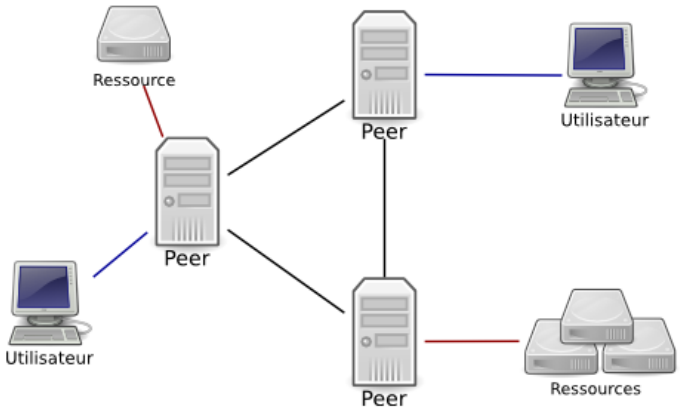
UNIVERSITÉ de Liège

Implémentation d'un Grid Peer-to-Peer



Travail de fin d'études
Licence en Informatique
Université de Liège

Par
Xavier Dalem
Juin 2007

Sous la supervision
du Professeur P.-A. de Marneffe

Remerciements

Je tiens à remercier...

Le Professeur Pierre-Arnoul de Marneffe, Département Montefiore, Université de Liège, pour m'avoir permis d'effectuer ce travail dans son service.

Mr Cyril Briquet, Département Montefiore, Université de Liège, de m'avoir fait découvrir les concepts du calcul en grille ainsi que pour ses conseils et nombreuses relectures de ce travail.

Table des matières

Remerciements **i**

Table des matières **ii**

Table des figures **iii**

Liste des tableaux **iv**

Listings **v**

Résumé **vi**

1 Objectif du travail **1**

2 Qu'est-ce qu'une grille ? **3**

2.1 Définition générale 3

2.2 Structure d'une grille 4

2.3 Politiques de partage 6

2.3.1 Philanthropisme 7

2.3.2 Mutualisme 8

2.3.3 Individualisme 8

2.4 La grille selon OurGrid 9

2.4.1 Comptabilisation de la consommation de ressources 10

3 Architecture existante **12**

3.1 Tâches 13

3.2 Peers 14

3.3 Ressources 15

3.4 Utilisateurs 16

3.5 Simulateur 16

3.6 Autres classes 17

4 Implémentation des classes Live **18**

4.1 Structure des tâches 19

4.2 Protocole d'appels distants 21

4.2.1 Émission d'un appel distant 22

4.2.2 Réception et traitement des appels 23

4.3 Peers 25

4.4 Ressources 26

4.4.1 Exécution des tâches 27

4.5 Utilisateurs 29

4.6 Lanceurs 30

4.7 Interface pour les applications de grille 32

4.7.1 Simulation en mode Live 33

5 Gestion des données **34**

5.1 Différents types de données 35

5.1.1 InlineGridData 36

5.1.2 FTPGridData 37

5.1.3 TorrentGridData 38

5.1.4 Utilisation conjointe des différents types de données 42
5.2 Le gestionnaire de données 43
5.2.1 Téléchargement 43
5.2.2 Partage 48
5.2.3 Conversion 50
5.2.4 Politique de choix des données et données multiples 51
5.3 Chemin des données 53

6 Découverte de peers **56**
6.1 Implémentation 56
6.1.1 Serveur 58
6.1.2 Client 60
6.2 Limites 62

7 Gestionnaire de fichier journal **63**

8 Utilisation de la grille **65**
8.1 Mise en place du middleware 65
8.1.1 Compilation 66
8.1.2 Lancement des nœuds 66
8.2 Soumission de tâches à la grille 71
8.2.1 Écriture d'une tâche 71
8.2.2 Lancement d'un utilisateur pour soumission 75

9 Expériences **77**
9.1 Soumission automatique 77
9.2 Configuration de test 78
9.3 Performances des protocoles selon le SDR 79

9.4 Performances des protocoles selon la taille des données 81
9.5 Utilité du cache 81

10 Développements futurs **84**
10.1 Persistance des données 84
10.2 Sécurité 85
10.2.1 Cryptographie 85
10.2.2 Protection des ressources 86
10.2.3 Protection des tâches 87
10.2.4 Protection du réseau 88
10.3 Résistance aux facteurs d'échelle 89
10.3.1 Gestion des sockets 89
10.3.2 Découverte de peers 90
10.4 Accessibilité 90

11 Conclusion **92**

Annexe : Code source **94**

Bibliographie **95**

Table des figures

2.1 Vue simplifiée d'une grille 7

3.1 Modèle de communication par services et handles 15
3.2 Modèle de communication par services et handles (détail) 17

4.1 Information liée à un TaskDescriptor 20
4.2 Structure des Jobs 20
4.3 Transfert des tâches 22

5.1 Transferts théoriquement possibles 36
5.2 Cache sur le GridDataManager 47
5.3 Chemin des données 54

6.1 Moteur de recherche 57

9.1 Performances des protocoles selon le SDR 80
9.2 Performances des protocoles selon la taille des données 82
9.3 Utilité du cache 83

10.1 Moteurs de recherche à intersection non vide 91

Liste des tableaux

3.1 Description des packages dans l'architecture existante 13

5.1 Structure des options de téléchargement de GridData 51

8.1 Options de compilation Ant 66

Listings

4.1 Signature du constructeur de RemoteCall 22
4.2 Création d'un RemoteCall 25
4.3 Vérification de la valeur de retour 25
4.4 Un exemple d'appel distant (LivePeerHandle) 26
7.1 Un exemple de configuration de journalisation 64
8.1 Déploiement du moteur de recherche 67
8.2 Déploiement d'un peer 67
8.3 Configuration d'un peer : contact 67
8.4 Configuration d'un peer : données 68
8.5 Configuration d'un peer : moteur de recherche 68
8.6 Configuration d'un peer : comportement 68
8.7 Déploiement d'une ressource 69
8.8 Configuration d'une ressource 69
8.9 Déploiement d'un parc de ressources 70
8.10 Écriture d'une tâche : implémentation de GridApplication 71
8.11 Écriture d'une tâche : setParameters() 71
8.12 Écriture d'une tâche : setInputData() 72
8.13 Écriture d'une tâche : compute() 72
8.14 Écriture d'une tâche : getResult() 73
8.15 Écriture d'une tâche : GridHasher 73

8.16 Soumission d'une tâche : écriture du descripteur 75
8.17 Soumission d'une tâche : lancement de l'utilisateur 76

Résumé

À une époque où de plus en plus d'ordinateurs sont connectés en réseau, le *Grid Computing* apporte une solution au problème de l'optimisation de l'utilisation de la puissance de calcul accessible à travers un réseau. Dans ce contexte, le *Grid Peer-to-Peer* représente le cas où la confiance entre les acteurs de la grille est minimale, typiquement sur un réseau public comme Internet.

Le but de ce travail est de développer un middleware, c'est-à-dire de fournir une implémentation réseau à une bibliothèque de *Grid Peer-to-Peer* déjà utilisée dans un simulateur. L'implémentation utilisera des protocoles de transfert de données existants, BitTorrent et FTP. L'implémentation devra aussi proposer un système basique de découverte de peers.

Chapitre 1

Objectif du travail

L'objectif de ce travail est de créer une implémentation réelle d'un Grid Peer-to-Peer sur base de composants déjà fonctionnels en simulation. Une fois mise en place sur un réseau, l'implémentation doit d'abord permettre de soumettre à la grille des tâches de calcul écrites dans le langage Java. Cette soumission s'effectuera à travers des appels réseaux.

Le déploiement de l'implémentation doit être rapide, et adapter des programmes existants ne dépendant pas d'un facteur humain durant leur exécution ne doit pas être contraignant.

Afin de permettre l'échange des données dont les tâches auraient besoin, un système de transfert de données basé sur des protocoles existants doit aussi être développé.

Une grille étant susceptible d'évoluer fortement au cours du temps, le travail devra proposer un système basique de découverte de peers mais fonctionnel dans un environnement de laboratoire.

Quand l'utilisation d'applications externes est nécessaire, l'implémentation doit favoriser l'utilisation de code Java pour faciliter le déploiement et conserver les avantages en termes de portabilité.

Le chapitre 2 introduit la notion de calcul en grille ainsi que la gestion des échanges entre les différentes parties ; le chapitre 3 décrit le contexte dans lequel s'intègre ce travail, l'architecture Peer-to-Peer existante, le simulateur de grille et les choix structurels ; le chapitre 4 explique comment les échanges réseaux et l'exécution des tâches de calcul ont été implémentés ; le chapitre 5 décrit la réflexion effectuée au sujet des transferts de données ainsi que son implémentation grâce à des protocoles déjà existants ; le chapitre 6 aborde la découverte de peers et le chapitre 7, la journalisation.

Nous continuerons en décrivant l'utilisation du système développé (chapitre 8). Nous présenterons ensuite les expériences effectuées (chapitre 9) ainsi que des possibilités de développements futurs (chapitre 10). Enfin, le chapitre 11 conclut en résumant ce qui a été accompli ainsi que des perspectives de développement.

Dans la suite, nous utiliserons les caractères **gras** pour les définitions, `monospace` pour les descriptions de packages, classes et méthodes et une police de plus petite taille pour les variables et extraits de code.

Chapitre 2

Qu'est-ce qu'une grille ?

Dans ce chapitre, la définition générale, puis la structure et les composants d'une grille seront d'abord présentés (sections 2.1 et 2.2). Différentes politiques de partage de ressources sont ensuite examinées (section 2.3). Enfin, OurGrid, le middleware P2P Grid *state of the art* est brièvement introduit (section 2.4).

2.1 Définition générale

Ces dernières années ont vu se développer le calcul distribué. D'abord à un niveau spécialisé dans des laboratoires scientifiques, puis dans une approche plus ouverte au public. L'idée est de se connecter à une grille pour résoudre un problème que l'on ne pourrait traiter avec ses seuls moyens. Dans ce contexte, il est devenu nécessaire de préciser exactement en quoi consiste une grille. À partir de quel moment un groupe d'ordinateurs est-il considéré comme une grille de calcul ? Quand un réseau privé devient-il une grille de stockage ?

Ian Foster [1] présente un canevas de définition au travers de trois critères :

Une grille est un système qui

- « **coordonne des ressources non sujettes à un contrôle centralisé** » : la grille est composée de différents sites de tailles variables, lesquels peuvent être administrés par des entités différentes ne se concertant pas forcément ;
- « **en utilisant des protocoles et interfaces standard, ouverts et généraux** » : la grille se doit d'être générale, bien définie et donc accessible aux programmeurs désirant interagir avec elle ;

- « **pour fournir des qualités de service (QoS) non triviales** » : l'utilité de la grille doit être supérieure à la somme des utilités de ses composants. En d'autres termes, une grille sera plus souvent disponible et plus fiable que ce qu'on pourrait espérer des membres pris individuellement. L'entité qui profite de ces ressources, par contre, n'est pas imposée. Dans le cas plus général, chacun peut profiter des ressources proposées par la ou les communautés dont il fait partie.

À première vue, cette définition semble présenter une contradiction : comment une grille coordonne-t-elle les ressources si ce n'est par un contrôle centralisé ? Pour expliquer ceci, on introduit ce qu'on appelle une *Virtual Organisation* [2, 3], ou VO. La VO est composée de membres (appelés *VO members*, ou *peers* dans ce document) ayant un but commun. De la même manière, ceux-ci ont sûrement aussi une politique commune pour le partage des ressources. Idéalement, elle devrait être compatible avec les besoins et les capacités des membres : à quoi bon préparer une tâche d'importance à un moment où d'autres processus (locaux, ou éventuellement d'une autre VO) accaparent les ressources ?

Cette définition exclut donc des middlewares comme le *Sun Grid Engine* [4], celui-ci étant soumis à un contrôle trop strict. D'autre part, les systèmes Peer-to-Peer les plus populaires (comme eMule [5]) sont principalement limités au transfert de fichiers et donc trop spécialisés même s'ils sont de plus en plus libres de tout contrôle centralisé et fournissent des QoS supérieures.

À noter que la grille considérée dans ce travail respectera le premier critère de par sa nature Peer-to-Peer mais contournera une partie du deuxième : les protocoles utilisés seront les plus généraux possibles sans être standard [6]. En effet, les standards en matière de grille sont pour l'instant toujours en préparation ou non finalisés, donc sujets à changement et relativement peu répandus.

Notre grille sera donc une grille légère (*lightweight)*, c'est-à-dire qu'elle ne se conforme pas aux standards de grille en ce qui concerne les fonctionnalités offertes, ou les protocoles et interfaces utilisés.

Les politiques d'ordonnancement choisies garantissent le respect du troisième critère.

2.2 Structure d'une grille

En utilisant les concepts introduits par Foster [1], Tuecke et Kesselman [2, 7], les définitions synthétiques suivantes [3] présentent les notions de base utiles pour la lecture de ce document.

Le **site** est la brique de base d'une grille. Il est constitué d'un ensemble de ressources de différents types – aussi bien matériels que logiciels – soumises à un même contrôle administratif. Une ressource, dans le cadre spécifique de ce travail, est un ordinateur équipé d'une machine virtuelle Java, d'un middleware d'exécution, d'un accès réseau et d'une capacité de stockage.

Le contrôle des ressources d'un site est assuré par un **peer**. Dans le cadre spécifique de ce travail, un peer est un ordinateur équipé d'une machine virtuelle Java, d'un middleware de gestion, d'un accès réseau et d'une capacité de stockage. Une ressource a donc deux « propriétaires » : un utilisateur physique qui possède l'ordinateur, et un peer qui contrôle le middleware de grille en exécution sur la ressource et a toute latitude quant à l'utilisation de sa puissance de calcul.

Les peers ont leurs propres politiques et objectifs. Ce sont eux qui décident de partager leurs ressources. Les motifs peuvent en être variés :

- accélérer le temps de calcul d'une tâche précise ;
- traiter des données trop importantes pour une seule entité ;
- fournir une garantie lors d'une opération cruciale, en l'effectuant de manière redondante sur plusieurs ressources.

Cette division en sites contrôlés par des administrateurs locaux permet de réduire le nombre de membres et donc l'information qu'on doit conserver à leur propos tout en se rapprochant de la structure physique du réseau.

Notons que l'administrateur d'un peer n'administre pas forcément toutes ses ressources. Il est concevable d'avoir un système où des ressources sous des contrôles administratifs différents se connectent à un même peer (par exemple, différents départements d'une université pourraient fournir leur puissance de calcul à un peer représentant l'université entière). En pratique, il est sans doute préférable que la topologie des peers soit calquée sur le contrôle effectif des ressources car cela simplifie la gestion du réseau et apporte une comptabilisation objective des échanges.

Le plus souvent, les échanges sont bidirectionnels. On peut toutefois imaginer qu'un peer n'ait aucune ressource à disposition et ne fasse partie de la VO que pour consommer des ressources. La compensation devra alors se faire en dehors du système de grille (par exemple par paiement bancaire). Ce type de comportement n'est pas exceptionnel : dans des programmes comme SETI@home [8] ou Folding@Home [9], un seul peer consomme les ressources de tous les autres, lesquels sont récompensés par le sentiment d'avoir fait avancer la recherche.

Une **grille** est un ensemble virtuel de sites composés de ressources hétérogènes. Les relations d'échange peuvent être libres (négociées au cas par cas par les peers seuls) ou contrôlées.

Une **organisation virtuelle** est une communauté de membres, composés chacun d'un peer et du site qu'il administre partageant leurs ressources grâce à une grille. Il est tout à fait concevable qu'un membre fasse partie de plusieurs organisations.

La mise en place d'une grille n'est évidemment pas triviale. On appelle **gridification** le processus de préparation d'un site pour le partage de ressources. D'un point de vue logiciel, l'accès à la grille devrait être le plus transparent possible.

Cette préparation est effectuée par un système d'administration de ressources, le **Resource Management System**, ou RMS. Chaque peer est équipé de son propre RMS.

C'est ainsi qu'on définit le Grid Computing comme la forme de calcul distribué basée sur la gridification du matériel de plusieurs sites.

Les tâches de calcul sont soumises à la grille par des **utilisateurs**. Cette soumission est réalisée par l'intermédiaire du peer dont ils dépendent. Ces utilisateurs ne sont pas forcément les propriétaires physiques des ressources dont nous parlons plus haut.

Dans ce travail, on qualifiera de **nœud** un peer, une ressource ou un utilisateur. Ces tâches *Grid* sont à ne pas confondre avec des tâches locales éventuellement exécutées par le système d'exploitation d'une ressource indépendamment du middleware *Grid* la contrôlant.

Les tâches *Grid* exécutées par un peer peuvent être en outre caractérisées d'après l'origine des noeuds impliqués. Chaque peer peut en effet :

- faire exécuter par ses ressources des tâches de ses utilisateurs (*Local Tasks*) ;
- faire exécuter par ses ressources des tâches soumises par un autre peer, donc pour des utilisateurs de cet autre peer (*Supplying Tasks*) ;
- faire exécuter des tâches de ses propres utilisateurs par les ressources d'un autre peer (*Consumption Tasks*).

La figure 2.1 présente un exemple de grille où des peers partagent leurs ressources pour que des utilisateurs puissent en profiter pour calculer leurs tâches.

2.3 Politiques de partage

Comme Foster le suggérait [1], le partage des ressources doit être coordonné. Dans une définition générale de la grille [3], on admet cependant aussi bien le contrôle individuel où chaque site suit des directives locales gérées par le RMS du peer gérant le site ; que le contrôle centralisé où tous les peers se conforment au plan d'un RMS unique (d'un site donné et contrôlant indirectement tous les sites). Dans ce dernier cas,

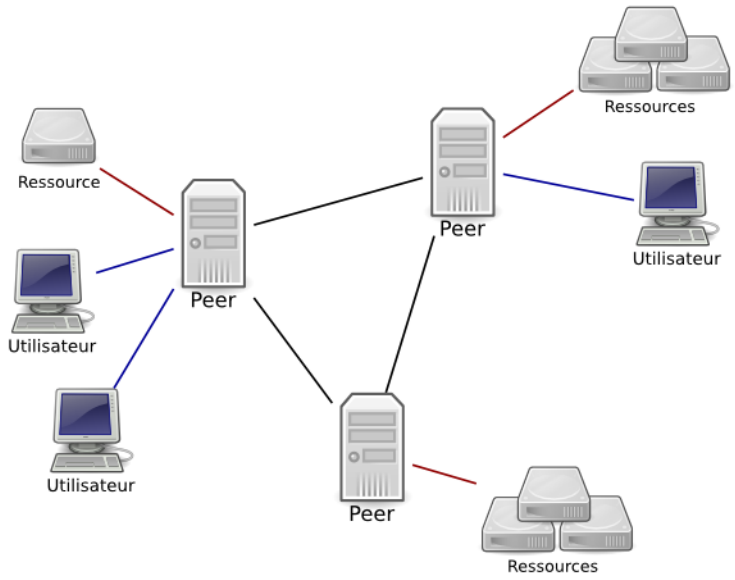


FIG. 2.1 – Vue simplifiée d'une grille

et si l'infrastructure est stable et/ou si on a une information parfaite la concernant, l'utilisation des ressources peut être plus efficace. Si au contraire les ressources ont un comportement difficile à prédire, il est préférable d'utiliser un contrôle individuel, plus flexible et réactif.

De ce fait, il apparaît qu'aucune des deux formes de contrôle ne prévaut en termes de qualité de service. Selon l'usage et la stabilité de l'environnement, il conviendra de faire un choix judicieux lors de la mise en place de la grille.

La classification existante suivante présente différentes politiques de partage de ressources [3].

2.3.1 Philanthropisme

Dans une approche philanthropique, le contrôle coordonne les ressources afin de maximiser l'utilité d'un peer au détriment des autres. Étant donné que toutes les ressources sont fournies à un seul membre, il est inutile de conserver le compte des allocations.

C'est le principe qui régit le *Calcul par Internet* (Internet Computing), ou *Calcul par Volontariat* (Volunteer Computing). C'est le type le plus connu du public, il est appliqué dans les exemples déjà cités de Folding@Home, SETI@home, etc. Un tel système n'est

pas aisé à mettre en place : il faut prévenir le public de l'existence du projet et lui donner une raison de participer. Un concept avantageux, mais difficile à appliquer par des laboratoires de recherche plus modestes.

2.3.2 Mutualisme

Dans une approche mutualiste, le contrôle a pour but de maximiser l'utilisation des ressources (en tant qu'ensemble et non individuellement). Les comptes ne sont pas gardés sur le long terme mais les peers espèrent quand même une certaine récompense pour les services qu'ils ont rendus.

Cette structure est assez tolérante vis-à-vis des problèmes temporaires que les peers peuvent rencontrer avec leurs ressources. Ils seront aidés par le reste de la communauté sur qui la charge sera répartie de manière équitable. Cette organisation demande une grande confiance entre les membres. Dans le cas contraire, des *freeriders* [10] seraient tentés de rejoindre la grille pour profiter des ressources sans en fournir, pénalisant l'ensemble des membres honnêtes, figeant pour ainsi dire la grille s'ils sont trop nombreux.

Une telle confiance peut être obtenue de deux manières. D'abord, on peut s'appuyer sur une infrastructure extérieure comme des certificats ou le fait que tous les postes d'une même entreprise sont sans doute sûrs. D'autre part, la confiance peut s'établir avec le temps, un peer remarquable par sa stabilité acquérant certains privilèges.

2.3.3 Individualisme

C'est ici l'utilité de chaque peer, individuellement, qui doit être maximisée. De ce fait, il devient possible de s'adapter à un contrôle décentralisé et il est nécessaire de maintenir le compte des ressources allouées sur le long terme. Les comptes sont conservés pour chaque peer. Ceux-ci ne fourniront pas plus qu'ils ne recevront et inversement.

Cette politique se base sur l'absence de confiance entre les membres. Cette contrainte pose quelques problèmes mais résiste bien aux freeriders si les ressources sont comptabilisées avec attention. Dans le cas idéal, les freeriders sont inexistants : s'ils ne fournissent rien, ils ne sont pas en mesure de faire de demandes de ressources. Ce n'est toutefois pas aussi facile : si tous les membres attendent qu'un autre fournisse des ressources pour rendre les leurs disponibles, on est face à un deadlock. Plusieurs solutions existent à ce problème de *boostrapping* de la grille (OurGrid [11], dont on parlera plus loin, utilise ce type de compromis). Par exemple, on peut de temps à autre

fournir un faible pourcentage de ressources à un peer qui n'a encore rien fourni, en espérant que cela débloque la situation et permette des échanges productifs à moyen terme. Il est aussi possible de maintenir une liste de priorités et de ne fournir des ressources aux peers inconnus que lorsque ceux à qui on en devait ne font pas de demandes.

Toutes ces solutions conduisent au freeriding, mais dans des proportions raisonnables et peu pénalisantes.

Bien sûr, ces politiques ne sont que des exemples tranchés et selon la confiance et les objectifs poursuivis, on peut trouver des politiques hybrides convenant mieux à telle ou telle situation.

2.4 La grille selon OurGrid

OurGrid [11] est un système de grille Peer-to-Peer orienté calcul, c'est-à-dire à contrôle décentralisé et politique individualiste. OurGrid est facile à mettre en place et requiert un minimum d'interaction humaine : tout le monde peut à loisir y lancer un projet et profiter des ressources collectives, pourvu qu'il soit prêt à partager sa propre puissance de calcul.

