

Modèle et Infrastructure de Programmation Pair à Pair

Alexandre Di Costanzo

▶ **To cite this version:**

Alexandre Di Costanzo. Modèle et Infrastructure de Programmation Pair à Pair. [Travaux universitaires] 2004, pp.50. inria-00001239

HAL Id: inria-00001239
https://inria.hal.science/inria-00001239

Submitted on 12 Apr 2006

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

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





DEA RSD

Modèle et Infrastructure de Programmation Pair-à-Pair

Alexandre di Costanzo

Projet OASIS, INRIA, CNRS-I3S, Université de Nice Sophia-Antipolis
Alexandre.Di_Costanzo@sophia.inria.fr

Encadrant : Denis Caromel

Mars-Juin 2004

Remerciements

Je remercie avant tout Denis Caromel qui m'a encadré durant mon stage. Il a su me guider par ses conseils avisés et je lui en suis reconnaissant.

Merci à Françoise Baude pour ses conseils et sa relecture de ce manuscrit.

Je tiens à remercier toute l'équipe du projet OASIS pour l'accueil chaleureux qui m'a été fait, et en particulier Christian pour m'avoir « supporté » dans son bureau.

Enfin, merci à Magali pour tout son temps passé à corriger mes fautes d'orthographe.

Table des matières

1 Introduction **5**
1.1 Contexte 5
1.2 Objectifs 5
1.3 Définitions 6

2 État de l'Art **9**
2.1 Grille et Internet Computing 9
2.1.1 GridRPC : NetSolve et Ninf 9
2.1.2 InteGrade 10
2.1.3 XtremWeb 10
2.2 Pair-à-Pair 11
2.2.1 Sun Microsystems Project JXTA 11
2.2.2 Windows Peer-to-Peer Networking 13
2.2.3 Peer-to-Peer Grids 13
2.3 Modèles de Programmation P2P 14
2.3.1 P2P-RPC 14
2.3.2 Objets et Passage de Paramètres Paresseux 15
2.4 Programmation Parallèle 16
2.5 ProActive 16
2.5.1 Modèle de Programmation 17
2.5.2 Infrastructure 17
2.6 Synthèse des Systèmes Étudiés 17

3 Une Infrastructure P2P à Auto Organisation Continue et Paramétrée **19**
3.1 Introduction 19
3.2 L'Infrastructure 19
3.2.1 Bootstrapping : Premier Contact 19
3.2.2 Overlay Networks de JVMs 20
3.2.3 Découverte et Auto Organisation Continue des Pairs 22
3.3 Implémentation avec ProActive 25
3.3.1 Les Ressources 25
3.3.2 Service P2P 25
3.3.3 Implémentation des Pairs 25
3.4 Conclusion 26

4 Modèle de Programmation et API Génériques **29**
4.1 Modèle de Programmation 29
4.1.1 Structure 30
4.1.2 Modèle de Programmation 30
4.2 API de Programmation 32
4.2.1 Solver 33
4.2.2 Problem 33
4.2.3 Result 34
4.3 Implémentation 34
4.4 Conclusion 35

5 Validation et Expérimentations **41**
5.1 Le Problème des N-Queens 41
5.1.1 Déploiement 42
5.1.2 Conclusion 44
5.2 Exemple TSP 44
5.2.1 Solver 44
5.2.2 Problem 45
5.2.3 Résultats 45

6 Conclusion et Perspectives **47**

Table des figures

3.1 Premier Contact. 20
3.2 Exemple d'overlay networks. 21
3.3 Exemple de topologie client-serveur. 21
3.4 Exemple de topologie hiérarchique. 22
3.5 Exemple de topologie sans organisation. 22
3.6 Coupure au sein d'un réseau P2P. 23
3.7 Attributs et méthodes d'un pair. 24
3.8 Représentation de l'implémentation d'un pair, grâce à l'outil IC2D. 26
3.9 Fonctionnement du réseau P2P. 27

4.1 Synthèse des différentes entités. 29
4.2 Gestion de l'infrastructure P2P en hiérarchique. 31
4.3 Liens entre la structure et le modèle de programmation. 33
4.4 Couple Problem - Worker ainsi, que les relations entres les Workers. 36
4.5 Synthèse du fonctionnement de l'API. 39

Liste des tableaux

2.1 Synthèse des Systèmes Étudiés 18

3.1 Actions fournies par un pair. 23
3.2 Paramètres du réseau P2P de JVMs. 24

4.1 L'objet Solver du modèle de programmation. 31
4.2 L'objet Problem du modèle de programmation. 32
4.3 L'objet Result du modèle de programmation. 32
4.4 Signature de la classe Solver. 34
4.5 Signature de la classe Problem. 35
4.6 Signature de la classe Result. 37
4.7 Signature de la classe Worker. 38

5.1 Résultat avec 20 et 21 Queens. 43
5.2 Exemple de la classe Solver pour TSP. 45
5.3 Exemple de la classe Problem pour TSP. 46

Chapitre 1

Introduction

1.1 Contexte

Depuis quelques années, notre société a un besoin croissant et important en puissance de calcul offerte par les ordinateurs, notamment dans la recherche scientifique (physique, chimie, biologie, etc.), pour résoudre des problèmes d'ingénierie ou d'environnement, par exemple, de simulations climatiques à l'échelle planétaire. La recherche n'est pas le seul domaine ayant besoin de puissance de calcul, on peut citer notamment l'industrie automobile ou encore le cinéma.
Afin, de répondre à cette nécessité, une solution, depuis 5 à 10 ans, est l'utilisation de *clusters*. Cette solution a un certain nombre d'inconvénients : c'est un ensemble de machines dédiées, de plus une fois un calcul terminé les machines restent sans occupation car le cluster est généralement innacessible par les utilisateurs, et enfin l'aspect infrastructure (espace physique important pour l'emplacement des machines, contrôle de la température de la pièce, nuisances sonores, etc.). Ces problèmes sont négligeables pour un cluster de petite taille (16 nœuds par exemple), mais lorsqu'on a besoin d'augmenter ses capacités, il n'est plus possible de ne pas en tenir compte.
Contrairement à cette approche, nous pouvons considérer les ressources disponibles au sein d'une entreprise, d'un laboratoire, ou d'une université. Ces ressources sont les PC de bureau des employés, des étudiants, etc. qui peuvent se compter en centaines voire milliers de postes. En analysant l'utilisation de ces machines, on peut en général observer qu'elles sont sans occupation pendant un temps important, la nuit. Nous avons d'un côté un besoin croissant de puissance de calcul et de l'autre, des machines qui ne font rien. Ainsi est né l'idée d'organiser en réseau Pair-à-Pair (P2P) les machines de bureau afin de servir d'infrastructure de calcul.
Le potentiel des réseaux P2P a été largement montré dans leur utilisation pour le partage de fichiers sur Internet. Contrairement aux Grilles de calculs, ces architectures ne sont pas (ou presque pas) déployées dans le monde du calcul scientifique. En effet, de part leurs caractéristiques : pairs hautement dynamiques, décentralisation, auto-organisé, etc., il est assez difficile de contrôler de tels environnements.

1.2 Objectifs

Les modèles et infrasturctures existants pour le calcul P2P sont peu sophistiqués : ce sont seulement des tâches qui travaillent indépendamment (avec en général pas de communication

entre celles-ci), et l'utilisation d'APIs de très bas niveau.
Bien sûr, il existe de nombreux protocoles qui directement au niveau de la couche communication savent créer des réseaux P2P et permettent ainsi à des machines de communiquer. Leur inconvénient, malgré le fait qu'ils soient très performants, est qu'ils sont au niveau des Sockets, donc assez bas niveau pour l'utilisateur. Notre infrastructure masque tout ceci à l'utilisateur qui peut se concentrer plus sur son application.
Il y existe aussi énormement d'APIs de programmation P2P, qui sont aussi très bas niveaux. Certes, elles sont complètes dans leurs offres de fonctionnalités mais requièrent du programmeur beaucoup d'effort pour les utiliser.
Dans ce cadre, l'objet de ce travail consistera en l'étude de modèles et d'infrastructures pour la programmation P2P. Cette étude permet de bien comprendre ces derniers, afin de témoigner des bénéfices que peuvent apporter la migration, l'asynchronisme, les communications typées, les groupes et l'impact possible des continuations automatiques (fonctionnalités de la bibliothèque ProActive [1]).
À partir de cette étude, nous proposons une infrastructure P2P, ainsi qu'un modèle de programmation pour le calcul P2P. Comme le calcul P2P semble bien adapté aux applications avec un faible rapport communication/calcul, nous essayerons d'appliquer à notre modèle conceptuel des algorithmes de recherche parallèle.
Nous propososons une implémentation de notre infrastructure ainsi que du modèle. Ces implémentations se feront, dans le cadre de l'équipe OASIS, à l'aide d'une bibliothèque en Java ProActive. Nous nous interdisons ainsi l'utilisation de pré-proccesseurs, toutes modifications du compilateur Java ou encore l'utilisation d'une JVM autre que standard.
Le chapitre 2 présente une étude non exhaustive des différents systèmes et modèles P2P qui existent. Ensuite le chapitre 3 détaille notre infrastructure P2P et son implémentation. Dans le chapitre 4, il est présenté notre modèle de programmation et de son API. Les expérimentations pratiques du chapitre 5 valident nos travaux.

1.3 Définitions

Il existe de nombreuses définitions différentes de la notion de réseaux Pair-à-Pair, par la suite nous utiliserons celle dite « Pur Pair-à-Pair » dans [2] :

Définition 1.3.1 *Un système distribué est appelé Pair-à-Pair (P-to-P, P2P, etc.), si les nœuds de ce réseau partagent une partie de leurs ressources (temps de calcul, espace disque, interface réseau, imprimante, etc.). Ces ressources sont nécessaires pour fournir les services et les contenus offerts par le système (e.g. le partage de fichiers ou les espaces de travail collaboratifs). Ils sont accessibles directement par les autres pairs sans passer par des entités intermédiaires. Les participants de ce genre de réseau sont à la fois des fournisseurs de ressources (services et contenus) et des utilisateurs de ces ressources. Ils sont ainsi clients et serveurs. Ces réseaux sont considérés comme « **Pur Pair-à-Pair** » si l'on peut enlever de façon arbitraire un nœud de ce réseau sans qu'il y ait aucune dégradation ou perte du service offert par ce réseau.*

Lorsqu'on dit « aucune dégradation », il faut comprendre sans dégradation non-linéaire, contrairement à une approche centralisée, la suppression de nœuds, les uns après les autres, peut entraîner une dégradation linéaire de la qualité de service.
Notre modèle de programmation et son API s'appliquent à la résolution de problèmes de

type *Branch and Bound* et plus particulièrement à l'adaptation au P2P des problèmes de type *Divide and Conquer*(diviser pour régner), la définition 1.3.2 définissant le concept d'algorithme de Branch & Bound.

Définition 1.3.2 *Les algorithmes de* ***Branch and Bound*** *sont en réalité des recherches arborescentes. Il s'agit d'effectuer une exploration implicite de l'espace des solutions optimumales du problème.*

Branch & Bound est en réalité le terme anglo-saxon pour parler de Séparation et Évaluation Progressive (SEP) ou encore de Branchement et Élagage. Dans la suite, c'est le terme de Branch & Bound que nous utilisons.

Chapitre 2

État de l'Art

Avant de commencer cet état de l'art, il nous faut un peu parler du projet SETI@home [3]. Cette infrastructure permet de résoudre un unique problème : SETI (Search for Extraterrestrial Intelligence). Ce projet a reçu l'aide de milliers d'utilisateurs, partageant la puissance de centaines de milliers de machines à travers le monde. C'est à l'heure actuelle le problème pour lequel il y a eu le plus grand temps de calcul effectué de l'histoire. En dépit de son succès, cette infrastructure ne permet pas de résoudre d'autres problèmes et son architecture n'est en fait, malgré les apprences pas P2P mais Maître-Esclaves.

2.1 Grille et Internet Computing

Le but de cette section n'est pas de faire l'état de l'art du monde de la Grille, mais de voir certaines de ses technologies qui peuvent servir de base à des architectures ou des modèles de systèmes P2P.

2.1.1 GridRPC : NetSolve et Ninf

Nous verrons, plus loin, un modèle de programmation pour les réseaux P2P basé sur RPC (voir section 2.3.1 page 14) qui repose, lui-même, sur GridRPC [4].
GridRPC propose un modèle de Grille, ainsi qu'une bibliothèque de programmation (Application Program Interface - API), sous forme d'un standard pour l'utilisation de RPC comme mécanisme au sein d'une Grille de calcul. Son but est d'unifier les différents systèmes grâce à une API commune. En d'autres termes, GridRPC décrit une API minimale pour fournir un mécanisme de base pour implémenter une large variété d'applications et de services pour la Grille.
C'est donc une abstraction de haut-niveau basée sur RPC, à laquelle a été rajouté la notion de tâches parallèles à gros-grains asynchrones, et ce, afin de masquer aux programmeurs la dynamicité et l'instabilité de la Grille.
Il est aussi possible d'utiliser une approche orienté-objets (notamment avec Java) pour avoir un modèle de programmation de plus haut niveau. Pour rester compatible avec les nouveaux standards de la Grille comme OGSA [5], il est possible d'implémenter GridRPC avec des Web Services comme base pour le modèle [6].
Il existe deux implémentations majeures du standard (qui sont d'ailleurs à l'origine de Gri-

dRPC) :

- NetSolve [7] est basé sur un modèle client-serveur qui fournit l'accès à des ressources physiques ou logicielles, à toute une variété d'interfaces clients (qui peuvent être en C, Fortran ou Matlab).
- Ninf [8] est lui aussi basé sur un modèle client-serveur. À noter, qu'il existe aussi Ninf-G qui est une implémentation de Ninf au-dessus de Globus [9].

