Rapport de DEA Réseaux et Systèmes Distribués
Laurent Baduel

Communication de Groupes Efficace pour Objets Actifs Répartis

INRIA Sophia-Antipolis, Equipe OASIS
Juin 2001

Remerciements

Je remercie tout d'abord Denis Caromel, qui m'a fait confiance en acceptant ma candidature pour ce stage. Notre travail en commun a été des plus agréables.

Merci à Fabrice Huet et Julien Vayssière, qui ont su répondre avec patience et gentillesse à mes nombreuses questions, et qui sont devenus bien plus que des collègues de travail.

Merci à Françoise Baude de m'avoir fourni des articles intéressants qui m'ont été particulièrement utiles.

Merci à Arnaud Contes d'avoir été un binôme de bureau très sympathique. Nous nous sommes immergés ensemble au cœur ProActive.

Enfin, merci à l'équipe OASIS dans son intégralité, pour la bonne humeur et la gentillesse de chacun de ses membres et pour l'excellente ambiance qui a régné tout au long de mon stage.

Table des matières

1 Introduction **8**
1.1 Objectifs . . . 8
1.2 La communication de groupes dans les systèmes distribués . . . 8

2 Théorie **10**
2.1 Définitions . . . 10
2.2 Structure de groupe . . . 10
2.2.1 Les groupes de pairs . . . 10
2.2.2 Les groupes client-serveur . . . 11
2.2.3 Les groupes de diffusion . . . 11
2.2.4 Les groupes hiérarchiques . . . 11
2.3 « 1 to N » contre « N to M » . . . 12
2.4 Adressage . . . 12
2.5 Fiabilité . . . 13
2.6 Ordonnancement . . . 13
2.6.1 Définitions . . . 13
2.6.2 Sémantique de chevauchement . . . 13
2.7 Sémantiques de réception . . . 14
2.8 Sémantique de réponse . . . 14
2.9 Homogénéité de groupes d'objets . . . 14

3 Etat de l'art **15**
3.1 ISIS . . . 15
3.2 Horus . . . 16
3.3 Amoeba . . . 17
3.4 V System . . . 17
3.5 Transis . . . 18
3.6 JavaGroups . . . 19

4 Présentation de ProActive PDC **20**
4.1 Objets Actifs . . . 21
4.2 Synchronisation . . . 21
4.2.1 Futurs . . . 21
4.2.2 Attentes par nécessité . . . 21
4.3 Communication par messages . . . 21
4.4 Mobilité . . . 22
4.5 MOP : Meta Object Protocol . . . 22

5 Réflexion sur la communication de groupes dans ProActive **24**
5.1 Buts . . . 24
5.2 La syntaxe . . . 24
5.3 L'interface `Group` . . . 26
5.4 Le proxy de groupe . . . 26
5.5 Groupe de futurs . . . 27
5.6 Synchronisation . . . 27

6 Implémentation 28
6.1 Architecture . . . 28
6.2 Groupes hiérarchiques . . . 28
6.3 Limitations . . . 30
6.3.1 Goupes d'objets non-réifiables . . . 30
6.3.2 Référence distante et perte de la cohérence . . . 30
6.3.3 Tolérance aux pannes . . . 31
6.4 Améliorations déjà effectuées . . . 31
6.4.1 Le `TransparentProxy` . . . 31
6.4.2 La méthode `optimize` . . . 31
6.5 Caractéristiques de la communication de groupes de ProActive . . . 32
6.6 Performances . . . 32
6.7 Extensions en cours . . . 34
6.7.1 Un pool de threads . . . 34
6.7.2 Transformation du proxy de groupe en objet actif . . . 34
6.7.3 La méthode `getFirstResult` . . . 35

7 Conclusion 36
7.1 Apports . . . 36
7.2 Perspectives . . . 36

A Exemple d'utilisation 37

B La Javadoc 39

Table des figures

2.1 Groupe de pair 11
2.2 Groupe client-serveur 11
2.3 Groupe de diffusion 11
2.4 Groupe hierarchique 12
2.5 1 to N et N to M 12

3.1 Les couches Horus 16
3.2 Les protocoles de Transis 18

4.1 Distribution par objets actifs 20
4.2 Objet actif 23

5.1 *objets groupés* et *groupe d'objet* 26

6.1 Structure d'un groupe 29
6.2 Perte de la cohérence 30
6.3 La méthode `optimize` 31
6.4 Tests de performances 33

Liste des tableaux

3.1 Caractéristiques de Isis 16
3.2 Caractéristiques de Horus 16
3.3 Caractéristiques de Amoeba 17
3.4 Caractéristiques de V System 18
3.5 Caractéristiques de Transis 19
3.6 Caractéristiques de JavaGroups 19

6.1 Caractéristiques de JavaGroups 32

Chapitre 1

Introduction

1.1 Objectifs

Mon stage, au sein de l'équipe OASIS (INRIA , Sophia-Antipolis) a eu pour but d'étudier les mécanismes de communications de groupes, afin de proposer un modèle de gestion de groupes typés pour une bibliothèque Java de programmation parallèle et distribuée, appelée ProActive. Ce travail s'inscrit dans la ligne d'intégration de nouvelles fonctionnalités à ProActive.

Ce rapport présente tout d'abord les diverses possibilités qu'offre la communication de groupes dans la conception et la réalisation d'applications distribuées. Après avoir exposé dans le chapitre 2 les concepts fondamentaux qui définissent les différentes caractéristiques d'un modèle de communication par groupes, le chapitre 3 présente plusieurs réalisations, en précisant pour chacune d'entre elles les choix de mise en œuvre qui ont été faits. Ensuite, dans le chapitre 4, la bibliothèque ProActive est présentée et détaillée dans le but de cerner ses fonctionnalités (objets actifs, migration, synchronisation) pour concevoir au mieux un modèle qui réponde aux attentes d'utilisateurs. Le chapitre 5 décrit le raisonnement qui a guidé l'élaboration de la communication de groupes dans ProActive. Pour finir le chapitre 6 se focalise sur l'implémentation, et expose les performances, limitations et améliorations possibles.

1.2 La communication de groupes dans les systèmes distribués

Le paradigme du groupe est reconnu comme étant une excellente méthode pour structurer les activités dans un système distribué [VsR92, VR92]. Depuis les premiers projets, notamment V-Kernel, les travaux sur les communications de groupes n'ont cessé d'évoluer ; particulièrement grâce au système ISIS qui a introduit des concepts très importants tels que l'ordonnancement total ou causal, et la garanti de la consistance.

Les applications types utilisant les systèmes de communication de groupes sont principalement les systèmes tolérant aux pannes [BDM98, EMS95, GS96], les réplications de machines à état, les transactions distribuées [Hav99], les bases de données dupliquées [ADMSM94], la répartition de charges et l'allocation de ressources [Kha96, KFL98], et les moniteurs.
Plus proches de l'utilisateur, on trouve des applications collaboratives telles que des systèmes de visioconférence, de « tableau blanc » ,... [ANM01, Wil95]. Le terme de « *groupware* », introduit dans [UN94], désigne les logiciels collaboratifs basés sur des systèmes à communication de groupes.

Des optimisations du temps de calculs sont obtenues par utilisation de communication de groupes. On note spécialement les mécanismes de diffusion et de duplication. Comme présenté dans [AS98], la diffusion permet, à faible coût de propager une information ou une requête vers un grand nombre de destinataires. Certaines structures de groupes sont très bien adaptées à ce problème. La duplication est le fait de copier un certain nombre de fois les données ou les actions à effectuer. Par duplication, on peut par exemple, fournir les mêmes données et requêtes à différents serveurs et n'attendre que la réponse du plus rapide.
Il est possible également de partager le travail entre les membres d'un groupe, soit en divisant la charge de travail equitablement entre les membres, soit en attribuant à chacun d'entre eux une part plus

ou moins importante du travail. Il s'agit alors de répartition de charges et d'allocation des ressources.

La tolérance aux pannes est produite en dupliquant les données sur plusieurs sites d'un système distribué (machines,réseaux,...). Dans le cas où un site tombe en panne une duplication des données (ou des services) est disponible sur un autre site.

Chapitre 2

Théorie

2.1 Définitions

Un groupe est un ensemble d'entités simultanément accéssibles.

La communication de groupe est l'envoie en une action d'un message vers plusieurs destinations. Les termes suivants sont définis :

- « *UniCast* » : communication d'un point vers un autre (une source, un récepteur).
- « *MultiCast* » : communication d'une (ou plusieurs) source(s) vers plusieurs récepteurs.
- « *BroadCast* » : communication d'une source (ou plusieurs) vers tous les récepteurs. Le broadcast est un cas particulier du multicast

Les entités susceptibles d'être incorporées au sein d'un groupe peuvent aussi bien être des processus, des processeurs ou des objets. Chacun de ces trois termes pourra être employé pour désigner un élément membre ou non d'un groupe.

Un mécanisme de communication de groupes possède plusieurs caractéristiques [VKCD99]. Il y a des choix de design à faire avant la réalisation. Aucun choix n'est réellement meilleur qu'un autre, certains sont mieux adaptés que d'autres à certains environnements et/ou applications.

2.2 Structure de groupe

Les groupes peuvent être ouverts ou fermés. Dans un groupe fermé, seuls les membres du groupe peuvent envoyer des messages au groupe alors que dans un groupe ouvert les non-membres en sont aussi capables.

La dynamicité est la capacité de pouvoir changer l'état d'un groupe en cours d'exécution du programme. Les groupes dynamiques peuvent voir le nombre de leurs membres varier au cours du temps. Si un groupe n'est pas dynamique, il est dit statique : une fois le groupe formé, il est impossible pour un objet de quitter ou de joindre le groupe.

Selon le type d'application, la forme du groupe peut être différente. Pour s'adapter aux besoins spécifiques de certains logiciels plusieurs formes de groupe sont définies. Principalement on les classe en quatre catégories :

2.2.1 Les groupes de pairs

Ils sont composés de membres qui travaillent en coopération pour achever un but particulier. Ce style de groupe est particulièrement utilisé dans les applications tolérantes aux pannes ou à partage de charge. Le problème principal des groupes de pairs est la mise à l'échelle. Plus ils deviennent grands plus leurs performances sont réduites.

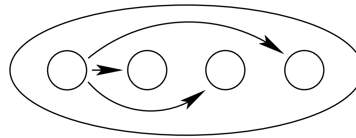


FIG. 2.1 – *Groupe de pair*

2.2.2 Les groupes client-serveur

Ils sont constitués d'un nombre de client éventuellement grand et d'un groupe de serveurs pairs. Une requête est envoyée vers un ou plusieurs membres du groupe par messages ou par appel distant de procédure (RPC), de la part d'un membre du groupe ou non. Dans tous les cas le serveur contacté réponds par communication point à point ou multicast à l'expéditeur de la requête et aux autres membres du groupe.

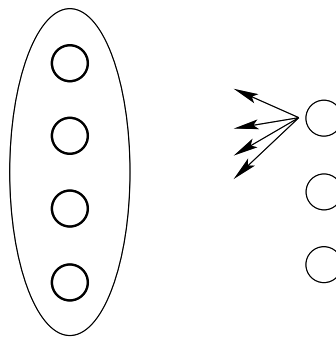


FIG. 2.2 – *Groupe client-serveur*

2.2.3 Les groupes de diffusion

Ils sont un cas spécial des groupes client-serveur. Ici un seul message est envoyé par un serveur vers tous les clients. Le groupe de diffusion est le seul type de groupe qui bénéficie pleinement d'une amélioration des performances procurée par l'utilisation de réseaux multicast.