Pour simplifier la gestion des calculs, OurGrid n'accepte pour le moment que des applications structurées sous forme de *Bag-of-Tasks* (BoT)[1]. Ces BoT sont des applications parallèles composées de tâches très indépendantes : elles ne requièrent aucune communication entre elles à l'exécution ni le stockage de données temporaires (entre deux exécutions) sur la ressource cible. De ce fait, elles résistent mieux au manque de confiance et à la stabilité incertaine du réseau ciblé (faible qualité de service) : si une ressource vient à se déconnecter, il suffit de recalculer sa tâche indépendamment des autres, sans que cela ne pénalise trop la performance générale du BoT. Restreindre les possibilités des tâches est aussi une mesure de sécurité pour le fournisseur de ressources. Dans OurGrid, la sécurité peut être encore renforcée en forçant les tâches à s'exécuter dans un environnement confiné (SWAN [10]) ne nécessitant pas d'adaptation des programmes ; ceci garantit aussi que le fournisseur ne va pas consulter les données peut-être sensibles du demandeur (cela pourrait être le cas si des laboratoires concurrents travaillent sur la même grille).

Sans garantie de qualité de service, il devient impossible de garantir l'équité. Elle peut toutefois être fortement encouragée par les méthodes de comptabilisation, lesquelles seront discutées plus bas.

[1] Il est toutefois prévu à terme d'étendre le champ d'action [12].

Un contrôle centralisé eût été inconcevable au vu des objectifs d'OurGrid : relancer des négociations de gestion des permissions à la connexion d'un membre dans un système aussi dynamique est une opération d'autant plus coûteuse que la grille a théoriquement une taille indéterminée.

Le principe des échanges dans OurGrid est de donner des cycles CPU à qui en a besoin quand le processeur n'est pas occupé (par une tâche locale ou d'un autre peer). Cette forme d'individualisme non strict permet de résoudre le problème du *bootstrapping* de la grille. Dans OurGrid, les tâches *Grid* locales ont aussi priorité sur toute tâche *Grid* distante : si un peer désire récupérer l'accès à une de ses ressources, le calcul en cours sera interrompu et les données déjà calculées perdues.

2.4.1 Comptabilisation de la consommation de ressources

Le partage des ressources dans OurGrid est basé sur le principe de *réseau de faveurs* [13]. Chaque peer mémorise les dettes qu'il a envers les autres peers, c'est-à-dire les faveurs qu'ils lui ont faites. À chaque fois qu'un calcul est terminé par un peer P' pour le compte d'un peer P, P ajoutera les unités de calculs consommées à sa dette. S'il effectue ensuite un calcul pour P', il réduira d'autant sa dette. Ce compte est conservé localement, éliminant du coup la nécessité d'une autorité de confiance centralisée.

La balance des ressources ne sera jamais négative, même si P donne beaucoup plus de ressources à P' que P' à P. Le résultat serait le même sans cette convention : les membres n'auraient qu'à changer d'identifiant pour récupérer une balance nulle.

La comptabilisation est utilisée lorsque plusieurs peers (soient P' et P'') demandent des ressources à un même peer P. P va alors consulter ses registres de dettes pour P' et P'' et offrir ses ressources au peer envers qui il a la plus grande dette, dans le but de la réduire. P favorisera donc toujours les membres de la VO qui lui rendent ses ressources.

Ce comportement n'est cependant pas pénalisant pour P. En effet, s'il est prêt à partager ses ressources, c'est que lui-même n'en a pas besoin à ce moment. Dans ce cas, offrir ses ressources à d'autres peers ne lui ayant pas encore rendu de service est faire preuve d'une bonne volonté dont ces peers se souviendront peut-être. De fait, lors de son entrée dans une grille, tout peer peut se comporter comme un freerider : il n'a peut-être pas encore eu l'occasion de partager ses propres ressources mais a déjà besoin de celles des autres.

D'autre part, si P accorde en priorité ses ressources aux peers auxquels il a confiance, le comportement n'est pas non plus pénalisant pour la grille dans son ensemble :

les freeriders n'obtiendront des ressources que quand la grille est partiellement inactive et que les peers n'ont pas non plus de tâches locales à exécuter.

L'équité n'est donc pas garantie mais elle est encouragée et l'impact des peers malveillants en est amoindri.

Il reste à définir le concept d'unité de calcul ou, en d'autres termes, comment estimer la valeur de la faveur qu'un peer fait à un autre [14] ? C'est en fonction de cette métrique que le réseau de faveurs gagnera (ou perdra) en efficacité.

Le temps seul n'est pas significatif : ce qui prend peu de temps sur une machine peut en prendre beaucoup plus sur une autre. Il faut donc relativiser le temps vis-à-vis de la puissance des peers. Étant donné qu'on ne peut leur faire confiance pour donner leur propre puissance (se déclarer plus lent reviendrait à s'assurer une faveur plus grande sans ressentir les désavantages liés à un réel manque de puissance, par exemple la probabilité plus élevée qu'une tâche distante soit interrompue par une tâche locale), l'approche suivie dans OurGrid [10, 14] est de calculer une puissance relative, c'est-à-dire que chaque peer pondère la puissance fournie par un autre peer par le rapport entre le temps moyen d'exécution des tâches locales et le temps moyen d'exécution des tâches soumises à ce peer antérieurement.

On est donc bien en présence d'un système totalement décentralisé, où chaque peer maintient ses comptes pour son usage personnel : les relations entre A et B n'influent pas sur celles entre A et C ou B et C.

Chapitre 3

Architecture existante

Mon travail s'intègre dans les recherches dans le domaine du *Grid Computing* [3] actuellement en cours au sein du service d'algorithmique du Département Montefiore de l'Université de Liège. Dans ce cadre, une architecture de Grid Peer-to-Peer ainsi que des classes de base ont été définies préalablement à ce travail.

Ces classes, définies comme faisant partie du middleware *Lightweight Bartering Grid* (package `lbg`), étaient déjà utilisées dans un simulateur qui sera traité dans la section 3.5. Elles ont continué à évoluer en parallèle à mon travail mais mises à part quelques adaptations nécessaire pour se conformer à des contraintes au départ non envisagées et propres au mode réel (dit **Live**), je devais essentiellement effectuer mon implémentation sans modifier l'API existante.

Les fonctionnalités offertes sont divisibles en deux parties :

- les classes de gestion peuvent être utilisées quel que soit le mode de fonctionnement (Live ou simulé) : leur comportement dépend de la stratégie choisie et non du mode de communication entre les nœuds (peers, ressources et utilisateurs). Elles s'occupent notamment des communications de haut niveau (e.g. propositions d'échange de ressources), de la comptabilisation des faveurs (voir section 2.4.1) et de la gestion de l'état des ressources. Ces classes ont principalement pour but de gérer le comportement du peer (voir section 3.2) ;
- les classes représentant les nœuds, ainsi que celles chargées de la communication entre eux sont, quant à elles, déclinées en versions simulée (*Simulated*) et réelle (*Live*). En pratique, la classe `Peer` n'est pas sous-classée de la sorte car, contrairement aux ressources et utilisateurs, son comportement ne change pas entre les deux modes. Selon les cas, on a des fonctionnalités de base définies par des classes abstraites non instanciables (e.g. `lbg.resource.Resource`) ou garanties par des interfaces (e.g. `lbg.peer.service.ExternalPeerHandle`).

`lbg`	Package principal
`|-- common`	Classes d'usage général
`|-- peer`	Représentation et gestion du peer
`. |-- config`	Configuration du comportement du peer
`. |-- core`	Classes d'ordonnancement et de gestion des requêtes
`. |-- register`	Comptabilisation des échanges de ressources
`. |-- service`	Communication entre peers, d'utilisateur / ressource à peer
`. |-- strategy`	Politiques d'ordonnancement, de négociation
`. .`	et de comptabilisation
`. |-- workflow`	Structures de gestion des tâches par groupes,
`.`	description de tâches
`|-- resource`	Représentation et gestion d'une ressource
`. |-- config`	Configuration du comportement de la ressource
`. |-- service`	Communication de peer à ressource
`|-- simulator`	Simulateur de grille
`|-- task`	Tâches à calculer
`|-- user`	Représentation et gestion d'un utilisateur
`  |-- service`	Communication de peer à utilisateur

TAB. 3.1 – Description des packages dans l'architecture existante

Les sections suivantes présentent systématiquement une liste des concepts et packages concernés. La table 3.1 présente l'arborescence du middleware.

3.1 Tâches

Packages concernés : `lbg.peer.workflow`, `lbg.task`

Les tâches de calcul sont d'abord décrites par des `JobDescriptor`s contenant des `TaskDescriptor`s. Ces descripteurs sont ensuite convertis en `Job`s de `BoT`s (voir section 2.4) contenant les `Task`s.

Cette structure a été conservée mais, limitées à la simulation, les classes ont été remaniées lors de l'implémentation réelle, laquelle sera décrite dans la section 4.1. Cette adaptation était une des rares situations justifiant une modification des classes de base de l'architecture existante.

3.2 Peers

Package concerné : `lbg.peer`

Le middleware installé sur chaque peer est articulé autour de l'objet Java `Peer`.

Un tel objet est qualifié par un ID unique pour toute la grille (sans quoi il ne pourrait la rejoindre car d'autres Peers connaîtraient déjà un Peer ayant le même ID et refuseraient le nouveau). Il s'agit principalement d'une interface vers ses composants, lesquels interagissent selon la situation.

Chaque peer est équipé des composants suivants :

Le négociateur est chargé des relations entre peers : c'est lui qui évalue et répond aux demandes de ressources provenant des autres peers de la grille. Les différentes stratégies de négociation sont implémentées dans le package `lbg.peer.strategy.negotiation`.

Le comptable est chargé de quantifier la consommation et la fourniture de ressources. Les différentes stratégies de comptabilisation sont implémentées dans le package `lbg.peer.strategy.accounting`.

L'ordonnanceur (`lbg.peer.scheduler`) analyse les demandes d'exécution et choisit les ressources qui doivent exécuter les tâches locales (*Local*) ou distantes (*Supplying*). Les différentes stratégies d'ordonnancement sont implémentées dans le package `lbg.peer.strategy.scheduling`.

Le gestionnaire de ressources gère les ressources du Peer et conserve leur état, chaque ressource ne pouvant exécuter qu'une tâche à la fois. Il peut envoyer des tâches à ses ressources, les annuler ou les terminer, ou fournir une liste de ressources répondant à un critère donné. Le gestionnaire ainsi que les classes annexes sont implémentés dans le package `lbg.peer.core.rms`.

Le gestionnaire de requêtes gère les demandes d'exécution de jobs par les utilisateurs, est capable d'en maintenir en attente et d'en sélectionner un pour exécution (il s'interface alors avec l'ordonnanceur). Le gestionnaire ainsi que les classes annexes sont implémentés dans le package `lbg.peer.core.qms`.

Le gestionnaire de tâches (`lbg.peer.core.TaskManager`) conserve des statistiques sur les tâches et fournit des procédures d'appel simplifiées pour effectuer des opérations de base (comme l'exécution d'une tâche). Il utilise les gestionnaires de ressources et de requêtes.

Le registre (`lbg.peer.register.GridRegister`) maintient une liste des peers connus, ainsi que les informations collectées à leur sujet au fil des interactions.

La configuration des trois premiers gestionnaires est spécifiée par un `StrategyDescriptor`.

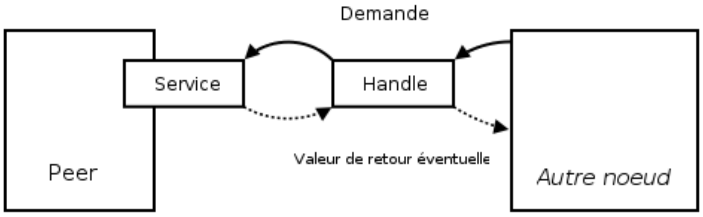


FIG. 3.1 – Modèle de communication par services et handles

En plus de ces classes de gestion, le Peer – comme les autres nœuds de la grille – est défini par un *service* et des *handles* qui lui permettent de communiquer avec le reste de la grille comme le présente la figure 3.1.

Les handles contiennent toute l'information requise pour contacter un Peer (par exemple, un nom de domaine, un identifiant) ; ce sont d'elles que dépend le mode de communication (dans le cas simulé, la demande est normalement directement passée à l'objet). On en distingue trois types – une `interface` pour chaque – pour joindre le peer :

L'externe (`ExternalPeerHandle`) permet la communication entre peers ;

L'interne (`InternalPeerHandle`) permet la communication d'une ressource au peer qui la dirige ;

L'interne-utilisateur (`UserPeerHandle`) permet à un utilisateur de soumettre des tâches au peer dont il dépend.

3.3 Ressources

Package concerné : `lbg.resource`

Le middleware installé sur chaque ressource est articulé autour de l'objet Java `Resource`, capable d'exécuter une tâche de grille à la fois. Le comportement d'une ressource est indépendant du type de la tâche (*Local* ou *Supplying*).

Une ressource est qualifiée par un ID unique (par peer) et une puissance pour l'instant fournie de manière arbitraire et utilisée uniquement à des fins statistiques par le simulateur (modéliser précisément la puissance d'une ressource demande de prendre en compte de nombreux facteurs [14, 15]). Dans le cas simulé, implémenté par la classe `SimulatedResource`, l'exécution d'une tâche se limite à la création d'un événement de

fin d'exécution de tâche en spécifiant une durée d'exécution aléatoire (sous certaines conditions définies par l'environnement).

Comme le peer, la ressource possède un handle et un service permettant notamment de démarrer une tâche, de récupérer les données de sortie ou d'annuler la tâche en cours.

Enfin, notons que si un administrateur veut permettre l'exécution de plusieurs tâches sur une même machine (l'arrivée des processeurs à plusieurs cœurs pourrait rendre ce comportement intéressant), il devra lancer plusieurs instances du middleware.

3.4 Utilisateurs

Package concerné : `lbg.user`

Un utilisateur, représenté par la classe `GridUser`, ne sert qu'à envoyer des jobs (déjà construits : le rôle de l'utilisateur n'est pas de créer les jobs mais seulement de les soumettre au peer) et recevoir leur résultat (tâche par tâche). Sa vision de la grille est très limitée : il ne connaît que le peer dont il dépend et conserve les informations concernant les tâches en cours grâce au JobID que lui fournit le peer. Pour rappel, les tâches soumises à un peer par un utilisateurs sont considérées par le peer comme des tâches locales (*Local Tasks*).

Comme les autres nœuds, l'utilisateur est défini par un ID unique par peer (encore une fois, si l'utilisateur ne remplit pas cette condition, il ne sera pas à même de s'enregistrer auprès du peer) et dispose d'une handle et d'un service, lesquels permettent notamment à son peer de l'avertir de la fin de l'exécution de l'une de ses tâches.

Un GridUser est une classe permettant la soumission de tâches à la grille, pas une interface permettant à un utilisateur physique de le faire. Pour cela, il est nécessaire d'utiliser un programme d'interface, lequel sera décrit section 4.6.

3.5 Simulateur

Package concerné : `lbg.simulator`

`Simulator` est la classe qui pilote le simulateur de grille à événements discrets. Au lancement, elle commence par créer un environnement de simulation défini par la classe `SimulatorEnvironment`. Un simulateur n'est utile que si sa configuration est aisément modifiable. Le `SimulatorEnvironment` permet, à partir d'un fichier répondant aux

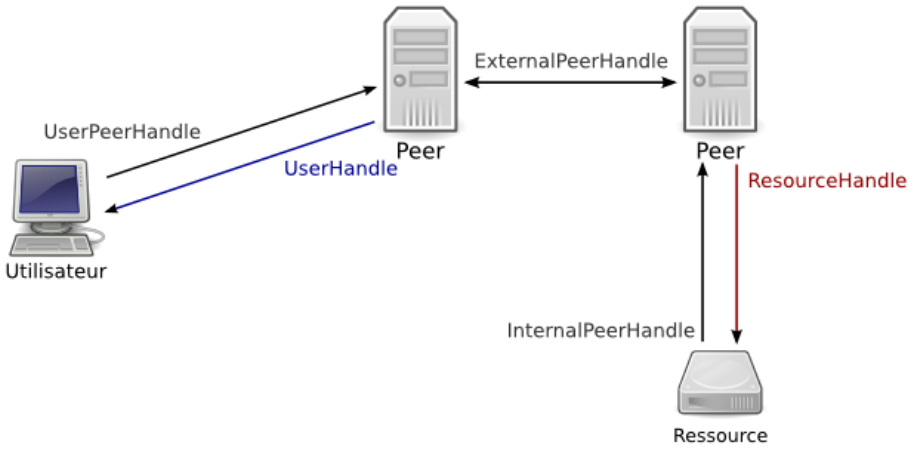


FIG. 3.2 – Modèle de communication par services et handles (détail)

conventions définies par `java.util.Properties`, de créer un ensemble de peers, chacun possédant un nombre de ressources défini, et de leur fournir à chacun un certain nombre de jobs dès le départ (chaque job possède un nombre de tâches lui aussi spécifié par le fichier de configuration).

Une fois l'environnement créé, la soumission de tâches, leur gestion par les peers et leur exécution par les ressources sont simulées par le biais d'événements (dans le sous-package `lbg.simulator.event`) grâce auxquels on est par ailleurs en mesure de conserver des statistiques sur l'exécution.

3.6 Autres classes

Le package `lbg.common` contient d'autres classes d'intérêt général. On y trouve notamment :

- L'exception standard dans le middleware : `GridException`, la seule exception lancée par LBG (mises à part les descendantes de java.lang.RuntimeException) ;
- Des utilitaires (`Common*Handling`) pour effectuer quelques tâches courantes, comme vérifier qu'un port réseau est valide ou créer un fichier à partir de deux chaînes de caractères représentant le nom du fichier et son contenu ;
- Une classe permettant de récolter des données pour analyse statistique.

Chapitre 4

Implémentation des classes Live

La première partie de mon travail consistait - sur base de l'architecture présentée dans la section précédente - à coder les équivalents réels (qualifiés de *Live*) aux classes simulées (qualifiées de *Simulated*).

En mode simulé, tous les nœuds (peers, ressources et utilisateurs) se trouvent sur une même machine, dans une seule machine virtuelle Java. Cet environnement privilégié permet de limiter les méthodes des handles à des appels directs au service concerné, lequel est fourni à l'instanciation du nœud correspondant. Il en va de même pour tous les objets transférés, qui peuvent être accédés par référence. Par contre, en mode Live, quand un objet est envoyé et retourné, il y a un processus de sérialisation qui entre en jeu. On ne peut dès lors plus comparer des références (e.g. sur des Tasks) mais plutôt les identifiants des objets. Cette distinction a causé quelques problèmes dans les gestionnaires du peer au début de l'implémentation réelle. Afin d'éviter que de tels problèmes se reproduisent, une classe `lbg.peer.SimulatedWithCloningPeerService` a été implémentée pour simuler une sérialisation. Cela mis à part, l'intégration s'est effectuée sans problème majeur.

Les tâches sont les éléments de base du calcul en grille. La section 4.1 décrit l'adaptation que leurs descriptions ont subies afin de permettre leur exécution effective.

Une autre modification prioritaire à apporter est l'adaptation des services et handles pour permettre la communication inter-nœuds par le biais de sockets. Sans elle, le middleware ne fonctionnerait pas en mode Live et il serait malaisé de développer des composants de plus haut niveau correctement. La section 4.2 décrit l'implémentation de cette adaptation.

Sur ces bases, il est alors possible d'implémenter les classes représentant les nœuds, à savoir les peers (section 4.3), ressources (section 4.4) et utilisateurs (section 4.5).

La section 4.6 décrit les outils mis à disposition pour le lancement des nœuds ainsi que pour la soumission des tâches de calcul. Ces dernières doivent par ailleurs être développées en se conformant aux règles décrites dans la section 4.7.

4.1 Structure des tâches

Avant de déterminer comment effectuer les appels, il faut commencer par définir quels types de paramètres doivent être transférés. Les tâches n'étant plus limitées à la simulation, il a fallu adapter les descripteurs et représentations du package `lbg.peer.workflow`.

Pour rappel, les tâches soumises dans le cadre de ce travail sont écrites dans le langage Java. La structure particulière du code compilé (appelé *bytecode*) permet l'exécution de d'une telle tâche sur n'importe quelle architecture disposant d'une machine virtuelle Java.

Lorsque l'utilisateur soumet un travail à un peer, il ne soumet en fait qu'une description de ce travail, c'est-à-dire un `JobDescriptor`, lui-même composé de `TaskDescriptor`s qui contiennent les informations nécessaires à la création d'une tâche réelle et qui étaient inutiles dans le cas simulé (pour ne pas changer le comportement du simulateur, des constructeurs surchargés génèrent des valeurs pseudo-aléatoires pour les arguments inutilisés), à savoir (voir figure 4.1) :

- Les noms de toutes les classes nécessaires à l'exécution d'une tâche. Dans un souci d'efficacité (limitation du trafic réseau), les couples $(nomdeclasse, bytecode)$ sont conservés par le `JobDescriptor`, les `TaskDescriptor`s ne contenant que les noms. Dans le cas où les tâches ne différeraient que par leurs paramètres d'entrée (ce procédé où le code et les données ne diffèrent pas d'une tâche à l'autre est appelé *parameter sweep* et est notamment utilisé par le projet Folding@Home [9]), cela évite de transférer plusieurs fois le même bytecode. Ces paires peuvent être fournies à l'instanciation ou lors d'appels à la méthode `add()`. Lors de la transformation des descripteurs par le peer, les `Task`s résultantes contiendront les références vers leurs bytecodes, lesquels ne se trouveront qu'une fois en mémoire, ce qui limite aussi la consommation mémoire du peer. La figure 4.1 ne tient pas compte de cette optimisation ;
- Le nom de la classe principale, celle qui doit être appelée lors de l'exécution de la tâche. Celle-ci ne peut pas être quelconque, les spécificités seront traitées dans la section 4.7 ;
- Un tableau de paramètres d'entrée de types quelconques (passés comme `Object`s, il devront être transtypés par la tâche) ;
- Les données d'entrée. Le chapitre 5 leur est consacré.

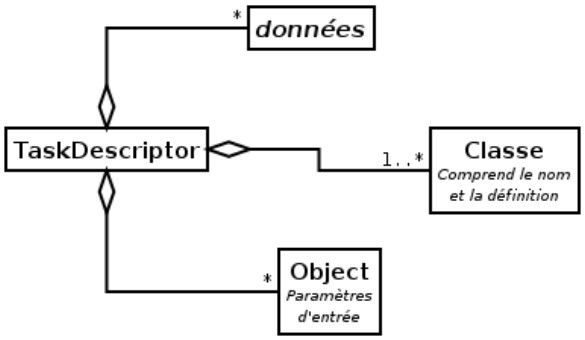


FIG. 4.1 – Information liée à un TaskDescriptor

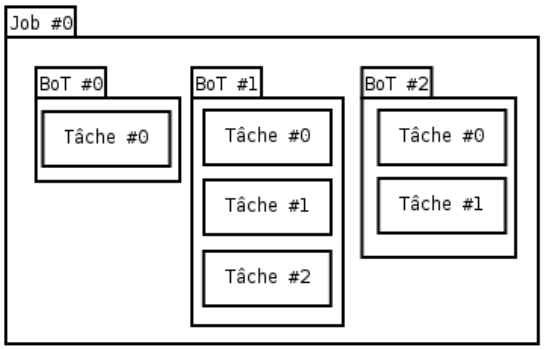


FIG. 4.2 – Structure des Jobs

Quand un peer reçoit un descripteur de job, il le convertit en un ensemble de structures internes (Job, Bag of Tasks, Tasks) en y ajoutant des identifiants comme le présente la figure 4.2.