Actuellement, le principal inconvénient à l'utilisation de GridRPC est le fait que les clients doivent connaître à l'avance l'adresse du ou des serveurs avant de faire un appel. Un autre inconvénient, qui n'en n'est pas un puisque l'on peut pas faire autrement, est que l'on doit aussi connaître la signature de la méthode à appeler, ce qui est incontournable pour faire du RPC, . Ces problèmes seront bientôt résolu par l'utilisation d'un système de découverte (comme Globus MDS) de fonctions, types et services.
Il est à regretter l'utilisation de RPC uniquement entre les clients et les serveurs. En effet, une fois l'appel de la méthode effectué sur le serveur, ce dernier délègue l'opération à une ou un ensemble de machines de sa Grille, sans faire de RPC, avant de retourner le résultat au client.

2.1.2 InteGrade

InteGrade [10] est un intergiciel qui permet d'utiliser les cycles CPU inutilisés des ordinateurs de bureau des grandes institutions (universités, grandes entreprises, etc.). Cet intergiciel est orienté-objets et permet la distribution des objets, notamment grâce à l'utilisation de CORBA comme base.
Contrairement à des approches Grilles plus conventionnelles, les Grilles InteGrade sont structurées comme des clusters, avec des groupes de nœuds (une 100^e de membres), les nœuds pouvant être des machines dédiées ou des postes utilisateurs partagés.
Un des aspects positifs de cette approche est que c'est l'utilisateur (du poste de travail) qui décide ou non de partager ses ressources avec le reste de la Grille. Dans le cas où l'utilisateur offre ses ressources, il pourra continuer de travailler sans dégradation des performances pour ses propres besoins. InteGrade intègre un système de surveillance et de prédiction de l'utilisation des ressources (*Local Usage Pattern Analyser* pour le contrôle local et *Global Usage Pattern Analyser* pour faire la synthèse de tout le système).
Le système fournit, aussi, un mécanisme de contrôle et de déploiement d'applications sur la Grille (*Application Submission and Control Tool*). Cet outil permet de contrôler, entre autre, l'avancement, par un mécanisme de points de reprise (checkpointing) de l'application.
Cette approche ne correspond pas vraiment à la définition 1.3.1 page 6. En effet, de part sa hiérarchie et le fait que certains des nœuds hébergent des éléments « vitaux » du système (par exemple le module *Global Usage Pattern Analyser*), il suffit de supprimer un de ces pairs pour dégrader ou supprimer le service rendu par la Grille.
Malgré cela, le point fort de cette architecture est l'utilisation des cycles CPU inutilisés des PC de bureau et de machines dédiées.

2.1.3 XtremWeb

XtremWeb est une plateforme expérimentale de *Global Computing* [11]. L'idée, derrière le Global Computing, est de récupérer le temps libre des ordinateurs connectés à Internet, ces ordinateurs étant répartis à travers le monde, on pourra ainsi, exécuter de manière largement

distribuée de très grosses applications. Le projet XtremWeb est né d'une requête des physiciens de l'observatoire Pierre Auger qui avaient besoin de grandes capacités de calcul.
Les problèmes liés au Global Computing sont : passage à l'échelle, systèmes hétérogènes, disponibilité des nœuds, tolérance aux fautes, sécurité, dynamicité, facilité d'utilisation et de mise en œuvre pour l'utilisateur. À remarquer, que ce sont les mêmes problèmes que pour la Grille en général, mais encore plus exacerbés car il n'y pas de moyen de supervision au sein du Global Computing, puisque tout est basé sur le volontariat.
Le projet a pour but de construire une plateforme pour expérimenter les capacités du Global Computing. La plateforme est structurée de telle manière que l'on puisse tester des *plug-in*.
On peut utiliser XtremWeb de deux manières :

- Comme un Utilisateur (*User*) : après s'être enregistré auprès d'un serveur d'administration d'XtremWeb, on peut faire tourner un *Worker* sur sa machine. De cette manière, lorsque sa machine ne fait rien, l'utilisateur peut collaborer au Global Computing.
- Comme un Collaborateur (*Collaborator*) : lui aussi doit être enregistré dans un serveur d'administration. Lorsque qu'un Collaborateur n'a pas de tâche à envoyer, il fonctionne comme un client Utilisateur, sinon, de façon pure P2P, le Collaborateur exécute son application de Global Computing sur la communauté de machines ainsi créée.

Ici les workers exécutent du code natif, pour ne pas dégrader les performances. Afin de garantir la sécurité du code, XtremWeb utilise des tests et de la cryptographie pour vérifier et protéger le code de modifications malicieuses.
Afin, de ne pas être bloqué par des firewalls, toutes les communications sont initiées par les workers. Un échange de communications entre un worker et un serveur est typiquement de ce genre : le worker s'enregistre auprès du serveur, lui demande une tâche, puis l'informe de son état durant le travail de la tâche et, ensuite, transmet au serveur le résultat. Le protocole de communication est indépendant de la couche communication qui peut être des sockets TCP-UDP/IP, CORBA ou encore Java RMI.
Ce qui est à regretter dans ce modèle est l'impossibilité qu'il y ait des communications entre les workers. Par ailleurs, XtremWeb propose uniquement une infrastructure pour déployer et contrôler ses tâches, mais aucune aide à la programmation de ces dernières. Cependant, de part l'obligation que les pairs ont de s'inscrire auprès de serveurs et que ces derniers s'avèrent devenir indisponibles, le réseau, créé par XtremWeb peut ne plus être accessible. XtremWeb est donc, pour notre part, plus Grille que P2P.

2.2 Pair-à-Pair

Nous n'aborderons pas ici, les réseaux P2P dits de distribution de contenu tel que FreeNet [12] et Gnutella [13], même si leurs architectures sont intéressantes à étudier.

2.2.1 Sun Microsystems Project JXTA

La plupart des applications P2P, actuellement existantes, sont dédiées à une seule fonction ou ne savent résoudre qu'une classe de problèmes. De plus, elles ne peuvent s'exécuter que sur un seul type d'architecture et encore, ne permettent pas de partager directement leurs ressources avec les autres. C'est dans le cadre de cette problématique que Sun a créé le projet : *Project JXTA* [14].
Le but de JXTA est de créer une plateforme qui permette de construire, simplement et fa-

cilement, un grand ensemble de services et d'applications distribués pour tout dispositif qui pourrait être un pair. JXTA permet, entre autre, aux développeurs de se concentrer sur le développement de leurs applications, en créant des logiciels distribués flexibles, interopérables, disponibles pour tout pair de l'Internet.
Le concept de réseau P2P, tel que le voit JXTA, correspond à la définition 1.3.1 page 6, avec l'idée qu'il n'est pas nécessaire d'avoir forcément une notion de hiérarchie entre les pairs. « À certains égards, le monde du calcul P2P est *juxtaposé* au modèle hiérachique client-serveur » [15]. JXTA est fait pour des applications collaboratives et orientées communications.
Un des points clé de cette approche est l'ouverture. En effet, JXTA est fait pour que toutes ses applications puissent être exécutées sur n'importe quelles plateformes (PC de bureau, serveurs, PDA, téléphone GSM, etc.) connectées pouvant communiquer.
La philosophie de JXTA est de fournir des mécanismes de base et de laisser les choix d'implémentation pour l'application aux développeurs. Pour ce faire, JXTA s'appuie sur des standards, ouverts, tel que XML, Java [1], et sur les concepts qui font d'Unix un puissant et flexible système d'exploitation : shell avec commandes interopérables, utilisation de pipes pour accomplir des tâches complexes.
JXTA se décompose en 3 couches :

- *JXTA Core* : pour implémenter les services d'administration. Création et gestion de groupes de pairs, communication et sécurité. Les pipes sont les canaux de communication entre les pairs. Un message est envoyé à un pair par un pipe, sa structure est en XML et peut contenir des données, du contenu et/ou du code. L'intégrité, la confidentialité et la sécurité peuvent aussi être garanties par le pipe.
- *JXTA Services* : afin de faciliter le développement d'applications. C'est une librairie qui étend les fonctionnalités offertes par le core. Cette couche fournit des mécanismes pour rechercher, partager, indexer et mettre en cache du code et du contenu. Le mécanisme de recherche de contenu peut être distribué et/ou parallèle et utilise XML comme représentation des requêtes.
- *JXTA Applications* : construction de services ou d'applications pour les pairs. Cette couche repose sur les couche core et services.

JXTA Shell est un outil qui se situe à cheval sur les couches services et applications. Il permet aux développeurs ou aux utilisateurs avancés, d'interagir, de contrôler ou de tester l'environnement de pair. Cet outil fonctionne à la manière d'un shell Unix.
JXTA fournit les mécanismes de base indispensables à la création, l'implémentation d'environnements, d'applications distribuées, tout en fournissant la sécurité, l'interopérabilité pour des applications en milieux hétérogènes. De part son ouverture, JXTA est trop bas-niveau pour la mise en œuvre d'applications de calculs complexes. Mais cette ouverture et l'utilisation d'XML en font un puissant système qui peut s'adapter à tous les environnements (interopérabilité même avec des plateformes limitées en mémoire, en CPU).
Le shell fourni par JXTA est un puissant outil de développement, ainsi qu'un précieux allié pour le contrôle des applications.

1. JXTA est un standard, il peut être implémenté dans d'autres langages que Java, par exemple, en C, C++, etc.

2.2.2 Windows Peer-to-Peer Networking

Windows P2P Networking [16] est une plateforme pour les développeurs permettant de créer des applications P2P pour des ordinateurs fonctionnant sous Windows XP.
L'architecture de cette plateforme repose sur TCP/IPv6 comme couche transport, elle est composée de 5 principaux modules, que les applications Windows (Win32) peuvent utiliser pour accéder aux services et ressources offertes par le réseau P2P. Ces 5 modules sont :

- *Graphing* : permet de maintenir un ensemble de nœuds interconnectés, ainsi que de faire du flooding et de la réplication.
- *Grouping* : fournit la sécurité, au-dessus du graphe de nœuds. Il définit les modèles de sécurité pour la création de groupes, les invitations et les connexions à un groupe.
- *Name Service Provider* : est un mécanisme pour accéder à un nom arbitraire de fournisseurs de services.
- *P2P Name Resolution Protocol* : pour la résolution des noms des pairs.
- *Identity Manager* : création et gestion des identités des pairs.

Windows P2P Networking permet de mettre en œuvre rapidement un réseau P2P, un inconvénient de cette plateforme est le manque d'hétérogénéité des pairs, en effet, Windows P2P Networking ne marche qu'avec Windows XP.

2.2.3 Peer-to-Peer Grids

En considérant les cas extrêmes, on peut voir les Grilles comme des infrastructures permettant l'accès à des super-ordinateurs et à leurs ensembles de données. La technologie P2P est illustrée par Napster et Gnutella, qui sont des communautés ad-hoc partageant des fichiers sur de « petites » machines. Chacune de ces approches offre des services, mais elles diffèrent dans leurs fonctionnalités et le style de leur implémentation.
Une approche intéressante, qui a été traitée par [17], est d'essayer de combiner les deux approches Grille et P2P. Ce modèle explore le concept de *Peer-to-Peer Grids* (P2P Grids).
Afin de rendre interopérable leur architecture, P2P Grids a choisi d'utiliser un IDL (Interface Definition Language) à base d'XML afin de définir les objets distribués, ces IDL-XML sont des WSDL (Web Services Definition Language). L'environnement résultant est construit en termes de composition de *services*.
Les services contiennent de multiples propriétés, qui sont elles-mêmes des ressources individuelles. Ici, tout est ressource. Un service correspond à un programme informatique ou à un processus ; les ports (interface entre un canal de communication et un Web Service) à des appels de sous-routines avec des paramètres d'entrée et des données en résultat. Le tout communiquant avec un protocole à base de message. Typiquement les services peuvent, dynamiquement, migrer d'un nœud à l'autre. Les services sont ainsi éparpillés sur des ordinateurs à travers le monde.
L'idée est de faire un « mix » des services structurés (façon Grille) et des services dynamiques sans structures (façon P2P). L'architecture de P2P Grids est d'organiser les pairs en groupes locaux, le tout arrangé dans un système global supporté par des serveurs de cœur. Cette architecture permet, entre autres, d'utiliser des applications de collaborations à grande échelle, afin d'être plus tolérant aux pannes ; le système permet de faire de la collaboration asynchrone, tout en laissant la possibilité de faire du synchrone, malgré le fait qu'Internet ne permette pas de faire du temps réel.
P2P Grids a l'avantage d'allier au mieux les technologies de la Grille et du P2P et de les

rendre interopérables, grâce à la notion de services et l'utilisation des Web Services. À noter, que l'utilisation d'XML est bien moins performante que d'autres notations ou systèmes de messages, mais elle permet d'avoir des descriptions plus complexes.

