Université de Nice-Sophia Antipolis Master 1 Informatique

Réseaux pair-à-pair structurés en ProActive

Rapport TER

Etudiants

KILANGA NYEMBO	Fanny
PELLEGRINO	Laurent
TROVATO	Alexandre

Encadrant

HUET	Fabrice

2008-2009

Remerciements

Nous tenons tout particulièrement à remercier Imen Filali et Brian Amedro pour leur aide ainsi que notre encadrant Fabrice Huet sans oublier l'équipe Oasis de l'Inria Méditerranée qui a eu la gentillesse de répondre à nos diverses questions tout au long de notre Travail d'Étude et de Recherche.

Table des matières

1 Introduction **3**

2 Etat de l'Art **4**
- 2.1 Historique des réseaux pair-à-pair 4
  - 2.1.1 Réseaux pair-à-pair non-structurés 5
  - 2.1.2 Réseaux pair-à-pair structurés 5
- 2.2 Protocoles étudiés 6
  - 2.2.1 CAN (Content Adressable Network) 6
  - 2.2.2 Chord 8
- 2.3 ProActive 8
  - 2.3.1 Les objets actifs 9
  - 2.3.2 Sémantique des communications 9
  - 2.3.3 Communications de groupes typés 10
  - 2.3.4 Fonctionnalités avancées 11
  - 2.3.5 Déploiement GCM avec ProActive 11

3 Travail effectué **16**
- 3.1 Choix de conception 16
  - 3.1.1 Représentation d'un pair 16
  - 3.1.2 Communication entre pairs 16
- 3.2 Implémentation 18
- 3.3 Déploiement 23

4 Gestion de projet **25**
- 4.1 Environnement de travail 25
  - 4.1.1 Outils utilisés 25
  - 4.1.2 Subversion 25
  - 4.1.3 Wiki 26
- 4.2 Gestion de risque 26
- 4.3 Moyens de contrôle 26
- 4.4 Répartition des tâches 27

5 Conclusion **28**

Chapitre 1

Introduction

Les réseaux pair-à-pair ont beaucoup évolué ces dernières années à tel point que les dernières générations de systèmes pair-à-pair fournissent à leurs usagers des applications de « télévision sur Internet » et de « Vidéo-à-la-demande ».

Que ce soit pour un usage grand public tel que les deux exemples cités précédemment ou bien pour des applications scientifiques afin de répartir des calculs ou stocker des données, il est de nos jours de plus en plus courant d'avoir la nécessité de recourir à l'utilisation de réseaux pair-à-pair.

Il existe deux approches différentes concernant les réseaux pair-à-pair : structurés et non-structurés. *ProActive* disposant déjà d'une implémentation de réseaux pair-à-pair non-structurés, notre travail consiste donc à :

- étudier les concepts de la programmation distribuée, en particulier avec *ProActive* ;
- étudier les protocoles *CAN* et *Chord* ;
- élaborer une librairie flexible basée sur *ProActive* pour former des réseaux pair-à-pair structurés ;
- valider l'implémentation en testant un protocole structuré à large échelle (c'est-à-dire sur plusieurs centaines de pairs) ;
- étendre l'implémentation de *CAN* et de *Chord* dans le but de pouvoir répondre aux critères du projet européen *SOA4ALL* consistant à utiliser des réseaux pair-à-pair structurés pour stocker intelligemment des données à grande échelle.

Ce rapport est découpé en plusieurs grandes parties qui reflètent les points importants de cette étude. Le chapitre 2 permet d'entrer dans le contexte des réseaux pair-à-pair et aussi de comprendre leurs évolutions. Nous y présentons également la librairie *ProActive* que nous avons étudié. Le chapitre 3 présente quant à lui le travail effectué tant sur le plan de l'implémentation, que sur les différents choix effectués tout au long de notre travail. Le chapitre 4 est consacré à la gestion du projet. Celui-ci présente notre environnement de travail, la gestion des risques, les moyens de contrôles mis en place ainsi que la répartition des tâches. Pour terminer, le chapitre 5 effectue un bilan de nos travaux et présente certaines perspectives.

Chapitre 2

Etat de l'Art

Pour échanger des fichiers, les ordinateurs doivent communiquer entre eux. Le mode de communication le plus répandu est la connexion dite « client-serveur [1] ».

Dans le cas de l'échange de fichiers sur des réseaux « pair-à-pair », les ordinateurs qui participent à l'échange sont tour à tour demandeur et donneur, client et serveur. Ce sont des « pairs » : la communication s'effectue sur un pied d'égalité.

Pour qu'un réseau pair à pair existe et fonctionne, il faut que le réseau soit doté de règles de fonctionnement communes à tous : c'est ce que l'on appelle le protocole du réseau. C'est lui qui spécifie comment et selon quel format les ordinateurs s'échangent des informations. Par exemple : qui possède tel ou tel fichier ? comment initier un transfert de données ? Les protocoles identifient l'ordinateur qui propose le fichier, et contrôlent le transfert des données.

2.1 Historique des réseaux pair-à-pair

Le protocole d'un réseau pair-à-pair englobe toutes les règles organisant les échanges de messages entre pairs. Toutes les activités sont ainsi encadrées : recherche des adresses IP, gestion et contrôle des fragments de fichiers échangés, etc. Cependant, la mise en place de ces protocoles génère un trafic d'informations très abondant sur ces réseaux. C'est à partir de cette constatation que l'on a vu apparaître deux types de réseaux pair-à-pair : ceux non-structurés [2] et ceux structurés cherchant à optimiser les protocoles sur divers critères (quantité de données échangées, complexité de recherche) en utilisant diverses topologies ayant des propriétés mathématiques avantageuses.

1. Un ordinateur, le client, émet une demande qu'on appelle une requête. Il l'émet à destination d'un autre ordinateur, le serveur, qui contient le fichier recherché et l'envoie au client. Le mode client serveur est, par exemple, le mode de fonctionnement de la navigation sur internet. L'internaute, grâce au navigateur, émet à partir de son ordinateur une requête au serveur du site internet qu'il consulte. Le serveur lui envoie en réponse les fichiers de la page web, les images, les vidéos, etc.
2. Réseau pair-à-pair dont les liens entre les pairs sont choisis arbitrairement.

2.1.1 Réseaux pair-à-pair non-structurés

Dans les réseaux pair-à-pair non-structurés chaque pair est complètement autonome et la propagation des requêtes se fait par inondation en interrogeant tous les pairs du réseau jusqu'à l'obtention du pair concerné. Les mécanismes sont simples et faciles à implémenter cependant les performances lors de la recherche d'un pair sur le réseau sont souvent limitées.

Le réseau *Gnutella*[Wik] est l'un des premiers réseaux non-structurés qui a mis en évidence le problème d'abondance de communication entre pairs. Pour exemple, dans sa version *0.4*, le volume des données liées au protocole était aussi important que celui des fichiers échangés.

2.1.2 Réseaux pair-à-pair structurés

Les réseaux pair-à-pair structurés recherchent l'efficacité. Pour cela ils organisent l'espace des pairs ainsi que la répartition des ressources sur les pairs selon une topologie ayant certaines propriétés avantageuses. Par exemple un espace en N dimensions pour *CAN* ou un anneau pour *Chord*. La gestion des ressources quant à elle s'appuie souvent sur l'utilisation d'une table de hachage distribuée.

Le principe des tables de hachage distribuées consiste à utiliser une fonction de hachage pour faire correspondre à chaque ressource une valeur de hachage sous forme d'une chaîne de bits de longueur fixée (supérieure ou égale à 128 bits). Cette chaîne de caractères devient l'« identifiant » du fichier. Les nœuds en charge d'une empreinte sont ceux dont l'identifiant est très proche.

Dans l'approche par tables de hachage distribuées, grâce à la structuration des connaissances, on peut trouver les nœuds en charge de l'empreinte en routant de proche en proche en un nombre d'étapes qui est logarithmique par rapport au nombre de nœuds. La recherche est donc beaucoup plus efficace : typiquement, une dizaine d'échanges de messages au lieu de quelques centaines voire quelques milliers si la donnée cherchée n'est pas répliquée.