Chaque `Job` possède un identifiant unique au sein du peer ; chaque `BoT`, un identifiant unique au sein du Job ; et chaque `Task`, un identifiant unique au sein du BoT. Une tâche sera donc identifiée de manière unique par un triplet $(JobID, BoTID, TaskID)$.

C'est à cette étape que les bytecodes sont récupérés depuis le `JobDescriptor` pour être ajoutés aux tâches. Un appel à `JobDescriptor.toTasks(String, int, int)` récupère pour chaque TaskDescriptor les noms des classes dont il dépend et crée une Task contenant des références vers ses bytecodes (en plus des noms de classe). À ce

moment, toutes les tâches créées depuis un JobDescriptor et dépendant d'une même classe ont une référence vers **le même tableau de bytes** en mémoire, ce qui conduit à une économie d'espace si plusieurs tâches d'un Job dépendent de classes communes. En pratique, cette condition est fréquemment remplie par des tâches dites de *parameter sweep*, qui consistent à exécuter plusieurs fois le même algorithme en modifiant les paramètres d'entrée. Ce procédé est notamment utilisé par le projet Folding@Home [9].

La création des Tasks permet aussi aux tâches de conserver des données de sortie, matérialisées par la classe `TaskOutput`. Une Task contient donc au moins les mêmes informations que le TaskDescriptor dont elle est issue. Le cas des données est particulier et sera détaillé dans le chapitre 5, mais la Task les conserve aussi.

En pratique, lorsqu'un peer reçoit un descripteur de job, il ne crée pour l'instant qu'un `Job` composé d'un seul `BoT` tel que spécifié. Ce comportement dépend des classes de base décrites dans la section 3.2 mais les classes Live n'y sont pas limitées car, une fois la conversion du JobDescriptor effectuée, elles ne manipulent que des Tasks et pas des *Bags of Tasks*.

En résumé, les tâches suivent le chemin suivant : d'abord sous la forme d'un groupe de TaskDescriptors conservés dans un JobDescriptor, elles sont envoyées par l'utilisateur à son peer, lequel les convertit en un Job de BoT de Tasks (voir figure 4.2). Le peer envoie ensuite les tâches une par une à ses ressources, en parallèle s'il en a plus d'une libre. S'il ne possède pas de ressources libres, le peer (alors dit *consommateur*) peut demander à d'autres peers qu'il connaît (les *fournisseurs*) d'exécuter ses tâches. Ce comportement est illustré à la figure 4.3.

Une fois la structure des objets à envoyer bien définie, il est possible de mettre en place un système de communication permettant ledit envoi.

4.2 Protocole d'appels distants

La création du protocole de communication se résume en fait à la mise en place d'une méthode générique permettant des appels par le réseau. Utiliser un procédé générique permet de réutiliser le composant de communication pour les échanges entre tous les types de nœuds, en limitant d'autant la redondance de code. L'implémentation est effectuée dans le package `lbg.protocol.rpc`

Sun offre déjà un outil semblable, RMI [16], mais il n'a pas été utilisé ici. En effet, RMI répond à beaucoup d'autres besoins que ceux qui nous concernaient dans le cadre du middleware. Il est donc plus lourd et plus difficile à déployer, aussi bien au niveau du code (les classes à transférer doivent implémenter des interfaces spécifiques) que

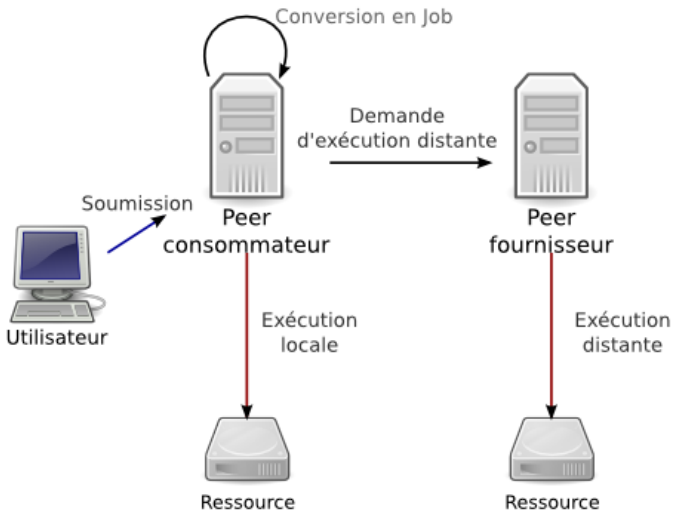


FIG. 4.3 – Transfert des tâches

pour l'administrateur du peer (il est nécessaire de lancer un serveur supplémentaire indépendamment du middleware).

4.2.1 Émission d'un appel distant

La classe maîtresse de l'implémentation choisie est `RemoteCall` : elle représente un appel distant et se base sur la réflexion[1]. Son instanciation (voir listing 4.1) en est d'ailleurs fortement influencée : on passe en argument le type de l'appel (c'est-à-dire le type de nœud auquel on s'adresse, défini par un type énuméré `RemoteCall.CallType`), le nom de la méthode, un tableau de classes et un tableau d'objets sérialisables, lesquels seront passés à la méthode lors de l'invocation sur le nœud distant.

```
public RemoteCall(CallType destination, String methodName, Class[]
    paramtypes, Serializable[] params) throws GridException;
```

Listing 4.1 – Signature du constructeur de RemoteCall

À noter que générer le tableau de classes (`paramtypes`) dynamiquement depuis les objets conduit à des inconsistances. En Java, le typage est dynamique. Si on passe à une méthode `qux(Foo)` un objet de type Bar (`implements Foo`), la JVM traitera l'appel sans problème en utilisant les propriétés de l'héritage. Par contre, si on récupère le type

[1]Voir le package `java.lang.reflect`

Bar de l'objet passé et cherche la méthode correspondante par réflexion, la recherche échouera vu qu'aucune méthode `qux(Bar)` n'existe et que la réflexion ne retourne pas les méthodes prenant en arguments des objets situés en amont dans l'arbre d'héritage.

Dans notre cas, il n'était pas non plus possible de récupérer automatiquement la classe parente : cela implique de remonter systématiquement jusqu'à un certain point de la hiérarchie. Déterminer ce point n'est pas trivial. Par exemple, un `LivePeerHandle` implémente trois interfaces : `InternalPeerHandle`, `ExternalPeerHandle` et `UserPeerHandle`. La ressource doit transtyper la handle en handle interne ; un peer la transtyper en handle externe. Il faudrait donc une logique supplémentaire qui alourdirait les appels si on désirait déterminer la bonne classe parente automatiquement.

La solution trouvée est de passer explicitement les identifiants des classes désirées à l'instanciation. Étant donné que le middleware effectue toujours les appels d'une manière déterminée à la compilation, cela n'est pas pénalisant et n'a de répercussions que sur les handles.

Une fois l'objet créé, la méthode `send()` est appelée. Elle prend en argument l'hôte et le port du nœud distant, ainsi qu'un booléen indiquant si une valeur de retour est attendue. Si c'est le cas, il conviendra de vérifier le type de l'objet retourné et de le transtyper. La méthode utilise des ObjectInput/OutputStreams pour la communication entre nœuds.

Passée la mise au point du procédé, cette approche très systématique permet de rapidement coder les différentes méthodes des handles : il n'y a que peu de changements entre les implémentations des appels.

`RemoteCall` contient aussi une méthode `getPeerHandle(String, int)` qui permet de faire un appel simplifié vers un peer dont on connaît l'adresse (IP ou DNS) et un des ports pour lui demander sa handle. Créer la handle localement requiert la connaissance de toutes les caractéristiques du peer. Vu la structure même de la grille, le peer est le seul type de nœud à ne pas la fournir spontanément : il n'a aucun moyen de prédire quel utilisateur va soumettre un Job ou quelle ressource va venir s'ajouter. Par conséquent, il ne peut donc pas leur envoyer sa handle à l'avance.

4.2.2 Réception et traitement des appels

Une fois le `RemoteCall` envoyé, il faut un serveur à l'écoute pour le gérer. C'est la fonction assurée par le `RemoteCallDaemon`.

Ce démon (bien qu'il ne soit pas considéré par la JVM comme tel car il doit maintenir le programme en exécution[2]) est instancié selon trois ou quatre paramètres :

- Un type `RemoteCall.CallType` d'appel à accepter. Il correspond au type du nœud à gérer (e.g. PEER pour un Peer). Si un appel d'un autre type est reçu, il est ignoré ;
- Un port sur lequel écouter ;
- Un `Object` à qui les appels seront appliqués. Si on suit le principe présenté dans les figures 3.1 et 3.2, il s'agit d'un service dont la classe dépend du type de l'appel (le transtypage se fait depuis `RemoteCall.doCall(Object)`) ;
- Un nombre optionnel de gestionnaires d'appels. Si cet argument est omis, la valeur DEFAULT_HANDLERS est utilisée.

Quand un démon est lancé (dans un processus à part), il commence par créer un pool de N gestionnaires d'appels, où N est la quatrième valeur présentée ci-dessus (l'argument ou une valeur par défaut). Pour fournir une bonne qualité de service et un faible temps de réponse, le démon se contente d'accepter les connexions entrantes : la récupération des requêtes, leur validation et leur exécution est déléguée à une classe privée dont les instances s'exécutent dans d'autres processus, libérant ainsi le démon pour d'autres demandes. Toutefois, la création de processus est une opération coûteuse et instancier un `Thread` à chaque connexion ralentirait considérablement le traitement. Pour éviter cela, tous les processus sont créés au lancement et réutilisés lors de l'arrivée de nouveaux appels RPC. Quand ils ne traitent pas une demande, les gestionnaires sont placés en attente passive sur un `java.util.concurrent.Semaphore`, classe introduite dans J2SE 5.0 (Java 5). L'utilisation de cette classe a permis d'implémenter rapidement le système de producteurs-consommateurs : le socket produit de l'information pour les N gestionnaires et ne peut produire de l'information que s'il y a au moins une place libre, c'est-à-dire un gestionnaire inactif.

Quand tous les gestionnaires sont occupés à traiter un appel, les candidats sont mis en attente : la première connexion est acceptée et attend la libération d'un gestionnaire, les suivantes sont mises en attente par un procédé interne aux `Socket`s Java

En cas d'erreur, la connexion posant problème est ignorée afin de garantir la continuité du service. Les erreurs sont de deux types :

- La connexion a échoué au niveau des entrées-sorties. Cela peut par exemple être le cas si la connexion est interrompue ou si une application étrangère tente de se connecter (e.g. un *port scan*) ;
- L'appel par réflexion a échoué, soit parce que la méthode demandée n'a pas été trouvée (un utilisateur malveillant pourrait forger des RemoteCall invalides), soit parce que l'appel de cette méthode s'est terminé suite à une exception.

[2]La JVM termine son exécution lorsque les seuls Threads encore actifs sont des démons.

4.3 Peers

La gestion réseau du Peer ne se différencie de la simulation que par deux classes : `LivePeerService` et `LivePeerHandle`.

`LivePeerService` est le service côté peer. Il ne fait, dans la plupart des cas, que rediriger les appels vers les gestionnaires appropriés. Le seul cas particulier est `uploadLocallyCompletedTask(Task)`, qui doit avertir de la fin de l'exécution de façons différentes selon qu'il s'agit d'une tâche locale ou non. Le comportement du service reste toutefois assez similaire à la version simulée, simplifiant d'autant l'implémentation Live.

`LivePeerHandle` est la handle (effective) du Peer regroupant les handles (logiques) externes, internes, et internes-utilisateur. Les méthodes consistent toutes en des appels par le biais de `RemoteCall`s.

Détaillons par exemple l'appel `LivePeerHandle.submitJob()` par lequel un utilisateur peut soumettre un Job qu'il a créé à un Peer.

On commence par instancier le `RemoteCall`. L'appel présenté dans le listing 4.2 concerne une méthode *submitJob()* (deuxième argument) prenant en arguments une `UserHandle` et un descripteur de job (troisième argument) et à laquelle on passe les paramètre `uh` et `jd`. Comme l'indique le premier argument, l'appel est destiné à un peer.

```
RemoteCall rc = new RemoteCall(RemoteCall.CallType.PEER, "submitJob",
  new Class[] {Class.forName("lbg.user.service.UserHandle"),
  Class.forName("lbg.peer.workflow.JobDescriptor")}, new Object[]
  {uh, jd} );
```

Listing 4.2 – Création d'un RemoteCall

On envoie ensuite l'appel à hostname:user_port en attendant une valeur de retour grâce à Object ret = rc.send(hostname, user_port, **true**);. Cette attente est bloquante, une propriété importante pour se conformer aux définitions des appels par services et handles qui avaient été définies en simulation (certaines classes de base étant partagées, ce type de comportement ne peut pas changer fondamentalement de la simulation au mode Live).

Enfin, on vérifie que la valeur de retour correspond au type attendu avant de la renvoyer (listing 4.3).

```
if (ret == null || !(ret instanceof Integer))
```

```
  throw new GridException("Invalid returned jobID");

return ((Integer) ret).intValue();
```

Listing 4.3 – Vérification de la valeur de retour

En groupant ces morceaux de code et en ajoutant la gestion des exceptions, on obtient le code suivant du listing 4.4.

```
public int submitJob(UserHandle uh, JobDescriptor jd) throws
    GridException {
  Object ret = null;
  try {
    RemoteCall rc = new RemoteCall(RemoteCall.CallType.PEER, "submitJob",
      new Class[] {Class.forName("lbg.user.service.UserHandle"),
      Class.forName("lbg.peer.workflow.JobDescriptor")}, new Object[]
      {uh, jd} );
    ret = rc.send(hostname, user_port, true);
  } catch (ClassNotFoundException e) {
    throw new GridException(
      "Critical: could not find one of the base classes", e);
  }

  if (ret == null || !(ret instanceof Integer))
    throw new GridException("Invalid returned jobID");

  return ((Integer) ret).intValue();
}
```

Listing 4.4 – Un exemple d'appel distant (LivePeerHandle)

4.4 Ressources

Les `Resource`s présentaient le même type de modifications que pour les Peers : une LiveResourceHandle et un LiveResourceService ont été créés, contenant respectivement des appels réseau semblables à ceux présentés ci-dessus et des appels triviaux vers les méthodes de Resource (plus précisément, de ses sous-classes Live et simulée).

À l'image de SimulatedResource, une classe LiveResource a été écrite. Par rapport à son homologue simulé, la version *Live* se distingue principalement en deux points.

Premièrement, elle doit se connecter à un peer et se déclarer comme faisant partie de son pool. L'ajout d'une ressource à un peer s'effectue de façon très simple : il lui

suffit de faire un appel à `addResource` sur la handle du peer (`InternalPeerHandle`) qui lui a été fournie. En passant en argument sa propre handle, la ressource s'assure que le peer saura la contacter en cas de besoin.
Il est aussi nécessaire de prévenir le peer lorsque la ressource termine l'exécution d'une tâche, de manière à ce qu'il puisse l'enlever de sa liste et en tenir compte dans son ordonnanceur.

Ensuite, une ressource Live traite des tâches réelles pour lesquelles il faut transférer les données d'entrée et de sortie, et qu'il faut exécuter.

4.4.1 Exécution des tâches

Exécuter une tâche ne se résume pas à un simple appel de méthode. La ressource doit répondre à trois critères :

1. Elle doit rester accessible pendant toute la durée de l'exécution ;
2. Elle doit offrir un environnement relativement sécurisé. Il est impensable qu'une tâche mal écrite (peut-être volontairement) puisse compromettre l'exécution de la ressource. Par extension, il doit être possible d'interrompre l'exécution d'une tâche à tout moment. Les raisons en sont diverses : la tâche peut avoir pris trop de temps par rapport à ce qui était prévu, le peer fournisseur peut avoir une tâche de priorité plus importante à exécuter ou le peer consommateur peut avoir annulé l'exécution. Il n'est pas possible de faire confiance aux tâches pour qu'elles s'arrêtent d'elles-mêmes via une méthode `abort()` : rien ne garantit que le programmeur de la tâche a effectivement respecté les directives ;
3. Elle doit charger dynamiquement le code et les données dont la tâche a besoin, et lui faire parvenir.

Le premier critère peut être satisfait grâce à un des deux mécanismes suivants. D'une part, il est possible d'utiliser des `Thread`s, l'implémentation Java des processus légers. Leur avantage est bien décrit par leur nom : ils sont plus faciles à lancer et consomment moins de ressources que leur alternative, les `Process`es. Bien que plus lourds, ceux-ci apportent toutefois deux avantages importants en matière de sécurité, le deuxième critère.

D'abord, leur espace mémoire est séparé de celui du processus qui les a lancés. On peut ainsi empêcher la tâche de perturber le fonctionnement de la ressource. Ensuite, les méthodes d'arrêt de Threads (notamment `Thread.stop()`) étant dépréciées depuis le J2SE 1.3 (Java 1.3) car introduisant des risques de deadlocks [17], les Processes sont les seuls à garantir la fin de l'exécution via une méthode `destroy()` (les risques de

deadlocks sont limités au processus fils et n'influencent pas le parent, c'est pourquoi la méthode est disponible). De cela découle qu'il est possible de permettre l'exécution de tâches multiprocessus (ce mode de programmation est de plus en plus courant) sans avoir à se soucier d'interrompre tous les Threads lancés : ils le seront automatiquement à la terminaison du Process.

La pénalité de temps induite par les Processes est aussi réduite par le fait que la résolution temporelle est de l'ordre de la seconde, c'est-à-dire que la mesure du temps d'exécution d'une tâche en vue d'une comptabilisation des ressources consommées est calculée en secondes.

Le middleware LBG comporte des utilitaires que les tâches pourraient vouloir exploiter. Il est possible de leur en donner l'accès aussi bien en utilisant des Threads que des Processes.

Enfin, le troisième critère requiert l'implémentation d'une sous-classe de `java.lang.classloader` qui permet notamment de charger une définition sur base de son bytecode et de son nom complet (voir les spécifications des noms binaires [18]) sans avoir besoin d'écrire le code sur disque.

L'implémentation choisie fonctionne selon le procédé suivant.

Lors de l'exécution d'une tâche, la LiveResource fait appel aux fonctionnalités de Resource pour conserver des données permettant la gestion de l'état de la ressource, démarre le téléchargement des données (dans un processus parallèle, les détails en seront vus dans la section 5.2.1) et confie l'exécution à un nouveau `LiveTaskHandler` qui s'exécute dans un Thread séparé. À ce moment, la ressource est déjà disponible pour répondre à de nouvelles requêtes. Au démarrage de son exécution, le handler commence par attendre que les données d'entrée de la tâche soient téléchargées puis crée un Process pour y exécuter la tâche.

La classe lancée par ce Process n'est pas encore la tâche mais un `LiveTaskRunner` avec lequel le handler peut communiquer d'une manière déterminée : il va lui passer la tâche sur son entrée standard et lira les données de sortie sur sa sortie standard, à la manière des *pipes* *NIX.

De son côté, après avoir reçu la tâche qu'il doit exécuter, le runner va charger dynamiquement l'ensemble des classes dont la tâche a besoin grâce aux informations qu'elle fournit et aux méthodes de `java.lang.ClassLoader` dont il hérite. Lors de ce chargement, il prendra note de la classe principale et vérifiera qu'elle implémente bien l'interface d'application en grille qui sera présentée dans la section 4.7.

Il redirige ensuite les flux standard : stdin devient un `lbg.common.NullInputStream` qui se comporte comme un FileInputStream pointant sur un fichier vide et

stdout ainsi que stderr sont redirigés dans un `lbg.common.EnhancedByteArrayOutputStream` qui conserve au maximum une certaine quantité d'information (et ignore les informations excédentaires). Rediriger les flux évite qu'une tâche ne corrompe le lien de communication avec le LiveTaskHandler ou ne communique directement avec celui-ci. Cela permet aussi de limiter les modifications à apporter à un programme existant pour l'adapter à la grille (pour autant qu'il ne fasse pas un trop grand usage de la sortie standard).

Une fois ces opérations terminées, le runner lance la tâche et retourne le contenu de l'EnhancedByteArrayOutputStream ainsi que les données de sortie après son exécution.

A priori, on aurait pu se passer de la création d'un LiveTaskRunner et forcer l'implémentation d'une méthode `main(String[])` par l'application. Son utilisation apporte toutefois trois avantages :

- D'abord, on est sûr du mode de communication des paramètres et données d'entrée / sortie car le transfert s'effectue entre deux entités dont on connaît les détails ;
- Ensuite, pour démarrer la tâche sans le runner, il eût été nécessaire de réécrire toutes les classes sur le disque en conservant la structure du fichier jar original pour que la nouvelle JVM charge correctement les classes provenant de différents packages, le *classpath* n'étant pas partagé entre les deux JVM (il doit être complet **avant** le lancement de la tâche) ;
- Enfin, le programmeur de la tâche n'a pas besoin de gérer lui-même les transferts. Il sait qu'il a accès aux données au lancement de sa tâche, et a aussi une API claire pour fournir des données de sortie.

4.5 Utilisateurs

Un utilisateur gère essentiellement deux opérations. La première est la soumission de Job et se résume à un appel de handle (celle qui le relie au Peer). La deuxième permet d'avertir l'utilisateur qu'une de ses tâches (ou la dernière tâche d'un de ses Jobs) est terminée. Il s'agit donc d'une opération de type *serveur* (l'utilisateur attend que le peer le prévienne), et un service (`LiveUserService`) ainsi qu'une handle (`LiveUserHandle`) ont été créés en conséquence.

Lorsqu'une tâche est complétée, l'utilisateur commence par mettre à jour ses statistiques (nombre de tâches complétées et temps écoulé depuis la soumission de la tâche) puis copie les données de sortie de la tâche sur disque dans un répertoire fourni à l'instanciation avant d'avertir l'utilisateur physique (celui qui a lancé le programme) de la complétion.

Dans le cas de la terminaison d'un Job, seule la signature de l'appel effectué par le Peer à l'utilisateur change afin d'indiquer que l'ensemble des tâches du Job est terminé. La dernière tâche à avoir été exécutée est passée à l'utilisateur et traitée en interne de la même manière que pour une simple terminaison de tâche à ceci près que l'utilisateur physique est averti de la complétion d'un Job et non d'une tâche.

Deux problèmes peuvent se produire durant ces courtes opérations :

1. L'utilisateur ne reconnaît pas le Job auquel appartient la tâche qu'on lui a renvoyé, c'est-à-dire que l'identifiant de Job contenu dans la tâche ne correspond à aucun des identifiants de Jobs qu'il a reçu lors des soumissions à son peer.

   Cette situation est possible car l'utilisateur n'est pas tenu de rester connecté pendant tout la durée de l'exécution, pour autant qu'il soit disponible à chaque complétion de tâche. Cette limitation est inhérente aux classes décrites en 3.2, il serait préférable dans un développement ultérieur que le peer recontacte l'utilisateur dès sa connexion si ce dernier est indisponible.