2.3 Modèles de Programmation P2P

2.3.1 P2P-RPC

Le concept de *Remote Procedure Call* (RPC) [18] est utilisé depuis longtemps dans le domaine de la programmation répartie. Il founit un mécanisme simple de communication entre des composants distribués.
Dans le monde de la Grille, le modèle de programmation RPC a été standardisé par GridRPC [4] (voir section 2.1.1 page 9) pour permettre l'interopérabilité des architectures qui reposaient sur ce modèle (NetSolve [7], Ninf [8]).
Du fait que les systèmes P2P sont beaucoup utilisés pour le partage de fichiers, il existe assez peu de modèles de programmation de haut niveau, pour des applications de calcul.
Il existe, à notre connaissance, un seul modèle RPC pour les réseaux P2P : P2P-RPC[19]. Ce modèle est dérivé de OmniRPC [20], un environnement de programmation pour clusters et Grilles de calcul. OmniRPC alloue de façon automatique les appels dynamiques sur l'ordinateur distant approprié. Il est aussi possible de faire de la programmation parallèle (multi-threads) en permettant aux clients de soumettre des requêtes multiples sur plusieurs ordinateurs distants.
P2P-RPC a été implémenté comme une couche au-dessus d'XtremWeb (voir section 2.1.3 page 10) et en étendant OmniRPC. Ce dernier, étant dédié à la Grille, pour ce faire, l'environnement d'exécution d'OmniRPC a dû être changé radicalement. Dans OmniRPC, un client demande à un gestionnaire de ressources de lui envoyer une liste des nœuds associés à leurs services, c'est le client qui décide avec quel nœud il va communiquer en RPC. Au contraire dans P2P-RPC, le client demande une exécution RPC. Un coordinateur gère les *workers* afin de satisfaire la requête. Le client ne connaît pas la localisation du nœud qui a en charge le traitement de sa requête et récupère ainsi le résultat dans un temps fini.
Ce modèle permet de fournir trois nouvelles fonctionnalités : la tolérance aux fautes, l'asynchronisme et les communications non connectées.
Il est à regretter que cette approche ne permette pas les comminucations entre les pairs (ici des workers).
Ce modèle n'apporte pas de nouveaux concepts par rapport à GridRPC (voir section 2.1.1 page 9), même si, ici, il est appliqué à XtremWeb.

Voici un exemple d'appels de fonctions avec P2P-RPC :

```
main(String [] args){
  XWrpc.Init(args);

  // Appel synchrone
  XWrpc.Call(``function'',params,results[n]);

  // Appel asynchrone
  XWrpc.CallAsync(``function'',params,results[n]);
   ...
```

```
  // On attend le resultat
  XWrpc.WaitCallAsyncAll();
   ...
  XWrpc.Finalize();
}
```

2.3.2 Objets et Passage de Paramètres Paresseux

Afin de tirer partie au mieux des architectures P2P tout en facilitant le travail des programmeurs, ont été proposés des modèles de programmation de plus haut-niveau, reposant sur le paradigme objet.
Nous allons étudier [21] qui propose une abstraction pour l'interaction d'objets distants dans un environnement P2P.
Pour eux, les ressources partagées par les pairs sont des objets ; de ce fait, le réseau de pairs peut être vu comme un environnement réparti et orienté objet (objets distants). Il existe d'ailleurs une implémentation de ce modèle en Java 1.5 (langage « objet » qui offre la généricité et un mécanisme d'introspection) et *Remote Method Invocation* (RMI, communication client/serveur de type RPC).
Les pairs des environnements P2P sont extrêmement dymaniques, il n'est donc pas possible de prévoir l'apparition, la disparition et la durée de disponibilité de ces derniers. Ainsi, lorsqu'un pair A appelle une procédure sur un pair B, il est possible que lorsque B retourne le résultat à A, ce dernier ne soit plus disponible, ou bien que A, ayant besoin du résultat, B ne soit pas disponible.
Une des solutions à ce problème est l'utilisation de communications asynchrones et de boîtes aux lettres. RMI reposant sur des communications synchrones, le modèle propose l'utilisation de *Proxy* afin de permettre des communications asynchrones, indispensables à tout modèle P2P.
En plus de cela, le modèle fournit la possibilité de choisir entre le passage par référence ou par valeur des paramètres des appels de méthodes. Dans le cadre d'un réseau avec une faible bande passante ou avec des temps de latence importants, il est intéressant pour les performances de l'application, de ne pas à avoir à transférer systématiquement tous les paramètres d'une méthode à chaque appel ; surtout, dans le cadre d'applications scientifiques où ces paramètres sont souvent de tailles assez conséquentes. En effet, un paramètre peut déjà avoir été copié et s'il n'a pas été modifié, il n'est pas utile de payer le coût de son transfert une fois de plus. Ce système de passage des paramètres s'appelle *lazy parameter passing* [22]. Lors d'un passage par valeur, il est possible de choisir le protocole de transfert (FTP, HTTP, etc.).
L'utilisation du mécanisme d'introspection offert par Java donne au modèle la capacité de charger des ressources (des objets, ici) dynamiquement. En effet, un pair ne peut pas connaître tous les types de ressources (objets) offerts par les autres. En d'autres mots, l'introspection permet l'utilisation dynamique de types non connus à la compilation.
Les points forts de cette approche sont son haut niveau d'abstraction et son approche complètement décentralisée. Le modèle peut facilement passer à l'échelle, notamment, grâce à son système de passage des paramètres.

2.4 Programmation Parallèle

Un modèle de branch and bound est constitué d'un algorithme d'opérations de haut-niveau sur des données de type sous-problème, par exemple. Chaque opération définit une règle de base, comme *branch*.
Il existe notamment une approche orienté-objets de [23] générique pour le branch and bound. Cette approche est originale dans le sens où la résolution d'un problème se fait de façon séquentielle, tout en étant parallèle. En effet, le modèle prend en entrée un problème, on y applique des opérateurs et des types de données. Ensuite, le problème est outillé et exécuté dans un évaluateur, qui lui parallèlise le problème pour l'exécuter en parallèle. Cet évaluateur intéragit avec différentes librairies, afin d'être plus efficace avec certaines classes de problèmes. Leur implémentation du modèle repose principalement sur 3 objets :

- *Problème* : c'est un objet abstrait qui représente un nœud dans l'arbre de recherche des solutions. Cet objet est donc capable de trouver une solution ou d'être une étape vers la solution du problème. Il contient des opérations de création de sous-problème, etc.
- *Solution* : c'est aussi un objet abstrait. Il représente les solutions du problème. Il permet entre autre de comparer des solutions entre elles.
- *Solveur* : contrairement aux deux autres objets, lui n'est pas abstrait. C'est ici que la solution finale est produite par la manipulation des objets *Problème et Solution*.

On peut remarquer que *Problème* et *Solution* sont des objets abstraits. En effet, ils sont ainsi pour que l'utilisateur puisse les implémenter, afin de pouvoir résoudre son problème. En revanche, *Solveur* ne l'est pas, l'utilisateur ne peut pas le redéfinir. Le *Solveur* utilise toujours le même évaluateur pour résoudre les problèmes Branch & Bound.
Nous avons étudié une approche plus spécialisée dans le Divide & Conquer parallèle, Satin [24]. C'est une bibliothèque en Java, qui utilise un compilateur Java modifié. En effet, Satin offre simplement trois mots-clés supplémentaires au langage Java :

- *SATIN* : ce mot-clé se place devant la déclaration d'une méthode. Il indique que cette dernière doit être appelée avec le mot-clé SPAWN.
- *SPAWN* : placé devant un appel de méthode, il indique qu'il faut l'exécuter en parallèle (dans une nouvelle thread).
- *SYNC* : cette opération permet d'attendre que toutes les invocations faites avec SPAWN soient terminées.

L'approche de Satin est très interéssante de par sa simplicité mais, peu exploitable dans notre cas, car nous nous sommes interdits l'utilisation de pré-proccesseurs, de compilateurs Java modifiés ou l'utilisation d'une JVM personnalisée, afin de rester le plus portable possible. Cependant, Satin utilise un système de *Lazy parameter passing* très proche de celui décrit dans la section 2.3.2 page 15.

2.5 ProActive

Notre projet se place dans le cadre du travail de l'équipe OASIS. Son travail est la conception et l'implémentation de ProActive, une bibilothèque Java pour le parallélisme, la distribution et le calcul concurrent sur les Grilles.

2.5.1 Modèle de Programmation

ProActive repose sur un modèle MIMD (*Multiple Instructions, Multiple Data*) et sur la notion d'objet actif [25], l'objet possède une activité propre.
Le modèle de ProActive présente les caractéristiques suivantes :

- des objets actifs et accessibles à distance,
- la séquentialité des activités (processus purement séquentiels),
- une communication par appel de méthode standard,
- des appels asynchrones vers les objets actifs,
- un mécanisme d'attente par nécessité (futur transparent),
- les continuations automatiques (un mécanisme transparent de délégation),
- l'absence d'objets partagés,
- une programmation des activités qui est centralisée et explicite par défaut,
- du polymorphisme entre objets standards et objets actifs distants.

En l'absence d'extention syntaxique les programmeurs écrivent un code standard avec ProActive. La bibliothèque est elle-même extensible par les programmeurs, la conception du système est ouverte pour des adaptations et des optimisations.

2.5.2 Infrastructure

Afin de déployer les applications, ProActive fournit un mécanisme de nœuds virtuels (Virtual Node - VN -) où les objets actifs sont créés. Un VN est défini par une chaîne de caractères. L'association entre une JVM Java et un VN se fait par le biais de descripteur de déploiement XML (XML Deployment Descriptor). Les JVM sont elles-mêmes lancées sur des machines, il est possible d'utiliser plusieurs protocoles tel que ssh, Globus, LSF, rsh, etc. Après activation, un VN contient un ensemble de nœuds, s'exécutant dans un ensemble de JVMs. L'utilisation des VNs et des descripteurs de déploiement permettent une abstraction du code source : machines, création, recherche et protocoles d'enregistrement.
Une caractéristique importante de ProActive, que nous allons utiliser par la suite, est la notion de ProActiveRuntime auto-réferençable. Un ProActiveRuntime peut être vu comme une JVM déployée par ProActive pour une utilisation par ProActive. Ainsi, on peut référencer des JVM (ProActiveRuntime) entre elles ou dans un serveur.
ProActive offre la possibilité de choisir entre plusieurs protocoles pour la couche transport (RMI, SOAP, etc.) ; pour notre étude, cet aspect de l'infrastructure n'est pas important.

2.6 Synthèse des Systèmes Étudiés

Voir tableau 2.1 page 18 de synthèse.

Systèmes	Caractéristiques				
	Catégorie	**Plateforme**	**Langages et Outils**	**Fonctionnalités**	**Réseaux Cibles**
GridRPC	Grille	Toutes (les plus courantes)	RPC, OGSA, Globus, Java	Normalisation, hétérogénéité, grande échelle	Internet
InteGrade	Grille	Toutes (les plus courantes)	CORBA	Utilisations des cycles CPU inutilisés	Intranet [a]
XtremWeb	Grille	Infrastructure UNIX, pour les clients toutes (les plus courantes)	C	Grande échelle et asynchrone	Internet
JXTA	P2P	Toutes (les plus courantes)	Java, C, XML, etc.	Vraiment P2P et grande échelle, hétérogénéité	Internet
Windows P2P Networking	P2P	MS Windows XP	Win32	Windows	Internet
P2P Grids	Grille et P2P	Toutes (les plus courantes)	WSDL, XML	Meilleur du P2P et de la Grille	Internet
P2P-RPC	Modèle de Programmation	Toutes (les plus courantes)	XtremWeb et GridRPC	Grande échelle et asynchrone	Internet
Objects et Lazy Parameter Passing	Modèle de Programmation	Indépendant	Java	Passage des paramètres	Internet
Satin	Grille	Satin	Java et Satin	Divide & Conquer	Cluster
ProActive	Grille, env. //, etc.	Toutes (les plus courantes)	Java, XML	Asynchrone, Communications de groupe, hétérogénéité	Internet

TAB. 2.1 – *Synthèse des Systèmes Étudiés*

[a] Nous entendons par *Intranet*, l'utilisation de machines de bureau disponibles au sein d'une même organisation.

Chapitre 3

Une Infrastructure P2P à Auto Organisation Continue et Paramétrée

Dans ce chapitre, nous allons présenter et discuter de l'infrastructure Pair-à-Pair proposée. Cette infrastructure sera par la suite testée dans la section 5 page 41. Elle permettra aussi de masquer tout l'aspect découverte et auto organisation des pairs dans le réseau P2P ainsi créé, lors de l'utilisation de notre modèle de programmation (voir section 4.3 page 33).

3.1 Introduction

Nous avons un certain nombre de machines de bureau ou autres accessibles aux travers de réseaux d'entreprises ou d'organisations multi sites. Malheureusement ces machines sont, en générale, assez dynamiques et donc difficilement utilisables pour du partage de calcul. L'idée de notre infrastructure est d'essayer d'avoir un modèle très simple pour la création d'un réseau P2P, conforme à la définition 1.3.1 page 6, à partir de ces machines.
Par ailleurs, notre infrastructure s'appuyant sur la bibliothèque ProActive (voir section 2.5 page 16) nous ne nous préoccuperons pas de la couche communication. Nous pourrons utiliser les différents protocoles de communication fournis par ProActive notamment RMI, Ibis, les Services Web ou encore Jini[1].
Du point de vu de l'infrastructure les pairs sont des « nœuds supports » de calcul à venir. L'implémentation étant en Java, les pais sont en fait des JVMs. Notre objectif est la création d'un réseau de JVMs.