Pour signifier le partage d'un fichier, son possesseur doit contacter le nœud en charge de l'empreinte du fichier pour lui indiquer son adresse IP. Un pair qui recherche le fichier n'aura qu'à contacter ce même nœud pour retrouver l'adresse IP du pair (ou des pairs) le partageant.

C'est donc sur ce genre d'approche qu'ont été élaborés les protocoles structurés tel que *CAN* ou *Chord*.

2.2 Protocoles étudiés

2.2.1 CAN (Content Adressable Network)

CAN est un protocole de réseau pair-à-pair structuré s'appuyant sur une topologie à N dimensions. Les ressources sont réparties dans une zone multidimensionnelle, délimitée par des coordonnées. Chaque pair s'occupe d'une zone. Supposons que nous sommes dans un repère en deux dimensions, alors chaque pair s'occupe de gérer une zone définie par deux coordonnées minimales et maximales tout en maintenant à jour la liste de ses voisins (nord, est, sud, ouest). Cette zone contient les ressources que le pair partage. Un exemple de réseau *CAN* en deux dimension est illustré sur la Figure 2.1.

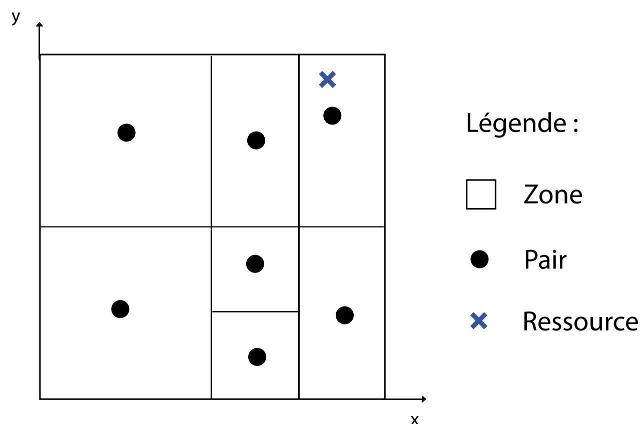


FIGURE 2.1 – Exemple de réseaux pair-à-pair CAN en 2 dimensions

Lookup

L'algorithme de *Lookup* (illustré sur la Figure 2.2) permet de trouver un pair sur le réseau. Pour cela il faut pouvoir router un message sur le réseau afin de trouver celui qui gère les coordonnées cherchées. Le routage consiste donc en trois étapes :

1. si le pair sur lequel on se trouve contient les coordonnées cherchées, le pair à été trouvé on peut donc renvoyer un message de réponse ;
2. sinon on parcourt les différentes dimensions : on part de la première, si sur cette dimension la cordonnées cherchée est contenue par le pair on passe à la dimension suivante sinon on envoie le message au pair voisin se trouvant le plus proche de la coordonnée cherchée sur l'axe courant ;
3. si l'on vient à avoir plusieurs voisins pour une dimension et une direction, il faut alors envoyer le message au voisin étant le plus proche de la coordonnée cherchée sur la dimension suivante.

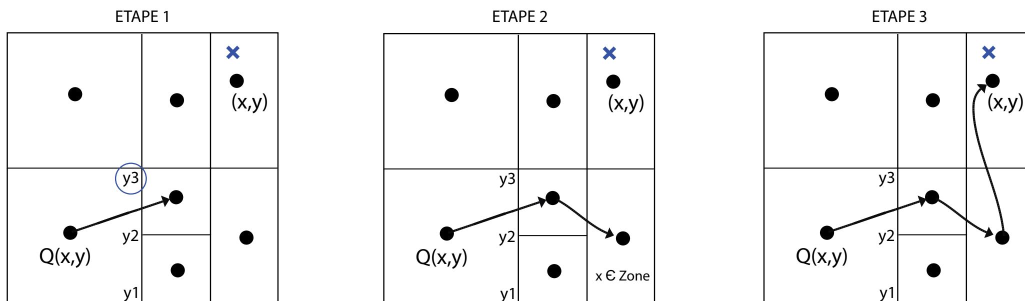


FIGURE 2.2 – Exemple de routage d'un message

Join

La fonctionnalité d'adhésion au réseau qui utilise l'algorithme de *Split* permet à un nouveau pair de joindre un réseau existant. Lors de l'arrivée d'un nouveau pair sur un réseau il faut pouvoir scinder l'espace géré. L'algorithme défini dans *CAN* se décrit en plusieurs étapes. En effet, le nouveau pair doit pouvoir se positionner sur le réseau. Pour cela, il faut :

- choisir des coordonnées aléatoirement ;
- par le mécanisme de routage, trouver le pair gérant ces coordonnées sur le réseau ;
- scinder la zone du pair trouvé, distribuer les ressources et mettre à jour les voisins ;
- informer les voisins du pair trouvé de l'arrivée d'un nouveau pair.

Leave

Cet algorithme permet à un pair de quitter proprement le réseau, sans perte d'informations. En effet, lors de son départ, le pair doit pouvoir distribuer ses ressources aux pairs voisins. Pour cela, il notifie ses voisins de son départ afin que sa zone soit fusionnée avec le pair dont il est issu lors de la dernière division de zone.

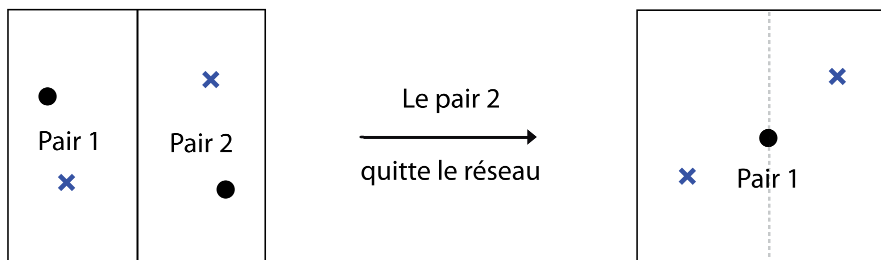
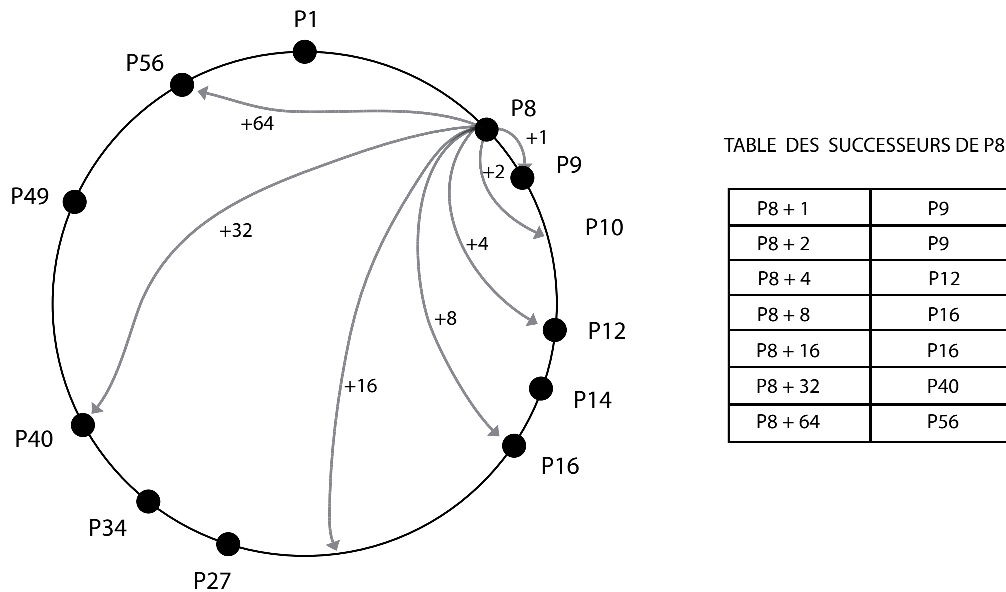


FIGURE 2.3 – Algorithme protocole CAN : Leave

2.2.2 Chord

Chord utilise quand à lui une topologie en anneau. Il a pour particularité de disposer d'algorithmes d'une complexité d'au plus $O(logN)$ requêtes pour trouver une information dans un anneau de N éléments en utilisant le principe des tables de hachage distribuées. Afin d'obtenir cette complexité, chaque pair maintient une liste de successeurs. Cette liste correspond aux pairs successeurs en puissance de 2 (comme illustré sur la Figure 2.4).