   Si l'utilisateur est redémarré entre la soumission et une complétion, il n'a pas accès à son historique de soumission (aucun procédé de persistance de ce dernier n'est pour l'instant implémenté). Dans ce cas, il accepte la tâche en avertissant l'utilisateur physique ;
2. Le fichier supposé contenir les données de sortie existe déjà. Cette situation est plus que probable si le Job soumis comporte plusieurs fois la même tâche et que les données retournées par celle-ci ont toujours le même identifiant. Dans ce cas, nous adoptons une politique similaire à celle du programme `wget` [19] : quand des fichiers existent, une nombre positif ou nul est ajouté à la fin du nom de fichier. Le nombre choisi est le premier disponible : pour un nom de fichier candidat *FILE* et des fichiers sur disque *FILE*, *FILE.0* et *FILE.2*, le nom sélectionné sera *FILE.1*.

4.6 Lanceurs

Nous avons jusqu'ici décrit les classes nécessaires à la création d'un système de grille. Les programmes eux-mêmes, ceux qui seront lancés par les administrateurs et utilisateurs, n'ont pas encore été traités.

Trois lanceurs simples (un par type de nœud dans la grille) mais généraux ont été écrits dans le package `lbg.middleware`, à savoir `LaunchPeer`, `LaunchResource` et `LaunchUser`. Le cas échéant, ils pourraient être utilisés comme base pour la création d'interfaces plus adaptées à une situation particulière (ils ne donnent pas accès à l'intégralité des services offerts par la grille) mais ils devraient suffire pour la ma-

jorité des situations. Leur fonctionnement se résume en 4 points (un exemple de leur utilisation sera présenté dans la section 8.1) :

- Définition et récupération des arguments en ligne de commande, lesquels permettent de spécifier les aspects clé des nœuds.
  Pour augmenter la réutilisabilité du code, une classe `lbg.common.ArgsParser` a été créée. Un tel parseur est d'abord configuré en lui spécifiant une liste d'options acceptées par le programme. Chaque paramètre peut présenter une option courte (`-l`) et une longue (`-long` façon Java, ou `--long` façon GNU), indiquer si elle est optionnelle, si elle prend un argument (et de quel type) avec une éventuelle valeur par défaut, ainsi qu'une description de sa fonctionnalité.
  Une fois le parseur configuré, un simple appel à `parse(argv)` vérifiera la ligne de commande et rendra les valeurs des paramètres disponibles pour de futurs appels aux méthodes `*ValueOf()`. Si une erreur se produit à cette étape, une exception est lancée, et le programme peut faire un appel à `usage()` pour obtenir une liste des options valides présentée de manière standard.
  À noter que cette classe étant prévue pour être réutilisable facilement indépendamment de ce travail, elle est la seule du middleware à ne pas lancer une GridException en cas de problème (à la place, une `java.text.ParseException` est lancée).
- Création de la classe correspondante au nœud, et de son service, en récupérant le handle du peer dont dépend le nœud s'il y a lieu (c'est le cas pour les ressources et les utilisateurs).
- Création des RemoteCallDaemons décrits plus haut.
- Affichage de statistiques (si leur affichage n'a pas été interdit par le paramètre idoine).

Durant les premiers stades du développement, ce comportement suffisait car il y avait peu de paramètres à définir. Par après, les fonctionnalités se sont étendues, de même que les paramètres. Cela a justifié la création des packages `lbg.peer.config`, `lbg.resource.config` et `lbg.user.config`, dont les classes prennent en arguments des objets `Properties` (Java permet le chargement de tels objets directement depuis un fichier à la syntaxe simple).

Pour permettre un déploiement aisé, il reste possible de spécifier les identifiants et noms d'hôtes en ligne de commande. Par exemple, dans une des configurations de test (les résultats en seront présentées dans le chapitre 9), tous les nœuds étaient lancés sur des machines partageant des répertoires par NFS [20]. Dans ce contexte, il est donc intéressant que tous les nœuds d'un même type puissent partager une partie de leur configuration. Étant donné que les nœuds tournent sur des machines différentes, il est aussi nécessaire de pouvoir leur spécifier leur ID et leur adresse (IP ou hôte) normalement spécifiés par le fichier de configuration indépendamment pour chacun.

En combinant les fichiers de configuration aux arguments en ligne de commande, on évite d'avoir à créer un fichier de configuration par nœud tout en limitant la taille de la ligne de commande pour le lancement.

D'éventuelles étapes supplémentaires peuvent avoir lieu, comme la soumission d'un Job pour un GridUser (l'opération critique dans ce cas est la création des descripteurs et est déléguées aux classes `PropertiesJobLoader` et `SubmissionUtils`, toutes deux dans le package `lbg.middleware`), mais ces points se retrouvent dans tous les lanceurs.

4.7 Interface pour les applications de grille

Pour permettre aux programmeurs de développer ou d'adapter une application pour le système de grille qui a été mis en place, il est nécessaire de leur fournir les spécifications du contexte dans lequel elle sera appelée. Dans cette optique, la classe principale de toute tâche valide devra implémenter `lbg.common.GridApplication` (cette condition est vérifiée avant l'exécution de la tâche), laquelle requiert l'implémentation des méthodes suivantes, qui seront appelées au lancement de la tâche **dans cet ordre** :

- `public void setParameters(Object[] parameters)` spécifie les paramètres d'entrée de la tâche, tels que définis dans le TaskDescriptor. Ces paramètres seront passés d'un noeud à l'autre et doivent donc tous être sérialisables. Il s'agira sans doute de `String`s dans la plupart des cas (c'est le comportement choisi pour les lanceurs), mais le middleware n'y est pas limité ;
- `public void setInputData(GridData data)` fournit à la tâche ses données d'entrée. La classe GridData est un *wrapper* pour les données soumises. Les détails de son implémentation sont traités dans le chapitre 5 ;
- `public void compute()` commande le démarrage de la tâche. La tâche ne devrait donc pas tenter de s'exécuter dès son instanciation : la ressource n'en serait pas affectée, mais la tâche n'a alors aucune garantie concernant la présence de ses données ;
- `public byte[] getResult()` récupère les données de sortie de la tâche. On peut raisonnablement estimer qu'il y en aura toujours, la tâche s'effectuant sans interaction supplémentaire avec l'extérieur. Sinon, elle peut retourner un tableau vide.

Retourner un tableau d'octets n'est pas une opération aussi contraignante qu'il pourrait paraître. La classe `java.io.ByteArrayOutputStream` (disponible depuis Java 1.0) permet de placer des données dans un tableau d'octets d'une manière assez transparente.

La classe principale devra aussi être instanciable par le biais d'un constructeur sans arguments.

4.7.1 Simulation en mode Live

Cette interface s'est montrée suffisamment souple pour adapter facilement des programmes existants. En effet, quand le middleware a atteint un niveau de stabilité convenable, il a été possible de faire tourner le simulateur en mode *Live* en ne modifiant que très légèrement le simulateur (pour éviter des accès directs au fichier de configuration de simulation).

Exécuter le simulateur comme application en mode *Live* consiste à soumettre à la grille une tâche qui, lors de son exécution, fera appel au simulateur décrit dans la section 3.5. De cette manière, on peut utiliser le middleware dans une configuration reconnue comme stable pour simuler ses autres configurations et ainsi gagner du temps en parallélisant les simulations plutôt qu'en les effectuant séquentiellement.

Pour ce faire, la classe `lbg.examples.RemoteSimulator` a été développée. Celle-ci implémente `GridApplication`, n'utilise pas de paramètre d'entrée et prend comme données d'entrée un fichier de configuration de simulation. Quand cette application est exécutée sur une ressource, elle fait appel aux classes du middleware qui ont été déjà été chargées par le `LiveTaskHandler` (voir section 4.4.1). Il n'est donc pas nécessaire de les inclure lors de la soumission.

Une approche similaire – bien que présentant une nuance importante – a été mise en oeuvre dans le projet OurGrid. Contrairement à LBG, où le simulateur est partie intégrante du code et fait appel aux mêmes classes de gestion et d'ordonnancement que la partie réelle, le simulateur d'OurGrid est un logiciel séparé basé sur un ensemble de modifications apportées à la partie réelle. Dans ce cadre, les développeurs d'OurGrid utilisent une version stable du logiciel réel pour simuler le comportement des politiques en cours de développement.

Chapitre 5

Gestion des données

Tant qu'à présent, nous avons envisagé la soumission de tâche comme le simple envoi de bytecode et de paramètres d'entrée simples. Toutefois, en situation réelle, il est aussi nécessaire de permettre à une tâche d'utiliser des fichiers contenant des données d'entrée.

Initialement, les données étaient placées dans un tableau d'octets transféré avec la tâche. Cependant, inclure des données dans la description d'une tâche n'est pas forcément la meilleure solution. Les transferts de données dépendent en effet d'autres objectifs que les exécutions de tâches, et sont soumis à d'autres contraintes. Par exemple, minimiser le temps de téléchargement des données est très important alors que garantir la réciprocité entre les clients de téléchargement n'est sans doute pas aussi important.

De ce fait, les possibilités offertes par le middleware ont été revues à la hausse pour tenir compte de ces contraintes, tirer parti des possibilités de la grille et apporter plus de flexibilité et de performance. Les différentes possibilités de transfert de données sont décrites dans la section 5.1.

Pour être transférées, les données doivent être partagées. Une fois transférées, elles peuvent être mises en cache pour réduire l'impact des temps de téléchargement. Le gestionnaire présenté dans la section 5.2 offre de telles fonctionnalités.

Enfin, la section 5.3 résume les solutions de téléchargement implémentées, et met en évidence la séparation des chemins suivis sur la grille par les données et par les tâches.

Les classes concernées par ce chapitre font partie du package `lbg.protocol.data`.

5.1 Différents types de données

Pour le transfert des données d'entrée d'une tâche, l'architecture LBG utilise un type de donnée générique GridData, imposant quelques champs et méthodes, notamment :

- Un identifiant de donnée (du type `peer.user.filename`) ;
- Une méthode pour écrire les données dans un fichier, qui permet de conserver les données en limitant l'utilisation mémoire ;
- Une méthode permettant la lecture des données par le biais d'un InputStream, dont la tâche se servira pour lire ses données.

Cette classe est aussi sérialisable. De ce fait, toutes ses sous-classes seront transférables par le réseau.

Les données étant uniquement reconnues grâce à leur identifiant, cela implique que si deux utilisateurs veulent soumettre les mêmes données pour des tâches différentes, ils devront le faire à tour de rôle en utilisant le même ID d'utilisateur et en se connectant au même Peer. Dans le cas contraire, ils seront tout de même en mesure de soumettre leurs données sans erreur, mais elles ne seront pas partagées (elle devront être envoyées deux fois au peer et celui-ci ne saura pas optimiser les transferts). En pratique, cela n'est pas une restriction : des utilisateurs désirant partager leurs données pour limiter les temps de transfert n'ont sans doute pas d'intérêt à se cacher les résultats de leurs exécutions.

Pour dépasser cette limitation, il faudrait utiliser un système centralisé qui attribuerait les identifiants (*Global Naming*). Toutefois, un tel système est en contradiction avec un des objectifs principaux de ce travail, c'est-à-dire de conserver une architecture aussi décentralisée que possible.

Une GridData représente toujours un et un seul fichier. Si l'utilisateur le désire, il peut soumettre plusieurs fichiers d'entrée, à raison d'une GridData par fichier.

Les données d'entrée sont conservées dans une ou plusieurs GridData de la création de la description d'une tâche sur l'ordinateur de l'utilisateur à la fin de son exécution sur une ressource en passant par sa conversion d'une description en une tâche réelle sur le peer consommateur.

L'intérêt d'utiliser une classe pour encapsuler les données est que l'on peut disposer de plusieurs options pour le transfert. La figure 5.1 résume différentes options envisageables :

- La ressource qui exécutera la tâche peut demander les données requises par celle-ci à l'utilisateur ;

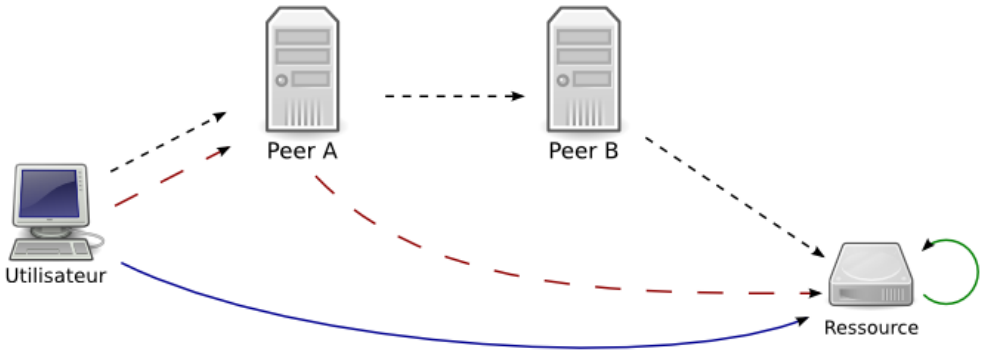


FIG. 5.1 – Transferts théoriquement possibles

- L'utilisateur peut envoyer les données au peer consommateur (A), à qui la ressource les demandera ;
- Plutôt que de faire suivre à la ressource, le peer consommateur (A) peut transmettre les données au peer fournisseur (B), qui contactera lui-même sa ressource ;
- Des ressources possédant déjà les données peuvent les fournir à la ressource exécutant la tâche. Cette option ne se suffit pas à elle-même : il est nécessaire qu'au moins une ressource ait eu accès aux données à l'aide d'une des précédentes techniques.

Les trois dernières options ont été implémentées. La première n'est pas mise en œuvre car elle demande une infrastructure importante au niveau de l'utilisateur et requiert que celui-ci reste connecté pendant les négociations entre peers : il ne sait pas a priori quand une ressource aura besoin des données qu'il offre et ne peut donc se déconnecter.

5.1.1 InlineGridData

Le type de données le plus simple. Dans ce contexte, les données sont contenues directement dans la GridData par un tableau de bytes et suivront donc la tâche dans tous ses transferts sur le réseau.

Ce type de données a ses avantages et ses inconvénients. D'abord, c'est un choix privilégié pour l'utilisateur. En effet, ce type de transfert ne demande aucune infrastructure à son niveau. Pas besoin de serveur de données, lequel pose souvent des problèmes de déploiement et de sécurité. Par contre, ce type de données n'est absolument pas

adapté à des gros fichiers. Le tableau doit en effet être entièrement chargé en mémoire et transféré en une fois. Si l'utilisateur n'a pas une connexion réseau stable, cette restriction est très pénalisante. Ensuite, il a été dit plus haut (voir section 4.1, les données mentionnées étant donc des GridDatas) que les données étaient associées à la tâche pendant toute sa vie (figure 4.3). Cela pose encore une fois un problème de performances : quand une ressource renvoie une tâche terminée (avec ses données de sortie) ou annulée, les données d'entrées reviennent elles aussi au peer, avec de nouveaux problèmes de consommation de mémoire. Enfin, il est à noter que l'utilisation des données *inline* est strictement limitée aux transferts de données survenant lors de soumissions de descripteurs de Job au peer par un utilisateur.

Pour contrer ces problèmes, il peut être envisagé d'ajouter une méthode à une tâche qui permettrait de purger les données *inline*. Avant de renvoyer la tâche à son peer, une ressource pourrait ainsi la débarrasser de toute charge superflue (en cas d'annulation, le peer se base sur les identifiants de la tâche retournée pour retrouver celle qu'il avait envoyée).

A l'opposé de cette vision simpliste et pratique mais inefficace et sujette aux erreurs, un deuxième type de données a été développé.

5.1.2 FTPGridData

Contrairement à l'InlineGridData, les données ne sont pas ici pas transférées avec la tâche. On ne transfère maintenant qu'une série d'informations qui permettent de récupérer les données d'un autre point du réseau. Transférer la GridData avec la tâche ne pose donc plus les problèmes précédemment rencontrés, l'overhead étant extrêmement réduit.

Dans le principe, on peut voir ce type de données comme une évolution du transfert *inline*. Il allie une certaine simplicité des transferts directs et centralisés à l'efficacité de ne transférer que l'ensemble des méta-informations nécessaires à un téléchargement ultérieur des données.

Une FTPGridData est définie par un identifiant, un hôte et un port (le port FTP est généralement le 21, mais il peut être spécifié librement dans cette classe de données), un utilisateur et un mot de passe, et un chemin absolu pour trouver les données sur le serveur FTP.

Ce type de données présente aussi un avantage pour l'utilisateur. S'il le désire, celui-ci pourrait envoyer à son Peer des données qu'il ne possède pas sur sa machine, mais qui sont stockées sur un serveur auquel il a accès. Évidemment, cela demande une certaine confiance envers le Peer qui aura accès à un compte FTP sur lequel il

pourrait théoriquement commettre des actions malveillantes, mais c'est une approche moins coûteuse en termes de mémoire que les InlineGridData, raison pour laquelle elle a été mise en œuvre lors des tests (voir chapitre 9).

Choix du moteur et paramètres

Plusieurs choix se présentaient pour la gestion du téléchargement par FTP. L'implémentation de Sun permet un téléchargement basique via la classe `java.net.URL` mais les possibilités sont très limitées et demandent une couche supplémentaire si on désire sauver le contenu dans un fichier. Cette implémentation est sous-tendue par un package `sun.net.ftp` mais celui-ci n'est ni soutenu, ni documenté [21].

Une alternative Open Source assez utilisée était edtFTPj [22]. Modulaire, bien documentée et respectant les principes de l'encapsulation, c'est cette implémentation qui a finalement été choisie. L'autre principale option était Jakarta Commons Net [23], une bibliothèque plus générale, mais correspondant moins à nos besoins.

Le protocole FTP [24] définit deux types de transfert. Le type ASCII convertit dynamiquement les caractères de fin de ligne lors du transfert de fichier. Le type BINARY copie le fichier tel quel, sans apporter de modification. Si transférer un fichier ASCII en binaire demande seulement à l'application qui va utiliser le fichier par après de s'adapter, transférer un fichier binaire en ASCII peut poser des problèmes de corruption de données. Pour éviter ce problème, les transfert de données FTP par FTPGridData sont toujours effectués en binaire.

5.1.3 TorrentGridData

Les objectifs de ce travail couvraient le développement d'un autre type de données, basé sur des transferts Peer-to-Peer. L'implémentation effectuée de ce type de données, TorrentGridData, s'appuie sur le protocole BitTorrent [25, 26]. Comme dans le cas des FTPGridData, seule de l'information nécessaire pour récupérer les données est contenue par une TorrentGridData. Cette information comprend le nom du fichier à transférer, des hashs pour le contrôle d'erreur et l'adresse d'un hôte appelé *tracker* listant les sources disposant des données et dont le fonctionnement est introduit ci-dessous. Selon les implémentations, il peut y avoir d'autres méta-informations mais elles sont toutes optionnelles. Le protocole BitTorrent permet de transférer plus d'un fichier dans un même torrent mais par cohérence avec les autres types de données, nous n'utilisons pas cette fonctionnalité.

Brève introduction à BitTorrent

BitTorrent est un protocole ouvert développé vers 2001 par Bram Cohen [25]. Le but était de créer un système Peer-to-Peer, c'est-à-dire où le téléchargement ne s'effectuerait pas d'un serveur vers un ensemble de clients, mais entre des clients tous intéressés par un même fichier. Étant donné que les clients n'ont pas toujours la possibilité de partager ou télécharger un fichier en une fois (pour des questions de connectivité et de bande passante), le fichier à partager est divisé en plusieurs pièces, ou *chunks*, qui seront les unités échangées entre deux clients. Grâce à ce principe, il est possible de réduire la charge sur le serveur, voire même de se passer de lui après un certain temps.

Par contre, l'overhead lié à un transfert de fichier est conséquent (par rapport à des protocoles classiques comme FTP) et la vitesse de téléchargement dépend fortement du nombre de clients. BitTorrent ne montre donc vraiment son utilité que pour des transferts de données relativement imposantes intéressant plusieurs clients à la fois. Des recherches [27] montrent aussi que BitTorrent a tendance à grouper les clients selon la bande passante montante dont ils disposent.

D'abord principalement utilisé pour l'échange de fichiers illicites (il est plus difficile pour les organisations de régulation de stopper un ensemble d'utilisateurs servant un fichier qu'un seul serveur), l'efficacité du protocole poussa de plus en de fournisseurs de contenu à se tourner vers lui pour réduire leurs frais sans pénaliser les consommateurs. Aujourd'hui, de nombreux logiciels libres sont distribués via BitTorrent (notamment les principales distributions GNU/Linux) et des entreprises commerciales commencent à l'utiliser pour distribuer leurs produits.

Pour publier un fichier, le premier utilisateur (*seeder*) crée un fichier `.torrent` contenant l'adresse (IP ou nom de domaine) du *tracker* ainsi que des informations sur le nombre de pièces, leurs hashs et éventuellement d'autres extensions selon la version du protocole.

Le seeder met ensuite le torrent à disposition des personnes avec qui il désire partager son fichier (par exemple sur un site Internet) et s'enregistre auprès du tracker, une entité conservant les informations nécessaires à la communication avec l'ensemble des peers intéressés par le fichier : l'*essaim* (*swarm*, en anglais).

Initialement, le seeder est le seul à posséder le fichier et commence à en envoyer des parties aux peers connectés. Quand les peers obtiennent leurs première parties, il peuvent commencer à les partager entre eux, limitant d'autant la charge du seeder qui pourra éventuellement arrêter de diffuser son fichier si suffisamment de sources le partagent.

Dans certains cas, il se peut que les peers téléchargent sans partager ce qu'ils ont déjà reçu, ils sont alors appelés *leechers*.

Le problème est que, comme pour la comptabilisation des services dans une grille, refuser totalement de partager avec les peers n'ayant pas encore eux-mêmes fourni des données risque de murer le réseau et d'empêcher des peers potentiellement intéressants de le rejoindre. Pour contrer ce phénomène, le protocole définit l'*optimistic unchoking*, qui vise à fournir une partie de sa bande passante à des peers n'ayant pas encore fourni de ressources dans l'espoir qu'ils deviennent des membres actifs.

Les avantages de ce protocole découlent du partage entre les peers. Grâce à ce mode de fonctionnement, le fournisseur n'a plus besoin d'autant de bande passante pour distribuer son ou ses fichiers car les peers contribuent à l'effort : plutôt que de servir N fois une partie, elle est servie 1 fois par N peers. Les utilisateurs ressentent eux aussi la différence : si le fichier est assez volumineux et possède assez de sources, le temps de recherche de peers auprès du tracker est souvent largement compensé par la vitesse de téléchargement. En effet, ils ne sont plus limités à un seul serveur mais peuvent télécharger le fichier depuis plusieurs sources à la fois (un chunk par source), en cumulant plutôt qu'en divisant la bande passante.

Un fait parfois écarté dans les discussions sur BitTorrent est que la réciprocité totale visée par le protocole est en pratique difficile à obtenir sur les lignes les plus courantes sur un réseau comme Internet, à savoir les lignes asymétriques. Les utilisateurs de telles lignes n'ont pas la possibilité de partager autant qu'ils ne profitent de l'essaim, lequel perd un peu en efficacité (en contrepartie, les peers les plus lents devraient partager plus longtemps, mais ce n'est pas toujours le cas).

BitTorrent est donc très adapté à l'architecture d'une grille : les peers et ressources de grille sont autant de membres potentiels de l'essaim BitTorrent.

