques. En effet, un tel engagement apporterait des contraintes à la création de l'application et porterait atteinte à la dynamicité pendant l'exécution. Les préoccupations au sujet des groupes doivent être abordées par le mécanisme de groupe sans impacts sur l'écriture du code du programmeur. L'interface commune des membres d'un groupe doit être suffisante pour exprimer le plus grand ensemble de schémas de communication, tels que diverses stratégies d'envoi et de réception de messages. Naturellement, une interface est nécessaire pour contrôler explicitement des groupes. Cette interface doit définir la création, l'ajout et la suppression de membres, . . . : des opérations qui ne sont pas accessibles par l'interface de membres. Nous visons clairement à séparer les préoccupations de gestion de groupe des préoccupations des aspects fonctionnels (c'est à dire des communications par invocation de méthode). La séparation des préoccupations est essentielle pour maîtriser la complexité des opérations collectives.

Conclusion

Selon leurs besoins respectifs, chaque communauté considère différemment les communications de groupe. Des problèmes peuvent être spécifiques à une certaine communauté et de fait, être d'importance mineur ou même totalement ignoré dans les autres communautés. Par exemple, les environnements tolérant aux pannes (par réplication) se concentrent sur l'ordonnancement des processus. D'un autre côté, pour construire des applications distribuées, l'importance se porte plutôt sur les schémas de communication. Nous nous intéressons à la conception d'applications réparties ; notre challenge est de fournir aux programmeurs les outils de communication de groupe les plus simples et les plus efficaces.

2.2 *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 [CAR 93, CAR 96], en respectant une sémantique précise [ATT 00].

Modèle de programmation

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 avec génération d'objets *futurs* transparents qui sont soumis à des mécanismes de synchronisation tels que l'*attente par nécessité* [CAR 93]. 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, notamment 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.

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

```
A a = (A) ProActive.newActive("A", parametres, noeud);
```

ProActive se base sur Java RMI pour les communications entre objets. Un appel à Java RMI est bloquant. Ceci peut causer des latences inutiles dans l'exécution d'un programme : par exemple l'attente d'un résultat qui ne sera utilisé que plus tard. Par défaut *ProActive* fournit des communications asynchrones (et à sens unique), mais peut aussi communiquer de façon synchrone.

- ***Appel synchrone*** : l'appel de méthode est bloquant, le fil d'exécution est suspendu jusqu'à l'arrivée du résultat de la méthode invoquée avant de reprendre le fil d'exécution.
- ***Appel asynchrone*** : l'appel est non bloquant, l'exécution du programme sans que le résultat soit revenu. Toutefois, un *rendez-vous* assure que la requête est bien parvenue dans le contexte de l'appelé avant que l'activité ne reprenne. Un objet *futur* est créé en attente du résultat.
- ***Appel à sens unique*** : l'appel est non bloquant (le rendez-vous est toujours présent). Aucun résultat n'est attendu ; aucun futur n'est créé.

Ces caractéristiques de synchronisation sont adaptées à chaque méthode d'un objet actif en fonction de sa signature. Sauf configuration explicite de l'utilisateur, une méthode ne renvoyant aucun résultat (`void`) sera à sens unique, une méthode renvoyant des objets non-réifiables sera appelée de façon synchrone, et une méthode renvoyant des objets réifiables sera appelée de façon asynchrone [1].

Un futur représente le résultat d'un appel de méthode qui n'est pas encore arrivé. Pour créer de l'asynchronisme lors d'un appel de méthode, *ProActive* construit et renvoie immédiatement un

[1] Le Protocole à Meta-Objets de *ProActive* qualifie de réifiable les classes qui peuvent être sous-classés ; c'est à dire toutes les classes à l'exclusion des classes finales (et des types primitifs).

objet vide : un futur. Pendant ce temps, la requête RMI est déléguée à un autre fil d'exécution. Lorsque la requête a été traitée, le résultat obtenu est placé dans le futur. Le futur implémente la même interface que l'objet résultat.

Dans le cas où le futur est utilisé (lecture, modification, appel de méthode) alors que sa valeur, le résultat de l'appel, n'est pas encore arrivée le mécanisme de l'*attente par nécessité* intervient. De façon transparente et automatique, l'activité est suspendue jusqu'à ce que le résultat parvienne au client.

Environnement

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 [BAU 02]. 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. Ils permettent d'abstraire du code source les préoccupations de création et de recherche de nœuds. Le but est de déployer une application n'importe où sans avoir à modifier le code source. Les nœuds associés à un nœud virtuel ne sont créés qu'à l'activation de celui-ci :

```
  // Retour d'un objet Descriptor a partir du fichier XML
Descriptor pad = ProActive.getDescriptor("file://descriptor.xml");
  // Retour du noeud virtuel decrit dans le fichier XML sous
  // forme d'objet Java
VirtualNode noeudVirtuel = pad.getVirtualNode("noeudV");
  // Activation la creation des noeuds associe au noeud virtuel
noeudVirtuel.activateMapping();
  // Renvoi des noeuds crees
Node[] noeud = noeudVirtuel.getNodes();
```

ProActive est bâti sur un Protocole à Meta-Objets (MOP). Pour représenter localement un objet distant le MOP crée un couple *souche* et *mandataire* sur la machine virtuelle locale. La souche implémente la même interface de l'objet distant, elle est générée puis compilée dynamiquement. Elle réifie les appels de méthode, c'est à dire qu'elle les transforme en objet `MethodCall`. Le mandataire quant à lui est responsable de la sémantique de communication. Il est également chargé de créer l'objet futur.

ProActive propose une migration faible des objets actifs. Les objets actifs possèdent une file d'attente des requêtes à servir. Cette file d'attente est soumise à une politique, FIFO par défaut mais que le programmeur peut redéfinir à sa guise. Lors d'une migration faible, le service des requêtes est suspendu entre deux requêtes ; à ce moment là, la pile d'exécution de la machine virtuelle est vide : les données et l'activité de l'objet peuvent être déplacées sans perte d'informations. Deux solutions de localisation des objets migrant sont proposés : la chaîne de répéteurs et le serveur de localisation.

Conclusion

En conclusion, l'essence de *ProActive* est : un modèle de programmation distribuée orienté-objet qui étendu pour fournir également un modèle de programmation à composants. De plus notre modèle est orienté vers le calcul sur grille car il incorpore des mécanismes adéquats pour aider au déploiement sur tous les types de support, notamment les grilles. *ProActive* cible entre autres les applications à très large échelle.

En plus de RMI, *ProActive* permet d'utiliser d'autres protocoles de communication tels que Jini, Ibis, HTTP, ... De nouvelles fonctionnalités sont en cours de développement. Les plus remarquables sont (par ordre décroissant de maturité) : une sécurité hiérarchique basé sur le déploiement [ATT 03], la tolérance aux pannes [BAU 04], des exceptions non-fonctionnelles [CAR 03], du balancement de charge, et du calcul pair-à-pair.

2.3 Communication de groupe typé

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.

Modèle du groupe typé

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'une souche qui est d'un type compatible avec l'objet original. Le rôle de la souche est de réifier l'appel de méthode. Ensuite un mandataire applique la sémantique de communication 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.

Interface de programmation d'application

Les groupes sont créés en utilisant la méthode statique :

```
ProActiveGroup.newGroup("NomDeLaClasse", parametres[], noeuds[]);
```

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 objets actifs sont construits puis immédiatement inclus dans le groupe. Prenons le cas d'une classe standard Java :