P8 + 1	P9
P8 + 2	P9
P8 + 4	P12
P8 + 8	P16
P8 + 16	P16
P8 + 32	P40
P8 + 64	P56

FIGURE 2.4 – Exemple de réseau pair-à-pair Chord

2.3 ProActive

ProActive est une bibliothèque Java pour le calcul parallèle, distribué et concurrent. Elle est conçue pour permettre de déployer une application sur un grand nombre de machines et ce, avec un maximum de transparence. Le langage Java a été retenu car il permet, avec sa machine virtuelle, une grande souplesse vis-à-vis des multiples architectures que l'on peut rencontrer sur une grille de calcul [3].

La bibliothèque permet la création d'objets distants, la mobilité des applications, les appels de méthodes asynchrones et des communications de groupe. Avec un ensemble réduit de primitives, *ProActive* fournit une API permettant la programmation d'appli-

3. Une grille de calcul permet de faire du calcul distribué : elle exploite la puissance de calcul (processeurs, mémoires, etc.) de plusieurs centaines, voir de millier d'ordinateurs afin de donner l'illusion d'un ordinateur virtuel très puissant. Ce modèle permet de résoudre d'importants problèmes de calcul nécessitant des temps d'exécution très longs en environnement « classique ».

cations distribuées pouvant être déployées aussi bien sur des réseaux locaux (LAN) que sur un ensemble de grilles de calcul interconnectées via Internet.

L'un des avantages de cette plateforme est qu'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 bibliothèque. 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, en respectant une sémantique précise.

2.3.1 Les objets actifs

Les objects actifs sont l'un des concepts de base pour développer une application répartie avec *ProActive*. Un objet actif dispose de sa propre *thread* qui lui permet d'exécuter les méthodes invoquées sur cet objet actif par d'autres objets (actifs ou non). *ProActive* permet au programmeur de ne pas manipuler explicitement les threads et surtout la synchronisation.

Un objet actif peut être créé sur n'importe quel hôte utilisé pour le déploiement de l'application. Une fois cet objet créé, son activité et sa position (locale ou distante) sont parfaitement transparents. Un objet actif peut ainsi être manipulé comme s'il s'agissait d'un objet java standard.

Le fonctionnement de *ProActive* s'appuie sur deux grands principes :

- une application distribuée est structurée en sous-systèmes. Il y a donc un objet actif (et donc une thread) pour chaque sous-système et un sous-système pour chaque objet actif. Chaque sous-système se compose ainsi d'un objet actif, et éventuellement de plusieurs objets classiques (dit « passifs »). La thread d'un sous-système n'exécute que les méthodes des objets qui le constitue ;
- il n'y a pas de partage d'objets passifs entre sous-systèmes.

Pour accueillir un objet actif, *ProActive* a introduit la notion de noeud qui permet d'identifier la *JVM* sur laquelle on souhaite le créer. Un noeud est une entité logique capable d'accueillir plusieurs objets actifs.

La création d'un objet actif se réalise alors comme suit :

```
MaClasse oa = (MaClasse) ProActive.newActive("MaClasse", parametres, noeud);
```

On peut remarquer qu'il existe d'autres approches pour la création d'un objet actif, notamment celle qui consiste à rendre actif un objet passif avec la méthode `turnActive()`.

2.3.2 Sémantique des communications

Pour ses communications entre objets, *ProActive* s'appuie sur Java RMI (Remote Method Invocation). Ce dernier a l'avantage d'être présent dans toutes les distributions standards de Java. Il est important de noter qu'un appel RMI est bloquant, ce qui peut entraîner des attentes inutiles dans l'exécution d'un programme, comme par exemple l'attente d'un résultat qui ne sera utilisé que plus tard.

Pour parer à cet inconvénient de taille, *ProActive* permet d'effectuer des communications aussi bien asynchrones que synchrones. Le choix du mode de communication se fait en fonction de la signature de la méthode :

Synchrone : la méthode retourne un objet non réifiable ou peut lever une exception. Dans ce dernier cas, l'appelant doit être bloqué pour éviter qu'il ne sorte du bloc `try/catch`. Au total, deux messages auront été échangés. L'un lors de l'appel de la méthode, et l'autre lors de la réception du résultat, une fois que la méthode aura été exécutée. Entre temps, la thread appelante est bloquée.

Asynchrone : la méthode retourne un objet de type réifiable et ne peut lever aucune exception. Deux messages seront là aussi échangés, le premier pour l'appel de méthode et le second pour la réception du résultat, mais entre temps, la thread appelante n'aura pas été bloquée. Dans l'attente du résultat, *ProActive* génère un objet *futur*. Une utilisation de cet objet sera bloquante tant que le vrai résultat ne sera pas arrivé.

Sens-unique : la méthode ne retourne rien (`void`) et ne lève pas d'exception. Un seul message est envoyé pour l'appel de méthode, et la thread n'est pas bloquée. Aucun *futur* n'est créé étant donné qu'il n'y a pas de résultat attendu.

Les objets *futurs* sont utilisés par *ProActive* pour créer un comportement asynchrone alors que les appels RMI sous-jacents sont, eux entièrement synchrones. En effet, dès qu'un appel de méthode sur objet actif est réalisé, *ProActive* construit et retourne immédiatement un objet vide simulant l'objet attendu : le *futur*. La requête RMI est alors déléguée à un autre fil d'exécution. Une fois la requête traitée, le résultat obtenu est placé dans le futur. Le futur implémente la même interface que l'objet résultat.

Ce dernier point explique la nécessité pour un type retour d'être réifiable (du moins si l'on souhaite profiter de l'asynchronisme), car l'objet retourné devra pouvoir être sousclassé. Une classe finale ou un type primitif ne peut donc pas convenir.

Dans le cas où l'objet *futur* est utilisé alors que le vrai résultat de l'appel qui en est à l'origine n'est pas encore arrivé, *ProActive* introduit le mécanisme *d'attente par nécessité* qui bloque l'exécution jusqu'à l'arrivée du vrai résultat.

De plus, l'échange des différents messages est fiabilisée grâce au fait que RMI se base sur le protocole TCP. L'appelant peut donc être assuré que le message a été correctement délivré à l'objet actif appelé et qu'il a été placé dans la file d'attente des requêtes que l'objet actif appelé doit servir.

2.3.3 Communications de groupes typés

Dans un programme distribué, il est fréquent pour le programmeur d'avoir à travailler avec un ensemble d'objets de même types, que nous appellerons *groupe typé*. La communication de groupe est un point crucial dans une application distribuée où la performance est un point important.

ProActive propose un mécanisme très flexible pour travailler avec un groupe d'objets comme s'il ne s'agissait que d'un simple objet unique.

Ce système de communication de groupe repose sur le mécanisme élémentaire d'invocation distante et asynchrone des méthodes. 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, ce mécanisme est la généralisation du mécanisme de méthode asynchrone sur des objets distants.

Du point de vue de la programmation, utiliser un 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, comme nous l'avons vu en abordant la sémantique des communications.

2.3.4 Fonctionnalités avancées

ProActive dispose de fonctionnalités avancées. Lorsqu'une méthode nécessite des paramètres, nous pouvons avoir deux comportements :

La diffusion (broadcast) va envoyer une copie des paramètres à tous les membres du groupe sans distinction. On utilise donc, ce comportement si l'on souhaite diffuser une même information à tout le monde. C'est le mode de fonctionnement par défaut des groupes de *ProActive*.