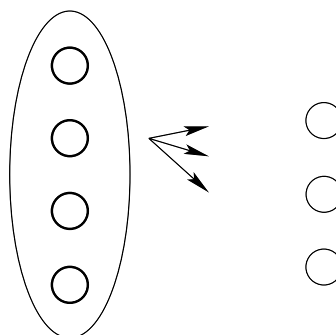


FIG. 2.3 – *Groupe de diffusion*

2.2.4 Les groupes hiérarchiques

Ils sont une extension de la forme client-serveur. Dans les applications qui réclament une distribution sur un grand nombre de machines, il est important de localiser les interactions entre membres. En regroupant les membres qui ont une forte interaction on constitue des sous-réseaux . A ces sous-réseaux on

fait correspondre des sous-groupes : c'est le mapping. Le principal inconvénient des groupes hiérarchiques est que le groupe de base représente un point centralisé, dont la panne serait critique pour l'ensemble du système.

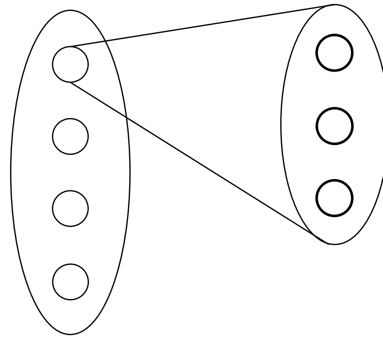


FIG. 2.4 – *Groupe hierarchique*

2.3 « 1 to N » contre « N to M »

On qualifie la communication de « 1 to N » lorsque que la source est unique et le destinataire un groupe. On la qualifie de « N to M » lorsque la source et le destinataire sont des groupes. Ces deux types de communications sont des communications multicast. La communication 1 to N peut être considéré comme une première étape vers la communication N to M. La plupart des systèmes de communication de groupes fournissent du 1 to N, l'implémentation de la communication N to M est laissé à la charge du programmeur. Le N to M produit une sémantique de retour spécifique : si un membre des N clients invoque une méthode sur le groupe des M serveurs, le retour de cette méthode doit être diffusé aux N clients.

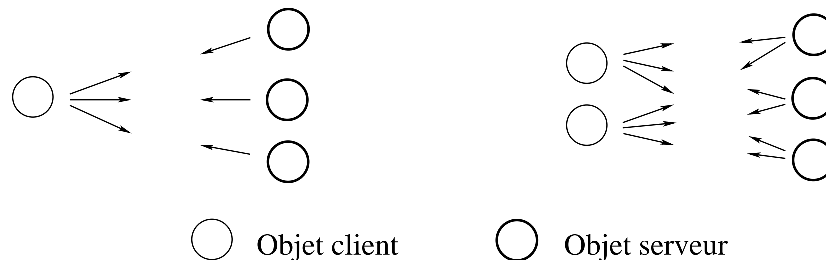


FIG. 2.5 – *1 to N et N to M*

2.4 Adressage

Il existe deux façons de concevoir l'adressage.

La plus évidente est de recourir à un expéditeur qui spécifie toutes les destinations. Pour un groupe de N éléments, N messages sont envoyés sur le réseau.

La seconde méthode est d'utiliser une unique adresse pour désigner l'intégralité du groupe. On introduit ici la notion de service pour communication de groupes : l'expéditeur poste un message vers le service de multicast et celui-ci le diffuse au groupe destinataire. Cette méthode permet à un processus d'envoyer un message sans connaître les membres du groupe destination. C'est le service qui possède les informations suffisantes pour mener à bien l'opération. Cependant le service est centralisé et représente un point faible pour l'application en cas de panne du site de service.

2.5 Fiabilité

La fiabilité dans une communication point à point est la certitude qu'un message envoyé est arrivé à destination. Dans un contexte de communication de groupes, la fiabilité assure que tous les membres d'un groupe ont bien reçu un message. Si un membre est dans l'incapacité de recevoir un message, les autres membres ne doivent pas considérer ce message. Ce mécanisme est relativement simple à mettre en œuvre lorsque les messages sont émis depuis un point centralisé mais il nuit aux performances. En effet, des attentes sont nécessaires pour détecter que le message est perdu ou défectueux.

2.6 Ordonnancement

2.6.1 Définitions

L'ordonnancement représente l'ordre d'arrivée des messages vers les membres du groupe. On peut différencier quatre types d'ordonnancement : le non-ordonnancement, l'ordonnancement FIFO, l'ordonnancement total et l'ordonnancement causal.

- Le non-ordonnancement ne fais aucune garantit sur l'ordre de réception des messages.
- L'ordonnancement FIFO garantit que les messages envoyés sont reçus dans le même ordre que leur ordre d'émission.
- L'ordonnancement total est un mécanisme qui garantit que tous les messages sont reçus par tous les destinataires dans le même ordre. Ce principe assure un fonctionnement cohérent de l'application mais induit des temps de latence pour s'assurer que tous les messages ont été reçus avant d'émettre les suivants. [VVR92] présente des optimisations possibles pour un système à ordonnancement total : RPM (Reliable Multicast Protocol). L'ordonnancement total permet d'éviter des incohérences : lorsque deux messages sont envoyés simultanément vers un même groupe, des problèmes peuvent survenir si les messages ne sont pas reçus dans le même ordre (par exemple des mises à jour et lecture de variables).
- L'ordonnancement causal est une autre façon d'éliminer ce problème. Une dépendance causale existe entre deux messages (requêtes) si le résultat de l'un dépend de l'autre. Deux messages ne peuvent être expédier simultanément s'ils sont en dépendances causales, sinon cela ne pose aucun problème. Des mécanismes de détection de causalité dans les systèmes distribués sont présentés dans [SM92].

Une méthode est proposé dans [TFC99] pour prouver formellement l'ordonnancement d'un système multicast.

Un protocole de communication de groupes à ordonnancement total, basé sur des envois de messages asynchrones est proposé dans [DKM93].

2.6.2 Sémantique de chevauchement

Si un objet peut être membre de plusieurs groupes, une sémantique de chevauchement doit être définie. Supposons que plusieurs processus sont membres des groupes G1 et G2 et que chacun de ces groupes garantissent un ordonnancement total. Un choix doit être fait quant à l'ordonnancement des messages de $G1 \cap G2$. Les différents types d'ordonnancement présentés précédemment sont possibles (aucun ordonnancement, FIFO, total ou causal):

- L'ordonnancement total garantit que les messages de chaque groupe sont reçus dans le même ordre par tous les membres qu'ils appartiennent ou non à l'intersection. Une seconde requête ne pourra être émis vers l'un ou l'autre des deux groupes en intersection que lorsque la précédente sera terminé.
- Dans le cas d'un ordonnancement causal, les messages pourront « s'entrelacer » s'ils ne sont pas en dépendance causal.
- L'ordonnancement FIFO va garantir que l'ordre de réception des messages est le même que l'ordre d'émision, aussi bien dans l'intersection que dans le reste des deux groupes.

– Le non ordonnancement ne garantit rien sur l'ordre d'arrivée des messages.

2.7 Sémantiques de réception

La sémantique de réception des messages est définie communément selon trois catégories. Par « k-delivery », une communication multicast est considérée comme réussie si k processus ont reçu le message. k est une constante dont la valeur peut être zéro. En « quorum delivery » une majorité de messages doit arriver à destination. L'idéal est l' « atomic delivery » qui assure que tous les processus vivants du groupe reçoivent le message. Une réception en « atomic delivery » est fibale (cf section 2.5).

2.8 Sémantique de réponse

Parallèlement la sémantique de réponse (ou de retour) s'occupe de négocier le nombre de réponse de la communication multicast. L'expéditeur peut s'attendre à ne recevoir aucune réponse, au moins une réponse, plusieurs réponses ou toutes les réponses.

Les termes de réponse « partielle » et « totale » sont également employés. Une réponse est dite partielle s'il s'agit de k-delivery ou de quorum delivery. On parle de réception « totale » si la sémantique est atomic delivery. On définit aussi les sémantiques de réception « aucune » qui est un cas particulier de la k-delivery où k vaut 0, et « unique » où k vaut 1.

2.9 Homogénéité de groupes d'objets

Dans le contexte d'un langage à objets, les groupes sont des groupes d'objets. On peut se demander si un groupe peut être constitué d'éléments hétérogènes. Les membres d'un groupe doivent-ils forcément implémenter une même interface, ou être polymorphe (propriété de l'héritage) ?

Si tous les membres du groupe implémentent une même interface, effectuer un appel de méthode ne pose pas de problème. Par contre si les groupes sont hétérogènes, il faut définir un mécanisme de filtrage des appels de méthodes sur les membres du groupe.

Chapitre 3

Etat de l'art

La communication de groupes peut être insérée dans une application répartie à différents niveaux.

On peut recourir à un *système d'exploitation* qui fournit des primitives capables d'opérer broadcasts et multicasts. On peut utiliser un *protocole de communication* particulier qui permet le multicast, ou alors il est possible d'employer une *bibliothèque spécialisée*.

Une grande majorité des mécanismes de groupe fonctionnent grâce à un service de multicast. Le groupe possède une adresse, et les messages sont postés vers cette adresse. Un service se charge d'ordonnancer et de rediffuser les messages, il s'occupe également de gérer la liste des membres du groupe. Il est à noter que plusieurs services peuvent se partager les tâches. Par exemple, un mécanisme de multicast fiable de CORBA proposé par Pascal Felber et Rachid Guerraoui utilise les services de nommage, de messaging, de consensus, de monitoring et de multicast [1] déjà présent dans CORBA [FGG96, FGS97, FGS98].

Une approche intéressante est celle de la bibliothèque JavaGroups de l'université de Cornell. La notion de groupe est exprimée à travers une interface, et plusieurs façons de gérer le groupe peuvent être utilisées.

Voici les principaux systèmes de communications de groupes :

3.1 ISIS

ISIS [Unia] a été développé à l'université de Cornell pour bâtir des applications distribuées éventuellement tolérantes aux pannes. Le modèle de programmation de ISIS est basé sur des groupes de processus virtuellement synchrones. Dans une synchronisation virtuelle, tous les membres d'un groupe reçoivent un flux ordonné et consistant d'événements. Cette synchronisation fait apparaître le système comme étant synchrone au niveau applicatif. Fondamentalement, ISIS est basé sur une approche de machine à état pour la programmation de la tolérance aux pannes. Un des avantages majeur de l'implémentation de ISIS est la capacité à fonctionner sur tout réseau fournissant un protocole internet de communications de groupes multicast (IGMP : Internet Group Multicast Protocol).

L'abstraction de communication de groupes employée dans ISIS est un service de multicast composé de plusieurs protocoles de base qui fournissent différents types d'opération. Le plus important de ces protocoles de base est CBCAST (Causal BroadCAST) qui permet l'envoi et la réception de messages de façon atomique et le maintient d'un ordonnancement causal. ISIS implémente également le protocole ABCAST (Atomic BroadCAST) pour assurer un ordonnancement total des messages, ainsi que GBCAST (Group BroadCAST) le protocole qui gère les messages de création / destruction / appartenance à un groupe. Dans l'implémentation de ISIS, un service de temps est à la base des mécanismes de CBCAST, ABCAST et GBCAST. Ce service est aussi lié au service d'appartenance au groupe (Group membership) pour fournir une description précise de l'état du groupe à un instant donné.

Les concepts de synchronisation virtuelle et de protocoles CBCAST / ABCAST / GBCAST sont exposés dans [Bir93].

1. Le service de multicast de CORBA permet seulement la diffusion de message sans offrir aucune qualité de service ni aucun ordonnancement

Appartenance aux groupes	sémantique dynamique de gestion de groupe.
Structure de groupe	Les groupes sont des groupes ouverts.
Ordonnancement des messages	ordonnancement total, causal ou non-ordonnancement.
Sémantique de réception et de réponse	réception atomique des messages et réponse totale.
Fiabilité des transmissions	mécanisme uniquement fiable de communication.