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

Voici un exemple de la création d'un groupe et de ses membres :

```
   // Pre-construction de parametres pour la creation des membres
Object[][] parametres = { {...} , {...} , ... };
   // Noeuds sur lesquels seront crees les membres (objets actifs)
Node[] nodes = { ... , ... , ... };
   // Un groupe de type "A" et ses membres sont crees en meme temps
A ag = (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-objets 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.

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 alors de rassembler ces données au sein d'un groupe et de passer ce groupe en paramètre à un appel de méthode.

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.

Fonctionnalités avancées

En plus de l'utilisation standard des groupes (invocation de méthode et gestion de l'appartenance des membres), le mécanisme a été étendu de façon à supporter quelques fonctionnalités supplémentaires. Quatre d'entre elles semblent fondamentales dans le cas d'un mécanisme de communication de groupe dédié à la conception d'application :

- ***Un système de traitement des erreurs***. Au sein de la plateforme Java les erreurs et les pannes sont exprimées par les `Exceptions`. Dans le cadre de *ProActive*, où la distribution est transparente, il est impossible de distingués les exceptions "fonctionnelles" qui peuvent être naturellement levées par la méthode invoquée des exceptions "non-fonctionnelles" qui résultent d'une erreur inattendue du système (par exemple, la déconnexion de l'appelé). Le mécanisme des communications de groupe adresse ces deux types d'exception grâce à un dispositif capable de collecter et retransmettre les erreurs survenues lors d'une communication.
- ***Une composition hiérarchique des groupes***. Pour construire de très grande application en terme de nœud et d'objet, nous fournissons le concept de groupe hiérarchique : un groupe d'objets qui est constitué totalement ou en partie de groupes : un *groupe de groupes*. Ce mécanisme aide à l'organisation et à la distribution des données. Il assure également le

passage à l'échelle des applications. Un groupe hiérarchique est très simplement construit en ajoutant la référence d'un groupe dans un autre groupe. Bien entendu, les types de ces groupes doivent être compatibles.

- ***Un accès distant à un service de communication de groupe***. Les groupes sont des représentations locales. Il est cependant possible de vouloir y accéder de façon distante. Un groupe accessible à distance devient un service : un message est d'abord communiqué au service avant d'être réexpédié aux membres du groupes. *ProActive* fournit un moyen simple de transformer n'importe quel objet en objet accessible à distance : il le transforme en objet actif. Nous appelons *groupe actif*, un groupe transformé en objet actif. En plus de l'accès distant un groupe actif acquiert également la capacité de migrer et de voir sa politique de service FIFO modifiée.

- ***Une distribution des données dépendante de l'activité des membres***. Dans le cas particulier où les groupes sont utilisés pour créer du parallélisme, sans se soucier de savoir quel membre traître quelle donnée, nous pouvons améliorer les performances du système en ordonnançant de façon plus flexible l'envoi des requêtes vers les membres. L'idée est d'envoyer plus de données aux membres les plus rapides pour diminuer le temps total de traitement de l'appel de méthode.

Conclusion

Les communications de groupe sont un dispositif crucial pour le calcul sur grilles et le calcul à haute performance. Le système présenté ici est à la fois simple et très expressif. Il fournit un modèle transparent, robuste, flexible, et simple d'utilisation qui vise à aider la construction d'applications réparties. Au travers d'invocation de méthode, le système adapte la sémantique d'appel, et gère la collecte et la synchronisation des résultats.

2.4 Implémentation et évaluation par micro-tests

La manière dont nous avons implémenté le mécanisme des communications de groupe dépend des propriétés que nous voulions obtenir et de celles que nous voulions maintenir. Nos considérations ont éliminé certains modèles et nous ont guidé vers de possibles implémentations. Notre choix s'est porté sur une implémentation générique, sujette à optimisations. Ces optimisations ont été testées dans le but de prouver leur efficacité, et incorporées dans l'implémentation.

Détails d'implémentation

Tel que mentionné dans [MAA 03], une approche possible pour implémenter les communications de groupe typé au sein de *ProActive* aurait pu être d'étendre notre bibliothèque avec une bibliothèque externe telle que MPI. Des travaux ont été réalisé dans ce sens : [CAR 00] et [GET 99]. Cependant le modèle de passage de messages s'adapte mal au modèle orienté-objet d'invocation de méthode. De plus MPI a été conçu pour manipuler des groupes statiques de processus et non pas des objets possédant leur propre fil d'exécution.

Une seconde solution aurait été d'interfacer *ProActive* avec une bibliothèque qui interagit directement avec un protocole réseau de type un-vers-plusieurs. Par exemple [BAN 98] et [ROS 98] proposent ce genre de service. En nous immisçant à un niveau plus bas que MPI nous pouvons gagner en flexibilité. Mais encore une fois, en imposant leur propre interface d'utilisation, ces bibliothèques cassent le modèle objet qui fournit une communication par appel de méthode. De plus le déploiement de ces protocoles est rarement assuré à l'échelle d'une grille.

Finalement nous avons opté pour l'approche dite *multi un-vers-un*. Le multi un-vers-un est la réplication de communication un-vers-un. Cette approche est parfois décriée car dans sa forme la plus simple, elle est moins performante que d'autres approches. Cependant nous l'avons choisi car elle permet de maintenir la flexibilité et surtout l'adaptabilité des communications vers chacun des membres d'un groupe, et ce dans un modèle orienté-objet non altéré. De plus cette approche est ouverte à plusieurs optimisations, qui finalement rendent les performances très compétitives.

Fonctionnalités

L'utilisation de plusieurs fils d'exécution (*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.

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 [MAA 01]. Dans le cas d'une diffusion des mêmes paramètres à tous les objets (*broadcast*), ces paramètres seront sérialisés par chaque fil d'exécution. 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.

Micro-tests

Une première implémentation apportait déjà des gains de performances [BAD 02b]. 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.

Nous avons réalisé des mesures de performances sur plusieurs types de plateformes. Nos premiers tests ont été réaliser sur une grappe de 216 bi-AMD Opteron 64 bits @ 2 GHz avec 2 Go de mémoire et interconnectés par un réseau Ethernet gigabit. Ensuite nous avons déployé nos tests sur une grille de calcul. *Grid'5000* est une plateforme expérimentale qui regroupe huit sites géographiquement distribués en France et dont l'ambition est d'atteindre les 5000 processeurs. Sur chacune de ses plateformes, notre mécanisme de communication de groupe produit de bonnes performances.

Multiplication de matrices

Pour valider la conception et l'implémentation des communications de groupe nous avons programmé une application numérique basique : une multiplication en parallèle de matrices denses. Nous avons délibérément choisi un algorithme qui utilise intensivement des communications collectives. Grâce aux optimisations introduites dans le mécanisme, à l'asynchronisme des communications, et à la synchronisation automatique fournit par l'attente par nécessité, les résultats sont concluants : le code produit est simple et les performances sont bonnes.

Conclusion

L'approche que nous avons choisit pour implémenter le mécanisme des communication de groupe a été guidée par une interface élégante qui permet une utilisation transparente des groupes dans la conception d'applications réparties. L'implémentation fournit flexibilité et adaptabilité. Des optimisations telles que l'invocation de méthode en parallèle et la factorisation des opérations communes contribuent à améliorer l'efficacité du système. Les expérimentations menées aussi bien sur une grappe que sur une grille montrent les bonnes performances et l'aptitude à passer à l'échelle.

2.5 Une application test : Jem3D

Dans la tendance de réaliser des calculs distribués basés sur une programmation objet, nous présentons la conception et l'implémentation d'une simulation numérique pour la propagation d'ondes électromagnétiques. Le but de ce travail est de souligner les bénéfices qu'apporte notre bibliothèque (modèle objet, portabilité, facilité de déploiement, . . .) pour les aspects d'ingénierie logicielle.

Architecture de base de Jem3D

Jem3D est la traduction en Java de EM3D-VFC, un logiciel écrit en Fortran 77 pour la simulation numérique de propagation d'ondes électromagnétiques en temps fini. La version actuelle résout les équations tridimensionnelles de Maxwell pour des milieux homogènes et hétérogènes.

Le modèle objet que nous proposons est tel, qu'il peut être réutilisé pour le développement d'outils de simulation basés sur des méthodes à volume fini et des maillages non-structurés. L'application est pour le moment limitée aux équations de Maxwell mais peut-être étendue pour traiter les équations d'Euler ou de Navier-Stokes. Le modèle orienté objet consiste essentiellement en deux types de classes:

- les classes qui concernent la définition de la géométrie (ou domaine de calcul).
- Les classes en relations avec l'application (par exemple les composants physiques et numériques).

Ses classes sont fortement liées au contexte physique sous considération (la propagation d'ondes électromagnétiques dans notre cas).

Le squelette du résolveur est une boucle constituée de trois étapes principales:

1. L'équilibre des flux magnétiques est calculé selon la distribution des champs magnétiques à l'étape précédente. Cet équilibre des flux intervient dans le calcul du champ électrique.
2. L'équilibre des flux électriques est calculé selon la distribution des champs électriques à l'étape précédente. Cet équilibre des flux intervient dans le calcul du champ magnétique.
3. L'énergie électromagnétique discrétisée est calculée. Cette valeur scalaire est utilisée pour observer la cohérence des calculs : d'après les analyses théoriques cette valeur doit rester constante.

Création de la version parallèle et distribuée de Jem3D

Les facettes de frontière sont les facettes d'un objet localisées à la frontière de l'espace de calcul (le *domaine*). Lors de la distribution, nous avons introduit les *facettes de frontière virtuelles* et les *sous-domaines*. Une facette de frontière virtuelle (FFV) représente une facette qui est à cheval sur deux sous-domaines (les sous-domaines sont des sous-ensembles du domaine de calcul). Dans un couple de sous-domaines qui partagent des facettes, chacun possède une référence (locale) vers les facettes partagées. Les FFVs sont donc répliquées sur chaque sous-domaine, et chacune des FFVs "jumelles" participe au calcul. Ces FFVs jumelles, qui sont des copies, s'échangent et combinent leurs valeurs, et restent cohérentes. Pour la mise à jour de valeur, il est de la responsabilité du sous-domaine de communiquer les valeurs aux sous-domaines voisins qui accèderont à leurs FFVs. Les sous-domaines sont implémentés en tant qu'objet actif.

Grâce au polymorphisme et à l'association dynamique, il n'est pas nécessaire de connaître le type réel des facettes : internes ou de frontière. Les méthodes qui leur sont appliquées sont donc inchangées ; le code qui parcourt l'ensemble des facettes n'est pas modifié.

L'architecture distribuée est totalement décentralisée. L'application communique de voisin à voisin sans l'intervention d'aucun superviseur. Les points de centralisation étant souvent sujets

à congestion lorsque le système est surchargé, notre approche décentralisée assure un meilleur passage à l'échelle.

Un modèle de communication de group pour améliorer les performances

En ce qui concerne les accès en lecture, une solution naïve aurait été de laisser chaque facette invoquer de façon indépendante une méthode pour effectuer les opérations de lecture. Comme l'algorithme est implémenté selon une version séquentielle qui itère sur la liste des facettes à chaque étape, cela implique que le calcul n'aurait lieu que lorsqu'une face aurait enfin obtenu la valeur distante, ajoutant ainsi de la latence du au protocole RMI et à la couche réseau. Comme nous savons qui a besoin d'une valeur précise (la FFV jumelle), l'idée est de pousser les données plutôt que de les tirer, évitant ainsi une communication qui transmet une requête de lecture.

De manière à obtenir ce comportement, chaque sous-domaine maintient un lien vers les voisins avec lesquels il partage une facette de frontière virtuelle. L'ensemble de ces voisins est stocké dans un groupe typé. Comme nous l'avons vu, un tel groupe est directement opérable grâce à des appels de méthode : seule une invocation de méthode est propagée aux membres. Le concept du groupe typé évite aussi la programmation d'une structure de données qui aurait nécessité l'utilisation d'un itérateur pour parcourir de façon séquentielle l'ensemble des voisins. Chaque FFV doit recevoir la valeur de sa jumelle : cela est possible simplement grâce à la diffusion dans les groupes typés.

Tests

Comme Jem3D est une application scientifique, entièrement écrite avec *ProActive*, et que ses principaux modèles de communication reposent sur les communications de groupe, nous sommes très intéressés par l'évaluation des performances, et ceux sur tous types de plateformes : grappes homogènes, hétérogènes, et grilles.

Nos premières expérimentations ont eu lieu sur une grappe de 32 machines puis nous avons étendu nos tests sur 64 machines. Ensuite nous avons déployé Jem3D sur des stations de travail pour atteindre 294 processeurs. Ces machines, présentes sur le réseau de l'INRIA Sophia-Antipolis, sont très hétérogènes. Enfin nous avons utilisé *DAS-2*, une grille de calcul néerlandaise, pour tester une version de Jem3D qui utilise Ibis comme couche de communication. Ibis est une version optimisée de RMI, destiné aux calculs hautes-performances.

Conclusion

Jem3D, la version Java de EM3D, a un grand potentiel d'évolution grâce à sa réalisation qui a suivit un modèle objet. Parallèlement, la dégradation des performances entre la version Java et la version Fortran semble raisonnable. Nous observons un facteur de 3,5. C'est un bon résultat selon [FRU 03] qui montre qu'une application Java est en moyenne entre 3,3 à 12,4 fois plus lente que la même application écrite en Fortran. De plus notre version Java est encore récente et peut sans doute bénéficier de quelques optimisations. En utilisant *ProActive* et sa communication de groupe typé comme bibliothèque de conception d'application à haut niveau, la version parallèle de Jem3D fut facile à obtenir et à déployer.

2.6 SPMD orienté-object

Dans ce chapitre, nous proposons le mécanisme de communication de groupe typé comme base à un modèle de programmation appelé *SPMD*[2]*orienté-objet*. Ce modèle se veut être une alternative au modèle standard de SPM par passage de message. Étant placé dans un contexte orienté-objet, nous montrons que notre mécanisme aide à la définition et à la coordination d'activités parallèles et distribuées. Notre approche offre à travers une extension de l'interface des groupes, de la flexibilité de structuration et une implémentation novatrice. L'automatisation de mécanismes clés de communication et de synchronisation simplifie l'écriture de code pour des activités parallèles.

Contexte et état de l'art

La programmation SPMD fournit une méthodologie pour organiser un programme parallèle sur une machine parallèle, une grappe, ou plus récemment une grille. Un unique programme est écrit et chargé sur chaque nœud d'une plateforme parallèle. Chaque copie du programme s'exécute indépendamment à côté des messages de coordination. Chaque copie du programme (ou processus) est identifiée par un numéro de rang. Cet identificateur unique est utilisé dans le code pour trouver le chemin d'exécution correspondant au processus.

En tant que langage orienté-objet relativement abouti, et compte tenu de ces récentes améliorations, Java devient une base sérieuse pour réalisation d'applications scientifiques. Les travaux précédents sur la programmation SPMD traitent principalement des modèles non objets, basés sur des échanges de messages. Cependant quelques projets ont tenté d'introduire une forme orienté-objet dans le modèle SPMD, soit en maintenant le passage de messages, soit en utilisant des invocations de méthode distantes.

SPMD orienté-objet

Il est possible de reproduire le parallélisme de la programmation SPMD dans un modèle orienté-objet en se basant sur le mécanisme des communications de groupe typé et en associant une activité à chaque machine participant au calcul. Les besoins d'un modèle SPMD sont les suivants:

- L'identification de chaque membre prenant part dans le calcul parallèle et si possible une notion de position relative entre ces membres.
- L'expression du programme exécuté par chaque membre prenant part dans le calcul.
- Un ensemble complet d'opérations de communication, notamment des opérations collectives (pour la communication mais aussi la synchronisation des processus).

Un *groupe OO SPMD* (OO pour orienté-objet) est défini comme suit : c'est un groupe d'objets actifs (exclusivement) dans lequel chaque membre possède une référence vers le groupe lui-même. Chaque objet actif se voit donc doté d'un numéro de rang : celui de sa place dans le groupe. Chaque membre est capable au groupe et à son numéro de rang dans le groupe. Les groupes OO SPMD ne sont pas immutables, mais il est de la responsabilité du programmeur de s'assurer que toute modification sur le groupe maintient sa propriété.

Un membre effectue un envoi ou une réception de données au travers du service asynchrone d'une méthode appelée à distance par un autre membre du groupe. Le service est nécessairement FIFO. Traditionnellement, dans les programmes SPMD, le contrôle de l'exécution repose exclusivement des expressions `if` ou `case` basées sur le rang du processus. Dans notre approche, le contrôle de l'exécution peut aussi être basé sur des groupes créés dynamiquement.

Pour simplifier l'accès aux processus avec lesquels une activité interagit le plus souvent, nous avons introduit la notion de *voisinage*. Étant donnée une *topologie*, c'est à dire une représentation

[2]SPMD signifie *Single Program Multiple Data* (Programme Unique, Données Multiples).

géométrique de la distribution des processus, il est fréquent que certains processus, géométriquement proches, communiquent plus que des processus éloignés. Les topologies sont des groupes. La création d'une topologie à partir d'un groupe fournit un ensemble de méthodes spécifiques d'accès aux processus et la notion de voisinage. Une topologie peut être créée à partir d'un groupe (copie des références des membres) ou par extraction depuis une topologie déjà existante.

En plus des mécanismes de futur et d'attente par nécessité, notre modèle de programmation OO SPMD propose, tout comme les modèles SPMD standard, des opérations collectives en charge de la synchronisation des activités : les *barrières*. Cependant nous proposons non pas une méthode de barrière mais trois. La *barrière globale*, implique tous les processus qui suspendent leur activité jusqu'à ce que tous aient soit parvenu à l'invocation de la barrière. La *barrière de voisinage* fonctionne comme la barrière globale mais n'implique qu'un sous ensemble des processus ; cela permet de ne pas bloquer inutilement certains processus. Enfin la *barrière sur méthode* n'implique que le processus qui l'appel, il suspend son activité jusqu'à ce qu'il ait reçu les requêtes de méthodes spécifiées au moment de l'invocation de la barrière.

Exemple et tests

Nous illustrons la programmation OO SPMD avec un exemple concret. Nous avons choisi les *itérations de Jacobi* parce que c'est une application simple, facile à distribuer dans le modèle SPMD classique. La méthode de Jacobi résout des équations linéaires. L'algorithme effectue des calculs, échange des données, et se synchronise avec les autres processus ; il recommence ensuite ces étapes jusqu'à ce qu'une condition d'arrêt soit vérifiée (convergence de valeurs ou nombre fixé d'itérations).

Cet algorithme est d'abord présenté sous sa forme SPMD classique, écrite en C avec la bibliothèque MPI. Ensuite nous présentons son écriture avec notre modèle OO SPMD. Nous montrons aussi comment utiliser les différentes barrières de synchronisation sur ce cas précis. Des tests de performances effectués sur une grappe montre les temps de calcul et prouve le bon passage à l'échelle du modèle.

Comparaison avec l'interface MPI

MPI (Message Passing Interface) est sans doute la bibliothèque la plus utilisée pour la programmation SPMD. Nous ne cherchons pas à coller exactement à la syntaxe de MPI, nous souhaitons au contraire bénéficier de la syntaxe objet des communications de groupe typé. Cependant une comparaison des interfaces de programmation peut aider à la compréhension. C'est pourquoi, en faisant le parallèle entre activité de *ProActive* et processus de MPI, puis groupe typé de *ProActive* et communicateur de MPI, nous comparons les principales méthodes des deux bibliothèques.

Conclusion

Nous avons introduit un nouveau modèle de programmation parallèle que nous avons appelé SPMD orienté-objet comme alternative possible au SPMD par passage de message. Avant tout ce modèle permet une plus grande flexibilité et un meilleur niveau d'abstraction. D'abord il assure que seul l'activité émettrice d'un message spécifie la communication, ensuite il fournit une interface ouverte de topologies pour le placement et l'interaction des activités, et enfin il propose différentes sémantiques de barrière pour la synchronisation de ces activités, le tout dans un style orienté-objet.

2.7 Travaux en cours et travaux collaboratifs

Les communications de groupe typé sont la base de nouveaux travaux. Le mécanisme des communications de groupe peut encore bénéficier d'améliorations, en terme d'expressivité du langage et aussi en terme de performances sur des réseaux à grande vitesse permettant des communications un-vers-plusieurs. Le mécanisme des groupes typés est aussi utile dans la définition de composants et dans la réalisation de réseaux pour le calcul pair-à-pair.

Composant de comportement

Nous proposons de prolonger la syntaxe de la création de groupe et de changer la syntaxe et la sémantique de la gestion de groupe. À cet effet, nous présentons un comportement interne dynamique, appelé le *comportement de groupe* pour chaque groupe typé, afin de définir la sémantique adoptée par le groupe lors d'une invocation de méthode. Par la définition d'un comportement et son assignation dynamique à un groupe, celui-ci peut changer son comportement interne à l'exécution et de nouvelles politiques peuvent être facilement mises en application et ce sans interventions sur la bibliothèque ni même sur le code de l'application. Grâce à la réflexion du langage Java, un comportement de groupe nouvellement créé peut être chargé pendant l'exécution du programme ; de cette façon, un groupe peut d'une manière transparente adapter son comportement au contexte dans lequel il agit.

Pour assurer la flexibilité et l'extensibilité, la configuration et la personnalisation d'un comportement de groupe sont obtenues par un objet `GroupBehavior`. Cet objet indique le comportement d'un groupe en réponse à la demande d'invocation de méthode et est la composition de quatre sémantiques définies ci-dessous. Chaque sémantique a un état par défaut et peut être modifié dynamiquement.

- La *sémantique de traitement des requêtes* définit à quels membres du groupe s'applique un appel de méthode.
- La *sémantique de distribution* définit la façon dont les paramètres d'un appel de méthode sont partagés entre les membres du groupe.
- La *sémantique de synchronisation* définit les conditions d'attente sur les résultats d'un appel de méthode avant que l'activité de l'appelant ne continue son exécution.
- La *sémantique de collecte des résultats* définit la façon dont les résultats sont retournés à l'appelant.

Utilisation d'IP multicast

Du point de vue de la communication à l'intérieur d'un groupe, une amélioration importante peut être apportée. L'idée est de tirer avantage de transmissions de données de type un-vers-plusieurs disponibles sur les réseaux modernes. Par exemple selon certaines informations sur le réseau, il est possible d'employer, quand cela est possible, une couche transport basée un protocole un-vers-plusieurs au lieu des mécanismes communément utilisés de type un-vers-un.

ProActive est particulièrement approprié pour mettre en œuvre un tel mécanisme, grâce à sa modularité élevée et ses mécanismes de personnalisation qui associe les services de communication au niveau transport du réseau physique.

Notre solution est basée sur la définition d'un nouveau composant, le *communicateur*, dont la tâche principale est de contrôler la transmission de données à l'intérieur d'un groupe pour chaque appel de méthode. Un tel composant est le seul composant à se rendre compte des services de communication fourni par les réseaux physiques et à pouvoir ainsi associer la sémantique de communication à la couche transport disponible, qui est la plus appropriée.

Composants

Le calcul de grilles et les réseaux pair-à-pair sont par définition hétérogènes et distribués, et pour cette raison ils conduisent à nouveaux défis technologiques : complexité dans la conception des applications, complexité du déploiement, complexité de la réutilisation du code et complexité de l'exécution. *ProActive* fournit une réponse à ces préoccupations par l'implémentation d'un modèle à composants extensible, dynamique, et hiérarchique appelé *Fractal*.

Fractal définit un modèle général de composants, avec une interface de programmation d'application en Java. Selon la documentation officielle, le modèle à composants de Fractal est un modèle modulaire et extensible qui peut être employé avec de divers langages de programmation pour concevoir, implémenter, déployer et modifier divers systèmes et applications, depuis des logiciels d'exploitation jusqu'aux plateformes de logiciel personnalisé et aux interfaces graphiques. Le modèle Fractal est basé sur les concepts de l'encapsulation, de la composition, du partage, du cycle de vie, des activités, et de la dynamicité. Un composant Fractal est formé de trois parties:

- un *contenu*, qui peut être récursif (composant composite).
- un *ensemble de contrôleurs* qui fournissent les propriétés nécessaires d'introspection.
- un *ensemble d'interfaces* par lesquelles le composant interagit avec d'autres composants.

L'implémentation de Fractal avec *ProActive* étend le modèle sur deux points. D'abord un composant peut être distribué sur plusieurs machines virtuelles. Ensuite nous définissons les *composants parallèles* qui encapsulent d'autres composants d'un même type et vers lesquels des appels de méthode sont envoyés simultanément. Ces composants définissent une *interface collective*. Bien entendu le mécanisme des groupes typés est un dispositif clé ces composants parallèles.

Calcul pair-à-pair

Le calcul pair-à-pair émerge comme un nouvel environnement d'exécution. Le potentiel de centaines de milliers de nœuds reliés ensemble pour exécuter une application est très attrayant, particulièrement pour le calcul de grille. Imitant le pair-à-pair de données, il serait possible de commencer un calcul qu'aucune panne ne pourrait arrêter. *ProActive* fournit une interface de programmation d'application pour le calcul pair-à-pair visant principalement à utiliser les cycles d'unité centrale disponibles de machines d'un réseau d'entreprise, éventuellement combiné aux machines d'une grappe ou d'une grille. Le but est de déployer des applications sur un ensemble décentralisé de nœuds et d'employer la plupart des ressources disponibles sur un réseau.

L'infrastructure de pair-à-pair fonctionne comme réseau de recouvrement. Elle se compose de services "pair-à-pair" (les pairs) qui deviennent à leur tour des nœuds de calcul. Un objet actif, appelé `P2PService` et déployé sur une machine virtuelle, met en application le service.

Dans cette architecture décentralisée, chaque pair possède une part de responsabilité vis à vis de la propagation de messages et du maintient de la connectivité. Les messages fonctionnels qui transitent sont essentiellement des demandes de ressources, c'est à dire des demandes de nœuds. Si un service pair-à-pair n'est pas capable de répondre à la demande de l'utilisateur, il transmet cette demande à ses voisins. Cette communication entre pairs est assurée par une communication de groupe typé. De même pour éviter un partitionnement du réseau pair-à-pair, les services émettent régulièrement un *battement de cœur*. Ce battement de cœur est un message transmit à tous les voisins du pair de façon à détecter les pannes et si le nombre de voisins "vivants" est jugé insuffisant un message de recherche de nouveaux nœuds est propagée. Ces messages sont également diffusés dans le réseau grâce aux communications de groupe typé.

Conclusion

3.1 Accomplissements

Java possède beaucoup d'avantages pour le calcul sur grilles. Avant tout, étant basé sur le concept de machine virtuelle, il est naturellement plus portable que les langages traditionnels, statiquement compilés. Cela rend l'exécution d'applications Java plus aisé sur des environnements de grilles, qui sont par nature hétérogènes. Aussi, Java est basé sur un modèle de programmation haut-niveau et fortement typé qui supporte la concurrence et la distribution des processus.

L'objet actif est l'unité de base de *ProActive* pour exprimer une activité et la distribution et ainsi construire des applications concurrentes. Un objet actif est créé à distance sur un noeud. Les appels de méthode sont envoyés aux objets actifs de façon asynchrone, avec création transparente d'un objet futur et synchronisation par attente-par-nécessité.

En plus des simples objets actifs, *ProActive* offre maintenant un mécanisme de communication de groupe qui permet l'invocation d'une méthode sur un ensemble d'objets, regroupés et référencés par un unique nom collectif. Un groupe de *ProActive* est aussi appelé groupe typé puisqu'il est composé d'objets appartenant à des classes qui hérite d'une même superclasse ou implémente une même interface. Un groupe typé est une "réplication" d'objets sur un ensemble de noeud, une communication de groupe et une "réplication" d'un appel de méthode sur ces objets. Chaque objet membre peut être une instance d'une classe différente mais tous doivent avoir une classe ou une interface ancêtre commun.

Alors que de nombreuses bibliothèques et plateformes de programmation qui fournissent des communications de groupe imposent des contraintes spécifiques aux programmeurs. Grâce à son Protocole à Meta-Objets, *ProActive* fournit un mécanisme plus flexible et transparent. Au travers de la réification des appels de méthode et de constructeur, le MOP rend possible d'initier une communication de groupe par l'invocation d'une méthode. En conséquence, utiliser un groupe typé prend exactement la même forme que l'utilisation d'un simple et unique objet. Quand un appel de méthode est invoqué sur un groupe, la sémantique de communication est implémentée au dessus d'un système de communication asynchrone qui traite de façon interne la construction et l'envoi de requêtes, l'ordonnancement des évènements de transmission de requêtes, notification d'erreurs, collecte des résultats,... Un tel système propage efficacement et de façon asynchrone les appels de méthode à tous les membres du groupe en utilisant plusieurs fils d'exécution. Un appel de méthode sur un groupe est asynchrone est produit un objet futur transparent qui collecte les résultats.

Actuellement, les groupes de *ProActive* fournissent au programmeur des outils pour la gestion de la distribution des paramètres d'entrée, tels que la diffusion ou la distribution. En choisissant la diffusion, les mêmes paramètres sont envoyés à tous les membres. Dans le cas de la distribution, une partie différente de l'ensemble des paramètres est envoyée vers chaque membre du group. Dans ce cas, les paramètres à distribuer doivent être explicitement passés sous forme de groupe, dont chaque membre est une fraction du paramètre. Le comportement par défaut est la diffusion, pour le changer le programmeur doit invoquer la méthode statique `setScatterGroup` de la classe `ProActiveGroup` sur le paramètre. Ainsi la sémantique de partage des paramètres repose uniquement sur les paramètres d'appel.

Le résultat d'une communication de groupe est également un group. Ce résultat est dynamiquement mis à jour avec les retours de résultats. Grâce à la synchronisation implicite du mécanisme d'attente-par-nécessité, ce résultat est immédiatement opérable : il peut être utilisé pour exécuter un appel de méthode même si tous les résultats qui le composent ne sont pas encore disponibles.

Nous avons introduit un modèle de programmation parallèle que nous appelons SPMD orienté-objet et que nous proposons comme une alternative au traditionnel SPMD par passage de messages. L'API résultante est déjà pleinement intégrée dans l'intergiciel *ProActive*, membre du consortium Object Web. Notre ambition est d'utiliser cette nouvelle approche dans des applications à taille réelle. Nous avons déjà réussi à appliquer avec succès le modèle des communications de groupe typé pour réaliser des simulations d'électromagnétisme. Notre travail actuel est d'appliquer l'ensemble de l'approche OO SPMD. Ensuite nous prévoyons de viser d'autres domaines d'application telles que la génétique (appliquer BLAST en parallèle) pour laquelle nous avons déjà développé une application mais pas encore sur le modèle OO SPMD.

3.2 Perspectives

Les groupes de *ProActive* définissent un modèle complet pour la programmation par groupe typé. Une implémentation a été réalisée, évaluée, et utilisée pour construire des applications. Le modèle de programmation OO SPMD propose une approche plus flexible de la programmation SPMD. Il permet une meilleure flexibilité pour la synchronisation des activités par barrières et supprime les boucles explicites. Il devient également possible de privilégier la réactivité et la réutilisation de code.

Cependant plusieurs problèmes restent ouverts. Les principaux points pour des travaux futurs sont listés ici:

- Un ***dimensionnement "intelligent" du réservoir à fil d'exécution***. Définir une solution générique qui permet d'allouer de façon optimale les ressources pour effectuer une communication de groupe n'est pas chose facile. De nombreux paramètres entre en compte : le nombre de membres dans le group bien sûr, mais aussi la fréquence de communication, la taille des données échangées, la charge du système, la bande passante du réseau,... Nous avons expérimenté dans plusieurs applications que le meilleur mécanisme de dimensionnement est souvent obtenu après observations et modifications de la part du programmeur. C'est pour cela que notre choix final est de laisser au programmeur la possibilité de redéfinir sa propre méthode qui dimensionne le réservoir selon ses conditions. Il serait cependant intéressant de regarder plus en profondeur de façon à voir si quelques modèles émergent pour répondre plus efficacement et plus généralement à cette préoccupation.

- Une ***étude sur l'ordonnancement de livraison***. Les communications de groupe typé fournissent un ordonnancement FIFO : étant donné une source, les messages sont reçus dans l'ordre dans lequel ils ont été émis. Cette sémantique suffit généralement pour la conception d'applications distribuées. Elle fournit de bonnes performances et une sémantique plus forte peut être ajoutée par le programmeur. Avec les groupes actifs, l'ordonnancement est total : les messages sont reçus dans le même ordre par tous les membres du groupes. Cela garantit par le fait qu'un groupe actif n'expose qu'un unique point d'entrée qui relaie les appels. Une étude précise sur l'impact de l'une ou l'autre de ces sémantiques sur l'écriture de code et les performances à l'exécution serait intéressante. Nous pouvons aussi considérer l'introduction de l'ordonnancement causal.

- Des ***mesures poussées sur l'utilisation d'IP multicast***. L'implémentation actuelle qui lie *ProActive* à une librairie d'IP multicast n'est qu'un prototype. Nous devrons produire une version finale qui s'intègrerait dans la distribution standard de *ProActive*. Nous pourrons aller plus loin dans l'analyse des performances avec cette version des groupes typés

sur IP multicast. Des mesures seront effectués sur grappes et grilles, avec des simples tests basiques et une applications numérique (probablement Jem3D). Il serait aussi très intéressant d'observer le comportement du système avec un schéma de communication mixte : par exemple, dans un environnement de grille où les communications inter-cluster sont assurées par des communications standards de *ProActive* (pour passer les pare-feux) alors que les communications intra-cluster sont assurées par IP multicast (pour bénéficier des capacités de réseaux très rapides).

- Une ***redistribution NxM***. Au sujet des composants, nous aimerions automatiser l'envoi des paramètres d'un appel de méthode, et symétriquement, la collecte des résultats, dans le cas d'une redistribution de M vers N. Dans notre implémentation actuelle du modèle Fractal ce problème n'est pas encore traité. Les communications de groupe typé assument que l'unité de base de la transmission de données est l'objet. Une solution serait de demander au programmeur d'implémenter une méthode de redistribution de ses données de n'importe quel nombre M d'objets vers n'importe quel nombre N d'objets, et d'utiliser cette méthode dons notre modèle à composant. Cependant nous cherchons toujours une solution plus autonome et transparente. Les communications de groupe typé avec des groupes de futurs ouvrent la voie à de nombreuses perspectives dans ce domaine.

- Une ***évaluation plus précise du modèle OO SPMD***. Notre modèle de programmation OO SPMD a été évalué avec un programme simple s'exécutant sur une grappe. Nous souhaitons tester une application à taille réelle : Jem3D semble encore le candidat idéale puisque son modèle correspond au modèle SPMD (l'algorithme est basé sur des itérations et chaque sous-domaine effectue la même tâche). Les performances de cette application mesurées sur une grille pourront nous apprendre beaucoup sur la validité de notre approche et son comportement sur des systèmes à très large échelle.

Part II

Thesis

Chapter 1

Introduction and objectives

1.1 Context

Many Grid applications such as simulations applied to scientific and engineering fields, or data acquisition and analysis from distributed measurement instrumentations and sensors deal with intensive computations and management of huge amount of data which have to be transferred and processed on multiple resources in order to improve the performance.

In recent years, many Grid middleware platforms and toolkits have been developed (Globus [GLO], Legion [NAT 02], Unicore [UNI], Condor-G [FRE 01], HiMM [SAN 03], etc.). These middleware platforms, typically adopt unicast communication mechanisms implemented atop reliable protocols. However, Grid systems could strongly benefit in many applications from a one-to-many or many-to-many communication mechanisms [JEA 03, MAI 02].

Providing a middleware for Grid computing with an effective and efficient implementation of the group abstraction at programming level could ease software development and reduce the communication overhead both in a small scale and in a large scale. Performances of programs on distributed memory parallel machines are highly dependent of the efficiency of interprocess communications. Parallel programming environments often offer poor support for high level communication models. This thesis deals with high level group communications in such architectures.

According to the Object Group design pattern [MAF 96], a group is a local surrogate for a set of objects distributed across networked machines to which can be assigned the execution of a task. The object group pattern specifies that when a method is invoked on a group, the runtime system sends the method invocation request to the group members, waits for one or more member-replies on the basis of a policy, and returns the result back to the client. Groups are usually dynamic, i.e. the set of group members can continuously change.

1.2 Needs

For few years, the interest in using Java for high-performance computing has increased. Java provides an object-oriented programming model with support for concurrency, garbage collection, and security. It features multithreading and *Remote Method Invocation* (RMI) [SUN98] (an object-oriented version of *Remote Procedure Call* (RPC) [BIR 84]).

Programming high-performance applications requires the definition and the coordination of parallel activities. Hence, a library for parallel programming should provide not only point-to-point but collective communication primitives on groups of activities.

In the Java world, the RMI mechanism is the standard point-to-point communication mechanism, and is adequate mainly for client-server interactions, via synchronous remote method calls. In a high-performance computing context, asynchronous and collective communications should be accessible to programmers, so the usage of RMI is not sufficient.

We have developed *ProActive* [PRO], a 100% Java library, for parallel, distributed, concurrent computing with security and mobility. RMI is by default used as the transport layer. Besides remote method invocation services, *ProActive* features transparent remote active objects, asynchronous two-way communications with transparent futures, high-level synchronization mechanisms, migration of active objects with pending calls and an automatic localization mechanism to maintain connectivity for both "requests" and "replies".

At programming level, groups can ease software development since they simplify the implementation of some high-level computing models, such as master-slave, pipeline, and work-stealing.

At communication level, groups can reduce the communication overhead for several reasons. First, the delivery of the same content to a collection of receivers can benefit from the group abstraction since specific optimizations can be applied even if the underlying transport layer is based on unicast communication. For instance, the network transfer of objects requires serialization before sending them. Since serialization takes a significant processing time, sending the same object to the members of the group is easily improved if the same serialized copy of the object is used for a unicast transfer towards each member. Second, group communication can be implemented through its mapping on a multicast transport layer.

1.3 Thesis organization

The goal of our research is to design an efficient and elegant communication mechanism dedicated to grid computing. It must provide a model that integrates cleanly into the object-oriented model of Java. The focus of this work is to extend the RMI mechanism to support a more expressive form that allows multi-point communications. This should improve performances while reducing the programming complexity of distributed applications. More generally, we aim at providing a model that helps the definition and the coordination of distributed activities.

The document is organized as follow:

- Chapter 2 gives an overview of the related works on group communication mechanisms in various domains. It identifies the main features being required depending on the domain of application. This chapter includes a state of the art presenting the most noticeable systems for group communication. A discussion about required features concludes the chapter.
- In Chapter 3, I present the *ProActive* middleware. I introduce the programming model, and then give a description of its features for parallel and distributed computing. This presentation describes some elements of the API.
- Chapter 4 introduces an object-oriented model for a high-level group communication mechanism. I describe the typed approach with the presentation of the group API added in the *ProActive* middleware. Thus, I introduce the features of this typed mechanism.
- Chapter 5 gives details of the implementation. It presents the main points of optimization introduced in the system. Then it evaluates performances with basic benchmarks on both clusters and grid platforms.
- In Chapter 6, I present a numeric application, named Jem3D, which implementation is heavily based on the group communication mechanism for intensive communications and process synchronization. Performances and scalability of Jem3D, obtained on several execution platforms, are presented.

- Chapter 7 introduces the object-oriented SPMD programming model. This chapter describes the background concepts of SPMD programming, and then presents our object-oriented approach. Codes and performances are analyzed using a typical program: the Jacobi iterations.
- In Chapter 8, I present features related to the group communication mechanism that are currently in development. They are the behavior component, the use of IP multicasting, the Fractal's components implementation, and the peer-to-peer computing.
- Finally, Chapter 9 summarizes the major achievements of this thesis, and presents perspectives.

Chapter 2

Related work

By opposition to point-to-point communication, multipoint communication involves two or more processes. Such kind of communication is used in a wide growing range of application, especially diffusion (*broadcast* and *multicast*) that consists in sending the same replicated data to many receivers. Many distributed applications require delivery of messages and membership services for a group of processes.

A group is a set of *objects* that are addressed as a single entity. Those *objects* can be, not exclusively, processes, activities, hosts or objects (as in object-oriented languages). Throughout this document, each term may be used interchangeably.

This chapter is organized as follows: in a first time Section 2.1 introduces the main properties of group communication. In the most general way, it defines the points on which all kinds of group communications have agreed upon. For each of those main topics, I present the usual answers supplied by the large communities working around communications systems. Then Section 2.2 gives an overview of the existing libraries and tools. This state of the art presents the most significant projects related to the wide scope of group communication. It details specificities of each project and shows their main interests depending on their targeted area of applications.

2.1 Group properties

[MAN 98] exposes the diversity of requirement in group communication. Some applications expect a total ordering for the message delivery, while others might not need it. In some applications, every process is able to send messages, while in others only few ones are authorized. In some application data is only in one place, while in others data is replicated in many places. In some applications groups contain few members and are static, while in others, groups contain large amounts of members (about 100,000) and are dynamics. Some applications are not aware about transmission reliability while others are.

We have identify four communities that study group communications; each one for its own purposes that may differ or be quite similar from the others. Firstly, the Internet community focuses on network aspects and protocol. Secondly, the operating system community is interested in distributed operating system. Thirdly, the distributed algorithm community is involved in fault-tolerant application and protocol design. Finally, the parallelism community is interested in execution platform for parallel applications.

2.1.1 Structure

Depending on their structures, groups can support a wide range of applications. One has to choose the group structure that best suits the application needs. To all programming needs or user applications match a group structure. Four group structures are proposed to provide the most appropriate policy (as presented in [BIR 91]):

1. A ***peer group*** is formed by a set of member cooperating for a particular purpose. Typically, the applications using peer groups are fault-tolerant or load sharing applications (see Figure 2.1 (a)). Peer group does not scale very well.
2. A ***client-server group*** is composed of a possibly large amount of clients and a peer group of servers. Messages are sent to the members of the group by a non member client (see Figure 2.1 (b)).
3. A ***diffusion group*** is a client-server group where a single message is sent by one of the server to the set of clients and servers. Such groups benefit from a multicast network (see Figure 2.1 (c)).
4. A ***hierarchical group*** is an extension of a client-server group. In applications distributed on a large number of computers, it is important to localize the interactions between members in order to reduce the number of exchanged messages to improve performance (see Figure 2.1 (d)). The *root* group becomes a centralized point whose failure is critical.

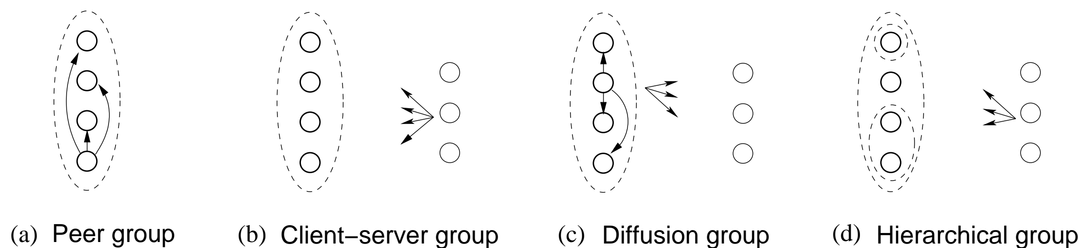


Figure 2.1: Group structures

A group is said *egalitarian* if all members of the group have the same activity and no one is in charge of extra services. For instance a hierarchical group is not egalitarian: some members that are groups relay the message. Another example of non-egalitarian group is the use of one process in the group (often the process of rank 0) to coordinate and control ordering or membership.

The management of a group is *distributed* if every member takes (a same) part to the management of the group. The management is *centralized* if only one member is in charge of it. Finally, the management is *hierarchical* if the management is performed by some members, each one in charge of a subset of members.

According to [TAN 90, TAN 94], groups can be classified in two categories: *closed groups* and *open groups*.

- In a ***closed group***, only members of the group can send and receive messages. Consequently, all communicating processes belong to the closed group. The closed group semantics implemented by the application layer is typical of a parallel processing application where a group of processes work together to formulate a result which does not require the interaction of members outside the group. Indeed, this group communication style is often implemented in a peer group or diffusion group programming style.
- In an ***open group*** not only the members of the group can send and receive messages; the non-members do not need to join the group to communicate with the members. An open group semantic best suits replicated server applications where non-members can send messages to the group. Usually, a client server group or a hierarchical group is the best structure for an open group.

2.1.2 Reliability and semantics

A communication is *reliable* if, without hardware failure, the sent messages are received by the receiver(s) and the data integrity is kept. Reliability is much more difficult to maintain for group

communication than for point-to-point communication. For instance, the System V [CHE 85] and Chorus [ROZ 92] only provide unreliable group communication, pretending to reduce the overheads in group communication, and consequently provide acceptable performances. However the need for reliable message delivery becomes crucial in case of requirement of consistent service or data.

If an application using unreliable group communication needs reliability, the reliable mechanism has to be performed at the user level. In that case, the overhead appears at the application layer and writing a reliable program becomes harder. That is why most group communication protocols provide reliable group communication at the transport level or at the interprocess communication level.

Communication networks are fundamentally unreliable. Packets can be lost through the network. So the group communication protocol or system must provide the basic mechanisms for reliability to ensure the exchange of messages from a sender to the group members.

Delivery semantic

The delivery semantic defines if a group communication succeeds to deliver a message to the group. Usually, there are five possibilities for delivery semantic:

1. With the ***zero delivery*** semantic the group communication is considered to be successful even if no message reaches a member of the group. Actually, in zero delivery semantic all deliveries are successful.
2. In ***single delivery*** semantic, only one member of the group needs to receive the message for the group communication to be considered successful.
3. The ***n-delivery*** semantic requires that at least n members of the group receive the message for the group communication to be successful. n-delivery may be a zero delivery if n=0 or a single delivery if n=1.
4. The ***quorum delivery*** semantic requires that a majority of the members receive the message for the group communication to be successful. The number of messages required to achieve the quorum may vary with the size of the group.
5. The ***atomic delivery*** semantic requires that all or none of the members receive the message for the group communication to be successful. The atomic delivery is the strongest delivery semantic.

Response semantic

As for the message delivery semantics, group communication has to provide a range of semantics for the responses. The message response semantic defines the number and type of awaited responses to consider succeeded the response of the group communication. Symmetrically to the delivery semantics, there are five response semantics:

1. With the ***zero response*** semantic the group communication is considered to be successful even if no response returns to the sender. Zero response semantic provides unreliable group communication.
2. In ***single response*** semantic, only one response from one member of the group is needed to consider the response successful.
3. The ***n-response*** semantic requires that at least n responses from members of the group return to the sender. The zero response semantic is an n-response semantic where n=0, the single response semantic is an n-response semantic where n=1.

4. The ***quorum response*** semantic requires that a majority of response is received by the sender of the message for the group communication to be successful. The number of responses required to achieve the quorum may vary with the size of the group.

5. The ***total response*** semantic requires that a response from all members of the group return to the sender.

2.1.3 Dynamicity

A group is *dynamic* if members can join and leave at any time. By opposite, groups that do not allow join and leave operations at any time are *static*. Dynamic groups can be classified in two categories: the *dynamic transient groups* and the *dynamic persistent groups*. A transient group disappears when the last member has left. As opposed to a transient group, a persistent group survives to the disappearance of its last member. Even empty the group continues to exist.

Dynamicity may be the source of lost of consistency between copies of local surrogates of a same group. Modifications on a surrogate may be not (or not in time) reflected on other surrogates, thus introducing inconsistency between them. Many group toolkits using local representation for their groups choose not providing dynamicity to avoid this problem.

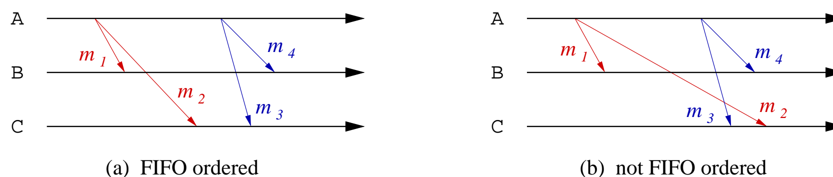
2.1.4 Ordering

The ordering of message delivery in group communication systems was an open discussion. [CHE 94] argued that group communication should not provide any ordering of message delivery, leaving the ordering issue to the programmer at the application layer. In opposition, [BIR 93a, BIR 94] and [COO 94] defended the opinion that a group communication should provide the full set of message delivery semantics presented below.

Ordering semantics are classified in four categories:

1. First, the ***no ordering*** semantic implies that all messages will be sent to the target group in no specific order.

2. The ***FIFO ordering (by source)*** semantic ensures that messages are received in the order they were sent by the source (First In First Out), for example see Figure 2.2 (a). Note that FIFO ordering messages are all referenced to the sender (i.e. the source), which is the process `A` in the figure.



(a) FIFO ordered (b) not FIFO ordered

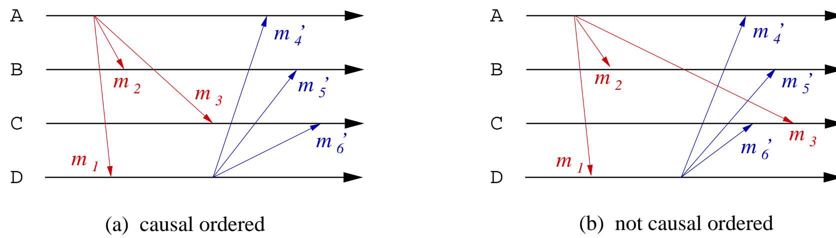
Figure 2.2: FIFO message ordering

In a group communication with multiple receivers, if FIFO delivery of point-to-point is not guaranteed, it may be possible that messages m_1 and m_3 may be delivered in FIFO order while messages m_2 and m_4 may not, breaking the FIFO ordering; see Figure 2.2 (b). FIFO ordering requires FIFO point-to-point communications, as a necessary condition.

3. The ***Causal ordering*** semantic implies that messages are received in the same order as they were sent (FIFO ordering) and, if the diffusion of a message *m* was initiated and this message leads to the sending of a message *m'* by one of its receivers, then all the messages *m* must be received before the messages *m'* by all the receivers of both messages.

Causal ordering has been inspired by the Lamport's definition of the relation *"happen before"* of events in distributed systems [LAM 78]:
"The relation $\rightarrow$ on the set of event of a system is the smallest relation satisfying the following three conditions: *(1)* if a and b are events in the same process, and a comes before b, then $a \rightarrow b$. *(2)* if a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$. *(3)* If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$."

Figure 2.3 (a) presents a causal ordered message delivery. We write $m_1 \rightarrow m_{4|5|6}$' the fact that $m_1 \rightarrow m_4$', $m_1 \rightarrow m_5$', and $m_1 \rightarrow m_6$'. The group receiving both *m* and *m'* messages is composed of B and C, we will note it {B | C}. Figure 2.3 (a) exposes {B | C} managing a causal ordered message delivery: all the *m'* messages induced by m_1, one of the *m* messages, are received after the *m* messages on B and C. In Figure 2.3 (b) the causal ordering is broken by the process C that receives the m_6' message before the m_3 message while $m_1 \rightarrow m_{4|5|6}$'.

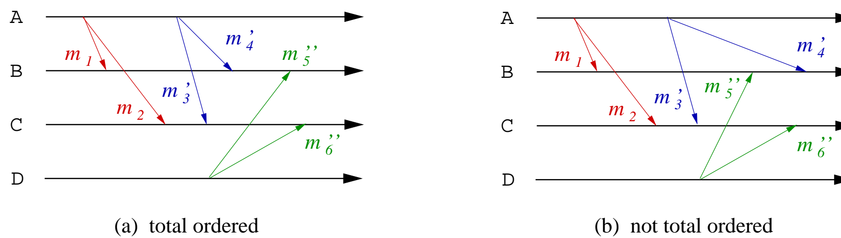


(a) causal ordered (b) not causal ordered

Figure 2.3: Causal message ordering

4. The ***Total ordering*** semantic supposes that all messages are reliably delivered in the same sequence to all members of the group. It guarantees that all members receive the messages in the same order. Causal ordering takes care of the relationship of messages while total ordering takes care of the same order of messages delivery for all members of a group.

   In Figure 2.4, the considered group is again {B | C}. The left picture (a) presents a total ordered message delivery, all group members receives the messages in the same order: first the *m* message, then the *m'* message, then the *m"* message. In the right picture (b) the total ordering is broken: process C receives *m*, *m'*, *m"* while process B receives *m*, *m"*, *m'*.



(a) total ordered (b) not total ordered

Figure 2.4: Total message ordering

The semantics of message ordering is an important factor in providing good application layer performance and a reduction in the complexity of distributed application programming. The order of message delivery to members of a group dictates the type of group application it is able to support.

2.1.5 User interface

Finally, there are different ways the programmer may trigger group communications in source code. Two approaches emerges. In the *communication by message* approach, the programmer builds a message containing the data he or she wants to diffuse, and then invoke a communication primitive that effectively send this message. In the *typed communication* approach, a group communication is achieved through a remote procedure call. We said "*typed*" because remotely accessible procedures compose an interface (a type) and return a typed result. Such communication automatically creates the message, sends it, and may receive a result (i.e. an other message).

2.2 Group toolkits

In this section some of the most significant group communications systems are presented. They are designed either for Internet, for distributed operating system, for UNIX operating system (as toolkit), for fault-tolerant applications, and for distributed application environments. The most distant is presented first, leading to the projects which are the closest to our research interest. The semantics they provide is emphasized and summed up in tables.

2.2.1 Internet multicast

Internet was asking for group communication. The need for group communication in the Internet world differs from need of group communication in the applicative world. IP Multicast, presented first, is the base of most of other protocols. It provides a basic broadcast communication scheme. Then, MTP, XTP, RAMP and RMTP will be briefly presented. Those protocols represent the set of main solutions.

IP Multicast

IP multicasting is the transmission of an IP datagram to a *host group*: a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same *best-efforts* reliability as regular unicast IP datagram. There is no guarantee that the datagram reaches all the destinations or it arrives in the same order.

The group membership is dynamic. Thanks to the *Internet Group Management Protocol* (IGMP) [DEE 89], there is no limitation on the location or number of members in a group and a host may be a member of more than one group. A group may be *permanent* (i.e. persistent); it keeps the same address independently of the membership, even if there is no more member. Otherwise IP Multicast allows also groups to be *transient*; a group keeps its address until it becomes empty.

IP Multicast does not provide reliability or ordering of communications. However it offers the basic mechanisms to build high-level protocols. IP Multicast is now integrated in IPv6 standards [HUI 96].

Membership	Dynamic
Structure	Open group
Ordering	No-ordered
Reliability	Not reliable
User interface	Communication by message

Table 2.1: IP Multicast properties

Multicast Transport Protocol

Multicast Transport Protocol (MTP), presented in [ARM 92] is a protocol build at the transport layer. It provides reliable and efficient communications over protocols build at the network layer (like IP Multicast). MTP creates groups named *webs*. Members of a web can be consumer (only), consumer and producer, or master (a master is also consumer and producer). The master controls the communications in a web. There is only one master by web.

The master initializes the web. Then members join the group specifying if they are just consumer or consumer and producer. The request for registering also contains information about the quality of the transmission: reliable or best-effort, one-to-all or all-to-all, the minimal throughput requested and the maximum size of a datagram.

The master schedules the communications and controls the throughput with a token mechanism. By giving the token to a producer, the master allows it to send some packets. Delivery failures are detected using *Negative Acknowledgments* (NAck). Only the lost packets are retransmitted. Consequently, each producer has to store the transmitted data during a sufficient amount of time.

Despite its qualities, the centralized approach of MTP introduces a bottleneck. It damages the scalability of the protocol. Moreover, the waiting for the token introduces a delay in all communications.

Membership	Dynamic
Structure	Closed group
Ordering	FIFO ordered
Reliability	Reliable or not reliable
User interface	Communication by message

Table 2.2: MTP properties

Xpress Transfer Protocol

Xpress Transfer Protocol (XTP) is a transport layer protocol [FOR 95]. It was designed for a large scope of applications: from distributed systems to real-time systems. XTP can be used over the network layer (IP) or directly over the data link layer (*Media Access Control* (MAC) and *Logical Link Control* (LLC)) or over the *ATM Adaptation Layer* (AAL).

XTP provides the same communication features as TCP, UDP, and IP Multicast, and reliable and FIFO ordered communications. XTP provides, independently, error detection on control (with positive acknowledgments), retransmission of lost packet, flow control, acknowledgment management, priority management, throughput control, and traffic description.

XTP groups are called *multicast groups*. A multicast group is composed of a single sender and many receivers. To build many-to-many communications, XTP requires the combination of multicast groups (as many as senders). Groups are dynamics, and membership is accessible at the application level.

XTP is a very complete protocol. However it is not very scalable: all receivers are connected with the sender. Furthermore, the combination of several one-to-many communications to build a many-to-many communication requires the opening of a lot of XTP connections and makes all the point-to-point communications concurrent.

Membership	Dynamic
Structure	Closed group
Ordering	FIFO ordered
Reliability	Reliable or not reliable
User interface	Communication by message

Table 2.3: XTP properties

Reliable Adaptive Multicast Protocol

Reliable Adaptive Multicast Protocol (RAMP) is a transport layer protocol [BRA 93]. It was initially built to send huge sized pictures to numerous users. It comes with the *Multicast Group Authority* (MGA), a service in charge of the group ID allocation and group management. RAMP

plans various qualities of service depending on receivers. To achieve this, RAMP uses two unreliable modes. The first one is best-effort; the second one ensures reliability only with the members asking for it.

RAMP allows the senders to use one among two modes. In the *Burst Mode*, sequences of packets are sent in a short time step. The receivers have to send a positive acknowledgment for each burst. In the *Idle Mode*, the senders never stop to send packet, even if there is no useful data (empty packets are sent). The burst mode is more suitable with low-speed network because the amount of data sent is minimized. The idle mode is more suitable when the receivers are numerous.

RAMP proposes dynamic groups and a differentiated quality of service for a unique data flow. It is well appropriate to reliable and heterogeneous networks which are the typical networks nowadays. As XTP, RAMP lacks of scalability due to the connection between senders and receivers.

Membership	Dynamic
Structure	Open group
Ordering	FIFO ordered
Reliability	Reliable or not reliable (on demand for each receiver)
User interface	Communication by message

Table 2.4: RAMP properties

Reliable Multicast Transport Protocol

Reliable Multicast Transport Protocol (RMTP) [LIN 96] is a reliable multicast transport protocol for the Internet developed by Lucent Technologies. Do not confuse with the RMTP technology developed by the IBM Tokyo Research Laboratory and the NTT Information and Communication Systems Laboratory.

RMTP provides ordered and reliable data stream from one sender to a group of receivers. RMTP is implemented using a multi-level hierarchical approach, in which the receivers are grouped into a hierarchy of local regions, with a *Designated Receiver* (DR) in each local region [PAU 97]. The local regions can be mapped on the underlying network. Receivers in each local region periodically send acknowledgments to their corresponding DR, DRs send acknowledgments to the higher-level DRs, until the DRs in the highest level send acknowledgments to the sender, thereby avoiding the acknowledgment implosion problem (see Figure 2.5). DRs cache received data and respond to retransmission requests of the receivers in their corresponding local regions, thereby decreasing end-to-end latency.

Most recent version of RMTP includes support for “asynchronous streaming” meaning that RMTP enables reliable multicast of a continuous stream of very small messages (less than 100 bytes) with rigorous end-to-end latency requirements per message.

Membership	Dynamic (but static for the DRs)
Structure	Open group
Ordering	FIFO ordered
Reliability	Reliable
User interface	Communication by message

Table 2.5: RMTP properties

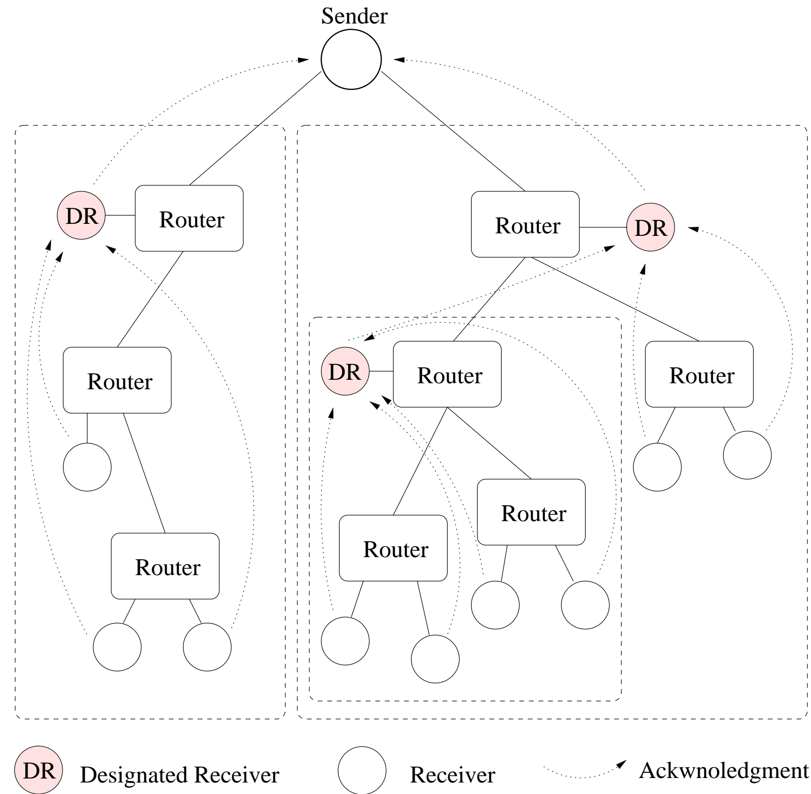


Figure 2.5: RMTP Network Architecture

2.2.2 Isis and Horus

Isis and Horus are both projects developed at the Cornell University to build distributed applications, possibly fault-tolerant.

Isis

Isis started by researches in fault tolerance in distributed systems [BIR 85]. It supports fault-tolerant distributed computing by automatically replicating data and code. The system implements a set of techniques for building software for distributed systems, Isis claims to exploit parallelism and to be robust against both software and hardware crashes. Isis provides a toolkit mechanism for distributed programming. The tools allow connecting simple non-distributed programs in order to obtain a distributed system. Tools are included for automating recovery, synchronizing distributed computations, managing replicated data and dynamically reconfiguring a system to accommodate changing workloads. [TAM 98] presents a singular example of such composition: a team of robots playing soccer. Individual agents collaborate to achieve a common goal.

The Isis programming model is based on *virtually synchronous* processes. During a virtual synchronization, all the members of a group receive an ordered and consistent flow of events. The synchronization is said virtual because events are synchronous in regards of logical time but asynchronous in regards of physical time. This synchronization lets appear the system to be synchronous at the application level.

One of the major advantages of the Isis implementation is the ability to run over any network supplying an Internet Group Multicast Protocol (IGMP). Isis abstracts the group communication. This abstraction takes the form of a multicast service composed of a set of basic primitives providing different kinds of broadcast operations. The most noticeable basic primitive is the *Causal*

BroadCAST (CBCAST). CBCAST sends and receives messages in an atomic fashion with a causal order. The *Atomic BroadCAST* (ABCAST) primitive ensures a total order and an atomic delivery of messages. Finally, the *Group BroadCAST* (GBCAST) primitive is in charge of groups' management (creation, destruction, membership, etc.), and specially to inform the members of a group that one of them disappears (a fault). GBCAST delivers messages in causal order.

CBCAST, ABCAST and GBCAST are based on a time service. This service is also linked with the group membership service in order to provide a precise description of the group state at a given time. The virtual synchronization concepts and the CBCAST, ABCAST, GBCAST primitives are presented in [BIR 93b].

Isis was successful. Lots of companies and universities employ(ed) the toolkit in a very large scope of activities: from telecommunications switching systems to financial trading floors. Isis exposed, the first, the real needs of group communication tools in distributed application development. The interests in Isis and its group communications made the project evolve in Horus, where the group communication takes a bigger importance.

Membership	Dynamic
Structure	Open group
Ordering	Total, causal, and FIFO
Reliability	Reliable
User interface	Communication by message

Table 2.6: Isis properties

Horus

The Horus project [REN 93] began as an effort to reorganize the group communication system of Isis [1]. It has evolved into a general purpose communication architecture with advanced support for the development of robust distributed systems in settings for which Isis was not suitable, such as applications that have special real-time or security requirements (for instance, automatic communication encryption in unsecured environment is not supported in Isis).

The major improvements target the architecture and the flexibility of the system. The strategy of the Horus system takes up the principle of the *streams* introduced by the UNIX System V. This strategy consists in separating the concerns into independent modules. Each module is in charge of a specific communication feature: flow control, acknowledgment, encryption, etc. Those modules can be stacked up like the streams. The user chooses the modules he needs depending on the execution context and composes them. Figure 2.6 presents the architecture of Horus.

The *MUlticast Transport Service* (MUTS), the lower layer, hides the underlying operating system (interfacing issues, network protocol, thread creation and synchronization, etc.). The MUTS gives an abstraction of the underlying system for the higher (user) layers. It provides an asynchronous, reliable, one-to-many message passing model over lots of network protocols.

The *Virtual synchronous subsystem* (Vsync) runs on the top of MUTS. It function is to extend MUTS into a full group communication environment, supporting fault-tolerant multicast. Vsync provides ordering semantics on multicast communication and basic process group abstraction, with strong semantics on the ordering.

Vsync is composed of the *Vsync membership layer* and, over, the *Vsync protocols* layer. The Vsync membership layer, also called VIEWS, is in charge of membership, atomicity and encryption. The Vsync protocols layer provides total ordering, multiplexing, progressive and conservative protocols. The qualities of service of Horus include best-effort and reliable communications (with different ordering: FIFO, causal, synchronous).

[1] In Egyptian mythology, Horus is the son of Isis.

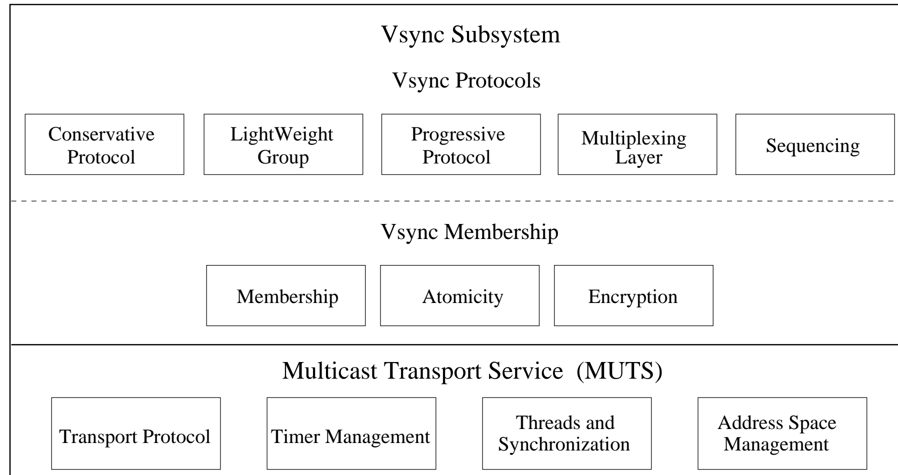


Figure 2.6: Horus layers

Finally, at the user level, Horus offers Isis compatibility libraries and tools. Optimizations and improvements of Horus regarding to Isis are presented in [REN 96, REN 94]. Isis, and consequently Horus, have contributed a lot towards the theory of virtual synchrony.

Membership	Dynamic
Structure	Open group
Ordering	Total, causal, FIFO, and non-ordered
Reliability	Reliable or not reliable
User interface	Communication by message

Table 2.7: Horus properties

2.2.3 Parallel Virtual Machine

Parallel Virtual Machine (PVM) [GEI 94] is a generic system for the programming of parallel and distributed applications communicating by exchange of messages. The PVM environment is composed of libraries and processes of service. The libraries (one for the C language and another for the FORTRAN language) connect the parts of a distributed application and the PVM system. Processes of service run on each node of the platform to manage the application's processes and their communications. PVM aims to operate heterogeneous systems in a transparent way.

PVM was a key component of the evolution of the community of distributed applications. PVM contributed to the emergence of use of network of workstations instead of parallel machines (i.e. *Symmetrical Multi-Processing* (SMP) computers). PVM gives the illusion to applications that they are running on a (virtual) parallel machine. If more than one user runs a PVM application on the same set of hosts, each application runs independently, on independent parallel virtual machine. PVM manages the tasks of an application and can be in charge of an automatic processes placement on the nodes composing the virtual machine. However there is no load-balancing concern in the placement mechanism; processes are placed using a cyclic manner with no regards to the load of the node.

PVM proposes point-to-point and multipoint, reliable, asynchronous, communications FIFO ordered by the source. Multipoint communications are achieved with direct addressing or with process groups. Process groups are dynamic and open (only in the same application of course). Group management is centralized; primitives return information about the composition of a group (ID, instance, size, etc.). The schemes of multipoint communication are multicast, barrier and reduce. Send and receive functions are blocking.

Group communications are not very efficient in PVM. The first step of a group communication is to acquire the list of group members from a group server. Then, a replication of point-to-point communication addressed to each member achieves the group communication. Some implementations provide optimizations for parallel machines. For network of computers, some implementations optimize the mechanism using reliable and ordered IP protocols.

Membership	Dynamic
Structure	Open group
Ordering	FIFO ordered
Reliability	Reliable
User interface	Communication by message

Table 2.8: PVM properties

2.2.4 Message Passing Interface

The *Message Parsing Interface* (MPI) results from a standardization effort to build parallel and distributed applications communicating by exchange of messages [MPI94]. The MPI forum, involving more than sixty peoples, led to a standard designed for high performance on both massively parallel machines and on workstation clusters.

MPI aims to ease design and portability of parallel and distributed applications. MPI is widely available, with both free available and vendor-supplied implementations targeting both parallel machines and networks of workstations. For instance, LAM [BUR 94] and MPICH [GRO 96] are famous free implementations of the MPI standard. The standard proposes bindings for C and Fortran 77 languages. An application written for a specific platform using one implementation should be able to run on any platform that provides an implementation of MPI. Actually, the standard is so large in terms of features that implementations proposed on various platforms do not implement all those features.

The MPI Forum chose to define an Application Programming Interface (API) that proposes efficient and reliable schemes of communication. The goal is to overlap computing phases with communications and avoid memory copies of messages, this, in a possible heterogeneous environment. The API is independent of the programming language. Finally, the standard defines point-to-point communications, collective communications (understand group communication in the MPI vocabulary) and processes management for communication.

Collective communications are performed using lists of processes named communicators. Communicators are static, but new communicators can be created at runtime descending from existing one. A collective operation is executed by having all processes in the group call the communication routine, with matching arguments. Communicators are key arguments, they define the group of participating processes and provides a context for the operation. Basic primitives such as *broadcast* or *gather* have a single sending or receiving process named *root*. Some arguments in the collective primitives are said *"significant only at the root"*; all participants except the root ignore them. Advanced primitives such as *all-gather* or *all-to-all* involve several sender and receiver processes in a more complex scheme.

Collective primitives may (not necessarily) return as soon as their participation in the collective communication is complete. The completion of a primitive call does not indicate that the other processes in the group have completed or even started the operation; it just indicates that the caller process is now free to access the communication buffer. So a collective communication may (or may not) have a synchronization effect on the calling processes [2].

[2] We ignore here the *barrier* primitive whose role is exclusively to synchronize processes and not to exchange data.

The initial standard document was updated by the MPI Forum. The new version (MPI-2) contains both significant enhancements to the existing MPI core and new features [MPI97]. Improvements mainly address dynamic process creation and management, one-sided communication, parallel Input/Output, and C++ bindings.

Membership	Static
Structure	Closed group
Ordering	FIFO and no-ordered
Reliability	Reliable
User interface	Communication by message

Table 2.9: MPI's collective communication properties

2.2.5 Object Group Service: a CORBA Service

The *Common Object Request Broker Architecture* (CORBA), specified by the *Object Management Group* (OMG), provides interoperability between languages, platforms, and implementations in an object-oriented way. CORBA does not offer basically a high-level group communication service. Several projects proposed their solution, for instance: Electra [MAF 95] and Orbix+Isis [ION 94].

[FEL 96] proposes the design, and [FEL 97, FEL 98b] the implementation, of a CORBA service for a reliable multicast communication. The *"object" group communication* takes the form of a service added in CORBA. This approach is similar to the one adopted by the OMG to enhance CORBA with transactions, persistence, event channels, etc. The *Object Group Service* (OGS) emerged as a new service based on other existing services: principally the naming, messaging, monitoring, and multicast services. The multicast service of CORBA only provides unreliable message broadcasts without any quality of service or ordering [OMG01]. Group communication may have be obtained with the event service [OMG04], but it presents some limitations: no guarantees concerning ordering, atomicity, and failures.

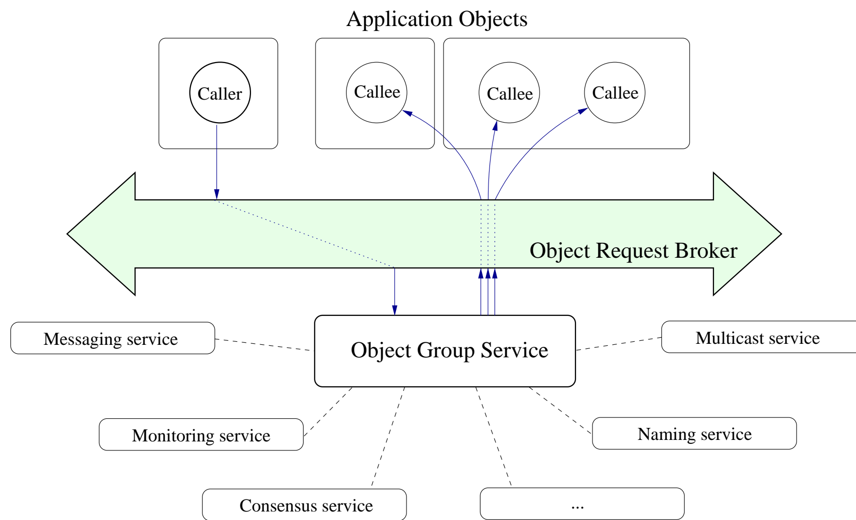


Figure 2.7: The Object Group Service

In a group, the objects are of the same type; like for any CORBA object, the type is defined with the *Interface Definition Language* (IDL). It allows a client to communicate with the objects of a group by invoking an operation (method) defined in the IDL interface. Old fashion group communication is still available with the `deliver()` operation, nevertheless only the values of type `any` can be send as messages.

Each client has to perform a specific binding phase to acquire a reference to the group. Then, it becomes possible to execute communication. By default a group communication returns only one result; to receive more results the client must invoke explicitly the OGS and so looses the benefits of transparency.

Groups are dynamic. When a member joins or leaves the group, all group members are notified; so each member knows the current membership. OGS uses a special interface to manage the status of an object with regard to a group; typically, join (`join_group()`) or leave (`leave_group()`) the group.

Membership	Dynamic, with a dedicated *view* for group management
Structure	Open group
Ordering	Total, FIFO, and no-ordered
Reliability	Reliable
User interface	Typed communication

Table 2.10: Object Group Service properties

2.2.6 JGroups

JGroups (previously named JavaGroups) is a reliable group communication toolkit written entirely in Java [BAN 98]. It was developed at the Cornell University. It is based on IP Multicast, but extends it with reliability and advanced group membership. Reliability includes lossless transmission of a message to all recipients (with retransmission of missing messages), fragmentation of large messages into smaller ones and reassembly at the receiver's side, FIFO ordering of messages, and atomicity. The JGroups' membership includes knowledge of who the members of a group are and notification when a new member joins, an existing member leaves, or an existing member has crashed.

The architecture of JGroups, shown in Figure 2.8, consists of 3 parts: *(1)* the building blocks, which are layered on top of the channel and provide a higher abstraction level, *(2)* the Channel API used by application programmers to build reliable group communication applications, and *(3)* the protocol stack, which implements the properties specified for a given channel.

JGroups offers building blocks that provide more sophisticated APIs on top of a Channel. Building blocks either create and use channels internally, or require an existing channel to be specified when creating a building block. Applications communicate directly with the building block, rather than the channel. Building blocks are intended to save the application programmer from having to write tedious and recurring code.

Dynamic group management is performed with *Channel*. A channel represents a group. To join a group, an object has to reach a channel by specifying its name. If the channel already exists, the object is added to the group. If the channel does not exist, it is immediately created. Members of a channel send messages only to all others members, and receive messages only from other members. A channel is destroyed when its last member leaves. Channels are similar to BSD sockets: messages are stored in a channel until a client removes the next one (pull-principle). When no message is currently available, a client is blocked until the next available message has been received. A channel can be implemented over a number of alternatives for group transport. Therefore, a channel is an abstract class, and concrete implementations are derived from it.

JGroups is based on a flexible protocol stack, which allows programmers to adapt it in order to best fit their application requirements. All messages sent and received over a channel have to pass through the protocol stack. Every layer may modify, reorder, pass or drop a message, or add a header to a message. A fragmentation layer might break up a message into several smaller messages, adding a header with an id to each fragment, and re-assemble the fragments

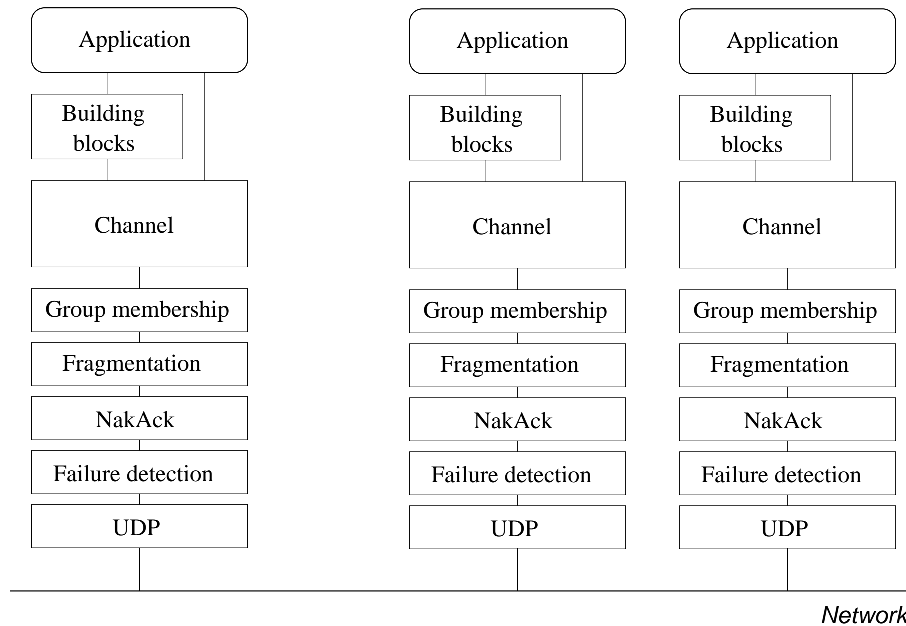


Figure 2.8: JGroups architecture

on the receiver's side. The composition of the protocol stack is determined by the creator of the channel. JGroups comes with already defined protocols (UDP, IP Multicast, TCP, etc.) and offers the ability to adapt to new protocols.

Through its flexible protocol stack architecture, JGroups can be adapted to many environments. This can be done by replacing, removing or modifying existing protocols, or by adding new protocols. JGroups is a good testbed for development and experimentation of new reliable multicast protocols written in Java.

Membership	Dynamic
Structure	Closed group
Ordering	Total, causal, and FIFO
Reliability	Reliable
User interface	Communication by message

Table 2.11: JGroups properties

2.2.7 Group Method Invocation

The first approach of the Vrije Universiteit, Amsterdam, to group communication in Java was *Replicated Method Invocation* (RepMI) [MAA 00]. RepMI is only designed for replication. The goal is to make parallel applications more efficient without increasing the complexity of implementation. The idea is to copy and distribute the objects frequently accessed. A local copy of such object reduces the overhead introduced by numerous remote accesses.

Read operations are only performed on one local (copy) object. Write operations are performed on the (closed) group of replicates named *cloud* using a synchronized method to ensure consistency. Each cloud has a single entry point (the *root*), which is the only object on which methods may be invoked from outside a cloud.

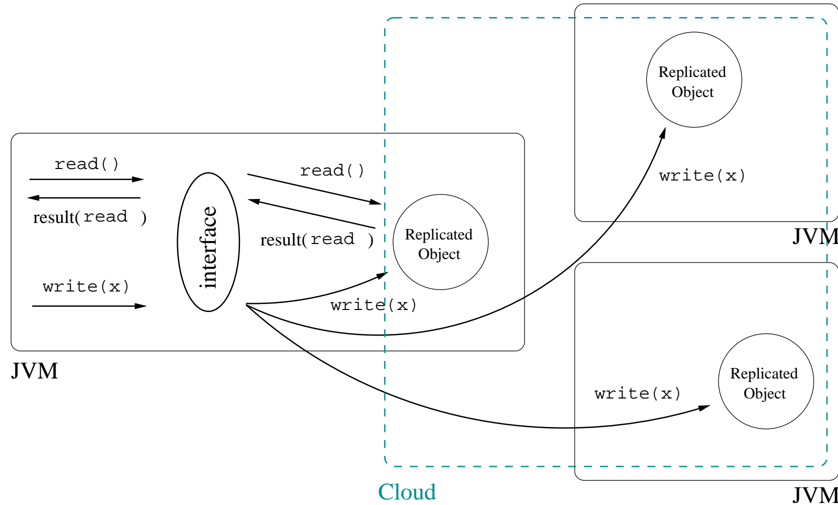


Figure 2.9: RepMI

RepMI was unsatisfactory to build distributed applications. RepMI is only suitable for expressing shared data. So the *Group Method Invocation* (GMI) was developed to provide a complete and flexible group communication mechanism [MAA 02, MAA 03]. Like RepMI, GMI expresses group communication using method invocation and is based on a specific compiler.

GMI extends the RMI model in three ways: *(1)* a stub may refer to a group of object, *(2)* method invocations and gathering of results may be managed in different ways, and *(3)* stubs and skeletons are configurable at runtime to provide different schemes of communication.

Groups are created dynamically but become immutable[3] as soon as they are created. Objects of the group must extend the `GroupMember` class that provides some methods needed by the GMI environment. Objects of the group must also implement the `GroupInterface`. This interface does not define any method; it is just a marker interface for the compiler. To finish, group members must implement a common interface. Only the methods defined in this last interface can be invoked as group communication.

Method invocation schemes can be configured independently for each method. GMI gives four schemes to the programmer: The *single invocation* that invokes a method on only one object of the group, the *group invocation* that forwards the call to every group members, the *personalized invocation* that communicates with every group members while parameters vary from a member to an other, and finally, the *combined invocation* that uses different group references and several threads to invoke the same method on a group using a combinator method on the results defined by the programmer.

Result-handling is also configurable. The provided schemes are: the *discard result* that does not return any result, the *return one result*, the *forward results* that returns all the results in a handler object defined by the programmer, the *combine results* that combines all the results using a combinator method defined by the programmer, and the *personalized result* that returns a result to each thread involved in the group communication.

GMI offers a flexible mechanism to achieve group communication in Java. It proposes an efficient object-oriented approach to build high-performance application, based on a specific compiler and the use of low-level multicast primitives provided by the network.

[3]The term *immutable* means unchangeable in object-oriented parlance. Immutable objects do not change once their constructor has executed.

Membership	Static: Immutable after a dynamic creation of the group
Structure	Open group
Ordering	FIFO ordering
Reliability	Reliable (but advocates that unreliability could be easily added in future)
User interface	Typed communication

Table 2.12: Group Method Invocation properties

2.3 Analysis of related work

In this section, I discuss about the points I judge negative in the last presented group toolkits that address object-oriented environments (the Java language and CORBA). Then, I outline the features I believe fundamental for a group communication toolkit intended to ease efficient distributed programming.

2.3.1 Drawbacks

Let us start with JGroups. It is a valuable project that recently succeeded being an important part of the underlying framework for implementing the clustering features of the JBoss J2EE Application server. However JGroups is more centered on the low layers of communication than on the API provided to the programmers (the building blocks). This choice allows adaptability and fine configuration of the protocols stacks at the cost of easiness in complex code writing. The building blocks that may offer a far more sophisticated interface than the channels are actually too few to provide a full set of communication schemes useful at a final application level. JGroups' author admits: "*The point is that I have really never put much effort into the building blocks, b/c my focus has always been protocol design.*" In practice, the building blocks may be totally ignored and the application may directly address to the channel layer. The channel programming style is a socket programming style. One can regret that JGroups, a Java toolkit, do not deal with the object-oriented programming style, i.e. remote method invocation.

The Object Group Service is a pioneer work in adding a group communication mechanism into CORBA, an object-oriented middleware, using an object-oriented programming style. The integration into the middleware is well thought; it attests much cares to comply with the CORBA models and philosophy. Actually the implementation is too much CORBA-focused and pains to express more general patterns for adding group communication into other middlewares. Also the main concern of OGS is the fault tolerance handling by replication. Thus as previously mentioned a group communication returns (by default) only one result from the group of replicated objects. To receive more results the client has to invoke explicitly the OGS and so loses the benefits of transparency. This group communication mechanism is less suitable for building any distributed application than for introducing fault tolerance in an existing application.

Finally, let us consider RepMI and GMI. RepMI provides only replication, but contrarily to OGS it is not principally designed for fault tolerance but for increasing performances of distributed application by accelerating access to shared data. Replication is not really group communication; it is only suitable for expressing shared data, so GMI was introduced. Both RepMI and GMI communicate through an interface to the groups members; they are extensions of the object-oriented communication scheme of RMI. GMI offers a large set of communication strategies, in the sending of method invocations and in the result handling. Actually, despite the effort to hide complex communication code to the programmer, he or she has to write the desired strategy for a group communication. Moreover, objects that are subjects to be added in a group must implement the marker interface `GroupInterface` and extend the `GroupMember` class, which

acts like the `UnicastRemoteObject` of RMI. This yields two problems: *(1)* It lacks of dynamicity, an instance of a class that does not extend `GroupMember` nor implements `GroupInterface` will never be able to belong to a group; and *(2)* It takes back the drawback of RMI that forces to inherit from a specific class and thus may disturb the original class hierarchy built by the programmer. Finally, one may regret that GMI groups are static; dynamicity is a valuable property for building distributed applications.

2.3.2 Proposal

According to the experience of [FEL 98a], we had to choose between three approaches for adding group communication to a middleware:

1. The ***integration approach*** consists of modifying and extending the part of the middleware acting as Object Request Broker (ORB) with group communication. This approach has been adopted in both Orbix+Isis [ION 94] and Electra [MAF 95] projects. A client is allowed to perform invocation to a group of objects as if it was a plain object. The communication layer of the middleware performs replicated invocations to group members, gathers the result and returns them to the client. This approach provides a complete group transparency in the writing of code.

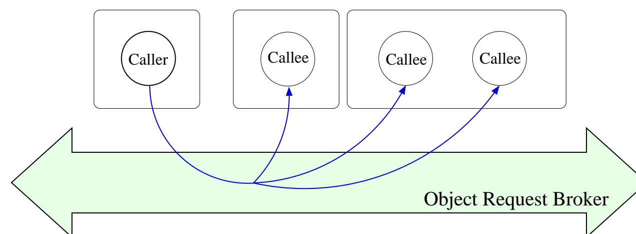


Figure 2.10: The integration approach

2. The ***service approach*** consists of providing the group communication as service in the middleware, on top of the ORB. This approach was adopted in the Object Group Service, the group mechanism introduced in the CORBA middleware and described in [FEL 96, FEL 97, FEL 98b] (see Section 2.2.5). The external service offers a basic set of primitives for handling group mechanism: management and remote method invocation. Achieving group transparency is not straightforward with this approach.

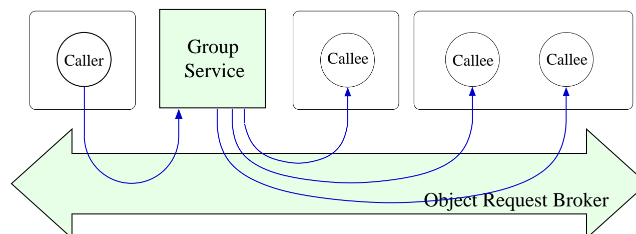


Figure 2.11: The service approach

3. With the ***interception approach***, the ORB is not aware of replication. Requests are intercepted transparently on client and server sides using low-level interception mechanisms; they are then passed to a group communication toolkit that forwards them using group multicasting. This approach does not require any modification to the ORB, but relies on OS-specific mechanisms for request interception. For instance, Eternal [MOS 98] uses the Unix operating system abilities to intercept the messages before they join the TCP/IP layer and redirect them into a group communication mechanism.

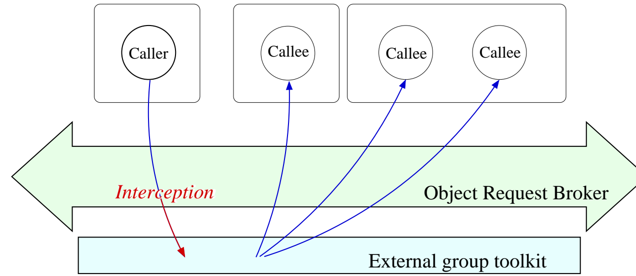


Figure 2.12: The interception approach

Considerations regarding implementations and the choice we made to build our group communication mechanism are presented in Chapter 5. Here, let us focus on the main points we want to contribute on. Our goal is to free the programmer from having to implement the complex communication code required for group communication, this by allowing the focus to be on the application itself. Group communications must be expressed using (remote) method invocations just like RMI expresses point-to-point communications. This integrates well with the object-oriented fashion of modern languages.

However we do not want to force the programmer to implement or extend specific interfaces and classes. Indeed, such an obligation would yield constraints at creation of the application and would arm dynamicity at runtime. Group issues must be addressed by the group mechanism with no impact on the programmer's code writing. The common interface of group members must be sufficient to express the largest set of communication schemes, such as various sending and receiving strategies. Of course, an interface is necessary to explicitly manage groups when needed. This interface must define creation, add, remove, etc. operations that are not accessible from the members interface. We aim at clearly separate the concerns for group management and the concerns for functional aspects (i.e. communications by method invocations). Separation of concern is essential for mastering complexity.

In addition, a solution that would seamlessly integrate the group transparency paradigm from the beginning of the call to the gathering of replies is still missing. Indeed, if transparency of method invocation is well assumed by existing toolkits, none provides a transparency of the result handling that still has to be aware of the group aspect with no lost of information. Two partial solutions are generally proposed. The first one consists of returning a single result that may be the combination of the whole results or an arbitrary selection among them. In this case, information about each individual results may have been discarded (even by combination or selection). The second solution is to explicitly invoke the group service or handle multiple results within a specific object: in that case transparency is broken. We propose to look for a solution that deals with multiple results gathering while conserving all individual replies and being type compatible with the expected result.

Conclusion

Depending on their needs, each community addresses group communication in its own way. Some specific issues might be dedicated to a community, so they should be given a minor importance or simply ignored by others. For example, environment to build fault-tolerant application focus on the ordering issue and does not matter of the scheme of group communication at the programmer layer. On the other hand, to build distributed applications a FIFO ordering is most of time sufficient but elaborate schemes of group communication at the programmer layer are required. We are interested in the issues of building distributed applications; our challenge is to provide the most easy to use and flexible group toolkit to the programmers.

Chapter 3

ProActive

This chapter presents the environment in which my work is included. *ProActive* is an open source[1] Java library for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, *ProActive* provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on cluster of workstations, or on Internet Grids. *ProActive* is only made of standard Java classes, and requires no changes to the Java Virtual Machine, no preprocessing or compiler modification; programmers write standard Java code. Based on a simple Meta-Objects Protocol, the library is itself extensible, making the system open for adaptations and optimizations. *ProActive* currently uses the RMI Java standard library as default portable transport layer.

Section 3.1 presents the programming model promoted by *ProActive*. It introduces the distribution model based on active objects and the way those objects communicate. Section 3.2 gives details of the library. It presents the deployment model, the Meta-Objects protocol on which *ProActive* relies, and the migration mechanisms.

3.1 Programming model

Due to its platform-independent execution model, its support for networking, multithreading and mobile code, Java has given hope that easy Internet-wide high-performance network computing was at hand. Numerous attempts have then been made at providing a framework for the development of such metacomputing applications. Unfortunately, none of them addresses *seamless sequential, multithreaded, and distributed computing*, i.e. the execution of the same application on a multiprocessor shared-memory machine as well as on a network of workstations, or on any hierarchical combination of both. *ProActive* addresses such features [CAR 98a].

3.1.1 Distribution model

The *ProActive* library was designed and implemented with the aim of importing reusability into parallel, distributed, and concurrent programming in the framework of a MIMD[2] model. Reusability has been one the major contributions of object-oriented programming, *ProActive* brings it into the distributed world. Most of the time, activities and distribution are not known at the beginning, and change over time. Seamless implies reuse, smooth and incremental transitions.

A huge gap yet exists between multithreaded and distributed Java applications which forbid code reuse in order to build distributed applications from multithreaded applications. Both

[1] Source code under LGPL license
[2] MIMD stands for Multiple Instruction Multiple Data

Java RMI and Java IDL, as examples of distributed object libraries in Java, put an heavy burden on the programmer because they require deep modifications of existing code in order to turn local objects into remote accessible ones. In these systems, remote objects need to be accessed through some specific interfaces. As a consequence, these distributed objects libraries do not allow polymorphism between local and remote objects. This feature is the first requirement for a metacomputing framework. It is strongly required in order to let the programmer concentrate first on modeling and algorithmic issues rather than lower-level tasks such as object distribution, mapping, and consequently communications in a distributed environment.

The model of distribution and activity of *ProActive* is part of a larger effort to improve simplicity and reuse in the programming of distributed and concurrent object systems [CAR 93, CAR 96], including a precise semantics [ATT 00]. It contributes to the design of a concurrent object calculus named ASP (Asynchronous Sequential Processes) [CAR 04, CAR 05b]. As shown in Figure 3.1, *ProActive* seamlessly transforms a standard centralized monothreaded Java program into a distributed and multithreaded program.

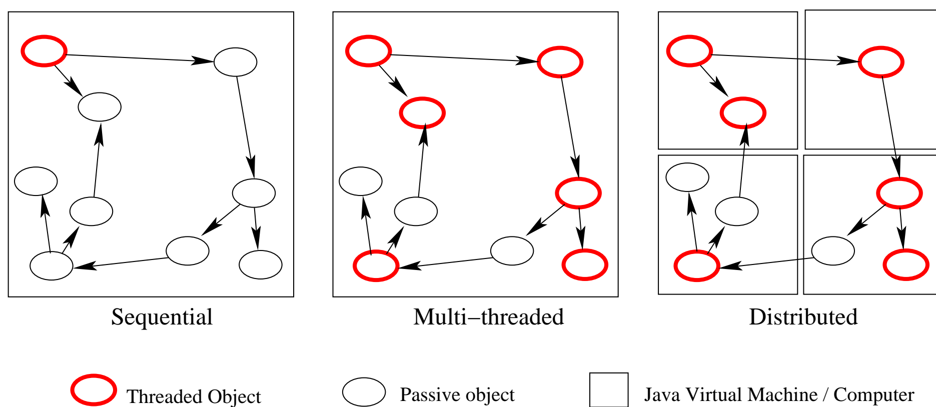


Figure 3.1: Seamless parallelization and distribution with active objects

3.1.2 Active objects

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Objects that are not active are designated as *passive*.

Given a standard object, we provide the ability to give it: location transparency, activity transparency and synchronization. This is obtained only with modifications of the instantiation code. For example, see the standard Java object created by:

```
A a = new A ("toto", 17);
```

There are three ways to transform a standard object into an active one:

1. The *Class-based* approach is the more static one. A new class must be created extending an existing class, and must implement the `Active` interface. The `Active` interface is a tag interface that does not specify any method. This approach allows adding specific methods useful in distributed environment and possibly to define a new service policy in place of the default *First In First Out* (FIFO) service (see Section 3.1.5 for further details about service policy).