La distribution (scatter) propose de distribuer les paramètres *en fonction* du membre. Il est ainsi, par exemple, possible de disperser sur plusieurs membres différentes parties d'un tableau. Pour ce faire, on rassemble les données à distribuer au sein d'un groupe que l'on passe en paramètre à l'appel de méthode. Le choix de ce mode, se fait en utilisant la méthode *setScatterGroup()* sur le groupe de données.

2.3.5 Déploiement GCM avec ProActive

Le déploiement d'une application est un point critique à ne pas négliger. C'est notamment le cas lorsqu'on souhaite tester un réseau pair-à-pair structuré à grande échelle.

Grid Component Model (GCM)

Le premier principe du déploiement GCM est d'éliminer complètement du code source le nom des machines sur lesquelles on déploie l'application ainsi que le protocole utilisé pour les recherches dans le RMI Registry et les accès aux machines sur lesquelles on déploie.

Le but est de pouvoir déployer n'importe quelle application n'importe où sans toucher au code source. Par exemple, il doit être possible d'utiliser indifféremment divers protocoles tels que *rsh*, *ssh*, *Globus*, *LSF*, etc. pour la création de JVM dont l'application a besoin. De la même manière, l'enregistrement ou la découverte de ressources existantes

peut être réalisé à l'aide de divers protocoles tel que *RMIregistry*, *Globus*, etc. La création, l'enregistrement et la découverte de ressources doit donc être réalisée en dehors du code métier de l'application.

Le second point clef du déploiement GCM est la capacité de décrire de manière abstraite une application ou une partie d'elle en terme d'activités. Cette description doit comprendre les différentes entités parallélisées ou distribuées de l'application. Il faut également noter que la description abstraite d'une application et la manière de la déployer sont des informations liées. Si nous avons par exemple un moteur de simulation qui doit s'enregistrer dans le RMI Registry avec un certain protocole, alors les autres entités se chargeant de réaliser les calculs doivent également utiliser ce protocole afin d'effectuer le lookup et lier le moteur pour leur exécution. De plus, une partie du programme peut s'occuper d'effectuer une recherche pour le moteur (en supposant qu'il est démarré indépendamment), ou bien créer explicitement le moteur de simulation elle même.

Pour résumer, afin d'abstraire la machine sur laquelle à lieu l'exécution et permettre un déploiement indépendant, la librairie doit fournir les éléments suivants :

- une description abstraite des entités distribuées d'un programme ou d'un composant parallélisé
- une liaison (faite en dehors du code métier) de ces entités vers des machines réelles en utilisant un protocol choisi pour la création, l'enregistrement et la recherche.

Pour atteindre ce but, le modèle de programmation utilise la notion de « Nœud virtuel ». Un Noeud Virtuel est identifié par un nom. Il est définit et configuré dans un fichier XML de déploiement spécifique nommé le GCMD (*Grid Component Model Descriptor*). Un Noeud Virtuel est donc utilisé, après activation, dans le code source d'une application. Il est lié à un ensemble de Noeuds ProActive.

Il est important de noter que la distribution des entités (objets actifs) a lieu sur des Noeuds et non pas sur les Noeuds Virtuels. Les Noeuds virtuels sont un concept d'application tandis qu'un Noeud est un concept de déploiement : c'est un objet qui vit à l'intérieur d'une JVM, hébergeant des objets actifs.

Le processus de déploiement est illustré sur la Figure 2.5 : l'application *ProActive* commence par charger le descripteur de déploiement et déploie à distance sur les machines indiquées dans le descripteur. Bien que le processus derrière le déploiement soit assez compliqué, tout cela est abstrait par *ProActive*. Dans l'application nous devons seulement indiquer le chemin vers le descripteur de déploiement GCMA (*Grid Component Model Application*) et dire à *ProActive* de démarrer les Noeuds et les objets actifs. Les communications étant manipulées par *ProActive* selon le descripteur.

Pour faciliter le déploiement tout en respectant ce qui a été dit précédemment, GCM utilise deux descripteurs de déploiement séparés : le GCMA et le GCMD. La liaison entre les deux est réalisée dans le descripteur d'application en incluant un ou plusieurs descripteurs de déploiement.

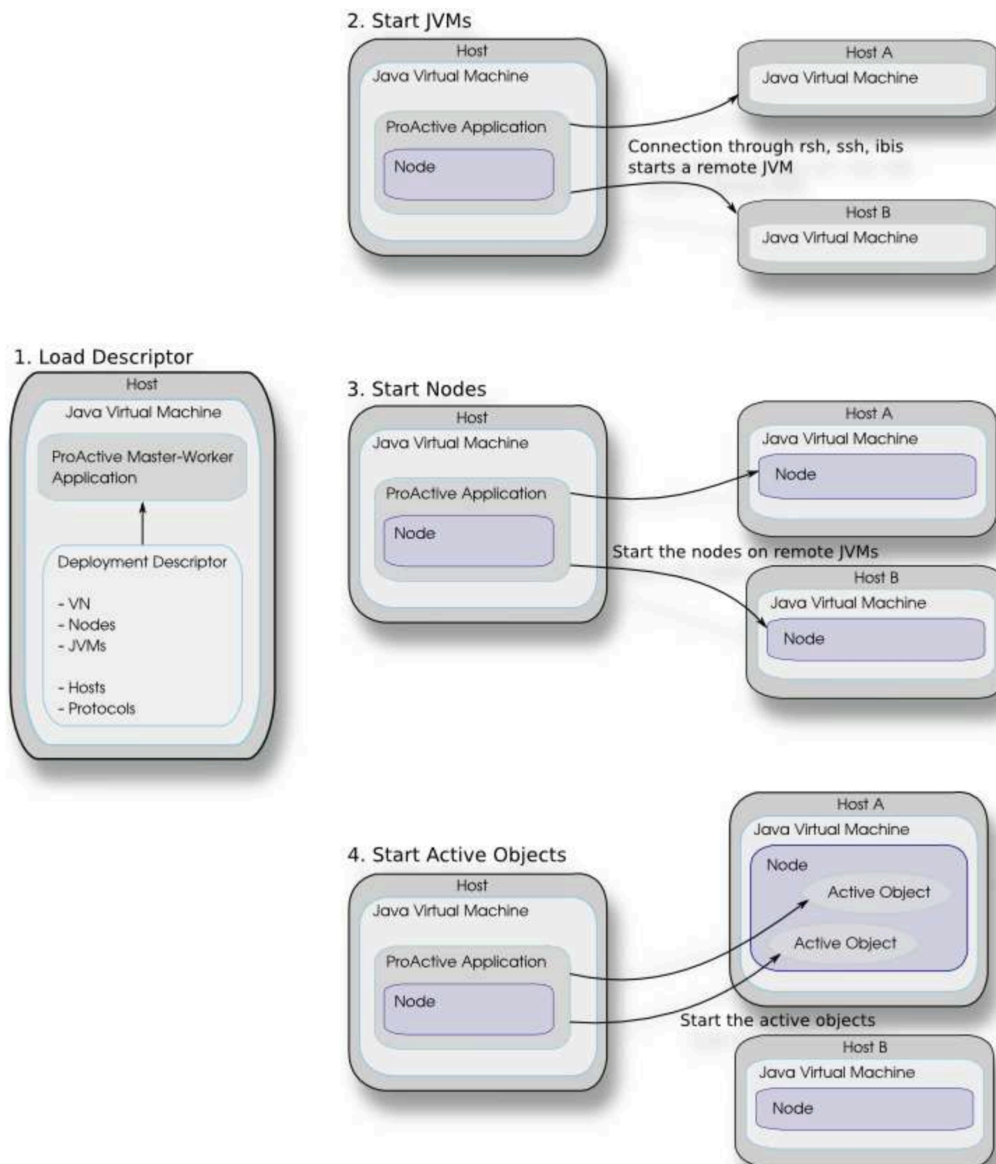


FIGURE 2.5 – Etapes de déploiement GCM

Le descripteur d'application (GCMA) permet de décrire comment lancer l'application souhaitée en indiquant les ressources dont elle a besoin. Les différentes options sont écrites dans un fichier au format XML.