Choix du moteur

Implémenter un client BitTorrent aurait demandé trop de temps et sortait du cadre de ce travail, il a donc fallu chercher un programme existant dont les bibliothèques pourraient être réutilisées. Malgré le succès du protocole, peu de clients Java sont disponibles sous licence Open Source.

Jtorrent [28] était une implémentation relativement avancée mais à l'abandon depuis 2003. Depuis quelques mois, les sources ne sont plus disponibles que via le CVS de Sourceforge.

Snark [29] est un projet visant à approcher GCJ [30] et BitTorrent. Il fournit les fonctionnalités de base y compris le tracking, mais le développement s'est aussi arrêté en 2003, même si les sources restent disponibles.

Le troisième projet, celui sur lequel le choix s'est porté, est Azureus [31]. Il s'agit d'un client assez imposant présentant diverses fonctionnalités comme une interface graphique complète, la gestion des torrents décentralisés (un système non encore standardisé permettant de se passer de tracker) ou l'UPnP pour la configuration dynamique des routeurs. Soutenu par une grosse communauté, son principal avantage est d'être régulièrement mis à jour.

Malheureusement, il présente aussi de nombreux défauts. Le code est assez mal documenté et difficile à lire, un fait encore amplifié par certains choix de design rappelant le C++ : les interfaces tiennent souvent lieu de headers et ne sont implémentées que par des classes nommées *NomDeL'InterfaceImpl* (une seule par interface, il est donc raisonnable de penser que le but n'est pas une future extension). Les problèmes les plus graves sont ceux qui sont pénalisants pour l'utilisateur final, à savoir :

- Le chargement en mémoire du noyau du programme est long : de l'ordre de quelques secondes sur un ordinateur récent ;
- Dès la configuration, le programme bloque la sortie standard pour y écrire ses propres messages de debug, empêchant les lanceurs vus en 4.6 de présenter des statistiques sur l'état des ressources du peer de grille ou des tâches ;
- Il ne semble pas possible de le démarrer sans une connexion directe à Internet (nécessaire a priori pour des informations de mise à jour), ce qui rend son utilisation caduque dans des réseaux privés ;
- Il crée des dossiers supplémentaires, à savoir `./plugins` et `$HOME/.azureus2` ;
- L'enregistrement de grosses quantités (plusieurs gigaoctets) de données en vue d'un partage consomme beaucoup de temps et de ressources processeur. Faute de référence, il est difficile de juger si la faute en incombe au moteur ou au protocole (les opérations impliquées, comme la division en chunks ou le calcul de hashs, sont effectivement coûteuses).

La création du répertoire de configuration `$HOME/.azureus2` a posé un problème sévère lors des expériences menées au laboratoire : les peers ne parvenaient pas à télécharger un fichier partagé et tracké par un autre peer. Ils avaient pourtant accès à toutes les informations nécessaires, le client officiel parvenant sans peine à télécharger le fichier posant problème. L'erreur provenait du fait que sur la configuration de test, le répertoire `$HOME` était partagé par NFS entre tous les peers, ce qui causait un conflit indéterminé au niveau d'Azureus. Il était donc nécessaire de déplacer le répertoire de configuration dans un emplacement non partagé (e.g. `/tmp`). Azureus n'offrant pas de méthode pour déplacer ce dossier (l'emplacement est hardcodé en plusieurs endroits et

dépend du système d'exploitation), la variable d'environnement `$HOME` est modifiée de manière programmatique pour pointer vers un répertoire convenable. Le changement est effectué localement à la JVM et n'implique donc pas de conséquences négatives pour le système.

Le choix du moteur ne doit donc pas être considéré comme définitif. Avant d'utiliser le logiciel à grande échelle, il serait bon de trouver ou écrire un moteur viable, de préférence en Java afin de pouvoir l'embarquer avec le middleware LBG. Cela permet d'avoir un plus grand contrôle sur les opérations, de conserver la portabilité du code et de faciliter le déploiement [32]. Seul le package `lbg.protocol.data` serait concerné par cette modification, plus précisément les classes GridDataManager, TorrentGridData et AzureusListener.

5.1.4 Utilisation conjointe des différents types de données

Les trois options proposées ne sont pas exclusives. Si un noeud n'a aucun intérêt à envoyer des données *inline* en conjonction avec un autre type de données (au moment où il reçoit la tâche et pourrait choisir une alternative, il possède déjà les données dans l'InlineGridData), il pourrait toutefois proposer des possibilités de téléchargement par FTP et BitTorrent ensemble, ou en offrir une plutôt que l'autre selon les cas. De même, le noeud désirant télécharger les données pourrait choisir son mode de transfert selon les restrictions de son réseau. En effet, l'utilisation abusive de BitTorrent à des fins illégales et les problèmes de sécurité liés à FTP [33] conduisent parfois les administrateurs à filtrer l'un ou l'autre de ces types de transfert.

Les administrateurs des nœuds ont donc toute marge quant aux types de données qu'ils peuvent ou veulent gérer. Ils ont la possibilité de proposer n'importe quelle combinaison de types. La question a été posée de savoir si le peer qui partage les données ne devrait pas pouvoir imposer ses options mais tant qu'il offre plusieurs possibilités, il est sans doute préférable de laisser l'administrateur de la ressource qui sera concernée décider de ce qui convient mieux à son réseau (rappelons que les peers et ressources ne sont pas forcément sous le même contrôle administratif, comme cela a été présenté dans la section 2.1).

Pour autant qu'ils correspondent au canevas (sont des sous-classes de GridData), ajouter d'autres types de données moins répandus ne demande pas de modifier d'autres classes dans le middleware LBG. Par exemple : transferts via TFTP, le réseau eDonkey2000 ou un protocole propriétaire.

Une alternative potentiellement intéressante est GridFTP [34], une extension au protocole FTP ayant pour but de mieux résister aux charges (en termes de nombre

de clients et de bande passante) propres aux applications qui consomment beaucoup de données [35]. Toutefois, la parenté de ce protocole avec FTP implique qu'il partage certaines de ses limitations, même s'il les repousse. Pour juger de la pertinence de son intégration au middleware, il serait donc utile de comparer ses performances à celles de BitTorrent.

Bien que les types de données vus plus haut aient tous une interface permettant une manipulation aisée, leur gestion n'incombe pas directement aux peers et aux ressources. Ces nœuds en délèguent la responsabilité à un gestionnaire de données fournissant des services plus avancés pour leur transfert et leur conservation.

5.2 Le gestionnaire de données

Bien qu'ayant accès à une méthode de téléchargement grâce à `GridData.toFile()`, les peers et ressources ne gèrent pas les données eux-mêmes mais délèguent la responsabilité au `GridDataManager`, ci-après désigné par l'acronyme **GDM** et conçu pour répondre aux besoins des deux types de nœuds. Maintenir un gestionnaire commun est plus contraignant que de développer des composants pour chaque type de nœud, mais limite grandement la redondance du code.

Le gestionnaire assure principalement trois fonctions : téléchargement (section 5.2.1), partage (section 5.2.2) et conversion (section 5.2.3).

5.2.1 Téléchargement

Le téléchargement des données est assuré par des méthodes spécifiques aux classes de données et dont ces dernières ont la responsabilité. Le GridDataManager leur fournit toutefois une infrastructure et des informations complémentaires si besoin est. Par exemple, le GDM peut lancer le moteur BitTorrent ou fournir l'IP locale du client ainsi que le port pour les connexions FTP actives (en fait le seul mode permis par notre architecture, les raisons en seront détaillées dans la section 5.2.2).

Toute instance de GridDataManager est capable de télécharger n'importe quel type de données grâce aux paramètres fournis à la création de l'objet (nom d'hôte, ports...).

Durant le téléchargement, le GridDataManager fournit des informations supplémentaires aux classes de données qui en ont besoin (c'est par exemple le cas de FTP [24]), mais son rôle principal est la gestion du cache. Le cache est un emplacement (dans notre cas, un répertoire fourni à l'instanciation) où sont stockés les fichiers dont on a pour

l'instant besoin, ainsi que ceux qu'on a déjà téléchargés (par exemple pour l'exécution d'une tâche précédente). Garder ces fichiers permet d'éviter de les télécharger une seconde fois si on en a besoin. Le répertoire de cache devrait être local (i.e. non partagé par un protocole transparent comme NFS) pour des raisons de performances. C'est aussi dans ce cache que le répertoire de configuration d'Azureus (voir 5.1.3) est placé. Le GDM est aussi capable de gérer des conflits (quand des données différentes partagent un même identifiant) entre des données déjà présentes en cache et de nouvelles candidates au téléchargement.

Pour gérer les conflits, le gestionnaire utilise des hashs (la classe GridData impose la présence d'une méthode le retournant). Le hash sera du type spécifié par `GridData.HASHING_ALGORITHM`. L'algorithme choisi pour l'instant est SHA-256 mais utiliser SHA-1, même s'il est maintenant vulnérable [36], ne devrait pas poser de problème de sécurité car lors d'une collision, c'est le fichier le plus récent qui est refusé. À noter que ces hashs portent sur le fichier original et non une représentation de ses parties comme les hashs de BitTorrent (voir section 5.1.3). Ces derniers ne peuvent donc être utilisés dans ce cas.

La gestion des conflits s'effectue comme suit :
Lors de la demande de téléchargement d'une donnée via la méthode `GridDataManager.getFile()`, le GDM commence par vérifier dans son cache s'il connaît déjà une donnée du même nom.

- Si des données du même nom sont déjà connues et...
  - ... si les hashs déclarés correspondent, le fichier déjà stocké est retourné. Les données ne sont pas téléchargées (*cache hit*).
  - ... si les hashs déclarés sont différents, une exception est lancée. Il pourrait s'agir d'une tentative par un utilisateur ou un peer malveillant de corrompre les données d'un autre peer, ou d'une simple erreur de la part de l'utilisateur. Étant donné qu'il est impossible de faire la distinction entre ces deux cas, les données sont refusées (*conflict*).
- Si aucune donnée de cet ID n'est répertoriée en mémoire et qu'un fichier de ce nom existe sur le disque, il pourrait s'agir de données provenant d'une session antérieure ou d'un Peer utilisant le même répertoire de cache (rien n'interdit en effet d'avoir plusieurs types de nœuds sur une même machine). Les mêmes tests de hash sont effectués (avec cette fois-ci une lecture sur disque) et ont les mêmes conséquences. S'il s'agit bien d'une récupération, les données sont ajoutées dans la structure de cache en mémoire (*cache hit*).
- Si aucune donnée correspondant à cet ID n'est répertoriée et qu'aucun fichier n'entre en conflit, les données sont copiées sur disque et les informations ajoutées dans le cache en mémoire (*cache miss*).

L'inconvénient de ce système est que si les données sont corrompues la première fois qu'elles sont envoyées (sans que cela soit détecté, par exemple si le fichier sur le disque de l'utilisateur était corrompu), il faut les retransmettre sous un autre nom ou attendre qu'elles soient éjectées du cache, ce qui n'arrivera même pas si le cache n'a pas de limite de taille.

Afin de gérer correctement les conflits, le cache présente une hiérarchie à trois niveaux.

Structure du cache

D'abord, le GDM conserve une liste de tous les fichiers qu'il connaît, soit parce qu'il les a téléchargés, soit parce qu'il les a récupérés d'une session précédente. La liste est un arbre binaire équilibré (`java.util.TreeMap`) de `CacheEntry`s. Une CacheEntry contient :

- l'identifiant de la donnée conservée ;
- sa taille ;
- son hash ;
- le fichier dans lequel elle est conservée sur disque ;
- la GridData reçue qui a engendré la création de l'entrée. S'il s'agit d'une InlineGridData, elle n'est pas conservée de manière à limiter la consommation en mémoire.

Cette liste ne comprend pas forcément tous les fichiers qui sont dans le répertoire de cache sur le disque dur : il est possible que des fichiers d'exécution précédentes du peer ou de la ressource n'aient pas encore été demandés et donc pas récupérés non plus. Ce cache n'a pas de limite de capacité.

Inclus dans ce cache se trouve le *Supplying Cache*, un cache contenant les données dont dépendent les tâches qui ont été exécutées ou sont toujours en exécution sur la ressource à laquelle appartient le GDM. Ce cache peut avoir une capacité limitée et regroupe les fichiers que le nœud ne doit pas forcément conserver indéfiniment. Par exemple, les fichiers envoyés par un utilisateur à son peer ne feront pas partie du *Supplying Cache* de ce dernier car d'autres nœuds pourraient avoir besoin du fichier à tout moment sans que le peer puisse estimer quand ils ne sont plus nécessaires. Par contre, tous les fichiers devant être stockés par une ressource sont inclus dans ce cache car après l'exécution d'une tâche, ils peuvent être supprimés si besoin est.

La limitation de taille du cache est calculée en nombre de fichiers. Il serait possible de modifier le GDM pour tenir compte d'une taille maximale en octets à condition d'implémenter dans le RMS (Resource Manager du peer) une vérification de non-dépassement de la capacité de stockage lors de chaque sélection d'une ressource, ce

qui n'est pour l'instant pas implémenté. Par comparaison, la politique actuelle dans le middleware *state of the art* (OurGrid) est de supposer un cache de capacité infinie [37], que ce soit en nombre de données ou en espace utilisé (il est toutefois prévu d'implémenter une limitation).

Enfin, le dernier niveau est le *Working Set*. Il s'agit d'un ensemble de fichiers qui ne devraient être supprimés en aucun cas et agit donc comme un verrou. Il est utilisé par une ressource pour indiquer qu'elle est en train d'exécuter une tâche qui dépend de ces données et que les effacer reviendrait à corrompre l'exécution de la tâche.

Cette politique à trois niveaux (présentée dans la figure 5.2) permet de gérer le cache de taille limitée sans interférer avec d'autres parties du middleware qui auraient besoin des données. Quand des données sont téléchargées et que le *Supplying Cache* est plein, le gestionnaire de données cherche des données appartenant à $Supplying\ Cache \backslash Working\ Set$ (grisé dans la figure 5.2), efface leur copie sur disque et les retire de ses structures de données (si les données concernées sont partagées par BitTorrent, le client Azureus est prévenu qu'il doit quitter l'essaim). Si $|Supplying\ Cache \backslash Working\ Set| > 1$, une politique de type LRU est utilisée : la GridData inutilisée depuis le plus longtemps est sélectionnée pour éjection.

Les niveaux de cache utilisent des structures de données adaptées à leur fonction. Le cache de premier niveau et le Working Set utilisent des arbres binaires équilibrés (resp. `java.util.TreeMap` et `java.util.TreeSet`) qui permettent de les manipuler en temps logarithmique en fonction du nombre d'entrées dans le cache alors que le *Supplying Cache* utilise une liste liée (`java.util.concurrent.LinkedBlockingQueue`) qui permet de gérer la politique LRU en temps constant. S'il est théoriquement possible que les performances de la liste dégénèrent en temps linéaire, la situation n'a que très peu de chances de se produire. Une telle situation implique que les entrées en fin de liste (candidates à la suppression par LRU) fassent partie du Working Set (elles ne peuvent donc être supprimées). Or, celui-ci n'est pour l'instant utilisé que sur les ressources où l'ordre des opérations de préparation des données le placera en tête de la liste (regroupant les données du *Supplying Cache* les plus récemment utilisées) plutôt qu'en fin.

Par sécurité, l'effacement des données s'effectue uniquement après la fin du téléchargement du fichier qui motive l'éjection du cache. Cela évite que l'on puisse soumettre un ensemble de fichiers corrompus pour éjecter toutes les données légitimes du cache (en supprimant les données avant le téléchargement, il suffit que ce dernier échoue pour que des données valides soient supprimées).

Sur un peer, le téléchargement est effectué dès l'arrivée de la tâche dans `LivePeerService.submitJob()`, de manière synchrone. Si le téléchargement échoue, l'utilisateur en sera averti.

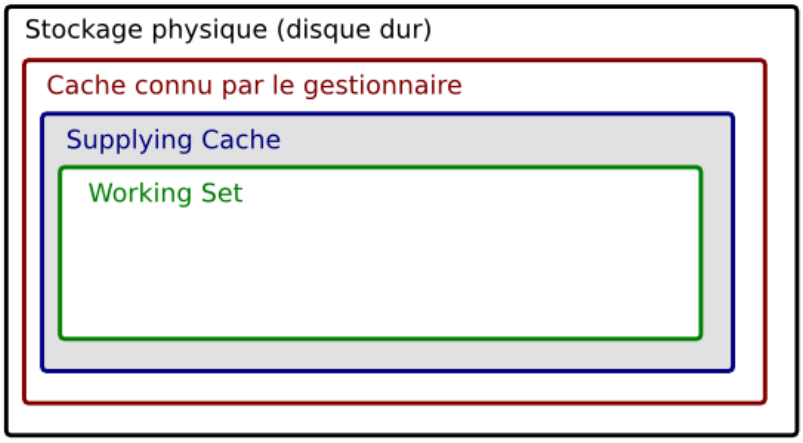


Fig. 5.2 – Cache sur le GridDataManager

Le comportement de la ressource, déjà abordé dans la section 4.4.1, est différent. Dans ce cas, lors de la soumission d'une tâche, la ressource commence par placer les données dont elle dépend dans le Working Set (le GDM vérifie que le Working Set peut être contenu intégralement dans le *Supplying Cache*). Le GDM démarre alors le téléchargement des données dans un Thread parallèle au moyen de la classe utilitaire `ThreadedDownloader`. Ce téléchargeur en parallèle permet un transfert *séquentiel* mais *asynchrone* : il ne télécharge pas plusieurs GridDatas en parallèle mais permet au Thread alors en charge du GDM de poursuivre son exécution. La ressource peut alors répondre au peer après avoir créé un LiveTaskHandler. C'est ce dernier qui attendra la fin du téléchargement des données requises avant d'enfin lancer la tâche. Si le téléchargement en parallèle a échoué entretemps, l'exécution de la tâche est annulée.

Pour limiter les opérations de création et de destruction de Threads, le téléchargeur en parallèle est toujours le même. Il s'agit d'un Thread qui est lancé une fois et attend ensuite qu'on lui fournisse des données à télécharger.

L'utilisation d'un téléchargeur en parallèle peut paraître redondante avec le LiveTaskHandler. En effet, celui-ci s'exécutant déjà dans un Thread parallèle, il pourrait télécharger les données en utilisant les méthodes synchrones du GDM sans pour autant bloquer la ressource (c'était d'ailleurs le comportement choisi au début du développement). Toutefois, prévoir une telle classe permet aussi aux peers de demander aux ressources de télécharger des données de manière proactive [38]. En modifiant le Working Set d'une ressource lui appartenant, un peer peut la forcer à télécharger un ensemble de fichiers en prévision de l'exécution ultérieure d'une tâche. Dans ce cas, un test de consistance supplémentaire est effectué afin de vérifier que le nouveau Working

Set comprend bien toutes les données dont la tâche courante a besoin (il s'agit d'un remplacement et non d'une fusion). Si le peer effectue l'appel avec un Working Set valide alors qu'un autre est en cours de téléchargement, le fichier en cours de téléchargement sera complètement transféré avant de débuter le téléchargement du nouveau Working Set, toujours en « recyclant » le Thread. De cette manière, on évite que le nouveau téléchargement entre en conflit avec l'ancien quand il est nécessaire de supprimer des entrées du cache pour libérer de l'espace.

5.2.2 Partage

Une fois les données téléchargées, le GridDataManager peut les partager. Le partage consiste en la mise à disposition par un noeud de données dont auront besoin d'autres noeuds. Cette fonctionnalité est accessible via la méthode `host(GridData)`.

Il fut décidé de séparer au maximum le GDM du reste de la grille, c'est-à-dire qu'on ne l'informe pas de la destination et de l'utilisation de ses données, et qu'il ne maintient pas ces informations par lui-même. Pour cette raison, le partage de données s'effectue sans contrôle de l'espace occupé. En effet, il est alors impossible pour le GDM de savoir quand un fichier de données n'est plus utilisé : une tâche dépendant des données que l'on voudrait supprimer pourrait toujours être en attente sur un peer. Il s'agit du cache de plus haut niveau traité dans la section précédente.

Le GDM est capable de partager les données en FTP et BitTorrent. En raison des problèmes de performances vus plus haut, il ne permet pas de partager des données *inline*. Le fonctionnement du partage dépend fortement du mode choisi, les opérations nécessaires pour chacun sont décrites ci-dessous.

FTP

Comme pour le client FTP, il a été décidé pour le serveur de se baser sur une bibliothèque externe. Le choix s'est porté sur Apache FTP Server [39] car il était écrit en Java, bien documenté et modulaire (l'interface graphique est bien distincte de la partie serveur), et en développement actif.

Le fait que le logiciel soit en développement n'a posé qu'un seul problème. Pour plus de sécurité, il était prévu de supprimer l'accès à certaines commandes. Dans la grille, un client FTP n'a en effet besoin que d'un accès en lecture seule : il serait dangereux qu'un utilisateur malveillant puisse envoyer des fichiers arbitraires sans passer par la grille car ils ne seraient pas pris en compte par le cache et resteraient donc sur l'hôte jusqu'à ce que l'administrateur s'en aperçoive (un problème rencontré dans des projets

de recherche existants [32]). Il ne doit pas non plus lister le contenu des répertoires (il connaît le nom du ou des fichiers qui l'intéressent) et n'a pas à changer de répertoire, les téléchargements du GDM s'effectuant dans un seul répertoire, « à plat ».

Apache FTP Server n'offre pour l'instant pas de moyen de limiter les commandes par des options de configuration comme le proposent d'autres serveurs plus avancés (e.g. VSFTPD [40]), mais cela semble prévu pour une version ultérieure. En attendant, des sous-classes ont été créées afin de coder la restriction « en dur » : RestrictedFtpServer et RestrictedRequestHandler. Elles délèguent la majeure partie de leurs opérations aux classes d'Apache mais ne permettent que 6 commandes FTP : `RETR`, `TYPE`, `PORT`, `USER`, `PASS` et `QUIT`.

Le protocole FTP propose deux modes de téléchargement (à ne pas confondre avec les types déjà vus en 5.1.2) : l'actif et le passif. En mode actif, le serveur est à l'écoute sur un port de contrôle sur lequel le client se connecte pour donner ses commandes. Quand un transfert de fichier est demandé, le client se met à l'écoute sur un port de données de son choix et demande au serveur de s'y connecter pour débuter le transfert (le serveur effectue une action, il est *actif*). En mode passif, le serveur écoute sur un port de contrôle dans les mêmes circonstances mais en cas de transfert, c'est lui qui ouvre un port de données sur lequel le client se connecte (le serveur est simplement à l'écoute, il est *passif*).

A priori, le mode passif est plus intéressant car il limite l'infrastructure à mettre en place. En mode passif, une personne administrant P peers et R ressources ($P \leq R$) devra ouvrir $NbPorts_{passif} = 2{*}P$ ports sur son réseau, à raison d'un de contrôle et d'un de données pour chaque peer. En mode actif, les peers ont besoin d'un port de contrôle et les ressources d'un port de données et on a $NbPorts_{actif} = P + R \geq NbPorts_{passif}$.