```
public class pA extends A implements Active { }

Object[] params = new Object[] {"toto", new Integer (17)};

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

The array of objects `params` represents the parameters to use for the remote creation of the object of type `A`. `node` is an abstraction to the physical location of an active object (refer to Section 3.2.1).

2. With the *Instantiation-based* approach, a Java class that does not implement the `Active` interface is directly instantiated without any modification to create an active object. The parameters `params` and `node` play the same role as previously.

```
Object[] params = new Object[] {"toto", new Integer (17)};

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

3. Finally, the *Object-based* approach is the more dynamic approach. It allows transforming an already existing Java object into an active object possibly remote. It is possible to turn active and remote objects for which the source code is not available, a necessary feature in the context of code mobility. If the `node` parameter is `null` or designate the local JVM new elements are created to transform the object into active object (those elements are meta-objects presented in Section 3.2.2). Otherwise, if `node` refers to a remote JVM a copy of the object is sent on the remote JVM and transformed into an active object. The original passive object remains on the local JVM.

```
A a = new A ("toto", 17);

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

3.1.3 Communication by messages

The active object creation primitives of *ProActive* locally return an object compatible with the original type regarding to polymorphism. So one can perform method call on this object, even if source code was not originally designed to achieve distribution. In distributed object-oriented programming, method call takes the place of *Inter-Process Communication*.

Let us see in details the `A` class:

```
public class A {
  public void foo () {...}
  public V bar () {...}
  public V gee () {...} throws AnException {...}
}
```

Both of those methods will be remotely invoked but the communication semantic will differ.

- The method named `foo` does not return any result, so call the method `foo` will perform only a communication from the caller to the callee. This is a *one-way* method call.
- The `bar` method requires a bidirectional communication. Firstly, from the caller to the callee of course, then from the callee to the caller in order to return the result. With *ProActive* this communication is separated in two steps detailed below. Between those steps the activity of caller does not stop. This is an *asynchronous* method call.
- The `gee` method is quite similar to the method `bar` except that it can raise an exception. As the activity of the caller can not continue, it might go out of the `try/catch` block. The call to `gee` is a *synchronous* method call. Methods returning a primitive type or a final class are also invoked in a synchronous way (details come below).

In both cases, a *rendez-vous* ensures that the method call reaches the callee. As RMI is the transport layer, and as RMI is reliable, the remote method call of *ProActive* remains reliable. Objects given as parameters are copied on the caller side to be transmitted to the callee side. The Table 3.1 summarizes the communication schemes according to method signatures.

Communication schemes	Conditions
One-way	return `void` and do not declare throwing any exception
Asynchronous	return a reifiable [3] object and do not declare throwing any exception
Synchronous	return a non-reifiable object or declare throwing an exception

Table 3.1: Communication schemes depending on method signature

Figure 3.2 exposes an asynchronous call sent to an active object and introduces the transparent *future objects* and synchronization handled by a mechanism known as *wait-by-necessity* [CAR 93]. Asynchronous method call is the most developed and usual mechanism, that is why it is detailed here. There is a short rendez-vous at the beginning of each remote call, which blocks the caller until the call has reached the context of the callee; on Figure 3.2, it means that step 1 blocks until step 2 has completed. In the same time a *future* object is created (step 3). A *future* is a promised result that will be updated later, when the reply of the remote method call will return to the caller (step 5). The next section presents synchronization and control of such *futures*.

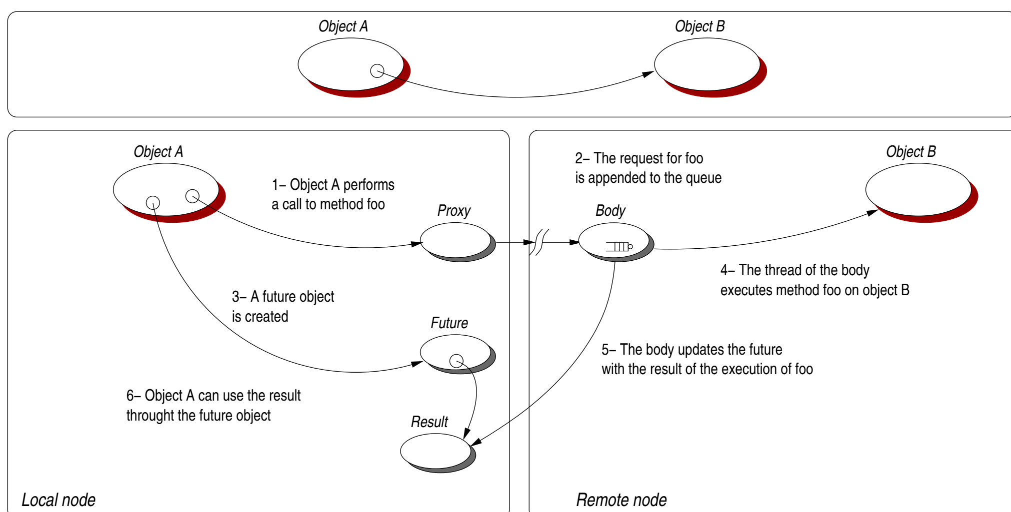


Figure 3.2: Execution of an asynchronous and remote method call

In a synchronous method call, the steps are nearly similar except two main differences. Firstly, the future is not created (no step 3). This is due to the incapacity of the Meta-Objects Protocol to create a future in the case the return type is not reifiable. Secondly, the activity of the caller stops until step 5 has completed (instead of steps 2/3 for an asynchronous call). This ensures that the try/catch block is not passed when the result arrives. A one-way call blocks the activity of the caller until step 2 is achieved; all the *future* operations (creation, update, etc.) are avoided.

ProActive features several optimizations improving performance. For instance, whenever two active objects are located within the same virtual machine, a direct communication is always achieved, without going through the network stack. This optimization is ensured even when the co-location occurs after a migration of one or both of the active objects.

[3] The definition of “reifiable” will be given later, in Section 3.2.2. For the moment just notice that the non-reifiable objects are final objects and primitive types, and reifiable objects are all the others.

3.1.4 Synchronization

As just explained, semantic of the communication depends on the method signature. *ProActive* automatically chooses the best semantic for the given method and automatically deals with the future but it may be possible that the programmer wants to control if by himself or herself. In a first time, we will see the default behavior with the *futures* and then, we will observe the control that the programmer is able to use to manage the asynchronism.

Wait-by-necessity

See our active object `a` now well known:

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

and the asynchronous method call:

```
V v = a.bar();
```

As previously seen, `v` is a future. *ProActive* provides an elegant way to automatically deals with future. It is called *wait-by-necessity*. Consider the new instruction:

```
v.glop();
```

There is no guarantee that the future `v` was updated when the method `glop` is invoked. If the result is arrived and the future updated when the call to `glop` is performed the activity do not stop. Else, if the future was not arrived, the *Wait-by-necessity* is released. This mechanism stops the current activity until the future returns, and then the activity resumes and executes the method. The *Wait-by-necessity* mechanism ensures a maximum efficiency of the asynchronism.

Explicit control

ProActive automatically chooses the best semantic for the given method but it may be possible that the programmer wants to control it by himself. *ProActive* allows the programmer to control the synchronization of asynchronous method calls. After the method call:

```
V v = a.bar();
```

The programmer can use the static primitives `isAwaited` and `waitFor` respectively to test the state of a future and to wait for a future.

```
ProActive.isAwaited(v);
ProActive.waitFor(v);
```

Explicit control gives a finer control to the programmer, no more at the method level but at the object level (*future*).

Besides, *automatic continuations* allow to pass in parameter (or return as a result) *future* objects without blocking to wait their final value. When the result is available on the object that originated the creation of the *future*, this object must update the result in all objects to which it passed the *future*. An automatic continuation is caused by the propagation of a *future* outside the activity that has sent the corresponding request.

3.1.5 Service policy and control of the activity

Customizing the activity of the active object is at the core of *ProActive* because it allows specifying the behavior of an active object. By default, an object turned into an active object serves its incoming requests in a FIFO manner. In order to specify another policy for serving the requests or to specify any other behavior one can implement interfaces defining methods that will be automatically called by ProActive.

The remote method calls being asynchronous, they are stored in the queue (as request) on the callee side. By default the requests are served by a FIFO Service: Active objects are sequential processes. The creation of active object with the class-based (remember Section 3.1.2) permits to change this service policy. The programmer must implement the `RunActive` interface with the `runActivity` method in order to define a new service policy.

Here is a subset of the primitives provided by *ProActive*:

```
void serveOldest (); // Serves the oldest request in queue

void serveOldest (String s); // Serves the oldest s request

void serveOldest (String s, String t); // the oldest of s or t request

void serveOldestWithoutBlocking (); // Serves without blocking

void serveMostRecentFlush (String s); // Serves the newest request
                                      // and removes the others
void serveOldestTimed (int t); // Serves the oldest during no more
                               // than t milliseconds
void waitForNewRequest (); // Non active wait for a request
```

For a concrete example, the following code presents a bounded buffer:

```
public class BoundedBuffer implements Active, RunActive {
  public void runActivity (Body body) {
    Service service = new Service(Body);
    while (body.isActive()) {
      if (this.isFull()) body.serveOldest("get");
      else if (this.isEmpty()) body.serveOldest("put");
      else body.serveOldest();
      body.waitForNewRequest();
} } }
```

The programming of the activity is explicit and the service also. This kind of programming method is very useful when a fine control of the activity is required.

3.2 Environment and implementation

ProActive is only made of standard Java classes, and requires no change to the *Java Virtual Machine* (JVM), no preprocessing or compiler modification; programmers write standard Java code. Using a no modified Java development and execution kit, and the standard Java classes ensure portability and allow running applications with all the JVM implementations. For debugging aspect, especially critical in distributed environment, it is more efficient to avoid source code modification. *ProActive* uses reflection techniques in order to manipulate runtime events such as a method call for instance. Supplementary code is dynamically generated in the same fashion used by *generative* or *active* libraries [CZA 00, VEL 98]. Based on a simple Meta-Object Protocol, the library is itself extensible, making the system open for adaptations and optimizations. *ProActive* currently uses the RMI Java standard library as a portable communication layer, even if the transport layer may be changed (by relying on the `Adapter` object that is in charge of protocol interface with *ProActive*).

3.2.1 Mapping active objects to JVMs: Nodes

Another extra service provided by *ProActive* (compared to RMI for instance) is the capability to *remotely create remotely accessible objects*. For that reason, there is a need to identify JVMs, and to add a few services. *Nodes* provide those extra capabilities: a *Node* is an object defined in *ProActive* whose aim is to gather several active objects in a logical entity. It provides an abstraction for the physical location of a set of active objects. At any time, a JVM hosts one or several nodes. The traditional way to name and handle nodes in a simple manner is to associate them with a symbolic name, which is a URL giving their location, for instance `rmi://lo.inria.fr/node1`.

Let us take a standard Java class `A`. The following instruction creates a new active object of type `A` on the JVM identified with `node1`.

```
// Creation of an active object on a JVM of lo.inria.fr
A a1 = (A) ProActive.newActive("A", params, "rmi://lo.inria.fr/node1");
```

No parameter or a parameter `null` will conduct the active object to be created on the local JVM (i.e. the JVM in which the `newActive` primitive is called).

```
// Creation of two active objects on the current JVM
A a2 = (A) ProActive.newActive("A", params);
A a3 = (A) ProActive.newActive("A", params, null);
```

Passing an active object as parameter triggers the co-allocation mechanism. The active object `a4` will be created in the JVM containing the active object `a1`.

```
// Creation of an active object on the JVM containing a1
A a4 = (A) ProActive.newActive("A", params, a1);
```

Note that an active object can also be bound dynamically to a node as the result of a migration.

Active objects will eventually be deployed on very heterogeneous environments where security policies may differ from place to place, where computing and communication performances may vary from one host to the other, etc. As such, the effective locations of active objects must not be tied in the source code.

A first principle is to eliminate from the source code: the computer names, the creation protocols and the registry and lookup protocols. The goal is to deploy any application anywhere without changing the source code. For instance, we use various protocols (`rsh`, `ssh`, Globus GRAM, LSF, etc.) for the creation of the JVMs needed by the application. In the same manner, the discovery of existing resources or the registration of the ones created by the application can be done with various protocols such as RMIregistry, Jini, Globus MDS, LDAP, UDDI, etc. Therefore, the creation, registration, and discovery of resources have to be done externally to the application.

To reach that goal, the programming model relies on the specific notion of *Virtual Nodes* (VNs): *(1)* A VN is identified as a name (a simple string), *(2)* a VN is used in a program source, *(3)* a VN is defined and configured in a deployment descriptor, and, *(4)* a VN, after activation, is mapped to one or to a set of nodes. The concept of virtual nodes as entities for mapping active objects has been introduced in [BAU 02]. Those virtual nodes are described externally through XML-based descriptors which are then read by the runtime when needed. They help in the deployment phase of *ProActive* active objects (and components).

Of course, active objects are created on Nodes, not on Virtual Nodes. There is a strong need for both Nodes and Virtual Nodes. Virtual Nodes are a much richer abstraction, as they provide mechanisms such as cyclic mapping, for instance. Another key aspect is the capability to describe and trigger the mapping of a single VN that generates the allocation of several JVMs. This is critical to get at once machines from a cluster of computers managed through Globus or LSF. It is even more critical in a Grid application, when trying to achieve the co-allocation of machines from several clusters across several continents.

Moreover, a Virtual Node is a concept of a distributed program or component, while a Node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between Virtual Nodes and Nodes: the function created by the deployment, the mapping. This mapping can be specified in an XML descriptor. By definition, the following operations can be configured in such a deployment descriptor: *(1)* the mapping of VNs to Nodes and to JVMs, *(2)* the way to create or to acquire JVMs, *(3)* the way to register or to lookup VNs.

Now, within the source code, the programmer can manage the creation of active objects without relying on machine names and protocols. For instance, the following piece of code allows creating an active object onto the Virtual Node `Dispatcher`. The Nodes (JVMs) associated in a

descriptor file with a given VN are started (or acquired) only upon activation of a VN mapping (`virtualNode.activateMapping()` in the code below).

```
    // Returns a Descriptor object from the xml file
Descriptor pad = ProActive.getDescriptor("file://descriptor.xml");

    // Returns the virtual node described in the xml file
    // as a Java object
VirtualNode virtualNode = pad.getVirtualNode("vnode");

    // Activates the mapping for the virtual node
virtualNode.activateMapping();

    // Returns the first node available among nodes mapped
    // to the virtual node
Node node = virtualNode.getNode();

    // Creates an active object on a node
A a = ProActive.newActive("A", params, node);
```

3.2.2 MOP: Meta-Objects Protocol

ProActive is built on top of a *Meta-Object Protocol* (MOP) [KIC 91] that permits reification of method invocations and constructor calls. As this MOP is not limited to the implementation of the transparent remote objects library, it also provides an open framework for implementing powerful libraries for the Java language. As for any other element of *ProActive*, the MOP is entirely written in Java and does not require any modification or extension to the Java Virtual Machine, as opposed to other Meta-objects protocols for Java [KLE 96]. It makes extensive use of the Java Reflection API.

An active object provides a set of services, in particular asynchronous communication. It is important to separate concerns to ensure extensibility and maintenance. A meta-object was introduced for each service provided by an active object. Figure 3.3 shows the final decomposition.

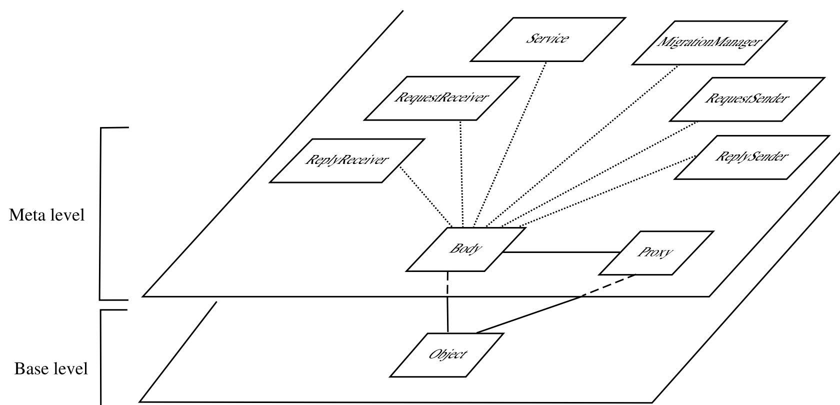


Figure 3.3: Base-level and meta-level of an active object

The MOP creates the couple *stub*/*proxy* and the *body* with its *meta-objects*. The stub is an entry point for the meta-level. The stub inherits from the type of the object. Due to its commitment to be a 100% Java library, the MOP has a few limitations: primitive types cannot be reified because they are not instance of a standard class, or final classes (which includes all arrays) because they cannot be subclassed. So primitive types and final classes are said *not reifiable*. The

stub overloads the public methods of the class. A method invocation creates a `MethodCall` object that represents the executed method call. This object contains the invoked `Method`, information about return type, and a copy of the parameters.

The proxy maintains a reference on the active object. It is responsible for the communication semantic: *(1)* it hides the concept of remote or local reference, and *(2)* it transmits the `MethodCall` object (embedded into a `Request`[4] object) to the body of the active object. If a programmer wants to implement a new meta-behavior using our meta-object protocol, he or she has to write both a concrete (as opposed to abstract) class and an interface. The concrete class provides an implementation for the meta-behavior he or she wants to achieve while the interface contains its declarative part. The concrete class implements the `Proxy` interface and provides an implementation for the given behavior through the method `reify`:

```
public Object reify (MethodCall c) throws Throwable;
```

This method takes a reified call as a parameter and returns the value returned by the execution of this reified call. Automatic wrapping and unwrapping of primitive types is provided. If the execution of the call completes abruptly by throwing an exception, it is propagated to the calling method, just as if the call had not been reified.

The body is the entry point for all communications addressed to the active object. It is the only part of the active object remotely accessible. The body is in charge of its attached meta-objects. A request queue is attached to the body. This request queue stores the messages sent by other active objects to the body. Requests are served with a FIFO service policy that can be customized by the programmer, as presented in the previous section.

3.2.3 Migration

Mobility is the ability to relocate at runtime the components of a distributed application. The *ProActive* library provides a way to migrate an active object from any JVM to any other one [BAU 00]. *ProActive* migrations are weak: it means that the code moves but not the execution state (on contrary to strong mobility). Activity restarts from a *stable state*.

Any active object has the possibility to migrate. If it references some passive objects, they will also migrate to the new location. Since we rely on the serialization to send the object on the network, an active object has to implement the `serializable` interface to be able to migrate. The migration of an active object is triggered by the active object itself, or by an external agent. In both cases a single primitive will eventually get called to perform the migration. The principle is to have a very simple and efficient primitive to perform migration, and then to build various abstractions on top of it. The name of the primitive is `migrateTo`. In order to ease the use of the migration, the `ProActive` class provides two sets of static methods.

The first set is aimed at the migration triggered from the active object that wants to migrate. The methods rely on the fact that the calling thread is the active thread of the active object:

- `migrateTo(Object o)`: migrate to the same location as an existing active object
- `migrateTo(String nodeURL)`: migrate to the location given by the URL of the node
- `migrateTo(Node node)`: migrate to the location of the given node

The second set is aimed at the migration triggered from another agent than the target active object. In this case the external agent must have a reference to the Body of the active object it wants to migrate.

- `migrateTo(Body body, Object o, boolean priority)`: migrate to the same location as an existing active object

[4] `Request` extends `Message`.

- `migrateTo(Body body, String nodeURL, boolean priority)`: migrate to the location given by the URL of the node
- `migrateTo(Body body, Node node, boolean priority)`: migrate to the location of the given node

The `priority` parameter represents two possible strategies: (1) The request is high priority and is processed before all existing requests the body may have received (*priority* = *true*); (2) The request is normal priority and is processed after all existing requests the body may have received (*priority* = *false*).

In order to implement autonomous active objects, a complete API was build on top of the `migrateTo` method. We use *itineraries* and *automatic execution*. An itinerary is a dynamic list of *destination, action* pairs, where destination is the host to migrate and action is the name of the method to execute on arrival. The method to execute differs from host to host. Automatic execution is handled by `onArrival` and `onDeparture` methods of the `MigrationStrategyManager` class. Those methods can call other methods and access attributes of the object. `onArrival` executes instructions when the object arrives on a new location, before it begins to serve external requests. `onDeparture` executes instructions just before the object leaves the node.

To answer the location problem (find a migrated object, maintain connectivity), we propose two solutions: the *forwarders* and the *location server*. A forwarder is a reference left by the active object when it leaves a host: this reference points the new location of the object. Multiple migrations create a chain of forwarders; some elements of chains may become temporarily or permanently unreachable because of a network partition or a single machine in the chain failure. Longer chains produce worse performance because of multiple “jumps” of the message. So *ProActive* uses *tensioning* to shortcut the chain of forwarders: after a migration, the first method call updates the location of the migrated object to the caller and creates a direct link. This mechanism is presented by Figure 3.4.

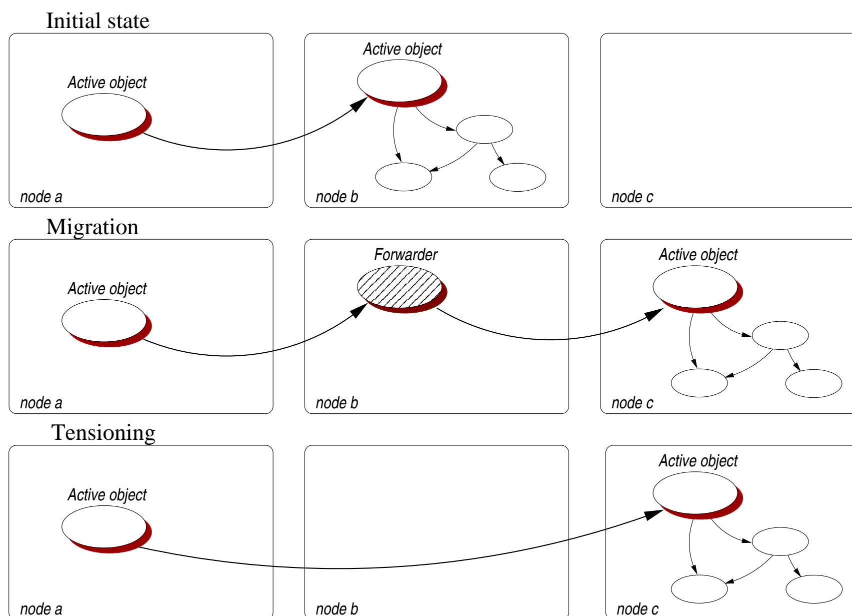


Figure 3.4: Migration and tensioning

With the second solution, the location server tracks the location of each active object. Every time an object migrates, it sends its new location to the location server. After a migration, all the references pointing to the previous location become invalid. When an object tries to communicate

with a migrated active object (the reference is no more valid), the call fails and a lazy mechanism transparently (1) queries the location server for the new location of the active object, (2) updates the reference regarding to the server's response, and (3) re-performs the call on the object at its new location. On contrary to the forwarder approach, the location server approach produces additional messages: firstly, by the migrated object to the server after its move, and secondly, by the failed communication. Those approaches are discussed and modeled in [HUE 02] regarding many parameters such as migration rate, communication rate, average time on site, average time of migration, communication latency, etc.

Conclusion

In summary, the essence of *ProActive* is as follows: a distributed object oriented programming model that we are extending smoothly to get a component based programming model (in the form of a 100% Java library, refer to Section 8.3 for more details); moreover this model is "grid-aware" in the sense that it incorporates from the very beginning adequate mechanisms in order to further help in the deployment and runtime phases on all possible kind of infrastructures, notably secure grid systems. This programming framework is intended to be used for large scale grid applications.

In addition to RMI, *ProActive* also permits the use of other communication protocols, such as Jini, Ibis, HTTP, etc. Many new features are currently in development. The more noticeable ones are (by order of decreasing maturity): hierarchical deployment-based security [ATT 03], fault tolerance [BAU 04, BAU 05], non-functional exception handling [CAR 03, CAR 05a], load balancing, and peer-to-peer computing.

Chapter 4

Typed group communication

The RMI model only provides synchronous point-to-point communication. The communication of *ProActive* enhances the RMI communication of Java with asynchronism, *futures*, automatic synchronization and *wait-by-necessity*. Despite those improvements, very often, a parallel and distributed application needs more advanced models like group communication. Many parallel and distributed applications need the ability to combine many objects (remote or not) in order to communicate with them in one shot. Such kind of operations has to be more efficient than a replication of a simple point-to-point communication while it must preserve a similar behavior. In addition we want to maintain a transparency regarding group communication, not only during the sending of method invocations but also during the gathering of replies.

In this chapter, I present the design and features of a typed group communication providing an elegant way to extend the *ProActive* communication scheme (and consequently the RMI communication scheme). Section 4.1 presents the objectives aimed for a group communication mechanism and the model I propose. Section 4.2 describes the programmer interface and at the same time the communication semantic. Finally, Section 4.3 presents advanced features provided by the group communication mechanism.

4.1 The typed group model

Alternate approaches for parallel and distributed computing in Java include the use of more dedicated parallel programming frameworks, such as parallel and distributed collections [FEL 02] which hide the presence of parallel processes, or implementations of MPI-like libraries in an SPMD programming style [NEL 01]. As defined in [BAD 02b], the group mechanism I propose is more general, as it enables to build such alternate parallel programming models, while being able to provide group communication to distributed applications originally not aimed at being parallel.

4.1.1 Objectives

ProActive is oriented to cluster computing, grid computing and desktop computing. The objectives and constraints in term of platform are the following: we aim at very large amount of computers. Grid computing may involve up to 100,000 computers. Of course, in such large amount of computers, computers are heterogeneous; they can be personal computers, members of a cluster, multi-processors or not. *ProActive*, with Java, hides the hardware, and exposes only JVMs. Any operating system providing a Java environment (JVM) is acceptable. There is no requirement about the network, except that any computer should be able to join any other computer; the topology of the network has not to be known. Finally, the environment is, of course, multi-user and multi-task.

The design of the group communication system is an important aspect in providing a flexible environment for the development of a wide range of distributed applications and services. So, about the group communication system itself, the objectives are:

- *Efficiency*. System resources must be saved. A group communication must be more efficient than a succession of point-to-point communications. The network latency must be overlapped. Efficiency is the key feature in distributed computing world.
- *Scalability*. This is the second key feature. In a wide network such as Internet, many computers are available. The combination of several clusters may also reach thousands of computers. The group communication system must be able to handle a very large number of members.
- *Transparency*. The call to group communication primitives for management (creation, membership, etc.) must be minimized. The source code must remain clear. The activity should be exhibited while the group management concerns should be pushed into the background. The addition of group communication should be the less intrusive it is possible in code writing. Existing code should be not or just slightly modified to benefit from group communication.
- *Dependability*. The behavior must remain coherent in case of failure: disappearing of computers, unexpected results (exceptions), etc. The robustness of a distributed application may depend on the robustness of its group communication system.
- *Flexibility*. The group communication must adapt itself to the needs of a large scope of applications. So it should not be too restricting, it means that the group communication must be sufficiently open to provide all semantics of communication required by the programmer to build him or her specific application.
- *User-friendliness*. The API must be quite easy to use and very functional. The group communication difficulties have to be hidden to the programmer. It must be managed by the environment runtime system.
- *Evolution*. The group mechanism must be able to adapt to new communication protocols (RMI-like or not). It should adapt also to non-functional features, principally security mechanisms, but also fault tolerance and extended asynchronism such as automatic continuation.

4.1.2 Typed groups

The group communication mechanism is built upon the *ProActive* elementary mechanism for asynchronous remote method invocation with automatic future for collecting a reply. As this last mechanism is implemented using standard Java, such as RMI, the group mechanism is itself platform independent: it requires no changes to the JVM, no preprocessing or compiler modification, like the rest of the library. A group communication must be thought of as a replication of more than one (say N) *ProActive* remote method invocations towards N active objects. Of course, the aim is to incorporate optimizations into the group mechanism implementation, in such a way as to achieve better performances than a sequential achievement of N individual *ProActive* remote method calls. In this way, our mechanism is a generalization of the remote method call mechanism of *ProActive*, built upon RMI, but as we will see further nothing prevents from using other transport layers in the future.

The availability of such a group communication mechanism, simplifies the programming of applications with similar activities running in parallel. It is natural to group together similar activities because they are subject to receive the same data or the same instructions. Similar method invocations target similar activities. In an object-oriented framework, this idea of similar activities is translated by the fact of implementing an interface. All members of a group have to implement a common interface, or extend a common superclass. Indeed, from the programming point of view, using a group of active objects of the same type, subsequently called

a *typed group*, takes exactly the same form as using only one active object of this type. The multi-communication, to each member of a group, is abstracted from the code; only the functional aspect remains.

The construction of such group is possible due to the fact that the *ProActive* library is built upon reification techniques: the class of an object that we want to make active, and thus remotely accessible, is reified at the meta-level, at runtime. In a transparent way, method calls towards such an active object are executed through a stub which is type compatible with the original object. The stub's role is to enable to consider and manage the call as a first class entity and applies to it the required semantic: if it is a call towards one single remote active object, then the standard asynchronous remote method invocation of *ProActive* is applied; if the call is towards a group of objects, then the semantic of group communications is applied. The rest of the chapter defines this semantic.

4.2 Application Programming Interface

The *ProActive* group API provides a valuable basis for building parallel programs. Its design goals aim to ease the construction of parallel applications, separating the concerns, thus helping to solve the problem of developing large applications; all that, maintaining an elegant and fully object-oriented syntax.

4.2.1 Group creation

Groups are created using the static method:

```
ProActiveGroup.newGroup("ClassName", ...);
```

The superclass or the interface common for all the group members has to be specified, thus giving the group a minimal type. Groups can be created empty and existing active objects can be added later as described in Section 4.2.2. Let us take a standard Java class:

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

   // Solution 1:
   // create an empty group of type "A"
A ag1 = (A) ProActiveGroup.newGroup("A");
```

Non-empty groups (groups and their members) can be built at once using two additional parameters: a list of parameters required by the constructors of the members and a list of nodes where to map those members. In that case the group is created and new active objects are constructed using the list parameters and are immediately included in the group. The nth active object is created with the nth parameter on the nth node. If the list of parameters is longer than the list of nodes (i.e. we want to create more active objects than the number of available nodes), active objects are created and mapped in a round-robin fashion on the available nodes. Remotely creating the objects at the same time as the group itself is a powerful deployment ability that reinforces dynamicity and avoids numerous and repetitive adding operations. Here are examples of some group creation operations:

```
   // Pre-construction of some parameters:
   //  For constructors:
Object[][] params = {{...} , {...} , ... };
   //  Nodes to identify JVMs to map objects
Node[] nodes = { ... , ... , ... };
```

```
    // Solution 2:
    // A group of type "A" and its members are created at once,
    // with parameters specified in params, and on the nodes
    // specified in nodes
A ag2 = (A) ProActiveGroup.newGroup("A", params, nodes);

    // Solution 3:
    // A group of type "A" and its members are created at once,
    // with parameters specified in params, and on the nodes
    // directly specified
A ag3 = (A) ProActiveGroup.newGroup("A", params,
      {"rmi://laurel.inria.fr/Node1", "rmi://hardy.inria.fr/Node2"});
```

The deployment of a group of activities can benefit a lot from the Virtual Node abstraction presented in Section 3.2.1. Groups and their active objects can be created using a virtual node instead of an array of nodes. First, the virtual node is activated (`activateMapping()`), then it is possible to use it into a group-and-objects creation. See the following example:

```
    // Activates the mapping for the VirtualNode vn
vn.activateMapping();

    // Solution 4:
    // A group of type "A" and its members are created at once,
    // with parameters specified in params, and on the nodes
    // of the virtual node (specified in the XML deployment file)
A ag4 = (A) ProActiveGroup.newGroup("A", params, vn);
```

Those creation processes assumes that the number of active objects to create is known. The number of created objects depends on the `params` size. In some cases, the number of objects we want to create is not fixed and depends on the number of available nodes. So, primitives are provided to build activities depending on `vn` size and not on `params` size. In those cases one object is created by node and each object is created with the same parameters. The `params` parameter is a one dimensional array containing the parameters used to create each active object.

```
    // Pre-construction of some the common parameters
    // Note that params is now a 1Dimensional array
Object[] params = { ... , ... , ... };

    // Solution 5:
    // A group of type "A" and its members are created at once,
    // with the same parameters specified in params, and on all
    // the available nodes
A ag5 = (A) ProActiveGroup.newGroup("A", params, vn);
```

Elements can be included into a typed group only if their class implements the interface, or equals or extends the class, specified at the group creation: the classes of all the members of a group have a common ancestor. Note that we do allow and handle *polymorphic* groups. For example, an object of class `B` (`B` extending `A`) can be included into a group of type `A`. However, only the methods defined in the class `A` can be invoked on the group.

The main limitation of the group construction is that the specified class of the group has to be *reifiable*, according to the constraints imposed by the Meta-Object Protocol of *ProActive*: the type has to be neither a primitive type (`int`, `double`, `boolean`, etc.), nor a final class, in which cases, the MOP would not be able to create a typed group object. However, those constraints are easy to explain, to identify, and to check.

4.2.2 Group of `Objects`: a `Collection` and a `Map`

The typed group representation presented in the preceding section corresponds to the functional view of groups of objects. In order to provide a dynamic management of groups, a second and complementary representation of a group has been designed. In order to manage a group, this second representation must be used instead. This (second) representation follows a more standard pattern for grouping objects: the interface `Group` extends the Java `Collection` interface which provides management methods like `add`, `remove`, `size`, etc. Those group management methods feature a simple and classical semantic (add in group, remove the nth element, etc.) which provides a ranking order property of elements of a group.

The management methods for a group are not available on the *typed group representation*, but instead, on the *group representation*. It is a design choice among two possibilities: one that would have consisted in using static methods of the `ProActiveGroup` class in order to manage groups, and as such, yielding to just one representation of a group. The other consists in associating to a group two complementary representations, one for functional use only, the other for management purposes only. At the implementation level, we are careful to have a strong coherence between both representations of the same group, which implies that modifications executed through one representation are immediately reported on the other one. In order to switch from one representation to the other, two methods have been defined (see Figure 4.1): the static method of the `ProActiveGroup` class, named `getGroup`, returns the Group form associated to the given group object; the method `getGroupByType` defined in the `Group` interface does the opposite.

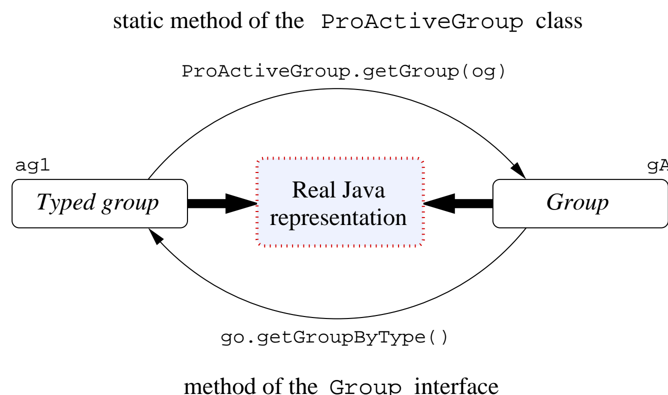


Figure 4.1: *Typed group* and *Group* representations

Below is an example of when and how to use each representation of a group:

```
  // Definition of one standard Java object and two active objects
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA, node);
  // Remember that B extends A
B b = (B) ProActive.newActive("B", paramsB, node);

  // For management purposes, get the representation as a group
  // given a typed group
Group gA = ProActiveGroup.getGroup(ag1);

  // Now, add objects to the group:
  // Note that active and non-active objects may be mixed in groups
gA.add(a1);
gA.add(a2);
gA.add(b);
```

```
    // A new reference to the typed group can also be built as follows
A ag1new = (A) gA.getGroupByType();
```

Notice that groups do not necessarily contain only active objects, but may contain standard Java objects as members. The only restriction is that their type must be compatible with the class of the group. We will see in the next section the implication of such heterogeneous groups on the management of communications towards group elements.

The `Group` interface also defines most of the methods of the `Map` interface. `Group` does not directly extend `Map` because it is impossible to extend both `Collection` and `Map` interface: the methods `remove` are incompatibles[1]. A `Map` is an object that maps keys to objects. A map cannot contain duplicate keys; each key can map to at most one value. It allows to *name* the object we put in a group, and to find it back with its unique name. See the example below:

```
    // Creates a new object
A a3 = (A) ProActive.newActive("A", params, node);

    // Adds a3 in the group in the Map fashion with a unique key
gA.add(b,"MyFavoriteObject");

    // Retrieves the a3 object with the key
A a3new = gA.get("MyFavoriteObject");
```

The collection interface provides ordering, while the Map interface provides indexing. Any object in a group has a rank, but not necessarily a key.

Group membership is dynamic. No consistency is assured between a group and its possible copies. Actually we consider that a copy of a group is an entirely new group (references to members are copied). In that way, we free ourselves from the consistency problem.

4.2.3 The communication is a method call

A method invocation on a group has a similar syntax to a standard method invocation:

```
Object[][] params = {{...} , {...} , ...};
Node[] nodes = {... , ... , ...};
A ag = (A) ProActiveGroup.newGroup("A", params, nodes);

    // A group communication:
ag.foo();
```

Of course, such a call has a different semantic which is as follows: the call is propagated to all members of the group using multithreading (further information are exposed in Section 5.2.1). Like in the *ProActive* basic model, a method call on a group is non-blocking and creates a transparent future object to collect the results. A method call on a group yields a method call on each of the group members. If a member is a *ProActive* active object, the method call will be a *ProActive* call and if the member is a standard Java object, the method call will be a standard Java method call (within the same JVM).

For example, Figure 4.2 presents a one-way group communication on the group `ag`. `ag` is composed of a remote active object a1, a local active object a2, and a standard Java object a3. The one-way call is asynchronously transmitted to each object regarding their location and their form (active or not). On the figure, the *call 1*, addressed to the remote active object, is a remote asynchronous call. The *call 2*, addressed to the local active object, is a local asynchronous call.

[1]The `remove` method of the `Collection` interface returns a boolean that represents if the collection changed after the call. The `remove` method of the `Map` interface returns the removed object. As Java does not allow method overloading with methods returning result with different types, those two methods are incompatibles.

Finally the *call 3*, addressed to the standard Java object, is a standard method invocation without any extra-service. The Group communication adapts each communication in a very fine way.

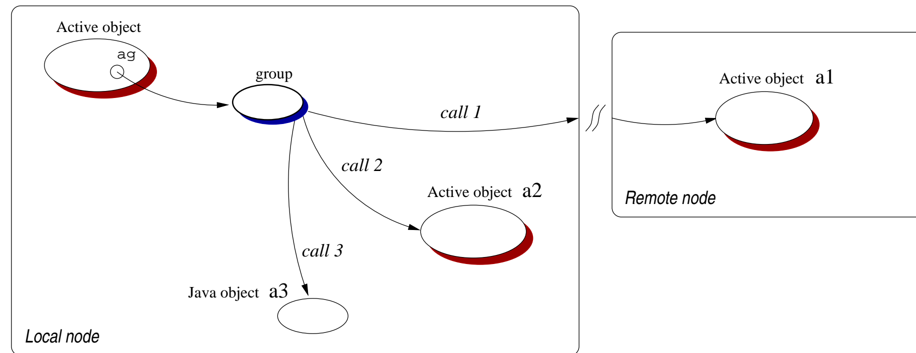


Figure 4.2: One-way method call on a group

The parameters of the invoked method are broadcasted to all the members of the group. As described in Section 4.2.6, another semantic is available in order to scatter the parameters to the group members instead of broadcasting them.

Like the Meta level hides the remote access for a basic *ProActive* method call (presented in Section 3.2.2), the Meta level also hides the multi-communication in a *ProActive* group communication.

4.2.4 Group of *futures*

A particularity of this group communication mechanism is that the *result* of a typed group communication *is also a group*.

Given the following code:

```
  // A method call on a group, returning a result
V vg = ag.bar(); // vg is a typed group of "V"
```

As shown in Figure 4.3, the result group is transparently built at invocation time, with a future for each elementary reply (in case of asynchronous call of course). It will be dynamically updated with the incoming results, thus gathering results. If one result is an active object (with a remote access), only the reference to this active object is send to the result group. Nevertheless, the result group can be immediately used to execute another method call[2], even if all the results are not available (more details come in the *wait-by-necessity* paragraph in the next section). The transformation of the result typed group into a group of object of type `V` in this example, is also immediately available (through the method `getGroup()` presented in Section 4.2.2).

The ranking order of elements in a group is a property that is kept through a method invocation: the nth member of a result group (i.e., of `vg`) corresponds to the result of the method executed by the nth member in the calling group (i.e., of `ag`). We will see later, in Section 4.3.2, that another property is maintained between the group onto which the call is performed and the group of corresponding results: the hierarchical structure. A result group has an identical form to the caller group.

As previously explained, groups whose type is based on final classes or primitive types cannot be built. So, the construction of a dynamic group as a result of a group method call is also limited. Consequently, only methods whose return type is either void or is a *reifiable type*, in the sense

[2] This call will be either a standard call or a *ProActive* remote call, depending of the real type of results

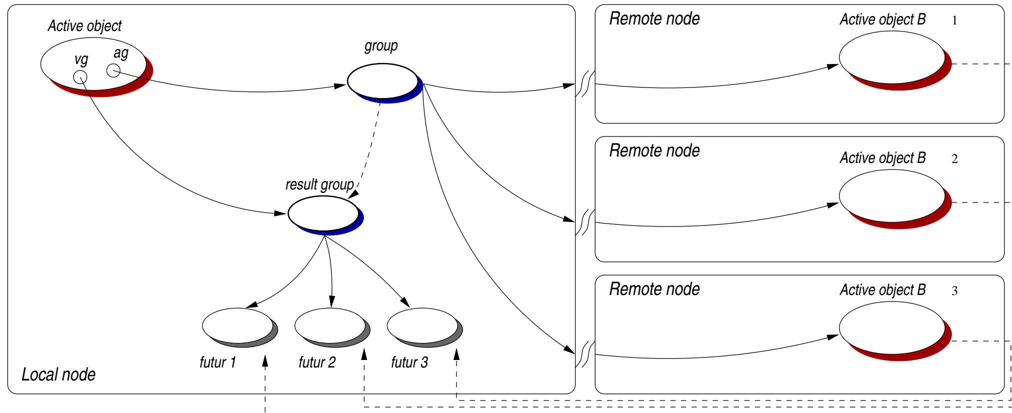


Figure 4.3: Method call on a group, with results

of the Meta Object Protocol of *ProActive* (see above), may be called on a group of objects; otherwise, they will raise an exception at runtime, because the transparent construction of a group of futures of non-reifiable types fails.

4.2.5 Synchronization

Once again, depending on the method signature, the communication scheme is automatically chosen. Let us remember the three communication schemes; one-way, asynchronous, and synchronous; and see their behavior in a group context.

- A one-way group communication is performed in an asynchronous way with no generation of a group of results. The caller blocks until each call reaches its receiver, it is a kind of *generalized rendez-vous*; generalized to the context of group communication.
- The asynchronous group communication follows the semantic of a one-way call (asynchronism and rendez-vous). The difference is that a group of results is created at the early beginning of the rendez-vous step. The creation of the result group over, as well as the rendez-vous, the activity of the caller is resumed (possibly before all the results have returned).
- Finally, a synchronous group communication is quite similar to an asynchronous communication, except that the caller blocks until all the messages are sent (the rendez-vous) and all results have returned.

The rendez-vous ensures a FIFO ordering of message delivery. The group communication extends also the synchronizations applied on the result after a basic *ProActive* method call. The *Wait-by-necessity* mechanism has evolved to be group-compliant and new primitives to control the synchronization have been introduced. Of course such mechanisms are only useful with an asynchronous method invocation (that is the most frequent case in our framework).

Whatever be the communication semantic, the method invocation on a standard Java object, member of a group, is always synchronous and never return a future. This because Java object are not able to express asynchronism.

Wait-by-necessity

The result group is immediately available to execute a method call, even if not all results are arrived on the caller side. In that case the *wait-by-necessity* mechanism implemented in the *ProActive* group communication system is used: the call is applied on the already returned replies (i.e.

group members) and if some replies are still awaited, then, the caller blocks. Then, as soon as one reply arrives in the result group, the method call on this result is executed: in this way, the asynchronism is pushed further. In the code below, a new `f1()` method call is automatically triggered as soon as one reply from the call `vg = ag.bar()` comes back in the group `vg`.

Eventually, the instruction `vg.f1()` completes when `f1` has been called on all members. It means that the caller continues its activity only when the call to `f1` has reached every member of `vg`. Consequently, the activity may not resume before the call to `bar` has *totally completed* meaning that all the results of the call `ag.bar()` have returned into `vg`.

```
  // A method call on a group, returning a result
V vg = ag.bar();
  // vg is a typed group of "V": operation below is also
  // a collective operation triggered on results
vg.f1();
```

Synchronization primitives

To take advantage with the asynchronous remote method call model of *ProActive*, some new synchronization mechanisms have been added. Static methods defined in the `ProActiveGroup` class enable to execute various forms of synchronization. For instance: `waitOne`, `waitN`, `waitAll`, `waitTheNth`, etc. Here are examples:

```
  // A method call (with result) on a typed group
V vg = ag1.bar();

  // To wait and capture the first returned member of vg
V v = (V) ProActiveGroup.waitAndGetOne(vg);

  // To wait all the members of vg are arrived
ProActiveGroup.waitAll(vg);
```

This explicit control gives a finer (and stronger) control to the programmer, no more at the method level but at the group level. It also may allow avoiding unnecessary waits. For instance, in a context where objects of a group are concurrent workers to achieve a same task, the interest is to obtain the result as quick as possible, with no matter about the identity of which object has returned the result. So methods like `waitAndGetOne` or `waitAndGetN` may be really useful. In any case, all calls are executed and all results returned, even if the programmer needs only the first or the nth results.

4.2.6 Broadcast vs. scatter

Regarding the parameters of a method call towards a group of objects, the default behavior is to broadcast them to all members. But sometimes, only a specific portion of the parameters, usually depending on the rank of the member in the group, may be really useful for the method execution, and so, the bigger parts of the parameter transmissions are useless: it is quite inefficient. In other words, in some cases, there is a need to transmit different parameters to the various members.

Give up the benefit of group communication in term of expressiveness and performance and perform a set of point-to-point communications to achieve such kind of method invocation would be bad. On the contrary, the group communication must provide a way to scatter the parameter(s) of a method call between the members of the group. A common way to achieve the scattering of a global parameter is to use the rank of each member of the group to select the appropriate part that a member should get to execute the method. There is a natural translation of this idea inside the group communication mechanism: *the use of a group of objects in order to represent a parameter of a group method call that must be scattered to its members*.

A *one to one* correspondence between the nth member of the parameters group and the nth member of the group is obtained by the ranking property already mentioned in Section 4.2.4.

Like any other object, a group of parameters of type `P` can be passed instead of a single parameter of type `P` specified for a given method call. The default behavior regarding parameters passing for a method call on a group is to pass a deep copy of the group of type `P` to all members [3]. Thus, in order to scatter this group of elements of type `P` instead, the programmer must apply the static method `setScatterGroup` of the `ProActiveGroup` class to the group. In order to switch back to the default behavior, the static method `unsetScatterGroup` is available.

The control of diffusion (broadcast) and distribution (scatter) is very fine. It can be specified parameter by parameter. Non-group object are always broadcasted. As presented in the code below, and illustrated in Figure 4.4, a distribution and a diffusion of data can be performed in the same group communication.