3.2 L'Infrastructure

3.2.1 Bootstrapping : Premier Contact

Les nouveaux pairs qui veulent rejoindre le réseau P2P créé sont confrontés au problème du bootstraping : « Comment rejoindre ce réseau? ».
Une solution à ce problème existe dans ProActive, c'est l'utilisation d'un protocole de communication spécifique qui est Jini.
Jini offre la découverte, la consultation et le rajout de services au sein d'un réseau [26]. Il est donc tout à fait envisageable d'utiliser Jini afin que les pairs d'un réseau P2P puissent

1. Jini n'est pas en soit un protocole de communication, mais d'enregistrement et de découverte.

se découvrir et se rejoindre de manière dynamique, transparente et autonome (sans contact préalable d'un serveur ou d'une liste de pairs connus à l'avance).
Il y a quand même un désavantage à l'utilisation de Jini. Le problème est que toutes ses fonctionnalités ne marchent qu'au sein d'un même sous-réseau, en broadcasting. Il n'est donc pas imaginable d'utiliser Jini si les pairs sont répartis sur Internet, ou sur plusieurs réseaux différents.
Afin, de résoudre le problème du bootstraping, nous avons choisi une solution simple qui consiste à ce que chaque pair, désirant rejoindre le réseau P2P, connaisse au moins un de ses membres. En effet, si je connais un membre de ce réseau, grâce à lui, je pourrai ensuite me connecter aux autres pairs et ainsi agrandir ma connaissance de ce système P2P.
Dans la pratique, nous avons utilisé une liste de pairs qui sont supposés être toujours ou assez souvent disponibles et constituant *de facto* le cœur de notre réseau P2P.
La figure 3.1 page 20 illustre notre solution ad-hoc pour le premier contact. Le *Nouveau Pair* qui veut rejoindre le *réseau P2P* pré-existant, essaie dans un premier temps de conctater les pairs qu'il connaît : *amda*, *nahuel*, *sakuraii* et *lo*. Dans notre exemple, il y a seulement 2 pairs de disponibles : *amda* et *sakuraii*. Cela suffit à intégrer le *Nouveau Pair* au *réseau P2P*.

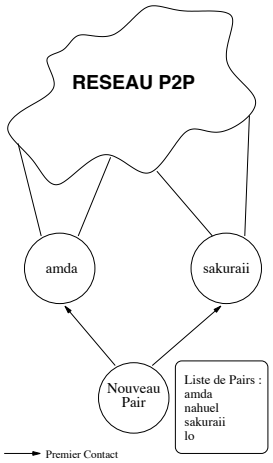


FIG. 3.1 – *Premier Contact.*

3.2.2 Overlay Networks de JVMs

Nous pourrions directement créer l'infrastructure au niveau des couches réseaux (IP, Sockets, etc.) ainsi, nous gagnerons en performances pour le réseau P2P et pour l'application. Mais notre objectif est de compenser ce manque d'optimisation par un gain de temps et de facilité dans la programmation d'application. Ainsi, ce n'est pas des CPUs, ou tous autre ressource physique, que nous partageons, mais directement des JVMs prêtent à accueillir des calculs. Nos pairs sont des JVMs.
L'infrastructure crée un réseau de pairs au-dessus d'un ou plusieurs réseaux existants, elle est donc une sorte d'*Overlay Networks*. Ainsi, les connexions entre les pairs sont des connexions virtuelles, indépendantes du réseau physique sous-jacent, comme peut le montrer la figure 3.2 page 21. Il est important que les réseaux P2P soient des overlay networks, car cela permet aux pairs de s'abstraire du réseau de communication, et ainsi permet de fédérer plusieurs réseaux

différents (IP, ad-hoc, etc.) entre eux. Ainsi, même si un des réseaux sous-jacents tombe en panne, le réseau P2P restera viable après la perte des pairs qui étaient dans cette partie du réseau physique.

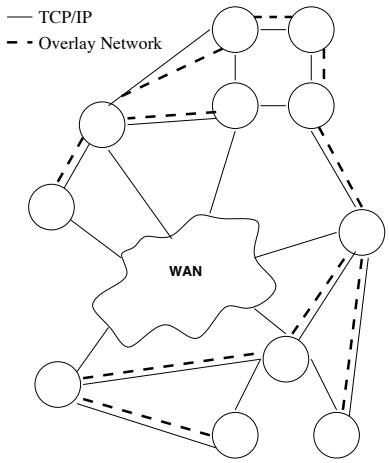


FIG. 3.2 – *Exemple d'overlay networks.*

Grâce à notre solution de bootstrapping, notre overlay network peut être, principalement, de 3 topologies différentes :

– *Client-Serveur* classique, comme le montre la figure 3.3 page 21.

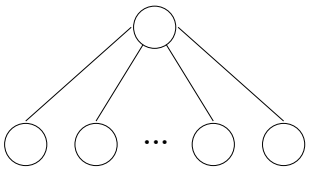


FIG. 3.3 – *Exemple de topologie client-serveur.*

– *Hierarchique* qui est une extension de la topologie client-serveur. Voir figure 3.4 page 22.
– *Sans organisation* c'est-à-dire de type graphe quelconque. Voir figure 3.5 page 22.

Il est tout à fait aisé pour un administrateur système d'organiser les pairs par le biais de notre protocole de bootstrapping, de manière à obtenir une topologie classique client-serveur ou encore hiérarchique selon ses besoins, pour ainsi créer son propre réseau P2P au-dessus de son réseau existant.
Afin de réaliser un overlay network et permettre le bootstrapping, les pairs doivent offrir la possibilité aux autres pairs de venir s'enregistrer auprès d'eux. Un pair possède ainsi une liste d'autres pairs qui sont venus s'enregistrer.
Les pairs doivent ainsi vérifier régulièrement qu'ils peuvent encore communiquer avec leurs voisins et maintenir le réseau P2P créé viable le plus longtemps possible.

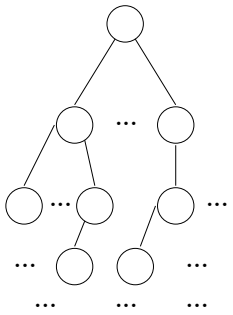


FIG. 3.4 – *Exemple de topologie hiérarchique.*

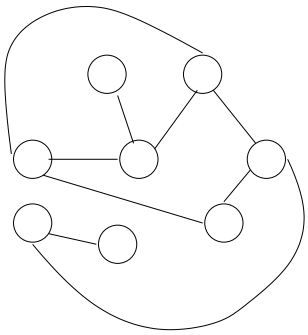


FIG. 3.5 – *Exemple de topologie sans organisation.*

3.2.3 Découverte et Auto Organisation Continue des Pairs

La particularité d'un réseau P2P est la dynamicité des pairs. En effet, un réseau P2P reposant sur un ou plusieurs réseaux sous-jacents, ces derniers pouvant être de différentes natures, il n'est donc pas posssible de faire des hypothèses sur la stabilité de ces réseaux. Les pairs sont effectivement de nature très différente : machine de bureau, ordinateur portable, serveur, etc., et de configurations différentes, certains peuvent avoir des systèmes moins stables que d'autres ou encore des accès aux réseaux de communication plus faibles en débit. En plus, de ne pas pouvoir connaître la disponibilité des pairs, il faut compter aussi sur un temps de latence entre les pairs très différent.

Auto Organisation Paramétrée

Nous entendons par *auto organisation* le fait de maintenir viable et utilisable un réseau de pairs. Malheureusement, l'infrastructure ne peut pas compter sur des entités extérieures ou d'avoir des serveurs dont l'objectif est la gestion du système, sinon elle ne pourrait pas être Pur P2P, le retrait d'une de ces entités entraînerait la panne du réseau P2P. Elle ne peut compter que sur elle-même.
Chaque pair, ou JVM, doit être serveur et client de l'infrastructure. Les clients venant s'enregistrer dans les serveurs, et les serveurs vérifiant que le client est toujours disponible. Ceci se résume en deux actions que chaque pair doit fournir, ces actions sont détaillées dans le tableau 3.1.

Actions	
Enregistrement	Cette action permet d'enregistrer un pair dans le pair courant. L'enregistrement peut être symétrique ou non. Dans le cas symétrique, le résultat de l'enregistrement est que les deux pairs se sont mutuellement enregistrés.
Désenregistrement	Permet de désenregistrer un pair au préalable connu, donc enregistré.

TAB. 3.1 – *Actions fournies par un pair.*

Avec ces deux actions, les pairs peuvent connaître d'autres pairs. Comme nous sommes dans un environnement volatile, il se peut que certains pairs, une fois enregistrés, ne soient plus disponibles et ne viennent pas se désenregistrer.
Il faut donc que chaque pair maintienne à jour régulièrement sa liste d'enregistrement, afin, de vérifier que ses connaissances soient toujours disponibles et joignables.
Le fait de se mettre à jour régulièrement ne résoud pas complètement un des problèmes liés à la dynamicité des pairs. Ce problème est qu'au bout d'un temps plus au moins long, les pairs peuvent se retrouver isolés des autres et briser le réseau P2P. Comme le montre la figure 3.6, A se retrouve isolé du reste du réseau, car B n'est plus disponible.

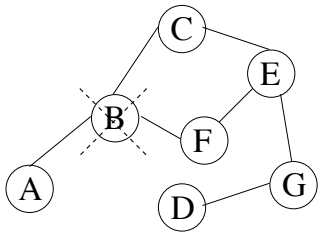


FIG. 3.6 – *Coupure au sein d'un réseau P2P.*

Notre solution, pour éviter l'émiettement, est qu'après la phase de bootstrapping, les pairs se constituent une *liste de connaissance*. C'est-à-dire, chaque pair essaie de connaître un certain nombre d'autres pairs, afin, de conserver le contact le plus longtemps possible avec le réseau P2P. Bien entendu, si tous les pairs que l'on connaît ne sont plus disponibles, on ne peut plus communiquer avec le réseau, on est coupé du reste du réseau. On considère qu'un pair n'est plus disponible, lorsque soit :

- il n'est pas revenu s'enregistrer pendant le Time To Update (TTU),
- notre tentative d'enregistrement au près de lui échoue.

Notre infrastructure a donc besoin de paramètres décrits dans le tableau 3.2.

Recherche Récursive de JVMs

Le but de l'infrastructure est d'offrir, grâce au partage, des JVMs (des pairs) pour du calcul. Un pair doit donc proposer l'action suivante : *obtenir pairs* qui retourne un certain nombre de pairs demandés par d'autres. Cette méthode est appelée de pair en pair de manière récursive jusqu'à obtenir le nombre nécessaire de JVMs.
L'utilisation de la topologie dite sans organisation à pour principale difficulté, qui est liée di-

Paramètres	
Time To Update (TTU)	C'est le laps de temps après lequel les pairs doivent mettre à jour leur table de connaissances, afin de supprimer les pairs qui ne sont plus disponibles.
Number Of Acquaintance (NOA)	Il est inutile qu'un pair connaisse le réseau P2P auquel il appartient, pour des raisons de performance, de nombres de messages, de mémoire, etc. Ce paramètre fixe donc la taille du nombre de connaissances qu'un pair doit s'efforcer d'obtenir, afin de ne pas être coupé du reste du réseau.
Time To Live (TTL)	Nombre de sauts entre les pairs, pour la recherche de ressources afin, de ne pas explorer le réseau P2P entièrement ou éviter les cycles.

TAB. 3.2 – *Paramètres du réseau P2P de JVMs.*

rectement à sa définition, la possible apparition de cycles en son sein, manque d'organisation. Ces cycles ne sont pas en soi un problème pour l'infrastructure, mais plutôt pour la recherche récursive de JVMs.
Une demande de pairs peut se retrouver bloquée dans un cycle ou encore explorer le réseau à l'infini.
La solution la plus simple que nous ayons trouvé à ce problème est l'introduction d'un *TTL* (Time To Live) en nombre de sauts entre les pairs. En effet, son utilité est double : tout d'abord il évite une exploration complète du réseau, ainsi que le problème des cycles entres les pairs qui bloqueraient la recherche de JVMs. Une fois ce TTL réduit à 0, les JVMs sont retournées au pair qui les a demandées, sans garantis de satisfaction de sa requête. C'est-à-dire qu'il se peut qu'il y ait moins de JVMs que ce qui a été demandé.
De plus, ces JVMs sont sans garantis d'être encore disponibles au moment où elles sont retournées au demandeur, mais, il y a quand même de fortes chances qu'elles le soient encore. Notre modèle de programmation solutionne ce problème.
La figure 3.7 page 24 résume toutes les actions et les paramètres d'un pair.