Voici un exemple de fichier GCMA :

```
<?xml version="1.0" encoding="utf-8"?>
<GCMApplication xmlns="urn:gcm:application:1.0"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="urn:gcm:application:1.0 http://proactive.inria
                    .fr/schemas/gcm/1.0/ApplicationDescriptorSchema.xsd">

  <environment>
    <javaPropertyVariable name="java.home" />
        <javaPropertyVariable name="proactive.home" />
        <javaPropertyVariable name="user.home" />
  </environment>

  <application>
        <proactive base="root" relpath="${proactive.home}">

      <configuration>
            <java base="root" relpath="${java.home}/bin/java" />

                <applicationClasspath>
                  <pathElement base="proactive"
                               relpath="dist/lib/ProActive_examples.jar" />
                  <pathElement base="proactive"
                               relpath="dist/lib/ibis-1.4.jar"/>
                  <pathElement base="proactive"
                               relpath="dist/lib/ibis-connect-1.0.jar" />
                  <pathElement base="proactive"
                               relpath="dist/lib/ibis-util-1.0.jar" />
                </applicationClasspath>

      </configuration>

      <virtualNode id="Grid2D" capacity="1" />
    </proactive>
  </application>

  <resources>
    <nodeProvider id="Grid2D">
          <file path="GCMD_Local.xml" />
        </nodeProvider>
  </resources>

</GCMApplication>
```

La section *environment* permet de spécifier les variables d'environnement devant être utilisées comme par exemple les variables d'environnement Java récupérées lors du lancement de l'application. La section *application* permet, quand à elle, d'indiquer le *classpath* à utiliser lors du déploiement sur les différentes machines. Cela permet, également de dire quel Noeud Virtuel utiliser, sachant qu'on peut indiquer pour chaque Noeud Virtuel la capacité, c'est-à-dire le nombre de Noeuds à créer. La section *resources* permet, quand à elle, de lier un Noeud Virtuel à un fichier de type GCMD à l'aide d'un identifiant.

Le descripteur de déploiement (GCMD) permet de décrire les ressources fournies par l'infrastructure et comment les acquérir. C'est comme son confrère, un fichier XML composé de différentes parties, chacune avec différentes options afin d'atteindre le but décrit précédemment.

Voici un exemple de fichier GCMD que nous allons commenter :

```
<?xml version="1.0" encoding="utf-8"?>
<GCMDeployment xmlns="urn:gcm:deployment:1.0"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="urn:gcm:deployment:1.0 http://proactive.inria.
                   fr/schemas/gcm/1.0/ExtensionSchemas.xsd">

  <environment>
    <javaPropertyVariable name="user.home" />
  </environment>

  <resources>
    <group refid="sshLan">
          <host refid="ComputeNode" />
        </group>
  </resources>

  <infrastructure>
    <hosts>
          <host id="ComputeNode" os="unix" hostCapacity="2" vmCapacity="2">
            <homeDirectory base="root" relpath="${user.home}" />
          </host>
        </hosts>

        <groups>
          <sshGroup id="sshLan" hostList="eon[1-20]" />
        </groups>
  </infrastructure>
</GCMDeployment>
```

De la même manière que pour le GCMA, on retrouve la section *environment*. La section *resources* permet de définir des groupes de machine à utiliser. La section *infrastructure* permet, quand à elle, de spécifier pour les groupes précédemment définis sous quel système d'exploitation elles tournent, leur adresses pour se connecter, etc. Les paramètres *hostCapacity* et *vmCapacity* définis dans la section *hosts* permettent respectivement d'indiquer un nombre d'application qui peut être déployé sur chaque machine faisant partie du groupe *hosts* défini et d'indiquer le nombre de Noeud *ProActive* à créer par JVM.

Chapitre 3

Travail effectué

3.1 Choix de conception

Notre contrainte principale est d'implémenter une librairie flexible de réseaux pair-à-pair structurés à l'aide de *ProActive*. Pour cela différents choix architecturaux ont été réalisés.

3.1.1 Représentation d'un pair

Les réseaux structurés ayant plusieurs opération basiques communes telles que le *lookup*, le *join*, le *leave*, etc. la classe `StructuredOverlay` permet donc d'abstraire les différentes méthodes communes aux divers protocoles de réseaux pair-à-pair structurés. Chaque protocole doit ensuite définir une classe héritant de `StructuredOverlay`. Par exemple, le protocole *Chord* doit implémenter une classe nommée `ChordOverlay` et hériter de `StructuredOverlay` afin de redéfinir les méthodes communes.

Nous avons choisi d'associer à un pair, un objet actif au sens *ProActive*. Nous avons également décidé qu'un pair est composé d'un objet de type `StructuredOverlay` passif (objet au sens java), d'un champs de type `OverlayType` indiquant le type de réseau structuré associé au pair instancié, ainsi que d'un objet `DataStorage` permettant d'abstraire les données partagées par le pair.

La Figure 3.1 illustre ce qui a été dit précédemment.

3.1.2 Communication entre pairs

La communication entre les différents objets actifs, ici pairs, se déroule par le biais de messages alors que ces communications pourraient s'effectuer directement par simple appel de méthode sur l'objet distant grâce aux abstractions de *ProActive*. Ce choix a été fait afin d'éviter de multiples problèmes d'interblocages [1] (*deadlocks*) entres les pairs. Ainsi en proposant cette solution, chacun des pairs pourra envoyer sa liste des actions à

1. Un interblocage est un phénomène qui peut survenir en programmation concurrente. L'interblocage se produit lorsque deux processus légers (thread) concurrents s'attendent mutuellement. Les processus bloqués dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique.

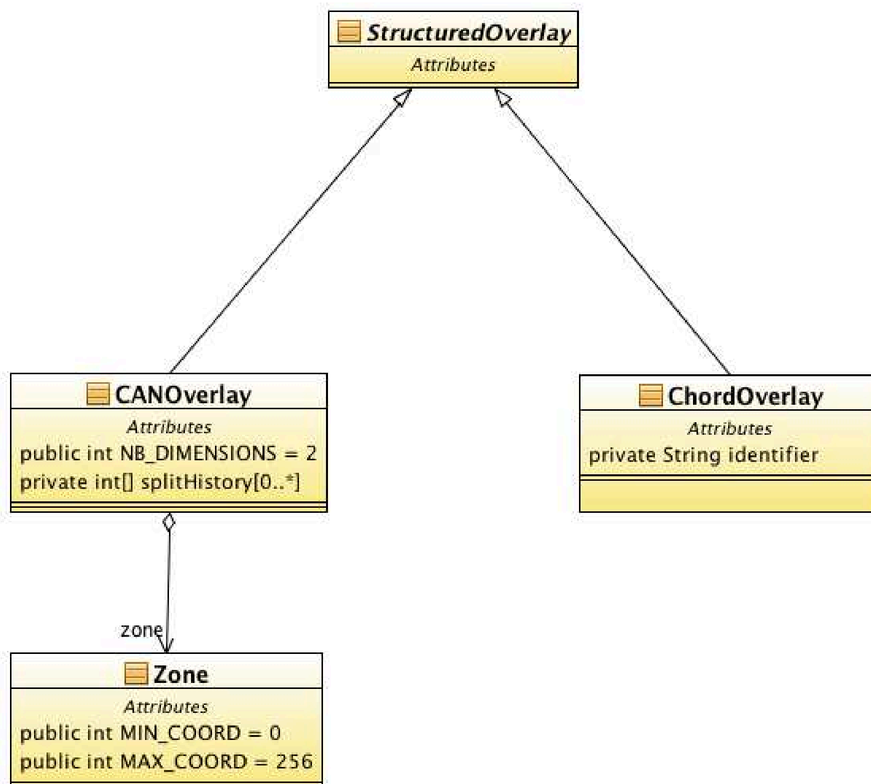


FIGURE 3.1 – Diagramme de classes partiel

effectuer à d'autres pairs. Celles-ci seront traités dans l'ordre sans être en concurrence avec les actions d'un autre pair.