```
  // Broadcast the object a and the groups bg and cg to all the
  // members of the group ag:
ag.foo(a, bg, cg);

  // Change the distribution mode of the parameter group cg:
ProActiveGroup.setScatterGroup(cg);

  // Broadcast the object a and the group bg but
  // scatter the members of cg onto the members of ag:
ag.foo(a, bg, cg);
```

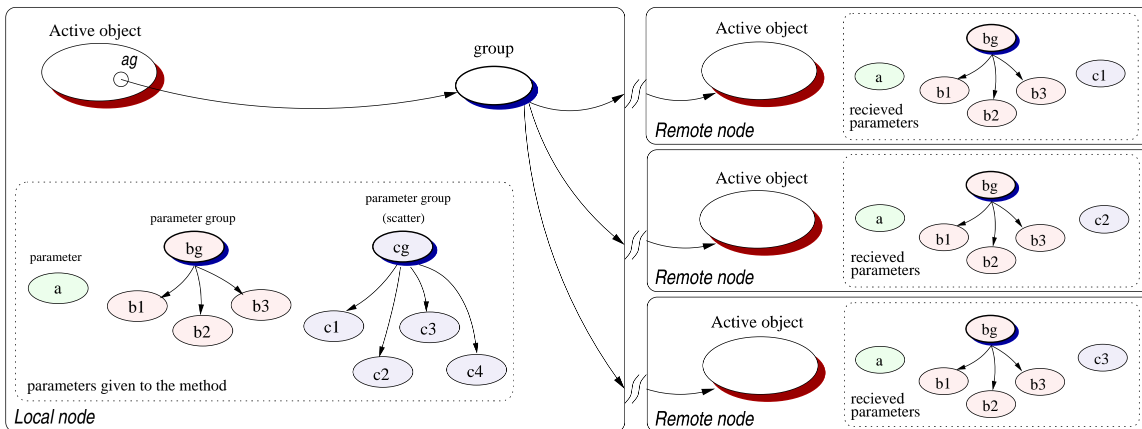


Figure 4.4: Scattered parameters

Notice that, should the parameter group be bigger than the target group; the excess members of the parameter group will be ignored. Conversely, should the target group be larger than the size of the parameter group, then the members of the parameter group will be reused (i.e. sent more than once) in a round-robin (cyclic) fashion.

Note that this parameter dispatching mechanism is in many ways a very flexible one. It provides:

- automatic sending of a group to all members of a group (default),
- the possibility to scatter groups in a cyclic manner (`setScatterGroup`),

[3] If the members of the group of type `P` are in fact active objects or groups, then only copies of the stubs are done. Indeed, the group collecting such members does not effectively contain a copy of those active objects, but only references to them.

- the possibility to mix non-group, group, cyclic-scatter group as arguments in a given call.

All of this is achieved without any modification to the method signature.

4.2.7 Operation semantics on result group

By default, there is absolutely no distinction between a group, newly created by the `newGroup` method, and a result group dynamically created by a method invocation applied on an already existing group.

Method invocation

However, regarding method invocation semantic, one may want to introduce divergences. Indeed, result group may be composed by one or many group members that are futures not yet updated. The behavior to adopt in front of method invocations on not updated futures fall in three categories. Given the following instructions:

```
   // ag is a typed group
V vg = ag.bar();
...
vg.gee();
```

1. The "*Target group order*" strategy: Execution of `gee()` is triggered in the order of the results in the group, as soon as results return, one after the other. The execution is deterministic.
2. The "*Sequential return order*" strategy: Execution of `gee()` is triggered in the order of returning result (future's update), one after the other. The execution is not deterministic.
3. The "*Return order in parallel*" strategy: Execution of `gee()` is triggered in the order of returning result, potentially all simultaneously. The execution is not deterministic and introduces parallelism.

By default, this last strategy is used. It agrees the most to the non-result group communication semantic and thus provides a more uniform framework. Section 8.1 presents extensions of the API that allow to express any other strategies.

Reduction

A *reduce* operation combines the elements provided in a group, using a specified operation, and returns the combined value. The typed group API does not provide a standard primitive to achieve such operation, but prefers to leave to the programmer the care to implement it. We assume two ways to achieve a reduce operation, a local and sequential one, and a possibly remote and parallel one. Here are their description:

- The sequential combination may be achieve by an iteration on the group members. Members are combined one after the other thanks to a static method, possibly extern from the class of the objects. For instance, given `ag` a typed group of `A` and `combine` a user-defined static method, the following code performs a reduce operation:

  ```
     // reduce may be of any type the programmer wants
  reduce = null;
  Iterator it = ProActiveGroup.getGroup(ag).iterator();
  while (it.hasNext()) {
    reduce = A.combine(reduce, it.next());
  }
     // reduce contains the result of the reduction
  ```

  If the group is a result group that contains futures, the execution of the method `combine` will trigger wait-by-necessity. The execution is deterministic.

- The parallel combination achieve a reduction using a group communication. In this case, the class of a typed group has to define a method which takes in parameter an user-defined "storage" object. The invocation of a combination method on the group with a storage object as parameter achieves reduction in a parallel fashion. Here is an example, with a typed group `ag` and a combination method named `combine` defined in the class `A`:

```
  // the group communication
  // storage was locally initialized by the programmer
ag.combine(storage);
  // storage contains the result of the reduction
```

  When the call is over, the storage object contains result of the reduce operation. This way to achieve a reduce is most of time more efficient than the sequential version because combinations may be done simultaneously following the result return order. However, the programmer have to ensure that simultaneous reading and writing accesses on the storage object value are safe.

In both cases we assume that we do not benefit from binomial propagation and combination of the results that many library for collective communication may provide. The group behavior component we are currently adding in the *ProActive* library is a third integrated solution to perform a reduce operation; it is presented in Section 8.1.

4.3 Advanced group features

In addition to the regular use of group, method invocation and group management, the mechanism is extended in order to support more advanced abilities. Four of them look quite fundamental for a complete group communication system: the error handling mechanism, the hierarchical composition of groups, the remote access to a group service, and a processing-based group activity.

4.3.1 Errors and exceptions

The failure model provides a mechanism to handle the failure of a method execution or of a method invocation. In the Java framework failures and errors are expressed with `Exceptions`. We can distinguish two kinds of exceptions: the exceptions raised during the method execution and the exceptions raised by the system or the middleware. The first exceptions may be expected while the second may not. The group communication system manages both exceptions in a unified manner.

The group communication system does not remove a failed member from a group. It is the programmer's responsibility to do that. Because it is impossible to guess how an application should react regarding an exception, the final treatment always yield to the programmer. The group communication system assumes that itself is not able to solve the problem (i.e. determine if the member is lost, unavailable for a moment, or if the exception is an expected behavior). It only avoids the propagation of errors; i.e. method invocations on a "failed" member.

In group communication, exceptions are not directly propagated. There are two main reasons for that. Firstly, because groups are transparent for the functional aspect (method invocation), the language does not expect a particular behavior dedicated to group. As soon as one exception is raised, the system will stop the call. This is not what we expect: we want the call to be finished by communicating with all members. Secondly, it is impossible to choose only one exception to propagate in case of multiple exceptions occurred. The language allows only one exception to be raised and caught. This is not satisfying for a group communication.

Asynchronous calls

In case of an asynchronous call, the caller thread may have already left the `try/catch` statement when the exception occurs, so the standard way to manage exception is no longer acceptable. We need a structure to store raised exception in order to inspect them at any time (before the call completes, or after). Given `ag` a typed group of `A`:

```
A ag = (A) ProActiveGroup.newGroup("A",params,nodes);
```

If a member of a group communication raises an exception, this exception is stored in the result group at the exact place where should be the awaited result. Exceptions are stored in an object named `ExceptionInGroup` that contains also a reference to the object on which we try to invoke a method and which triggers the exception. This allows identifying the object that possibly failed and eventually remove it. See the following method invocation on the group `ag`:

```
V vg = ag.bar(); // vg may contain exceptions
```

The typed group `vg` may contain exceptions. To examine those exceptions, the method `getExceptionList()` returns an object `ExceptionList` that extends `RuntimeException` and implements the `List` interface. The `ExceptionList` is a `List` of `ExceptionInGroup`. An `Iterator` allows to iterate on each exception and observe them or perform some treatments as presented below:

```
   // Gets the Group interface
Group gV = ProActiveGroup.getGroup(vg);

   // Retrieves the exceptions list
ExceptionList el = gV.getExceptionList();

   // Iterates on the exceptions
Iterator it = el.iterator();
while (it.hasNext()) {
   ...  // Treatment
}
```

A method invocation on a group ignores the members that are `Exceptions`. The call is only propagated to valid members. In order to maintain the ordering property a `null` reference is placed in the result group at the same index than the exception member. As for the exception members, a method invocation is not relayed on null members.

```
   // The call to f2 is not relayed on the exceptions
   // contained in vg. wg will contain null members.
W wg = vg.f2();
```

The method `purgeExceptionAndNull()` removes the null and exception members. The lost of those unnecessary members breaks the ordering property. It can be compared to the `trimToSize()` method of the `ArrayList` and `Vector` classes that trims the capacity of those collection instances to be the list's current size. Beware, a call to `purgeExceptionAndNull()` impacts the ranking order of the group.

```
Group gW = ProActiveGroup.getGroup(wg);
gW.purgeExceptionAndNull();
gV.purgeExceptionAndNull();
   // now, vg and wg do not contain anymore null or exception members
```

Figure 4.5 summarizes the presented operations.

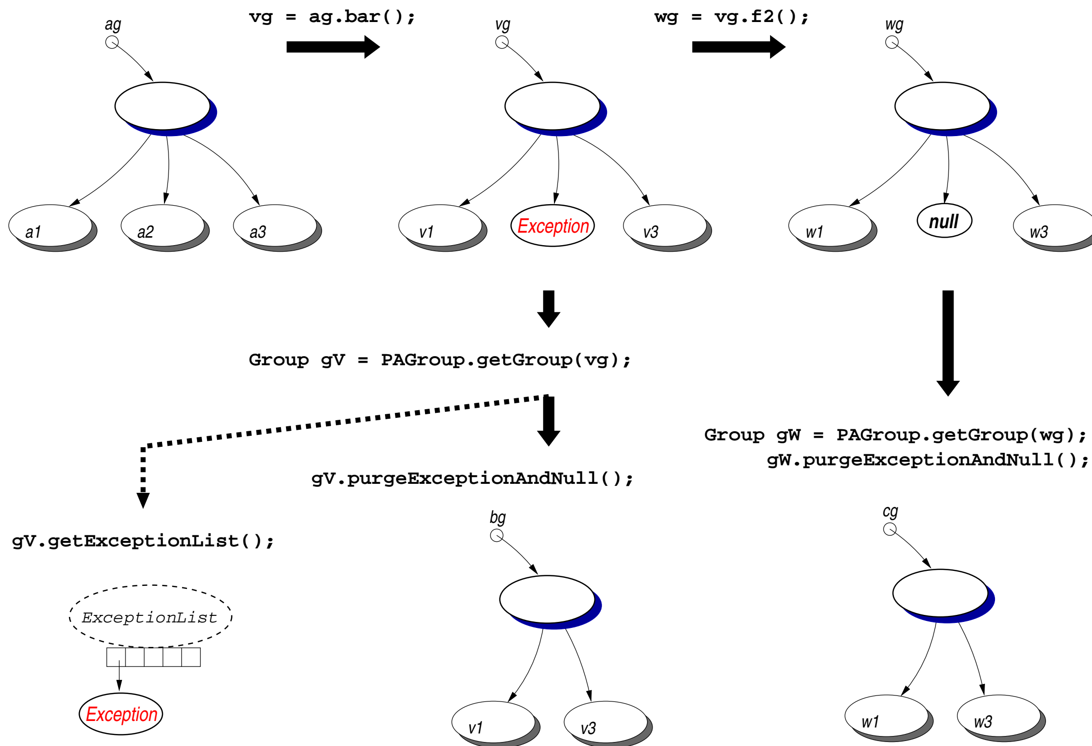


Figure 4.5: Exception mechanism of an asynchronous method call on group

Synchronous calls

The mechanism for synchronous call is identical to the mechanism presented above for asynchronous call. Even with a synchronous execution, no exception is raised. They are stored in the result group to be potentially inspected later with an `ExceptionList` object. The reason is still the same: we have to handle possibly many and different exceptions and we do not want to favor one regardless the others.

In consequence, the `try/catch` statement surrounding the invocation of a method that can throw exception will never be reached. The exception mechanism of group communication intercepts the raised exceptions and put them in the result group. However the `try/catch` statement has to be written, because the Java language forces the exception to be caught (or thrown). This is one bad effect of the group transparency.

Another way would have been to throw the `ExceptionList` object. But this is not the expected exception. The `try/catch` statement problem would have been the same: it would have never been reached.

One-way calls

In the case of one-way method call, the exception handling mechanism slightly differs from the mechanism deployed in asynchronous and synchronous method call. There is no longer a structure (the result group) able to store the raised exceptions. Exceptions are embedded into `ExceptionInGroup` and inserted in an `ExceptionList` similarly to the (a)synchronous method, but the `ExceptionList` is systematically raised if it contains at least one exception. `ExceptionList` extends `RuntimeException`, so it is not an obligation to write a `try/catch` block. In the following example, the `ExceptionList` is caught to be analyzed:

```
try {
  ag.foo();
}
catch (ExceptionList el) {
  Iterator it = el.iterator();
  while (it.hasNext()) {
    ... // Treatment
} }
```

If the programmer chooses to write a `try/catch` block, he or she has to keep in mind that a one-way call has an asynchronous semantic. An exception may be raised when the block is already passed. For practical purposes the `try/catch` block should be placed in a synchronous method that encapsulates the group invocation to be able to catch all the exceptions.

Figure 4.6 exposes the behavior in a one-way call context:

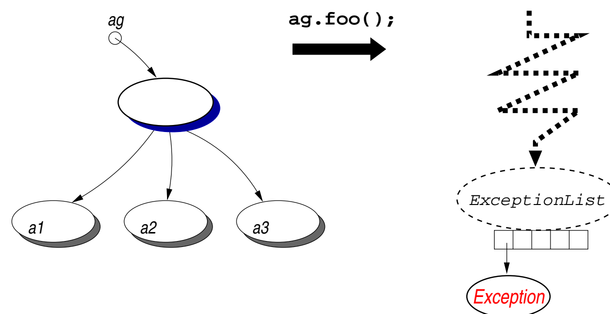


Figure 4.6: Exception mechanism of a one-way method call on group

4.3.2 Hierarchical group

To build large applications, we provide the concept of hierarchical group: a group of objects that is built as *a group of groups*. This mechanism may help in the data structuring of the application and makes it more scalable.

A hierarchical group is easily built by just adding group references to a group. This operation is very simple because groups are typed objects, and thus subject to be added into another typed group. Compatible type is the only condition to be part of a group. Here is an example showing the creation of a hierarchical group, (Figure 4.7 illustrates the operations):

```
  // Two groups
A ag1 = (A) ProActiveGroup.newGroup("A",...);
A ag2 = (A) ProActiveGroup.newGroup("A",...);

  // Get the group representation
Group gA = ProActiveGroup.getGroup(ag1);
  // Then, add the group ag2 into ag1
gA.add(ag2);
  // ag2 is now a member of ag1
```

Note that one can merge two groups, rather than add them in a hierarchical way. This is provided through the `addMerge` method of the `Group` interface. For instance, the instruction:

```
  // Add the members of ag2 into ag1
gA.addMerge(ag2);
```

adds all the members of a group into another one as shown in Figure 4.8.

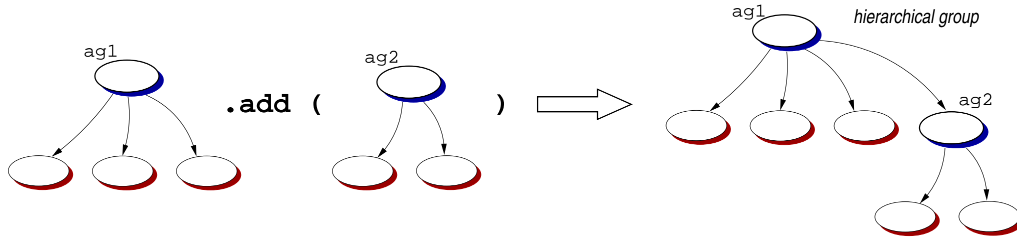


Figure 4.7: The add method

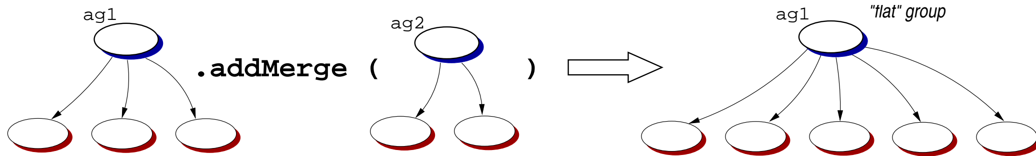


Figure 4.8: The addMerge method

As seen previously, a group of results has the same form of the caller group. This is ensured by the property: the nth member of the result group corresponds to the result of the method executed by the nth member in the caller group.

This correspondence remains true in the case of hierarchical groups. As the result of a method call applied on a group is also a group, the members which are a group return a group as result. Finally, the result group of a method invocation on a hierarchical group is a hierarchical group with the exact same form as shown in Figure 4.9 (group on the left is the result of a method call invoked on the hierarchical group on the right: `vg = ag.bar()`).

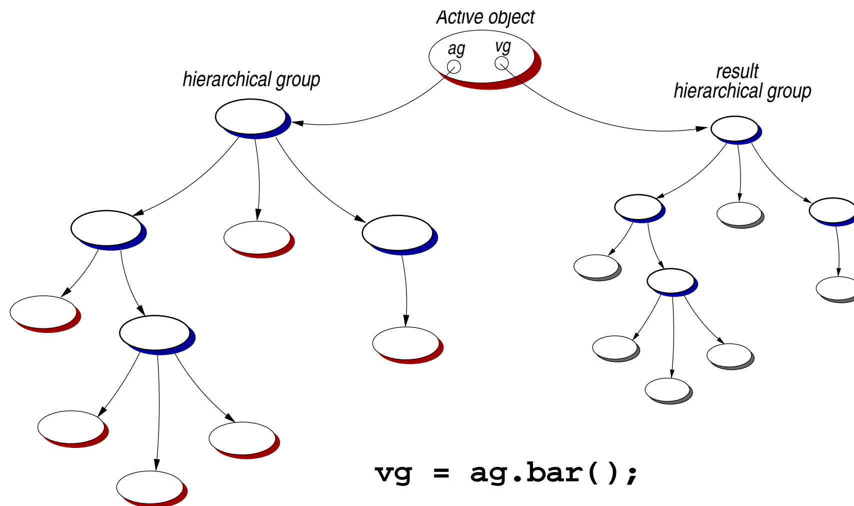


Figure 4.9: Hierarchical groups

With hierarchical groups, the rendez-vous is still the same than with regular groups, but has some special implications. The caller always waits for the call has reached all members of a group to continue. The fact is that members of a group added into another are not members of the root group. In the example, members of `ag2` will never be considered as members of `ag1`. However as the rendez-vous ends when all the members of the root group have received the method call,

and as the reception for the sub-groups is also subject to a rendez-vous, the activity of the caller is resumed as soon as all members and all sub-group members have received and queued the method call[4]. The rendez-vous can be said “recursive”[5].

Depending on the parameters they receive (scatter or not), communication schemes (scatter or broadcast) of the root group and its sub-group may differ. In that way, one can combine the schemes in order to build a more complex scheme. For instance the root group may scatter its data between its members, and some of its sub-groups may broadcast their piece of data to all their members.

4.3.3 Active group

Groups are local. Because one may want to access them remotely, we have to provide a way to achieve this. A group remotely accessible looks like a service: a message is first addressed to the service, and then forwarded to the group members.

ProActive provides an easy way to transform any object into a remotely accessible object. As such, a typed group may be turned into an active object. We name it an *active group*. Thus, a group earns all the abilities of an active object. It becomes among other properties, remotely accessible, served by a FIFO policy, and subject to migration.

There are two manners to obtain an active group:

1. The *Instantiation-based* approach: a Java class is directly instantiated to create an active group. The parameters `params` and `nodes` play the same role as previously in the basic group context: they are used to build the group members.

   ```
     // Pre-construction of parameters
   Object[][] params = {{...} , {...} , ... };
   Node[] nodes = { ... , ... , ... };

     // Creation of an active group
   A active_ag =
       (A) ProActiveGroup.newActiveGroup("A", params, nodes, node);
   ```

   If the `node` parameter is `null` or not specified, the active group is created locally. Otherwise, if `node` refers to a remote JVM the active group is created and activated in this designed node.

2. The *Object-based* approach: this is more dynamic. It transforms an already existing typed group into an active group.

   ```
     // Creation of a typed group
   A ag = (A) ProActiveGroup.newGroup("A", params, nodes);

     // The typed group turned active
   A active_ag = (A) ProActiveGroup.turnActiveGroup(ag, node);
   ```

   If `node` refers to a remote JVM the group is copied to the remote location and turned active.

An active group remotely exposes only its functional interface. The management interface (the `Group` interface) is remotely unavailable. So any management operations have to be done locally to the active group. It is also possible to create another active object which drives the

[4]This behavior differs from the case where a root group member has a reference on a group and relays itself the method invocation to this group (by method override for instance). In that case the rendez-vous only ensures that the method call has reached the group member; there is no guarantee about the propagation of the call on the “sub-group” members.

[5]Implementation details come in Section 5.1.2, but we can notice now that a call on a hierarchical group is triggered by recursive calls on the `reify` methods in charge of the communication semantic.

group for management purpose. By providing the `Group`'s methods, this active object may simulate a remote access to the `Group` interface.

Active groups are particularly effective to communicate with remote clusters. Indeed in a grid environment, a group call departing from a computer in a cluster and addressed to computers of another cluster must use an active group. Otherwise multiple replicated messages would pass through the interconnection network to reach a common destination cluster. Because this is a waste of network resources and because, the interconnection network may be Internet, with no guarantee of Quality of Service and high latency, the use of active group is strongly recommended.

With an active group, a unique "entry point" relays the calls to the members. This property linked with the rendez-vous to communicate with any active object assures that message delivery is totally ordered. Of course, the transmission of a call to this unique entry point is supported by the rendez-vous of *ProActive*; thus successive calls to such a group will be totally ordered. However, within an asynchronous call, the generalized rendez-vous defined by group communication (see Section 4.2.5) is no longer maintained. An active group ensures that the method invocation is received by the entry point but not necessarily transmitted to the members. Indeed, this generalized rendez-vous has become useless since the group exposes a unique entry point and subsequently assures a reliable delivery. Nevertheless the communication semantic remains unchanged with synchronous method invocation: the caller waits for all results are returned to resume its activity.

Migration of a typed group is performed with the static methods `migrateTo` provided by the `ProActive` class. Only the group migrates (with its standard Java object members, but not with the members that are active objects). Standard Java objects have to be migrated with the group because they are not remotely accessible, and in consequence would be lost after the migration. On contrary, the active objects, members of the group, remain in their location. Those objects may be migrated individually using the `migrateTo` methods. Otherwise if their common interface implements a method that triggers a migration, a migration of all members may be triggered by the method invocation on the group. The mobility of the group and also of its members allows for instance to migrate the objects before a cluster shuts down, and thus save the data and the state of the application while the application keeps running.

Finally, like for any other active object, service policy of an active group may be redefined by the programmer, thus giving a total control on the method execution schedule. Of course, the default policy is a FIFO service of the requests. This is done by redefining the `runActivity` method in the class of the group.

Combination of active and hierarchical groups

Combined to hierarchical groups, active groups are a very effective and efficient solution to aggregate clusters in a grid environment. This allows to easily distribute data and activities in a complex hardware structure, as presented in Figure 4.10.

4.3.4 Dynamic dispatch group

In the particular case where groups are used to produce parallelism regardless which data is processed by which group member, the basic behavior of group communication (synchronization and data to member mapping) can be improved in order to best schedule the overall computation. A dynamic dispatching of the group parameter will be achieved, based on the relative observed speed of execution.

The idea is to send more pieces of the parameter data to faster members than to slower members in order to provide a more efficient distribution of the call. The goal is to reduce the time needed to treat a method call on a group by a distribution of method call adapted to the

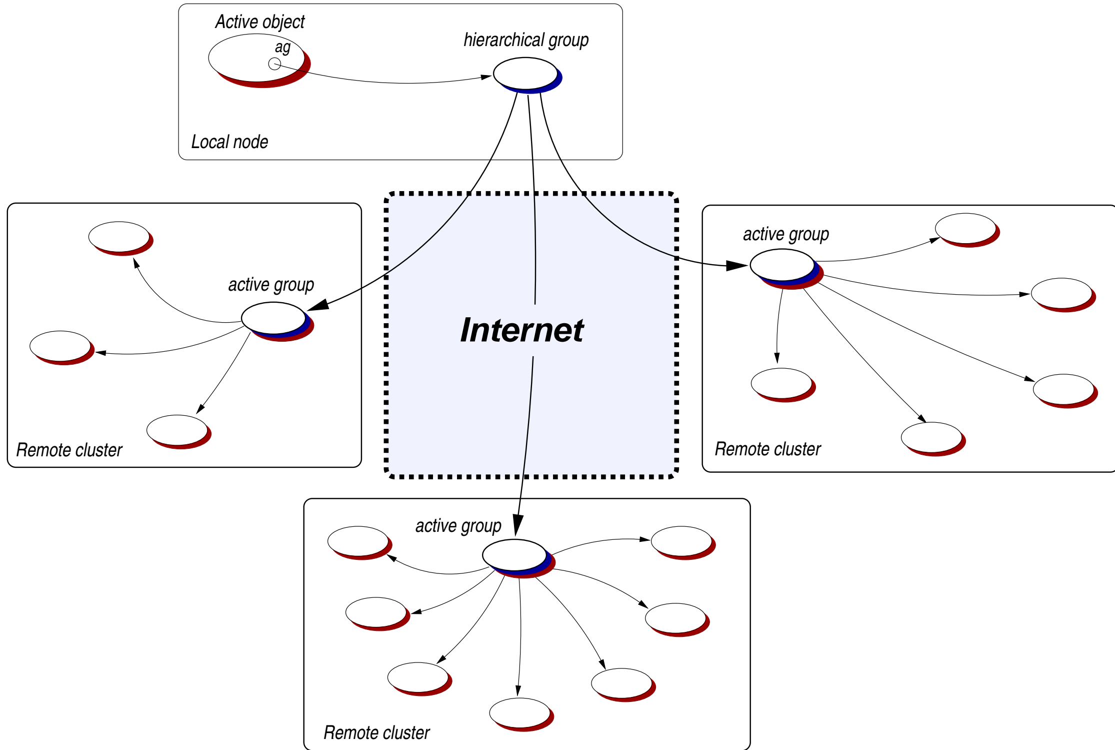


Figure 4.10: Hierarchical and active groups

performances of the group members. The scattering of data remains on the programmer's responsibility, as in standard scatter communication (build the scattered parameter groups). What will mainly differ in a *dynamic dispatch group* is the dispatching of parameter to group member. The cyclic manner to dispatch parameter to group members (presented in Section 4.2.6) does not consider the difference in term of performance of the members. In a dynamic dispatch group, the requests are not sent entirely to the group members at the beginning of the call; the requests are first queued on the caller side and then successively sent to a group member available for a computation. As soon as a group member ends a computation it asks for a new request to the caller. By this way, faster workers may serve many requests while slower workers serve each a single request.

Of course, the ranking property ensuring that the nth member of a group parameter will be passed to the nth member of the worker group is no longer maintained. The allocation is not deterministic: it is impossible to predict which parameter will be sent to which worker. Moreover, there is no warranty for the order of service of the requests because different members serve methods at different speed. The service of a request r_1 sent to a slow worker w_1 at time t_1 may end after the service of a request r_2 sent to a faster worker w_2 at time t_2, where $t_1 < t_2$.

Finally, the structure of the result group is no longer based on the group on which the method is invoked. In the dynamic dispatch group context, the construction of the result group is based on the first parameter that is a scatter group [6]. It means that the nth member of the result group is the result of the method invoked with the nth member of the parameter group on a worker.

Dynamic dispatch groups are created with the method `newDynamicDispatchGroup` of the `ProActiveGroup` class. Parameters are similar to those used to build a standard typed group:

[6] Indeed, if there are several groups in the parameters of the call, the first scatter one is considered to lead the call and define the size of the result group.

```
   // Construction of parameters and nodes
Object[][] params = {{...} , {...} , ... };
Node[] nodes = { ... , ... , ... };

   // Dynamic dispatch group creation
A ddag = (A) ProActiveGroup.newDynamicDispatchGroup("A",params,nodes);
```

Of course, communications remain performed in a remote method invocation style:

```
   // A dynamic dispatch group communication
ddag.foo(a,b,sg); // sg is a scatter group
```

At least one parameter of the method invocation has to be a scatter group, otherwise a dynamic dispatch group triggers a standard broadcast operation. In case of multiple scatter groups used as parameter, the first one is arbitrarily chosen to lead the call and the others are scattered in a round-robin fashion.

Figure 4.11 shows how a dynamic dispatch group behaves in comparison with a basic group. To compute N data with M active objects, a basic group needs at least N/M method invocations, and as a consequence, N/M result groups are created (see Figure 4.11 (a)). On the contrary, a dynamic dispatch group requires only *one* method invocation and produces only *one* result group (see Figure 4.11 (b)). Only one result group makes the gathering and the combination of result data easier than it could be possible with basic group communication that requires a combination of the several result groups. The mechanism also ensures that all members will be working until there is no more request to serve, ensuring a more efficient global execution (see request queues of active objects on the figure). It is interesting to notice that active object in dynamic dispatch group got two requests in the queue. At the beginning of the call, two requests are sent to each object, then a new request is sent as soon as the service of a request finishes. It allows overlapping the data transfer over the network: a request is immediately available (no latency) and a new request is transferred while the other is processed.

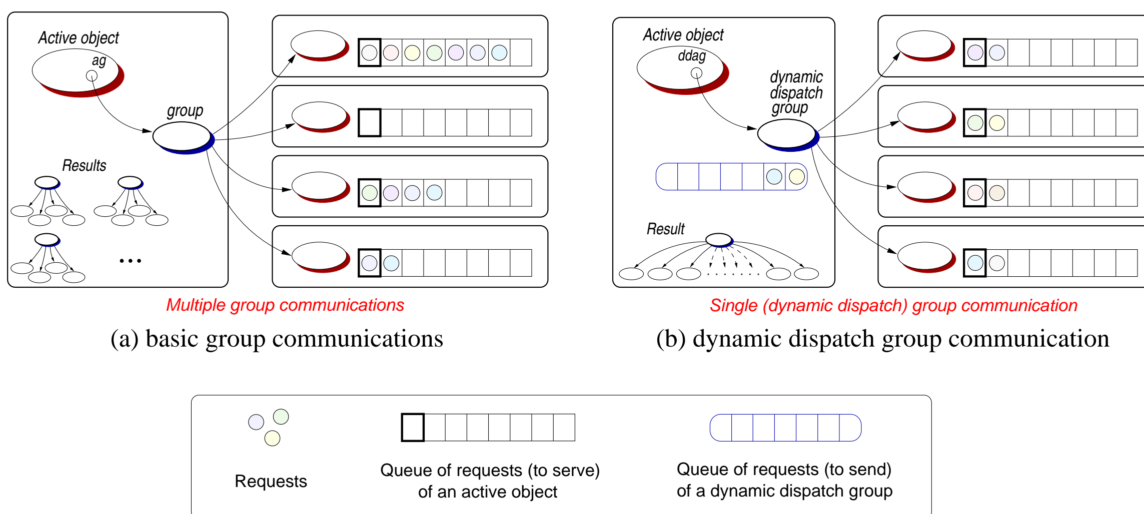


Figure 4.11: Differences between basic groups and dynamic dispatch groups: behavior and usage

Using dynamic dispatch groups, the computations are structured around the data to be processed instead of around the available workers. It fits well with *stateless system* and *embarrassingly parallel problem* solving. Stateless systems have no record of previous interactions and each request has to be based entirely on information that comes with it. A problem of size N is embarrassingly parallel if it is quite easy to achieve a computational speedup of N without any interprocess communication and each process can be given $1/N$ of the computations that can be

independently done. Dynamic dispatch groups are well suited to applications where the amount of data is greatly superior to the amount of workers. For instance, this is the case for SETI@home [SET] or BLAST applications [ANA 04].

Conclusion

Group communication is a crucial feature for high-performance and Grid computing. The group communication system presented here is both simple and very expressive. It provides a transparent, robust, flexible, and easy-to-use framework to build distributed and parallel applications. Through a remote method invocation scheme, it allows automatic choice of the communication semantic. The system handles itself automatic grouping, in a typed manner, and synchronization of the results.

The following table summarizes the properties of the *ProActive*'s typed group mechanism:

Membership	Dynamic
Structure	Open group
Ordering	FIFO ordering (total ordering with active groups)
Reliability	Reliable
User interface	Typed communication

Table 4.1: *ProActive* group properties

Chapter 5

Implementation and micro-benchmarks

The way we have implemented the group communication mechanism depends on the properties we want to obtain and on the existing properties we want to maintain. Our considerations discard some models and give directions to possible implementations. The chosen implementation was subject to optimizations; such optimizations have been tested in order to prove their efficiency, and then, incorporated in the final implementation. This chapter presents the implementation of the presented group communication. It also exposes performances of the system through micro-benchmarks representing basic operations.

The first section of this chapter presents our motivation and explains our choices of implementation. Section 5.2 presents the main optimization points of the group mechanism. After that, in Section 5.3, benchmarks of plain group communications are shown and discussed. Finally, in Section 5.4, a small application (matrix multiplication) shows the benefit from using group communication.

5.1 Implementation details

This section describes the considered alternatives to implement the group communication mechanism. It exposes our choice and then the ways we had to improve the system performances, in term of delay of operation achievement and in term of resources saving.

5.1.1 Motivations

As mentioned in [MAA 03], an approach to implement group communication could be to extend our Java library with support for collective communication through an external library such as MPI. Some works were done in this direction, [CAR 00, GET 99]. However this increases expressiveness at the cost of adding a separate model based on message passing, which does not integrate well with the remote method invocation of an object-based model. Also, MPI was designed to deal with static groups of processes rather than with objects and threads. MPI uses collective communication. Such communication operation, for instance a broadcast, must be explicitly invoked by all participating processes. This requires processes to execute in lockstep, a model that does not fit very well with object-oriented programs which are most of the time multithreaded.

A second solution is to interface *ProActive* with a library dealing directly with a multicast protocol. For instance, [BAN 98] and [ROS 98] propose such kind of service. By stepping in at a lower level than a library like MPI, we gain in flexibility. But once again, because those libraries provide their own API, it breaks the object-based model using remote method invocation. It also

introduces a deployment problem in grid environment: a part of the grid or one network interconnection may not support the multicast protocol.

Finally, a group communication (1→N) can be obtained by the replication of point-to-point communication (1→1) to each member of the group. It is called *multi-unicast*. The *multi-unicast* is frequently disparaged. This model is blamed to be non-optimal in term of performances. The criticism targets not only the time of execution, but also the consumed resources. The required time to send messages is supposed longer because each message needs to be copied in the low-level communication layers in destination to a receiver. As well, this technique is costly in resources: the amount of additional work holds the availability of the processor and needs more memory to record the identical copies of the message to send.

Despite this criticism, the group communication mechanism is built upon an optimized *multi-unicast* technique. A group communication is the replication of N remote method invocations towards N objects. Of course, optimizations added into the group mechanism implementation, achieve better performances than a sequential accomplishment of N individual remote method calls. By not redefining a new communication mechanism, this approach gives a major advantage: it allows maintaining a fully object-oriented model. In addition, this high-level mechanism preserves the properties of basic point-to-point, principally the asynchronism. The ability to adapt to others RMI-like protocol is also maintained.

After considering the integration, service, and interception approaches presented in Section 2.3, we chose the integration approach. It best follows the *ProActive* philosophy, by not introducing centralized points (services) in charge of group communications. Centralized architectures provided by a service approach are especially sensitive to failure[1] and may produce poor performances. The interception approach is very system dependent. This arms adaptability and portability of a group toolkit. In addition to its compliance with the *ProActive* model, a major advantage of the integration approach is the transparency, naturally leading to the typed group communications.

5.1.2 A proxy for the group

As mentioned in Section 3.2.2, the proxy is responsible for the communication semantic. This meta-object maintains a reference to an active object hiding its location (local or remote), is in charge to create the *futures* objects and thus the asynchronism, and is in direct contact with the transport library (RMI, Ibis, HTTP, etc.) through an adapter object.

It was natural to create a new proxy dedicated to group communication. Indeed, group communication is a new communication semantic. This new proxy has to adapt to the group context. Firstly, it has to maintain a reference to not only one object but many, possibly remote, possibly passive. Then, it has to create a complex *future* structure to handle many returns during a method invocation. The asynchronism mechanism is different and has to be reconsidered. Finally the group proxy has to manage a set of remote objects with which the communication may use different transport libraries.

The `ProxyForGroup` class extends the `AbstractProxy` class and implements the `Proxy` interface. The `reify` method defined in the interface is the central point of communication behavior. This method takes a reified method call as parameter, performs the call with the best adapted semantic, and returns the result value of the call. In case of asynchronous calls, the returned value is a *future*. In a group communication context, the result value is a group.

The proxy for group stores references to the member objects. It was unnecessary to create a fully new and redundant multi-semantic and multi-protocol proxy. Communications rely on existing point-to-point communications, i.e. on existing proxies. Consequently the stored references

[1] A service can be constituted of several objects deployed on different nodes, and thus reducing the probability of a total service failure.

are *ProActive* references: the couple stub, plus proxy. The `reify` method of a group proxy propagates the incoming method call by invoking the `reify` method of the proxy of each member. Of course, the `reify` method of group introduces group concerns such as parallel propagation mechanism, complex synchronization, and exceptions handling.

As the proxy stores the references to group members, it is natural that the proxy implements the `Group` interface. So, in addition to communication semantic, the proxy is also responsible for group management. The `ProxyForGroup` class inherits the `add`, `remove`, `get`, `waitOne`, `waitAll`, etc., methods from this interface. The proxy for group is the "real Java representation" of group presented in Figure 4.1. It is also the object returned by the `ProActiveGroup.getGroup` method.

5.2 Features

A prototype implementation [BAD 01] already gave us acceptable results: a group communication produced better achievements compared to the similar action performed with sequential method invocations. Nevertheless, new improvements were added to enforce the efficiency of the mechanism; to make it faster, more scalable, and less resources consuming [BAD 02a, BAD 03].

5.2.1 Thread pool

Using several threads allows sending messages simultaneously. Doing this way, for each group member to which a *ProActive* call instead of a standard Java call must be made, the delays required to make the rendez-vous are recovered and no more added. In order to maintain the *ProActive* method invocation semantic based on rendez-vous, we introduce an additional synchronization. We extend the notion of rendez-vous for group communication: doing this, an asynchronous group communication blocks until the method invocation has reached all group members.

Because group membership is dynamic, a fixed number of threads used to communicate with the group members is not appropriate. Firstly, whatever be the chosen number of threads, the number of group members is subject to growth a lot, and then the threads will become insufficient to ensure a proper delay to perform the group communication. Secondly, even with a very large number of threads the performances may not be optimal. In the case the group members remain in a lower quantity than the threads, many threads are not used. The system is overloaded by those unused threads (more memory used, additional context switch, etc.).

A better approach is to adapt the number of threads depending on the number of group members. So, the amount of threads becomes dynamic as the amount of members in the group. However, a one-to-one *group members to threads ratio* is no more suitable: too many threads will harm performances, particularly in case of large groups, and the network may be flooded.

The best solution is to associate to a group an adaptive pool of threads in which the member/thread ratio may be adapted by the programmer depending of the requirement of its application. The best ratio for efficient group communications may depend on the group size, the size of exchanged data, the frequency of communication, etc. The default value of the ratio is 8. It was chosen empirically to best fit to most of the applications we have tested.

A fixed *number of additional threads* is also present in order to maintain a fixed number of threads if needed. There are two interests of this. Firstly, one may want to maintain a fixed number of threads for a particular purpose: for instance maintaining only one thread in order to emulate a mono-threaded sequential service. To achieve this, the programmer has to set the ratio to 0 and set the additional number to 1. Secondly, additional threads allow to early benefit of multithreading. It means that the programmer needs not to wait for the number of group members to reach the ratio number to obtain a second thread in the pool. This is very useful

because, even small sized groups produce better performances with two or more threads. For instance a group with only 7 members, runs 2 threads if its additional number is set to 1 (1 is the default value). Thus, this group benefits from the multithreading. In summary, we propose what we name "the linear approach": at any moment of the execution, the number of threads involved in a group communication is:

$$if \;\; ratio \neq 0 \;\; : \qquad nbThreads \;\; = \;\; \lceil \frac{nbGroupMembers}{ratio} \rceil \;\; + \;\; additionalThreads$$

0 is a special value for the ratio. It means that the thread pool size is no longer depending on the group size: it is no more dynamic, so:

$$if \;\; ratio = 0 \;\; : \qquad nbThreads \;\; = \;\; additionalThreads$$

The *ProActive* property ensuring that the caller blocks until a call has reached the callee is generalized to the group communications. In a group context, the rendez-vous ensures that the caller blocks until all group members have received the call. A kind of barrier implemented in the thread pool does this job. It is in charge of blocking the caller until all calls were processed by the threads.

Figure 5.1 plots the average time (in milliseconds) spent to perform one asynchronous method invocation depending on the number of objects in a group. The group members are distributed on 16 machines (cluster of Pentium III @ 933 MHz interconnected with a 100 Mb/s Ethernet network). The curves represent the performances depending on the number of threads used to make the calls. The more we used threads the smaller is the delay to make the group communication. The four upper curves are associated with a fixed number of threads. The lowest is associated with a dynamic number of threads. It shows better performance, because the number of threads is (automatically and transparently) at any moment the adequate number needed.

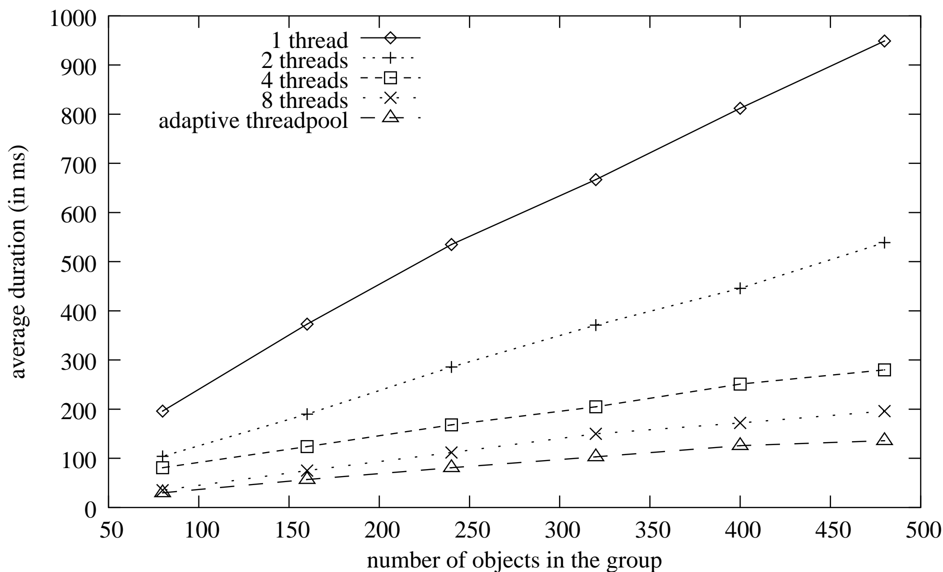


Figure 5.1: Adaptative thread pool

The linear approach is the simplest approach to handle the thread pool dimensioning problem. It works well with average sized groups, and with method requiring short and long time of service. That is why it is the default way to manage the pool size. However it is not fully satisfying. The groups are subjects to grow to very large size, and then, the amount of threads will arm performance by overloading the system. Even with a big member-to-thread ratio, the resulting number of threads may be too big. A solution is to introduce a limit to the number of created threads. Moreover small groups also suffer of a strict linear scheme if the ratio is too high. Not enough threads will serve the method invocations. Some kinds of logarithm-based formulas seem to be the more compliant way to handle highly dynamic groups that quickly vary from few members to many.

As it is very difficult to define a generic approach that could adapt to any group, we choose to let the programmer define the formula to compute the pool size. The linear formula is the default behavior but it can be redefined. By implementing the `ThreadPoolDimensioner` interface one can define an object in charge of the dimensioning. This interface defines the `changePoolSize()` method. The object is then attached to the group and invoked to adapt the thread pool size.

5.2.2 Factorization of common operations

Many operations are common while invoking a method on a group of objects. In a basic approach they were duplicated for each member of the group. Of course, those operations may be factorized to save memory and processing resources.

First is the reification operation. This operation of the Meta Object Protocol transforms the method invocation into a Java object. It involves reflection techniques that are known to be time expensive. In the typed group framework of ProActive, the method invocation being the same for all group members because of the object-oriented syntax, the operation only has to be done once. In addition to the time saved, factorizing reification process also saves memory and CPU resources by avoiding creating many replications of a same method call. With group communication, reification operation runs in *O(1)* instead of *O(n)*. This operation becomes more and more efficient when the size of the group increases.

Second point subject to factorization is the serialization of the method parameters sent during the group communication. In a broadcast operation the same data is marshaled to be serialized and sent to every group member. The serialization operation is located on the caller side and so, subject to factorization. As the Java serialization process is very slow [MAA 01], we want to avoid the repetition of this operation. Our solution is to pre-serialize parameters. Before the RMI mechanism steps in, the parameters (and codebase information) are converted into a byte array. This allows to be more efficiently sent several times by RMI. Only the effective arguments packed in the method call object are serialized. Other fields included in the request are not serialized; especially routing information such as receiver (and sender) identity which is different for each group member. Of course, this *unique serialization* does not apply in the case of a scatter group in which parameters for each member differ. In such situation, the standard RMI mechanism keeps the full responsibility of serialization and sending for all messages to all group members. The unique serialization operation becomes better and better when the size of the parameters of a group communication increases.

Figure 5.2 presents the average time (in milliseconds) spent to perform one asynchronous method invocation depending on the amount of data to send (objects used as parameters). The group contains 80 objects distributed on 16 machines (cluster of Pentium III @ 933 MHz interconnected with a 100 Mb/s Ethernet network). The upper curve exposes the performances without any operation factorized. The curve in the middle plots the performances obtained by factorizing the reification operations. The last curve represents the performances obtained by factorizing the reification operations and the serialization. The gap between the two upper curves represents the time spent by the multiple reifications of the same method invocation. As the number of group members remains the same during the whole experiment, the benefit of the factorized

reification remains the same (one operation instead of 80). This is why the gap remains almost constant. Meanwhile the factorized serialization becomes more effective depending on the parameters size. Joint factorization allows better performances (up to a 3.9 ratio in the example presented in Figure 5.2).

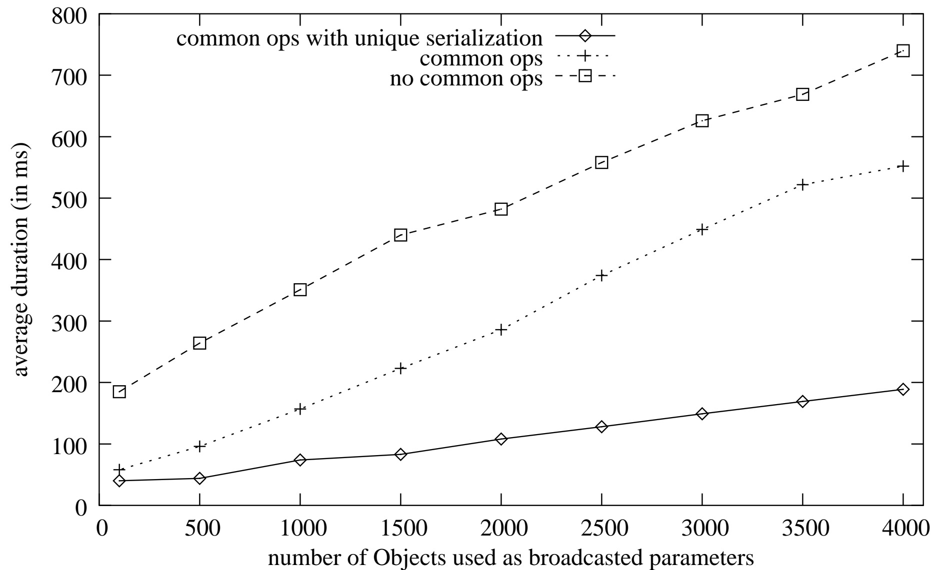


Figure 5.2: Factorization of common operations

5.3 Micro-benchmarks

This section presents performance measurements of basic method invocations on group. It exposes, and discusses, the performances of the three schemes of communications. To have a concrete idea on how the group mechanism behaves, benchmarks are done in the two main modern environments for computing: clusters and Grid.

5.3.1 In a cluster context

First experimentation is measurement of a group communication. Figure 5.3 presents the durations for a group method calls, depending on the number of members in the group. The three curves plot the performances obtained with the three communication schemes: one-way call, asynchronous call, and synchronous call. The invoked methods are empty methods: they do not perform any computation nor take any parameter. They act like a “ping” operation.

Remote objects, members of the group, are distributed one by host. I use a cluster composed of 216 bi-AMD Opteron 64 bits @ 2 GHz and 2 GB of memory, interconnected by a Gigabit Ethernet network. It runs under Debian with a kernel 2.6.9. The Java Virtual Machine is a 1.5.0 by Sun for 64 bits processors.

In each configuration, the method calls trigger a rendez-vous during the concurrent and remote invocations. It means that the delay exposed in the figure represents the duration for the caller activity to be blocked. At the end of this time:

- for a one-way communication, the call has reached all the group members.
- for an asynchronous communication, the call has reached all the group members and the result group has been created containing all the *futures* (some of them may have been already updated with the expected results).

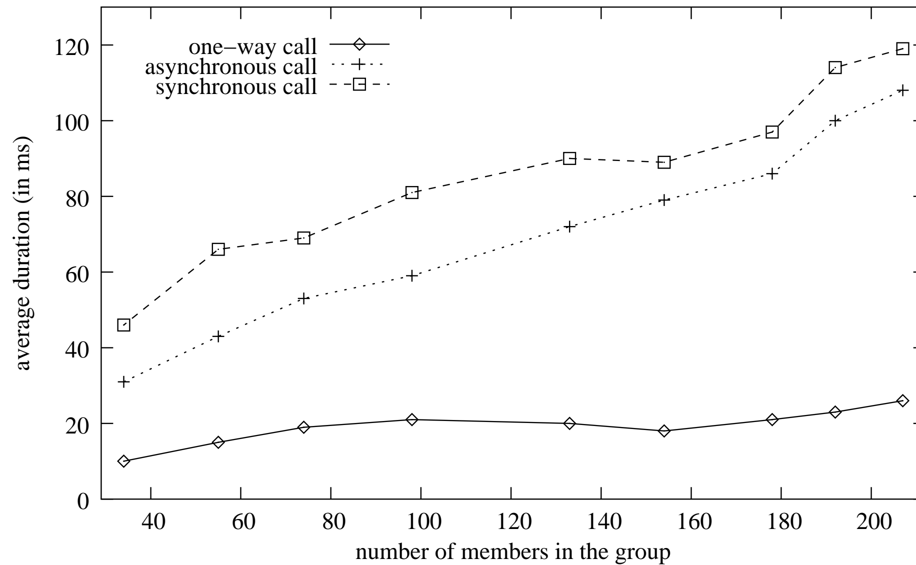


Figure 5.3: Method call on cluster

- for a synchronous communication, the call has reached all the group members and the result group has been created and contains all the results.

In this experiment synchronous (and asynchronous) method return a `null` value in order to not make the serialization process of the returned value interfere with the call duration. Figure 5.3 shows that one-way calls are faster than asynchronous calls, which are faster than synchronous calls. This is logic because each kind of call has more operations to perform: creation of the result group and *futures*, from one-way call to asynchronous call, and wait for the *futures* to be updated, from asynchronous call to synchronous call.

In order to best understand the latency of typed group communication, here is the description of a single *ProActive* communication (performed on the same computers). The details regard on one-way calls. The average duration is 5.1 ms. We have break it up in five parts: *(1)* the reification operation, 0.81 ms; *(2)* the serialization, 0.86 ms; *(3)* the network transfer, 0.97 ms; *(4)* the deserialization, 1.95 ms; and *(5)* the operations on server side, 0.51 ms.

Figure 5.4 is a similar experimentation. It differs only by the number of group members. In this experiment, the number of machines is set to 205 (i.e. 410 processors) and does not evolve. Each machine hosts a node on which several objects are created: from 1 to 24. Measurements target the achievement of one-way, asynchronous, and synchronous communication from one object on a node to the others. Because the test involves many objects, up to 5000, I change the member-to-thread ratio to 20, to avoid an oversupply of threads by the default ratio: 8.

We observe that the asynchronous curve and the synchronous curve remain quite close. The reason is due to the invoked methods. The synchronous method and the asynchronous method do not perform any computation. The time for the service is near zero: the methods only return a `null` object. Synchronous call blocks until the service ends, not the asynchronous ones. As the time of service is small in the invoked methods, the difference between the curves is small too.

The next graph presents the speedup induced by group communication. The previous experiment was reproduced without group communication. The curves in Figure 5.5 plots the ratio $\frac{duration\ without\ group}{duration\ with\ group}$. The values for the *duration with group* are the ones of Figure 5.4. The values for the *duration without group* are obtained using serial sends. We notice that one-way calls take the more advantage of group communication. In this test it goes to a 10.78 speedup. Next

are the synchronous calls, with a culminating point at 5.99. Finally the best speedup achieved by asynchronous communication is 4.78. The three curves begin to go up, until the group size reaches around 2500 objects, then the performances decrease. It goes down all the way to 4000 objects. After that speedups seem to remain regular. We blame the number of threads for that. A member-to-thread ratio set to 20 was too small for large groups. Around the breakpoint, at 2500 objects, the group runs with 125 threads. After this limit the caller was overloaded by too many threads. As said in Section 5.2.1, defining an effective and generic formula to obtain a correct thread pool dimensioning is not easy.

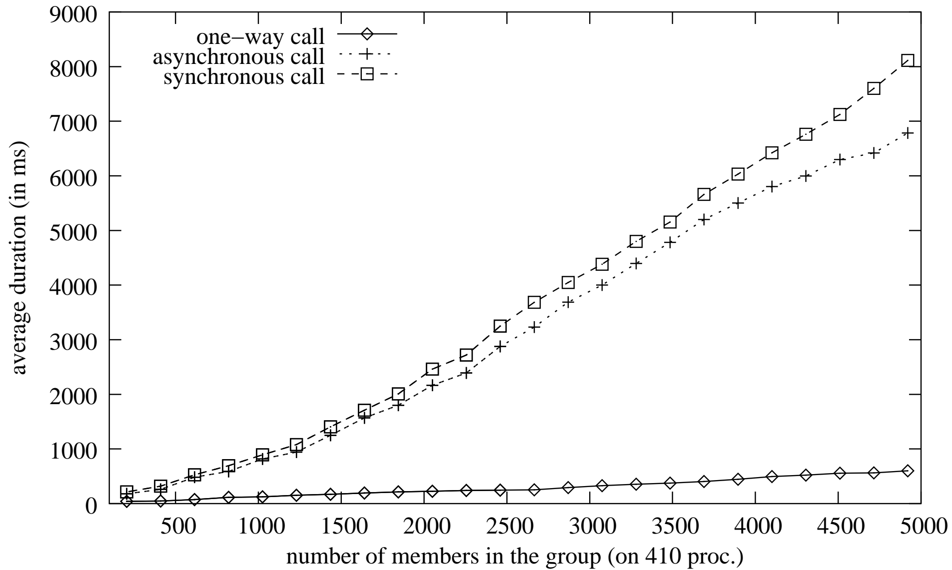


Figure 5.4: Method call on cluster depending on group size

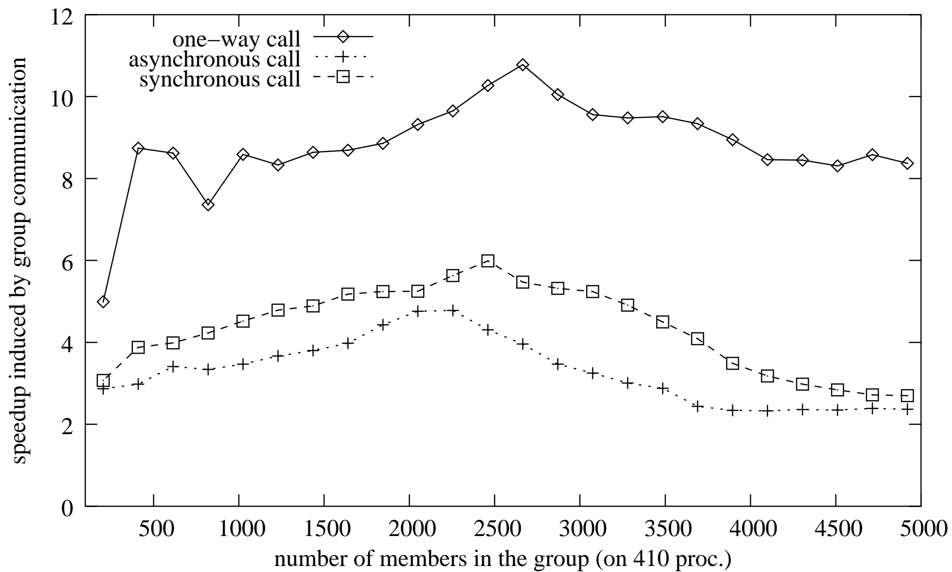


Figure 5.5: Speedup on cluster

5.3.2 In a Grid context

Now, let us come to a grid environment. The *Grid'5000* project, funded by the ACI GRID, aims at building an experimental Grid platform gathering 8 sites geographically distributed in France combining up to 5000 processors. The current plans are to assemble a physical platform featuring 8 clusters, each with an hundred to a thousand computers, connected by the Renater Education and Research Network. All clusters are connected to Renater at 2.5 Gb/s (10 Gb/s is expected in the near future). This high collaborative research effort is funded by the French ministry of Education and Research, INRIA, CNRS, the Universities of all sites and several regional councils.

The main objective of the project is to provide the community of Grid researchers in France with an experimental platform for their research, fully configurable for each experiment. The scope of the experiment that could be conducted on *Grid'5000* covers all the software stack layers between the user and the Grid hardware (clusters and networks). Typically a Grid researcher will be able to configure the platform with its favorite network protocols, Operating System kernel and distribution, middleware, runtimes and applications and run experiments on this setting. *Grid'5000* will also provide a set of software tools to allow easy experiment preparation, run and control, and fast experiment turn around.

The experiments currently envisioned by the *Grid'5000* participants concern: high speed network protocol design and evaluation, operating system adaptation and improvement in the perspective of a single system image for Grids, adding sandboxing and visualization in Operating System for the Grid, testing the benefit of object oriented middleware for application coupling, evaluating a large variety of fault tolerance techniques at the runtime level, testing application *gridification*, evaluating novel algorithms for High Performance Computing on the Grid.

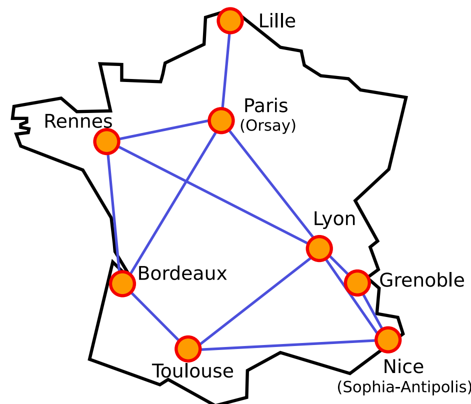


Figure 5.6: The map of current *Grid'5000*

Because the clusters and systems installations on all sites are not completed (even not started in Lille, the 8th site), I use only a subset of what will be the *Grid'5000* platform. Only the clusters of Nice (Sophia-Antipolis), Paris (Orsay), Lyon, Rennes, and Toulouse are available for computation.

- The cluster in Nice is composed of 106 computers with bi-AMD Opteron 64 bits @ 2 GHz and 2 GB of memory. Operating system is Red Hat 9 with a 2.4.21 kernel. Interconnection is assured by a 1 Gb/s Ethernet network.

- The cluster in Paris is a part of *Grid eXplorer* [GDX]. It is composed of 216 bi-AMD Opteron 64 bits @ 2 GHz and 2 GB of memory, interconnected by a Gigabit Ethernet network. It runs under Debian with a kernel 2.6.9.

- The cluster in Lyon is composed of 56 computers with bi-AMD Opteron 64 bits @ 2 GHz and

2 GB of memory running under Debian, kernel 2.6.8. Interconnection is assured by a 1 Gb/s Ethernet network.

- Three clusters compose the *Grid'5000*'s cluster in Rennes. The first one is a 64 bi-AMD Opteron @ 2.2 GHz, 2 GB of memory, Debian 2.6.10, cluster. The second is composed of 64 bi-Intel Xeon @ 2.4 GHz with 1 GB of memory running under Debian with a kernel 2.4.22. Finally the third cluster is composed of 32 bi-Power Macintosh G5 @ 2 GHz with 1 GB of memory, and running under Darwin, kernel 7.8.0. All machines are interconnected with Gigabit Ethernet networks, in a cluster, and between clusters.
- The cluster in Toulouse is composed of 31 computers with bi-AMD Opteron 64 bits @ 2.2 GHz and 2 GB of memory. Operating system is Fedora Core 3, kernel 2.6.10. Interconnection is assured by a 1 Gb/s Ethernet network.

All clusters run a Sun Java Virtual Machine 1.5.0, except the Macintosh in Rennes that run a 1.4.2 version. In conclusion, the aggregation of those five clusters constitutes a Grid with a total of 1138 processors, distributed on five sites, in a wide area (1175 km from Nice to Rennes, 670 km from Paris to Toulouse).

On this grid platform, I reproduce the experimentation led on the cluster (Figure 5.3). The goal is to observe the impact of a grid environment (high latency, shared high-speed links of communication between sites, etc.) on the performances of the group communication mechanism. Figure 5.7 presents the results. Source code remains the same as the previous benchmarks. Only the external deployment descriptor file changes in order to deploy on several remote locations. Remote objects are fairly distributed on the five sites.

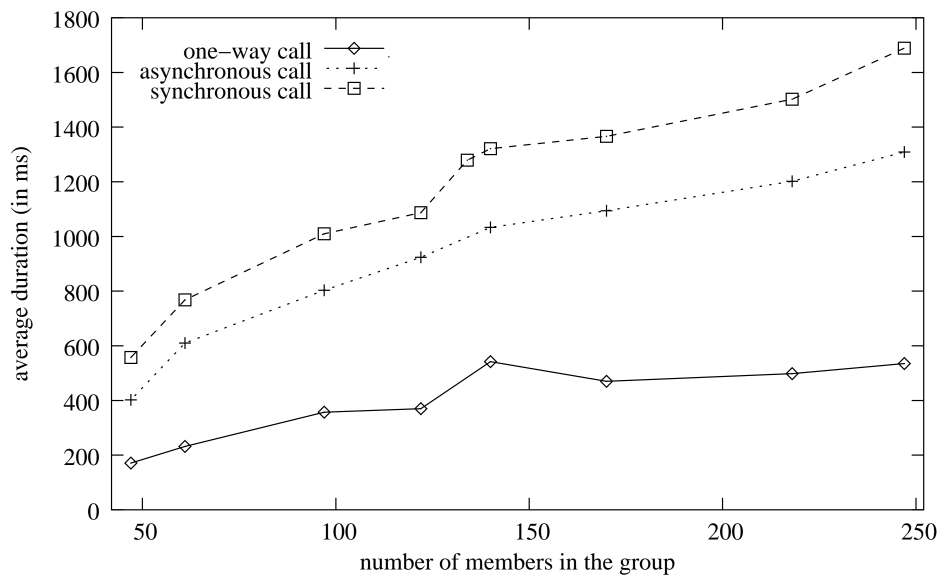


Figure 5.7: Method call on Grid

Relations between the communication schemes performances remain the same. Meanwhile group communication achievement becomes slower: up to a factor 10 compared to the cluster experiment. As the computers of each cluster look quite similar in term of performances, we blame the grid network for this decreased efficiency. More precisely the round trip time (RTT) inside the cluster is around 0.07 milliseconds while it is around 12 to 20 milliseconds in the period of the experiment (with some culminating values at more than 650 milliseconds to reach Lyon's cluster).

Next experiment regards performances with large deployment involving hierarchical and active groups. The deployment is achieved in two steps:

1. In the first step, a descriptor file deploys *ProActive* nodes located on each cluster. Then the "first level" group is created. This group consists of objects that play the role of entry point in a cluster. In this experiment, five objects are deployed, one on each cluster.

2. During the second step, the five objects activate in turn deployment descriptors local to each cluster. The deployment mechanism creates new nodes on all available machines. The "second level" group members are created on those nodes. The roots of these groups are the entry point objects.

At the end of those operations, a hierarchical group composed of active groups (i.e. the five entry point objects) is built and deployed on five clusters. It has a form similar to the one presented by Figure 4.10.

All experiments made with more than 569 nodes (the current number of machines in *Grid'5000*) are done with some machines running two nodes. In order to reach more than 1000 nodes we create two nodes by host. As all hosts are bi-processor machines, we can say that we deploy one node by processor. Figure 5.8 shows a hierarchical group of 1010 objects each one deployed on a different processor. The totality of the *Grid'5000*'s processors is not used because in such amount of machines, some are temporarily down while some are reserved by other users in exclusive access.

The performance measurements are exposed in Figure 5.8. It is no sense to compare the curves for one-way and asynchronous call of this experiment to the curves obtained in the previous experiments (Figure 5.7). In a hierarchical context, one-way call and asynchronous call block until the method has reached the first level members, not all the members of the sub-groups. In this case, the first level members remain the five entry point objects in the whole experiment. That is why the two curves remain parallel disregarding of the growing amount of objects in the sub-groups. The curve representing the synchronous call is more informative. Even in a hierarchical context, synchronous calls block the caller until all the members and the sub-members receive the method invocation and return a result. So this curve can be directly compared to the synchronous curves of previous experiments. As expected, the hierarchical approach is more efficient. The factor is about 10. The high latency between clusters is paid only once, from the caller to the remote cluster. Then intra-cluster communications, which are much more efficient, achieve the method invocation delivery and the gathering of local results. Finally, in accordance with the default behavior of hierarchical groups, inter-clusters communication callback the caller to advise it that the call has ended and the results are collected. Figure 5.8 clearly confirms that a hierarchical approach provides better scalability and performances.

5.4 Matrix multiplication

To validate the design and the implementation of group communication, we have programmed a basic numerical application pertaining to a parallel dense matrix multiplication. We have chosen the algorithm based on the *Broadcast-Broadcast Approach* described in [LI 96]. This algorithm pertains to our work as it extensively uses collective communications. As our group communication features some asynchronism, we foresee performance improvements compared to the same algorithm implemented without using the group mechanism but only point-to-point *ProActive* method calls.

Like most of the algorithms for parallel dense matrix multiplication, the *Broadcast-Broadcast Approach* algorithm performs a multiplication of the form C = αAB + βC on a two dimensional logical process grid with P rows and Q columns. In this demonstration we consider only the case where P=Q.

Once the distribution is done, sub-matrices of the two matrices to multiply are located on each computer which takes part in the computation. The *Broadcast-Broadcast Approach* algorithm consists in four steps:

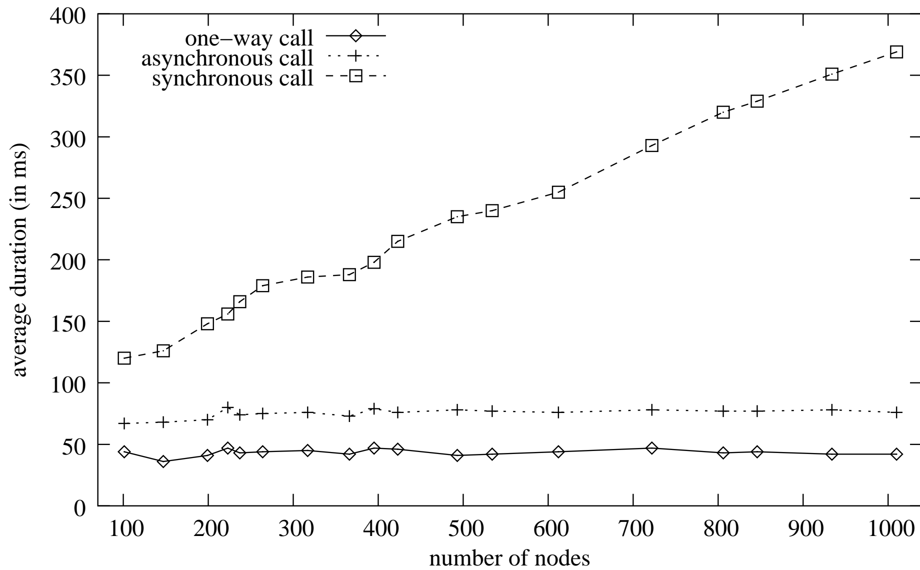


Figure 5.8: Method call on Grid with a hierarchical group

1. Broadcast the sub-matrices of A along the rows.
2. Broadcast the sub-matrices of B along the columns.
3. Update partial C sub-matrices with A and B sub-matrices multiplication in each process.
4. Repeat Step 1 through Step 3 P times.

At the end of those steps, the sub-matrices of C contain the result of the A*B multiplication.

It is obvious that each process of the logical grid will be represented by one active object, whose class represents a sub-matrix. The active objects of each row (resp. column) of the logical grid build up one group. Broadcast communication of sub-matrices along one row (resp. one column) will be achieved thanks to the group method call mechanism. Here is an implementation of the algorithm:

```
  // The method multiply is a basic centralized matrix multiplication;
  // it updates the right sub-matrix of C that in this code does not
  // need to be explicit, as it is obtained as result of the call to
  // this chunk of code.

  // row[i] and column[i] return the i-th row and i-th
  // column of the logical grid, in a typed group form.

  // The distributed matrix multiply method implementation:
for (int i=0 ; i<P ; i++)
  A.row[i].multiply(B.column[i]);
```

The mechanism of group communication provides a simpler implementation. The two lines of code replace about twenty lines of pseudo-instructions seen in [LI 96].

Figure 5.9 shows the time spent in order to compute the matrix multiplication depending of size of one side of square matrix. Three implementations of the algorithm are tested. Two of them use the *ProActive* library. The first one does the computation in a “centralized” manner; it means that the algorithm is deployed on only one computer. The second implementation is quite similar, it uses the *ProActive* library without group communication mechanism but it is distributed on nine computers. Finally, the last implementation uses the group communication mechanism. Experimentations were done using either one (see curve *centralized*) or nine Intel

Pentium III @ 933 MHz on the same 100 Mb/s local area network (see curves *distributed* and *distributed using groups*).

Obviously the centralized approach is quickly dismissed. With very small matrices (< 400*400 doubles) it is more efficient because the cost for computation is smaller than the cost for sending messages. When the matrix size increases, the distributed algorithms become more efficient. Again, this experiment proves that the implementation using groups achieves better performances than the one without.

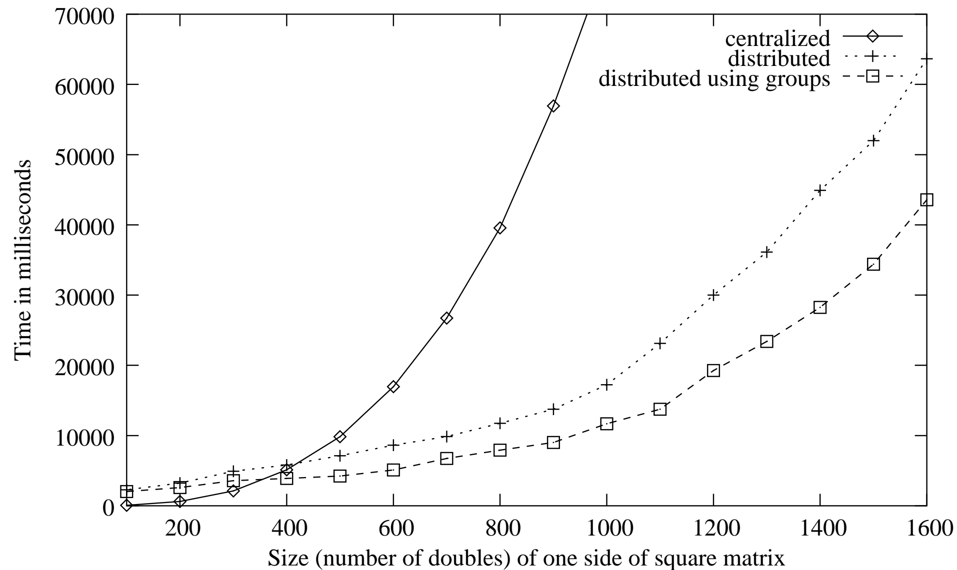


Figure 5.9: *Broadcast-Broadcast* matrices multiplication performances

Conclusion

The approach we chose to implement the group mechanism was driven by an elegant API allowing a seamless use of the group communication in object-oriented distributed programs. The implementation provides flexibility and adaptability. Hand-optimizations such as parallel achievement of the method invocation and factorization of common operations contribute to improve efficiency. The experiments performed on both cluster and grid environment expose both good performances and scalability.

Chapter 6

Applicative benchmark: Jem3D

Within the trend of object-based distributed computing, my research group (OASIS), conjointly with the CAIMAN research group[1] presented the design and implementation of a numerical simulation for electromagnetic waves propagation.

The general objective of this collaboration between computer scientists and applied mathematicians was to use modern programming languages and libraries such as Java and *ProActive*, for the design of a problem solving environment for complex applications in the bio-electromagnetic field. Such an environment ideally integrates software components for geometric modeling from medical images, unstructured grid generation, numerical simulation and scientific visualization. An example of such an environment based on CORBA is given in [SHI 01]. While such work uses the option of wrapping legacy code, the work presented here has concentrated on a full-fledged object-oriented version, for the sake of extensibility and adaptability. Overall, from an existing Fortran application named EM3D ([PIP 02]), a modular and extensible object-oriented version was developed using Java: Jem3D.

The approach we proposed was to define an object-oriented model of a code in Java, and use it for programming a sequential version. Then, in order to take advantage of parallelism and distribution, we use *ProActive*, for its additional characteristics compared to the standard Java RMI API, particularly the typed group communication. We have deliberately chosen not to use an explicit message-passing library (MPI, or Java version of it like MPJ [CAR 99], or MPIjava [CAR 98b]) for taking advantage of distribution: we aim at enforcing code reuse by applying the remote method invocation mechanism instead of explicit message-passing.

The aim of this work was to emphasize on the benefits we get on software engineering aspects (possible extension of the Java version, full portability, ease of deployment, etc.) through a complete rewriting of the Fortran version. Recent works such as [HEN 03] also mention the advantages of using object-oriented practices for finite element analysis; the main difference is that we do not rely on direct parallel solvers. We do not get the performance of executing native code resulting from Fortran or C++ programming; see for instance works that wrap MPI-based legacy codes as Java or CORBA components [LI 01, DEN 03]. Even under those conditions, with a pure object-oriented programming approach entirely based on point-to-point or collective method invocations in Java; we still get good performances and speedup. This work was subject to the publication [BAD 04b].

EM3D: a parallel solver for electromagnetic waves propagation

The Fortran EM3D software has been designed for the numerical simulation of electromagnetic waves propagation in the time domain. The software numerically solves the 3D Maxwell equations for homogeneous or heterogeneous linear media. It relies on a *Finite Volume Time Domain*

[1] OASIS and CAIMAN are both research groups at INRIA in Sophia-Antipolis, France.

(FVTD) method designed on unstructured tetrahedral meshes, potentially applicable to general hybrid meshes. The FVTD method adopts a cell centered formulation[2] (a control volume is taken to be a tetrahedron) with a centered numerical scheme for the computation of convective fluxes, combined to an explicit leap-frog time integration scheme. The resulting solver is second-order accurate in time and space for regular meshes, and provides unsteady solutions that conserve a certain form of discrete electromagnetic energy [PIP 02]. It is interesting to note that such finite volume formulations were originally designed for computational fluid dynamics, such as 3D Euler or Navier-Stokes equations [LAN 96]. This clearly motivates the development of a general object-oriented framework that would facilitate the development of various simulation softwares for *Partial Differential Equations* (PDE).

Finally, the parallelization of EM3D combines a domain partitioning strategy with a message passing programming model using the *Message Passing Interface* (MPI). The partitioning of the computational domain is obtained with the ParMETIS tool [KAR 99].

6.1 Basic architecture of Jem3D

The proposed object-oriented model is such that it can be reused for the development of simulation software tools which are based on finite volume type methods on unstructured meshes. The application of this model is currently limited to the Maxwell equations for electromagnetic waves propagation but it can be extended to deal with Euler or Navier-Stokes equations that model compressible flow calculations [LAN 96]. Moreover, the model could also be extended to include classical finite element type discretization methods. The three main features of the model are:

- The ability to deal with 2D and 3D computational domains,
- The possibility of choosing between different types of discretization elements (triangle, quadrangle, tetrahedron and hexahedron),
- The inclusion of the two main classes of finite volume methods i.e. the vertex centered and element centered formulations.

The object-oriented model essentially consists of two types of classes:

- Classes that are concerned with the definition of the geometry (or computational domain),
- Classes that are related to the application (for instance physical and numerical components).

These classes are strongly linked to the physical context under consideration (electromagnetic waves propagation in the present case).

Definition: *Domain*: A domain is the overall volume of the calculation.

6.1.1 Geometry definition

The finite volume methods adopted in [PIP 02] and [LAN 96] rely on the use of an unstructured mesh for the discretization of the computational domain. The construction of such meshes can be based on various types of discretization elements. The standard situation is such that only one type of discretization element is considered for the definition of a given unstructured mesh. However, in the general case, the computational domain could be discretized by combining several types of elements (hybrid discretization). The classes considered here are concerned with the definition of the discretized geometry with an unstructured mesh. In order to do so, one essentially needs two basic geometric entities: the vertex and the element. The element is used to connect a number of vertices and an unstructured mesh is defined by filling the computational domain with elements. These two geometric entities are included in our object-oriented model

[2] Other widely used finite volume methods rely on a vertex centered formulation.

through the definition of several classes: *Vertex2D* and *Vertex 3D* (which extends *Vertex2D*) are simple concrete classes for the definition of a vertex in 2D and 3D; *Element*, *Element2D* and *Element3D* are abstract classes for the definition of an element in 2D and 3D (see Figure 6.1).

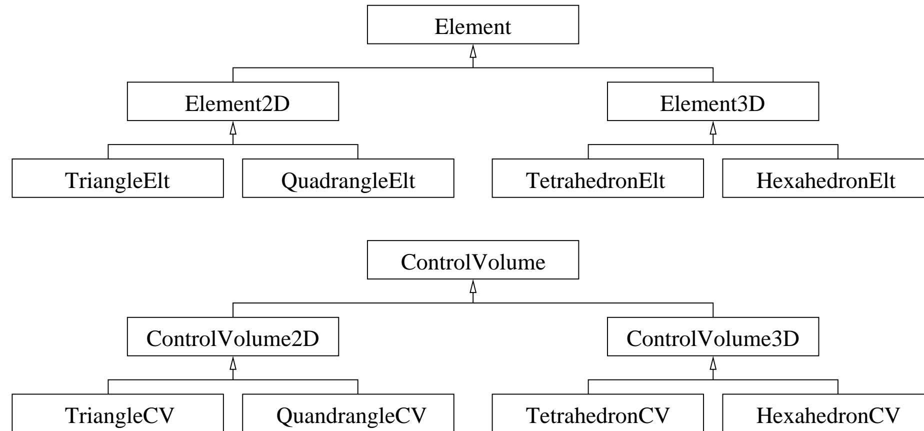


Figure 6.1: Definition of an element and a control volume in 2D and 3D

6.1.2 Application aspects

Starting from the discretized geometry, it is then necessary to define the classes related to the numerical methods (finite volume methods in the present case). Finite volume methods typically yield the computation of a flux balance through the boundary of a control volume (also called a cell). Indeed, a control volume is a geometric entity that can be seen as another building block for the discretization of the computational domain but it is not the natural basic entity for the definition of an unstructured mesh. In some sense, it is introduced artificially since it represents the calculation support of finite volume type methods. Note that for finite element type methods, the calculation support is simply given by the element. In the finite volume framework, the unknowns of the problem are averages of the physical quantities computed over control volumes while in finite element methods, the unknowns are the values of the physical quantities associated to the vertices of the mesh.

In the object-oriented model, the control volume is defined through a hierarchy of classes partially shown in Figure 6.1. At that point, it is worthwhile to make two remarks:

- As for the vertex and the element entities, the definition of the control volume includes classes dedicated to the 2D and 3D cases. In addition, we have taken into account the two main families of finite volume methods i.e. the vertex centered and element centered formulations. In a vertex centered formulation, a control volume is built around a vertex using partial contributions from the set of elements attached to this vertex. In an element centered formulation, the control volume is simply taken to be an element (triangle, quadrangle, tetrahedron or hexahedron). In Figure 6.1, the latter formulation is illustrated with the choice between *HexaedronCV* and *TetrahedronCV*.
- In practice, the flux balance is evaluated as the combination of elementary fluxes computed through a series of facets that describe the boundary of the control volume. This yields another hierarchy of classes for the definition of various types of facets (see Figure 6.2).

Finally, we note that the EM3D solver is based on an element centered formulation where the control volume is a tetrahedron and the facet is a triangular face. Therefore, at the lowest level of the hierarchy of classes for the definition of a facet, we currently have the various types of triangular faces that are considered in the EM3D solver: either an internal face or a boundary face and, for a boundary face, several subclasses corresponding to the different types of boundary conditions.

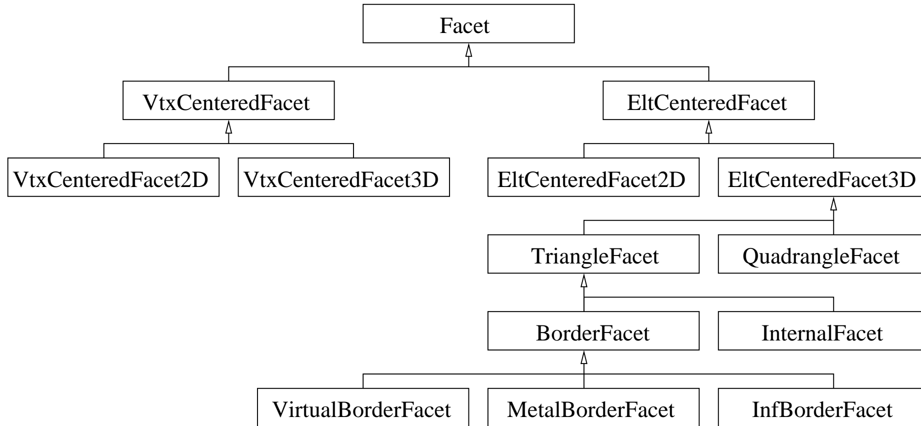


Figure 6.2: Definition of a facet in 2D and 3D

6.1.3 Overall skeleton and control

The overall skeleton of the EM3D solver is shown in Figure 6.3 and so will be reproduced as such in Jem3D. Let $t^n = t^0 + n\Delta t$, $\mathbf{E}$ and $\mathbf{H}$ respectively denote the discrete time, the discrete electric field and the discrete magnetic field (both fields are vectors of size $3 \times N_{\mathbf{CV}}$ consisting of the x, y and z components of the physical quantity computed on each control volume). In the leap-frog time integration scheme adopted in EM3D, each time step allows the calculation of $\left(\mathbf{E}^{n+\frac{1}{2}}, \mathbf{H}^{n+1}\right)$ from $\left(\mathbf{E}^{n-\frac{1}{2}}, \mathbf{H}^{n}\right)$. In practice, the main time stepping loop of Figure 6.3 is decomposed in three phases:

1. The flux balance for the magnetic field is computed from the distribution of the magnetic field obtained at the previous time step (i.e. $\mathbf{H}^n$). This flux balance is used to update the electric field (i.e. to compute $\mathbf{E}^{n+\frac{1}{2}}$ from $\mathbf{E}^{n-\frac{1}{2}}$ using the flux balance for $\mathbf{H}^n$). Group communications perform propagation of data and control the end of the step (ensure that all required data was exchanged before continue to the next step).

2. The flux balance for the electric field is computed from the distribution of the electric field resulting from the previous phase (i.e. $\mathbf{E}^{n+\frac{1}{2}}$). This flux balance is used to update the magnetic field (i.e. to compute $\mathbf{H}^{n+1}$ from $\mathbf{H}^n$ using the flux balance for $\mathbf{E}^{n+\frac{1}{2}}$). Group communications perform propagation of data and control the end of the step (ensure that all required data was exchanged before continue to the next step).

3. The discrete electromagnetic energy (which is a scalar value) is computed from the distributions $\mathbf{E}^{n+\frac{1}{2}}$ and $\mathbf{H}^{n+1}$. This particular quantity is used to monitor the simulation in the sense that, according to the results of the theoretical analysis [PIP 02], it should remain constant.

The first and second phases are implemented using loops over the lists of triangular faces using different numerical schemes for the calculation of fluxes through internal and boundary faces. Since the original EM3D code is programmed in Fortran 77, the information related to the definition of internal and boundary faces (as well as for vertices and tetrahedra) are stored using array data structures.

In the Java version of EM3D, lists of vertices, elements, control volumes and facets are implemented using the `ArrayList` class from the standard Java API. `ArrayList` is a resizable-array implementation of the `List` interface. Like an array, it contains components that can be accessed using an integer index. The size of an `ArrayList` can grow or shrink as needed to accommodate adding and removing items after the `ArrayList` has been created. This class is equivalent to `Vector` except that it is unsynchronized: it permits simultaneous and faster access.

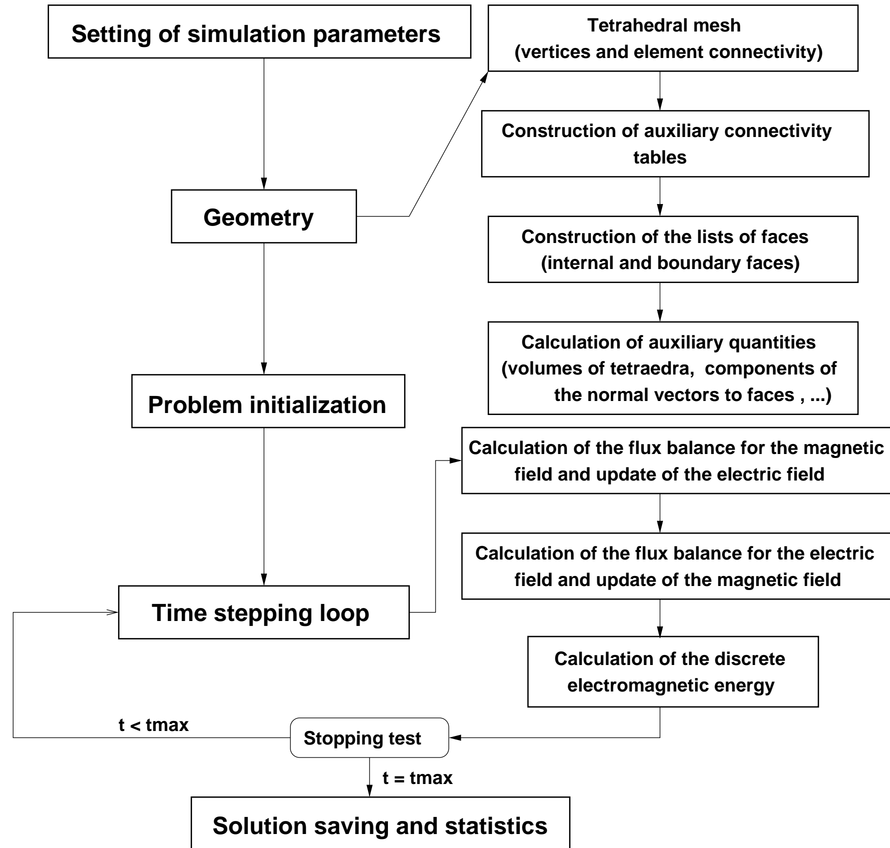


Figure 6.3: Overall application skeleton

The components of the object-oriented model as described can be viewed as contributing to a general library on top of which a particular application can be built. Such an application based upon a vertex centered formulation has yield to Jem3D, the Java version of the EM3D solver, but several others could be considered in the future.

6.2 Design of the parallel and distributed version of Jem3D

This section explains how, using the *ProActive* library and its typed group communication mechanism, we have programmed an efficient, parallel, and distributed version of Jem3D starting from the sequential one.

6.2.1 Basic ideas and principles

Figure 6.4 describes the architecture of the sequential version of Jem3D: all (triangle) facets, whatever be their real type (internal or not), are grouped in an `ArrayList` of facets; all (tetrahedron) control volumes are grouped into an `ArrayList`. As each internal facet belongs to two control volumes (CV), one can see the corresponding two references (from a face to two CVs).

After the initialization phase, the main loop repetitively executes the three phases presented in Figure 6.3, by going over the `ArrayList` of facets. The three phases read or update some values (i.e. the *X,Y,Z* coordinates of the electric and magnetic fields) of the corresponding CV(s).

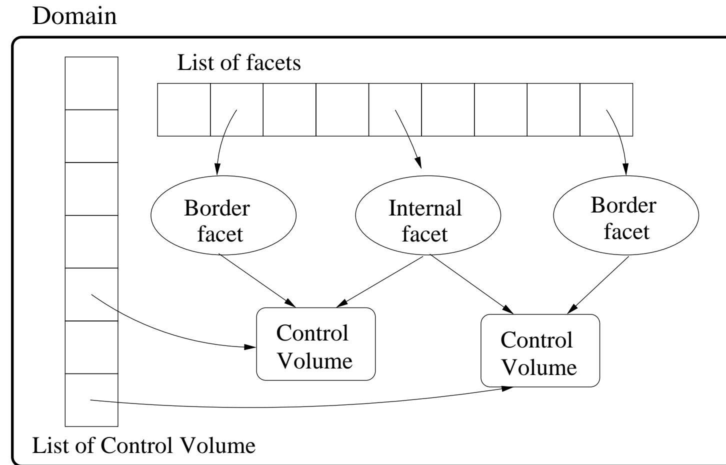


Figure 6.4: Architecture of the sequential version of Jem3D

The partitioning follows a standard decomposition of the entire domain into a set of geometric sub-domains. As we will see, our object-oriented approach brings a specific advantage: sequential references to some data-structures (e.g. facets, CVs) can be turned into remote references in a transparent manner for the code using them.

Definition: *Sub-Domain*: A sub-domain defines a part of the distributed application whose elements are located in the same address space.

The partitioning first occurs on facets: each one is assigned to a unique sub-domain. As a consequence, some CVs will be shared by two sub-domains (or sometimes more); indeed a shared CV is referenced by facets belonging to different sub-domains. Of course, specific programming techniques are used in order to read and update shared CVs.

6.2.2 Partitioning, local and remote objects

Figure 6.5 shows the architecture for the distributed version of Jem3D. The underlying idea for the parallelization is to apply a standard and natural geometric decomposition of the 3D computational domain into sub-domains. As such, some facets will contribute to control volumes that may be located onto neighbor sub-domains.

Definition: *Border Facet*: We name border facet the facets of the thetrahedra located on the boundary of a (sub-)domain.

We introduce the *Virtual Boder Facets* (VBF) to represent these facets that belong to two sub-domains. In a couple of neighbor sub-domains, both have a reference to a VBF designating the shared facet. Each VBF contributes to the computation. Two twin VBFs which are copies of the same facet must exchange and combine their values to obtain the value of the facet. For the update access, it is the responsibility of the sub-domain to trigger a remote method call onto the corresponding neighbor sub-domain – implemented as an active object –, which itself sets values in the twin VBF. Eventually, the value of the facet is set in both VBFs.

Thanks to polymorphism and dynamic binding, there is no need to explicitly deal with the effective real types of facets: internal or virtual border. As a result, the control volumes that reference virtual border facet, as well as the loop that uses them, can execute unchanged.

The architecture features a totally decentralized approach. The application is fully *peer-to-peer*: each sub-domain communicates with the others without any centralized supervisor. As

centralized points are usually bottlenecks due to overload problems, we aim at achieving a better scalability.

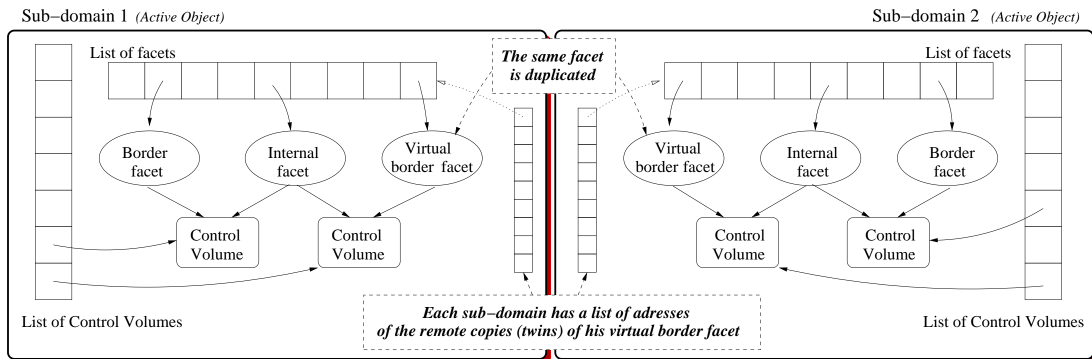


Figure 6.5: Architecture of the distributed version of Jem3D

6.3 A group communication model to enhance performances

Regarding a read access, a naive solution would have been to let each facet independently triggers a remote method call to read (*pull*) the values of its corresponding control volume (through an access via the remote sub-domain active object). As the algorithm implemented in the sequential version loops over facets in each phase, this implies that the computation could proceed only when a given facet effectively gets the remote values, adding up RMI and network latency. As we actually know who will need a given value, the idea is to *push* it rather than pulling it, avoiding one way of the communication needed for a pull.

In order to achieve that behavior, a sub-domain maintains a link to all its neighbors with which it shares a facet, in order to be able to push new values to the corresponding virtual border facets. The set of neighbors is stored using a typed group. As seen, such a group is directly operable with method calls: only one method call is enough to reach all members of the group. Here, the main point is that this avoids programming a data structure that would require an iterator in order to visit each neighbor sub-domain, and as such, perform the communications sequentially. Each virtual border facet has to receive the value of its twin. It has been again possible to take advantage of the group communication feature in order to simply program this operation.

More precisely, at initialization time, each sub-domain executes the following (refer to Figure 6.5 for illustration):

```
   // Builds group of neighbor sub-domains
SubDomain neighbors =
    ProActiveGroup.newGroup("SubDomain", {sd1,sd2,...});

   // For each sub-domain j, builds up a VBFFieldExchange
   // collecting all references to virtual border facets that
   // are shared by the current sub-domain with sub-domain j
VBFFieldExchange VBFValues_j = ...