Toutefois, il semble que pour des raisons de performances, les serveurs FTP allouent un intervalle de ports en mode passif plutôt qu'un seul port (la RFC 959 [24] ne fixe pas le comportement à adopter) et refusent les connexions quand tous les ports de l'intervalle sont utilisés à leur maximum. Cela est problématique car il n'est pas possible au lancement d'un peer d'estimer combien de ressources pourraient s'y connecter pour télécharger un fichier, ce facteur dépendant des tâches qui seront exécutées par après.

C'est pour cette raison que le mode choisi dans le cadre de ce travail est le mode actif. Ce choix implique de fournir des informations supplémentaires à la FTPGridData lors du téléchargement par le client, à savoir un numéro de port et une adresse pour que le serveur puisse s'y connecter (le client ne peut pas toujours retrouver l'adresse automatiquement, par exemple si le réseau est NATé).

Enfin, le dernier paramètre à prendre en compte par les serveur FTP est la gestion des utilisateurs. Dans le GDM, le serveur FTP est configuré pour utiliser un seul

utilisateur dont l'identifiant et le mot de passe sont générés aléatoirement au lancement du serveur. La longueur des chaînes ainsi que l'ensemble des caractères utilisés pour ces chaînes aléatoires sont tant qu'à présent statiques (contrôlés par des variables privées) mais rien n'empêcherait de paramétriser ce comportement dans un développement futur.

BitTorrent

Le partage de données via BitTorrent implique la mise en place d'un tracker comme présenté dans la section 5.1.3.

Pour ce faire, on commence par demander à Azureus (le moteur choisi) de créer sa représentation interne d'un torrent pour le tracking puis d'en copier sur disque une version utilisable par le client (Azureus ne semble pas permettre d'effectuer cette conversion directement en mémoire). À ce moment, le peer joue le rôle de tracker, mais n'est pas encore apte à partager ses données (i.e. il peut seulement conserver une liste des peers de l'essaim).

Sur base de la représentation sur disque, on peut ensuite lancer un gestionnaire de téléchargement qui partagera les données. Lancés dans des Threads parallèles, les gestionnaires d'Azureus continueront le partage après l'exécution de la méthode.

Pour limiter l'impact en cas d'utilisation de BitTorrent (compte tenu des problèmes du moteur abordés plus haut), le core d'Azureus n'est effectivement configuré et instancié que quand le gestionnaire doit rapatrier le contenu d'un TorrentGridData ou en effectuer le tracking. De cette manière, s'il n'est jamais nécessaire de traiter des données via BitTorrent, on ne perdra pas de temps à charger le moteur inutilement.

5.2.3 Conversion

En groupant les possibilités offertes par les fonctionnalités de partage et de téléchargement, il est possible de convertir n'importe quel type de donnée en n'importe quel autre. En fait, le deuxième type n'a même pas besoin d'être différent du premier. Le Peer pourrait décider de repartager sur son propre service des données déjà accessibles sur un autre (par exemple parce que ses ressources peuvent le contacter par un lien à plus haut débit).

La conversion est une étape importante du transfert de données. Jusqu'ici, il avait été considéré que les données créées par l'utilisateur restaient intactes tout au long des transferts de tâches. Or, l'utilisateur disposant généralement d'une infrastructure moins développée que celle d'un peer, le type de données le plus approprié est l'InlineGridData

Identifiants ╲ Options	Inline	FTP	BitTorrent
U.P.A	`null`	`<ref 1>`	`null`
U.P.B	`null`	`<ref 2>`	`<ref 3>`

Tab. 5.1 – Structure des options de téléchargement de GridData

et ce type pourtant inefficace serait donc prépondérant sur la grille. Pour permettre à un utilisateur de soumettre aisément des données sans pénaliser le reste de la grille, on convertit celles-ci à l'arrivée du JobDescriptor sur le peer consommateur (c'est-à-dire dans l'appel à `LivePeerService.submitJob()`).

Au niveau de l'implémentation, la conversion est une opération « virtuelle » : elle n'existe pas en tant que méthode explicite au niveau du gestionnaire de données mais l'appel à la fonction de partage `host()` assure effectivement ces fonctionnalités.

La conversion impliquant le partage des données, la même remarque s'applique concernant la gestion de l'espace occupé sur disque : il n'est pas possible de contrôler quelles données sont périmées et peuvent être supprimées.

5.2.4 Politique de choix des données et données multiples

Comme présenté dans la section 5.1.4, il est possible de partager les données selon les différents protocoles déjà abordés (d'offrir différentes options), cela de manière non exclusive.

On désire aussi que, même si chaque GridData ne représente qu'un fichier, un utilisateur puisse soumettre des tâches dépendant de plusieurs fichiers en entrée.

À cette fin, on a décidé de permettre à une tâche de conserver des tableaux de GridDatas à deux dimensions :

- La première dimension représente toutes les GridDatas représentant des fichiers différents, dans un ordre quelconque ;
- La deuxième représente les différentes options de téléchargement pour un même fichier. Certaines de ces options peuvent ne pas exister.

Par exemple, si le Peer P d'un utilisateur U partage un premier fichier A en FTP et un deuxième en FTP et BitTorrent, le tableau de GridData sera présenté comme l'indique le tableau 5.1.

En pratique, les options sont indexées par des constantes (`GridDataManager.SHARED_*`).

Le téléchargement des différents fichiers (la première dimension du tableau à deux entrées) se fait séquentiellement. En effet, certains types de données (e.g. FTP) fonctionnent en TCP et ont donc besoin d'un quadruplet $(IP_{dest}, Port_{dest}, IP_{src}, Port_{src})$ unique par transfert de fichier. De ce fait, pour télécharger plusieurs fichiers à la fois en toute généralité (donc y compris d'un même serveur), il serait nécessaire d'utiliser plusieurs ports locaux différents. Ce principe résiste mal aux facteurs d'échelle si l'on désire contrôler les ports disponibles strictement (afin, par exemple, d'augmenter la sécurité). De plus, un téléchargement parallèle impliquerait la résolution de problèmes non triviaux au niveau de la cohérence du cache.

Gestion des options pour le partage

La gestion des options lors du partage consiste à déterminer en quels types la GridData fournie sera partagée. Comme cela a été abordé dans la section 5.2.3, la conversion, et donc le choix des options de celle-ci, se fait au moment de la soumission de Job, plus précisément dans `JobDescriptor.convert()`. Deux comportements sont alors possibles.

Si le gestionnaire a été configuré avec une politique par défaut, l'ensemble des données dont dépendent les tâches du Job sera partagé dans le ou les types déterminé(s) à l'instanciation. Dans ce cas, le partage est effectué grâce à la méthode `GridDataManager.host()`, qui retourne un tableau d'options comme décrit dans la section 5.2.4.

Si le gestionnaire n'a pas de politique par défaut, la méthode `convert()` applique une heuristique visant à limiter le temps de transfert des données [15]. Contrairement à l'approche utilisant une politique par défaut, différentes données d'une même tâche ne seront pas forcément partagées dans le même type :

- Si les données sont de petite taille (moins de 20 Mio [1]), alors l'*overhead* apporté par BitTorrent est trop important (voir section 5.1.3) et le transfert par FTP lui est préféré.
- Si les données sont de taille importante (plus de 50 Mio), on choisit BitTorrent : le gain de temps dû au partage Peer-to-Peer est supérieur à la perte de temps demandée par la découverte des peers de l'essaim.
- Si les données ont une taille intermédiaire (entre 20 Mio et 50 Mio), le choix du protocole dépendra de l'utilisation des données au sein du job :
  - Si les données sont fortement partagées (utilisées par au moins 10 tâches du Job), on choisit BitTorrent...
  - ... sinon, on utilise FTP.

[1] Un Mio (mébioctet, méga binaire) comprend $1024^2 = 1048576$ octets par opposition au Mo (mégaoctet, méga « commercial ») qui en comprend 10^6. Cette notation a été adoptée par l'IEEE dans le standard IEEE 1541.

L'heuristique effectivement implémentée dans le middleware diffère légèrement de l'heuristique présentée en ce sens que les données sont toujours partagées par FTP, même quand BitTorrent est activé. De cette manière, un administrateur pourra toujours configurer ses ressources pour préférer FTP à BitTorrent, par exemple parce que son débit montant ne lui permet pas de participer efficacement à l'essaim. Notons que les ressources doivent tout de même être capables de télécharger des données par BitTorrent, le peer ayant la possibilité d'en imposer son utilisation grâce à la première approche décrite ci-dessus.

Cette deuxième approche fait appel à la méthode `GridDataManager.host(int)`, dont l'argument entier est une disjonction binaire des flags `GridDataManager.HOST_*` qui décrivent les options possibles. Cette méthode retourne un tableau d'options comme décrit dans la section 5.2.4.

Gestion des options pour le téléchargement

Pour s'intégrer dans la grille, les ressources et les peers doivent être capables de télécharger n'importe quel type de données. On garantit ainsi que ces nœuds aient toujours accès à au moins un type de données téléchargeable. Toutefois, on permet à l'administrateur d'un nœud de spécifier sa préférence pour le téléchargement. Il pourrait par exemple décider d'utiliser FTP plutôt que BitTorrent car sa bande passante est limitée.

Au niveau du middleware, cette politique est conservée comme un tableau des différentes constantes `GridDataManager.SHARED_*` classées par ordre décroissant de préférence.

Les données *inline* ont implicitement la priorité la plus élevée : posséder une instance d'InlineGridData implique qu'on possède déjà les données. Les retélécharger en utilisant un autre protocole consommerait de la bande passante pour rien.

5.3 Chemin des données

On a pu voir que les GridData ne contiennent pas forcément les données mais seulement les méta-données nécessaires à leur récupération. C'est le cas des données FTP et BitTorrent, mais pas des *inline*. De ce fait, les données ne suivent pas toutes le même chemin sur la grille pendant leur transfert entre l'utilisateur et la ressource effectuant le calcul de la tâche.

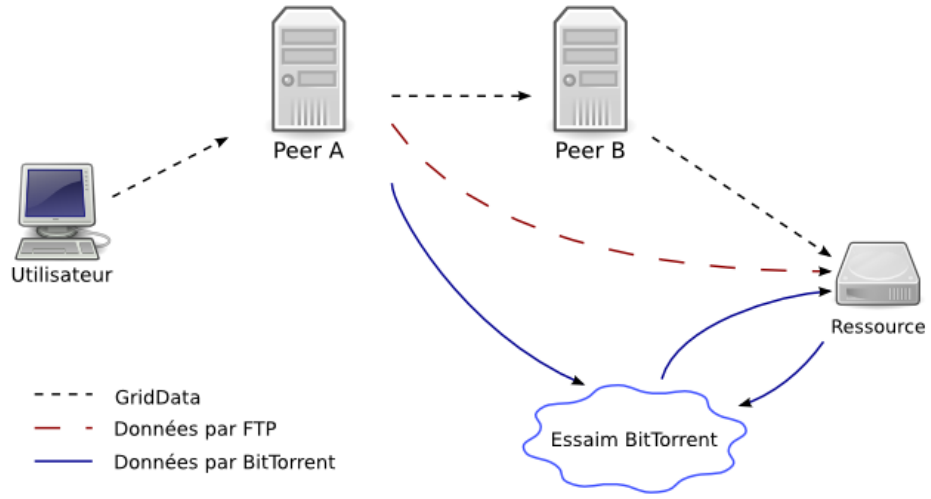


FIG. 5.3 – Chemin des données

Dans tous les cas, l'objet GridData suit *exactement* le même chemin que les tâches, vu section 4.1 (en traits courts noirs sur la figure 5.3). Il est d'abord créé par l'utilisateur et envoyé par le biais de la méthode `LiveUserPeerHandle.submitJob()` qui contacte le peer grâce au système de handles et services décrit dans la section 3.2. Il suit ensuite le même chemin que la tâche dans le peer pour en ressortir soit vers un peer producteur par `LiveExternalPeerHandle.computeSupplyingTask()`, soit vers une ressource par `LiveResourceHandle.runTask()`.

À part le cas *inline* où elles sont attachées à la GridData, les données suivent un chemin qui dépend du protocole :

- **FTPGridData :** Les données vont du peer consommateur à la ressource (en traits longs rouges sur la figure 5.3) ;
- **TorrentGridData :** Les données proviennent de l'essaim BitTorrent (en traits pleins bleus sur la figure 5.3). Le peer consommateur fait partie de l'essaim (voir section 5.1.3) : il en est initialement le seul membre et sera rejoint par des ressources.

Dans la figure 5.3, A est consommateur de temps de calcul, et fournisseur des données requises par ces calculs. B est fournisseur de temps de calcul et sa ressource est consommatrice des données. Il est possible qu'une autre ressource de B fournisse aussi ces données, auquel cas B est à la fois fournisseur et consommateur des données.

L'utilisation d'une même classe pour répondre aux besoin des peers et des ressources apporte des possibilités supplémentaires, même si elles n'ont pas été implémentées ici.

Il serait ainsi réalisable d'effectuer une deuxième conversion de données au niveau du peer fournisseur, lequel pourrait alors transférer les données à ses ressources en fonction de la topologie de son réseau. Limiter les contacts des ressources avec le réseau extérieur (e.g. Internet) offre un plus grand contrôle sur celles-ci mais dans le cas de BitTorrent, cela réduit la taille de l'essaim et donc l'efficacité du protocole.

On pourrait aussi gérer le cache des données locales de manière plus subtile, en les supprimant après un certain délai plutôt que de les garder indéfiniment. Toutefois, comme nous en avons parlé section 5.2.2, estimer ce délai de manière fiable n'est pas trivial et était tout à fait hors des objectifs définis.

Chapitre 6

Découverte de peers

Pour obtenir une grille fonctionnelle, il faut offrir un service de découverte de peers, c'est-à-dire un système qui permet aux peers de découvrir l'existence et les coordonnées de leurs semblables en vue d'entamer des négociations pour le partage de ressources.

La première architecture choisie consistait à donner à un peer la liste de tous les peers avec qui il pouvait communiquer. Cette approche est simple mais montre évidemment vite ses limites et cela même dans un environnement plus stable que la grille peer-to-peer visée par ce middleware.

Il a donc été décidé de dépasser le cadre d'un environnement de laboratoire et d'offrir un service répondant aux critères suivants :

- Il doit être possible pour un peer de s'enregistrer à n'importe quel moment de la vie de la grille. On dira que ceux-ci font partie du **registre**, une table de tous les peers enregistrés ;
- Un peer connaissant le service de découverte doit pouvoir accéder facilement à la liste des peers enregistrés auprès du service ;
- La grille n'est pas stable au cours du temps : les peers peuvent se déconnecter à tout moment. Le service ne devrait donc plus fournir leurs références à partir du moment où ils sont déconnectés.

6.1 Implémentation

L'implémentation choisie (dictée par des contraintes de temps disponible) se base sur un système client-serveur où un moteur de recherche centralisé (le serveur) traite les demandes provenant des peers (les clients). Comme les peers doivent communiquer

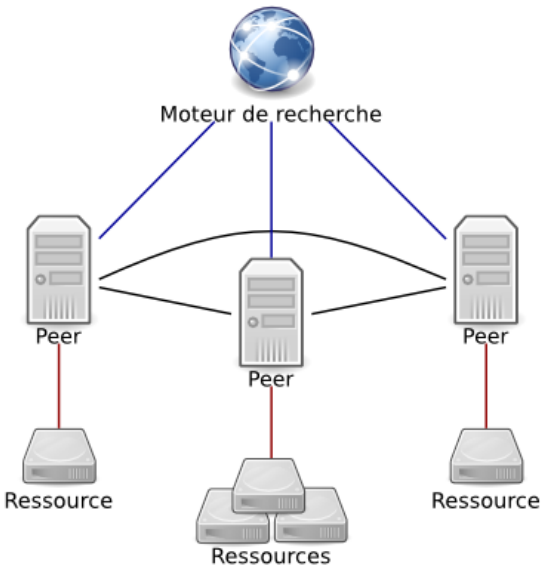


FIG. 6.1 – Moteur de recherche

entre eux pour se partager des ressources et contacter le moteur de recherche pour se connaître, on obtient une structure hiérarchique présentée dans la figure 6.1.

Pour réduire le trafic induit par des consultations fréquentes du registre de peers, le client de recherche implémente un système de cache (tout à fait indépendant du cache présenté dans le chapitre 5). Lors de tels appels successifs, seul le premier contactera effectivement le serveur, les appels suivant utilisant le cache local (mis à jour lors du premier contact). De ce fait, à un moment donné, un peer de la grille ne connaît pas forcément les mêmes peers que le serveur du service de recherche. Il est difficile de garder une vision à jour d'un système aussi sujet aux changements qu'une grille peer-to-peer [41] mais pour peu que le peer mette à jour son cache régulièrement, les effets d'un tel décalage peuvent être très faibles.

Les classes composant cette implémentation font partie du package `lbg.search-engine`.

Le client et le serveur communiquent à l'aide de handles comme décrit dans la section 3.2. Leurs implémentations sont dans le package `lbg.searchengine.service`. Notons que ce système s'est révélé suffisamment souple car l'ajout du support des communications propres au moteur de recherche n'a demandé que peu de modifications dans le package `lbg.protocol.rpc` (elles n'avaient pas été prises en compte lors du développement du système RPC vu l'implémentation tardive du service de découverte de peers).

6.1.1 Serveur

Le serveur est représenté par la classe `SearchEngineServer`. Il fournit des méthodes répondant aux besoins des trois critères cités dans l'introduction de ce chapitre, à savoir l'ajout de peers au registre, leur suppression et la consultation du registre.

Ajout

L'ajout d'un peer au registre se fait grâce à l'ajout de son `ExternalPeerHandle`, via la méthode `add()`. Par sécurité, le moteur de recherche n'ajoute au registre que des peers pour lesquels la méthode `ExternalPeerHandle.isAlive()` renvoie `true`. On n'insérera donc dans le registre que des peers joignables au moment de l'ajout.

Si un peer tente d'ajouter une handle avec un ID déjà connu, celle-ci est refusée. On évite ainsi qu'un peer ne tente de prendre la place d'un autre mais cela implique que si un peer change d'adresse, son ancienne handle devra être supprimée du registre avant qu'il puisse fournir sa nouvelle.

Suppression

Pour des raisons de sécurité, la suppression d'un peer du registre du serveur n'est pas une opération disponible depuis l'extérieur. Afin de déterminer quels peers se sont déconnectés, le serveur vérifie périodiquement l'état de l'ensemble des peers connus. Si un peer n'est pas joignable par le serveur, sa handle sera retirée du registre. On ne fait en effet pas appel aux connaissances des peers du registre quand il s'agit de déterminer la connectivité d'un autre peer (le service est totalement centralisé).

Consultation

A priori, l'opération de consultation du registre se résume à retourner l'ensemble des handles connues (par exemple sous la forme d'un tableau).

Toutefois, nous avons dit que les clients conservaient un cache local pour limiter le nombre de transferts. Ce cache peut aussi servir à limiter la taille des transferts. En effet, si le cache d'un client a été mis à jour à un instant donné, on peut se limiter à lui fournir les handles des peers qui se sont enregistrés depuis : il connaît déjà les autres.

Pour implémenter cette optimisation, il faut adapter la structure du registre ainsi que les signatures des méthodes.

Le registre est donc redéfini comme une liste de paires (H_i, T_i), où T_i est l'instant auquel un peer a enregistré sa handle H_i après du service de découverte.

On définit alors deux méthodes pour récupérer les handles :

1. `getHandles()` retourne l'ensemble des handles connues. Cette opération reste utile pour un client qui n'a encore jamais contacté le serveur ;
2. `getHandles(long t)` retourne les handles qui ont été enregistrées depuis l'instant t.

Plutôt que de retourner un tableau de handles, ces méthodes renvoient une instance de `SearchEngineResponse`, laquelle comprend les handles ainsi que le moment auquel la requête a été traitée. De cette manière, les horloges du client et du serveur n'ont pas besoin d'être synchronisées : tous les temps de référence sont basés sur l'horloge du serveur.

Structure du registre

En interne, le registre est conservé grâce à trois structures de données.

D'abord, une table de hachage (`java.util.HashMap`) permet de vérifier en temps constant qu'un peer n'est pas encore connu.

Ensuite, deux listes handles_list et entry_times (basées sur des tableaux en utilisant `java.util.ArrayList`) conservent les handles des peers enregistrés et leurs moment d'entrée, c'est-à-dire que

$\forall i$, $handles_list[i]$ a été enregistré au temps $entry_times[i]$

et

$\forall i, j, i < j, entry_times[i] \leq entry_times[j]$

Les `ArrayList`s fournissant des opérations d'indexation en temps constant, la consultation de l'intégralité du registre de N entrées prendra un temps $O(N)$.

La consultation via la méthode `getHandles(long)` pouvait être effectuée de deux manières : on peut parcourir entry_times pour trouver la première handle intéressante (celle qui a le plus petit temps d'entrée supérieur à l'argument passé à la méthode) depuis le début ou la fin. Rappelons qu'entry_times est une suite croissante.

Soit M le nombre de handles à retourner. Si on parcourt la liste depuis le début, trouver la première handle prendra un temps $O(N - M)$. Si on parcourt la liste depuis la fin, cela prendra un temps $O(M)$. Si les clients du moteur de recherche

sont régulièrement mis à jour, on aura $M \ll N$: seule une faible partie du registre doit être retournée (c'est le but du cache sur le client).

Au final, la méthode `getHandles(long)` prend donc un temps $O(2M)$.

L'ajout de handles dans les `ArrayList` est définit comme se faisant en temps amorti, c'est-à-dire qu'ajouter K éléments se fait en un temps $O(K)$. La suppression est l'opération la plus coûteuse : elle prend un temps $O(N)$. Toutefois, en situation normale, le nombre de suppressions et d'ajouts devrait être largement inférieur au nombre de consultations.

6.1.2 Client

Le client est implémenté par la classe `SearchEngineClient`. Comme le serveur, il possède des méthodes permettant l'ajout et la suppression de handles ainsi que la consultation du registre.

Ajout

La méthode d'ajout n'a pas pour but de contacter le serveur pour lui faire parvenir la handle du peer gérant le client (cette opération est effectuée automatiquement lors de la connexion initiale au serveur) mais bien de modifier le cache local pour y ajouter une handle que le serveur ne connaît peut-être pas, par exemple parce que ce peer ne désire partager ses ressources qu'avec un ensemble de peers auxquels il a donné ses coordonnées (adresse IP et ports).

Nous avons indiqué dans la section 6.1 que le client n'avait pas forcément la même vision de la grille que le serveur. La sémantique de la méthode d'ajout que nous venons de définir implique que non seulement le cache peut ne pas contenir toutes les handles connues par le serveur, mais il peut aussi contenir des handles que le serveur ne connaît pas.

Notons que bien que cette méthode soit implémentée et fonctionnelle, elle n'est pas encore utilisée par le middleware. De ce fait, une handle ne sera inconnue du serveur que si celui-ci l'a supprimée de son registre et que le peer gérant client ne l'a pas encore effacée de son cache local.

Suppression

L'optimisation consistant à ne transférer qu'une partie des handles lors de la mise à jour (voir section 6.1.1) implique qu'il n'est pas possible de se baser sur le serveur pour éliminer les handles périmées du cache du client. De ce fait, c'est au peer qu'il incombe de supprimer les handles des peers qu'il n'a pas réussi à contacter. Cette suppression est possible grâce à la méthode `remove()`.

Consultation et mise à jour

Le client présente différentes méthodes pour la consultation du registre de peers.