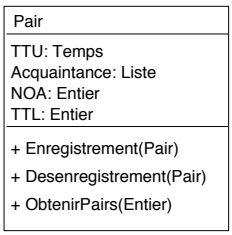


FIG. 3.7 – *Attributs et méthodes d'un pair.*

3.3 Implémentation avec ProActive

3.3.1 Les Ressources

Ce que nous entendons par ressources sont en fait des JVMs, ou plus particulièrement des ProactiveRuntimes. En effet, ProActive ayant besoin d'un environnement particulier pour s'exécuter, nous devons utiliser des *ProActiveRuntimes* qui sont des JVMs un peu particulières.
Un ProActiveRuntime est une JVM standard dans laquelle s'exécute un peu de code ProActive, afin de permettre d'abstraire la couche communication.
ProActive fournit tout un outillage pour l'acquisition et l'utilisation distante de ProActiveRuntimes. Il est aussi possible d'enregistrer les ProActiveRuntimes entre-eux.
Une JVM standard n'est pas outillée pour les communications avec d'autres JVMs. C'est pourquoi, nous nous appuyons sur les ProActiveRuntimes. Il n'est pas interdit de parler de JVM ProActive à la place de ProActiveRuntime.

3.3.2 Service P2P

Les JVMs ProActives fournissent une abstraction de la couche réseau et ainsi du réseau sous-jacent. Afin de créer un réseau P2P le plus étendu possible, il faut que chaque machine (PC de bureau, serveur, etc.) puisse le rejoindre dès sa mise en service. Nous avons donc introduit le concept de *Service P2P*. Tous les pairs potentiels doivent exécuter ce service implémenté par une JVM ProActive. Ce service permet, entre autre, la création et le maintien du réseau P2P, ou encore des JVMs ProActive pour y faire s'exécuter un bout d'une application de calcul.
Un service P2P est un démon Unix ou un service Windows, il se lance au démarrage de la machine. Notre démon est aussi persistant dans le sens où, si la JVM ProActive s'arrête, il en relance une.

3.3.3 Implémentation des Pairs

L'infrastructure est un réseau de JVMs P2P, les pairs sont donc des JVMs ProActive. Comme nous l'avons déjà dit ProActive repose sur la notion d'objet actif [25], c'est-à-dire un objet qui possède une activité propre. En Java, le concept d'objet actif peut être vu de manière vulgarisée comme un objet associé a une *Thread* qui contrôle son exécution.
Un objet actif peut être référencé à distance et a une queue de requêtes pour gérers ces appels distants.
Un pair, en plus d'être une JVM, est serveur avec une file d'attente (FIFO). Ce serveur sert des requêtes d'enregistrements de pairs distants, renvoyer des listes de pairs disponibles à d'autres pairs.
Notre idée de pair correspond bien au concept d'objet actif, à ceci près, qu'un pair doit aussi contrôler sa liste de connaissances. Un objet actif, dans ProActive, n'est pas *multi-threadé*. Nous avons donc séparé les activités d'un pair en deux parties :

- *P2PService* : cette première partie est la partie principale du pair. Elle consiste à servir toutes les requêtes d'enregistrement, de désenregistrement, ou encore de demande de JVMs. Biensûr le service P2P est implémenté avec un objet actif. Cet objet rejoint l'idé évoquée précédemment de Service P2P.

– *Updater* : contrairement à la partie service qui s'occupe principalement de servir des requêtes externes, celle-ci ne s'occupe que de mettre à jour, régulièrement (TTU), la liste des connaissances du pair. Il interagit avec le service auquel il est associé et avec les services des autres pairs qu'il connaît. Cette interaction se fait par le biais de requêtes. Il est à remarquer que l'updater ne sert pas de requête. En effet, les Updaters ne communiquent pas entre eux mais directement avec les P2PServices, on peut ainsi avoir différentes implémentations des pairs. Nous aurions pu utiliser une simple *Thread* de Java pour son implémentation, mais avec ProActive, il est assez difficile de faire cohabiter dans de bonne condition des *threads* et des objets actifs. C'est donc lui aussi un objet actif.

Ces deux entités partageant la même liste de connaissances, il en ressort des problèmes d'accès concurrents à cette dernière. La solution, la plus simple, est d'utiliser un troisième objet actif qui représente la liste de connaissances laquelle, par sa queue des requêtes, bloque tout accès concurrent.
La figure 3.8 page 26 montre une représentation de notre implémentation. Cette représentation a été produite avec un outil de ProActive, IC2D, qui permet de monitorer une application ProActive. Le premier cadre représente l'hôte sur lequel on se trouve, ici *sakuraii* ; le second cadre, la JVM ProActive avec son nom ; le troisième cadre est le nœud ProActive et les 3 ovales sont nos objets actifs.



FIG. 3.8 – *Représentation de l'implémentation d'un pair, grâce à l'outil IC2D.*

La figure 3.9 page 27 illustre plusieurs pairs communiquant entre eux, afin de maintenir le réseau P2P.

3.4 Conclusion

Nous avons réussi à réaliser une infrastructure la plus simple possible fournissant le minimun pour la création de réseaux P2P dédiés au partage de ressources de calcul. Ce n'est pas un réseau P2P, à proprement parler, que l'infrastructure fournit mais, plus des JVMs P2P.
Une remarque concernant notre travail pour réaliser cette infrastructure est que ProActive nous a permis l'implémenter, avec rapidité, de manière assez proche de sa définition.
Nous avons pu déployer cette infrastructure afin de la tester, comme nous allons le présenter dans les sections expérimentations 5 page 41.
Notre infrastrusture se démarque par plusieurs points :

– elle est vraiment *multi plateformes* : nous avons effectué des tests dans un même réseau

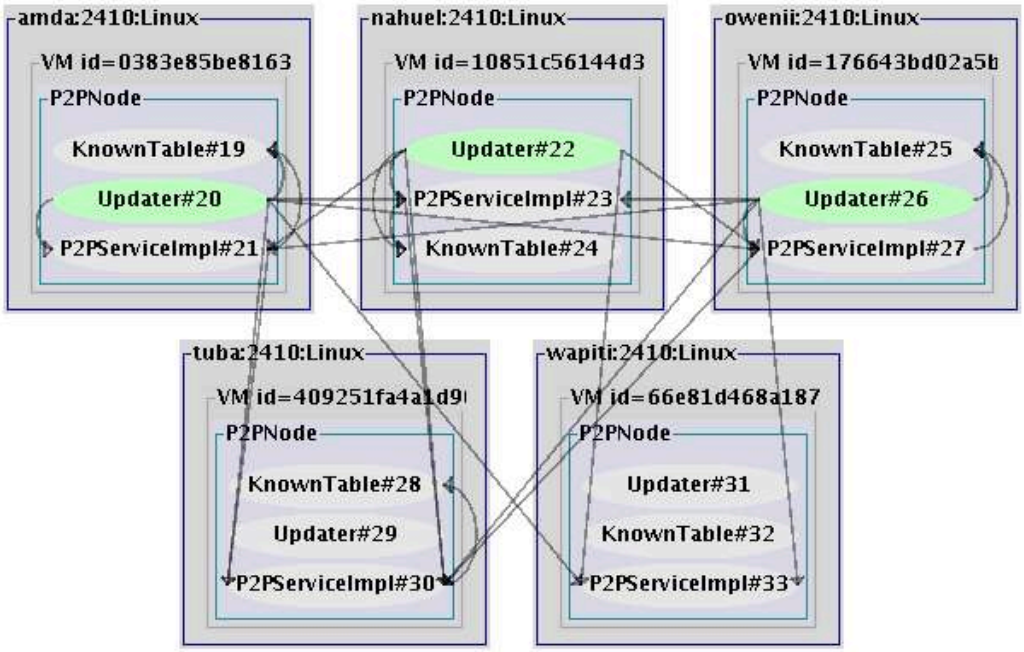


FIG. 3.9 – *Fonctionnement du réseau P2P.*

P2P avec des machines fonctionnant sous GNU/Linux et sous Windows. Grâce à Java, il est envisageable que cela fonctionne encore sur des architectures différentes.

- L'infrastructure est *paramétrable* : on peut régler le NOA et le TTU pour l'auto organisation, ainsi que le TTL pour la recherche récursive de pairs.
- De par sa conception, notre infrastructure respecte la définition 1.3.1 page 6 et est ainsi *« Pur Pair-à-Pair »*.

Chapitre 4

Modèle de Programmation et API Génériques

Dans un premier temps, nous verrons en détails le modèle de progammation que nous proposons. Ensuite, nous expliquerons comment nous avons traduit ce modèle en une API de programmation utilisant ProActive, facile d'utilisation pour la résolution de problèmes branch and bound. Dans la section expérimentations (chapitre 5 page 41), nous nous penchrons sur l'implémentation du problème du voyageur de commerce (TSP) et du problème des N-Queens.

4.1 Modèle de Programmation

Le modèle est un peu plus complexe, par rapport au modèle de [23], mais part du même principe, qui est de fournir un évaluateur standard et de laisser aux utilisateurs le soin d'implémenter leur problème de façon simple (sans se soucier du parallélisme). Dans notre cas, la chose est un peu plus complexe car, nous nous situons dans un environnement P2P dynamique. Notre modèle d'API de programmation s'appuit sur l'infrastructure P2P vu dans le chapitre 3 page 19. La figure 4.1 page 29 permet de mieux comprendre où il se situe vis-à-vis de ProActive et Java,et de voir qui intéragit avec qui.

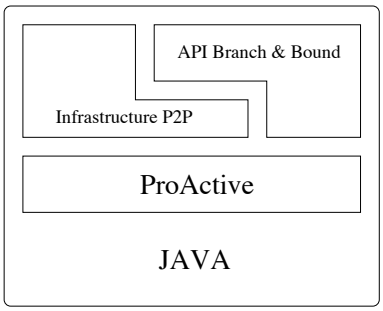


FIG. 4.1 – *Synthèse des différentes entités.*

4.1.1 Structure

Le modèle fournit un objet *Worker* qui permet la gestion et l'utilisation de l'infrastructure P2P sur laquelle notre modèle va s'exécuter.
Afin de toujours rester très simple dans notre approche, nous avons choisi de gérer l'infrastructure initialement comme une architecture maître-esclaves, puis de manière P2P avec des esclaves dynamiques.
Un premier Woker sert donc de maître (racine), c'est lui qui s'occupe de trouver par le biais de l'infrastructure P2P les premiers pairs pour le début de l'application . Ensuite, c'est lui qui s'occupe de récupérer le résultat final.
Le Worker a aussi un rôle d'escalve, ils sont au début créés par le Worker racine, par la suite, et en fonction des besoins (principalement la taille et la complexité du problème à résoudre) les Workers peuvent créer d'autres Workers dans un style réellement P2P. C'est eux qui vont s'occuper de résoudre le problème et les sous-problèmes.
Les rôles des Workers sont :

- lien entre le modèle de programmation et l'infrastructure P2P,
- topologie logique au niveau applicatif,
- gestion des erreurs et de la volatilité,
- fournit des inforamtions sur le nombre de machines disponibles.

Malheureusement, le Worker racine est le talon d'Achille de notre modèle. Pour que notre modèle puisse fonctionner, nous avons posé comme hypothèse le fait que le Worker racine s'occupe de récupérer et fusionner les derniers résultats. Si les Workers fils perdent le contact avec le Worker racine, alors une fois leurs calculs terminés, ils ne pourront les lui retourner, et tout sera perdu. Une solution envisageable est l'utilisation d'un système de boîte aux lettres dans les Workers. Ce mécanisme permettrait de conserver le ou les résultats jusqu'aux retours du Workers parents (ceux qui les ont créé).
Bien entendu, les Workers doivent gérer des Workers dynamiques. C'est-à-dire que si un Wroker fils tombe, alors le Worker parent de celui-ci devra en relancer un autre pour le remplacer. Une fois le Worker disponnible à nouveau, son père lui renvoit la tâche qu'il était en train de faire. Attention, le Worker recréé n'est pas forcement, et il y a peu de chances, sur le même pair qu'avant sa panne.
Comme le montre la figure 4.2 page 31, notre structure est très hiérarchique. La raison est qu'il est plus facile pour la conception du modèle de programmation d'avoir ce type de hiérachie proche du Branch & Bound.

4.1.2 Modèle de Programmation

Le modèle de programmation est une approche objet générique pour la résolution de problèmes.
Notre modèle repose donc sur 3 objets :

- *Solver* : c'est l'équivalent du Solveur du Generic Branch & Bound. Il s'occupe ainsi, de la solution finale et de la manipulation des 2 objets suivants. Il est associé au Worker racine, c'est par lui qu'il peut communiquer, se déployer sur l'infrastructure P2P. Le Solver est aussi abstrait ainsi, l'utilisateur peut définir un traitemant avant et après l'évalutation du problème, pour par exemple, effectuer un pré-découpage de ce dernier. Le Solver est décrit dans le tableau 4.1 page 31, permet à l'utilisateur de redéfinir seulement deux

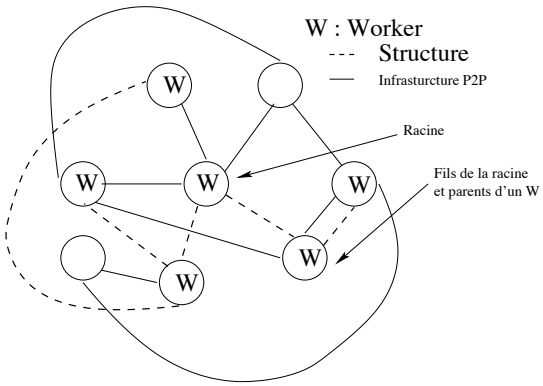


FIG. 4.2 – *Gestion de l'infrastructure P2P en hiérarchique.*

méthodes, *initialisation* et *terminaison*, qui sont exécutées avant et après le début de la résolution

Solver	
Initialisation()	Par exemple un prè-découpage du problème à résoudre.
Résoudre()	Pour lancer l'évaluation du problème.
Terminaison()	Éventuellement pour faire une dernière addition.