   // A group of VBFFieldExchanges to be scattered to shared VBFs
VBFFieldExchange exchangeValues =
    ProActiveGroup.newGroup("VBFFieldExchange",
                             {...,VBFValues_j,...});
ProActiveGroup.setScatterGroup(exchangeValues);
```

`SubDomain` is the class name of the sub-domain active object. The variable `neighbors` stands for the *neighbor sub-domains group*, while `exchangeValues` for the *neighbor-shared VBF values group*. As seen in Chapter 4 the `setScatterGroup` is a *ProActive* feature allowing to specify that a group parameter, subsequently used in a group communication, has to be isomorphically dispatched to the members of the group: the ith element of group parameter goes (as the remote method call parameter) to the ith target object in the group.

Then, at the beginning of each phase of the main loop after an update of the VBFs, the following simple instruction is executed in order *to push* appropriate values on each remote and shared VBF:

```
neighbors.push(exchangeValues);
```

This means that on each sub-domain *j* referenced in the group `neighbors`, it calls the method `push` taking as parameter the corresponding `VBFFieldExchange` referencing VBFs that are shared with sub-domain *j*. Method `push` will set values of each VBF in the corresponding remote VBF on sub-domain *j*. Subsequently, those values will be available locally by the control volumes when needed.

6.4 Benchmarking

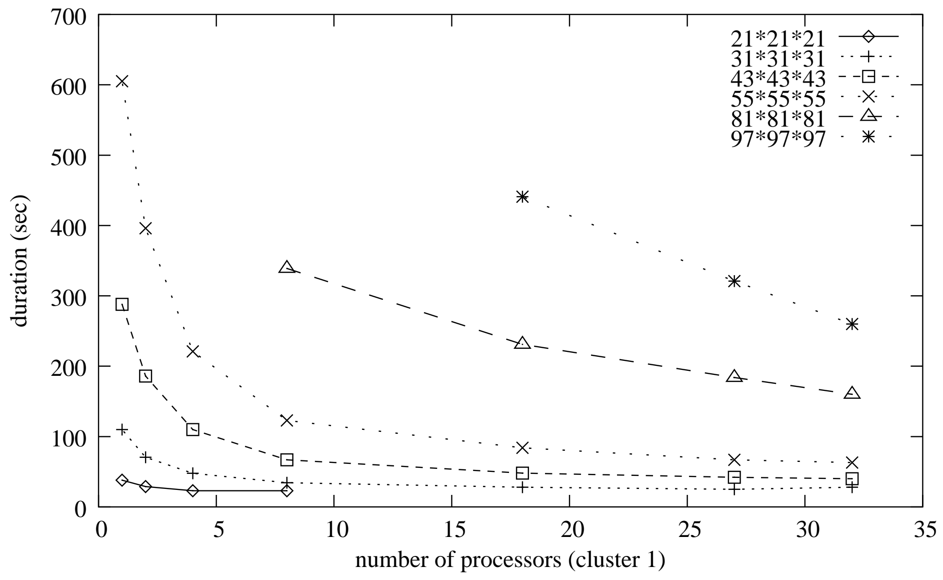
As Jem3D is a real application entirely written with *ProActive* and as the main communication patterns are based on the typed group communication, we are very interesting in evaluating its performances on various platforms. Experiments are done on homogeneous and heterogeneous clusters, with fast and slow networks.

6.4.1 Benchmarks on cluster

To do measurement up to 32 processors, the benchmarks use a cluster of 16 Intel Pentium IV bi-Xeon @ 2 GHz, 1 GB (RDRAM), Linux Red Hat 2.4.17, interconnected with a 1.5 Gb/s Ethernet. In order to measure performances on 64 processors, we add a second cluster of 16 bi-Pentium III @ 933 MHz, 512 MB (SDRAM), Linux Red Hat 2.4.17, interconnected with a 100 Mb/s Ethernet. Each computer belonging to the second cluster communicates with computers in the first cluster through a 100 Mb/s Ethernet link. We use the Sun Java Virtual Machine 1.4.0.

The bench aims at computing the time evolution of the eigenmode (1,1,1) in a cubic metallic cavity. Reported results are the total execution time for 100 time steps. To give an idea of the data involved, a mesh size of e.g. 81x81x81 represents 521,441 vertices, 3,072,000 tetrahedra, 6,220,800 faces. All of them are represented at runtime by objects in the Java version. Only sub-domains are active objects. In all experiments we map one node by processor, and one sub-domain by node. The whole domain's mesh for computation is fairly split between sub-domains.

In a first time, let us compare the sequential Fortran version, with the Java version. On each of the configurations we have tested (different numbers of processors, different amount of data) we have noticed an average ratio of execution time of one loop ranging from 3.5 to 3.7.



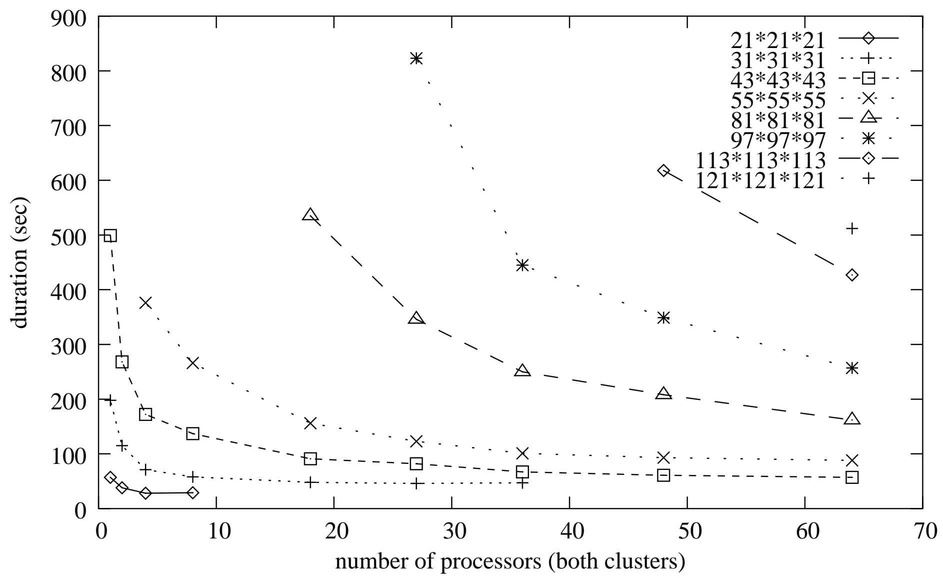


Figure 6.6: Average duration of 100 iterations

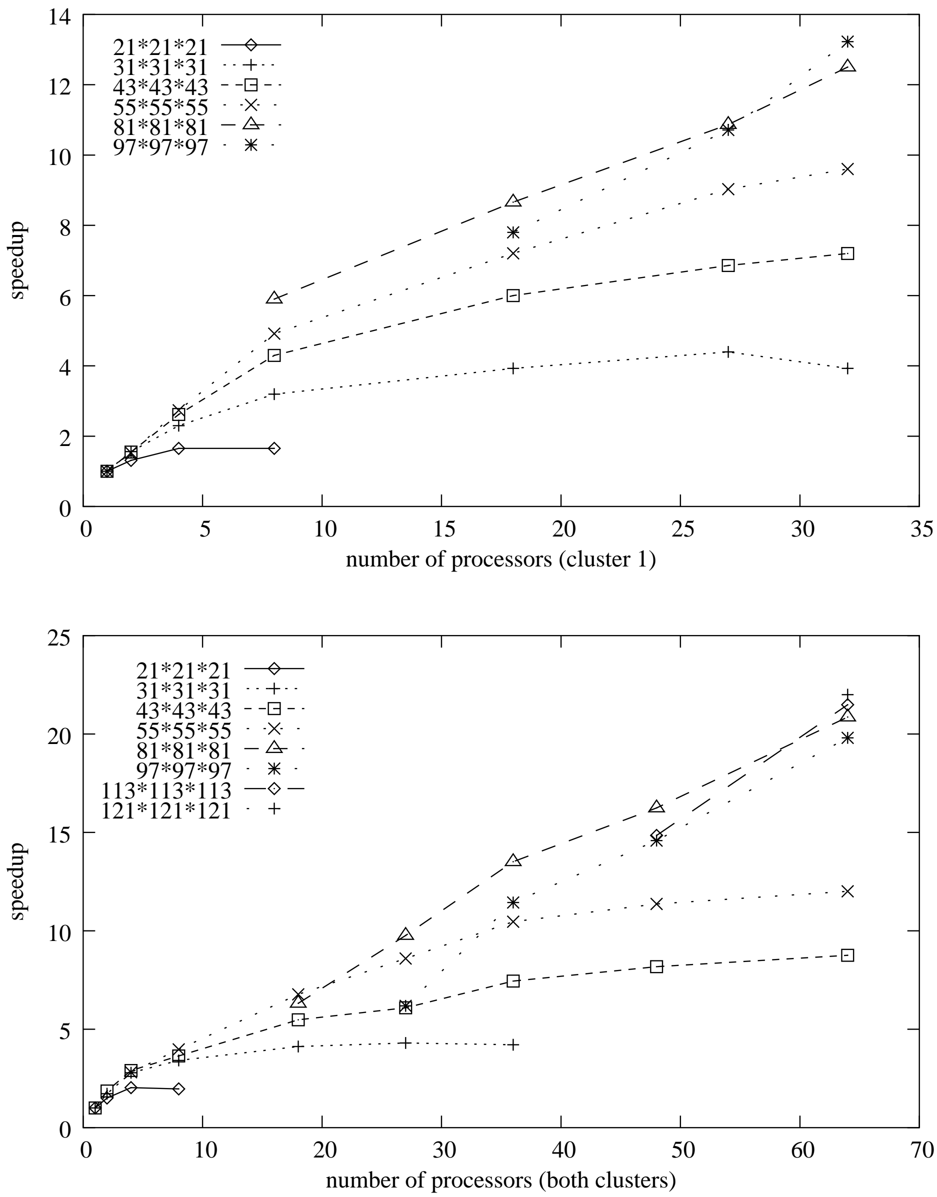


Figure 6.7: Speedup

We have benchmarked the Java parallel version. Results are reported in Figure 6.6. Both graphics present the average duration in seconds of several executions of the benchmark (benchmark which loops in the main loop for 100 time steps). The upper one plots experiment results when running on the 16 most powerful computers (cluster 1), whereas the lower one is when running on all the computers (cluster 1 and 2). Due to the synchronization step at the end of each loop, the measured time is the slowest computer (i.e. the longer time). Moreover, the duration highly decreases up to 8 processors for small data sizes (less than 55x55x55); then adding resources becomes rather ineffective: from 8 to 18 processors speedup only improves from 5 to 7 on a 55x55x55 mesh. For larger size problems, starting as low as 81x81x81, the parallelization is useful all the way up to 64 processors.

Figure 6.7 presents the time speedup of hundred iterations of the main loop, depending on the number of processors involved in the computation. The upper graphic shows measurements using the most powerful cluster, and the lower one using both clusters. For some of the problem sizes, the reference execution time (i.e. the sequential execution) is *extrapolated*, because these problem sizes were so huge that the problem could not been solved sequentially on one processor; in practice, the experimentations lead to an out-of-memory error. The extrapolation is computed in the following way: first, we calculate the time spent in the treatment of a control volume, which is to be considered as the elementary data in the application, by dividing the total execution time of the benchmark by the number of control volumes involved. Secondly, we estimate the execution time for large size problems by multiplying the elementary time by the number of control volumes in the problem. The curves expose an efficiency in the range of 30% to 35% for the larger problems using all available processors. Reducing the number of processors, the efficiency steadily increases up to 75% on two processors for all cases.

6.4.2 Benchmarks on an Intranet heterogeneous cluster

In order to experiment with very large data set and large number of processors, we run benchmarks on the INRIA production network, such configuration being sometimes called *Intranet grid* since it relies on desktop machines interconnected via an Intranet network. The deployment scheme remains the same: no additional feature is introduced to ease or make more efficient neither the deployment phase nor the computation phase. Experiments are done by night in order to avoid at most interferences produced by the regular users of the desktop machines.

Such kind of grid is very heterogeneous, as well in term of machines (single or multi-processor, CPU speed, memory size, etc.), as in networks (bandwidth, protocol, etc.)[3], and as in operating systems (distribution, kernel version, etc.). We choose to discard slow computers. They are subjects to harm performances in a too large dimension. We fix the acceptation threshold to 1 GHz computers with at least 1 GB memory and a 100 Mb/s connection. Fastest computers were Pentium IV @ 3 GHz. Thanks to Java, all operating systems that provide a Java Virtual Machine were accepted. We were able to “collect” up to 252 machines, with a total of 294 processors.

As previously, *(1)* the bench aims at computing the time evolution of the eigenmode (1,1,1) in a cubic metallic cavity, *(2)* in all experiments one node is mapped on one processor, and one subdomain on one node, and *(3)* the total domain of computation is fairly divided into sub-domains.

[3]In practice, most of the Intranet networks are Ethernet networks, only the network bandwidth varies, not the protocol. This is the case in the INRIA's Intranet.

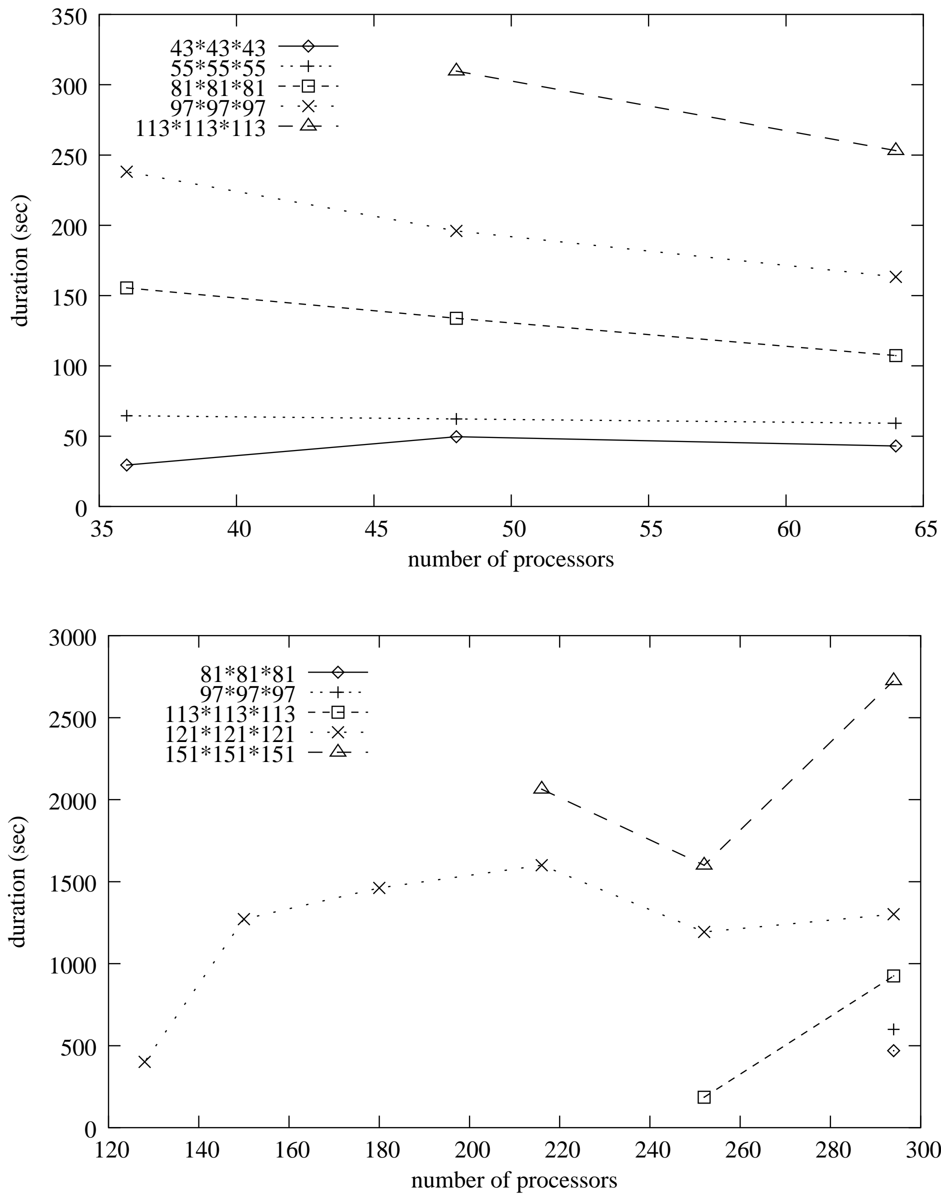


Figure 6.8: Intranet computing

Figure 6.8 presents the average duration of 100 iterations on small and large Intranet cluster. Results are split in two graphs in order to obtain a better readability: both graphs are obtained within the same experimental context. With small Intranet clusters (less than 64 processors) curves appear very regular (see the upper curves): the computation time decreases while the number of involved processors increases. Duration and speedup exposed here, are better than the one obtained on the “regular” clusters in the previous section (see Figure 6.7). The reason is that we systematically choose the most competitive computers among the 252 available ones. Those 36, 48 and 64 more competitive computers are faster than the computers in the cluster, thus it achieves better results.

It is more difficult to discuss about the lower curves produced with more than 128 processors. Irregularity may be caused by overloads of some computers, causing swap and general slowdown of the computation. Or it might be caused by unexpected uses of the computers by their owner during the measurements. We don't find a generic pattern to justify this behavior. However, the set of measurements is complete for 294 processors and seems quite coherent. The execution time varies with the mesh size; it requires more time to compute bigger meshes.

6.4.3 Benchmarks on a grid using a fast RMI protocol

As the group communication of *ProActive* is a high-level mechanism, it is possible to change the underlying communication protocol. RMI, the base of Java's distributed computing, has important shortcomings for high-performance Grid computing. In [HUE 04], Ibis is proposed as an alternative to RMI, and tested with the Jem3D application.

Ibis is a project at the Vrije Universiteit Amsterdam that aims to design and implement an efficient and flexible Java-based programming environment for Grid computing, in particular for distributed supercomputing applications. Ibis boosts RMI performance using several optimizations, especially to avoid the high overhead of runtime type inspection that current RMI implementations have. The philosophy behind Ibis is to try to obtain good performance without using any native code, but allow native solutions to further optimize special cases. For example, a Grid application developed with Ibis can use a pure-Java RMI implementation over TCP/IP that will run "everywhere". However, when the application runs, for instance on a Myrinet cluster, the RMI runtime system can request Ibis to load a more efficient communication implementation for Myrinet that partially uses native code.

Following [FOS 02] stating that a grid is a system that coordinates resources without centralized control, using standard protocols and interfaces to achieve a non trivial quality of service, we experiment on a grid in the Netherlands named *Distributed ASCI Supercomputer 2*. DAS-2 is a wide-area distributed computer of 200 bi-Pentium III nodes running at 1GHz, with 1GB of memory. The machine is built out of clusters of workstations, which are interconnected by SurfNet, the Dutch university Internet backbone for wide-area communication, whereas Myrinet, a popular multi-Gigabit LAN, is used for local communications.

Once again, the bench computes the time evolution of the eigenmode (1,1,1) in a cubic metallic cavity. The mapping is one node on one processor, and one sub-domain on one node. The total domain of computation is fairly divided into sub-domains, independently on which cluster it is located. We have no control on the sub-domain allocation: two sub-domains that highly interact may be deployed on two different clusters.

Thanks to the *ProActive* deployment scheme, it was quite easy to deploy on the DAS-2. Taking full advantage of it, multi-cluster experiments are performed by requesting nodes on each of its parts. Figure 6.9 shows the result of the experiments. The upper graph demonstrates that execution remains regular in spite of the data distribution over a grid. The lower graph presents the speedup. On 150 nodes the speedup is 97 (efficiency 64.67%). The average efficiency of those experiments is 75.99%. The best performance is 85.4% on 80 nodes with a 201*201*201 mesh. Two curves plots the performances of the application using standard RMI. It allows to visualize the benefits of the fast RMI protocol. Ibis improves performances by a factor around 1.48.

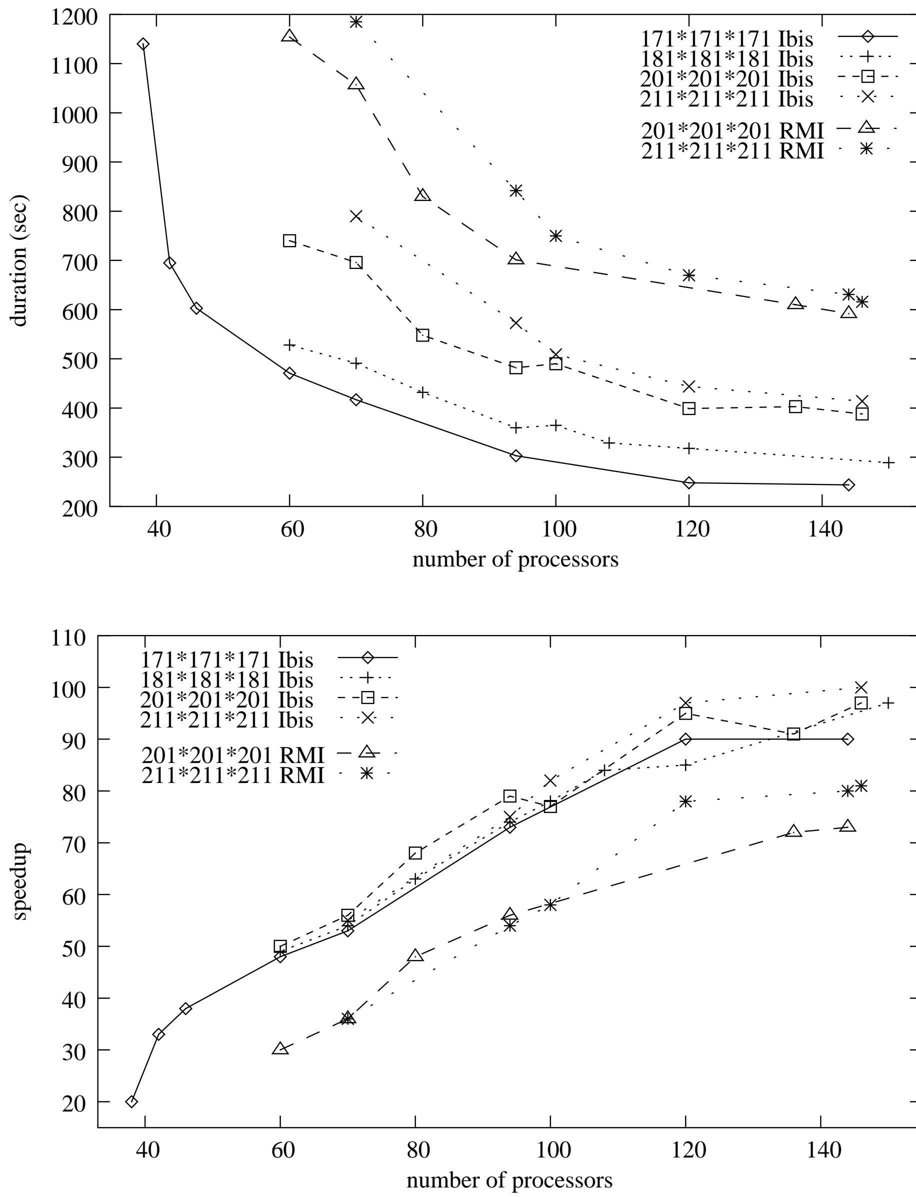


Figure 6.9: Grid computing

Conclusion

The Java version of EM3D has great potential for extension and adaptability: going from element to volume methods, using structured, unstructured, or hybrid meshes. At the same time, the performance penalty for such an abstract architecture implemented in Java seems reasonable: in a factor of 3.5 to 4 compared to the Fortran version. This is a good result according to [FRU 03] that shows Java applications have a factor from 3.3 to 12.4 slower than the corresponding Fortran operations. Moreover, this Java version is rather recent compared to the Fortran one, and there are still a lot of rooms for code optimizations. Using *ProActive* as high-level library, the parallel version was easy to obtain, maintaining the structuring of the sequential one. As a consequence, if an evolution of the sequential version occurs, the parallel one should remain, and evolve automatically.

Getting at performance figure, first it is important to note that the parallel object-oriented approach, using fully standard and portable elements of the Java platform, is already effective on the problem size. The parallel version allowed us on the clusters to get results with data size significantly larger than the sequential version (121x121x121 versus 43x43x43). The former number is to be compared to the largest size the Fortran version can currently execute: 161x81x81 (which is equivalent to 101x101x101 mesh). Even if this is due in the current Fortran program to a problem of static array allocation which could be improved with some restructuring, it probably tells something about the flexibility of a more dynamic approach.

Analysis of the executions confirms that progress should come from two yet-to-be-improved pieces of the current platform: serialization and standard RMI. As it is well known [GET 01, PHI 00], the standard Java RMI mechanism is rather slow for cluster and grid computing. So, in order to reach better scalability using the parallel version, the first step is to use fast implementations of the transport layer. We choose Ibis to achieve this. Experiments demonstrate that numeric applications can be written with a Java library and give acceptable results especially if this library provides efficient group communications and transport protocol.

In order to give a more tangible representation of Jem3D computations, the following pictures show screenshots of the graphical interface plugged on the computations. For instance, Figure 6.10 shows an iso-surface of the electric field induced in a cubic metallic cavity at a given time. Figure 6.11 presents JECS, a Java Environment for Computational Steering and Visualization of 3D numerical simulations. It was mainly developed by Saïd El Kasmi, member of the CAIMAN team. It is a generic distributed environment that supports interactive visualization and remote computational steering of parallel and distributed applications in a collaborative manner. The aim of this environment is to provide tools for easy coupling of running numerical simulations to a remote visualization server and allow visualization in a real-time fashion. Finally, Figure 6.12 presents a Jem3D computation involving a more complex mesh: a plane. Upper picture shows the mesh in wireframe, while the lower picture shows the electro-magnetic values on the surface of the plane.

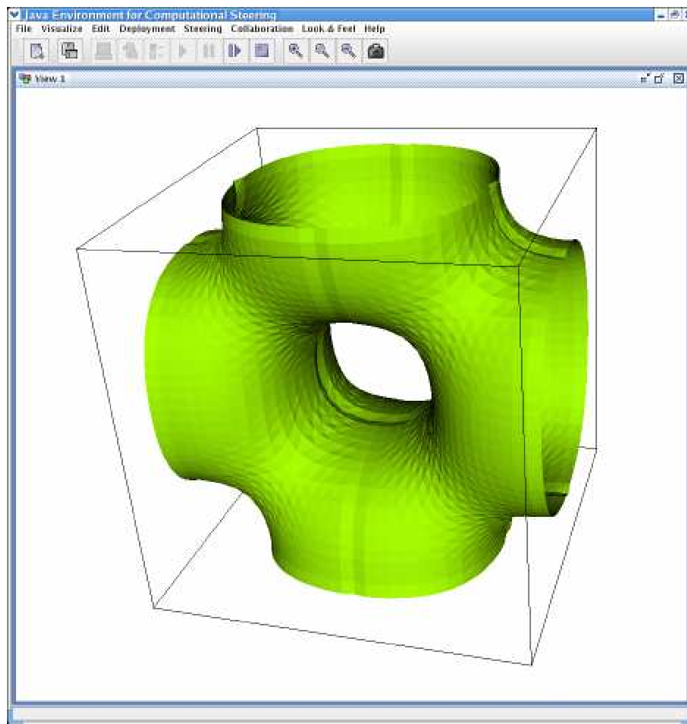


Figure 6.10: Rendering of a Jem3D computation

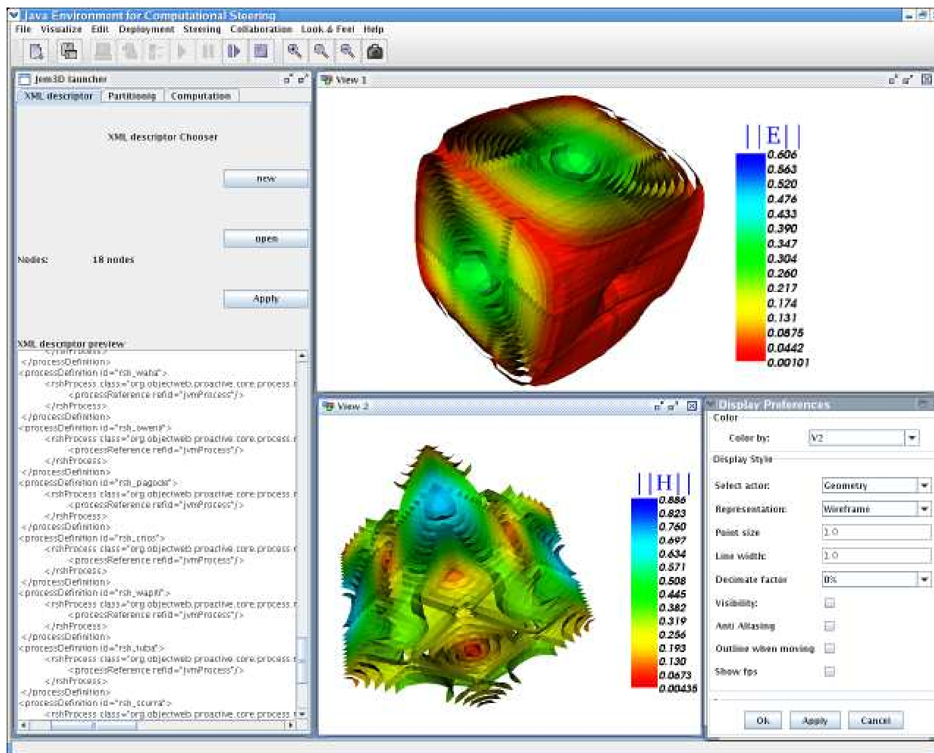


Figure 6.11: JECS: a Java Environment for Computational Steering and Visualization

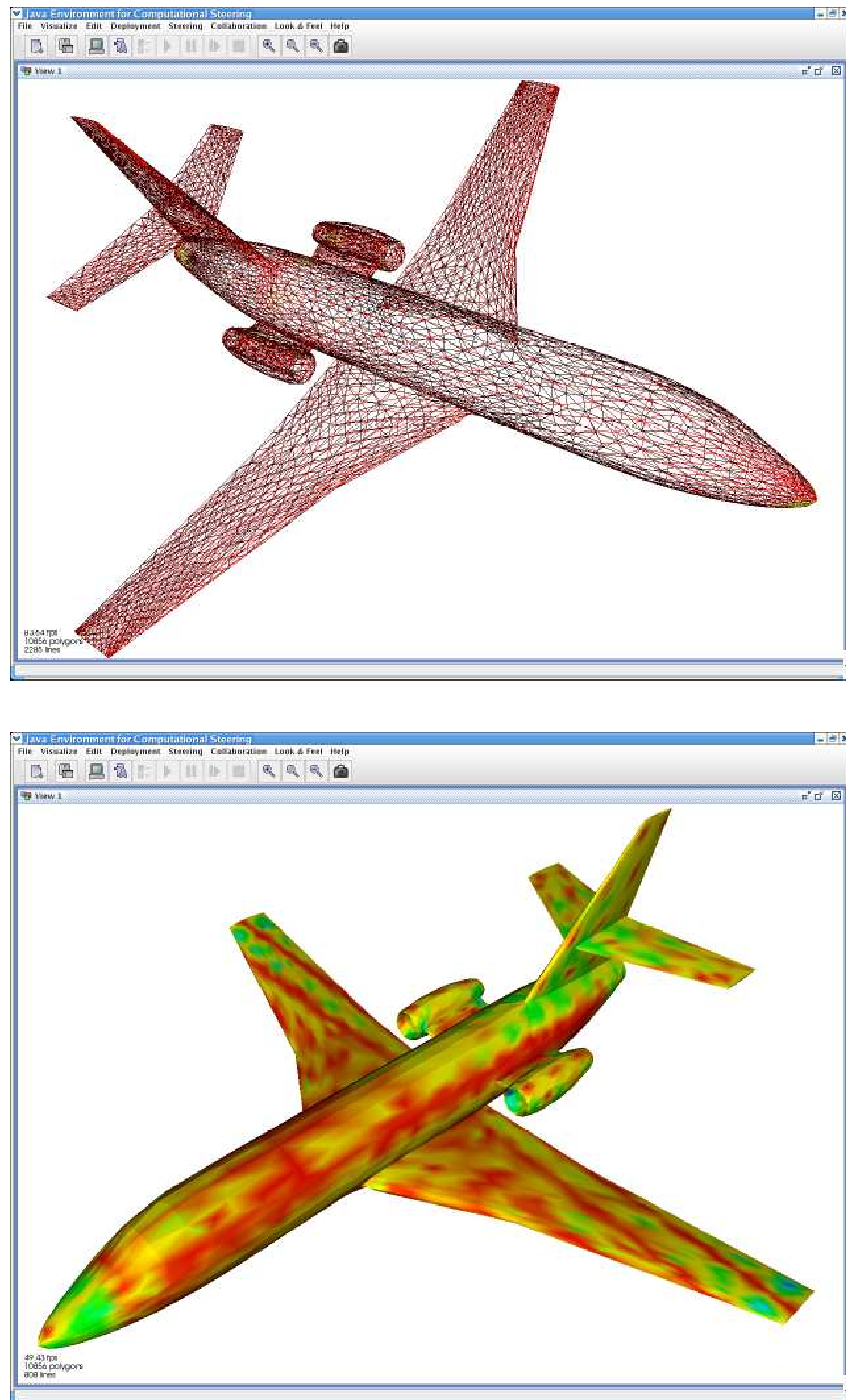


Figure 6.12: A more complex and irregular mesh

Chapter 7

Object-Oriented SPMD

Last advances in Java Virtual Machines, compilers [BUR 99, ANT 00] and communication schemes allow handling Java as a suitable environment for high-performance computing. Multithreading and remote method invocations are key features to build parallel and distributed programs in an object-oriented programming model. *ProActive* already extends those features: firstly with the creation of threaded objects (the active objects) and, secondly with asynchronous communications. The main disadvantage of RMI is that it only provides point-to-point communication with a client/server model. This is not enough: most of distributed applications require a multipoint communication model and a suitable parallel programming model, such as for instance the most popular: the Single Program Multiple Data (SPMD) one. Many projects tried to introduce MPI-like collective operations in Java. MPJ [BAK 98, CAR 00] encloses language bindings of MPI into Java. This solution does not fit well with both object-oriented model of Java and process conception of MPI. Other projects support MPI operations on Java using a C implementation bound to Java with the *Java Native Interface* (JNI). This approach cancels the *"run every where"* ability of Java and introduces an overhead [GET 99].

This chapter presents the mechanism of typed group communication as the basis of Object-Oriented SPMD, an alternative to the standard message-based SPMD programming model. While being placed in an object-oriented context, we will show that the mechanism helps with the definition and the coordination of parallel and distributed activities. The approach offers, through modest expansion of the group API, structuring flexibility and innovative implementation. The automation of key communication mechanisms and synchronization simplifies the writing of the code for the parallel activities.

Firstly, Section 7.1 presents the approaches proposed by several related projects. Then Section 7.2.1 exposes the design and the principles of our SPMD model. Section 7.2 describes the API we propose relying on the group communication mechanism. Section 7.3 presents an example and performances measurements. Finally, Section 7.4 summarizes our API by comparing it with the MPI API.

7.1 Context and related works

As a relatively straightforward object-oriented language, and with regards to recent improvements, Java becomes now a plausible basis for a scientific parallel programming language. Related works of SPMD programming mainly deals with non-object models, based on message passing. However some projects tried to introduce an object-oriented form in the SPMD model, either by maintaining message passing or by using remote method invocation.

7.1.1 SPMD programming

SPMD stands for Single Program Multiple Data. SPMD programming is a common way to organize a parallel program, on both clusters of workstations and parallel machines, and more recently also on grids [FOX 02]. A single program is written and loaded onto each node of a parallel computer. Each copy of the program runs independently, coordination events apart. So the instruction streams executed on each node can be completely different, alas for the most common pattern, i.e. master-slave, only two different streams are needed. Each copy of program (process) owns a rank number: a unique ID. The specific path through the code is in part selected by this unique ID.

Traditionally, in the SPMD model, the language itself does not provide implicit data transmission semantics. In general, the communication patterns are **explicit message-passing** implemented as library primitives. This simplifies the task of the compiler, and encourages programmers to use algorithms that exploit locality. Data on remote processors are accessed exclusively through explicit library calls.

SPMD model maps easily and efficiently to distributed and to parallel applications and distributed memory computing. The most famous environments implementing a message-passing SPMD model are PVM (Parallel Virtual Machine) and MPI (Message Passing Interface).

7.1.2 SPMD programming with an object-oriented flavor

Message-Passing SPMD

In the 1990's, due to the increasing success of object-oriented programming, many research groups have experimented the idea to both combine the usage of an object-oriented programming language (such as C++ or Java) and MPI (or PVM) for writing and running parallel and distributed applications. One of the precursors has been the MPI-2 specification itself, collecting the notions of the MPI standard as suitable class hierarchies in C++, and defining most of the library functions as class member functions. This specification has been further extended in Object-Oriented MPI (OOMPI) [SQU 96] in order to be able to deal with the transmission of objects. Essentially, OOMPI provides mechanisms to build user-defined data types according to the MPI spec, in order to represent those objects, and further communicating them. More precisely OOMPI is a class library specification that encapsulates the functionality of MPI into a functional class hierarchy to provide a simple, flexible, and intuitive interface.The MPI-1 specification, however, does not deal with objects. It only specifies how data may be communicated. OOMPI provides the capability for sending the data that is contained within objects. Moreover, since the data contained within an object is essentially a user-defined structure, mechanisms are required to build MPI user-defined data types for object data and to communicate that data in a same manner as communicating primitive data types.

Those approaches have been even further developed with the success of Java and have eventually leaded to two main categories of propositions for having message-passing SPMD within Java:

- a wrapping of the native MPI implementation library itself within the object oriented language (e.g. mpiJava [BAK 99], or JavaMPI [MIN 97] where wrappers are automatically generated)
- an *MPI-like* implementation of a message-passing specification as MPI, written using the object-oriented language itself, and available as a library. Notably, MPIJ [JUD 98] which seeks to be competitive with native MPI implementations. The most achieved is MPJ [CAR 00], in which notions such as Communicators, Datatype for the type of the elements in the message buffers, etc., are modeled as classes.

Overall, in the early 2000's, those works – done under the auspices of the JavaGrande Forum [JGF] – were considered as a first phase in a broader venture to define a more Java-centric

high performance message-passing environment. The main aim was to succeed to conciliate both performance and portability, while not departing from the consensual goal of offering MPI-like services to Java programs.

Remote method based SPMD

All propositions grounding up on remote method invocation for communication among activities take for granted that this enables the exchange of any typed data, by automatic marshaling-unmarshaling. Clearly, this better suits to the object oriented paradigm than explicit message-passing, in which send and receive must be explicitly programmed in matched pairs. One work grounding on Java remote method invocation, but generalizing it so it can support communication between more than two parties is CCJ [NEL 01]. Specifically, CCJ aims at adding collective operations to Java's object model (implementing everything on top of RMI). Parallel activities are expressed as threads groups and not as objects groups (in fact, activities in Java are expressed by threads which are orthogonal to objects). As threads may belong to several groups, this implies that any method of the CCJ API (e.g. `barrier`, `broadcast`, `reduce`, etc.) aiming at executing an MPI-like collective operation must have the reference of the group of threads as parameter (in a similar way as passing the communicator as parameter in any MPI communication). Also, in CCJ, all threads have the same program and, in particular, any collective operation must be called by all threads in the implied group.

Differently to the approach followed in CCJ, our concept for collective communications is to group Java objects into *groups*, and extend the remote method invocation mechanism such that it transparently applies to a group of possibly remote objects. It fits much better in the object-oriented approach: triggering the execution of a chunk of code (described in any public method in the class) in parallel is done simply by calling the corresponding method on the group, remotely and possibly asynchronously. By doing this, remote method invocation is exploited as the *only* communication mechanism between any numbers of remote activities.

We have experimented that having a group of objects towards which methods are invoked is a suitable OO abstraction for building *distributed* applications – even if it usually requires the additional usage of multicast delivery protocols such as causally or totally order delivery. The suitability of groups and associated group method invocation mechanisms are more rarely studied as a suitable support for parallel computing (notable exceptions being GMI [MAA 02] in Java, ARMI [SAU 03] in C++).

7.1.3 Our SPMD programming approach

As exposed in [BAD 05a], we propose a pure object-oriented SPMD programming model as an extension of our typed group communication mechanism. For this, the objects groups supporting the distributed computation will also be further organized following a topology, i.e. adding the notion of an ID for each member in the SPMD group and the way to easily reference its neighbors. Collective operations will be revisited and extended with barrier synchronization such as providing a complete *Object Oriented SPMD* model.

The SPMD programming solution we define is a smooth and perfectly integrated extension of the active object principle. We want to demonstrate to the programmer that using it, he can define programs grounded on a single concept, the *active object*. Using this paradigm, he can seamlessly target the whole spectrum of applications: from sequential mono-threaded, concurrent and multi-threaded, distributed, up to parallel and distributed ones.

GMI generalizes Java RMI. As such, it is confronted with its constraints, specially, the need for the programmer to take care of possible concurrent executions of a same method (implying to mix functional code with the usage of regular Java monitor mechanisms). On the contrary, the active object pattern is a cleaner abstraction for distributed computing, and as such should end up easier for programming Object-Oriented SPMD applications.

7.2 Object-Oriented SPMD

The proposed active objects group mechanism presented in Chapter 4 is already a usable and even efficient basis to program non embarrassingly parallel applications using a pure object-oriented paradigm, i.e. using only object-oriented method invocation for e.g. computational electromagnetism, (see [BAD 04b, HUE 04] and Chapter 6). But, some of the features specific to SPMD programming were lacking, and their addition constitutes the core of this section. We name the resulting proposition as *Object-Oriented SPMD* (OO SPMD for short).

7.2.1 Design and principles

As previously mentioned, it is possible to reproduce SPMD parallelism in a pure object-oriented way, by relying on the mechanism of a group of objects and associating a thread, an activity to each 'machine' that has to participate in the parallel computation. For instance, in GMI [MAA 02], the main thread naturally supports the task involved in the parallel SPMD computation, while all distributed threads communicate by concurrently applying methods on the group of objects. As noticed in [MAA 03], this SPMD style can even be combined with the client/server one such as to yield a mixed style, which we think is very appropriate to one of the many possible applications of grid computing: the coupling of, on one side a parallel object-oriented SPMD computation, and on the other side, an external and remote application that is in charge of, for instance, steering, visualization, etc.

In GMI, the underlying computing model is Java RMI, which is extended towards groups. But, RMI concurrency-related problems must be explicitly taken into account by the programmer. If instead, the *ProActive* active object model extended towards groups is used, concurrency management is automatic and transparent: only one request is served at a time, and the default service policy of method invocation requests is FIFO (it can be personalized if needed). So programmers should be able to concentrate on their functional code. Meanwhile, according to the active object pattern, the 'main' method is no more usable to express and run the core of the parallel task. On the contrary, the 'main' thread is devoted to support the sequential service of requests. This implies to think a bit differently about the way to express the core of the SPMD task, i.e. how the control flow dedicated to the parallel algorithm is implemented (more details on that latter).

7.2.2 Requirements

The SPMD specificities lacking in our typed group communication mechanism fall into three categories:

- identification of each member taking part in the parallel computation, and concept of member position relatively to the others; for instance a neighboring relation among members. It can be expressed with a basic ranking order or with more complex organizations such as topologies.

- expression of the program run by each member taking part in the parallel computation. In pure object groups based paradigms (e.g. as GridRPC for grid computing on Network Enabled Servers like NetSolve [NET] or Ninf [NIN]), members act in a sense as *passive* servers only activated by method calls triggered by clients. Servers do not have their own activity. On the contrary, in SPMD computing, all members are active by their own even if, for simplicity, they all execute the same program (e.g., in all flavors of MPI, in CCJ [NEL 01], in GMI [MAA 02], this program is run by the main thread on each process or participating JVM). In *ProActive*, each active object is by essence the support of a proper activity (there is no `main`, but a `runActivity` method). This activity aims at enacting the sequential service of requests (see Section 3.1.5). So, in our approach, the SPMD program will not be expressed as a classical *big loop*, but as the implicit result of a succession of request services executed in FIFO order. As will be emphasized below, this way of expressing the core

of any member's SPMD program enables behaviors pertaining to reactivity, adaptability, dynamicity usually considered to be far away from the traditional SPMD model.

- full range of collective operations (communication and global synchronization) among the members. Considering the presentation of the typed group communications in Chapter 4, only the expression of global synchronization barriers is lacking and so needs to be considered below.

7.2.3 Main principles of OO SPMD

An OO SPMD group is defined as follows: it is a group of active objects (exclusively) where each member has a reference, a group proxy, towards the group itself (see Figure 7.1). Each active object in the SPMD group is also provided with a specific *rank* in the group. With the additional feature that this reference is known by each member in the group automatically, at the time the group is created. Each member is able to access the group and to get its rank in the group. SPMD groups are not immutable. It is the programmer's responsibility to ensure that possible modifications of an SPMD group (add new member, remove member, etc.) maintain the property.

```
  // A group of type "A" and its members are created at once by
  // an external active object
Object[][] params = {{...}, {...}};
Node[] nodes = { ... , ... , ... };
A ag = (A) ProSPMD.newSPMDGroup("A", params, nodes);

  // The computation on each member may now be started, i.e.
  // invoking a method called e.g compute() defined in class A
ag.compute();
```

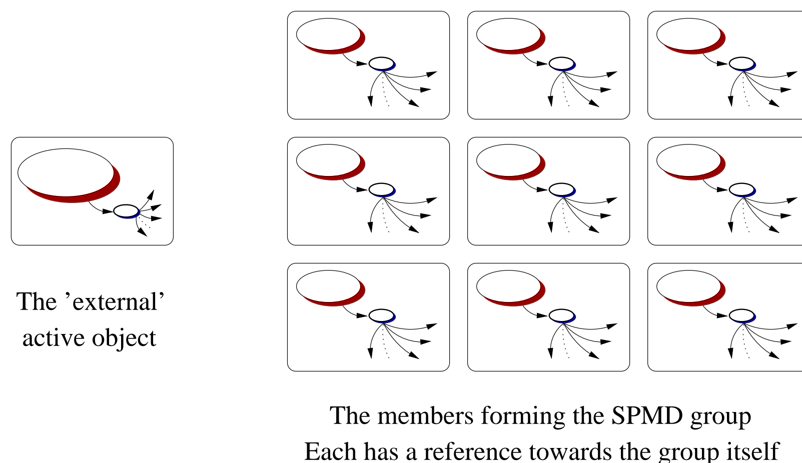


Figure 7.1: An SPMD group

On each group member created, one of the first actions to run is to get the reference of the group it belongs to, the rank, etc. One must be careful to clearly distinguish a classical Java reference to the object (this), and a *ProActive* asynchronous reference to it, as an active object. This last one enables the active object to implement the parallel task. Traditionally in SPMD, the parallel task is expressed as an iterative or recursive loop, which essentially handles message receptions and triggers the corresponding treatment, according to the message's tag (a `case` or an `if` control structure is usually programmed). In OO SPMD, the parallel task on any member of the SPMD group is run by repetitively invoking asynchronous methods to itself (so, the need to have an asynchronous reference). A member triggers data receptions and the corresponding treatment through the asynchronous service of methods remotely called by other members in the group. All method services are FIFO-ordered.

```
  // A reference to the typed group I belong to
A a = (A) ProSPMD.getSPMDGroup();
  // An asynchronous reference to myself
A me = (A) ProActive.getStubOnThis();
  // My rank in the group
int rank = ProSPMD.getMyRank();
 // Start the 'iterative' loop by sending myself
 // an asynchronous method call
me.loop();
 // To iterate, loop() again calls me.loop()
```

Notice that any method calls triggered by other members can be served between services of the method calls sent by a member to itself (the service policy is assumed to be FIFO). This is useful for effectively receiving and treating the data sent by others through remote method invocations.

Concretely, the parallel task is implemented as iterative asynchronous calls of a method (e.g. named `loop`) by the member to itself, such as to maintain an activity. This implies that the reception of data from other active objects in the system (belonging either to the SPMD group or not) is possible only between two successive services of the method (e.g. successive services of `loop`): indeed, receiving such data is effective by serving the corresponding request that is next in the request queue. This implies that any wait in the loop is prohibited if the member wants to receive data from other members. Of course, triggering the next loop pertaining to the activity must be done through the asynchronous reference of the member, never through `this`.

Moreover, in a traditional SPMD program, execution control is exclusively based on `if` statements and process ID or rank numbers. In our approach, switching execution control can be also based on dynamically created groups at any moment at runtime. Such groups can be derived from existing ones (sub-groups, or group combination for instance) or according to any kind of properties (rank, fields of the object, etc.).

7.2.4 Topologies

To simplify the access to neighbors in the group with which a given member must communicate according to the parallel algorithm, it is useful if the SPMD group is further organized according to Cartesian topologies (as in MPI). At this time, we offer the following: line, plan, ring, cube, hypercube, torus, torusCube (torus in 3 dimensions) and tetrahedron but, contrary to statically designed topologies, the addition of new topologies is open. Figure 7.2 presents possible logical organization given to a group through topologies. Topologies may also be obtained from another topology by combination or extraction.

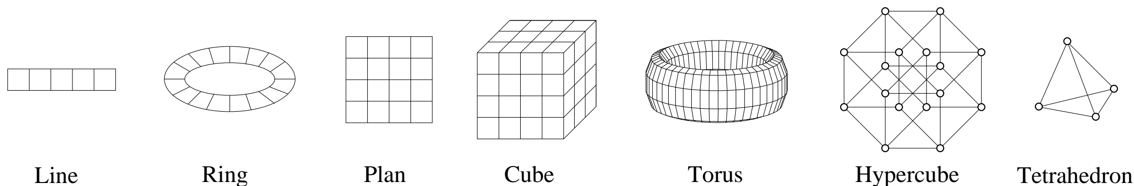


Figure 7.2: Topologies

Topologies are groups: any existing group may be understood as a topology. Creating a topology from a group allows to access to a specific set of methods and introduce the neighborhood relationship between group members. This property is translated by inheritance in an object-oriented framework like ours: the `Topology` abstract class inherits from the interface `Group` [1].

[1] More precisely, the `Topology` class extends the `ProxyForGroup` class that implements the `Group` interface.

Figure 7.3 presents the class hierarchy of already existing topologies. `Topology` or any other classes can be extended to create new topologies or to redefine the access method to the neighbors. 3Dimensional structures (`Cube` and `TorusCube`) extend 2Dimensional ones (resp. `Plan` and `Torus`) that themselves extends 1Dimensional structures (resp. `Line` and `Ring`): width, height and depth are successively added to go from a 1D to 3D logical representation of activities organization and interaction.

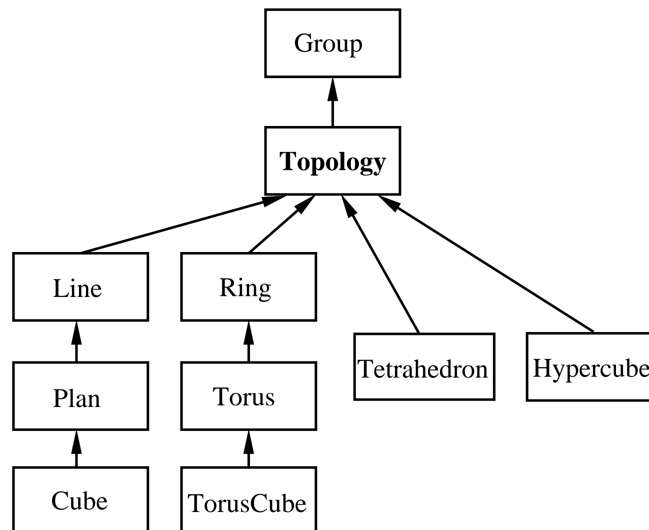


Figure 7.3: Topologies classes

A topology is built by copying a group: it is quite similar to a copy-constructor. References to the group members (and not the members themselves of course) contained in the group are copied to the newly created topology. The group and the topology become two distinct objects, so the modifications performed on one object is not reflected on the other. Here is a topology creation example using the previously obtained SPMD group `a`:

```
  // Organize my group as a 2D plan
Plan topology = new Plan(a, aWidth, anHeight);
```

The topologies provide methods to easily access the neighbors of a specified activity, i.e. to access to the activities which are the most interacting with the given activity. The set of methods depends on the topology. For instance, a `Line` topology provide `left` and `right` methods while a `Cube` topology provide `left`, `right`, `up`, `down`, `ahead`, `behind`, `line`, `plan`, etc. Those methods ease the conception and the organization of distributed applications by avoiding long, painful and error-prone codes which manipulate group index in order to emulate a more complex structure. Additionally all topologies provide a `neighbors` method which returns a group composed of the closer objects of a given member. For instance, the `neighbors` method of `Line` returns a group composed of the left and right neighbors; the `neighbors` method of `Plan` returns a group composed of the left, right, up and down neighbors. The notion of neighborhood is strongly attached to the topology. By extending a topology, the programmer may reformulate the neighborhood definition to best fit the needs of the application. Here is a basic example about how to get and communicate with neighbors:

```
  // Get a reference to my neighbors in the plan
A left = (A) topology.left(me);
A down = (A) topology.down(me);

  // One-way communication with neighbors in an asynchronous fashion
left.foo(params);
down.foo(params);
```

There are two ways to obtain Topology objects. The first one is to explicitly invoke a constructor with the new operation. The second one is to extract a new topology from an existing one. It is obvious that a plan can be considered as a set of horizontal or vertical lines, for example. Many topologies provide methods returning topologies. By extracting sets that possibly have a common property, such methods contribute also to easily build distributed applications. Here is an example:

```
  // Get a reference to the topology formed by the first line of
  // the plan
Line line = topology.line(0);
  // Get a reference to the topology formed by the first column of
  // the plan
Line line = topology.column(0);
```

As topologies are groups, and any group is also a typed group, all topologies can be viewed as typed group: in this case we can also call it a *typed topology* for a better understanding. The getByType method converts the topologies into typed groups (i.e. typed topologies), as presented with standard group mechanism in Section 4.2.2. Symmetrically, the getGroup static method of the ProActiveGroup class does the opposite: it gives a group (assignable to a topology) from a typed topology. Like a typed group, a typed topology exposes a common interface of its members. Method invocations achieve communication towards group members. Already presented group communication semantic is applied to perform such method call. The following example presents communications addressed to topologies:

```
  // Convert the topologies into typed topologies
A gplan = topology.getByType();
A gline = line.getByType();

  // communicate with the topologies member
gplan.foo();
gline.foo();
```

7.2.5 Synchronization barriers

The only collective behavior related methods of our OO SPMD API pertains to global barriers. Indeed, as already explained in Section 4, all collective (resp. point-to-point) communications within the group can be expressed as applicative-level method calls triggered via the group proxy (resp. via the asynchronous reference of the target group); so only the coordination of those activities requires additional methods.

The standard definition of a global barrier is that all members in the group (or those enrolled in the barrier, see below) must not proceed further in their computation while not all the members have reached the barrier. Given the active object model, we propose a slightly different but more appropriate semantic: from the viewpoint of a member reaching a barrier, it is effective (i.e. it blocks the member) only in the future: more precisely the exact moment when the current service has terminated. In practical terms, all instructions lying after the barrier in the current method being served will be executed, so care must be taken (see an example in Section 7.3.2). Nevertheless, the meaning of what is a global synchronization barrier is as usual, but instead of pertaining to the next instruction, it pertains to the next request's service: when encountering a barrier, the service of the first request incoming from an SPMD group member and waiting in the request queue will be able to proceed on any enrolled member only when all have reached the barrier. As mentioned further, requests originated from "external" objects may be served.

Technically, when an active object executes a call to a global barrier this triggers the storage in the front of its request queue of a specific token. Associated to this token is the total number of members (including the member itself) to wait for, i.e. that must reach the barrier. Each time a given global barrier is reached by a member, this triggers the decrement of this number on each

member enrolled in the barrier. Eventually, the barrier is released on each enrolled member, as soon as the number reaches zero. An activity that has invoked a barrier tags its outgoing requests with the barrier ID until it blocks. These outgoing requests that are triggered after the barrier has been crossed in the same service execution. This tag indicates to the request receivers that the request must not be served before the specified barrier has been released.

A suspended activity remains able to receive requests and put them in queue. In that way, all objects are allowed to communicate with the suspended object. Even if those methods will not be served instantaneously, the sending overlaps the wait. A barrier is limited to its participants. It means that requests sent by an object not belonging to the SPMD group involved in the barrier can be served, even if the activity is blocked by a barrier.

Actually, we propose three kinds of barriers, two global and one more local:

- First, a *total barrier*, within which a string parameter represents a unique identity name for the barrier. It is assumed that this blocks all the members in the SPMD group.