TAB. 3.1 – *Caractéristiques de Isis*

3.2 Horus

Horus a été développé à l'université de Cornell en tant que version améliorée de ISIS. Horus a été implémenté comme un sous-système de communication de groupes totalement portable. Un micro-noyau fournit les primitives de multicast. La motivation principale de Horus était de séparer clairement les mécanismes de base de multicast et l'utilisation finale au niveau applicatif. Horus est bâtit selon plusieurs couches.

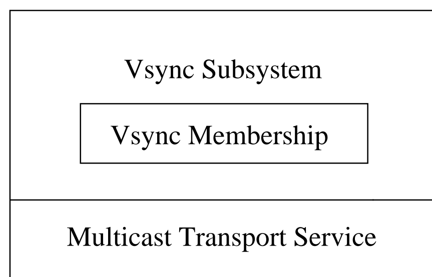


FIG. 3.1 – *Les couches Horus*

Le MUST (MUlticast Transport Service) est la couche fournissant les primitives de multicast telles que la communication asynchrone et fiable de 1 vers N. Le MUST est capable de supporter plusieurs protocoles de communication.

Le Vsync (Virtual Synchronous subsystem) est placé au-dessus du MUST, il fournit un environnement complet de programmation par communication de groupes. Vsync est implémenté comme une collection de différents services offrant la même interface. Pour chaque service, Vsync est composé de 2 sous-couches :

Vsync Membership fonctionne directement sur le MUST et offre une administration de groupe fiable et un transport sûr.

Vsync Subsystem fonctionne sur Vsync Membership, il fournit les ordonnancements total et causal et la tolérance aux pannes.

Horus étend la sémantique de ISIS. Les améliorations et optimisations de Horus sont présentées dans [RBM96, vRHB94].

Appartenance aux groupes	sémantique dynamique de gestion de groupe.
Structure de groupe	les groupes sont des groupes ouverts.
Ordonnancement des messages	ordonnancement total, causal ou non-ordonnancement.
Sémantique de réception et de réponse	réception atomique des messages et réponse totale.
Fiabilité des transmissions	mécanismes fiable et non-fiable de communication.

TAB. 3.2 – *Caractéristiques de Horus*

3.3 Amoeba

Amoeba [Unic] a été développé à l'université de Vrije, Amsterdam. Amoeba est un système d'exploitation distribué. La communication de groupes a été inclue comme partie intégrante du système d'exploitation. Le noyau du système Amoeba fournit des RPC et un ensemble de fonctions de communications de groupe.

Amoeba utilise un protocole de broadcast du réseau (le plus souvent ethernet) [KT94]. Quand une application souhaite envoyer un message à un groupe, le message est donné au service de communication inter-processus qui forme un paquet, puis ce paquet est envoyé à un autre service, le séquenceur, qui se charge de le diffuser. Si le service de séquenceur tombe en panne, un processus de ré-élection en initialise un nouveau. L'avantage de ce service de séquenceur est qu'il fournit simplement un ordonnancement total des messages. De plus, la fiabilité des communications de groupes est assurée par le protocole de broadcast.

Contrairement aux autres systèmes, Amoeba possède une sémantique de groupes fermés. Un processus qui ne fait pas parti d'un groupe ne peut pas communiquer avec un membre de ce groupe. Lorsqu'un appel RPC est effectué, le client ne sait pas s'il s'adresse à un serveur simple ou à un groupe de serveurs. Cela permet de préserver le concept de transparence entre le client et le serveur.

Le principal inconvénient de la communication de groupe d'Amoeba est qu'elle ne supporte pas la mise à l'échelle.

[KT96, KTV92] présentent les caractéristiques d'Amoeba, et propose l'implémentation d'un système de tolérance aux pannes.

Appartenance aux groupes **Structure de groupe**	sémantique dynamique de gestion de groupe. les groupes sont des groupes fermés.
Ordonnancement des messages	uniquement ordonnancement total.
Sémantique de réception et de réponse	Il n'y a pas de réelle sémantique de réception, cependant on peut considérer que les appels se font par RPC. La sémantique de réponse peut être totale ou aucune.
Fiabilité des transmissions	mécanisme de communication fiable.

TAB. 3.3 – *Caractéristiques de Amoeba*

3.4 V System

V System est lui aussi un système d'exploitation distribué, développé à l'université de Stanford, Californie. V System place les primitives de communication de groupes au sein de RPC. C'est par un « simple » appel RPC qu'un client dialogue avec un groupe de serveurs. Le protocole de communication tente de propager la requête à tous les serveurs. La communication est non-fiable et non-ordonnée. Dès qu'un processus serveur a répondu à la requête, celle-ci est considérée comme réussie et terminée. La sémantique de la communication de groupes de V System , n'attend qu'une seule réponse. Si au niveau applicatif on attend plusieurs réponses, de nouveaux appels RPC seront effectués.

La méthode de communication de V System est étonnamment simple et efficace. Cependant il ne fournit qu'une sémantique assez pauvre et aucun mécanisme de tolérance aux pannes.

Appartenance aux groupes	sémantique dynamique de gestion de groupe.
Structure de groupe	les groupes sont des groupes ouverts.
Ordonnancement des messages	uniquement le non-ordonnancement.
Sémantique de réception et de réponse	La sémantique de réception n'est pas clairement définie, on peut la considérer comme des appels RPC. La sémantique de réponse peut être toutes les réponses ou plusieurs réponses.
Fiabilité des transmissions	mécanisme de communication non-fiable.

TAB. 3.4 – *Caractéristiques de V System*

3.5 Transis

Transis [oJ] est un protocole fiable et efficace [Cho97] réalisé par l'université de Jérusalem. Transis a été développé en vu de fournir une grande variété de services pour la communication entre processus UNIX. Transis fonctionne selon un mécanisme de broadcast par domaine . Par manipulation des domaines et sous-domaines (ajouter, retirer,...) un groupe est formé, ensuite par utilisation du broadcast sur le domaine on communique vers tous les membres. Transis utilise des services de communication et de gestion de groupe. Les groupes de processus sont virtuellement synchrones . Cependant Transis étend le concept de synchronisation virtuelle de ISIS : la consistance est maintenue malgré les modifications que peuvent subir les domaines et les réseaux (même les pannes).

L'architecture de Transis décrite dans [ADKM92] est composée de deux protocoles : Lansis et Xport.

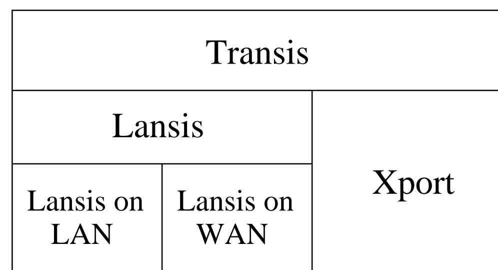


FIG. 3.2 – *Les protocoles de Transis*

Le protocole Lansis fournit un environnement de broadcast de domaine cohérent. Par abus de langage on parle de domaine « Lansis ». Tous les processus appartenant à un même domaine « Lansis » ont la même vision du système et se comportent de la même manière . Le protocole Lansis est lui-même composé de « Lansis pour LAN » et « Lansis pour WAN », respectivement pour les réseaux de petite et grande échelle . Cela permet à Transis de rester efficace malgré une mise à l'échelle.

Le protocole Xport permet l'interconnexion des domaines « Lansis ». Il fournit une communication fiable en utilisant un port sélectif pour transférer les messages entre les domaines. Un port sélectif filtre les messages ne laisse passer que ceux qui concernent le domaine.

Lansis est basé sur le principe que tous les messages doivent être reçus par tous les processus du domaine. Si un message est perdu par un processus, celui-ci envoie une requête aux autres processus qui ont reçu le message pour qu'on lui renvoie. Lansis utilise un système composé d'acquittements et de non-acquittements pour assurer la fiabilité et un ordonnancement partiel des messages vers tous les membres du domaine. Cet ordonnancement partiel ne correspond pas forcement à l'ordonnancement " réel " de l'utilisateur.

C'est la couche Transis qui fournit les ordonnancements, total, causal et le non-ordonnancement. Il est à noter que Transis fournit également un ordonnancement dit « sûr » : un nouveau message n'est expédié qu'après que tous les processus du groupe aient confirmé la réception du message précédent. Les algorithmes d'ordonnancement et de gestion de groupes implémentés dans Transis utilisent un mécanisme

d'anneau à jeton logique. Grâce à ce système, il est possible de borner le temps maximal de toutes les opérations.

[AAD94] présente les aptitudes de Transis à réaliser des applications distribuées tolérantes aux pannes grâce à la réplication.

Appartenance aux groupes	sémantique dynamique de gestion de groupe.
Structure de groupe	les groupes sont des groupes ouverts.
Ordonnancement des messages	ordonnancement total, causal, sûr et non-ordonnancement.
Sémantique de réception et de réponse	La sémantique de réception des messages est atomique et la sémantique de réponse est totale.
Fiabilité des transmissions	mécanisme (uniquement) fiable.

TAB. 3.5 – *Caractéristiques de Transis*

3.6 JavaGroups

JavaGroups [Unib] est une bibliothèque Java qui fournit une communication de groupe fiable. Cette bibliothèque a été développée par l'université de Cornell. La gestion dynamique des groupes se fait par canal. Un canal représente un groupe. Quand un élément désire joindre un groupe, il cherche à accéder à un canal en spécifiant un nom. Si ce canal n'existait pas encore, un nouveau groupe est créé, si le canal existait déjà l'élément est ajouté au groupe correspondant au canal. Les membres d'un canal envoient des messages vers tous membres et reçoivent les messages uniquement des membres du canal. Un canal peut être réutiliser : si tous les membres l'ont quitté, le canal est détruit, puis il peut être recréer par reconnexion de membres. Un canal connaît tous ses membres. Une adresse unique est affectée à chaque canal.

JavaGroups n'implémente pas les moyens de communication. Une classe abstraite Channel (Canal) est définie, laissant l'implémentation aux classes qui en dérivent. Pour le moment JavaGroups fournit trois implémentations de canal.

- **JChannel** qui est l'implémentation par défaut.
- **EnsChannel** basé sur Ensemble, une bibliothèque de communication fiable écrit en ML.
- **IbusChannel** basé sur iBus qui émule un bus logiciel.

L'application n'utilise que la classe abstraite ce qui rend les méthodes de communication totalement invisible au programmeur. Il est impossible de changer d'implémentation en cours d'exécution.

Appartenance aux groupes	sémantique dynamique de gestion de groupe.
Structure de groupe	les groupes sont des groupes fermés.
Ordonnancement des messages	ordonnancement total et non-ordonnancement.
Sémantique de réception et de réponse	La sémantique de réception des messages est atomique et la sémantique de réponse est totale.
Fiabilité des transmissions	Le mécanisme de communication est (uniquement) fiable pour le moment mais une nouvelle implémentation pourrait fournir un mécanisme non-fiable.

TAB. 3.6 – *Caractéristiques de JavaGroups*

Ces quelques systèmes illustrent les principales orientations de la communication de groupes. Il en existe beaucoup d'autres, que l'on peut apparenter plus ou moins à l'un ou l'autre des systèmes présentés.

Chapitre 4

Présentation de ProActive PDC

ProActive PDC (Parallèle Distribué et Concurrent) est une bibliothèque Java, conçue pour la programmation parallèle, distribuée et concurrente, ainsi que pour la métaprogrammation [OAS01].

Dans le contexte d'un modèle MIMD (Multiples Instructions, Multiples Données) à objets actifs, et à partir d'un ensemble restreint de primitives, une bibliothèque simple et flexible est définie pour le parallélisme, la distribution et la programmation concurrente. Il s'agit d'un middleware[Vay97].

ProActive n'ajoute aucune extension syntaxique à Java. Les programmeurs écrivent du code standard. La bibliothèque est elle-même extensible par les programmeurs, elle est un système ouvert aux optimisations et modifications.