Grâce au système de queues de requêtes, *Request Queue*, de *ProActive*, chacun des messages reçu par un pair est enregistré dans la queue propre à l'objet correspondant et est exécuté dans l'ordre de réception. Cela certifie alors que les requêtes envoyées entre pairs seront exécutées, et que si ce pair n'est plus présent, une exception Java sera alors levée permettant de signaler un problème lors d'une communication entre deux pairs.

En réponse à un envoi de message réalisé avec succès, une réponse est renvoyée au destinataire. Cette réponse peut ne contenir aucune information spécifique au message qui a été envoyé, auquel cas la réponse sera de type `ResponseMessage`. Ou bien dans le cas contraire avoir un type de réponse adapté au message envoyé. Dans ce cas la classe de réponse définie héritera de `ResponseMessage` afin de pouvoir contenir certaines informations tels que le RTT [2] associé à l'envoie du message. Une réponse générique a été définie sous le nom de `ActionResponseMessage` afin d'avoir en retour une réponse permettant uniquement de savoir si l'action envoyé via l'intermédiaire d'un message s'est bien déroulée.

3.2 Implémentation

Afin d'obtenir une librairie flexible il se doit de pouvoir facilement intégrer de nouveaux protocoles sans complètement modifier la conception de la bibliothèque. Pour cela nous avons dès le départ décidé d'utiliser le patron de conception [3] Strategy.

Le design pattern Strategy permet la séparation de plusieurs algorithmes. Ainsi l'algorithme peut varier indépendamment du client qui l'utilise. Dans notre cas, nous l'utilisons pour séparer les différents algorithmes de réseau pair-à-pair structurés tel que *CAN* et *Chord*. Cela nous permet d'abstraire les signatures de méthodes identiques pour chacun des types de protocoles. Ainsi, pour l'ajout d'un nouveau protocole de réseau pair-à-pair, les méthodes communes devront obligatoirement être définies et encapsulées dans un nouveau type d'objet qui étendra au sens Java, la classe `StructuredOverlay`.

CAN

Ci-dessous sont décrits nos choix d'implémentation ainsi que les algorithmes implémentés pour le protocole *CAN*. Les algorithmes ont tous été tirés d'articles scientifiques [RFH$^+$01] [Rat02] et pour certains adaptés.

2. Le Round-Trip delay Time ou RTT est, dans les réseaux informatiques, le temps que met un signal pour parcourir l'ensemble d'un circuit fermé. Il représente ici le temps pris en milli-secondes entre l'envoie d'un message depuis un pair et la réception d'une réponse depuis le même pair.

3. Un patron de conception (design pattern en anglais) est un concept destiné à résoudre les problèmes récurrents suivant le paradigme objet. Les patrons de conception décrivent des solutions standard pour répondre à des problèmes d'architecture et de conception des logiciels. À la différence d'un algorithme qui s'attache à décrire d'une manière formelle comment résoudre un problème particulier, les patrons de conception décrivent des procédés de conception généraux. On peut considérer un patron de conception comme une formalisation de bonnes pratiques.

Structures de données

Un point crucial pour l'implémentation de *CAN* est la structure de donnée utilisée afin de stocker les voisins d'un pair. Pour cela nous nous sommes appuyés sur le fait qu'un voisin se trouve dans une direction donnée pour une dimension fixée par rapport au pair courant :

- soit du côté inférieur ;
- soit du côté supérieur.

Pour stocker les références distantes des pairs voisins nous avons créé une classe `NeighborsDataStructure` dans laquelle nous utilisons un tableau à double dimension où chaque ligne correspond à une dimension et chaque colonne aux deux directions possibles. Chacune des cases du tableau contient une liste Java de type `Vector`. Ce choix ayant été fait car tous les appels de méthodes de la classe `Vector` sont synchronisés.

De la même manière les objets de type `Zone` associés aux pairs étant très souvent utilisés, ils sont mémorisés en parallèle dans le même type de liste que les références vers les pairs afin d'éviter des appels distants inutiles.

L'utilisation des groupes *ProActive* aurait pu être une meilleure solution que l'utilisation des listes de type `Vector`. Cependant afin d'éviter des parcours inutiles de la structure de données, nous avons montré qu'il était très intéressant de trier les références des pairs en fonction de la coordonnée minimale associée à la zone gérée par chaque pair voisin, pour une direction. Pour cette raison nous avons préféré utiliser la classe Java `Vector` qui permet en redéfinissant la méthode d'ajout, de positionner un objet à la position souhaitée afin d'obtenir une liste triée selon le critère défini précédemment.

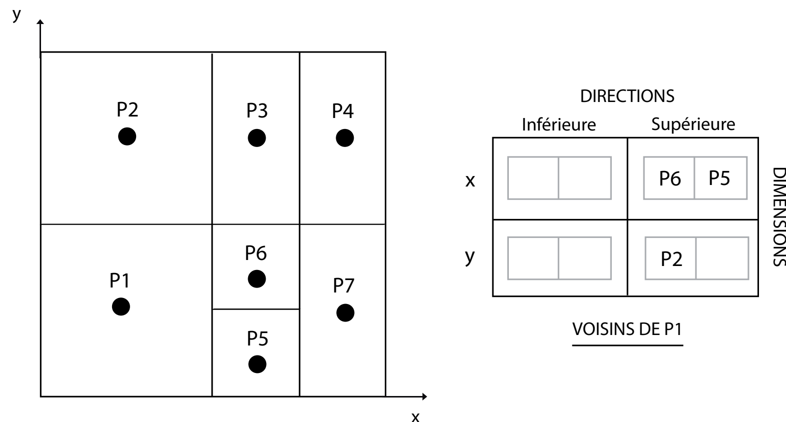


Figure 3.2 – Exemple de stockage des voisins du pair P1

Algorithmes implémentés

Join L'algorithme défini dans *CAN* se décrit en plusieurs étapes. Dans un premier temps, le nouveau pair doit pouvoir se positionner sur le réseau. Pour cela, nous avons différentes possibilités :

- lui affecter des coordonnées aléatoires ;
- laisser un *Tracker*[4] lui donner directement une référence vers un pair en fonction de la charge du réseau qui peut être évaluée en terme de communications, de taille de la zone gérée ou de ressources stockées, ce qui a pour but d'alléger le réseau.

Dans notre cas, nous avons décidé d'utiliser un *Tracker* afin de ne pas avoir de soucis lors du passage à l'échelle.

Le *Tracker* donne donc au pair voulant rejoindre le réseau, une référence vers un pair distant. Ensuite, le pair fait une demande d'adhésion au pair récupéré par message en utilisant la classe `CANJoinMessage`. Lors du traitement du message, le pair concerné est alors chargé d'affecter au nouveau pair :

1. sa nouvelle zone qui est le résultat de l'algorithme de *Split* (toutes les dimensions sont parcourues cycliquement) ;
2. son historique de split avec la dernière opération mais dans le sens opposé (pour conserver l'ordre de grandeur) ;
3. les ressources concernées par cette zone ;
4. les voisins du nouveau pair à partir des siens.

Pour finaliser cette adhésion, tous les voisins du pair présent doivent être informés de l'arrivée d'un nouveau pair afin de conserver l'état structuré du réseau. Cela est réalisé à l'aide de l'envoi d'un message de type `CANAddNeighborMessage` aux voisins concernés.