  ```
  ProSPMD.barrier("MyBarrier");
  ```

- A *neighbor barrier*, involving not all the members of an SPMD group, but only the active objects specified in a given group. Those objects, which contribute to the end of the barrier state, are called neighbors as they are usually local to a given topology. An active object that invokes the neighbor barrier must be in the group given as parameter.

  ```
  ProSPMD.barrier("bar", neighborsGroup);
  ```

  It is interesting to notice that the following instruction:

  ```
  ProSPMD.barrier("bar", ProSPMD.getSPMDGroup());
  ```

  is similar to a total barrier call. If the neighborhood involved in a neighbor barrier is the whole SPMD group, then the neighbor barrier becomes a total barrier.

- A *method barrier* stops the active object that calls it, waiting for a request on all the specified methods to be served. The order of the methods does not matter, nor the active objects they come from. As such, this barrier is purely local, and does not trigger extra messages to be exchanged as the two others.

  ```
  ProSPMD.barrier({"foo","bar","gee"});
  ```

  One may want a sequential treatment of the methods, it means block the current activity until, first `foo` has arrived, then `bar`, then `gee`. To achieve this, just invoke several times the method barrier in the desired order, just as follow:

  ```
  ProSPMD.barrier({"foo"});
  ProSPMD.barrier({"bar"});
  ProSPMD.barrier({"gee"});
  ```

  Actually, a method barrier needs not the involvement of the SPMD group or of a neighbor group. In opposition to the previous barriers (total and neighbor), the method barrier blocks the service of all requests are they originated from a member of the SPMD group (or neighbor group) or not.

Of course, none of those barriers is implemented with an active wait. Resources are not consumed while waiting. The activity passively waits for the condition to be satisfied to resume.

7.2.6 Extensibility and reactivity

High performance computers and high speed networks provide now a sufficient level of power to consider coupling numerical codes. It is no more about run only one code, but several, collaborating together in order to produce a more precise result, involving a bigger amount of concepts.

Software environment must provide the possibility to integrate and couple several numerical codes. This integration needs parallel and distributed mechanisms. Parallelism to address performance issues, distribution to satisfy resources and security requirements.

This necessity to unify parallelism and distribution has implication on the programming model. Two approaches emerge:

- Extend parallel programming in order to take account of the distribution. The matter is to increase the functionalities of a parallel environment. For instance, make the different codes communicate through existing message passing libraries.
- Use distributed programming and its communication mechanisms, such as distributed objects and remote procedural call, for parallel programming.

In the first case, extensions do not allow a software component programming model. Parallelism and distribution use the same communication mechanisms, it makes the codes difficult to maintain and without any clear interface to use them in application requiring coupling. On the other hand, the second case is able to handle such problems. Sure, it imposes the use of a specific communication model.

Thanks to the queuing of requests, the model we propose allows our objects to remain available for external applications. An active object member of an SPMD group is still responsive to requests coming from an object outside the SMPD group. Requests coming from other codes may entwine with the requests of the application. This property is very interesting to monitor the application at runtime for instance. This feature represents a step to a more wide-ranged scope of coupled applications.

7.3 Example and benchmarks

We illustrate OO SPMD with a concrete example. We choose *Jacobi iterations* because it is a simple application, easy to distribute in a traditional SPMD manner. The algorithm performs local computation and communication to exchange data. The Jacobi method is a method of solving a linear matrix equation. Each element is solved by computing the mean value of the adjacent values. The process is then iterated until it converges; it means until the difference between old and new value in absolute becomes lower than a given threshold.

The following code shows the main loop (an iteration based loop) of a solver. In each iteration, the value at a point is replaced by the average of the up, down, left, and right neighbor values. External boundary values are fixed statically at the beginning of the application and do not change at runtime.

```
while (!converged) {
  for (y=1 ; y<MATRIX_HEIGHT-1 ; y++) {
    for (x=1 ; x<MATRIX_WIDTH-1 ; x++) {
      new(x,y) = ( old(x,y-1) + old(x,y+1) +
                   old(x-1,y) + old(x+1,y) )/4;
      if (abs(new(x,y)-old(x,y)) < THRESHOLD) {
        converged = true;
      }
      exchange(new,old);
} } }
```

The structure of this code is quite simple, so we use a coarse-grained data-parallel approach to transform it into a similar parallel code. The arrays `old` and `new` are distributed over nodes taking the form of active objects. Each active object, named `SubMatrix`, is responsible for receiving boundary values from adjacent sub-matrixes and computing its own part of data.

The parallel algorithm depends on the data distribution scheme. We choose a two-dimensional distribution scheme. It is essential for the data distribution to be correctly balanced between nodes. It allows to minimize the amount of data exchanged and to allocate a good amount of data. Figure 7.4 illustrates two possible distributions on 6 nodes. The first one is a one-dimensional distribution, where the matrix is partitioned in *stripes*. The second distribution scheme is two-dimensional; the matrix is partitioned in *squares*.

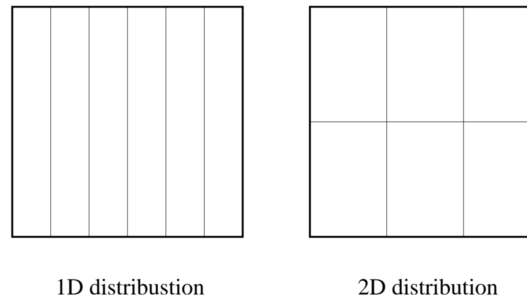


Figure 7.4: Data distribution schemes

As shown in Figure 7.5, communications occur at block boundaries. So the amount of data exchanged is minimized by the two-dimensional distribution which has a better internal area / border ratio. With this partition, each sub-matrix may communicate with two, three, or four neighbors, depending of their position (respectively at a corner, a border, or in the center of the whole matrix). This partition is more effective when the data to processor ratio is large.

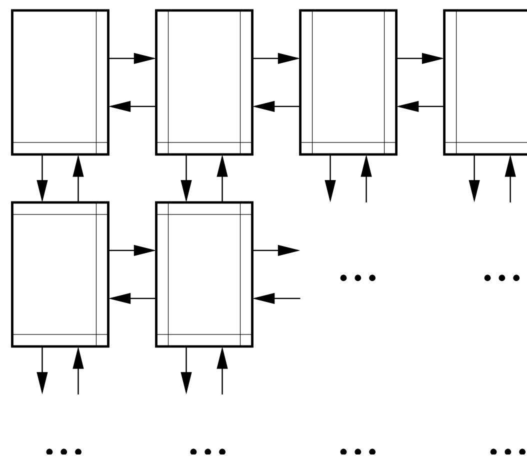


Figure 7.5: Distributed algorithm

Communications appear at sub-matrix boundaries to send boundaries values to neighbors and receive values from neighbors. A copy of the boundary of each sub-matrix is present in its neighbor sub-matrix. Storage of boundary data is allocated at the producer, and at the consumer sub-matrixes. This is a static allocation because the size and the location of boundary buffers is fixed and never evolve during Jacobi. This is induced by the Jacobi algorithm itself, but if needed, our framework could support strongly dynamic algorithms.

7.3.1 MPI Jacobi

Using a message passing approach based on asynchronous send and receive, with the MPI library, the resulting parallel code is something like:

```
while (!converged) {
  internal_compute(&converged);
  MPI_Send(north_border, SUBMATRIX_WIDTH,
           MPI_DOUBLE, north, 1,
           MPI_COMM_WORLD, &status);
  MPI_Recv(border_received_from_north,
           SUBMATRIX_WIDTH, MPI_DOUBLE,
           north, 1, MPI_COMM_WORLD, &status);
    // send and receive for south, east, west
  ...
  boundaries_compute(&converged);
  exchange(new,old);
} } }
```

The send and receive operations are repeated for each communication with a neighbor (up to 4), even if the operations are the same.

7.3.2 OO SPMD Jacobi

Using our OO SPMD approach, the code becomes much more concise. The whole matrix is distributed and understood as a two-dimensional topology using the `Plan` topology. The neighborhood of any `SubMatrix`, named `neighbors` in the example, is automatically obtained through methods of the `Plan` topology.

In a first time, we obtain a reference on the activity itself to be able to asynchronously invoke method on it (put the requests in queue instead of execute them instantaneously).

```
   // Gets a reference to the active object itself
SubMatrix me = (SubMatrix) ProActive.getStubOnThis();
```

Three implementations of a Jacobi iteration present the three barriers. `jacobiIteration` is the name of the method that does an iteration.

- The total barrier requires that all sub-matrices invoke the barrier before continuing the execution and thus recursively go to the next iteration. Thereby, all sub-matrices are synchronized: they are all in the same iteration and simultaneously go in the next iteration.

```
public void jacobiIteration() {
  internal_compute(); //updates converged
  neighbors.send(boundariesGroup);
  ProSPMD.barrier("barrierUID"); // invoke a total barrier
  me.boundaries_compute(); //updates converged
  me.exchange();
  if (!converged) me.jacobiIteration();
}
```

  The total barrier is not very well adapted to the Jacobi iterations. Unconnected sub-matrices wait each others even if they have no interaction.

- With a neighbor barrier, a sub-matrix resumes its activity as soon as its neighbors and itself have invoke the barrier.

```
public void jacobiIteration() {
  internal_compute(); //updates converged
  neighbors.send(boundariesGroup);
  ProSPMD.barrier("barrierUID", neighbors); // a neighbor barrier
  me.boundaries_compute(); //updates converged
  me.exchange();
  if (!converged) me.jacobiIteration();
}
```

The neighbor barrier produces only to the objects members of the neighbor group. It sends less messages on the network, and furthermore, allows two neighbors to have one iteration in difference.

- The method barrier does not necessarily relate to the SPMD group or to the neighbors. It waits for the specified methods to resume the activity.

```
public void jacobiIteration() {
  internal_compute(); //updates converged
  neighbors.send(boundariesGroup);
  ProSPMD.barrier({"send", ... , "send"}); // a method barrier
  me.boundaries_compute(); //updates converged
  me.exchange();
  if (!converged) me.jacobiIteration();
}
```

In our example, the number of `send` methods to wait for, depends on the number of neighbors that sends their boundary values to the activity. As already mentioned, a method barrier does not emit any message.

The three implementations are very similar. Actually only the line where we invoke the barrier changes, and thus modifies the way the barrier is preformed. In both of them, the variable `converged` is updated by the `compute` methods. As soon as one activity sets `converged` to `true`, it sends a “stop” message to all other activities using a group communication in order to end the computation.

Synchronization is done by data flow, and the barrier ensures that the sub-matrix and its neighbors have exchanged their own boundaries values before computing the whole boundaries. The method calls performed after the barrier must be asynchronous (put in the queue of the active object); otherwise they would be served immediately, i.e. before the execution of the barrier. Overall, according to the semantic of the barriers, the data (i.e. parameter of `send`) will have been exchanged before the barrier will be released, guarantying that any member gets the data in order to compute the boundaries values (`boundaries_compute`).

Data communication to all neighbors is performed using a *scatter group* (the group of boundaries values `boundariesGroup`): as the real parameter of the *send* method is a group declared as of type *scatter*, it is transparently scattered to each member of the neighbors group. As for the MPI version, the construction of the structures containing boundaries values was not specified on this chunk of code. It only consists of building a group containing the boundaries.

A very interesting property of our model is that it remains *reactive*. It means that any part or any member of an OO SPMD application may also serve incoming method call requests incoming from another application. This is allowed by the fact that the parallel task is expressed as asynchronous calls to a method (`jacobiIteration` for instance): an external request is thus able to come in between requests addressed to the active object. We think this flexibility is very appropriate to one of the many possible applications of grid computing: the coupling of, on one side a parallel object-oriented SPMD computation, and on the other side, an external and remote application that is in charge of, for instance, steering, visualization, etc.

7.3.3 Benchmarks

Data scalability

The first benchmark, presented by Figure 7.6, uses a cluster of 16 bi-Pentium III @ 933 MHz 512MB (SDRAM) - 256 Kb L2 cache, Linux RedHat 2.4.20, interconnected with a 100 Mb/s Ethernet. Even if machines are bi-processor, we used only one processor per machine during our experimentations. For the C/MPI version we used gcc 3.3.2 and MPICH 1.2.5.2. For the Java version, we used the Sun Java Virtual Machine 1.5.0.

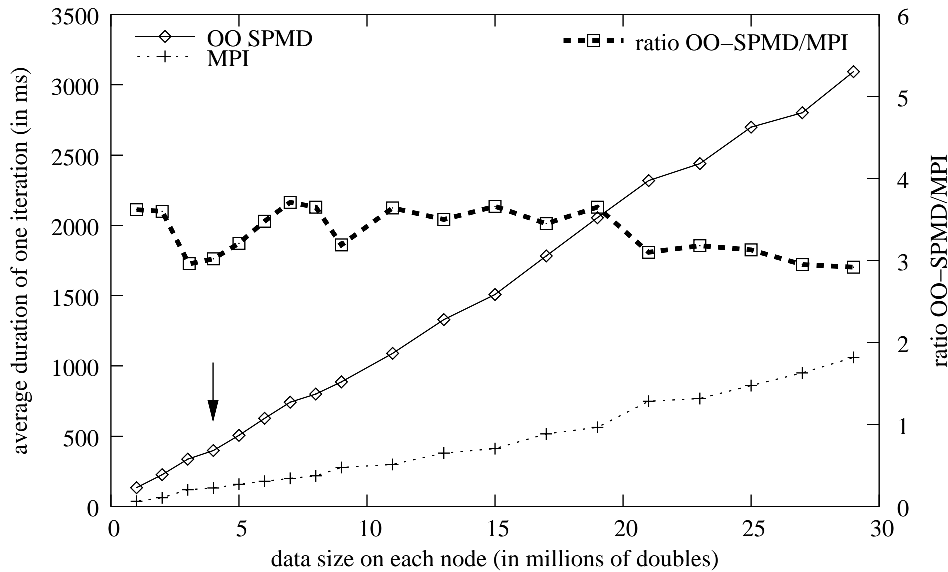


Figure 7.6: Performances of C/MPI and Java/OO SPMD versions

The graphic presents the average duration, in milliseconds, of one Jacobi iteration depending of the data contained on each node, in millions of double. The average time was computed after 100 iterations. Of course, the C language with MPI remains more efficient than Java with RMI. But the ratio of 3.3 (average) of performance is maintained despite the growth of data (see the bold curve). It is interesting to notice that 3.3 is also the ratio of performance between Java and C for the sequential versions. Our approach thus allows an efficient distribution and is scalable regarding data. For 29M of doubles, speedup of the C/MPI version is 15.41; speedup of the Java/OO SPMD is 15.23.

Deployment scalability

Figure 7.7 presents the Jacobi application running on up to 130 machines. This experimentation was done using a Peer-to-Peer deployment scheme provided by *ProActive* within an Intranet configuration. The machines used are desktop computers, simultaneously used by their users. They are heterogeneous (slowest is a Pentium III @ 993 MHz 512MB, fastest is a bi-Pentium IV @ 3,2 GHz 2GB), they are interconnected with 100 Mb/s network, they are running under Linux (with different kernel versions). Deployed applications run with a lower priority (nice 19) in order to not disturb regular users. We used the Sun Java Virtual Machine 1.4.2.

To further analyze the performances, it is important to notice that for all measurements, each node is responsible for the same amount of data (2000x2000 doubles). The overall size of the problem grows with the number of nodes involved in the computation. The line plots the average duration, in milliseconds, of one Jacobi iteration depending of the number of nodes involved. As previously, the average time was computed after 100 iterations.

Compared to the previous benchmarks, for the same amount of data per node (see arrow in Figure 7.6), execution is 7 times slower. We blame the lower priority of execution and the older JVM for this loss of performance. Besides, the performance remains regular, regardless of the number of used nodes. From this, we conclude that the application is scalable.

7.4 Comparison with the MPI API

We do not try to fit to the exact syntax of MPI. Our choice is to benefit from the typed syntax of the group communication. In MPI, heavy-weight processes communicate with point-to-point

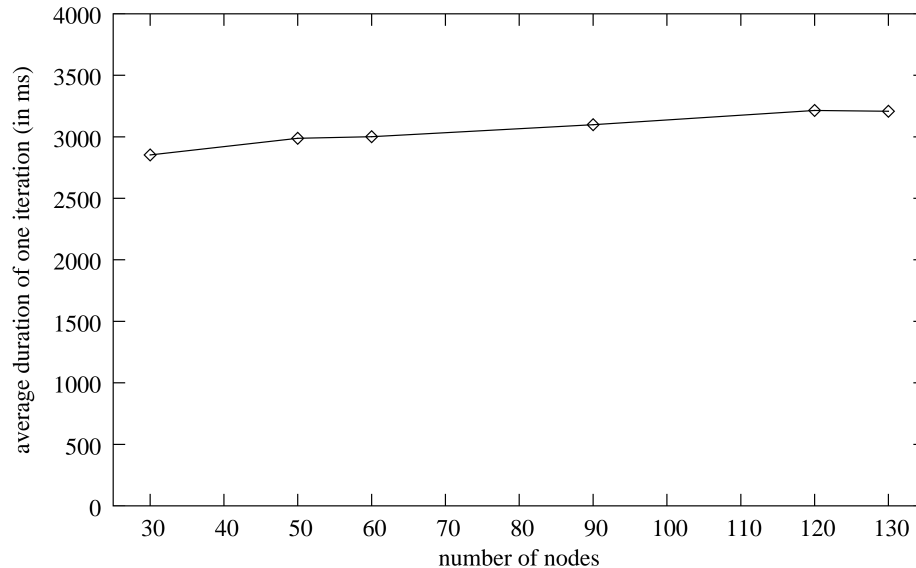


Figure 7.7: OO SPMD scalability in a peer-to-peer experiment

and collective operations through communicators. A communicator is a collection of processes ordered by their rank. In a communicator any process is able to retrieve its rank and the number of process members of the communicator. All processes belong to the communicator named `MPI_COMM_WORLD`. New communicators can be built from existing ones but all communicators are immutable: once created, they can not be modified.

The Java multithreading model and the MPI heavy-weight process do not fit very well. However we can map easily the concepts of the *ProActive* library and its group communication onto the concepts of MPI. Naturally, our OO SMPD translates processes into active objects. Communicators are translated in SPMD groups.

We consider the following methods as the most significant of MPI: `MPI_Init` and `MPI_Finalize`, `MPI_Comm_size`, `MPI_Comm_rank`, `MPI_Send` and `MPI_Recv`, `MPI_Barrier`, `MPI_Bcast`, `MPI_-Scatter`, `MPI_Gather`, and `MPI_Reduce`. Let us see how those methods are translated in our framework. We begin with the external user command `mpirun` that initiates the processes.

`mpirun` is responsible for the instantiation of the processes. `mpirun` is a user command so the creation of processes is a static operation placed outside the source code. There is no equivalence for `mpirun` in our API. However, we can assimilate this command to the dynamic creation of SPMD groups. In our mechanism the creation of activities is explicitly done in the source code with the `ProSPMD.newGroup` primitive.

`MPI_Init` initializes the MPI execution environment. `MPI_Finalize` terminates it. Those functions, essential in all MPI programs, have no direct translation in our API because we do not use a special environment to operate SPMD operations.

`MPI_Comm_size` determines the number of processes associated to a communicator. MPI's communicators are represented by groups in our API, so the method `getMyGroupSize` translates `MPI_Comm_size`. This method, provided by the static class `ProSPMD`, returns the size of the SMPD group the activity belongs to.

`MPI_Comm_rank` specifies the rank of the caller process in a communicator. In our API, the method `getMyRank` provides the same information. `getMyRank` is a static method of the `ProSPMD` class.

`MPI_Send` sends a message to a process. This message is effectively received by the recipient when this one invokes the `MPI_Recv` method possibly specifying the identity (the rank) of the sender and the number identifying the message. MPI provides many primitives for sending and receiving messages regarding be they synchronous or not, blocking or not, etc. [2]. In our framework, the invoked method automatically adapts the communication scheme, so the programmer does not have to care about the correct scheme of communication to use. Simple functional methods allow communicating data. For instance, a *getter* returns data to the caller. The caller initiates the communication in a *pulling* way: an activity asks for the data to another activity. On contrary, a *setter* sends data (embedded as a parameter) to a remote activity. In this case the initiative comes from the activity that owns the data. It is a *pushing* approach. A method using parameters and returning results allows a simultaneous bilateral communication.

`MPI_Barrier` stops the execution of the process that invokes the method until all other processes of the communicator reach the same instruction. The `ProSPMD` class supplies not only one method `barrier` but several. We have seen previously the difference between those different versions. We aim to provide more efficient barriers, with strong and weaker semantics. Two characteristics define our barriers. The first one is the ability to maintain a reception activity (receive requests and put them in queue) while the service activity is locked by a barrier. It allows to overlap the wait for the end of the barrier with message receptions. It allows also to serve requests coming from external objects that do not belong to the SPMD group, and consequently do not take part in the SPMD group activity. The second one is the possibility to block on a specific method invocation, thanks to the method barrier, and thus to do not produce additional messages to manage such barrier.

`MPI_Bcast` sends a message from a process to many processes through a communicator. Every process member of the communicator receives a copy of the message. Obviously, in our framework, a typed group communication provides such kind of communication: a copy of the parameters of the invoked method is sent to every member of the SPMD group.

`MPI_Scatter` sends the same messages with different data to many processes. The data exchanged from the sender to the receivers are fairly scattered between all the receivers with a regular split depending on the number of receivers. A typed group communication, using another scattered typed group as parameter translates the `MPI_Scatter` primitive. As presented in Section 4.2.6, the parameter group is regularly scattered between the members of the callee group depending on the rank of the objects. Notice that the sharing of data is done according to the structure of the group build by the programmer and given as parameter, so it allows a very fine control on the sharing of data.

`MPI_Gather` gathers data from the processes that belong to a communicator. Of course, such operation is performed, once again, by a typed group communication in our framework. The result of a group communication is local to the caller, so a group communication behaves like the gathering operation of MPI `MPI_Gather`. A result group is composed of *futures* that seamlessly gather the results in an asynchronous way.

`MPI_reduce` combines data from the processes that belong to a communicator using a predefined operation (min, max, sum, product, ...). Because our model deals with objects and no more with primitive types, we leave the programmer write the combination method. After that, each object of an SPMD group invokes this method specifying an object that will store the temporary result until every group member has returned its reply. Then the external object may return the final result to all members of the group. Another solution would be to perform a gather operation, and to iterate over all elements in order to combine them with the programmer's method. This second method is less parallel than the first one. In both cases we assume that we do not benefit from binomial propagation and combination of the results that most of the MPI implementations

[2]For instance: `MPI_Send`, `MPI_Bsend`, `MPI_Ibsend`, `MPI_Irsend`, `MPI_Issend`, `MPI_Ssend`, etc.

provide. The group behavior component we are currently adding in the *ProActive* library is a third integrated solution to perform a reduce operation; it is presented in Section 8.1.

Figure 7.8 summarizes the translations of the MPI's primitives into our framework. It clearly exposes that the large amount of primitives used in MPI can be hugely reduced in our object-oriented framework for SPMD programming. The programmer only focuses on the functional code and forgets the troublesome manipulation of many communication primitives.

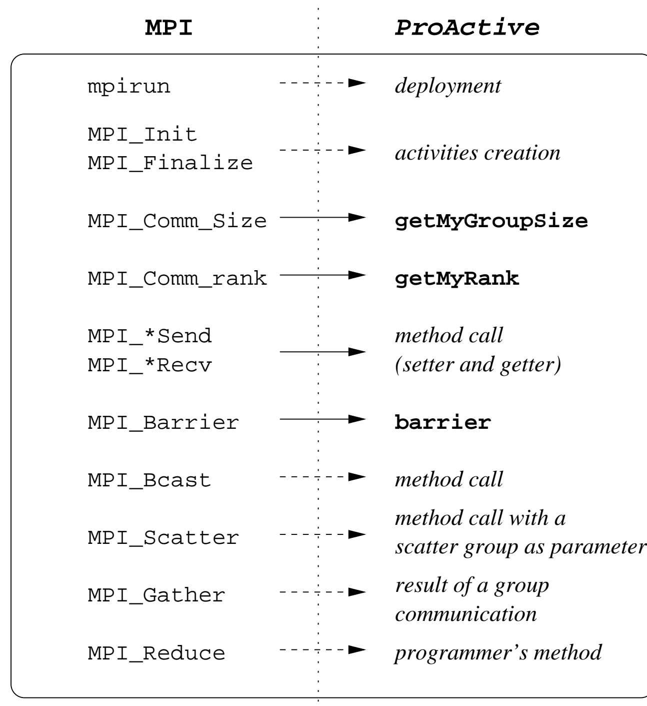


Figure 7.8: MPI to Java translations

Contrary to MPI that requires all members of a group to collectively call the same communication primitive, our group communication scheme lets the possibility to one activity to call a method on the group. MPI deals with simple data structures (arrays) while RMI is aware about complex objects. Organize the data in arrays to be exchanged through collective operations is painful. The programmer benefits from the remote method invocation scheme to make the parallel activities easily communicate.

Conclusion

We have introduced a parallel programming model, which we name *Object-Oriented SPMD* as an alternative to the traditional Message-Passing SPMD style. Overall, it allows more flexibility, and a higher level of abstraction. Firstly, it enforces members taking part in the computation just the required involvement in collective operations. E.g. in MPI, a call to `MPI_broadcast` must be run by all members, even if for all except the sender, this call aims only at receiving the message. On the contrary, using our solution, a method invocation towards a single active object to trigger a point-to-point interaction, or towards a SPMD group of active objects to trigger a collective interaction between all the members only differ by the target object reference. This way, we promote asynchronous remote method invocation and the active object pattern as the only required communication and structuring mechanism. Secondly, our approach to SPMD programming has potential for evolution. Instead of defining the parallel task as a single 'big' loop as in traditional SPMD programming, OO SPMD enables to receive and treat data in a more flexible order (discarding the need to program sometimes intricate `case` statements depending of received message's tag).

Chapter 8

Ongoing and collaborative work

This chapter introduces ongoing collaborative work based on the works presented in the previous chapters. The group communication mechanism may benefit from new enhancements, in term of language expressiveness and eventually in term of performance on high-speed and multicast networks as well. The group mechanism has also been useful in order to build components and peer-to-peer overlays networks for distributed computing.

The chapter is organized as follow. Firstly, Section 8.1 presents an extension to the group API that allows more precise and adaptable behavior of group communication. In Section 8.2, IP multicasting is introduced in the group communication mechanism. Section 8.3 shows how a typed group communication is useful to build the notion of composite component with a multi-port. Finally, Section 8.4 presents the use of the group communication in a peer-to-peer infrastructure for grid and cluster computing.

8.1 Group behavior component

Grid applications typically deal with huge amount of data and often the same data have to be transferred and processed on many resources. Nevertheless, the majority of existing middleware platforms for Grid computing does not provide suitable programming and communication models to make easy software development and to improve communication performances when a large set of receivers is involved. Some middlewares for wide area network computing, such as *ProActive*, provide the group abstraction to transparently deal with a number of similar receivers. We propose an extension of such a mechanism in order to improve its features for Grid environments. In particular, *ProActive* groups have been extended both at programming and communication levels in order to support different internal behaviors. Those works were done in collaboration with Nadia Ranaldo and Eugenio Zimeo [BAD 05d].

In order to simplify distributed programming, more abstractions and high-level distributed models should be provided by a group communication mechanism at programming level, in order to free the programmer from the implementation details of system aspects of programming such as object distribution, mapping and load balancing mechanisms. This leads also to a performance improvement, thanks to the possibility to automatically and transparently adapt the application to the system configuration.

We propose to extend the syntax of group creation and to change the syntax and semantics of group management. To this end, we introduce a dynamic internal behavior, called Group Behavior, for each *ProActive* group, so as to define the semantics adopted by the group for a method invocation. Through the definition of a behavior and its dynamic assignment to a group, this one can change its internal behavior at runtime and new policies can be easily implemented and attached without interventions on the library or even on the application code. In fact, through the Java reflection, a newly created group behavior can be loaded during the program execution

to install a different behavior in a running group. This way, a group can transparently adapt its behavior to the context in which it operates.

In recent years, several group semantics have been defined. Each of them contributes to specify the behavior of a group. In particular, from the point of view of the method invocation the following semantics can be individuated:

- ***Request mapping***: it handles the mapping of each request to the group members. Some examples are *(1) One*, the request is assigned to only one group member, selected with a scheduling policy (for example random, round-robin, more sophisticated policies based on QoS) ; *(2) Fixed*, the request is scheduled for a defined number of group members; *(3) All*, the request is propagated to all the group members.
- ***Input parameters distribution***: it allows for splitting the input parameter of each group method before sending the request to the group members selected for the request mapping. Examples are: *(1) Broadcast*, an input parameter of the method invocation is sent to all the scheduled group members; *(2) Scatter*, a group that receives the invocation of a method could be able to split the value, received as parameter, in a number of chunks and to pass each one to the same method of each member.
- ***Output parameters collection***: it handles the return value replied to the caller. Examples are *(1) Gather*, the output parameter is obtained collecting the partial results of the group members; *(2) Merging*, the output parameter is obtained by assembling the partial results of the group members.
- ***Synchronization***: it specifies the condition that blocks the caller when a return parameter of a group method invocation is used. *(1) All*, the totality of the scheduled group members execute the request, and all the results are to be collected and returned to the caller; *(2) Majority*, the execution request is active until the majority of the scheduled group members have executed the request and replied the results; *(3) One*, in this case, groups can be used to improve the reactivity related to the processing triggered by a method of the group by moving the invocation to all the scheduled members and collecting the result coming from the most reactive or the nearest member; *(4) Fixed*, a number of executions specified by the user are required.

From the point of view of communication inside a group, the following schemes can be adopted:

- ***Unicast***, a point-to-point communication. In this case each member is contacted separately in order to receive different input data.
- ***Multicast***, a point-to-multipoint communication. In this case all the members receive the same input.
- ***Multicast with scattering***, a point-to-multipoint communication. In this case the group is subdivided in "sub-groups" and, for each sub-group, a different input data is delivered to all the members that compose the sub-group.

Communication semantics have to be selected according to the behavior chosen for the group. For example the multicast semantic is adopted when a request execution is sent to a part of the group members and the input parameters are sent with the broadcast semantic, etc., whereas the unicast semantic is adopted for a request execution when an input parameter is scattered and each part has to be sent to a different group member.

For each one of the semantics reported above, a reliable or unreliable schema can be adopted, depending on the selected semantics of the group. Some group semantics for the creation phase can also be individuated. Examples are the policy for the selection of host nodes on which to allocate the group members, the management of each constructor parameter of the group members and also the semantic that determines the condition of success of a group creation. In this section only the method invocation semantics are analyzed, whereas those related to the group creation phase are subject to ongoing work.

To ensure flexibility and extensibility the configuration and customization of a behavior for a group is obtained through GroupBehavior. Such class specifies the behavior of a group in response to the method invocation request and is the composition of the four semantics defined above. Each semantic has a default implementation and can be modified at runtime.

A semantic is associated to an instance of one of the following interfaces:

- `RequestMappingSemantic`
- `InputDistributionSemantic`
- `SynchronizationSemantic`
- `OutputCollectionSemantic`

Each interface has some methods that have to be implemented to define a specific semantic. Such methods are invoked by a component of the framework, called `GroupBehaviorEnactor`.

`RequestMappingSemantic`

The body of the method `Vector getMembers(MethodCall mc, Vector memberList)` specifies the group members at which the request has to be sent. It receives an instance of the `MethodCall` class, which contains information on the current method invocation on the group (opportunely captured at runtime by the MOP: refer to Section 3.2.2), in particular on the method signature and the effective arguments. The other input parameter is the list of the current group members.

`InputDistributionSemantic`

The implementation of the method
`Vector manageInputs(MethodCall mc, Vector memberList, Communicator comm)`
specifies how the input parameters have to be distributed to the group members for a method invocation request. It receives the `MethodCall` instance which represents the current method invocation request, the list of the group members chosen for the request execution by the `RequestMappingSemantic`. The last input parameter represents a component responsible for the implementation of the logical communication semantic to use for data transmission inside a group (more details come in Section 8.2). Such class has the method
`void setLogicalCommunication(String commSchema, Parameters qos)`
which permits a user to configure a logical communication semantic for a method execution request. The method uses a string representing a communication schema supported by the middleware and some parameters of *QoS* which have to be satisfied. `Parameters` is a class that contains instances of `Parameter`, which is a couple of attribute-value. Currently, we consider only a parameter, which represents the reliability level defined by the attribute `reliability` and can assume the values `"reliable"` and `"unreliable"`. The communication schemas currently supported by our protoype implementation are: `"unicast"`, `"multicast"` and `"multicast-scattered"`.

From the programming point of view the possibility to specify the logical communication semantic inside a group is provided without any awareness on the leveraged transport layers supported by the physical networks. For example, although unicast group communication could be implemented by employing a unicast transport protocol such as TCP or UDP, multicast group communications could be implemented both by using a unicast transport protocol and a multicast one, depending on the availability of the underlying transport layers.

Finally the return parameter is a vector which contains the result of the distribution semantic applied to each effective argument, obtained by the `MethodCall` instance, corresponding to the input parameter identified by its index in the parameter list.

SynchronizationSemantic

Through the implementation of the method `void waitFor(MethodCall mc, Vector futures)` it is possible to specify the synchronization policy when a result of a group method invocation is used for another method call. Such method is invoked on a vector of future objects, each of which is associated to the asynchronous call on a group member scheduled for the execution.

This method can be easily implemented leveraging the static methods of *ProActive* related to the synchronization on a future object or a vector of future objects.

OutputCollectionSemantic

It determines how to reply to the caller the final return parameter of a group method invocation through the method `Object manageOutput(MethodCall mc, Vector futures)`. It receives an instance of `MethodCall` and a vector of *future* objects, containing the stubs of the results. The implementation of this method defines the operation to perform on the *futures* before returning a final result for the method invocation. This operation may be for instance, combination, selection [1], transformation into a typed group, etc.

Group behavior usage

The extension of the *ProActive* group requires only few modifications to the syntax of the current version. In particular the client application creates an instance of a group specifying the `GroupBehavior` to apply. As a consequence, the static methods of the `ProActiveGroup` class have been modified to include this parameter. Group creation is now performed through the method `newGroup` which specifies the group class, the constructor parameters, the nodes, and the group behavior. Here is an example of group creation specifying a group behavior:

```
  // Parameters and nodes
Object[][] params = {{...} , {...} , ... };
Node[] nodes = { ... , ... , ... };

  // Creation of the semantics
RequestMappingSemantic ms = new UserMappingImplementation();
InputDistributionSemantic ds = new UserDistributionImplementation();
SynchronizationSemantic ss = new UserSynchronizationImplementation();
OutputCollectionSemantic oc = new UserOutputImplementation();

  // Creation of the GroupBehavior object
GroupBehavior behavior = new GroupBehavior(ms,ds,ss,oc);