- `search(String)` permet de chercher la handle d'un peer dont on connaît l'identifiant. Si cette handle n'est pas en cache, le client met celui-ci à jour et revérifie avec la dernière version du registre. En plus de cette mise à jour effectuée suite à un cache miss, une mise à jour régulière est effectuée de manière périodique
- `search(int)` retourne un nombre donné de handles aléatoires provenant du cache du client de recherche.
- `search()` fournit le même résultat que `search(int)` en utilisant un nombre de handles par défaut (défini par `SearchEngineClient.DEFAULT_SEARCH_COUNT`).
- `searchAll()` retourne l'ensemble des handles contenues dans le cache sous la forme d'un tableau.

Afin d'améliorer les performances, le tableau retourné par `searchAll()` est mis en cache de la façon suivante :

- Si le cache du client n'a pas été modifié depuis la dernière création du tableau, ce dernier est retourné. Par cache du client, on entend la structure qui conserve les handles fournies par le serveur et est mise à jour de manière périodique ou par un appel à `search(String)`.
- Si le cache du client a été modifié, il est nécessaire de regénérer le tableau.
  - Si la taille du précédent tableau suffit à stocker l'ensemble des entrées courantes, il est réutilisé.
  - Sinon, un tableau de taille au moins deux fois supérieure au précédent tableau est créé. En doublant ainsi la taille à chaque fois, on réduit le nombre de fois que ce cas sera rencontré et économise le temps nécessaire à toutes ces allocations. L'allocation exponentielle implique que tous les éléments alloués ne sont pas forcément utilisés. Pour pouvoir déterminer à quel moment terminer un parcours de ce tableau, l'élément suivant la dernière handle est mis à `null` s'il existe.

La sélection aléatoire de la méthode `search(int)` est opérée en générant des indices dans le tableau en cache (le même que celui de `searchAll()`) grâce à une distribution uniforme fournie par la classe `lbg.common.CommonMath`. Si une entrée du tableau a déjà été prise en compte, on prend l'entrée suivante et ainsi de suite jusqu'à ce que l'on trouve une entrée qui n'a pas encore été prise en compte.

Cette technique est similaire à l'adressage ouvert par sonde linéaire dans les tables de hachage et souffre du même problème de génération de *clusters* d'entrées utilisées : si le nombre d'entrées à rechercher est proche du nombre d'entrées contenues dans le cache, la méthode dégénère. Toutefois, dans le cadre de ce travail, `search(int)` est principalement utilisée afin de prélever des échantillons de taille réduite et ce problème ne devrait donc pas être significatif.

Changement de moteur de référence

Le client possède aussi une fonctionnalité qui n'est pour l'instant pas exploitée (car elle demande un support en d'autres points de la grille, notamment au niveau du peer). Il lui est possible, via la méthode `renewReferenceServer()`, de changer de serveur de référence en cours d'exécution.

Comme dans le cas de l'ajout, cette méthode est implémentée mais n'est pas encore utilisée par le middleware. Son utilisation ne demanderait toutefois la modification d'aucune classe du middleware à part la ou les classes appelantes.

6.2 Limites

L'implémentation décrite présente un problème propre à tous les modèles client-serveur simples : le serveur est unique et non répliqué. De ce fait, il subit seul la charge du service de découverte et résiste sans doute mal aux facteurs d'échelle (dans le cas d'une grille de plusieurs centaines de peers).

Le serveur représente un *Single Point of Failure*. S'il venait à lâcher, l'ensemble de la grille serait dans une situation précaire : les caches permettraient de continuer les négociations entre peers connus mais au fil des déconnexions des peers, la grille se morcellerait. Ce problème est apparent dans la figure 6.1.

La section 10.3.2 propose des solutions qui résoudraient ce problème à divers degrés, mais leur implémentation sort du cadre de ce travail.

Chapitre 7

Gestionnaire de fichier journal

Dans le cadre du *debugging* et du déploiement du système de grille, il peut être intéressant de conserver un journal des différents événements. On peut ainsi savoir pourquoi une exécution échoue ou simplement garder trace des différents événements comme l'arrivée des résultats d'une tâche soumise par un utilisateur ou l'état du transfert de données (ces informations furent très utiles dans le cadre du *debugging* du middleware).

Plusieurs solutions répondent à ce besoin.

D'abord, on peut tout imprimer sur la sortie standard (ou sur la sortie d'erreur). La procédure est simple et bien connue mais ne permet pas de distinguer les logs provenant de différentes parties de l'application et demande de faire appel au système d'exploitation si on désire rediriger les logs (e.g. dans un fichier).

Ensuite, il existe des solutions plus élaborées, la plus utilisée étant Log4j [42]. Log4j permet de filtrer les messages selon leur importance (simple information, avertissement, erreur grave) et de les rediriger selon leur source dans des fichiers, sur des serveurs distants... Développée par IBM en 1996 puis reprise par le projet Apache, cette implémentation a acquis une maturité certaine et est utilisée dans de nombreuses applications aussi bien libres (e.g. Azureus) que commerciales (e.g. Adobe Flex [43]).

La dernière solution envisagée est l'API de logging disponible depuis Java 1.4. Le package `java.util.logging` a pour but de reproduire les fonctionnalités Log4j pour les intégrer au standard Java. La principale différence avec ses concurrents est qu'il dispose de moins d'outils pour le formattage et la redirection mais, comme ses concurrents, il est extensible et les fonctionnalités inexistantes peuvent donc être fournies par des développeurs tiers. C'est cette solution qui a été choisie pour LBG car elle ne demande pas d'inclure de sources d'un autre projet et suffisait à nos besoins.

L'implémentation de Sun se base sur l'utilisation de *Loggers*, *Handlers* et *Formatters*. Les Loggers sont identifiés par une chaîne de caractères représentant une hiérarchie similaire aux packages (*parent.fils.petit-fils*). Les Handlers sont chargés de la redirection des messages (sortie standard, fichier...) et confient la présentation des messages aux Formatters (les spécifications incluent pour l'instant un formattage en texte brut et un en XML).

Les messages peuvent provenir de différents points du middleware LBG. Les différentes sources et leurs Loggers sont :

- `lbg.peer :` liste les erreurs non fatales lors des appels réseaux, par exemple parce que les données fournies par l'appelant sont invalides (les erreurs fatales relèvent d'un bug ou d'une corruption irrécupérable des structures de données internes) ;
- `lbg.resource :` regroupe les informations sur le début et la fin de l'exécution d'une tâche, et pour qui la tâche est exécutée ;
- `lbg.user :` regroupe les informations concernant la complétion des jobs et tâche, notamment la durée d'« exécution » (aux yeux de l'utilisateur, il s'agit du temps pendant lequel sa tâche a transité sur la grille depuis sa soumission) et l'emplacement où ont été sauvées les données de sortie ;
- `lbg.data :` regroupe les informations sur le début et la fin du téléchargement ;
- `lbg.search :` liste les événements sur le moteur de recherche.

Ce système permet à un utilisateur de spécifier exactement quels messages il souhaite afficher et comment les formater grâce à un fichier de configuration au format *Properties* (la configuration par défaut dépend des implémentations de la machine virtuelle). Le listing 7.1 présente un fichier de configuration applicable à LBG et permettant de n'afficher sur la console que les erreurs et les avertissements, sauf au sujet des données, pour lesquelles on désire des informations détaillées.

```
handlers = java.util.logging.ConsoleHandler
# Let the console show all
java.util.logging.ConsoleHandler.level = ALL
# Filter out information messages, application-wide...
.level = WARNING
# ...except for data
lbg.data.level = FINEST
```

Listing 7.1 – Un exemple de configuration de journalisation

Chapitre 8

Utilisation de la grille

Même s'il n'est sans doute pas encore approprié pour une utilisation à grande échelle (notamment à cause de problèmes de sécurité qui seront discutés dans le chapitre 10), le middleware LBG est déjà opérationnel. Ce chapitre présente la mise en place d'une grille basée sur le middleware LBG par des administrateurs (section 8.1) ainsi que la soumission de tâches de calcul à celle-ci par des utilisateurs (section 8.2).

8.1 Mise en place du middleware

Le déploiement du middleware nécessite une machine virtuelle compatible Java 5 (seule l'implémentation de référence de Sun a été testée). Il est aussi nécessaire de télécharger les bibliothèques (au format `.jar`) d'Azureus [31], edtFTPj [22], Apache FTP Server [39] ainsi que commons-logging et commons-math [44].

Le répertoire du programme se compose des fichiers et répertoires suivants :

- **src/** : les sources du middleware ;
- **doc/** : sa documentation ;
- **classes/** : emplacement où sera placé le code compilé ;
- **jars/** : emplacement où sera placée l'archive du code compilé et où les dépendances doivent être copiées ;
- **build.xml** : fichier de configuration de compilation Ant [45] (voir section 8.1.1).

dist	Compile le middleware sans *warnings*
dist-dev	Compile le middleware avec *warnings*
doc	Compile la documentation avec Javadoc, sans les membres privés
doc-dev	Compile la documentation avec Javadoc, avec les membres privés
doxydoc	Compile la documentation avec Doxygen [46]
gridhasher, hellogrid	Compile les exemples
clean	Supprime le code compilé
cleaner	Supprime le code compilé et la documentation javadoc

TAB. 8.1 – Options de compilation Ant

8.1.1 Compilation

La compilation du middleware s'effectue à l'aide d'un fichier de configuration Ant [45] `build.xml`. Ant est un outil de compilation similaire à *make*, mais spécialisé pour la compilation Java.

Une fois les archives prérequises (voir ci-dessus) placées dans le répertoire `jars/`, il suffit d'entrer `ant dist` depuis le répertoire racine du projet pour effectuer la compilation. Celle-ci créera un répertoire `classes/` comprenant le bytecode ainsi qu'un fichier jar ayant la même fonction.

À l'image d'un Makefile, le fichier build.xml peut aussi proposer plusieurs autres options de compilation. Un bref descriptif est disponible grâce à la commande ant −projecthelp, une partie des commandes disponibles étant résumée dans le tableau 8.1.

8.1.2 Lancement des nœuds

Une fois le code compilé, on peut procéder au lancement des nœuds de traitement des tâches, à savoir les peers qui géreront les tâches et les ressources qui les exécuteront.

Tous les programmes acceptent une option `-?` qui résume les options disponibles pour le lancement. Seules les options principales sont décrites ici.

Moteur de recherche

Avant de lancer les peers, il est nécessaire de démarrer le service de découverte (voir section 6), sans quoi les peers ne sauraient procéder à aucun échange de ressources.

Cette opération ne devant être effectuée qu'une fois par grille, les administrateurs des peers n'auront généralement pas à s'en soucier. Le listing 8.1 présente une commande permettant le lancement du serveur.

```
java -cp "classes/" -ea lbg.middleware.LaunchSearchEngine\
    --id id_du_moteur --host hote_du_moteur -p port_d_ecoute_du_moteur
```

Listing 8.1 – Déploiement du moteur de recherche

Le nom d'hôte du moteur est l'adresse à laquelle le moteur pourra être contacté par les peers auxquels il accepte de fournir le service. Dans un environnement NAT, il faudra donc fournir l'IP externe du routeur si on désire accepter les connexions de peers extérieurs.

Peers

Les peers peuvent alors être lancés par les administrateurs grâce à la commande présentée dans le listing 8.2.

```
java -cp "classes/:jars/commons-math-1.1.jar:jars/Azureus2.5.0.4.jar\
    :jars/commons-logging-1.1.jar:jars/ftpserver-dev.jar:./" -ea\
    lbg.middleware.LaunchPeer -c peerconfig.properties
```

Listing 8.2 – Déploiement d'un peer

Le seul argument requis (`-c FILE`) est l'emplacement du fichier de configuration du peer. Au départ, tous les paramètres étaient disponibles sur la ligne de commande mais au fil de l'affinement des politiques d'ordonnancement ainsi que de l'extension des possibilités, cette approche a montré ses limites.

Les différents paramètres d'un peer sont les suivants :

- ID du peer, hôte et ports par lesquels les autres nœuds (peers, ressources et utilisateurs) peuvent le contacter (voir listing 8.3).

```
PEER_ID = candi02
PEER_HOSTNAME = candi02.montefiore.ulg.ac.be
PEER_PORT = 60400
RESOURCE_PORT = 60401
USER_PORT = 60402
```

Listing 8.3 – Configuration d'un peer : contact

- Ports disponibles pour la gestion de données, politique pour le transfert de données et gestion du cache. La politique proposée ici est de télécharger les données par FTP en priorité, et de les partager uniquement par BitTorrent. Si la propriété `DATA_HOST_POLICY` n'est pas spécifiée, alors l'heuristique présentée dans la section 5.2.4 est utilisée.
  Le répertoire de cache englobe tous les niveaux présentés dans la section 5.2.1. S'il n'existe pas déjà, il sera créé. La limitation de taille ne concerne que le *Supplying Cache*, lequel n'est pas encore utilisé sur le peer (pour rappel, le *Supplying Cache* conserve les fichiers n'appartenant pas aux utilisateurs du peer). Ces options sont présentées dans le listing 8.4.

```
FTP_DATA_PORT = 60403
FTP_REQUEST_PORT = 60404
BITTORRENT_TRACKER_PORT = 60405
BITTORRENT_DATA_PORT = 60406
DATA_RCV_POLICY = FTP, TORRENT
DATA_HOST_POLICY = TORRENT
CACHE_DIR = /tmp/peer/
SUPPLYING_CACHE_CAPACITY = 10
```

Listing 8.4 – Configuration d'un peer : données

- Spécification de l'adresse à laquelle le moteur de recherche peut être contacté ainsi que de la fréquence (intervalle en secondes) à laquelle celui-ci doit être contacté pour demander de nouvelles informations (voir listing 8.5).

```
SEARCH_HOSTNAME = candi42.montefiore.ulg.ac.be
SEARCH_PORT = 60411
SEARCH_TIMEOUT = 30
```

Listing 8.5 – Configuration d'un peer : moteur de recherche

- Politique de gestion et d'ordonnancement du peer. Ces propriétés définissent précisément le comportement du peer (voir chapitre 3). Des valeurs d'exemple sont proposées dans le listing 8.6.

```
PEER_DEFAULT_ACCOUNTANT = OGTimeAccountant
PEER_TARGET_NEGOTIATION_SUPPLYING_BEHAVIOR =
    FavorsSupplyingNegotiation
PEER_TARGET_NEGOTIATION_CONSUMPTION_BEHAVIOR =
    DataConsumptionNegotiation
PEER_TARGET_SCHEDULING_LOCAL_BEHAVIOR = PreemptiveLocalScheduling
PEER_TARGET_SCHEDULING_PREEMPTION_BEHAVIOR = LimitedCancellation
PEER_TARGET_SCHEDULING_SUPPLYING_BEHAVIOR =
    QAwareFavorsSupplyingScheduling
```

```
PEER_TARGET_SCHEDULING_CONSUMPTION_BEHAVIOR =
    FavorsConsumptionScheduling
PEER_TTG_POLICY = true
PEER_STORAGE_AFFINITY = true
PEER_DATA_REPLICATION = true
PEER_IDLE_REPLICATION_RATIO = 0.5
REQUESTORS_COUNT_THRESHOLD = 1
REQUESTORS_TIME_THRESHOLD = 20
GRANTORS_COUNT_THRESHOLD = 1
GRANTORS_TIME_THRESHOLD = 20
REQUEST_SUPPLYING_TIME_THRESHOLD = 30
```

Listing 8.6 – Configuration d'un peer : comportement

Le peer vérifie l'ensemble des options avant de démarrer. S'il rencontre une erreur, il indiquera quelle propriété est concernée. L'effet des politiques sort du cadre de ce document.

Ressources

Les ressources peuvent être démarrées d'une manière similaire au peer, grâce à la commande du listing 8.7.

```
java -cp "classes/:jars/commons-math-1.1.jar:jars/Azureus2.5.0.0.jar\
    :jars/edtftpj.jar:jars/commons-logging-1.1.jar\
    :jars/ftpserver-dev.jar:./" -ea lbg.middleware.LaunchResource\
    -c resourceconfig.properties
```

Listing 8.7 – Déploiement d'une ressource

Le fichier de configuration est semblable à celui du peer. On y trouve les informations pour contacter le peer et la ressource (celle-ci les fournit au peer quand elle s'y connecte), la politique de téléchargement et les informations concernant le cache. Une propriété précise aussi la puissance de la ressource (voir section 3.3). Des valeurs d'exemple pour ces propriétés sont proposées dans le listing 8.8.

```
RESOURCE_ID = candi03
RESOURCE_HOSTNAME = candi03.montefiore.ulg.ac.be
RESOURCE_PORT = 60410
RESOURCE_POWER = 50
PEER_HOST = candi01.montefiore.ulg.ac.be
PEER_PORT = 60401
CACHE_DIR = /tmp/res/
DATA_RCV_POLICY = FTP, TORRENT
```

```
SUPPLYING_CACHE_CAPACITY = 3
BITTORRENT_DATA_PORT = 60408
FTP_DATA_PORT = 60409
```

Listing 8.8 – Configuration d'une ressource

Vers un déploiement automatique

Les méthodes de lancement décrites ci-dessus sont fonctionnelles mais, dans la plupart des cas, un peer aura plus d'une ressource à sa disposition. S'il est du ressort de l'administrateur du peer de lancer les ressources sur un réseau (intranet) de machines, créer manuellement l'ensemble des fichiers de configuration peut être fastidieux.

Pour cette raison, les ressources acceptent sur la ligne de commande des arguments permettant de surcharger les identifiant et nom d'hôte trouvé dans le fichier de configuration, lesquels seront alors ignorés. L'administrateur peut donc copier un même fichier de configuration sur toutes ses ressources ou le partager via NFS [20]. Les intranets étant souvent composés de machines de configuration homogène, il est probable que les ressources adoptent toutes la même politique de traitement des données

Quand les fichiers sont copiés, un simple script shell (le listing 8.9 utilise la syntaxe de bash) permet de lancer l'ensemble des ressources :

```
for RES in $(seq -f candi%02g 3 15); do
    ssh candi$RES.montefiore.ulg.ac.be "java -cp $CLASSPATH \
        lbg.middleware.LaunchResource -q -c $CONFIG -i candi$RES \
        -h candi$RES.montefiore.ulg.ac.be \
        > /var/log/lbg/res_$RES 2>&1 &" > /dev/null 2>&1
done
```

Listing 8.9 – Déploiement d'un parc de ressources

On suppose ici que la variable `$CLASSPATH` contient un classpath commun à toutes les ressources et déjà déployé. De même, la variable `$CONFIG` contient le chemin vers le fichier de configuration.

Lors des tests réalisés (leurs résultats seront présentés dans le chapitre 9), de telles options ont aussi été utilisées pour lancer plus d'un parc de ressources (donc plus d'un peer) en une fois.

Une fois la grille mise en place, on peut commencer à soumettre des tâches à celle-ci.

8.2 Soumission de tâches à la grille

Cette section décrit la marche à suivre pour un programmeur désirant soumettre une tâche de calcul à la grille, de la rédaction de la tâche à la procédure de soumission.

8.2.1 Écriture d'une tâche

Pour commencer, il faut écrire la tâche que l'on souhaite exécuter. À des fins d'exemple, nous allons reprendre en détails la création de `lbg.examples.hash.GridHasher`. Cette classe calcule les condensés de ses données d'entrée grâce à l'algorithme spécifié dans ses paramètres d'entrée.

La première étape est d'implémenter l'interface GridApplication (voir listing 8.10), vue section 4.7 et qui définit la classe principale, celle qui sera lancée par le middleware sur la ressource. GridApplication impose l'implémentation de 4 méthodes : `setParameters()`, `setInputData()`, `compute()` et `getResult()`.

```
public class GridHasher implements GridApplication {
    public GridHasher() {}
    public void setParameters(Object[] parameters) { /* ... */ }
    public void setInputData(GridData[] datas) { /* ... */ }
    public void compute() { /* ... */ }
    public byte[] getResult() { /* ... */ }
}
```

Listing 8.10 – Écriture d'une tâche : implémentation de GridApplication

La méthode `setParameters()` (listing 8.11) va nous permettre de spécifier quel algorithme doit être utilisé pour calculer le condensé. Pour le développement de ce programme, nous conservons deux variables d'instance md et result permettent de conserver respectivement l'algorithme de hachage choisi et les données de sortie de la tâche.

```
public void setParameters(Object[] parameters) {
    try {
        md = MessageDigest.getInstance((String)parameters[0]);
    } catch (java.security.NoSuchAlgorithmException e) {
        result = e.getMessage().getBytes();
    }
}
```

Listing 8.11 – Écriture d'une tâche : setParameters()

En cas d'erreur, on stocke le message dans le résultat : la tâche s'exécutant à distance, on n'a pas accès aux logs en cas d'erreur (il est toutefois prévu d'étendre le peer en ce sens).

La méthode `setInputData()` (listing 8.12) fournit à la tâche ses données d'entrée. Nous les conservons dans un champ supplémentaire.

```
public void setInputData(GridData[] datas) {
    this.datas = datas;
}
```

Listing 8.12 – Écriture d'une tâche : setInputData()

La méthode `compute()` (listing 8.13) est celle qui sera appelée pour démarrer effectivement la tâche, c'est elle qui va effectuer le calcul. Dans notre cas, il s'agit de calculer le condensé de chaque donnée passée en entrée et de les conserver dans la variable result.

Notons qu'on a ici accès aux méthodes utilitaires disponibles dans le middleware ; celle qui nous intéresse ici est `GridDataManager.hashToString()`.

```
public void compute() {
    String output = "";

    for (GridData data: datas) {
        try {
            InputStream is = data.getInputStream();

            // Hash
            byte[] buffer = new byte[BUFFER_SIZE];
            int read = is.read(buffer, 0, BUFFER_SIZE);
            while (read != -1) {
                md.update(buffer, 0, read);
                read = is.read(buffer, 0, BUFFER_SIZE);
            }
            byte[] hash = md.digest();

            // Make a nice output
            output +=
                lbg.protocol.data.GridDataManager.hashToString(hash)
                + "\t" + data.id + "\n";
        } catch (Exception e) {
            result = (e.getMessage() + "\n").getBytes();
            return;
        }
    }
```

```
    result = output.getBytes();
}
```

Listing 8.13 – Écriture d'une tâche : compute()

Enfin, il reste à coder la méthode qui sera appelée pour récupérer les données de sortie de la tâche (listing 8.14).

```
public byte[] getResult(){
    return result;
}
```

Listing 8.14 – Écriture d'une tâche : getResult()

En ajoutant les imports, la gestion des erreurs et les champs utilisés durant l'exécution, on obtient la classe présentée dans le listing 8.15.

```
// GridHasher.java

package lbg.examples.hash;

import java.io.InputStream;
import java.security.MessageDigest;

import lbg.protocol.api.*;
import lbg.protocol.data.*;

/** Hash a set of data and return a user-readable hash.
 * @author Xavier Dalem */
public class GridHasher implements GridApplication {
    static private int BUFFER_SIZE=1024;
    private byte[] result = null;
    private GridData[] datas;
    private MessageDigest md = null;

    public GridHasher() {}

    public void setParameters(Object[] parameters) {
        if (parameters == null) {
            result = new String("null parameters\n").getBytes();
            return;
        }
        if (parameters.length != 1) {
            result = new String("Invalid number of parameters: "
                + parameters.length + "\n").getBytes();
            return;
        }
```

```
        try {
            md = MessageDigest.getInstance((String)parameters[0]);
        } catch (java.security.NoSuchAlgorithmException e) {
            result = e.getMessage().getBytes();
        }
    }

    public void setInputData(GridData[] datas) {
        this.datas = datas;
    }

    public void compute() {
        if (result != null)
            return;
        if (md == null) {
            result = new String("null md\n").getBytes();
            return;
        }

        String output = "";

        for (GridData data: datas) {
            try {
                InputStream is = data.getInputStream();

                // Hash
                byte[] buffer = new byte[BUFFER_SIZE];
                int read = is.read(buffer, 0, BUFFER_SIZE);
                while (read != -1) {
                    md.update(buffer, 0, read);
                    read = is.read(buffer, 0, BUFFER_SIZE);
                }
                byte[] hash = md.digest();

                // Make a nice output
                output +=
                    lbg.protocol.data.GridDataManager.hashToString(hash)
                    + "\t" + data.id + "\n";
            } catch (Exception e) {
                result = (e.getMessage() + "\n").getBytes();
                return;
            }
        }

        result = output.getBytes();

        try { Thread.sleep(2000); } catch (InterruptedException e) {}
    }
```

```
    public byte[] getResult(){
        return result;
    }
}
```

Listing 8.15 – Écriture d'une tâche : GridHasher

La tâche est maintenant écrite. Une fois compilée et placée dans un fichier jar, on peut procéder à sa soumission pour exécution sur la grille.