TAB. 4.1 – *L'objet Solver du modèle de programmation.*

- *Problem* : ici, aussi c'est un objet abstrait qui devra être implémenté par l'utilisateur. Il représente une étape ou un nœud vers la solution finale. Il n'est pas forcément le problème en entier mais, est aussi un sous-problème. Afin de pas faire de confusion entre cette entité et le problème à résoudre, nous utiliserons aussi le terme de *nœud*. Chaque Problem est associé à un Worker, c'est par les Workers qu'ils peuvent se diviser, communiquer, etc. Lors d'une sous-division c'est le Worker qui s'occupe de gérer les pannes. Le tableau 4.2 page 32 décrit les fonctionnalités d'un Problem. L'utilisateur devra redéfinir toutes ces méthodes. Toutefois, et contrairement au modèle Generic Branch & Bound, les méthodes de regroupements sont appelées de manière automatique par notre modèle. Ce n'est donc pas au programmeur d'inclure les appels à ces méthodes dans son code, ce qui simplifie grandement l'élaboration et l'écriture de Problem.
- *Result* : c'est une sous-solution ou la solution d'un problème ou des sous-problèmes. C'est un objet asbtrait et l'utilisateur doit définir comment comparer les résultats afin de choisir le plus pertinent. Un Result est associé à un Problem. Le Solver à aussi un Result qui à la fin de l'évaluation du problème est le résultat, la solution finale. Le tableau 4.3 page 32 décrit les fonctionnalités d'un Result que l'utlisateur aura à définir. En effet, elles sont dépendantes du problème à traiter.

Communications

Notre principal apport, par rapport à l'approche du Generic Branch & Bound, est la possibilité que les problèmes ont de communiquer entre eux. Nous avons définis 4 niveaux de communications :

- *Entre frères* : des Problems créés par le même nœud (le même parents).

Problem	
Calculer(Paramètre)	Calcule un Résultat pour les paramètres donnés.
Diviser()	Découpe ce Problème.
Diviser(n)	Fait n fois Diviser().
Rassembler(Problem[])	Attend la fin de tous les Problèmes en paramètre. Dans le but de retourner le meilleur des Résultats trouvés.
Entier Diviser?()	Test pour savoir si l'on doit appeler l'une des méthodes Diviser.
DonneRésultat()	Permet de voir où en est le calcul du Résultat, pour éventuellement, le sauvegarder.

TAB. 4.2 – *L'objet Problem du modèle de programmation.*

Result	
Afficher()	Permet d'afficher le résultat.
Comparer(Result)	Pour comparer ce Result avec un autre. Afin de déterminer le meilleur des deux résultats pour une fusion.

TAB. 4.3 – *L'objet Result du modèle de programmation.*

- *Avec le parent* : Le Problem envoie des informations uniquement au Problem qui l'a créé.
- *Avec ses enfants* : Le Problem envoie des informations aux Problems qui l'a créés (ses nœuds filles).
- *Avec tous le monde* : Tout les Problems reçoivent l'information.

Les communications se font toutes par le biais des Workers, qui sont associés aux Problems. Ce sont les Workers qui ont la connaissance de la topologie des Problems. Les Worker servent ainsi de boîte aux lettres pour les Problems. C'est donc au Problem de consulter régulièrement sa boîte sur son Worker. Un tel mécanisme de communication est utile, par exemple, dans le cadre des problème de Satisfaisabilité (SAT), les nœuds peuvent découvrir de nouvelles clauses pour résoudre le problème. Cette nouvelle clause, si elle est communiquée, pourra permettre aux autres nœuds de ne pas se fourvoyer dans de mauvaises pistes.

La figure 4.3 page 33 synthetise les relations entre les Workers, Problems, Solver et Results.

4.2 API de Programmation

Nous allons détailler l'implémentation de notre API de programmation. Solver, Problem et Result compose l'API de programmation. Nous verons aussi comment la relations entre l'API de programmation et l'infrastructure P2P ce fait grâce au Worker. un exemple d'implémentation du problème du TSP avec l'API sera proposée. Afin de montrer qu'il est possible d'écrire des applications de Branch & Bound en milieu P2P.
Bien entendu, comme pour l'infrastructure, cette implémentation se fait avec l'aide de la bibliothèque ProActive.

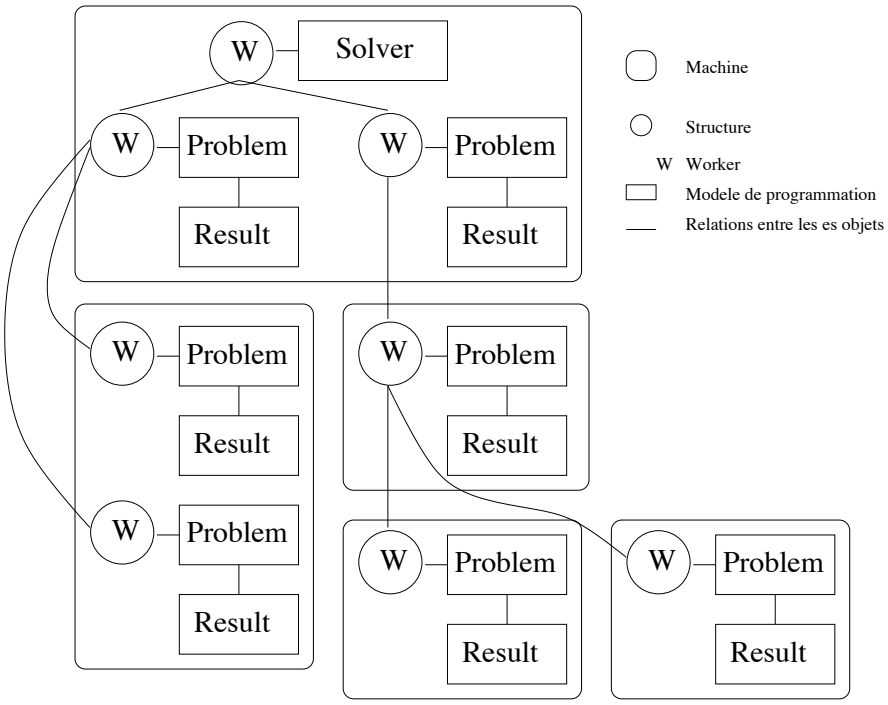


FIG. 4.3 – *Liens entre la structure et le modèle de programmation.*

4.2.1 Solver

Le Solver est le point d'entré et de sortie pour la résolution des problèmes. Il prend donc en entrée un problème et retourne en sortie la solution de ce problème, le résultat.
Comme expliqué précédemment, l'utilisateur peut définir deux opérations: Initialisation et Terminaison. Ainsi, Solver est une classe Java abstraite, dont les méthodes abstraites sont *Init()* (Initialisation) et *end()*. Le constructeur du Solver prend un Worker afin, de lancer la résolution du problème, le Solver fournit une méthode *solve()* qui prend en entré des paramètres, qui serviront aux Problems, et retourne un Résultat (Result).
Pour connecter notre Solver à l'infrastructure P2P, ce dernier devra prendre en paramètre de son constructeur un Worker. La création de plusieurs sous-problèmes, Problems, implique que la méthode init() prenne en entrée les parmètres du problème et retourne un tableau de Problems. Cette méthode est appelée en premier par la méthode solve(), une fois que tous les résultats sont revenus c'est la méthode end() qui s'occupe de les traiter et de retourner le résultat finale. Le tableau 4.4 page 34 décrit la signature de la classe Solver.

4.2.2 Problem

La classe Problem est l'implémentation de l'objet Problem de notre modèle de programmation. C'est aussi une classe abstraite que l'utilisateur aura en charge de compléter. Le constructeur ne sert qu'à instancier un nouveau Problem. À par les méthodes *getCurrentResult()* (DonneRésultat()) et *split(int n)* (Diviser(n)), toutes les méthodes sont abstraites. Afin, de pouvoir communiquer avec l'infrastructure P2P et le reste de l'application, Problem est associé à un Worker. Grâce au Worker, le Problem pourra communiquer avec les autres Pro-

```
public abstract class Solver {
    private Worker root = null;

    public Solver() {
        this.root = (Worker)
          ProActive.turnActive(new Worker());
    }

    public abstract Problem[] init(Object[][] params);

    public Result solve(Object[][] params) {
        Problem[] problems = init(params);

        this.root.getPeers(problems.length);
        // ...
        this.root.createDaughter(daugtherPbs);

        Result results = root.executeDaughter(params);

        return end(results);
    }

    public abstract Result end(Result result);
}
```

TAB. 4.4 – *Signature de la classe Solver.*

blems, c'est au programmeur de vérifier sur le Worker s'il n'y a pas de nouveau message pour le Problem. Il devra passer par le Worker, pour envoyer des informations aux autres Problems. Lorsque Problem a besoin de se diviser, il demande a son Worker s'il peut se diviser en n s'il fournit au Worker n Problems (ses fils), sinon cela dépend des choix de l'utilisateur. Le tableau 4.5 page 35 décrit la signature de la classe Problem.

La figure 4.4 page 36 montre le couple Problem - Worker, ainsi que les relations entre les Workers. Pour ne pas trop surcharger la figure nous n'avons pas reproduit les relations entre les Workers, Problems et P2PService à chaque niveau, un Worker seul dans une machine représente la même chose que la machine racine.

4.2.3 Result

Result est l'implémentation notre objet Résultat du modèle. C'est une classe abstraite puisse que l'utilisateur doit définir la comparaison entre deux Result afin, de déterminer le meilleur des deux : *isBetterThan(other)*. Result est en fait un pseudo-décorateur d'un Object qui est le véritable résultat. Pour permettre l'affichage de de ce résultat le programmeur devra redéfinir la méthode *toString()* de Java dans son véritable résultat. Le tableau 4.6 page 37 décrit la signature de la classe Result.

4.3 Implémentation

Nous allons ainsi décrire l'implémentation de notre structure de programmation, détaillée dans la section 4.1.1 page 30, Donc du Worker qui sert de point de jointure entre l'infrasturcture P2P et l'API de programmation.

```
public abstract class Problem implements Serializable {
    private Result result = null;
    protected Worker worker = null;

    public Problem(Worker worker) {
        this.worker = worker;
    }

    public abstract Result execute(Object[] params);

    // To split this task in a daughter task.
    public abstract Problem split();

    // To split this task in n daughters tasks.
    public Problem[] split(int n) {
        Problem[] problems = new Problem[n];
        for (int i = 0; i < n; i++)
            problems[i] = this.split();
        return problems;
    }

    // To merge all results from daughters tasks
    // when they finish their works.
    public abstract Result gather(Result[] results);

    // Return the number of sub-problem.
    public abstract int shouldSplit();

    public Result getCurrentResult() {
        return this.result;
    }
}
```

TAB. 4.5 – *Signature de la classe Problem.*

ProActive offre les communications de groupes typés [27]. Ce mécanisme permet la création et la gestion d'un groupe d'objets actifs. Les groupes vont nous permettre de gérer notre hiérarchie de Workers. Les Workers sont donc des objets actifs de ProActive et profite ainsi des autres puissants mécanismes de ProActive comme : les communications asynchrones entre les Workers, les continuations automatiques et la migration pour par exemple de l'équilibrage de charge.
Comme nous allons déjà un peu expliqué, les Workers sont associés à des Problems, afin de leur permettre de communiquer entre-eux, les Workers fournissent un système de communication par boîtes aux lettres que les Tasks doivent consulter régulièrement, c'est à l'utilisateur de définir ces moments dans la méthode execute(). Les Workers sont aussi outillés pour la gestion de la hiérarchie de Worker que nous avons détaillé. L'avantage d'avoir les Workers comme des objets actifs est qu'ils ont leurs propres activités, comme cela ils peuvent servir des requêtes de divisions du Problem, ou échanger des communications avec d'autres Workers. Le tableau 4.7 page 38 décrit la signature de la classe Worker.

4.4 Conclusion

La figure 4.5 page 39 synthétise le fonctionnement de l'API pour la résolution d'un problème.