  // Creation of a typed group with the defined group behavior
A a = (A) ProActiveGroup.newGroup("A", params, nodes, behavior);
```

Overall, this work has led to the following: thanks to the reification, the semantic related to the management of method invocations on groups can be intercepted and customized at runtime in order to logically show a specific behavior.

8.2 Using IP multicast

From the point of view of the communication inside a group, a major improvement can be made. The idea is to perform the data transmission leveraging the potentialities of the network connections effectively available at the moment. For example some network information can be used in order to adopt, when it is possible, as an alternative to the commonly used unicast transport communication based on TCP/IP, a transport layer based on multicast protocols.

[1] Combination (for instance addition or multiplication), and selection (for instance minimum or maximum) are able to express reduce operations.

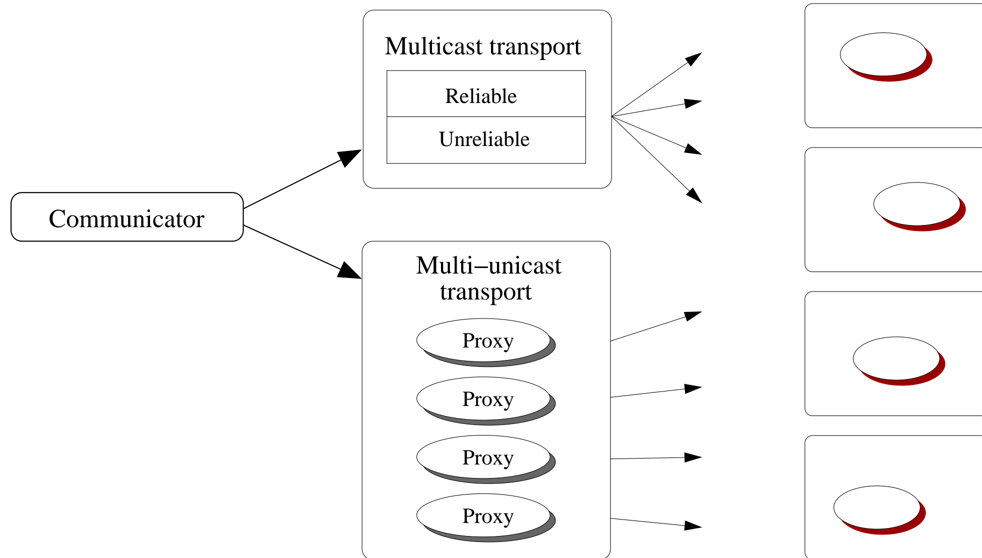


Figure 8.1: The communicator component

In this case, differently from real-time multimedia distributed systems, which tolerate unreliable data streaming to reduce latency, other ditributed systems often require reliable multicast protocols to deliver replicated application data without losses and errors. These considerations have motivated an intense research activity which has led to many protocol definitions for implementing reliability in multicast communication. The paper [CHI 98] proposes an interesting solution integrated in a Java framework, JRMS (Java Reliable Multicast Service) [ROS 98], that provides several reliable multicast protocols.

ProActive is particularly suitable to implement such a mechanism, thanks to its high modularity and customization mechanisms related to the mapping of logical application data communication to the real services available at transport level for data transmission on physical networks.

Our solution is based on the definition of a new *ProActive* component, the `Communicator` (presented on Figure 8.1), which has the main task to manage the data transmission inside a group for each method execution request. Such component is the only component to be aware of the communication services delivered by the physical networks and so to be able to map the logical communication semantic onto an available transport layer, which is the most suitable one.

For multi-unicast communications, the `Communicator` can access to the standard `Proxy` services, one for each member scheduled for the request execution, which is able to handle the transmission on the network of a request adopting one among the available unicast transport layers. For multicast communication, the `Communicator` can access to the `MulticastProxy`, a completely new component, able to handle the transmission of a request adopting a multicast mechanism.

The default communication protocol adopted in *ProActive* is RMI. It limits the possibility to improve the performances of applications. In fact RMI is implemented on TCP which requires multiple connections to simulate a group method. For this reason, we propose an implementation of *ProActive* groups atop a transport layer based on IP multicast.

Integrating reliable multicast inside a middleware for Grid computing is still an open issue. Some solutions aim at easily porting existing applications to multi-destination environments by enriching TCP with multicast capabilities [JEA 03]. To efficiently exploit a multicast protocol, the Grid computing middleware should be able to manage the sub-parts of the Grid infrastruc-

ture in which the multicast communication is supported at data-link or network layers. This is the case of a cluster in which the resources are connected through a common LAN which supports broadcast communication at data-link layer, or a set of workstations directly connected to an IP multicast-enabled router.

Unreliable multicast typically provides scalability up to tens of thousands of nodes, but its semantics are generally too weak for application developers to depend upon. Messages are subject to long and unpredictable transmission delays, message loss, and out of order delivery. Processes may crash and network links may fail; such failures are hard to detect when the communication delays are unpredictable and messages can be lost. To avoid those, we use a reliable multicast service to integrate into the transport layer of *ProActive*.

The master-slave pattern [BUS 96] for distributed programming was implemented to test our proposal. Two implementations of this programming model are shown and compared, the first one adopts the standard group mechanism and the second one adopts the extended group mechanism. A slave object is implemented by means of a group member, each of which is opportunely distributed onto remote machines.

The canonical matrix multiplication is used as case study. Class `Matrix` is used to represent the abstract data-type matrix, and delivers the methods necessary to perform the row-for-column multiplication. In particular `Matrix` delivers a constructor `Matrix (float[][] m)` where the parameter `m` is a two-dimensional array of float, and the method `Matrix multiply (float[][] a)`, that performs the multiplication algorithm where the current instance represents the right matrix and the matrix passed as parameter the left matrix. Such matrix has to be split in equivalent sub-parts, using a row-based decomposition, each of that has to be sent to a different group member that represents a slave object. On the other hand, the constructor parameter will be the overall right matrix, which so will be the same for each of them.

A performance evaluation of the proposed approach was conducted by comparing the performances obtained with two different implementations of the case-study. An implementation was based on the standard groups (with unicast communications), the other one was based on extended groups (with multicast communications). For the first case, the default *ProActive* implementation based on Java RMI was adopted, while for the second one, a prototypical version of the *ProActive* group mechanism was implemented using a reliable multicast protocol, TRAM (included in JRMS 1.1).

The testbed was a cluster of eight nodes, each one equipped with Intel Pentium II @ 350 MHz, 128 MB of RAM and 100 Mb/s Ethernet network card, and a machine equipped with Intel Pentium IV @ 2.4 GHz, 256 MB of RAM and 100 Mb/s Ethernet network card.

Figure 8.2 shows the execution times of the application matrix multiply, considering a fixed matrix dimension to 1000x1000 float numbers, and a varying number of slaves. As it is possible to note, the implementation based on reliable multicast exhibits better performances, mainly due to the reduced traffic on the network. The adopted implementation in fact employs multicast communication for sending and gathering data from group members, each of which receives the right matrix with the broadcast semantic, so strongly reducing the network utilization compared to repeated unicast communication adopted by the original *ProActive* groups.

8.3 Components

Computing grids and peer-to-peer networks are inherently heterogeneous and distributed, and for this reason they drive new technological challenges: complexity in the design of applications, complexity of deployment, reusability and performance issues. *ProActive* provides an answer to these problems through the implementation of an extensible, dynamical and hierarchical component model named *Fractal* [FRA].

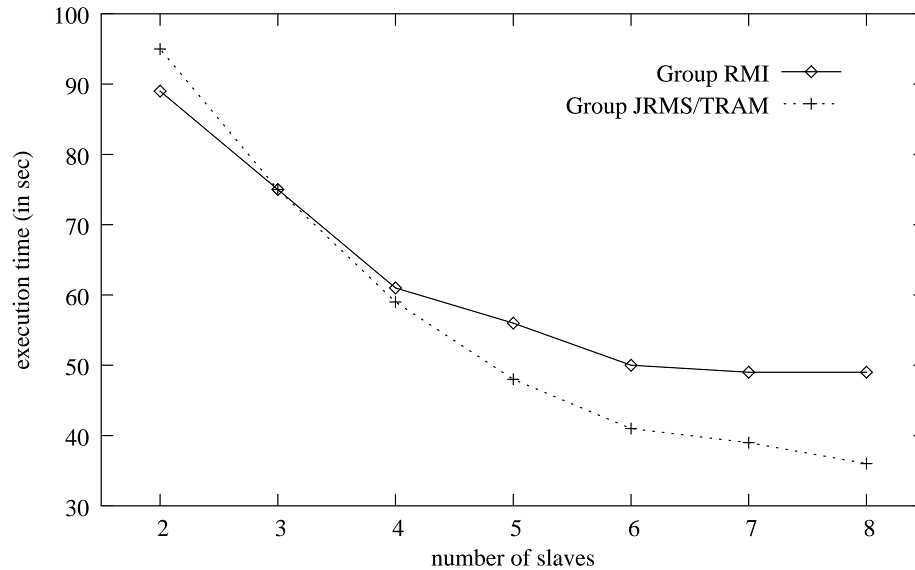


Figure 8.2: Group RMI vs Group IP Multicast

Fractal defines a general conceptual model, along with an application programming interface in Java. According to the official documentation, the Fractal component model is *"a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces"*. The Fractal model is based on the concepts of encapsulation, composition, sharing, life-cycle, activities, control, and dynamicity [BRU 02]. A Fractal component is formed out of three parts:

- ***a content*** that can be recursive. As a component can contain other components, the model is hierarchical.
- ***a set of controllers*** that provide introspection capabilities for monitoring and exercising control over the execution of components.
- ***a set of interfaces*** with which the component interacts with other components. These interfaces can be either client or server, and are interconnected using bindings.

As *ProActive* offers many features, such as distribution, asynchronism, and mobility that would be interesting for Fractal component; an implementation of the Fractal API based on *ProActive* was written by Matthieu Morel [BAU 03].

A *ProActive* component is parallelizable and distributable as we aim at building grid-enabled applications by hierarchical composition [BAD 04a, BAD 05b]. Build a component acts as a glue to couple codes that may be parallel and distributed codes requiring high performance computing resources. Hence, components should be able to encompass more than one activity and be deployed on parallel and distributed infrastructures. Such requirements for a component are summarized by the concept we have named *Grid Component*. Figure 8.3 sums up the different kinds of grid components we provide.

The implementation for *ProActive* currently defines two extensions to the base component model of Fractal:

- ***Distributed deployment***: components can be deployed onto distributed virtual machines, using the deployment facilities of *ProActive*.
- ***Parallel components***: this type of components is a specialization of the composite components, as they encapsulate other components. They encapsulate other components of *the*

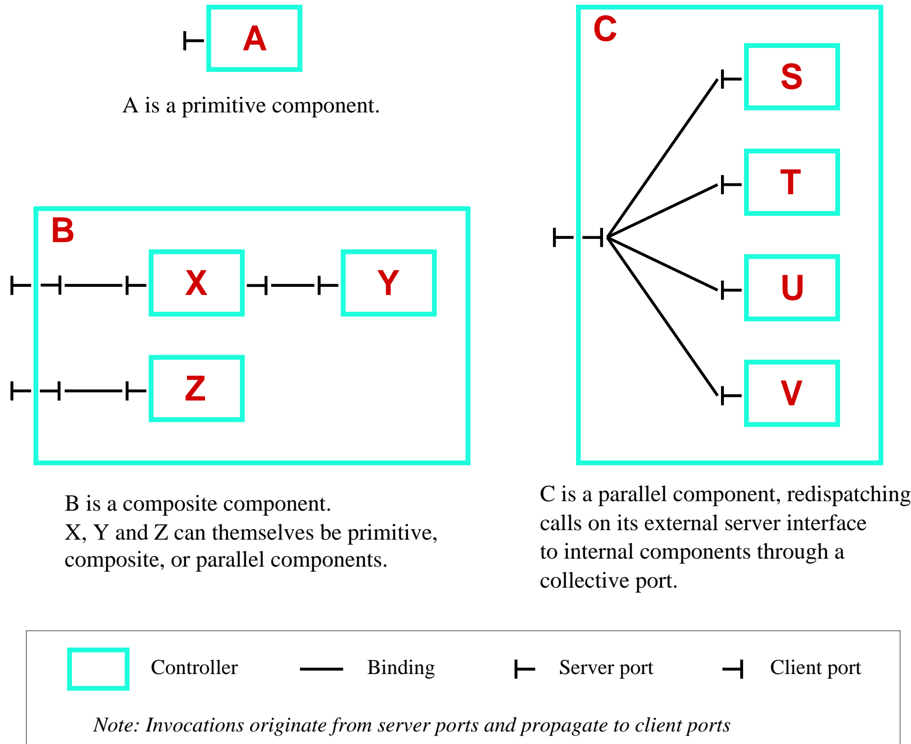


Figure 8.3: The three types of grid components

same type, and all incoming calls are forwarded to the corresponding internal interfaces of the enclosed components. This allows parallel processing while just manipulating one entity, the enclosing parallel component. We name *collective ports* the ports allowing such operation.

The typed group API is a key feature for parallel components. The implementation of collective port is based on the *ProActive* groups. According to the Fractal specification, a collective interface only has sense for a client interface, that would like to be bound to several server interfaces. Besides, a server interface can always be accessed by several client interfaces, the calls being processed sequentially. Specifying a server interface as collective would not change its behavior.

The *ProActive* group API allowing group communication in a transparent manner, the implementation of the collective interfaces slightly differs from the Fractal specification: instead of creating one new interface with an extended name for each member of the collection, we just use one interface (that is actually a group). Collective bindings are then performed transparently as if they were multiple successive bindings on the same interface. Using a collective interface will then imply using the typed group formalism, including the possibility to choose between scattering and broadcasting of the calls. A feature is that unbinding operations on a collective interface will result in the removal of all the bindings of the collection.

In order to couple parallel components, we plan to provide facilities for composing collective ports. Every parallel component has a set of meta-objects associated to it and could serve as a sophisticated re-dispatcher: for the set formed from each client port of interest of each inner component of the parallel component (thus defining the notion of a collective client port of a parallel component), to the server collective port of interest it is bound to. The objective – but maybe not the solution – is similar to what is achieved by introducing collective communications as tees in the ICENI Grid oriented component model [MAY 02]: switch, combiner, splitter, gather, broadcast; the same regarding the collective port extending CCA ports, experimented in

[KEA 01], which is in fact implemented as a combination of translation components (i.e. customizable components, efficiently called by the framework, to tackle translation/redistribution of data, collective invocation and returns, e.g. a MxN component).

As presented in [BAD 05c], our challenge is to provide a solution adapted to the component-oriented model we propose, that is, without the explicit introduction of additional components (either generic or programmer-modifiable), but only through the definition of Fractal ports and the usage of the *ProActive* group communication mechanism. An illustration of this objective is given in Figure 8.4: the idea is to automate the broadcasting or scattering of invocation parameters, and symmetrically the gathering of reply parameters. In the case where the communications occur from M components to N components this also requires a redistribution policy of the parameters and replies from M invokers to N receivers. Schematically, this could end up by also synchronizing M calls and gathering their parameters, and then scatter them onto N calls. Regarding the replies, the inverse operation is needed: gathering N replies and automatically scattering them onto M invokers. In this sense, our design of collective ports composition is related to the one in [DEN 04], which is based on collective RMI calls extended with the usage of a MxN redistribution scheme introduced in recent CCA compliant implementations of parallel data redistribution [DAM 03, BER 04].

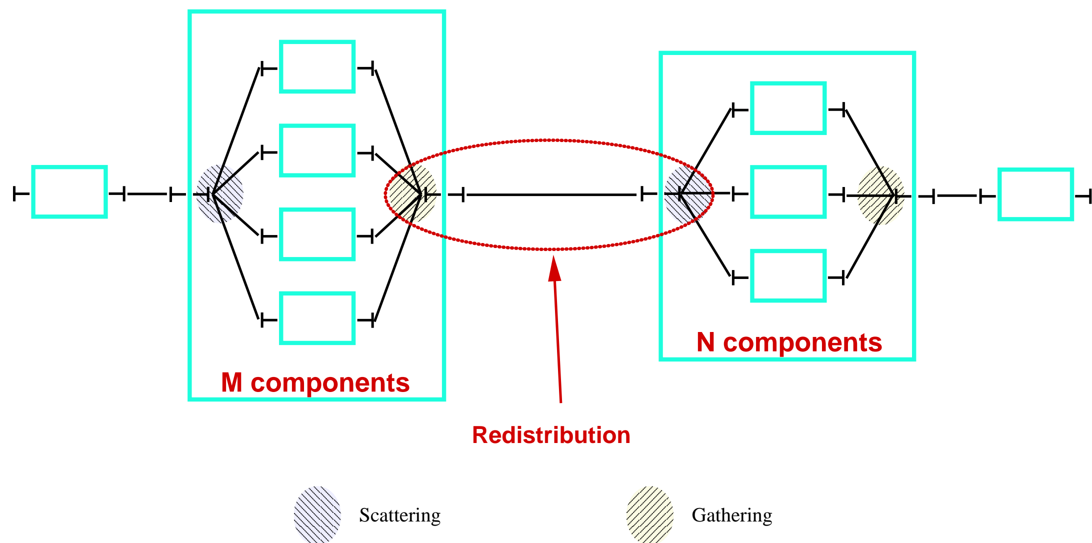


Figure 8.4: Redistribution from M components to N components

8.4 Peer-to-peer computing

Computational *peer-to-peer* (P2P) is emerging as a key execution environment. The potential of 100,000 of nodes interconnected to execute a single application is rather appealing, especially for Grid computing. Mimicking data peer-to-peer, one could start a computation that no failure would be able to stop. *ProActive* provides an API for peer-to-peer computing that aims to use spare CPU cycles from organization or institution's workstations possibly combined with grids and clusters. The goals are to deploy applications on a decentralized set of nodes and to use most of available resources on a network. This API was mainly written by Alexandre di Costanzo.

The peer-to-peer infrastructure works as an overlay network. It is composed of peer-to-peer services (the peers) which in turn become computational nodes. An active object, named `P2PService` and mapped on a *ProActive* JVM, implements the service. Figure 8.5 presents a network of hosts where some JVMs are running and several of them are running a `P2PService`.

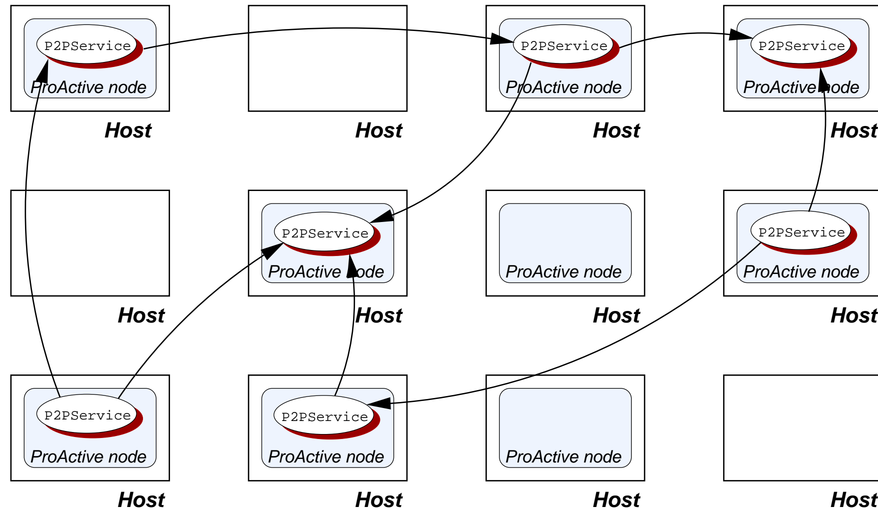


Figure 8.5: A peer-to-peer infrastructure

The peer-to-peer infrastructure is self-organized and configurable. When the infrastructure is running, it is kept up by itself, with no action from the user. There are three main configurable parameters to achieve this:

- The ***Time To Update*** (TTU) represents the duration between connectivity examinations: when its TTU expires, a peer checks if its known peers are still available.

- The ***Number Of Acquaintances*** (NOA) is the minimal number of peers that one peer have to know. In order to keep the infrastructure running, each peer tries to maintain references on this number of other peers.

- The ***Time To Live*** (TTL) is not really a duration in time. TTL refers to a number of messages retransmission. It is the depth of message propagation in "hops".

The bootstrapping, or first contact problem, is to define how a new peer can join the peer-to-peer infrastructure. A solution for that is to use a specific protocol. *ProActive* provides an interface for JINI, a network-centric services protocol. JINI can be used for discovering services in a dynamic computing environment, such as a fresh peer which would like to join a peer-to-peer network. This protocol is perfectly adapted to solve the bootstrapping problem. However JINI is limited to work only in the same sub-network that means JINI doesn't pass through firewall or NAT and can't be considered to be used for Internet.

Therefore, we choose a different solution for the bootstrapping problem. It is inspired from data peer-to-peer networks: when creating a new peer, we specify one or several addresses of supposed peers which are running in the peer-to-peer infrastructure. After that this list of addresses is then dynamically updated when the peer discovers new peers or when it detects known peers' failure. The bootstrapping follows those steps:

1. A fresh peer has a list of "server" addresses. These are peers that have a high potential to be available and present in the peer-to-peer network. In a certain way, they act as the peer-to-peer network core.

2. With this list the fresh peer tries to contact each server. When a server is reachable the fresh peer adds it in its list of known peers, and increases its *Number Of Acquaintances*.

3. Then the fresh peer knows some other peers; it is fully integrated in the peer-to-peer network. Its list is dynamically updated.

In the case of the fresh peer can not contact any peers from the list, the fresh peer will try every *Time To Update* to recontact all of them until one or several of them become finally available. At any moment, if the *Number Of Acquaintances* of a peer reaches 0 (the peer knows nobody because all of its acquaintances became unavailable), the peer will try to reconnect to the peer-to-peer network as if it was looking for a first contact.

As a peer-to-peer infrastructure is a dynamic environment, the list of acquaintances is dynamic. Many acquaintances could become unavailable, and then should be removed from the list. All peers have to keep their lists up-to-date. In order to verify the acquaintances availability, the peer sends an *Heart Beat* to all of its acquaintances, every *Time To Update*. Unreachable peers are removed from the list of acquaintances.

Those communications between peers are performed with one-way group communication, but for sending response of a request message, it's a point-to-point communication. The group communication rendez-vous guarantees the message is successfully received by the receiver, if this one is up and reachable. If an acquaintance is down, the failure of heart beat message delivery produces a Java exception caught by the group exception handling mechanism and redirected as an exception of the peer-to-peer API.

The `P2PAcquaintanceManager` manages the list of acquaintances; this list is represented by a typed group of `P2PService`. The `P2PAcquaintanceManager` is an active object linked to the `P2PService` object on each node of the peer-to-peer network. At its initialization it constructs an empty `P2PService` group. It exposes few access group methods, such as `remove`, `add`, and `getSize` methods. All other peers (such as the `P2PService` active objects) have to use `P2PAcquaintanceManager` methods to access to the group. The manager also exposes the `P2PService` interface. Those methods are bound on the typed group; that is the way the messages are propagated from a peer to its acquaintances (i.e. from the manager to the remote peer-to-peer services). Figure 8.6 presents the role of the `P2PAcquaintanceManager` in the broadcast of heart beat messages.

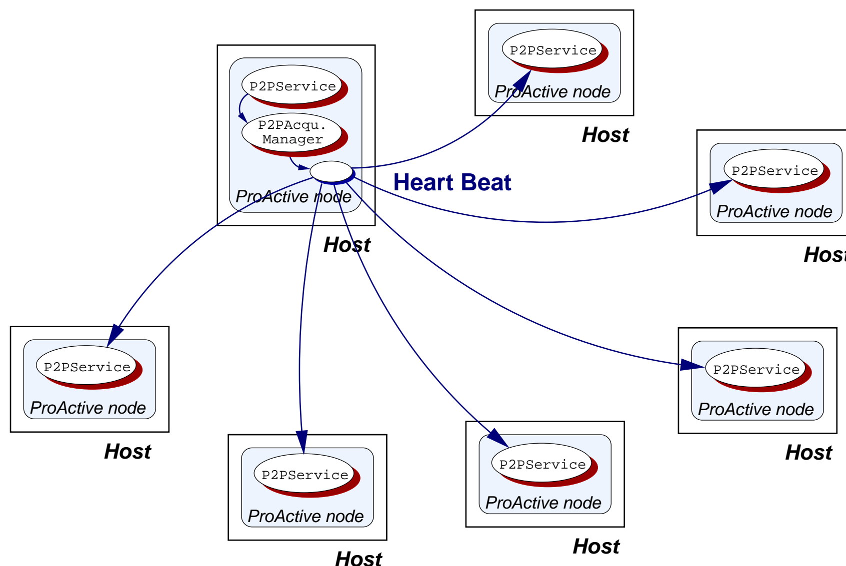


Figure 8.6: An heart beat sent as a group communication

Deployment on such peer-to-peer network is transparently achieved thanks to the deployment descriptors. The activity deploying the application, send a request for available hosts to a known peer of the network, specified in the XML descriptor. The peer looks for available hosts in its list of acquaintances and if there is not enough hosts, it forwards the request to it acquaintances to ask for more available hosts. The request is recursively propagated until the required number of hosts was reached (or a time out stops the recursive search).

Chapter 9

Conclusion

9.1 Achievements

Java has many advantages for Grid computing. Foremost, by being based on a virtual machine concept, it is inherently more portable than traditional, statically compiled languages, making it much easier to execute Java applications in a heterogeneous Grid environment. Also, Java is based on a high-level, object-oriented, type-safe programming model and it has built-in support for multithreading and distributed computing.

The basic unit of activity and distribution used by *ProActive* to build concurrent applications is the Active Object. An active object is remotely created on a host involved in the computation. Methods calls sent to active objects are asynchronous with transparent future objects and the synchronization is handled by a mechanism known as wait-by-necessity.

In addition to simple active objects, *ProActive* offers a group communication mechanism that allows for method invocations on sets of objects, grouped together and referenced by a single collective name. A *ProActive* group is also called typed group since it is composed of objects belonging to classes inheriting from the same superclass or implementing the same interface. Typed group is the "clonation" of an active object on a set of nodes and a group communication is the "replication" of a remote method invocation on them. Each member can be an instance of a different class but all the members must have the same ancestor.

While many libraries and programming frameworks delivering group abstraction impose specific constraints on programmers, thanks to the use of a Meta-Object Protocol, *ProActive* delivers a more transparent and flexible mechanism. *ProActive* MOP, through the reification of method invocation and constructor call, makes it possible to initiate group communication invoking a method of the group object. As a consequence using a typed group takes exactly the same form as using only one active object. When a method call is invoked towards a group, the semantics of communication are implemented on an asynchronous underlying communication system which internally handles execution requests as sequences of events related to request transmissions, request dispatching, failure notifications, result collecting, etc. Such communication system asynchronously and efficiently propagates the call to all members of the group using multithreading. A method call on a group is asynchronous and provides a transparent future object to collect the results.

Currently, *ProActive* groups provide the programmer with some mechanisms for the management of input parameters, such as broadcasting and scattering. By adopting the broadcasting, the same parameter is sent to all the members. On the other hand, by adopting the scattering, a part of the overall parameter is transferred to the members. In this case, the parameter has to be explicitly passed as a group, which is built splitting the original parameter in several parts. The default behavior is the broadcasting, while in order to scatter a parameter the programmer has to invoke the static method `setScatterGroup` of the `ProActiveGroup` class to the input parameter group. So, the scatter policy is tied only to a specific input parameter instance.

The result of a typed group communication is also a group, requiring so an explicit management of its group members when an aggregation policy has to be adopted. The result will be dynamically updated with the incoming partial results. Thanks to the wait-by-necessity synchronization mechanism, a result can be immediately used to execute a method call, even if all the results are not available.

We have introduced a parallel programming model, which we name Object-Oriented SPMD as an alternative to the traditional Message-Passing SPMD style. The resulting OO SPMD API already forms part of the *ProActive* open-source library, freely distributed through the Object Web consortium for open-source middleware. Our ambition is to have this approach used on real size applications. We already successfully applied the typed group communication mechanism to solve simulation in electromagnetism. Our current work is to apply the whole OO SPMD approach to it. Next, we plan to target other application domains, such as biogenetics (applying BLAST in parallel), for which we already have developed applications, but not yet using OO SPMD.

9.2 Perspectives

The *ProActive* groups define a complete framework for typed group support. An implementation has been realized, evaluated, and used to build applications. The OO SPMD programming model proposes a more flexible approach of SPMD programming. It allows techniques to introduce flexibility in barrier synchronization and to remove any explicit loop. It makes possible to privilege reactivity and reuse.

However, several issues remain open. The main focuses for future work are listed here:

- A ***smart thread pool dimensioning***. Define a generic solution that allows to optimally allocate resources for a group communication is not easy. Many parameters step in: the amount of group members of course, but also the communication frequency, the size of exchanged data, the system load, the network speed, etc. Because we have experimented in many applications that the better dimensioning mechanism is often obtained by practical observations and then user adaptations, our final choice is to let the programmer define his or her own method to correctly size the pool. But it should be interesting to look deeper in order see if some patterns may emerge to build dimensioning mechanisms that can answer more generically to this issue.
- A ***study on ordering delivery semantic***. Basic typed group communications provide a FIFO ordering semantic: given a source, messages are received in the order they were sent. This semantic is generally sufficient for the conception of distributed application. It provides good performances; and stronger semantic may be added by the programmer if needed as an extension of the group mechanism. With active groups, the ordering is total: messages are delivered in the same order to all members of the group. This is guaranteed because an active group provides a unique entry point that relays the calls. A study of the impact of one or the other semantic on application writing and execution should be interesting. We may also consider the introduction of a causal ordering semantic.
- A ***large multicast benchmark***. The current implementation that links the *ProActive* group and an IP multicast library is a prototype implementation. We should produce a final and releasable version that could come with the standard *ProActive* distribution. We should also go further in the analysis of performances of this IP multicast version of our typed group communication. We should evaluate performances on cluster and grid platforms, with basic benchmarks and with a numeric application (probably Jem3D). It should be also very interesting to observe the behavior of the system with a mixed communication scheme; for instance in a grid environment where inter-cluster communications are assumed by standard *ProActive* communication (to pass through firewalls) while intra-cluster

communications are assumed by IP multicasting (to benefit from the high-speed network abilities).

- A ***MxN redistribution***. Regarding the components, we would like to automate the sending of method call parameters, and in a symmetric manner, the gathering of results in the case of M to N data redistribution. In our current implementation of the Fractal model, this issue is not yet addressed. Typed groups assume that the unit of data transmission is the object. A solution should be to ask the programmer to implement a method that redistributes his or her data from any M objects to any N objects, and to use automatically this method in our component framework. But we are still looking further for a more autonomous and transparent solution. Typed group communications with group of futures open the way to a wide range of new perspectives in this area.

- A more ***accurate OO SPMD model evaluation***. The OO SPMD programming model has been evaluated with a simple algorithm running on a simple cluster. We would like to try it with a real sized numerical application: possibly Jem3D since it fits well with SPMD models (its algorithm is based on iterations and each sub-domain performs the same task), or a totally new application built from scratch. Then performance measurements of this application running on a grid may teach a lot about the validity of our approach and its behavior in a very large scale system.

Bibliography

[ANA 04] SANTOSH ANAND. GeB, Grid-enabled BLAST with Distributed Objects. Master's thesis, DEA RSD, University of Nice Sophia-Antipolis, September 2004.

[ANT 00] GABRIEL ANTONIU, LUC BOUGÉ, PHILIP J. HATCHER, MARK MAC BETH, KEITH MC GUIGAN, and RAYMOND NAMYST. "Compiling Multithreaded Java Bytecode for Distributed Execution". In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, pages 1039–1052, München, Germany, August 2000. Springer Verlag. Lecture Notes In Computer Science vol. 1900.

[ARM 92] SUSAN M. ARMSTRONG, ALAN O. FREIER, and KEITH A. MARZULLO. "Multicast Transport Protocol (RFC 1301)". Technical report, IETF Network Working Group, February 1992.

[ATT 00] ISABELLE ATTALI, DENIS CAROMEL, and ROMAIN GUIDER. "A Step Towards Automatic Distribution of Java Programs". In *FMOODS 2000*, pages 141–161, Stanford University, California, USA, September 2000. Kluwer Academic.

[ATT 03] ISABELLE ATTALI, DENIS CAROMEL, and ARNAUD CONTES. "Hierarchical and Declarative Security for Grid Applications". In *Proceedings of the International Conference On High Performance Computing*, Hyderabad, India, December 2003. Springer Verlag.

[BAD 01] LAURENT BADUEL. Communications de groupes efficaces pour objets actifs répartis. Master's thesis, DEA Réseaux et Systèmes Distribués, University of Nice Sophia-Antipolis, June 2001.

[BAD 02a] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. Communication de groupes typés dans ProActive. Winter School, École thématique sur la globalisation des ressources informatiques et des données (GRID), December 2002.

[BAD 02b] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Efficient, Flexible, and Typed Group Communications in Java". In *Joint ACM Java Grande - ISCOPE Conference*, pages 28–36, Seattle, Washington, USA, November 2002. ACM Press. ISBN 1-58113-559-8.

[BAD 03] LAURENT BADUEL, and DENIS CAROMEL. "Communications de groupe typé pour objets répartis". In *RenPar15 (French-speaking meetings of Parallelism, 15th edition)*, pages 119–126, La Colle-sur-Loup, France, October 2003. INRIA Edition. ISBN 2-7261-1264-1.

[BAD 04a] LAURENT BADUEL, FRANÇOISE BAUDE, DENIS CAROMEL, ARNAUD CONTES, FABRICE HUET, MATTHIEU MOREL, and ROMAIN QUILICI. Components for numerical GRIDs. Communication in European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS, July 2004.

[BAD 04b] LAURENT BADUEL, FRANÇOISE BAUDE, DENIS CAROMEL, CHRISTIAN DELBÉ, SAÏD EL KASMI, NICOLAS GAMA, and STÉPHANE LANTERI. "A Parallel Object-Oriented Application for 3D Electromagnetism". In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[BAD 05a] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Object-Oriented SPMD". In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[BAD 05b] LAURENT BADUEL, FRANÇOISE BAUDE, DENIS CAROMEL, ARNAUD CONTES, FABRICE HUET, MATTHIEU MOREL, and ROMAIN QUILICI. "*Grid Computing: Software Environments and Tools*", chapter Programming, Deploying, Composing, for the Grid. Springer Verlag, 2005.

[BAD 05c] LAURENT BADUEL, FRANÇOISE BAUDE, DENIS CAROMEL, LUDOVIC HENRIO, FABRICE HUET, STÉPHANE LANTERI, and MATTHIEU MOREL. "Grid Components Techniques: Composing, Gathering, and Scattering. Can it be Used for Coupled Problems?". In *Proceedings of Coupled Problems, Thematic Conference of ECCOMAS*, Santorini Island, Greece, May 2005.

[BAD 05d] LAURENT BADUEL, FRANÇOISE BAUDE, NADIA RANALDO, and EUGENIO ZIMEO. "Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups". In *JPDC, 7th International Worshop on Java for Parallel and Distributed Computing at IPDPS*, Denver, Colorado, USA, April 2005.

[BAK 98] MARK BAKER, BRYAN CARPENTER, SUNG HOON KO, and XINYING LI. "mpiJava: A Java interface to MPI". In *First Workshop on Java for High Performance Network Computing, Europar '98*, University of Southampton, United Kingdom, September 1998.

[BAK 99] MARK BAKER, BRYAN CARPENTER, GEOFFREY FOX, SUNG HOON KO, and SANG LIM. "mpiJava: An Object-Oriented Java interface to MPI". In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, San Juan, Puerto Rico, April 1999.

[BAN 98] BELA BAN. "Design and Implementation of a Reliable Group Communication Toolkit for Java". Technical report, Department of Computer Science, Cornell University, September 1998.

[BAU 00] FRANÇOISE BAUDE, DENIS CAROMEL, FABRICE HUET, and JULIEN VAYSSIÈRE. "Communicating Mobile Active Objects in Java". In *Proceedings of the 8th International Conference on High Performance Computing and Networking Europe*, volume 1823 de *LNCS*, pages 633–643, Amsterdam, The Netherlands, May 2000. Springer Verlag.

[BAU 02] FRANÇOISE BAUDE, DENIS CAROMEL, FABRICE HUET, LIONEL MESTRE, and JULIEN VAYSSIÈRE. "Interactive and Descriptor-Based Deployment of Object-Oriented Grid Applications". In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BAU 03] FRANÇOISE BAUDE, DENIS CAROMEL, and MATTHIEU MOREL. "From Distributed Objects to Hierarchical Grid Components". In *Proceedings of the International Symposium on Distributed Objects and Applications*, Catania, Italy, November 2003. Springer Verlag, Lecture Notes in Computer Science, LNCS.

[BAU 04] FRANÇOISE BAUDE, DENIS CAROMEL, CHRISTIAN DELBÉ, and LUDOVIC HENRIO. "A Fault Tolerance Protocol for ASP calculus: Design and Proof". Technical report RR-5246, INRIA, Sophia-Antipolis, France, June 2004.

[BAU 05] FRANÇOISE BAUDE, DENIS CAROMEL, CHRISTIAN DELBÉ, and LUDOVIC HENRIO. "A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability". In *Proceedings of EuroPar*, Lisboa, Portugal, August 2005.

[BER 04] FELIPE BERTRAND, and RANDALL BRAMLEY. "DCA: A Distributed CCA framework based on MPI". In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments at IPDPS*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[BIR 84] ANDREW D. BIRELL, and BRUCE J. NELSON. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BIR 85] KENNETH. P. BIRMAN, AMR EL ABBADI, WALLY DIETRICH, THOMAS A. JOSEPH, and THOMAS RAEUCHLE. "An Overview of the Isis Project". In *IEEE Distributed Processing Technical Committee Newsletter*, January 1985.

[BIR 91] KENNETH P. BIRMAN, ANDRÉ SCHIPER, and PAT STEPHENSON. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[BIR 93a] KENNETH P. BIRMAN. "A Response to Cheriton and Skeen's Criticism of Causally and Totally Ordered Communications". Technical report, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1993.

[BIR 93b] KENNETH P. BIRMAN. "The Process Group Approach to Reliable Distributed Computing". In *Communications of the ACM, vol 36*, pages 37–53, 1993.

[BIR 94] KENNETH P. BIRMAN. "A Response to Cheriton and Skeen's Criticism of Causally and Totally Order Communications". In *Operating System Review, 15th ACM Symposium on Operating System Principles*, volume 28. ACM Press, January 1994.

[BRA 93] ROBERT BRAUDES, and STEVE ZABELE. "Requirements for Multicast Protocols (RFC 1458)". Technical report, IETF Network Working Group, May 1993.

[BRU 02] ERIC BRUNETON, THIERRY COUPAYE, and JEAN-BERNARD STEFANI. "Recursive and Dynamic Software Composition with Sharing". In *Proceedings of the Seventh International Workshop on Component-Oriented Programming at ECOOP*, Malaga, Spain, June 2002.

[BUR 94] GREG BURNS, RAJA DAOUD, and JAMES VAIGL. "LAM: An Open Cluster Environment for MPI". In *Proceedings of Supercomputing Symposium*, pages 379–386, Washington D.C., USA, November 1994.

[BUR 99] M. BURKE, J. CHOI, S. FINK, D. GROVE, M. HIND, V. SARKAR, M. SERRANO, V. SREEDHAR, H. SRINIVASAN, and J. WHALEY. "The Jalapeño Dynamic Optimizing Compiler for Java". In *Proceedings of JavaGrande / ISCOPE Conference*, pages 129–141, San Francisco, California, USA, June 1999. ACM Press.

[BUS 96] FRANK BUSCHMANN, REGINE MEUNIER, HANS ROHNERT, PETER SOMMERLAD, MICHAEL STAL, PETER SOMMERLAD, and MICHAEL STAL. "*Pattern-Oriented Software Architecture: A System of Patterns*", volume 1. John Wiley and sons, August 1996. ISBN 0471958697.

[CAR 93] DENIS CAROMEL. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[CAR 96] DENIS CAROMEL, FABRICE BELLONCLE, and YVES ROUDIER. "The C++// Language". In *Parallel Programming using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.

[CAR 98a] DENIS CAROMEL, WILFRIED KLAUSER, and JULIEN VAYSSIÈRE. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998.

[CAR 98b] BRYAN CARPENTER, and AL. "MPI for Java - Position Document and Draft API Specification". Technical report JGF-TR-03, Java Grande Forum, November 1998.

[CAR 99] BRYAN CARPENTER, GEOFFREY FOX, SONG HOON KO, and SANG LIM. mpiJava 1.2: API Specification, October 1999. http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html.

[CAR 00] BRYAN CARPENTER, VLADIMIR GETOV, GLENN JUDD, ANTHONY SKJELLUM, and GEOFFREY FOX. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, September 2000.

[CAR 03] DENIS CAROMEL, and ALEXANDRE GENOUD. "Non-Functional Exceptions for Distributed and Mobile Objects". In *Proceedings of the Workshop on Exception Handling in Object Oriented Systems at ECOOP*, Darmstadt, Germany, July 2003.

[CAR 04] DENIS CAROMEL, LUDOVIC HENRIO, and BERNARD SERPETTE. "Asynchronous and Deterministic Objects". In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages*, pages 123–134, Venice, Italy, January 2004. ACM Press. ISBN 1-58113-729-X.

[CAR 05a] DENIS CAROMEL, and GUILLAUME CHAZARAIN. "Robust Exception Handling in an Asynchronous Environment". In *Proceedings of the Workshop on Exception Handling in Object Oriented Systems at ECOOP*, Glasgow, United Kingdom, July 2005.

[CAR 05b] DENIS CAROMEL, and LUDOVIC HENRIO. "*A Theory of Distributed Objects*". Springer, 2005. ISBN 3-540-20866-6.

[CHE 85] DAVID R. CHERITON, and WILLY ZWAENEPOEL. Distributed Process Groups in the V Distributed System. *ACM Trans on Computer Systems*, 3(2):77–107, 1985.

[CHE 94] DAVID R. CHERITON, and DALE SKEEN. "Understanding the Limitations of Totally Order Communications". In *Operating System Review, 14th ACM Symposium on Operating System Principles*, volume 27, Asheville, North Carolina, USA, 1994. ACM Press.

[CHI 98] DAH-MING CHIU, STEPHEN HURST, MIRIAM KADANSKY, and JOSEPH WESLEY. "TRAM : A tree-based reliable multicast protocol". Technical report, Sun Microsystems, Palo Alto, California, USA, July 1998. TR-98-66.

[COO 94] ROBERT COOPER. "Experience with Causally and Totally Ordered Communication Support: a Cautionary Tale". In *Operating System Review, 15th ACM Symposium on Operating System Principles*, volume 28. ACM Press, January 1994.

[CZA 00] KRZYSZTOF CZARNECKI, ULRICH EISENECKER, and AL. "*Generic Programming*", volume 1766 of LNCS, chapter Generative Programming and Active Libraries (Extended Abstract), pages 25–39. Springer Verlag, 2000.

[DAM 03] KOSTADIN DAMEVSKI, and STEVEN PARKER. "Parallel Remote Method Invocation and M-by-N Data Redistribution". In *4th Los Alamos Computer Science Institute Symposium*, October 2003.

[DEE 89] STEPHEN E. DEERING. "Host Extentions for IP Multicasting (RFC 1112)". Technical report, IETF Network Working Group, Stanford University, California, USA, August 1989.

[DEN 03] ALEXANDRE DENIS, CHRISTIAN PÉREZ, THIERRY PRIOL, and ANDRÉ RIBES. "Padico: A Component-Based Software Infrastructure for Grid Computing". In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, April 2003. IEEE Computer Society.

[DEN 04] ALEXANDRE DENIS, CHRISTIAN PÉREZ, THIERRY PRIOL, and ANDRÉ RIBES. "Bringing High Performance to the CORBA Component Model". In *SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, California, USA, February 2004.

[FEL 96] PASCAL FELBER, BENOÎT GARBINATO, and RACHID GUERRAOUI. "The Design of a CORBA Group Communication Service". In *Proceedings of the 15th Symposium on Reliable Distributed Systems (SRDS-15)*, Niagara-on-the-Lake, Canada, 1996.

[FEL 97] PASCAL FELBER, RACHID GUERRAOUI, and ANDRÉ SCHIPER. "A CORBA Object Group Service". Technical report, EPFL, Computer Science Department, 1997.

[FEL 98a] PASCAL FELBER. "*The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*". PhD thesis, École Polytechnique Fédérale de Lausanne (Swiss Federal Institute of Technology), Lausanne, Switzerland, 1998.

[FEL 98b] PASCAL FELBER, RACHID GUERRAOUI, and ANDRÉ SCHIPER. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[FEL 02] VIOLETA FELEA, and BERNARD TOURSEL. "Methodology for Java Distributed and Parallel Programming Using Distributed Collections". In *Proceedings of the Workshop on Java for Parallel and Distributed Computing at IPDPS*, Fort Lauderdale, Florida, USA, April 2002.

[FOR 95] XTP FORUM. "Xpress Transport Protocol Specifications: XTP Revision 4.0". Technical report, XTP Forum, Santa Barbara, California, USA, May 1995.

[FOS 02] IAN FOSTER. What is the Grid? A Three Point Checklist. GridToday, July 2002.

[FOX 02] GEOFFREY FOX, MARLON PIERCE, DENNIS GANNON, and MARY THOMAS. "Overview of Grid Computing Environments". Informational, Global Grid Forum, 2002.

[FRA] The Fractal Project. `http://fractal.objectweb.org`.

[FRE 01] JAMES FREY, TODD TANNENBAUM, MIRON LIVNY, IAN FOSTER, and STEVEN TUECKE. "Condor-G: A Computation Management Agent for Multi-Institutional Grids". In *Proceedings of the 10th International Symposium on High Performance Distributed Computing (HPDC-10'01)*, pages 55–63, San Francisco, California, USA, August 2001. IEEE Computer Society.

[FRU 03] MICHAEL FRUMKIN, MATTHEW SCHULTZ, HAOQIANG JIN, and JERRY YAN. "Performance and Scalability of the NAS Parallel Benchmarks in Java". In *JPDC, Workshop on Java for Parallel and Distributed Computing at IPDPS*, Nice, France, April 2003.

[GDX] Grid eXplorer. `http://www.lri.fr/~fci/GdX/`.

[GEI 94] AL GEIST, ADAM BEGUELIN, JACK J. DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, and VAIDY SUNDERAM. "*PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing*". MIT Press, 1994.

[GET 99] VLADIMIR GETOV, PAUL GRAY, and VAIDY SUNDERAM. "MPI and Java-MPI: Contrasts and Comparisons of Low-Level Communication Performance". In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, Oregon, USA, November 1999.

[GET 01] VLADIMIR GETOV, GREGOR VON LASZEWSKI, MICHAEL PHILIPPSEN, and IAN FOSTER. Multiparadigm Communications in Java for Grid Computing. *Communications of the ACM*, 44(10):118–125, October 2001.

[GLO] The Globus Project. `http://www.globus.org`.

[GRO 96] WILLIAM GROPP, EWING LUSK, NATHAN DOSS, and ANTHONY SKJELLUM. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[HEN 03] BRIAN J. HENZ, and DALE R. SHIRES. "An Object-Oriented Programming Framework for Parallel Finite Element Analysis with Application: Liquid Composite Molding". In *PDSECA, Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications at IPDPS*, Nice, France, April 2003.

[HUE 02] FABRICE HUET. "*Objets mobiles : conception d'un middleware et évaluation de la communication*". PhD thesis, Université de Nice Sophia-Antipolis, Nice, France, December 2002.

[HUE 04] FABRICE HUET, DENIS CAROMEL, and HENRI E. BAL. "A High Performance Java Middleware with a Real Application". In *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA, November 2004.

[HUI 96] CHRISTIAN HUITEMA. "*IPv6: The New Internet Protocol*". Prentice Hall, 1996. ISBN 0-13-241936-X.

[ION 94] IONA, and ISIS. "An introduction to orbix+isis". Technical report, IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994.

[JEA 03] KARL JEACLE, and JON CROWCROFT. "Reliable High-Speed Grid Data Delivery Using IP Multicast". In *Proceedings of the UK e-Science All-Hands Meeting*, Nottingham, United Kingdom, September 2003.

[JGF] Java Grande Forum. `http://www.javagrande.org`.

[JUD 98] GLENN JUDD, MARK CLEMENT, and QUINN SNELL. DOGMA: Distributed Object Group Metacomputing Architecture. *Concurrency: Practice and Experience*, 10(11-13):977–983, September 1998.

[KAR 99] GEORGE KARYPIS, and VIPIN KUMAR. Parallel Multilevel k-way Partition Scheme for Irregular Graphs. *SIAM Review*, 41(2):278–300, 1999.

[KEA 01] KATARZINA KEAHEY, PATRICIA FASEL, and SUSAN MNISZEWSKI. "PAWS: Collective Interactions and Data Transfers". In *Proceedings of 10th International Symposium on High Performance Distributed Computing*, San Francisco, California, USA, August 2001.

[KIC 91] GREGOR KICZALES, JIM DES RIVIÈRES, and DANIEL G. BOBROW. "*The Art of the Metaobject Protocol*". MIT Press, July 1991. ISBN 0262610744.

[KLE 96] JÜRGEN KLEINÖDER, and MICHAEL GOLM. MetaJava: An Efficient Run-Time Meta Architecture for Java. International Workshop on Object Orientations in Operating Systems (IWOOOS), 1996.

[LAM 78] LESLIE LAMPORT. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[LAN 96] STÉPHANE LANTERI. Parallel Solutions of Compressible Flows Using Overlapping and Non-Overlapping Mesh Partitioning Strategies. *Parallel Comput.*, 22:943–968, 1996.

[LI 96] JIN LI. A Poly-Algorithm for Parallel Dense Matrix Multiplication on Two-Dimensional Process Grid Topologies. Master's thesis, Falculty of Mississippi State University, Mississippi, USA, December 1996.

[LI 01] MAOZHEN LI, OMER F. RANA, and DAVID W. WALKER. Wrapping MPI-based Legacy Codes as Java/CORBA components. *Future Generation Computer Systems*, October 2001.

[LIN 96] JOHN C. LIN, and SANJOY PAUL. "RMTP: A Reliable Multicast Transport Protocol". In *INFOCOM*, pages 1414–1424, San Francisco, California,USA, March 1996.

[MAA 00] JASON MAASSEN, THILO KIELMANN, and HENRI E. BAL. "Efficient Replicated Method Invocation in Java". In *Java Grande*, pages 88–96, San Francisco, California, USA, June 2000. ACM Press.

[MAA 01] JASON MAASSEN, ROB VAN NIEUWPOORT, RONALD VELDEMA, HENRI E. BAL, THILO KIELMANN, CERIEL J. H. JACOBS, and RUTGER F. H. HOFMAN. Efficient Java RMI for parallel programming. *Transactions on Programming Languages and Systems*, 23(6):747–775, 2001.

[MAA 02] JASON MAASSEN, THILO KIELMANN, and HENRI E. BAL. "GMI: Flexible and Efficient Group Method Invocation for Parallel Programming". In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'02)*, Washington D.C., USA, March 2002.

[MAA 03] JASON MAASSEN. "*Method Invocation Based Communication Models for Parallel Programming in Java*". PhD thesis, Vrije Universiteit, Amsterdam, The Netherlands, June 2003.

[MAF 95] SILVANO MAFFEIS. "*Run-Time Support for Object-Oriented Distributed Programming*". PhD thesis, Fakultat der Universitat Zurich, 1995.

[MAF 96] SILVANO MAFFEIS. "The Object Group Design Pattern". In *Proceedings of the Second USENIX Conference on Object-Oriented Technologies*, Toronto, Canada, June 1996.

[MAI 02] MOUFIDA MAIMOUR, and CONGDUC PHAM. "An Active Reliable Multicast Framework for the Grids". In *Proceedings of the International Conference on Computational Science*, pages 588–597, Amsterdam, The Netherlands, April 2002.

[MAN 98] ALLISON MANKIN, ALLYN ROMANOW, SCOTT BRADNER, and VERN PAXSON. "Criteria for Evaluating Reliable Multicast Transport and Application Protocols (RFC 2357)". Technical report, IETF Network Working Group, June 1998.

[MAY 02] ANTHONY MAYER, STEPHEN MC GOUGH, MURTAZA GULAMALI, LAURIE YOUNG, JIM STANTON, STEVEN NEWHOUSE, and JOHN DARLINGTON. "Meaning and Behaviour in Grid Oriented Components". In *3rd International Workshop on Grid Computing at Grid2002*, pages 100–111, Baltimore, Maryland, USA, November 2002. volume 2536 of LNCS.

[MIN 97] SAVA MINTCHEV, and VLADIMIR GETOV. "Towards portable message passing in Java: Binding MPI". In *Recent Advances in PVM and MPI*, number 1332 in LNCS. Springer Verlag, 1997.

[MOS 98] LOUISE E. MOSER, MICHAEL MELLIAR-SMITH, and PRIYA NARASIMHAN. Consistent Object Replication in the Eternal System. *Theory and Practice of Object Systems*, 4(2):81–92, April 1998.

[MPI94] "MPI: A Message-Passing Interface Standard". Technical report, MPI Forum, University of Tennessee, Knoxville, Tennessee, June 1994.

[MPI97] "MPI-2: Extensions to the Message-Passing Interface". Technical report, MPI Forum, University of Tennessee, Knoxville, Tennessee, August 1997.

[NAT 02] ANAND NATRAJAN, ANH NGUYEN-TUONG, MARTY A. HUMPHREY, and ANDREW S. GRIMSHAW. The Legion Grid Portal. *Concurrency and Computation: Practice and Experience*, 14(13-15):1365–1394, 2002.

[NEL 01] ARNOLD NELISSE, THILO KIELMANN, HENRI E. BAL, and JASON MAASSEN. "Object-based Collective Communication in Java". In *Joint ACM Java Grande - ISCOPE Conference*, pages 11–20, Palo Alto, California, USA, June 2001. ACM Press. ISBN 1-58113-359-6.

[NET] NetSolve. `http://icl.cs.utk.edu/netsolve`.

[NIN] Ninf-G. `http://ninf.apgrid.org`.

[OMG01] Object Management Group. "*Unreliable Multicast Inter-ORB Protocol*", October 2001. OMG Specifications.

[OMG04] Object Management Group. "*Event Service Specification (version 1.2)*", October 2004. OMG Specifications.

[PAU 97] SANJOY PAUL, KRISHAN K. SABNANI, JOHN C. H. LIN, and SUPRATIK BHATTACHARYYA. Reliable multicast transport protocol (RMTP). *IEEE Journal of Selected Areas in Communications*, 15(3):407–421, 1997.

[PHI 00] MICHAEL PHILIPPSEN, BERNHARD HAUMACHER, and CHRISTIAN NESTER. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[PIP 02] SERGE PIPERNO, MALIKA REMAKI, and LOULA FEZOUI. A Nondiffusive Finite Volume Scheme for the Three-Dimensional Maxwell's Equations on Unstructured Meshes. *SIAM J. Numeri. Anal.*, 39(6):2089–2108, 2002.

[PRO] ProActive. `http://www-sop.inria.fr/oasis/ProActive`.

[REN 93] ROBBERT VAN RENESSE, KENNETH P. BIRMAN, ROBERT COOPER, BRABDFORD GLADE, and PATRICK STEPHENSON. The Horus system: Reliable Distributed Computing with the Isis Toolkit. IEEE Computer Society Press, September 1993.

[REN 94] ROBBERT VAN RENESSE, TAKAKO M. HICKEY, and KENNETH P. BIRMAN. "Design and performance of Horus: A lightweight group communications system". Technical report, Department of Computer Science, Cornell University, Ithaca, New York, USA, 1994. TR94-1442.

[REN 96] ROBBERT VAN RENESSE, KENNETH P. BIRMAN, and SILVANO MAFFEIS. Horus: A flexible group communication system. *Communications of the ACM*, 39(4), 1996.

[ROS 98] PHIL ROSENZWEIG, MIRIAM KADANSKY, and STEVE HANNA. "The Java Reliable Multicast Service: A Reliable Multicast Library". Technical report, Sun Microsystems, Palo Alto, California, USA, September 1998. TR-98-68.

[ROZ 92] M. ROZIER, V. ABROSSIMOV, F. ARMAND, I. BOULE, M. GIEN, M. GUILLEMONT, F. HERRMAN, C. KAISER, S. LANGLOIS, P. LÉONARD, and W. NEUHAUSER. "Overview of the Chorus Distributed Operating System". In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle, Washington, USA, 1992.

[SAN 03] MICHELE DI SANTO, NADIA RANALDO, and EUGENIO ZIMEO. "A Broker Architecture for Object-Oriented Master/Slave Computing in a Hierarchical Grid System". In *Proceedings of Parallel Computing*, Dresden, Germany, September 2003.

[SAU 03] STEVEN SAUNDERS, and LAURENCE RAUCHWERGER. "ARMI: An Adaptive, Platform Independant Communication Library". In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 230 – 241, San Diego, California, USA, June 2003. ISBN 1-58113-588-2.

[SET] SETI@home: the Search for ExtraTerrestrial Intelligence. `http://setiathome.ssl.berkeley.edu`.

[SHI 01] MATTHEW S. SHIELDS, OMER F. RANA, and DAVID W. WALKER. "A Collaborative Code Development Environment for Computational Electro-Magnetics". In *IFIP TC2 / WG2.5 Working Conference on the Architecture of Scientific Software*, pages 119–141. Kluwer Academic Publishers, 2001.

[SQU 96] JEFFREY M. SQUYRES, BRIAN C. MCCANDLESS, and ANDREW LUMSDAINE. "Object Oriented MPI: A Class Library for the Message Passing Interface". In *Proceedings of the POOMA conference*, Santa Fe, New Mexico, USA, February 1996.

[SUN98] Sun Microsystems. "*Java Remote Method Invocation Specication*", October 1998. `ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf`.

[TAM 98] MILIND TAMBE, JAFAR ADIBI, YASSER ALONAIZON, ALI ERDEM, GAL A. KAMINKA, STACY C. MARSELLA, ION MUSLEA, and MARCELLO TALLIS. "ISIS: Using an explicit model of teamwork in RoboCup'97". In *RoboCup'97: Proceedings of the first robot world cup competition and conferences*. Springer Verlag: Heidelberg, Germany, 1998.

[TAN 90] ANDREW S. TANENBAUM, ROBBERT VAN RENESSE, HANS VAN STAVEREN, GREGORY J. SHARP, SAPE J. MULLENDER, JACK JANSEN, and GUIDO VAN ROSSUM. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[TAN 94] ANDREW S. TANENBAUM. "*Distributed Operating Systems*". Pearson Education, 1994. ISBN 0132199084.

[UNI] Unicore. `http://unicore.sourceforge.net`.

[VEL 98] TODD L. VELDHUIZEN, and DENNIS GANNON. "Active Libraries: Rethinking the roles of compilers and libraries". In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, Pennsylvania, USA, 1998.

TYPED GROUPS FOR THE GRID

Abstract

Group communication is a crucial feature for high-performance and Grid computing. While previous works and libraries proposed such a characteristic, the use of groups imposed specific constraints on programmers, for instance the use of dedicated interfaces to trigger group communications; this thesis presents a more flexible mechanism. More specifically, it proposes a scheme where, given a Java class, one can initiate group communications using the standard public methods of the class together with the classical dot notation; in that way, group communications remain typed.

Furthermore, groups are automatically constructed to handle the result of collective operations, providing an elegant and effective way to transparently gather operations. This group communication system is based on a Meta-Object Protocol. This system allows an object notation and a dynamic management of the results (ex: `B groupB = groupA.foo();`). This flexibility also allows to handle results that are groups of remotely accessible objects, and to use a group as a means to dispatch different parameters to different group members. In addition, hierarchical groups can be easily and dynamically constructed; an important feature to achieve the use of several clusters in Grid computing. Performance measures and a numerical software demonstrate the viability of the approach.

Last works led to an Object-Oriented SPMD (Single Program Multiple Data) programming style, based on the typed group communication, which allows extended numerical programming abilities while keeping all the benefit of the typed approach. For this, the objects groups supporting the distributed computation can also be further organized according to a topology, i.e. adding the notion of an ID for each member in the SPMD group and the way to easily reference its neighbors. Collective operations were revisited and extended with barrier synchronization such as providing a complete Object-Oriented SPMD model.

Keywords : Group Communication, Object-oriented programming, Middleware.

GROUPES TYPÉS POUR LA GRILLE

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 imposent des contraintes spécifiques aux programmeurs (par exemple l'utilisation d'interfaces consacrées) pour effectuer des communications de groupes, cette thèse présente un mécanisme qui se veut plus flexible. En particulier, nous proposons un modèle, où, étant donnée une classe Java, les communications de groupes sont déclanchées par appel aux méthodes publiques de la classe en conservant la notation pointée; de cette façon les communications et les groupes deviennent typés.

De plus, des groupes sont automatiquement construits pour collecter les résultats d'une opération collective. Ce système est basé sur un Protocole à Méta-Objets. Cela permet une notation objet et une gestion dynamique des résultats (ex: `B groupB = groupA.foo();`). Cette flexibilité permet également de gérer les résultats qui sont eux mêmes des groupes d'objets accessibles à distance, et d'utiliser un groupe comme paramètre d'appel de méthode pour que ses membres soient distribués entre les membres d'un groupe d'appel. De plus, des groupes hiérarchiques peuvent être facilement et dynamiquement construits : une importante fonctionnalité de déploiement dans un contexte de grilles. Des mesures de performances et une application numérique démontrent la viabilité de l'approche.

Nos derniers travaux mènent à un style de programmation SPMD (Single Program Multiple Data) orienté-objet basé sur les communications de groupes typés et qui permet un contrôle étendu sur des applications de calculs intensifs tout en préservant les bénéfices d'une approche typée. Les groupes d'objets soutenant le calcul distribué sont organisés selon une topologie, c'est à dire l'ajout de la notion d'une identification pour chaque membre dans le groupe SPMD et la possibilité de référencer facilement ses voisins. Les opérations collectives ont été revisitées et étendues par des barrières de synchronisation de façon à fournir un modèle complet de programmation SPMD orienté-objet.

Mots-clefs : Communication de groupe, Programmation orientée-objet, Intergiciel.