ProActive est uniquement constituée de classes Java standards, et ne requiert aucune modification de la Machine Virtuel Java, ni du compilateur, ni de rajout de preprocessing.

ProActive PDC est basée sur la bibliothèque Java RMI standard.

ProActive fournit:

- des Objets Actifs
- des appels de méthodes asynchrones avec futurs transparents
- une communication par messages (Request and Reply)
- des agents mobiles (migration faible)

La sémantique de ProActive est séquentielle, multi-threadée, et distribuée. Le polymorphisme entre objet standard et objet actif objets permet une meilleure réutilisation du code. Le mécanisme de synchronisation automatique par future (avec attente par nécessité) facilite la conception d'application collaborative. ProActive est interfacé avec RMI registry, bien sûr, mais aussi avec Jini et Globus.

La transformation d'un programme séquentiel en un programme à objets actifs est décrite dans [DC98]. En voici un exemple :

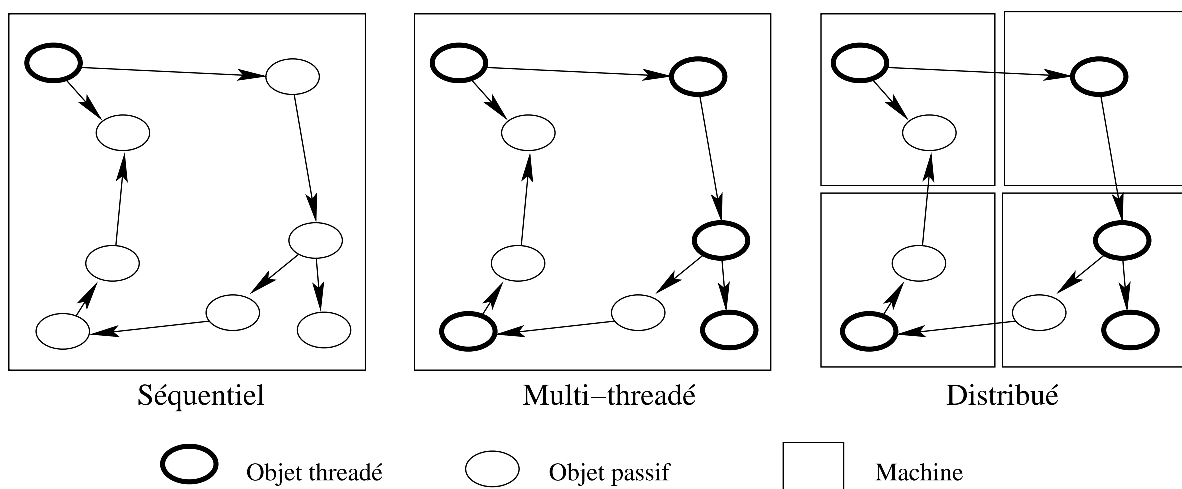


Fig. 4.1 – *Distribution par objets actifs*

4.1 Objets Actifs

Chaque activité est associée à un Objet Actif. Une activité est un objet actif. Par activité, on entend fil d'exécution ou thread. Dans un modèle uniforme (ou homogène) tous les objets sont actifs. Cette approche étant souvent « trop lourde », on recourt à un modèle non-uniforme dans lequel seuls certains objets sont actifs. C'est le choix fait dans ProActive.

Tous les processus sont des objets mais tous les objets ne sont pas des processus.

4.2 Synchronisation

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 vraiment utile que plus tard. Pour éviter cela ProActive utilise des mécanismes de futurs et d'attente par nécessité qui permettent des communications

- **synchrones :** l'appel est bloquant, on attend l'arrivée du résultat de la méthode invoquée avant de reprendre le fil d'exécution.
- **asynchrones :** l'appel est non bloquant, on continue l'exécution du programme sans que le résultat soit revenu. Un futur est créé.
- **à sens unique :** L'appel est non bloquant et aucun résultat n'est attendu.

Cela permet une programmation plus efficace. Ces caractéristiques de synchronisation sont adaptées à chaque méthode d'un objet actif en fonction principalement de son type de retour. Sauf configuration explicite de l'utilisateur, une méthode ne renvoyant aucun résultat (`void`) sera à sens unique, une méthode renvoyant des objets non-réifiables [1] sera appelée de façon synchrone, une méthode renvoyant des objets réifiables sera appelée de façon asynchrone.

4.2.1 Futurs

Un futur représente le résultat d'un appel de méthode qui n'est pas encore arrivé. Pour créer son asynchronisme, lors d'un appel de méthode (via RMI), ProActive construit et renvoie immédiatement un futur. Pendant ce temps, la requête RMI est déléguée à une autre thread. Lorsque la requête RMI est terminée avec succès, le résultat obtenu est « placé » dans le futur. Le futur implémente la même interface que l'objet résultat.

4.2.2 Attentes par nécessité

Que se passe-t-il si le futur est utilisé (lu, modifié, appel de méthode) alors que sa valeur, le résultat de l'appel RMI, n'est pas encore arrivée ? Dans ce cas, ProActive utilise l'attente par nécessité. Le fil d'exécution est suspendu jusqu'à ce que le résultat parvienne au client.

4.3 Communication par messages

ProActive fournit un niveau d'abstraction qui représente les communications comme des envois de messages. Un objet client envoie un message *Requête* vers un objet actif serveur. L'objet actif retourne un message *Réponse*. Un objet actif peut traiter les requêtes selon un ordre FIFO, LIFO ou une politique de service spécifique créée par l'utilisateur.

Un message Requête contient des informations sur l'expéditeur et sur la méthode invoquée (nom, paramètres, type du retour : un objet qui représente l'appel de méthode). Grâce à ces informations (et à la connexion RMI) le message Réponse retrouve le client et lui livre le résultat.

1. voir section 4.5 : MOP

4.4 Mobilité

ProActive offre une migration faible des objets actifs. Les objets actifs de ProActive possède une file d'attente des requêtes. C'est sur cette file d'attente que l'on applique les politiques de service FIFO ou LIFO. Dans une migration faible, on arrête de servir les requêtes de la file d'attente entre deux requêtes, à ce moment la pile d'exécution est vide, on peut sérialiser les données de l'objet actif et le faire migrer sans perte d'informations.

4.5 MOP : Meta Object Protocol

Pour représenter localement un objet distant le MOP crée un couple stub+proxy sur la machine virtuelle locale.

Le stub implémente la même interface de l'objet distant. Le stub est généré puis compilé à l'exécution du programme. Le stub créé par le MOP est en fait une classe héritée de la classe de l'objet. Les méthodes sont surchargées pour que les appels de méthode soient réifiés c'est à dire transformés en objet `MethodCall`.

Le stub sert également à représenter un futur.

Malheureusement un stub ne pas toujours être créé. Cela dépend de l'objet que l'on considère. Si à partir d'un objet on peut générer un stub, cet objet est dit réifiable[2]. Deux cas posent problème de non-réification :

- La classe de l'objet est `final`, on ne peut pas la sous-classer.
- L'objet n'est pas un objet mais un type primitif (`int`, `long`, `char`, ...) et ne peut donc pas être sous-classé.

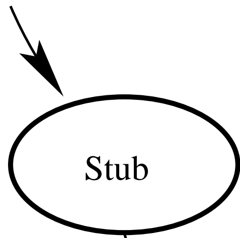
Lorsqu'il y a non-réification, le stub ne peut pas être créé. Si ce stub devait représenter un futur cela implique que l'appel de méthode ne peut être asynchrone. De même, seuls les objets réifiables peuvent devenir actifs.

Le proxy est l'élément qui relie le stub à l'objet distant. Le proxy implémente la sémantique de communication, il produit l'asynchronisme en créant les futurs.

La figure 4.2 montre la structure d'un objet actif et le rôle de chaque élément qui le compose lors d'un appel de méthode.

2. Un abus de langage est commis, un objet **est** une réification. Cependant le terme réifiable est employé dans ce rapport pour désigner un objet à partir duquel il est possible de fabriquer un stub

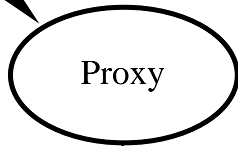
référence détenue par l'objet appelant



Le stub est le répresantant local de l'objet actif.
Il hérite de l'objet actif.

Le stub réifie l'appel de méthode, c'est à dire le transforme en objet MethodCall.

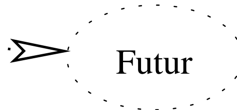
`MethodCall :`
* *nom de la méthode*
* *paramètres*
* *type du résultat*
* *...*



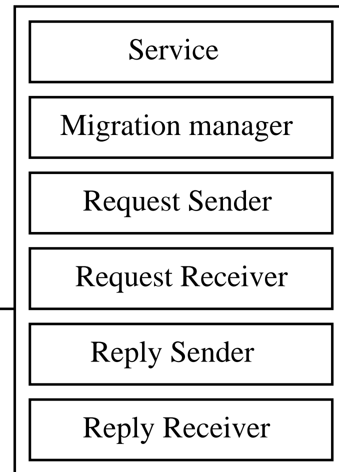
Le proxy est chargé de transmettre l'appel de méthode réifiée à l'objet actif. L'objet actif (et le Body) peuvent se trouver sur la même JVM ou sur une autre JVM acessible par le réseau.

Le proxy implémente la sémantique de communication, et crée l'asynchronisme en construisant des futurs.

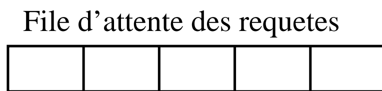
création

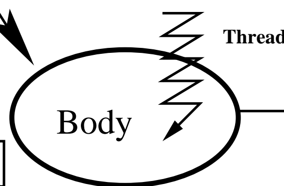


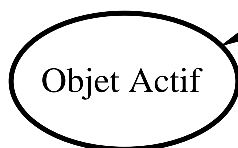
Réseau



File d'attente des requetes







Le Body reçoit la requête (Request Receiver), la place dans la file d'attente et la sert, selon une politique de service (Service).
Une fois le résultat obtenu, il est renvoyé (Reply Sender) à l'objet actif qui a provoqué l'appel.

L'objet actif appellant a utilisé le service Request Sender pour expedier sa requête et Reply Receiver pour recevoir le résultat.

FIG. 4.2 – *Objet actif*

Chapitre 5

Réflexion sur la communication de groupes dans ProActive

ProActive se base sur les communications RMI de Java, fiables et par rendez-vous. Toute communication RMI se fait grâce à la classe `UnicastRemoteObject`. La classe `MulticastRemoteObject` n'existe pas. Un ensemble de primitives multicast fournies par la machine virtuelle Java aurait pu simplifier la technique de diffusion des messages.

IP multicast ne convient pas, car les mécanismes de ProActive font abstraction des couches matérielles. La communication de groupes va donc être implémentée entièrement au niveau logiciel.

5.1 Buts

Les objectifs fixés imposent quelques obligations de syntaxe et de sémantiques. La manipulation de groupes doit se faire de manière transparente. Comme Java, elle doit rester fortement typée. Un objet de classe `A` et un groupe d'objets de classe `A` doivent être utilisé de la même façon. Ainsi si la classe `A` implémente la méthode `foo()`, un groupe d'objets `A` nommé `gA` peut appeler la méthode par un simple `gA.foo()` comme le ferait un simple objet de la classe `A`. Les groupes doivent être des groupes d'objets actifs.

Les groupes doivent proposer une gestion de groupes dynamique. Au cours de l'exécution du programme de nouveaux groupes peuvent être créés, de nouveaux membres ajoutés ou des membres retirés du groupe.

De plus le sujet impose également de fournir une sémantique de synchronisation suffisamment complète. Une extension des mécanismes d'attente et de synchronisation est nécessaire. On souhaite conserver les types de communication de ProActive (synchrone, asynchrone, à sens unique). La répercution à l'échelle du groupe des différences entre communications synchrones et asynchrones sont présenté dans [Cri96].