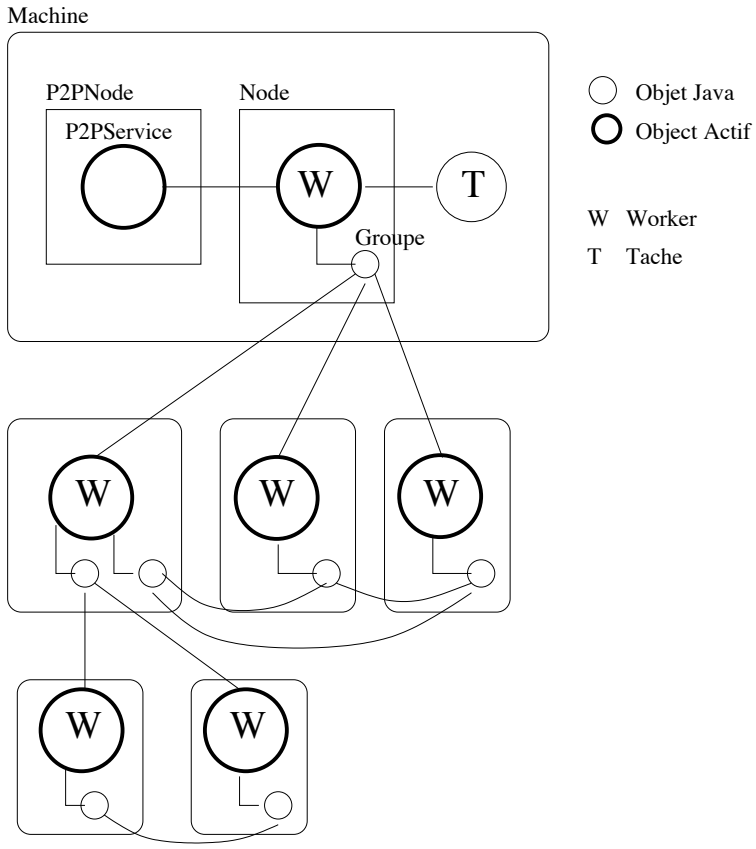


FIG. 4.4 – *Couple Problem - Worker ainsi, que les relations entres les Workers.*

Nous venons de décrire un modèle de programmation pour des applications de Branch & Bound s'exécutant en parallèle, au sein d'un système dynamique P2P. Nous verrons dans le chapitre relatant des expérimentations, que ce modèle a su très bien s'adapter à notre infrastructure dynamique.
De plus, ce modèle est très simple d'utilisation pour l'utilisateur qui peut écrire de façon très naturelle ses problèmes à résoudre. Notre modèle de par sa conception, division et regroupement, se prête extrêmement bien à tous les problèmes de type Divide & Conquer ainsi, que tous les problèmes qui ont besoins de communications entre leurs sous-problèmes.
Ainsi, notre modèle est assez général et permet de bien séparer la partie Branch & Bound de l'écriture du problème. Cette approche peut être ainsi considérée comme générique.
Malheureusement, pour l'instant, la suppression de pairs dans le réseau P2P est un frein à la résolution d'un problème. En effet, si chaque sous-problème demande un important temps pour être calculé, alors, lorsque le pair n'est plus disponible, son travail est perdu. Nous envisageons par la suite de trouver un remède à cette faille, en essayant de sauvegarder de temps en temps les solutions en cours de résolution afin qu'en cas de panne, on puisse repartir de la dernière sauvegarde et de manière dynamique.
Notre modèle peut être considéré comme « Pur P2P » grâce aux Workers. Par contre, une coupure du Worker racine (associé au Solver) ne permetterait pas aux autres Workers de lui retourner leurs solutions et le calcul serait perdu. Une solution serait de le répartir ou de le

```
public abstract class Result {
    private Object result = null;

    public Result(Object result) {
        this.result = result;
    }

    // Compare 2 results to choose the better
    public abstract boolean isBetterThan(Result other);

    // Get the real result
    public Object getResult() {
        return result;
    }

    // Modify the real result
    public void setResult(Object result) {
        this.result = result;
    }

    // Show the real result
    public String toString() {
        return this.result.toString();
    }
}
```

TAB. 4.6 – *Signature de la classe Result.*

copier entre les pairs.

```
public class Worker {

    public Worker(){ /* ... */}

    public Worker(Problem problem){ /* ... */}

    public Worker(Worker mother, Problem problem){ /* ... */}

    // Launch the associated Problem
    public Result execute(Object[] params){ /* ... */}

    // To reserve some Peers
    public int getPeers(int n){ /* ... */}

    // To cancel the peer book
    public void relaxPeers(){ /* ... */}

    // To create some new sub-Problems
    public void createDaughter(Problem[] problems){ /* ... */}

    // Start Daughter Problems, with gather on all
    // sub-results.
    public Result executeDaughter(Object[][] params){ /* ... */}

    // To send informations
    public void sendInfoToAll(Object info){ /* ... */}

    public void sendInfoToBrother(Object info){ /* ... */}

    public void sendInfoToDaughter(Object info){ /* ... */}

    public void sendInfoToMother(Object info){ /* ... */}

    public void receiveInfo(Object info){ /* ... */}

    // To check the info box
    public Vector getInfos(){ /* ... */}
}
```

TAB. 4.7 – *Signature de la classe Worker.*

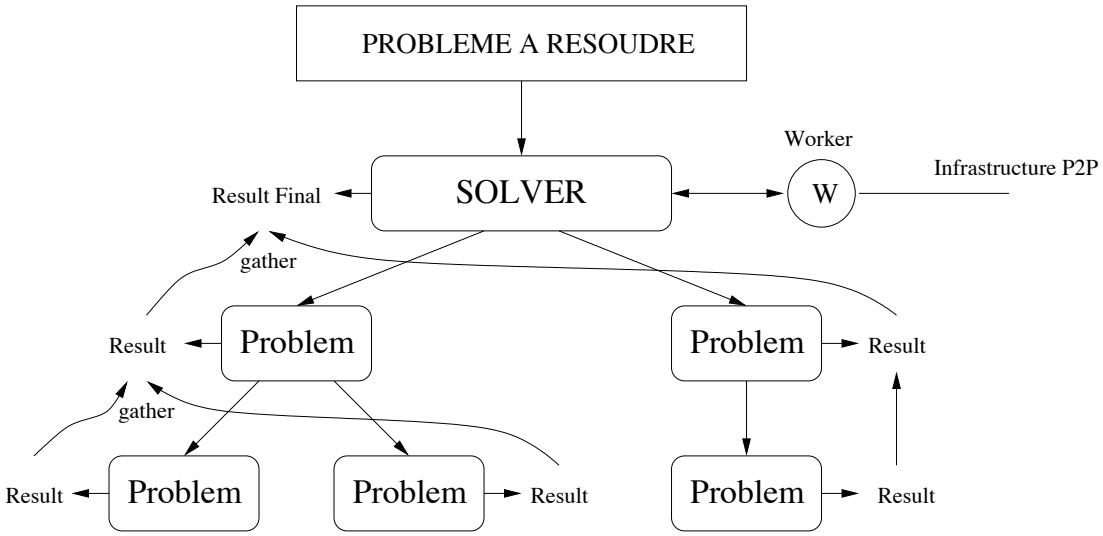


FIG. 4.5 – *Synthèse du fonctionnement de l'API.*

Chapitre 5

Validation et Expérimentations

5.1 Le Problème des N-Queens

Nous allons tester notre infrastructure avec cette application pour une raison pratique. La raison est que c'est la seule application à notre disposition écrite en ProActive qui fasse du maître-esclaves. Afin de tester uniquement l'acquisition dynamique de ressources (esclaves).

Le Problème

Le problème des N-Queens (ou des N-Reines) est de répondre à la question suivante : « Comment placer n reines sur un échiquier de $n \times n$ sans qu'elles soient en prises? »
Dans notre cas, nous nous intéresserons au nombre de solutions possibles pour n, c'est-à-dire non pas d'en trouver une, mais de les dénombrer. Par exemple, pour $n = 8$ il y a 92 solutions et pour $n = 21$ 314666222712 solutions.
C'est un problème NP-Complet, c'est-à-dire qu'il n'existe pas d'algorithme en temps polynomial pour le résoudre.

Sa Solution

Nous présentons, ici, une solution qui est très loin d'être optimisée pour résoudre ce problème. Ces avantages sont d'être 100% Java et écrite en ProActive.
Nous expliquerons que brièvement son fonctionnement, pour plus de détails se reporter à [1] ; le but étant d'avoir une application pour expérimenter notre infrastructure.
L'application est de type maître-esclaves et se compose de trois parties :

- *NQueensManager* : C'est le maître, c'est lui qui s'occupe de créer les workers sur les nœuds mis à sa disposition, de distribuer les tâches aux workers et de récupérer les résultats.
- *Worker* : C'est l'esclave. Il reçoit une tâche du NQueensManager, la calcule et retourne le résultat. Dès qu'il a fini son travail le NQueensManager lui renvoie une nouvelle tâche.
- *Task* : Le problème est découpé de manière statique en tâches. Puis ces tâches sont affectées à des workers libres.

On peut ainsi traiter en parallèle autant de tâches qu'il y a de workers.

5.1.1 Déploiement

Nos expérimentations se sont déroulées au sein de l'INRIA de Sophia Antipolis. Nous avons commencé avec les 24 machines de bureau de l'équipe OASIS. Par la suite, nous avons pu obtenir, pour un usage nocturne, l'accès à une partie des machines de bureau de l'institut, ce qui représente un potentiel de 130 machines à notre disposition.
Pour des raisons de sécurité (liés à l'institut) et au facilité d'expérimentation, nous n'avons eu accès qu'aux GNU/Linux de l'institut. Toute fois, nous avons réalisé des expériences avec une machine de bureau et un ordinateur portable fonctionnant sous MS Windows, afin de montrer l'utilisation d'un parc de machines hétérogènes. À notre grand regret, nous avons eu à notre disposition uniquement des machines d'architecture x86, bien que ces machines soient très disparates dans leur configuration matérielle (CPU, mémoire, etc.).
Comme nous venons de l'expliquer l'architecture de l'applications des N-Queens est de type maître-esclaves. Pour nos expérimentations nous n'avons eu qu'à changer son déploiement sans changer l'application.

5.1.1.1 Déploiement Statique → Dynamique

ProActive fournit à ses utilisateurs un mécanisme de descripteurs de déploiement [28]. Ces descripteurs sont des fichiers XML, ils permettent de séparer le code de l'application de son déploiement sur les machines. Ainsi, on peut programmer son application sans se soucier d'inclure dans son code les adresses des machines, etc.
L'application des N-Queens utilise ce mécanisme pour son déploiement. Malheureusement, l'utilisation des descripteurs, notamment au sein d'une institution pour l'utilisation de machines de bureau, nécessite la connaissance statique du parc de machines disponibles, et de spécifier individuellement sur quelles machines l'on veut créer une JVM puis un nœud. La rédaction de son descripteur de déploiement n'est donc pas une chose si facile, surtout que son format est en XML.
Notre première étape a été de remplacer l'utilisation du descripteur de déploiement par notre mécanisme de service P2P qui permet de rejoindre un réseau P2P et d'acquérir des JVMs.
Avant de lancer l'application, nous avons, au préalable, créé notre réseau P2P avec notre infrastructure. Dans cette première expérimentation, nous nous sommes contenté des 24 machines du projet OASIS. Nous avons commencé par lancer un service sur une machine, puis sur toutes les autres machines en leur demandant de s'enregistrer au-près de cette première machine, selon une topologie de départ clients-serveur.
Dans toutes nos expérimentations, nous n'avons utilisé qu'un seul protocole de communication : RMI.
Nous avons utilisé comme valeurs des paramètres du réseau P2P :

- *TTU* : 10 minutes, en effet, les machines de l'INRIA ne sont pas très dynamiques, elles sont assez stables et très rarement redémarées, elles sont diponibles pendant plusieurs jours sans problème . Nous avons choisi cette valeur de façon arbitraire. Mais dans un cadre plus dynamique, il est possible de choisir des valeurs plus faibles.
- *NOA* : 10 pairs. Cette valeur est largement trop grande, pour 24 machines, mais largement suffisante pour les 130 machines que nous aurons par la suite.
- *TTL* : 10 sauts, encore un choix arbitraire, mais qui nous a permis d'obtenir toutes les machines de l'INRIA.

Nous avons conservé ces valeurs pour les expérimentations. Tous les pairs ne sont pas obligés d'avoir tous les mêmes valeurs pour leurs paramètres. Il serait d'ailleurs judicieux de faire des tests à ce sujet.
Nous avons exécuté les services P2P en utilisant la commande *nice* d'Unix avec une priorité de 10. Afin, de ne pas gêner les membres du projet, car nous avons pu utiliser toutes les machines du projet 24h/24h.
Une fois l'infrastructure supposée déployée, l'application, commence par rejoindre le réseau P2P puis demande 30 ressources (OASIS a une trentaine de machines), ensuite elle découpe le problème en tâches, crée un nœud dans chacune des JVMs qu'elle a obtenue, instancie un *worker* sur chaque nœud et lance le calcul en envoyant des tâches aux workers, au fur et à mesure de leurs achèvements de calcul.
Le tableau 5.1 page 43 montre les résultats obtenus pour 20 et 21 Queens.

Queens	Workers (Nb. Machines)	Temps 1 CPU	Temps Réel	Nb. Solutions
20	23	73h7'22"	3h15'20"	39029188884
21	23	569h23'16"	24h59'44"	314666222712
23	23 à 130	-	depuis le 20/06	-

TAB. 5.1 – *Résultat avec 20 et 21 Queens.*

Une observation, que nous pouvons faire est que : $Temps\ 1\ CPU = Wokers * Temps\ Reel$.
- 20-Queens : $23 \times 3.25h = 74.75h$ soit pratiquement 73h, à deux heures près.
- 21-Queens : $23 \times 25h = 574h$ ce qui est assez proche de 569h.