Leave Cet algorithme permet à un pair de quitter proprement le réseau, sans perte d'informations. En effet, lors de son départ, le pair doit pouvoir distribuer ses ressources aux pairs voisins. Pour cela, il envoie un message de type `CANLeaveMessage` informant ses voisins de son départ afin que sa zone soit fusionnée avec le pair dont il est issu lors de la dernière division de zone.

Merge Le *Merge* réalise les mêmes opérations que l'algorithme de *Split* lors du *Join*, mais en sens inverse. En effet, la zone occupée par le pair quittant le réseau est ajouté à celle gérée par au voisin se situant dans la dimension et le sens du dernier *Split*. Si le pair à plusieurs voisins, alors cet algorithme est appelé récursivement. Cependant cet algorithme n'est pas utilisable à grand échelle à cause de la quantité de données à transférer : lors d'un appel récursif de l'algorithme, l'échange des ressources entre les pairs peu devenir important. Ainsi, si le réseau est lourdement chargé, le temps de transaction ne sera pas acceptable.

4. Un traqueur est un serveur connaissant et gardant en permanence une référence vers une partie des pairs du réseau dont il est rattaché.

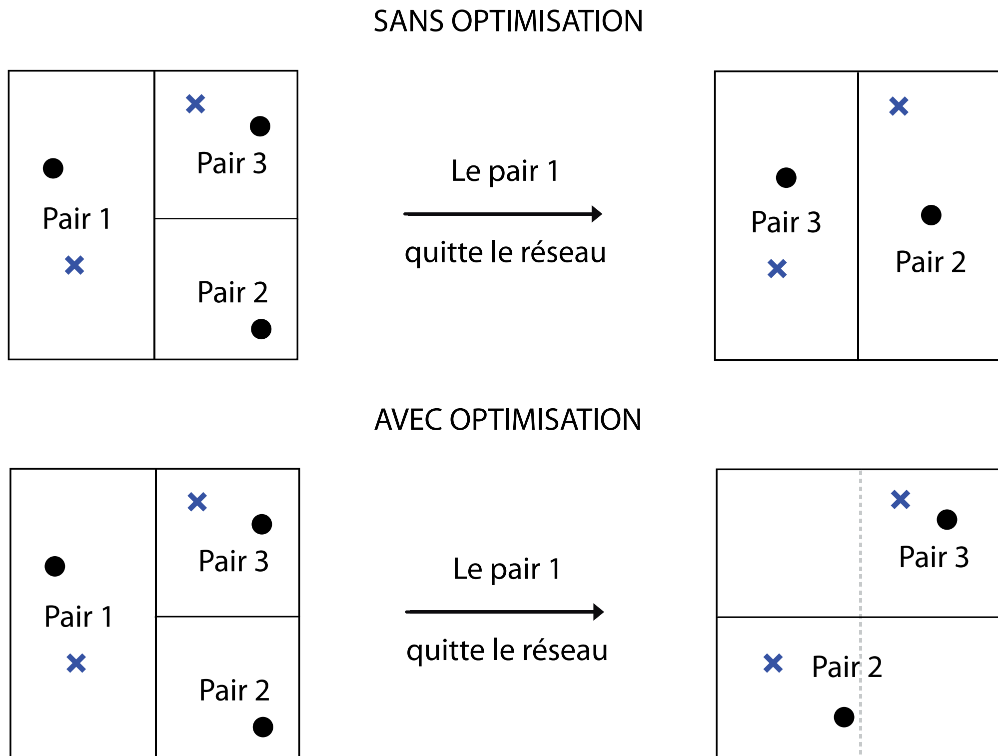


FIGURE 3.3 – Optimisation de l'algorithme du merge

Cet algorithme n'est pas utilisable à grande échelle à cause de la quantité de données à transférer : lors d'un appel récursif de cet algorithme, l'échange des ressources entre les pairs peu devenir important. Ainsi, si le réseau est lourdement chargé, le temps de transaction ne sera pas acceptable.

Pour éviter un transfert de ressources trop important, nous avons mis au point avec notre encadrant un algorithme permettant à un pair de fusionner avec d'autres lorsqu'il souhaite quitter le réseau. Plutôt que d'utiliser la récursivité, le principe consiste à distribuer la zone avec tous les voisins se situant dans la dimension et le sens du dernier *Split*.

L'optimisation est illustrée sur la Figure 3.3.

L'implémentation de l'algorithme de Merge n'étant pas des plus simples, nous avons réalisé une application graphique permettant de visualiser l'algorithme. Un aperçu écran est visible sur la Figure 3.4.

Lookup L'algorithme de *Lookup* permet de trouver un pair sur le réseau. Nos différentes actions se faisant par messages, il faut pouvoir router le message de pair en pair jusqu'à trouver celui gérant les coordonnées cherchées.

Afin d'éviter à avoir à tester le type des messages à traiter, nous avons utilisé dans l'implémentation le patron de conception *Double Dispatch*.

Figure 3.4 – Aperçu écran de l'application permettant de visualiser l'algorithme de Merge dans CAN

Double Dispatch est un concept permettant d'envoyer un appel de méthode différent selon le type de l'objet courant. Cela consiste simplement à appeler la méthode nécessaire par l'intermédiaire de l'objet courant. Dans notre cas, cette technique est utilisée pour le traitement des requêtes contenues dans les messages de communication. En effet, chacun des types de message est pourvu d'une méthode *ResponseMessage handle(Message)*, qui est chargée d'appeler la bonne méthode à exécuter par le pair concerné. Cela nous permet de simplifier l'exécution des requêtes par l'appel d'une unique méthode qui se chargera par la suite de réaliser le traitement qu'elle suggère ; tout en évitant à avoir à tester le type du message reçu pour le traiter.

Par exemple, les deux messages permettant de joindre ou de quitter le réseau n'attendent pas le même résultat, mais, après l'appel de la méthode *handle(Message)* sur le message, le traitement souhaité sera tout de même celui attendu, rejoindre ou quitter le réseau.

3.3 Déploiement

Afin de mettre en pratique ce que nous avons appris sur *ProActive* et le déploiement GCM, avant de commencer la conception de la librairie, nous avons réalisé une application permettant de démarrer beaucoup d'objets actifs sur plusieurs nœuds (en utilisant les vingt machines à huit cœurs chacune, disponibles à l'*INRIA* sous le nom de *eon1* à *eon20*). Au final, nous avons réalisé un programme simple qui démarre plusieurs machines virtuelles Java et plusieurs objets actifs afin de tester le déploiement à grande échelle. Cette application a ensuite été réutilisée et étendue afin de pouvoir valider l'implémentation de notre librairie à grande échelle. La Figure 3.5 illustre le répartition et les références entres les objets déployés après le lancement de l'application réalisée sous IC2D [5].

5. Interactive Control and Debugging of Distribution (IC2D) permet de monitorer des objets actifs déployés sur diverses machines.

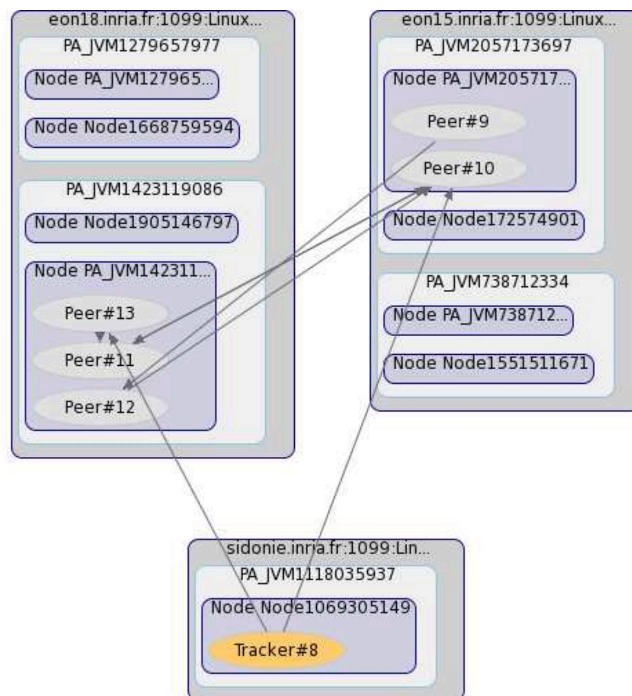


FIGURE 3.5 – Déploiement d'un réseau CAN avec 5 pairs sous IC2D

Chapitre 4

Gestion de projet

4.1 Environnement de travail

Dans le cadre du projet et afin d'éviter des conflits liés à la configuration propre à chacun, nous avons décidé d'avoir un environnement de programmation commun. Cela nous permet, ainsi, de pouvoir nous entraider plus facilement en cas de problème.