ProActive réclame une communication de groupes pour deux principales utilisations:

- **La duplication**
- **La structuration d'applications réparties** à large échelle

5.2 La syntaxe

Il y a deux principaux avantages à fournir cette syntaxe pour la communication de groupes. Le premier c'est qu'elle est extrêmement simple d'utilisation, on manipule un groupe, comme on manipule un simple objet. Le second avantage est la facilité avec laquelle on peut transformer une application « standard » (sans groupe) en une application utilisant les groupes. Les seules modifications à apporter concernent la création des variables : il faut créer un groupe d'objet au lieu de créer un simple objet. Cela peut être particulièrement intéressant pour les applications réclamant une tolérance aux pannes. En dupliquant les données dans un système distribué, on diminue le risque de les perdre à cause de la panne d'un des composants du système.

Pour que l'on puisse manipuler un groupe comme un simple objet, il faut que le groupe soit « encapsulé » derrière un stub qui implémente la même interface que l'objet. Les groupes seront donc des **groupes typés d'objets polymorphes**. ProActive nous fournit des stubs représentant des objets actifs

distants. Les stubs sont construits à l'exécution, dynamiquement, par ProActive. La communication de groupes va donc s'en servir afin d'obtenir la syntaxe souhaitée.

Cependant cette syntaxe n'est pas suffisante pour pouvoir créer, joindre ou quitter dynamiquement un groupe. Effectivement, un stub hérite de la classe de l'objet actif et n'implémente pas les méthodes `add`, `remove`, `createGroup`, ...

Deux possibilités s'offrent pour résoudre ce problème :

- **modifier le MOP** de ProActive, pour que, lorsqu'un stub est généré à partir d'une classe, une interface de communication de groupes soit rajoutée en plus de l'interface de la classe.
- **ajouter une deuxième syntaxe** dédiée à la manipulation de groupe, la première syntaxe servant uniquement à l'appel de méthode.

Le choix s'est porté sur la deuxième solution. Si le MOP se charge de rajouter l'interface de communication de groupes, celle-ci sera présente dans chaque stub généré, même s'il n'est pas destiné à servir à la communication de groupes : ceci pour la simple raison, qu'il est impossible de prédire si le stub que l'on génère va être utiliser pour représenter un groupe ou non. Cette méthode n'est pas convenable. La deuxième solution, quant à elle, n'a pas d'inconvénient majeur, mais nécessite de conserver une cohérence forte entre les deux représentations d'un même groupe qui évoluent parallèlement.

On définit la première syntaxe comme étant celle qui manipule des *objets groupés* et la seconde comme celle qui manipule un *groupe d'objets*.

Les *objets groupés* fournissent l'interface des objets.
Les *groupes d'objets* fournissent l'interface des groupes.

Les *objets groupés* et *groupes d'objets* sont deux représentations différentes d'un même groupe:

Les *objets groupés* sont polymorphiqument compatibles avec les objets de la classe du groupe, voir exemple 5.2.1.

Exemple 5.2.1 Création d'un nouveau groupe d'objet de classe `A` sous forme d'*objet groupé*

```
A a;

a = (A) ProActiveGroup.newActiveGroup("A");
```

Les *groupes d'objets* ne sont pas compatibles avec les membres du groupe, ils sont des instances d'une classe implémentant l'interface `Group` (voir section 5.3). Il est impossible en Java de créer une instance d'une interface, c'est pourquoi l'exemple 5.2.2 présente uniquement la déclaration d'un *groupe d'objet*.

Exemple 5.2.2 Déclaration d'un nouveau groupe sous forme de *groupe d'objets*

```
Group ga;
```

Il est à noter que lors de la déclaration d'un *objet groupé* le type du groupe est spécifié, alors qu'il ne l'est pas dans la déclaration d'un *groupe d'objets*.

Des méthodes doivent permettre de passer de l'une à l'autre de ces deux représentations. Les *objets groupés*, à travers une méthode statique `getGroup` de ProActive fournissent leurs groupes correspondants. Les *groupes d'objets* doivent implémenter une méthode `getGroupByType` qui retourne un objet représentant le groupe : un *objet groupé* (voir figure 5.1).

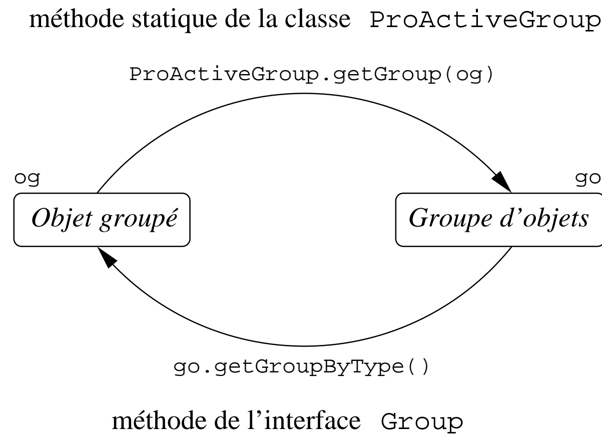


FIG. 5.1 – objets groupés *et* groupe d'objet

Exemple 5.2.3 transistions entre *groupe d'objets* et *objet groupé*

```
A a, a_bis;
Group ga;
...
a = (A) ProActiveGroup.newActiveGroup("A");

ga = ProActiveGroup.getGroup(a);

a_bis = (A) ga.getGroupByType();
```

L'exemple 5.2.3 présente des transitions entre *objets groupés* et *groupe d'objet.*

La cohérence est maintenue entre les deux représentations d'un groupe. Les actions effectuées sous une forme affectent également l'autre représentation du groupe.

5.3 L'interface `Group`

Comme dans la bibliothèque JavaGroups, une interface Group est définie dans le but de pouvoir éventuellement changer l'implémentation de la gestion de groupe. Cette interface est volontairement très sommaire pour minimiser le nombre de méthodes à re-définir en cas de nouvelle implémentation. L'interface `Group` définit les méthodes essentielles à la gestion d'un groupe (plus la méthode getGroupByType précédemment évoquée) :

- `add (élément)` *ajoute l'élément au groupe*
- `remove (élément)` *retire l'élément du groupe*
- `size ()` *renvoie le nombre d'élément dans le groupe*
- `iterator ()` *renvoie un itérateur sur le groupe*
- `getGroupByType ()` *renvoie un objet groupé représentant le groupe*

5.4 Le proxy de groupe

ProActive ne contient aucun service centralisé, pour ne pas déroger à ce principe, la communication de groupes se fera sans service spécifique.

Le groupe n'est pas implémenté au niveau du stub. Le proxy est en contact avec l'objet actif à travers le réseau. Le mécanisme de communication de groupes doit donc se situer entre le stub et le proxy. Un nouvel élément, **le proxy de groupe** est introduit dans la librairie ProActive.

Le proxy de groupe contient la liste de tous les éléments du groupe. Lors d'un appel de méthode, il transmet l'appel de méthode réifié par le stub à tous ses membres. C'est ainsi que l'on obtient une communication de 1 vers N. La communication de groupes garde les propriétés de la communication vers un seul objet. Si l'appel est asynchrone, les N appels sont asynchrones, si l'appel est synchrone, les N appels sont synchrones, si l'appel est à sens unique, les N appels sont à sens uniques.

5.5 Groupe de futurs

Un appel de méthode sur un objet actif crée un futur . Il est donc possible qu'un appel de méthode sur un groupe d'objets actifs (un *objet groupé*) crée un groupe de futurs. Un groupe de futurs doit posséder les mêmes propriétés qu'un groupe d'objets actifs. C'est à dire avoir deux représentations du groupe, une sous forme d'*objet groupé* et l'autre sous forme de *groupe d'objet*, et pouvoir passer de l'une à l'autre par les méthodes `getGroup` et `getGroupByType`. Les groupes de futurs doivent conserver la même structure que le groupe d'objets actifs dont ils sont résultats. Il est expliqué dans la section 6.2 pourquoi et comment la communication de groupes de ProActive fournit la possibilité de créer des groupes hiérarchiques, mais ce qu'il est important de noter ici c'est que : *Le groupe résultat d'un groupe hiérarchique est aussi un groupe hiérarchique ayant la même forme.*

5.6 Synchronisation

Pour maîtriser les appels de méthode synchrones et asynchrones sur des groupes, de nouveaux mécanismes de synchronisation sont indispensables. Il faut être capable de savoir si aucun, une partie ou tous les résultats sont revenus. Pour cela on enrichie ProActive de deux prédicats :

- `allAwaited(`*objet groupé*`)` *renvoie vrai si aucun résultat n'est revenu*
- `allArrived(`*objet groupé*`)` *renvoie vrai si tous les résultats sont revenus*

En combinant ces deux prédicats, on peut obtenir celui qui manque : si AllAwaited renvoie faux et AllArrived renvoie faux alors seule une partie des résultats est revenue, c'est le troisième prédicat.

En plus des prédicats qui testent l'état du groupe résultat, des mécanismes d'attente sont nécessaires pour permettre au programmeur de maîtriser la synchronisation de son application. Deux attentes sont particulièrement utiles : l'attente du retour d'au moins un résultat et l'attente du retour de tous les résultats.

- `waitForOne(`*objet groupé*`)` *reprends le fil d'exécution lorsqu'un résultat est revenu*
- `waitForAll(`*objet groupé*`)` *reprends le fil d'exécution lorsque tous les résultats sont revenus*

Il est à noter que le mécanisme d'attente par nécessité est toujours valide, il n'agit pas au niveau du proxy de groupe mais au niveau de chaque élément du groupe. Son effet reste le même qu'il s'agisse d'un simple oject actif ou d'un groupe.

Les méthodes `kAwaited(`*entier*`)` et `waitForK(`*entier*`)` n'ont pas été implémentées dans la première version de la communication de groupes de ProActive mais le seront dans la prochaine.

Chapitre 6

Implémentation

6.1 Architecture

La communication de groupe est ajoutée dans ProActive sous forme de package. Pour l'utiliser il suffit d'importer le package :

```
import fr.inria.proactive.group;
```

De la même façon que dans la bibliothèque JavaGroups la notion de groupe est définie par une interface. C'est dans le proxy de groupe qu'il est le plus naturel d'implanter les mécanismes de gestion des groupes. Le proxy de groupe possède déjà la liste des membres pour relayer les appels de méthode. Il aura donc en plus de ce rôle, le rôle de gestionnaire de groupes.

Si le proxy de groupe contient une `Collection` de stubs pointant vers des objets actifs, l'objet `MethodCall` représentant un appel de méthode, doit être dé-réifié pour le stub et re-réifié par celui-ci à l'attention du proxy qui transmettra enfin la requête à travers le réseau. Pour ne pas briser la chaîne de réification de l'appel de méthode, c'est à dire éviter une dé-réification et une re-réification, la liste de membres du proxy de groupe sera composé de proxys. Les stubs ne sont pas nécessaires (et même superflus) dans le mécanisme de communication de groupes. Ne pas inclure le stub dans la liste ne pose pas de problème : un nouveau stub peut toujours être généré par la méthode `getStubForBody` prenant un body (accessible via le proxy) en paramètre.

Toute classe implémentant l'interface `Proxy`, doit définir la méthode `reify`. Cette méthode prend la réification d'un appel de méthode (un objet `MethodCall`) en paramètre et renvoie un `Object`. Cet objet est un stub représentant un futur. Lors d'un appel de méthode sur un groupe, la méthode `reify` de la classe `ProxyForGroup` construit un nouveau groupe. Ensuite ce nouveau groupe est rempli des futurs de chaque membre, en appelant la méthode `reify` sur chaque proxy du groupe d'origine.