$87 * 26 = 2262$ et pour 20 $1.5 * 24 = 36$. Cette remarque nous permet de vérifier que tous les workers ont travaillé de façon continue.
Ces premières expérimentations nous ont permis de constater qu'il était possible, et de façon très simple, de déployer des applications sur notre infrastructure P2P.

5.1.1.2 Application Statique → Dynamique

Dans la partie précédente, une fois l'application deployée, son exécution du point de vue acquision de nouveaux workers disponibles dans le réseau P2P est statique. En effet, une fois la première acquisition de ressources effectuée, l'application ne cherche plus de nouvelles ressources ; si un worker s'arrête, elle le perd pour toujours.
Dans cette phase de l'expérimentation, nous avons cherché à montrer que notre infrastructure permet l'acquisition dynamique de nouvelles ressources dynamiques. Pour ce faire, nous avons un peu modifié le *NQueensManager*. Afin de demander un certain nombre de ressources à son démarrage (toujours 30, ici), lorsque que tous les workers seront occupés le manager demandera au réseau P2P de nouvelles ressources.
Pour rendre notre test le plus réaliste possible, nous avons décidé de créer un premier réseau P2P avec les machines du projet OASIS, ce réseau marchant 24h/24h. Le soir, nous avons accès aux autres machines de l'INRIA, donc à partir de 22h00 chaque soir, nous avons une centaine machines supplémentaires qui viennent rejoindre notre réseau P2P. Ces machines sont à notre disposition jusqu'à 7h00 le lendemain matin. En gros, le réseau P2P que nous avons est constitué de 20 machines le jour et 120 machines le soir.
Afin de pousser notre expérimentation, nous avons lancé le calcul de 23-Queens, qui à l'heure de la rédaction de ce rapport tourne toujours. Notre observation, montre que l'application

n'est pas dérangée par l'acquisition dynamique des nœuds et qu'elle profite pleinement des 20 machines la journé et des 120 le soir. Nos services s'exécutant sur les machines du soir sont tué au petit matin ainsi, nous perdons la centaines de tâches qui était en cours. Ces tâches seront relancé plus tard par le NQueensManager.

5.1.2 Conclusion

Nos expérimentations nous ont permis de valider notre infrastructure, au moins pour son fonctionnement au sein d'une organisation ou d'une entreprise, en utilisant les cycles CPU des machines de bureau inutilisée durant les soirées. Nous avons l'intention de remplacer le déploiement statique par une acquisition dynamique des nœuds dans la plupart des applications ProActive, notamment le portage de l'application Jem3D [29], une application de calcul d'électromagnétisme qui pour l'instant ne peut être déployée que de manière statique (et ce sur environ 300 processeurs) au sein de l'INRIA.

5.2 Exemple TSP

Un voyageur de commerce (Traveling Salesman Problem - TSP -) doit visiter n villes données en passant par chaque ville exactement une fois. Il commence par une ville donnée et termine en retournant à la ville départ. Les distances entre les villes sont connues. Le problème du TSP est de trouver un chemin pour minimiser la distance parcourue.
Notre approche pour résoudre le TSP est la plus simple possible, afin de montrer que notre modèle de programmation est simple à mettre en œuvre. Bien-sûr la solution que nous allons proposer n'est pas la meilleure, mais elle a pour seul avantage d'illustrer l'utilisation de l'API de programmation.

5.2.1 Solver

Le Solver de TSP prend en entrée une matrice donnée de taille $n \times n$ qui représente les distances entre les villes. Par définition une distance est positive, nous avons pris comme convention qu'une distance négative entre deux villes représente l'absence de chemin. Nous vérifions aussi que le graphe représenté par la matrice soit connexe. La ville de départ est donnée.
Notre algorithme est exact, c'est-à-dire qu'il va retourner la meilleure solution pour le graphe donné explorant tous les chemins.
Le découpage en sous tâches parallèles est réalisé de la façon suivante. Pour un sommet de degré deg, si ce sommet a un degré $\geq deg$ alors l'algorithme créé un sous-problème pour chaque voisin qui va chercher l'itinéraire le plus courts, à partir de son sommet. Chacun de ces sous-problèmes va faire de même pour ses voisins si son degré est $\geq deg$. Si un sous-probème a un degré $\leq deg$, il fait un parcours en profondeur et en séquentiel de son sous-graphe afin de chercher le meilleur itinéraire. À la fin de chaque sous-problème, le sous-problème en question envoie son meilleur résultat à tous les autres. À chaque nouvelle meilleure solution trouvée, la valeur est envoyée à tous, par notre mécanisme de communications entre les Workers, afin d'accélerer la recherche en évitant plus tôt les explorations inutiles.
Le tableau 5.2 montre le code du Solver pour TSP.

```
public class SolverTSP extends Solver {

    public Problem[] init(Object[][] params) {
        // params[0] number of cities
        int [][] matrix = params[0];
        int startCity = params[1];
        // params[2] min deg to split
        return createFirstProblems(startCity, matrix, params[2]);
    }

    public Result end(Result result) {
        return result;
    }
}
```

TAB. 5.2 – *Exemple de la classe Solver pour TSP.*

5.2.2 Problem

Le tableau 5.3 montre une implémentation possible de la classe Problem de l'API, pour résoudre le TSP.

5.2.3 Résultats

Expérimentations en cours.

```
public class ProblemTSP extends Problem {
    int currentCity, minDeg; int[][] matrix;

    public ProblemTSP(int[][] cityDistance, int minDeg) {
        super();
        this.matrix = cityDistance;
        this.minDeg = minDeg;
    }

    public Result execute(Object[] params) {
        this.currentCity = params[0];
        // the path to go from start city to this one
        Itinary itinary = params[1];
        int split = this.shouldSplit();
        if (this.shouldSplit() == 0){
           // Explore the sub-graph
           while(/* have next vertex */){
             // Calcul the distance of the itaniry with the
             // current vertex
             Result currentItinary = current.explore();

             if (currentItinary.isBetterThanMe(best())){
                // New better Solution found
                setBest(currentItinary);
                // Notify all of better solution found
                this.worker.sendInfoToAll(currentItinary);
             }
           }
           // Return the best itinary
           return best;
         } else {    // Split
           int peers = this.worker.getPeers(split);
           this.worker.createDaughter(createSubPbs(split(peers)));
           Result result = this.worker
               .executeDaughter(createParamsSubPbs(peers));
           return result;
        }
    }

    public Problem split() {
        return new ProblemTSP(this.matrix, minDeg);
    }

    public Result gather(Result[] results) {
        return shortestItinary(results);
    }

    public int shouldSplit() {
        int deg = neighbours(currentCity);
        if (deg >= this.minDeg) return deg; // Split
        else return 0; // No split
    }
}
```

TAB. 5.3 – *Exemple de la classe Problem pour TSP.*

Chapitre 6

Conclusion et Perspectives

Le travail réalisé pendant ce DEA comporte deux grandes parties complémentaires:

- une infrastructure de noeuds (JVMs) P2P pour le calcul parallèle,
- un modèle de programmation avec API pour la programmation et la résolution de problèmes en mode P2P.

L'implémentation de la seconde partie repose sur la première. Deux applications nous ont permis la réalisation de premiers tests d'expressivité et d'efficacité. À notre connaissance, l'ensemble constitue la première infrastructure permettant une programmation P2P avec communication entre tâches parallèles ; cela ouvre la voie à la résolution P2P de problèmes non « agréablement parallèles ». L'infrastructure comporte quelques points clés :

- *auto-organisation* : l'infrastructure s'organise dynamiquement et en continu en noeuds de calcul inter-connectés,
- *paramétrisation* :
  - Time To Live (TTL), nombre de sauts entre pairs pour la recherche de ressources,
  - Number Of Acquaintance (NOA), nombre de connaissances qu'un pair doit s'efforcer d'obtenir,
  - Time To Update (TTU), durée pour la mise à jour des connaissances.

Globalement, le système peut être qualifié de *pur P2P*, dans la mesure où aucune machine ne joue le rôle statiquement défini et immuable de serveur. La partie programmation se veut facile à utiliser et flexible. Elle comporte aussi certaines originalités importantes :

- *dynamicité des sous-tâches* : les Workers ont la possibilité de créer dynamiquement d'autres Workers,
- lien et intéraction avec l'infrastructure P2P sous-jacente

Ces deux éléments autorisent une interaction forte et dynamique entre les ressources disponibles et le taux de parallélisme créé pour l'application. Ceci permet un contrôle dynamique de la granularité en mode P2P, élément indispensable à une programmation répartie et parallèle efficace.

Outre la validation sur des exemples conséquents, une extension future de notre travail serait l'intégration de paramétres de recherche [30] de type DFS ou BFS comme paramètres du modèle de programmation. Il serait également intéressant d'essayer de modéliser notre infrastructure P2P comme un réseau de Kelly, de Jackson ou autre. Ces modélisations pourraient permettre d'évaluer les performances théoriques de l'infrastructure et du modèle de programmation. Nous pourrions aussi trouver des relations entre les différents paramètres et aider à la détermination de valeurs efficaces.

Bibliographie

[1] ProActive. Inria, 1999. http://www-sop.inria.fr/oasis/ProActive.

[2] Rudiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In IEEE, editor, *2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Department of Computer and Information Science Linko pings Universitet, Sweden, august 2001.

[3] Seti@home, 2004. http://setiathome.ssl.berkeley.edu.

[4] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. *GridRPC: A Remote Procedure Call API for Grid Computing.* Edinburgh, Scotland, July 2002.

[5] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[6] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Matsuoka, and Satoshi Sekiguchi. Evaluating web services based implementations of gridrpc. In *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 2002), 23-26 July 2002, Edinburgh, Scotland, UK*, pages 237–245. IEEE Computer Society, 2002.

[7] Keith Seymoour Sudesh Agrawal, Jack Dongarra and Sathish Vadhiyar. *NetSolve: Past, Present, and Future - A Look at a Grid Enabled Server*, chapter 24, pages 613–622. 2003. Making the Global Infrastructure a Reality, Berman, F., Fox, G., Hey, A.

[8] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for a global world-wide computing infrastracture. In *HPCN'97 (LNCS-1225)*, pages 491–502, 1997.

[9] Globus grid project. http://www.globus.org.

[10] Alfredo Goldman Andrei Goldchleger, Fabio Kon and Marcelo Finger. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. In *ACM/IFIP/USENIX International Workshop on Middleware for Grid Computing*, Rio de Janeiro, February 2003.

[11] Vincent Néri Gilles Fedak, Cécile Germain and Franck Cappello. Xtremweb : A generic global computing system. In *CCGRID2001, workshop on Global Computing on Personal Devices*, May 2001.

[12] The free network project. http://freenet.sourceforge.net/.

[13] Gnutella. http://www.gnutella.com/.

[14] Project JXTA. http://www.jxta.org/.

[15] Inc. Sun Microsystems. Project jxta: An open, innovative collaboration, April 2001. http:// www.jxta.org/project/www/docs/.

[16] Microsoft Corporation. Introduction to windows peer-to-peer networking. Technical report, January 2003.

[17] Geoffrey Fox, Dennis Gannon, Sung-Hoon Ko, Sangmi Lee, Shrideep Pallickara, Marlon Pierce, Xiaohong Qiu, Xi Rao, Ahmet Uyar, Minjun Wang, and Wenjun Wu. Peer-to-peer grids. http://grids.ucs.indiana.edu/ptliupages/publications/.

[18] Sun Microsystems Inc. Rpc: Remote procedure call protocol specification version 2. Technical report, In Tech. Rept. DARPA-Internet RFC 1057, SUN Microsystems, Inc., June 1998.

[19] Samir Djilali. P2p-rpc: Programming scientific applications on peer-to-peer systems with remote procedure call. In IEEE, editor, *3rd International Symposium on Cluster Computing and the Grid*, pages 406–413, May 12 - 15 2003.

[20] Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, and Satoshi Sekiguchi. OmniRPC: A Grid RPC facility for cluster and global computing in OpenMP. In *Lecture Notes in Computer Science*, volume 2104, pages 130–135, July 2001.

[21] Patrick Thomas Eugster and Sebastien Baehni. Abstracting remote object interaction in a peer-2-peer environment. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 46–55. ACM Press, 2002.

[22] P. Eugster. Lazy parameter passing. Technical Reports in Computer and Communication Sciences, 2004.

[23] A. de Bruin, G.A.P. Kindervater, H.W.J.M. Trienekens, R.A. van der Goot, and W. van Ginkel. An object oriented approach to generic branch and bound. EUR-FEW-CS-96-10, july 1996.

[24] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Proc. Euro-PAR 2000*, number 1900 in Lecture Notes in Computer Science, pages 690–699, Munich, Germany, August 2000. Springer.

[25] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd., September-November 1998. http://www-sop.inria.fr/oasis/proactive/.

[26] *The Jini Specifications, Second Edition.* Addison-Wesley, 2000.

[27] Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Joint ACM Java Grande - ISCOPE Conference*, pages 28–36, Seattle, 2002. ACM Press.

[28] Francoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssiere. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[29] Laurent Baduel, Françoise Baude, Denis Caromel, Christian Delbé, Saïd El Kasmi, Nicolas Gama, and Stéphane Lanteri. A parallel object-oriented application for 3D electromagnetism. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[30] Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, January/February 1999.