4.1.1 Outils utilisés

Voici les différents outils utilisés qui nous ont permis de mener à bien notre Travail d'Étude et de Recherche :

- Langage de programmation : Java 6
- Outil de développement : Eclipe
- Outil de documentation : JavaDoc et LaTeX
- Notation : UML

4.1.2 Subversion

La réalisation d'un projet en groupe nécessite une organisation particulière. De ce fait, l'utilisation d'un système de gestion de versions afin de pouvoir travailler ensemble sur plusieurs fichiers est quasiment inévitable. Nous avons pour cela décidé d'utiliser Subversion car nous savions tous l'utiliser et cela nous permet, ainsi, d'éviter l'apprentissage d'un outil plus évolué tel que Git.

Une branche nous a été créée sur le dépôt des sources de *ProActive* afin de pouvoir travailler avec la dernière version en date sans altérer le tronc principal.

Subversion est un logiciel de gestion de sources et de contrôle de versions. Ce type de programme a plusieurs fonctions, notamment de :

- garder un historique des différentes versions des fichiers d'un projet ;
- permettre le retour à une version antérieure quelconque ;
- garder un historique des modifications avec leur nature, leur date, leur auteur... ;
- permettre un accès souple à ces fichiers, en local ou via un réseau ;

– permettre à des utilisateurs distincts et souvent distants de travailler ensemble sur les mêmes fichiers.

4.1.3 Wiki

Un Wiki a également été mis en ligne afin d'y référencer les divers articles étudiés, y lister les différentes tâches à réaliser au fil du temps, écrire des notes pour l'élaboration du rapport final, etc. mais aussi pour que notre encadrant puisse suivre l'évolution de notre projet.

4.2 Gestion de risque

Le risque est inhérent à tous les projets. Cela désigne une condition ou un événement incertain qui, lorsqu'il se produit, a un effet négatif sur les objectifs du projet et bien souvent une incidence sur l'échéancier du projet.

Nous savions que cela pourrait se produit notamment dû au fait que nous ayons dû nous familiariser avec la librairie *ProActive* et que nous risquions de rencontrer certains problèmes lors de son utilisation car nous manquions encore d'expérience. Nous avons donc décidé qu'en cas de retard, nous implémenterions uniquement le protocole *CAN* et pas le protocole *Chord*. Notre conception devrait permettre de le faire aisément par la suite.

4.3 Moyens de contrôle

Nous avons globalement essayé de suivre le modèle de Cycle en V [1] qui permet, en cas d'anomalie, de limiter un retour aux étapes précédentes. En parallèle de l'implémentation une série de tests à l'aide de *Junit4* a donc été réalisée. Cela permet un contrôle de la qualité du projet tout en servant également de tests de non-régression. Ainsi en cas de problème si les tests couvrent une partie suffisamment large de l'application comme ce fût notre cas, ils permettent souvent de mettre en évidence les parties à corriger en cas de bogue.

Afin de permettre une réutilisation du code, nous avons été très attentif durant tout le projet sur la nécessité d'avoir des commentaires bien structurés et suffisamment explicatifs. Cela nous a permis de générer une JavaDoc complète permettant à quiconque devant reprendre notre travail, de comprendre ce qui a été fait.

1. Le cycle en V est devenu un standard de l'Industrie logicielle depuis les années 1980 et depuis l'apparition de l'Ingénierie des Systèmes est devenu un standard conceptuel dans tous les domaines de l'Industrie. Le monde du logiciel ayant de fait pris un peu d'avance en termes de maturité, on trouvera dans la bibliographie courante souvent des références au monde du logiciel qui pourront s'appliquer au système. Les différentes étapes consiste dans l'ordre en l'Analyse des besoins et faisabilité, à la Spécification logicielle, à la Conception architecturale, à la Conception détaillée, au Codage, aux Tests unitaires, Tests d'intégrations, Tests de validations et à la Vérification d'Aptitude au Bon Fonctionnement.

A mi-parcours, nous avons également présenté notre projet ainsi que son évolution et nos divers problèmes à l'équipe OASIS de l'INRIA Méditerranée. Cela nous a permis d'avoir des retours forts intéressants et donc d'effectuer en conséquence, certaines modifications.

4.4 Répartition des tâches

Notre travail d'étude et de recherche a été réalisé à temps plein durant un peu plus de 9 semaines consécutives.

Voici ci-dessous une approximation du pourcentage des tâches accomplies par chacun des membres du projet :

Liste des tâches	F. KILANGA NYEMBO	L. PELLEGRINO	A. TROVATO
Gestion de projet	0%	100%	0%
Apprentissage *ProActive*	34%	33%	33%
Etude articles	40%	30%	30%
Conception UML	33%	33%	34%
Implémentation Java	20%	40%	40%
Tests unitaires	50%	25%	25%
Rapport final	33%	34%	33%

Afin de pénaliser personne nous avons essayé de nous partager ces tâches pour que ce ne soit pas toujours la même personne qui travaille sur la même partie. C'est pour cette raison que le tableau récapitulatif ci-dessus présente une répartition assez équivalente du travail effectué sur certaines tâches.

Concernant l'implémentation, lorsque nous l'avons pu, nous n'avons pas hésité à appliquer la méthodologie « extrême programming » consistant à ce qu'une personne écrive le code tandis qu'une seconde vérifie en temps réel ce qui est écrit afin de déceler au plus vite, certaines parties du code qui pourraient être boguées, tout en alternant régulièrement les places entre les deux personnes appliquant cette méthode.

Chapitre 5

Conclusion

Travailler sur ce projet a été une aventure passionnante. Elle nous a permis de nous familiariser avec l'univers de la programmation distribuée en nous intéressant à l'utilisation et au fonctionnement de *ProActive*. C'est une bibliothèque qui nous étonne, de jour en jours par son étendue et ses capacités d'abstraction.

De façon globale, nous nous sommes approchés des objectifs demandés même si certains points n'ont pas pu être réalisés comme l'implémentation du protocole *Chord*.

Notre encadrant nous a fait part du fait que notre implémentation serait réutilisée dans le cadre du projet *SOA4ALL* consistant à utiliser des réseaux pair-à-pair (notamment *CAN* et *Chord*) afin de fournir un service de haut niveau permettant de stocker à grande échelle une quantité importante de données pouvant être retrouvées suivant certains mécanismes de requêtes. Il devrait être aisé d'implémenter le protocole *Chord* manquant en héritant de la classe abstraite `StructuredOverlay` et en implémentant les messages d'action spécifiques.

Bibliographie

[ABRR05] Brian Amedro, Vladimir Bodnartchouk, Judicaël Ribault, and Vincent Robitzer. Evaluation et optimisation des benchmarks paralleles de la nasa (npb) avec la bibliotheque proactive, 2005.

[LJZ+07] Ruqian Lu, Caiyan Jia, Shaofang Zhang, Lusheng Chen, and Hongyu Zhang. An exact data mining method for finding center string, April 2007.

[Pro99] ProActive. INRIA, 1999. http ://proactive.inria.fr.

[Rat02] Sylvia Ratnasamy. *A Scalable Content-Addressable Network*. PhD thesis, University of California at Berkeley, 2002.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-adressable network, 2001.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M.Frans Kaashoe, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications, 2001.

[Wik] Wikipedia. Gnutella. http ://fr.wikipedia.org/wiki/Gnutella.