La classe ProActiveGroup contient les méthodes statiques d'attentes : les prédicats `AllWaited` et `AllArrived` ainsi que les méthodes `WaitForOne` et `WaitForAll`.

C'est par la méthode newActiveGroup qu'un nouvel objet groupé est construit. Appelée sans paramètres, cette méthode crée un objet groupé représentant un groupe vide. Cette méthode est surchargée pour créer des groupes d'objets identiques ou non, sur la même machine virtuelle ou non en fonction de certains paramètres.

La figure 6.1 représente la structure d'un groupe.

6.2 Groupes hiérarchiques

Le proxy de groupe contient une liste de proxys. Le proxy de groupe peut donc contenir des proxys de groupe. C'est par ce principe que l'on obtient des groupes hiérarchiques. L'intérêt principal des groupes hierarchiques est qu'ils permettent facilement la mise à grande échelle de l'application. Ils sont un moyen efficace de structurer une application; en assignant un sous-groupe à un sous-réseau par exemple.

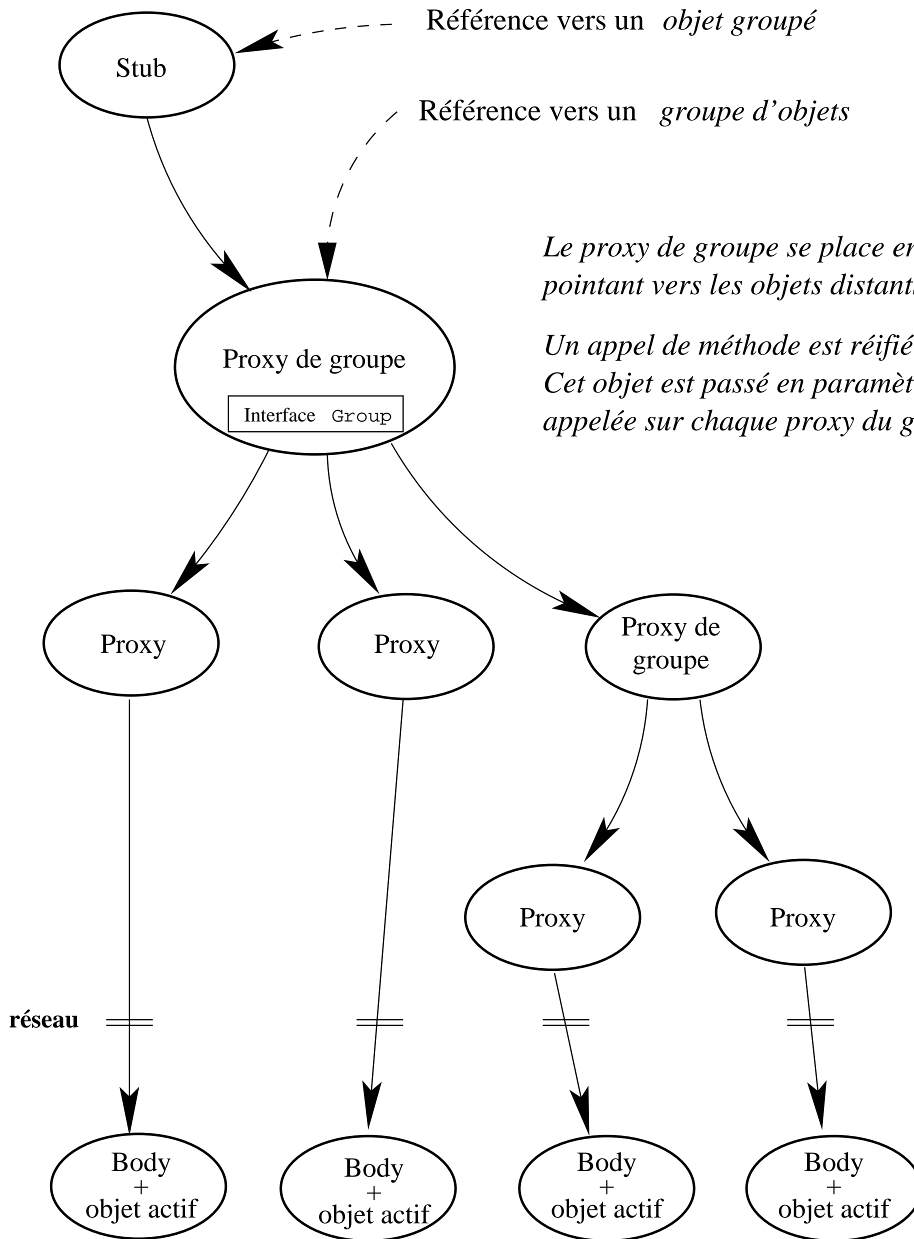


FIG. 6.1 – *Structure d'un groupe*

6.3 Limitations

6.3.1 Goupes d'objets non-réifiables

Il est impossible de créer un groupe sous forme d'*objet groupé* avec des objets non-réifiables. Ceci est du au fait que l'*objet groupé* que l'on manipule est en réalité un stub. Si la classe de l'objet avec laquelle on veut créer un *objet groupé* ne permet pas la réification, le stub ne peut pas être créer. Ceci pose problème lors d'un appel de méthode sur un groupe qui retourne un objet non-réifiable. Dans la version standard de ProActive, cette restriction est détournée en utilisant un appel synchrone et en renvoyant directement le résultat (et non pas un futur). Avec la communication de groupes le problème est plus important, puisque le rôle du stub+proxy ne se limite plus à représenter un futur, mais aussi un groupe :

Une méthode qui retourne un objet non-réifiable ne peut pas être invoquée sur un groupe.

6.3.2 Référence distante et perte de la cohérence

Un groupe ne peut pas être directement référencé à distance. C'est une représentation exclusivement locale. Pour obtenir localement un groupe distant, ProActive produit automatiquement une copie du groupe. Cela pose des problèmes de consistance des données. Si un élément est ajouté ou supprimé au groupe localement, les modifications ne seront effectives QUE localement. Le groupe distant ne sera pas changé. Pour limiter ce problème, une méthode `freeze` est créée. Cette méthode permet de « figer » le groupe dans son état courant. Une fois cette méthode invoquée sur le groupe, celui-ci ne pourra plus accueillir de nouveaux éléments, ni perdre de membres. Un appel à `freeze` est irrémédiable. Le maintient de la consistance est obtenu au prix de la dynamicité.

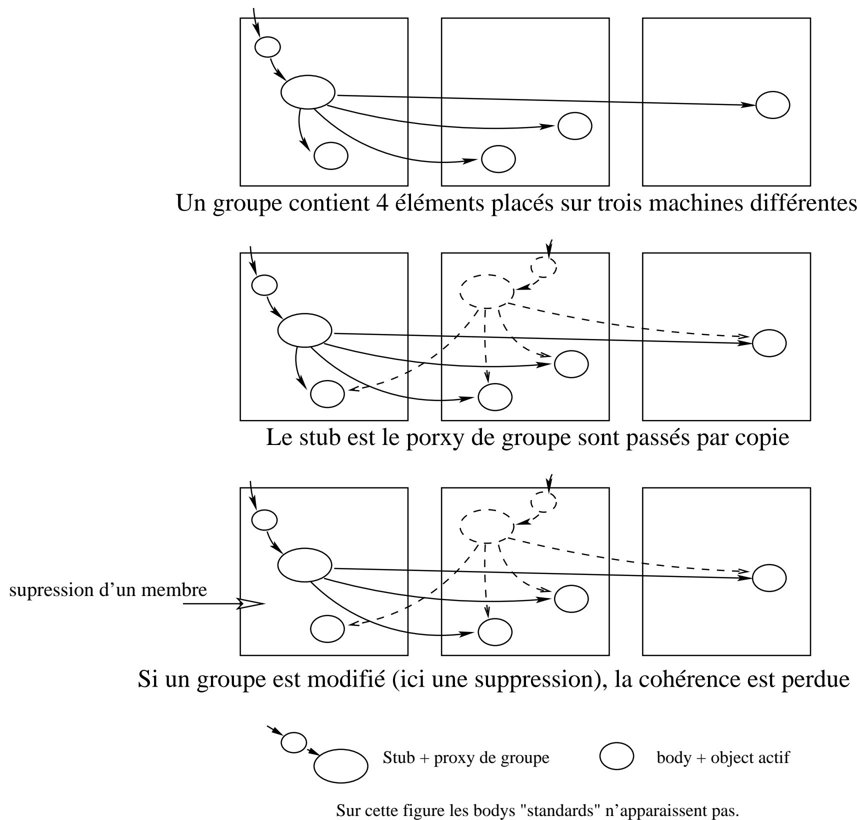


FIG. 6.2 – *Perte de la cohérence*

6.3.3 Tolérance aux pannes

La communication de groupes de ProActive ne supporte pas encore les pannes. Si un membre d'un groupe devient inaccessible par partitionnement du réseau ou panne de la machine, l'appel de méthode effectué sur ce membre sera bloquant. Les mécanismes RMI et TCP ne détectent pas la panne avant plusieurs minutes, laissant le système immobilisé avant de lever une exception.

6.4 Améliorations déjà effectuées

6.4.1 Le `TransparentProxy`

Le modèle de la communication de groupes de ProActive est suffisament ouvert pour proposer des groupes d'objets non-actifs. Avec un nouveau proxy : le `TransparentProxy` il est maintenant possible d'ajouter des objets Java standards dans les groupes. Les groupes ne sont plus réservés uniquement aux objets actifs mais à tous les objets Java réifiables par le MOP de ProActive. Le rôle d'un `TransparentProxy` est de dé-réifier l'appel de méthode `MethodCall` pour l'exécuter sur un objet Java. Les groupes peuvent être mixtes, c'est à dire contenir à la fois des objets actifs et des objets standards. Comme pour les objets actifs, la seule restriction à l'ajout d'un objet standard dans un groupe est la compatibilité de sa classe avec celle des éléments du groupe : les groupes sont des groupes typés d'objets polymorphes.

6.4.2 La méthode `optimize`

Lorsqu'un appel de méthode asynchrone retourne un objet actif comme résultat, deux proxys sont construits. D'abord un proxy pour futur et ensuite, à l'arrivée résultat, un proxy « standard » (`ProxyForBody`) pointant vers l'objet actif. Une fois le second proxy construit, le premier ne sert plus à rien, cependant il reste là car aucune fonction de ProActive ne le supprime.

Ce problème est encore plus gênant dans la communication de groupes car ce n'est pas un objet proxy qui persiste malgré son inutilité mais autant que de membre dans le groupe. Pour palier à ce problème une méthode `optimize` permet de supprimer les proxys de trop. Cela supprime une référence lors des appels de méthodes sur les objets actifs et libère des ressources mémoire.

Parallèlement, la méthode `optimize` re-alloue l'espace mémoire pour être à l'exacte dimension des membres du groupe.

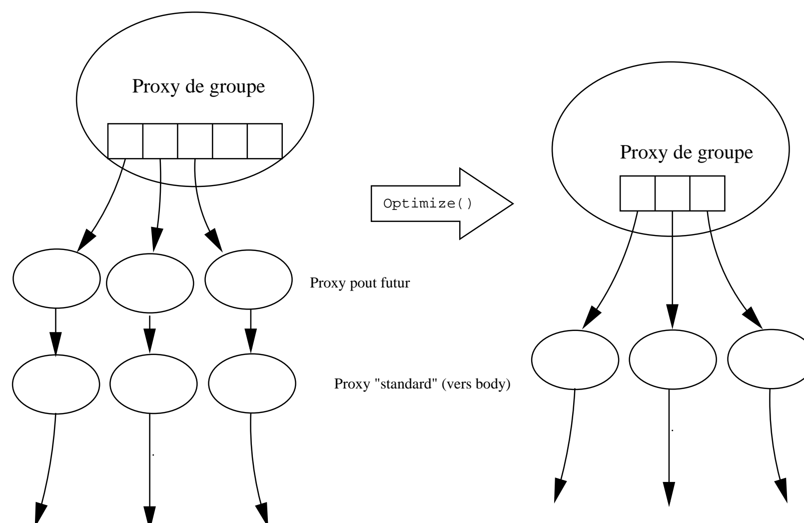


FIG. 6.3 – *La méthode `optimize`*

6.5 Caractéristiques de la communication de groupes de ProActive

Voici les caractéristiques de la communication de groupes de ProActive :

Appartenance aux groupes	ProActive utilise deux sémantiques l'une pour la gestion dynamique de gestion de groupe (créer, joindre, quitter, détruire) : les *groupes d'objets*. L'autre pour les appels de méthodes : les *objets groupés*.
Structure de groupe	les groupes sont des groupes ouverts.
Ordonnancement des messages	ordonnancement total.
Sémantique de réception et de réponse	la sémantique de réception des messages est atomique et la sémantique de réponse est totale, partielle ou aucune (les appels à sens unique).
Fiabilité des transmissions	mécanisme de communication uniquement fiable.

TAB. 6.1 – *Caractéristiques de JavaGroups*

6.6 Performances

L'efficacité de la communication de groupes est assurée par RMI, qui conserve en cache les derniers objets sérialisés. De cette façon seule la première sérialisation des paramètres de la méthode appelée coûte. Le marchalling des paramètres est ainsi optimisé. L'appel de méthode sur tous les éléments du groupe est optimisé. De plus pour un groupe de N éléments, N-1 réifications d'appel de méthode sont évitées.

Les tests effectués représentent le temps exprimé en millisecondes nécessaire pour réaliser un certain nombre d'appel (en abscisse sur les graphes). Les courbes en pointillés se réfèrent à des appels de méthodes sur des objets actifs. Les courbes en trait continu se réfèrent à des appels de méthode sur des groupes d'objets actifs.

Dans ces tests, les groupes sont composés de dix objets actifs. Ainsi un appel de méthode sur un groupe est " équivalent " à dix appels de méthode.

La méthode invoquée est une méthode asynchrone qui crée et retourne un objet actif. Deux versions existent de cette méthode :

- L'une étant sans paramètre
- L'autre possédant plusieurs paramètres qui devront être sérialisés puis transmis à travers le réseau.

Plusieurs séries de tests ont été réalisées sur différents types de réseaux. Les membres du groupe ont été placés :

- Tout d'abord en local, sur la même machine, dans la même JVM que l'objet actif appelant.
- Ensuite une machine du même réseau local (LAN)
- Et pour finir sur un réseau étendu : entre l'INRIA et l'université de Nice (MAN)

Les machines utilisées sont de performances équivalentes (processeur Pentium III). Le kit de développement Java est le JDK 1.3 de IBM.

Les graphes sont présenté dans la figure 6.4.

Les courbes montrent une économie de temps grâce à l'utilisation de la communication de groupes. Le gain de rapidité est plus important lors d'appel de méthode sans paramètre. Ceci est du au fait que la sérialisation est un mécanisme lent et que le gain de performance du à la communication de groupe est

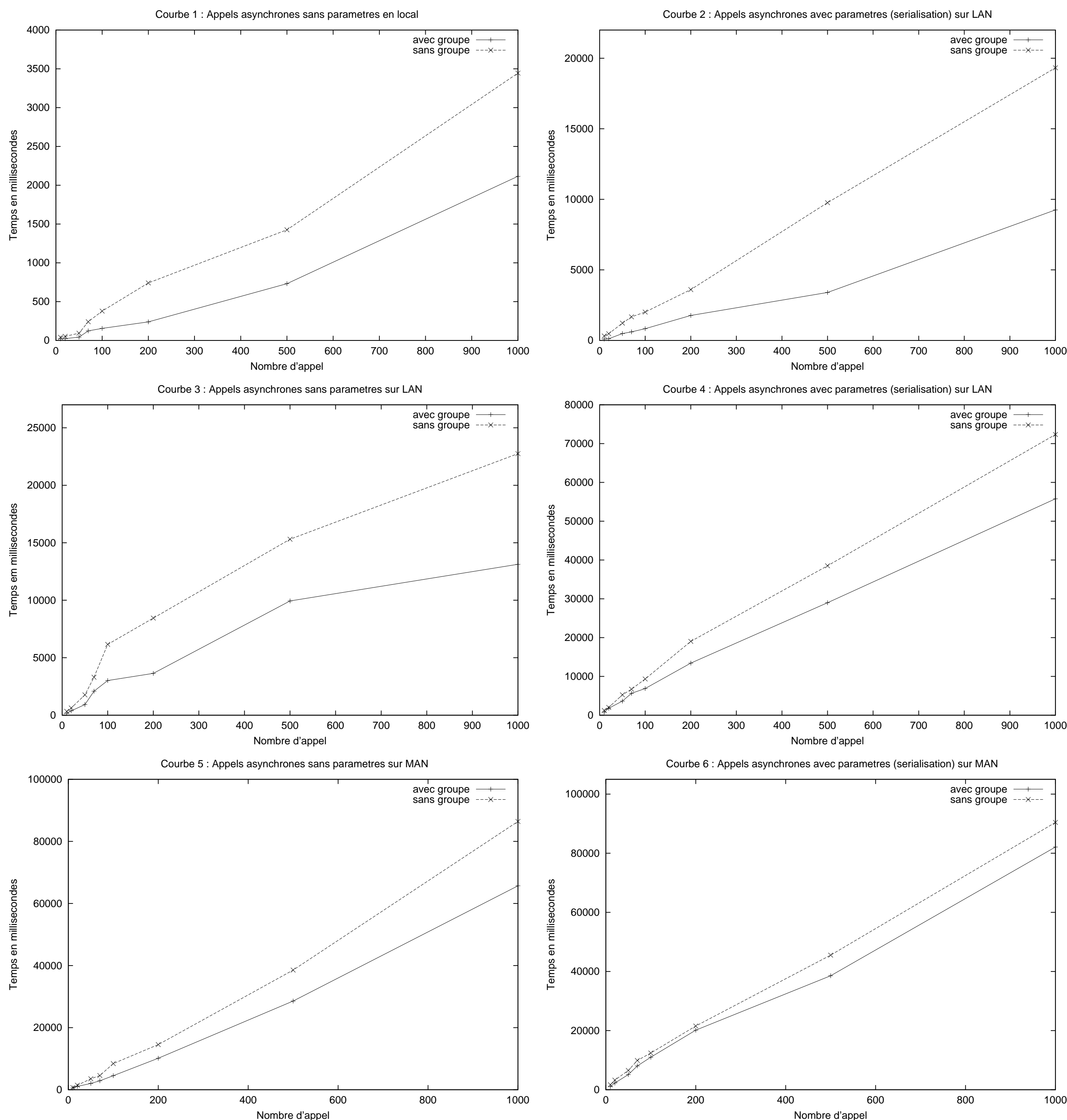


FIG. 6.4 – *Tests de performances*

« noyé » dans les latences induites par la sérialisation. De la même manière, les délais de communication longs (MAN) réduisent l'efficacité des groupes. On en déduit l'utilité de construire un nouveau mécanisme d'appel de méthode sur les groupes d'objets actifs (voir section 6.6.1).

L'amélioration des performances due à RMI n'est pas observable dans ces tests. Les appels sont réalisés les uns après les autres, aussi bien pour les groupes que pour les objects actifs. L'effet de cache est appliqué dans les deux cas.

D'autres tests sont encore à faire, tels que l'étude du maintient de la performance malgré une mise à grande échelle. L'observation de l'économie que produit un groupe par rapport à des appels " simples " en fonction du nombre d'élément dans le groupe. Des mesures précises sur l'optimisation réalisée par le cache RMI, ...

6.7 Extensions en cours

6.7.1 Un pool de threads

Dans la version actuelle de la communication de groupes de ProActive, la méthode `reify` est appelée tour à tour sur chaque membre du groupe. Quelque soit le type d'appel : synchrone, asynchrone ou à sens unique, cela n'est pas parfaitement efficace. Voici les deux cas :

- Lors d'un appel synchrone sur le groupe, la méthode est appelée de façon synchrone sur chaque membre du groupe. Pour chaque membre il faut attendre que l'appel synchrone soit terminé pour passer au membre suivant. Ce mécanisme est plus rapide que N appels synchrones sur des objets actifs « simples » (N-1 réifications d'appel de méthode sont évitées) mais il peut encore être amélioré.
- Lors d'un appel asynchrone ou à sens unique le problème est moindre. Dès que l'appel est réalisé, on passe à l'élément suivant dans la liste. Une latence peut apparaître si le temps de connexion avec les objets actifs distant est long.

Pour améliorer l'efficacité de la communication de groupe, il serait possible d'utiliser des threads. A chaque fois qu'une méthode est appelée sur un membre, on utilise une nouvelle thread qui réalise l'invocation de la méthode `reify`. Cela permet d'éviter le blocage durant les appels synchrones et de minimiser le temps passé sur chaque membre lors d'appels asynchrones.

Malheureusement, créer une nouvelle thread est coûteux en temps. De plus, selon ce modèle, la durée de vie la thread est courte et il faut sans cesse en créer de nouvelles. Pour que l'utilisation de threads soit intéressante, la solution est de construire au préalable un pool de threads. Ces threads restent inactives jusqu'à ce que l'on ait besoin d'elles. Au moment d'un appel de méthode sur un groupe les threads sont chargées d'appeler la méthode `reify` sur tous les membres du groupe. Ensuite, elles retrouvent leur état passif jusqu'au prochain appel de méthode.

Deux questions se posent au sujet de ce pool de thread. La première : « Comment le dimensionner ? Combien de threads sont-elles nécessaires en fonction du nombre d'élément dans le groupe ? » et la deuxième « A partir de quel délai d'établissement de connexion cette solution est-elle réellement avantageuse pour les appels de méthode asynchrones ? ».

6.7.2 Transformation du proxy de groupe en objet actif

Une seconde amélioration de la communication de groupes serait de transformer l'actuel proxy de groupe en objet actif. Les avantages, tous liés au statut d'objet actif, seraient nombreux.

L'avantage majeur serait la possibilité de définir des politiques de service sur les appels de méthode. L'utilisateur pourrait choisir de traiter les requêtes selon un ordre dépendant du critère de son choix.

Le proxy de groupe pourrait migrer d'une machine à une autre.

Ensuite, il serait possible d'avoir une référence distante vers un groupe. Pour le moment, un groupe est défini au sein d'une machine virtuelle et aucune référence venant d'une autre machine virtuelle n'est

possible.

6.7.3 La méthode `getFirstResult`

Il serait souhaitable de définir une méthode `getFirstResult` qui offrirait la possibilité à l'utilisateur de ne plus attendre un objet groupé en résultat à un appel de méthode sur un groupe mais bien le type de résultat renvoyé par la méthode. Dans cette optique on considère que le premier résultat à revenir au client est le résultat de l'appel de méthode sur le groupe. Cela serait utile dans la construction d'une application dont l'enjeu principal est la performance.

Chapitre 7

Conclusion

7.1 Apports

Durant ce stage un mécanisme de multicast a été développé. Grâce à ce mécanisme, la bibliothèque ProActive est enrichie d'une communication de groupes fiable et efficace. L'approche employée a permis l'intégration de cette nouvelle fonctionnalité sans modifier ni la sémantique de communication point à point, ni la synchronisation. Ces notions fortes de ProActive ont été étendues.

La principale caractéristique de la communication de groupes de ProActive est la capacité de manipuler un groupe d'objets comme un simple et unique objet. Ceci est possible grâce à la syntaxe *objets groupés*. Cette syntaxe permet de ne plus se soucier de la manipulation des groupes une fois ceux-ci formés et de se concentrer uniquement sur la conception de l'application.

ProActive propose également une deuxième syntaxe, *groupes d'objet*, qui fait apparaître un groupe d'objet en tant que groupe et non plus en tant que simple objet. Cette syntaxe, plus « classique » est destiné uniquement à la gestion dynamique des groupes.

Posséder deux syntaxes bien distinctes permet de séparer les comportements.

La communication de groupes de ProActive permet la création dynamique et automatique de groupes de futurs.

7.2 Perspectives

Les choix de design de la communication de groupes de ProActive, ont été effectués en voulant offrir le maximum de fonctionnalités à l'utilisateur final. La simplicité et la flexibilité de la syntaxe permettent une réalisation rapide et efficace d'applications distribuées et/ou collaboratives. Les mécanismes de synchronisations de ProActive ont été étendus à la communication de groupe. L'obtention de bonnes performances était également un challenge majeur.

Une implémentation de communication de groupes efficace pour ProActive a été réalisée mais de nombreuses extensions et optimisations sont encore possibles. Notamment la couverture du temps de connexion RMI par l'utilisation de threads. La garanti d'une seule sérialisation des paramètres lors d'une communication vers un groupe. L'utilisation de primitives multicast réseau intégrées dans Java serait aussi à étudier.

Sur un niveau plus conceptuel, la transformation du proxy de groupe en objet actif permettrait d'étendre encore bien plus les possibilités de la communication de groupes; celle-ci bénéficierait de toutes les capacités d'un objet actif.

Cette étude, longue et délicate, n'a malheureusement pas pu être réalisée durant ce stage.

Annexe A

Exemple d'utilisation

Supposons l'existence des classes `A` et `B` toutes deux réifiables au sens du MOP de ProActive. `A` implémente les méthodes `foo()` et `bar()` et `B` implémente la méthode `see()`.

```
public class A implements Active{
      ...
      void foo() {...}
      B bar () {...}
      ...
}
```

```
public class B {
      ...
      void see() {...}
      ...
}
```

Cette démonstration présente la manière d'utiliser la double syntaxe de groupe de ProActive. Toutes les fonctionnalités n'ont pas été exposé (suppresion, égalité, synchronisation, ...)

Voici maintenant ce que pourrait être un programme utilisant ces classes :

```
public static void main (String args[]) {

/* déclaration des variables sans distinguer les éventuels groupes des objets actifs */
A a,ga ;
B b,gb ;

/* création d'un objet actif A */
a = ProActive.newActive("A",Node) ;
/* appel de méthode sur l'objet actif */
a.foo() ;
b = a.bar() ;

/* Création d'un objet groupé de A */
/* éléments est un tableau d'éléments à inclure dans le groupe à sa création */
ga = ProActiveGroup.newActiveGroup("A",éléments[]);

/* Appels de méthode sur l'objet groupé */
ga.foo() ;

/* le resultat de l'appel produit un objet groupé de B */
gb = ga.bar() ;

/* On attend le retour de tous les membres de gb */
ProActiveGroup.waitForAll(gb) ;

/* Le groupe d'objets correspondant à l'objet groupé gb est créé */
Group bgroup = ProActiveGroup.getGroup(gb) ;

/* On ajoute un élément au group */
bgroup.add(b) ;


/* une méthode est appelé sur l'objet groupé gb */
gb.see() ;
/* un appel à getByType pour ré-obtenir un objet groupé à partir du groupe
   d'objet est inutile : la cohérence est automatiquement maintenue entre
   les deux représentations du groupe */
// cette appel serait de la forme :
//  gb = bgroup.getByType() ;

}
```

Annexe B

La Javadoc

Seules les trois classes les plus significatives sont présentées :

- L'interface `Group`
- La classe `ProActiveGroup`
- La classe `ProxyForGroup`

fr.inria.proactive.group

Interface Group

All Known Implementing Classes:
ProxyForGroup

public interface **Group**

Method Summary

void	**add**(Object o) add the Object o into the group,
void	**addMerge**(Object ogroup) add all the member of the group ogroup in to the current group
Object	**getGroupByType**() return an Object representing the group
Class	**getType**() return the Class of group's member
Iterator	**iterator**() return an Iterator for the group
void	**remove**(Object o) remove the Object o from the group
int	**size**() return the number of member in the group

---

fr.inria.proactive.group

Class ProActiveGroup

```
java.lang.Object
  |
  +--fr.inria.proactive.group.ProActiveGroup
```

---

public class **ProActiveGroup**
extends java.lang.Object

Method Summary

`static boolean`	**`allArrived`**`(Object o)` return true if all the member are arrived
`static boolean`	**`allAwaited`**`(Object o)` return true if all member of are awaited
`static Group`	**`getGroup`**`(Object o)` return the Group corresponding to the Object o representing a group
`static Object`	**`newActiveGroup`**`(String className)` create a new object representing an empty group
`static void`	**`waitForAll`**`(Object o)` suspend the current thread until all member of the group reprensenting by object arrive (Futures)
`static void`	**`waitForAllDeep`**`(Object o)` suspend the current thread until all member of the group and hierarchicals groups reprensenting by object arrive (Futures)
`static void`	**`waitForOne`**`(Object o)` suspend the current thread until one member of the group reprensenting by object o arrives

---

fr.inria.proactive.group

Class ProxyForGroup

```
java.lang.Object
  |
  +--fr.inria.proactive.mop.AbstractProxy
        |
        +--fr.inria.proactive.group.ProxyForGroup
```

All Implemented Interfaces:
Group, Proxy

---

public class **ProxyForGroup**
extends fr.inria.proactive.mop.AbstractProxy
implements fr.inria.proactive.mop.Proxy, Group

Method Summary	
void	**add**(Object o) add an element into the group
void	**addMerge**(Object oGroup) add all member of the group oGroup into the Group
boolean	**equals**(Group g) test the equality between the `Group` (`this`) and `g`
Object	**getGroupByType**() return an Object representing the group
Class	**getType**() return the `Class` of group's member
Iterator	**iterator**() return an `Iterator` for the group
void	**optimize**() simplify double proxy (proxyForFuture+proxyForBody) and trims the capacity of member list instance to be the list's current size. This operation is used to minimize the storage of the MemberList instance
Object	**reify**(MethodCall c) The proxy's method the method reify build a new group for the responses (futures) of the MethodCall c, then apply the method reify on all member of the group
void	**remove**(Object o) remove an element of the group
int	**size**() return the number of member in the group
String	**toString**() return a String describing the group (including hierarchical group)

Methods inherited from class AbstractProxy
getTarget, setTarget

Bibliographie

[AAD94] Ofir Amir, Yair Amir, and Danny Dolev. A highly available application in the transis environment. *Lecture Notes in Computer Science*, 774:125–??, 1994.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *FTCS-22: 22nd International Symposium on Fault Tolerant Computing*, pages 76–84, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[ADMSM94] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication, 1994.

[ANM01] H. E. Bak A. Nelisse, T. Kielmann and J. Maassen. Object-based collective communication in java. *ACM 2001 Java Grande / ISCOPE Conference*, 2001.

[AS98] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS 98-4, 1998.

[BDM98] O. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms, 1998.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. In *Communications of the ACM, vol 36*, pages 37–53, 1993.

[BS94] Ozalp Babaoglu and Andre Schiper. On group communication in large-scale distributed systems. In *ACM SIGOPS European Workshop*, pages 17–22, 1994.

[Bud97] R. Budhia. Performance engineering of group communication protocols, 1997.

[Cho97] G. Chockler. An implementation of reliable multicast services in the transis group communication system, 1997.

[Cri96] F. Cristian. Synchronous and asynchronous group communication, 1996.

[DC98] J. Vayssiere D. Caromel, W. Klauser. Towards seamless computing and metacomputing in java. In *in Concurrency Practice and Experience*, pages 1043–1061. Wiley and sons, 1998.

[DKM93] Danny Dolev, Shlomo Kramer, and Dalia Malki. Early Delivery Totally Ordered Multicast in Asynchronous Environments. In *The Twenty-Third International Symposium on Fault-Tolerant Computing*, 1993.

[EMS95] Paul D. Ezhilchelvan, Raimundo A. Macedo, and Santosh K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *International Conference on Distributed Computing Systems*, pages 296–306, 1995.

[FGG96] P. Felber, B. Garbinato, and R. Guerraoui. The design of a corba group communication service, 1996.

[FGS97] P. Felber, R. Guerraoui, and A. Schiper. The corba object group service, 1997.

[FGS98] Pascal Felber, Rachid Guerraoui, and Andre Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[Gol92] Richard A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, 1992.

[GS96] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Reliable Software Technologies - Ada-Europe'96*, pages 38–57. Springer-Verlag, 1996.

[Hav99] Stefan Havenstein. Object groups in transactional corba systems, 1999.

[KFL98] Roger Khazan, Alan Fekete, and Nancy A. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *International Symposium on Distributed Computing*, pages 258–272, 1998.

[Kha96] R. Khazan. Group communication as a base for a load-balancing, replicated data service, 1996.

[KT94] M. Frans Kaashoek and Andrew S. Tanenbaum. Efficient Reliable Group Communication for Distributed Systems, 1994.

[KT96] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the amoeba group communication system. In *International Conference on Distributed Computing Systems*, pages 436–448, 1996.

[KTV92] Frans M. Kaashoek, Andrew S. Tanenbaum, and Kees Verstoep. Group communication in amoeba and its applications. In *OpenForum*, pages 365–381, Utrecht (the Netherlands), 1992.

[Maf95] S. Maffeis. Adding group communication and faulttolerance to corba, 1995.

[MLBD93] L. Mathy, G. Leduc, O. Bonaventure, and A. Danthine. A group communication framework, 1993.

[OAS01] Equipe OASIS. Proactive pdc installation and user guide, 2001.

[oJ] Hebrew University of Jerusalem. Javagroups, a reliable multicast communication toolkit for java http://www.cs.huji.ac.il/labs/transis/.

[RBM96] R. Van Renesse, K. Birman, and S. Maffeis. Horus: A flexible group communication system, 1996.

[SM92] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Lisboa 92 - An Advanced Course on Distributed Systems*, 1992.

[TFC99] C. Toinard, Gerard Florin, and C. Carrez. A formal method to prove ordering properties of multicast systems. *Operating Systems Review*, 33(4):75–89, 1999.

[UN94] T. Urnes and R. Nejabi. Tools for implementing groupware: Survey and evaluation, 1994.

[Unia] Cornell University. The isis project http://www.cs.cornell.edu/info/projects/isis/isis.html.

[Unib] Cornell University. Javagroups, a reliable multicast communication toolkit for java http://www.cs.cornell.edu/info/projects/javagroupsnew/index.html.

[Unic] Vrije University. Amoeba www home page http://www.cs.vu.nl/pub/amoeba/.

[Vay97] Juilen Vayssière. Programmation parallèle et distribuée en java. Master's thesis, Supélec, 1997.

[VKCD99] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study, 1999.

[VR92] Paulo Veríssimo and Louís Rodrigues. Group Orientation: A Paradigm for Distributed Systems of the Nineties. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, 1992.

[vRHB94] R. van Renesse, T. Hickey, and K. Birman. Design and performance of horus: A lightweight group communications system, 1994.

[VsR92] P. Ver and s Rodrigues. Group orientation: a paradigm for modern distributed systems, 1992.

[VVR92] W. Vogels, P. Verissimo, and L. Rodrigues. Requirements for high performance group support in distributed systems, 1992.

[Wil95] Erik Wilde. Group management and communication support for collaborative applications, 1995.

[WMK94] Brian Whetten, Todd Montgomery, and Simon M. Kaplan. A high performance totally ordered multicast protocol. In *Dagstuhl Seminar on Distributed Systems*, pages 33–57, 1994.