8.2.2 Lancement d'un utilisateur pour soumission

Le middleware LBG ne permet pas la soumission de tâches de calcul telles quelles : elles doivent faire partie d'un job, c'est-à-dire d'un ensemble de tâches. Rien n'empêche toutefois de soumettre un job d'une seule tâche.

Pour soumettre un Job, on en rédige une description au format de propriétés Java. Des exemples sont disponibles dans le répertoire `tests/` des sources, le listing 8.16 en présente un autre (commenté dans le texte).

```
# Une liste des taches a executer, les noms sont arbitraires.
# Ici, nous n'executerons qu'une tache.
task_list = example_hasher

# Le fichier jar contenant le bytecode de la tache a executer.
example_hasher.jar = jars/gridhasher.jar
# Le nom complet de la classe implementant GridApplication
example_hasher.main_class = lbg.examples.hash.GridHasher
# Les parametres d'entree. Ici, il s'agit de l'algorithme choisi
example_hasher.parameters = SHA-1
# Les fichiers dont il faut calculer les condenses
example_hasher.datas = mon_premier_fichier, mon_deuxieme_fichier
```

Listing 8.16 – Soumission d'une tâche : écriture du descripteur

Une fois que la description est écrite, on peut procéder à la soumission du descripteur de Job. Pour ce faire, on crée un *utilisateur*. Un utilisateur est la représentation dans la grille d'une personne désirant soumettre des tâches de calcul. Il les soumet à un *peer* qu'il utilise, lequel décidera alors d'exécuter sa tâche sur ses *ressources* ou en faisant appel à un autre peer.

Pour l'exemple, l'utilisateur tourne sur l'hôte candi54, dispose du port 60407 et soumet sa tâche à un peer à l'écoute sur l'hôte candi01, port 60402. On exécute la

commande suivante du listing 8.17 (sous Linux. Pour une commande Windows, remplacer les : par des ; et les / par des \).

```
java -cp "classes/:jars/commons-math-1.1.jar:jars/Azureus2.5.0.4.jar"
    lbg.middleware.LaunchUser -H candi01 -P 60402 -h candi54 -p 60407
    -i mon_nom_dutilisateur -f mon_fichier_de_description
```

Listing 8.17 – Soumission d'une tâche : lancement de l'utilisateur

Au lancement du programme utilisateur, celui-ci contacte le peer pour lui soumettre sa description de Job. Si la soumission échoue parce que le peer n'est pas joignable ou que le Job était mal décrit, l'utilisateur en est prévenu.

Une fois le job correctement soumis, le programme attend les données de sortie des tâches composant le Job jusqu'à ce que l'ensemble de celle-ci ait été correctement retourné. À chaque complétion de tâche, un message est affiché (voir section 7) et les données de sortie et logs d'exécution sont placés dans le répertoire courant (fichiers d'extensions `.lbglog` et `.lbgout`). Il est possible de spécifier un autre répertoire via l'argument approprié.

Une fois tous les Jobs soumis correctement exécutés, le programme utilisateur termine son exécution.

Chapitre 9

Expériences

Afin d'estimer les gains apportés par les politiques décrites dans le chapitre 5, des expériences ont été réalisées.

9.1 Soumission automatique

Les lanceurs vus dans la section 4.6 conviennent pour des exécutions communes, mais ne sont pas adaptés aux exécutions automatisées qui se montrent utiles lors de tests de performances. À cette fin, une nouvelle classe a été écrite : `AutomaticTaskSubmitter`, elle aussi dans le package `lbg.middleware`. Cette classe est capable de soumettre des jobs à la grille selon plusieurs paramètres :

- Le nombre de tâches par job, la classe étant prévue pour lancer plusieurs jobs à intervalles réguliers ;
- Le nombre de données par tâche ;
- Le *SDR*, ou *Shared Data Ratio* [15, 38]. Ce nombre définit le ratio de données partagées entre les tâches d'un même job :

  $$SDR = \frac{Nombre\ de\ données\ uniques}{Nombre\ de\ données\ total}$$

  Dans la suite, nous le représenterons en pourcents plutôt que sous sa forme décimale.
- Le code de la tâche à exécuter et un serveur FTP servant les données nécessaires.

La génération des données de test fonctionne selon un procédé particulier. En effet, étant donné que la grille différencie les données par leur identifiant, il est possible de générer N GridDatas différentes sur base d'un seul fichier de données en générant les identifiants aléatoirement. On ne calcule ainsi qu'une seule fois le hash, et n'héberge

qu'une seule copie des données sur l'utilisateur : une économie de temps et d'espace disque.

Pour les tests, on désire modéliser le cas le plus désavantageux, celui qui maximise la distance entre les tâches utilisant la même donnée (le peer a la possiblité de les réordonner [38]). Pour ce faire, on va utiliser le principe suivant :

- On peut voir l'ensemble des données aléatoires comme un seul tableau. Soit A ce tableau.
- Les tâches contiennent toutes un tableau des données dont elles dépendent. Soit B la concaténation des tableaux de données des tâches : si T_1, T_2 dépendent respectivement de B_1, B_2 alors $B = \{B_1, B_2\}$
- Pour maximiser la distance entre des données identiques, on va construire B comme étant la concaténation d'un certain nombre de copies de A. De cette manière, deux données identiques sont séparées de $|A| - 1$ données différentes : $B = \{A, A, \ldots, \textit{partie de } A\}$

Construire B revient donc à distribuer des copies de A entre les tableaux de dépendances en données B_i des tâches T_i.

Mise à part la création des descripteurs de Jobs et de tâches, le soumetteur de tâches automatique se comporte comme le programme de soumission interactif (voir les sections 4.6 et 8.2.2), le nœud géré par la classe étant un GridUser. Rappelons que GridUser n'a pas la responsabilité de créer les descripteurs mais seulement de les soumettre à son peer.

Le comportement du programme de soumission interactif est d'envoyer les données au peer sous la forme de données *inline* (voir section 5.1.1). Néanmoins, celles-ci sont très coûteuses car intégralement conservées en mémoire. Pour cette raison, le soumetteur automatique se base sur un serveur FTP externe qui doit être déployé à part pour servir les données. Dans le cadre de ces tests, nous avons utilisé VSFTPD [40] car il était léger, rapide et facile à déployer.

9.2 Configuration de test

Les tests ont été réalisés sur les ordinateurs du réseau du service d'algorithmique du Département Montefiore de l'Université de Liège.

La grille de test est composée de 3 peers fournisseurs de 8 ressources chacun, pour un total de 24 ressources. Pour soumettre les tâches, on introduit un peer consommateur ainsi qu'un utilisateur s'y connectant. Enfin, le serveur du moteur de recherche est

démarré sur un $30^{ème}$ ordinateur. Les peers et leurs ressources disposent de processeurs Pentium IV 3.02 GHz et de 1024 Mio de RAM. L'utilisateur et le serveur de recherche disposent de Celerons 2.4 Ghz et de 512 Mio de RAM. Tous sont connectés via un réseau Fast Ethernet (100 Mbps) switché.

Les peers et ressources sont déployés comme présenté dans la section 8.1.2, en partageant les fichiers de configuration par NFS. Leurs caches sont par contre conservés dans des répertoires conservés sur les disques durs de chaque ordinateur (sinon, les transferts par NFS auraient des effets d'ordre supérieur aux protocoles de transferts choisis par la grille).

Les peers utilisent tous des politiques et algorithmes d'ordonnancement permettant d'optimiser les performances [38].

Pour réduire le temps de soumission d'une tâche, le serveur FTP de l'utilisateur (voir section 9.1) est hébergé sur la même machine que le peer consommateur.

Dans les expériences suivantes, l'utilisateur soumet comme Job l'application de hachage proposée dans la section 8.2.1. Le paramètre de la tâche (l'algorithme choisi) est SHA-1, et chaque tâche hache un unique fichier.

On mesure alors le temps d'exécution des Jobs qui, compte tenu du fait que les tâches hachent les fichiers en des temps négligeable (quelques secondes au plus), sont principalement influencés par les temps de téléchargement des données requises par les tâches.

9.3 Performances des protocoles selon le SDR

Cette expérience compare l'efficacité des protocoles FTP et BitTorrent en fonction du partages des données au sein d'un Job.

Pour ce faire, l'utilisateur soumet un Job de 100 tâches dépendant chacune d'un fichier de données de 256 Mio. L'expérience est répétée pour les deux protocoles avec des SDR de 1%, 25% et 75%. Les ressources ont toutes une taille de cache de 5 fichiers.

La figure 9.1 présente les résultats de l'expérience.

Un SDR de 1% implique que toutes les tâches du Job de 100 tâches soumis dépendent du même fichier de données. Lors de la première exécution d'une tâche sur une ressource, ce fichier doit être téléchargé. Dans le cas de BitTorrent, l'essaim est composé de l'ensemble des ressources et l'efficacité est maximale car la charge est reportée sur l'essaim et non sur le peer consommateur de ressources (et fournisseur de données).

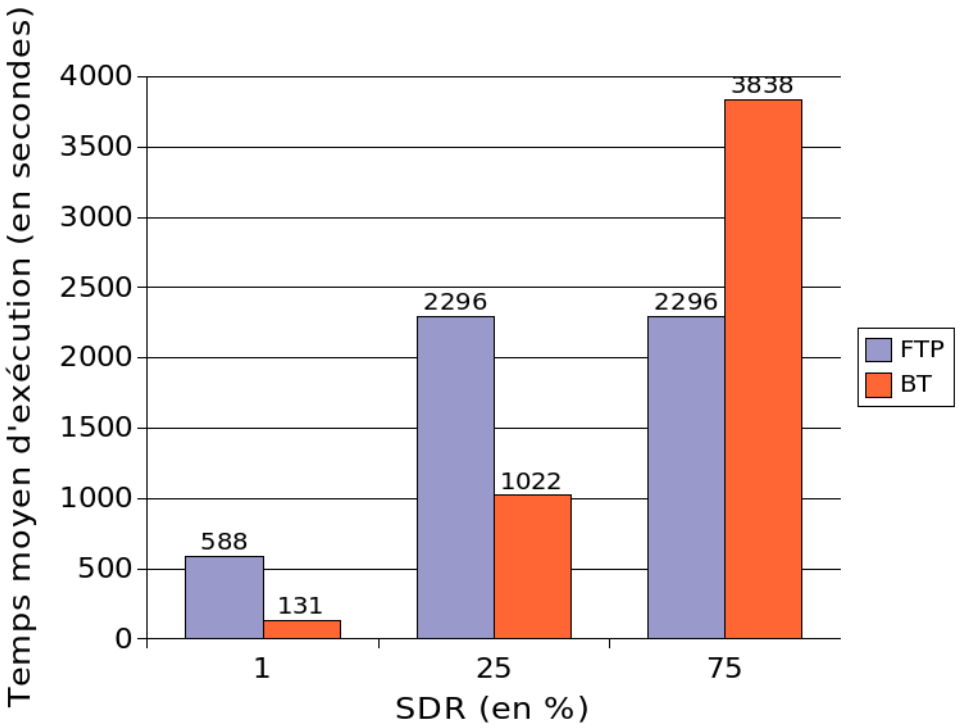


FIG. 9.1 – Performances des protocoles selon le SDR

Dans le cas de FTP, toutes les ressources doivent se partager la bande passante du peer consommateur de ressources et le temps de téléchargement est multiplié par 4 par rapport à BitTorrent. Lors des exécutions suivantes, toutes les ressources ont les données (en fait, l'unique fichier) en cache. Les données ne doivent donc pas être téléchargées à nouveau.

Quand le SDR atteint 25%, toutes les exécutions de tâches ou presque demandent de télécharger un fichier. En effet, étant donné qu'un peer fournisseur sélectionne la première ressource disponible pour exécuter une tâche, il n'y a qu'une chance sur 25 pour que les données dont elle dépend soient déjà en cache (un SDR de 25 pour 100 tâches d'une donnée implique l'existence de 25 données uniques). Si le cache ne permet plus d'éviter un nombre significatif de téléchargements, il contribue cependant à conserver des essaims BitTorrent de taille appréciable. En effet, une fois que les données sont téléchargées, elles restent à disposition des autres ressources qui en auraient éventuellement besoin par la suite. C'est pour cette raison que l'utilisation de BitTorrent résulte en des temps de téléchargement en moyenne 2 fois plus rapides que FTP.

FTP redevient compétitif quand le SDR augmente (dans ce test, à 75%). En effet, les données sont alors très peu partagées, ce qui implique d'une part qu'il sera nécessaire de les télécharger à chaque exécution, et d'autre part que les essaims BitTorrent seront à chaque fois de taille réduite. BitTorrent perd donc considérablement en performance.

9.4 Performances des protocoles selon la taille des données

Cette expérience compare l'efficacité des protocoles FTP et BitTorrent en fonction de la taille des données à transférer.

Pour ce faire, l'utilisateur soumet un Job de 100 tâches dépendant chacune d'un fichier de données et avec un SDR de 25%. L'expérience est répétée pour les deux protocoles avec des tailles de données de 1 Mio, 64 Mio et 256 Mio. Les ressources ont toutes une taille de cache de 5 fichiers.

La figure 9.2 présente les résultats de l'expérience. Les constatations concernant la nécessité de télécharger ou non les données sont similaires à celles de la section 9.3.

Pour des données de très petite taille (1 Mio), on observe les résultats attendus [15] : même si les données sont suffisamment partagées (voir section 9.3), l'overhead lié à BitTorrent (voir section 5.1.3) est trop important par rapport à leur taille. C'est pour cette raison que FTP permet de gagner quelques secondes.

Quand la taille des données augmente, BitTorrent se fait progressivement plus efficace. À 64 Mio, la différence entre FTP et BitTorrent reste ténue, mais BitTorrent commence à gagner en efficacité. Pour une taille de données atteignant 256 Mio, BitTorrent devient beaucoup plus efficace qu'FTP.

9.5 Utilité du cache

Cette expérience a pour but de présenter l'utilité du cache.

L'utilisateur soumet 4 fois le même Job de 25 tâches avec un SDR de 100%, c'est-à-dire que le partage de données est uniquement inter-jobs. Avant chaque soumission, il attend que les données de sortie du Job précédent lui soit revenues. Il sait donc que les ressources ayant exécuté le premier Job sont libres, et que ses données sont dans leur cache.

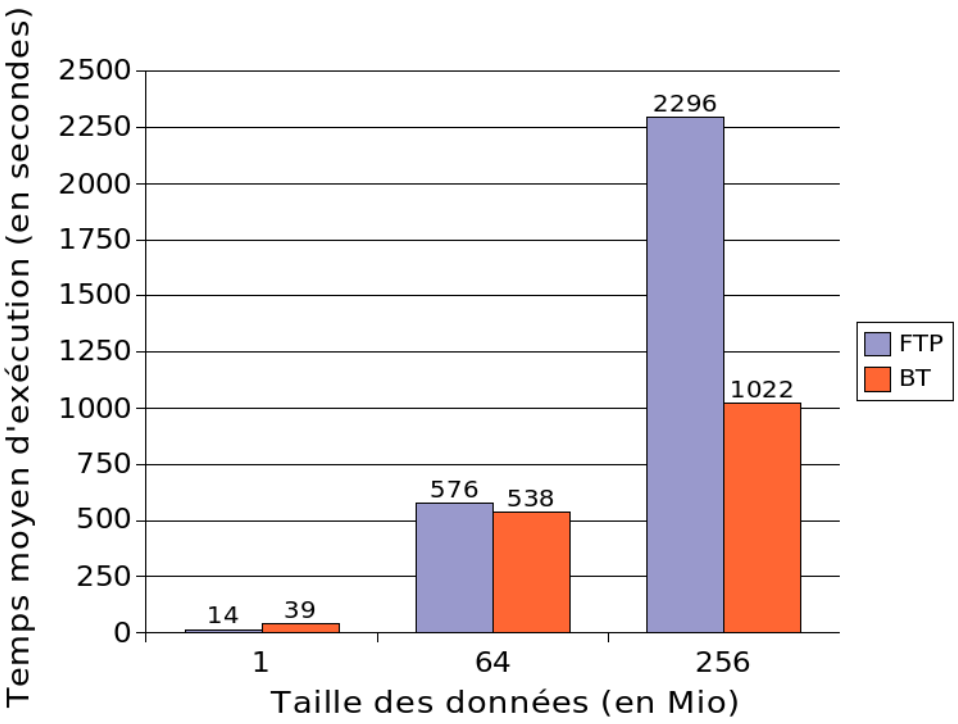


FIG. 9.2 – Performances des protocoles selon la taille des données

L'expérience a été réalisée en utilisant BitTorrent et FTP, avec cache (tailles de 1 et 5 fichiers) et sans cache (un cache vide est un cache effacé immédiatement après chaque complétion de tâche).

La figure 9.3 présente les résultats de l'expérience. Comme attendu, le cache réduit drastiquement le temps moyen d'exécution d'un Job (il est divisé par deux) car le peer consommateur envoie de préférence ses tâches aux fournisseurs possédant déjà les données dont elles dépendent, et que les fournisseurs exécutent de préférence les tâches sur les ressources ayant ces données en cache.

Les performances de BitTorrent peuvent paraître surprenantes. On peut les justifier par le raisonnement suivant. Lors de la soumission du premier Job, celui-ci a un SDR de 100%, ce qui implique que chaque tâche a ses données et que celles-ci ne sont pas partagées. Or, BitTorrent n'est pas du tout prévu pour des essaims d'un seul peer [25] et offre alors des performances inférieures à FTP [15]. Lors des soumissions suivantes, on distingue deux situations quand une tâche arrive sur une ressource :

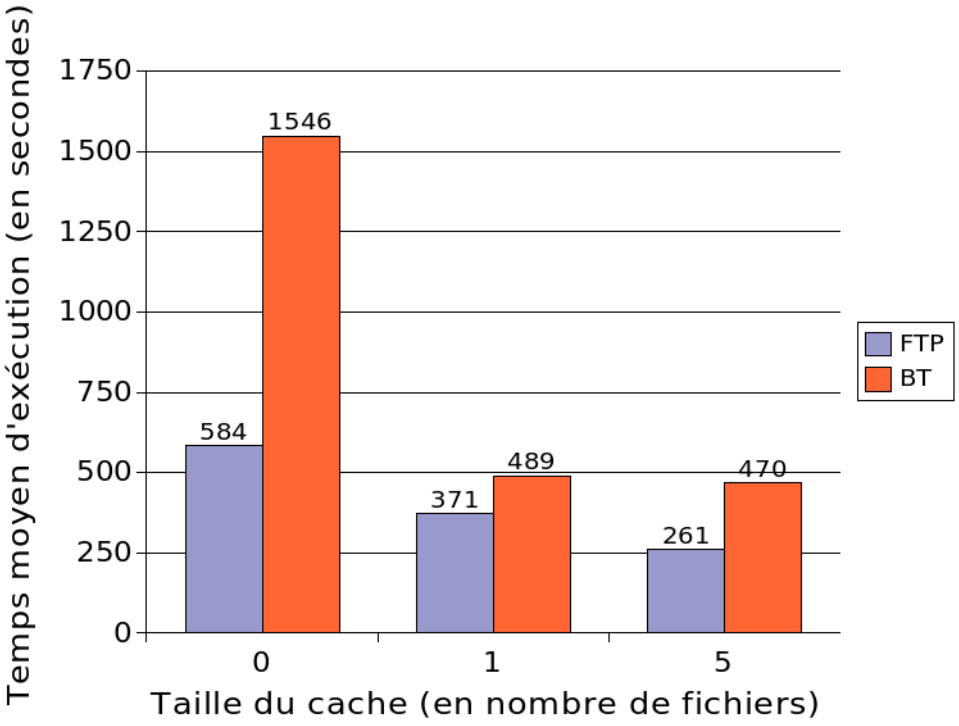


FIG. 9.3 – Utilité du cache

- Le cache contient les données : celles-ci sont récupérées du cache et ne sont donc pas téléchargées. On n'observe aucune différence entre BitTorrent et FTP (le temps de transfert est nul). La présence d'un cache limite donc l'impact de ces conditions extrêmes sur BitTorrent.
- Le cache ne contient pas les données (il est de taille nulle ou ne dispose pas des données requises) : elles doivent être téléchargées. Si les caches sont de taille nulle, la même situation que pour la première soumission se reproduit (un seul peer dans l'essaim) et FTP renforce son avantage. Sinon, l'essaim contient au moins un peer et BitTorrent gagne un peu en performance. Cette situation est similaire à celle rencontrée dans l'expérience de la section 9.4 dans le cas du SDR de 75%.

Chapitre 10

Développements futurs

Si le middleware proposé est déjà fonctionnel (voir la section 4.7 et le chapitre 8), il reste toutefois certains points qui sortent du cadre de ce travail mais gagneraient à être améliorés dans un développement ultérieur.

10.1 Persistance des données

Les algorithmes d'ordonnancement et de négociation utilisés dans le middleware s'accommodent assez bien du caractère changeant d'une grille peer-to-peer. Cependant, les nœuds sont peu conscients de leur propre instabilité. Si un des avantages d'une grille peer-to-peer est qu'on peut l'installer sur un réseau de machines de bureau, il faut aussi tenir compte du fait que ces mêmes machines n'offrent en général pas la même qualité de service que des serveurs dédiés de haute qualité. Il est donc concevable que ces machines soient déconnectées du réseau pour un laps de temps plus ou moins court, ou qu'elles cessent de fonctionner (panne, erreur de manipulation).

Dans ce cadre, il serait utile que les peers et utilisateurs conservent sur disque leurs registres et historiques, au moins de façon périodique. Ainsi, à son lancement, un peer pourrait recommencer les négociations avec ses semblables en sachant lesquels sont dignes de confiance et lesquels semblent être des *freeriders*. Les utilisateurs, quant à eux, pourraient reconnaître les jobs qui reviennent d'une session précédente (pour l'instant, ils sont acceptés mais les jobs non reconnus provoquent l'affichage d'un message d'avertissement).

Une défaillance technique n'est pas la seule raison pour laquelle un utilisateur pourrait être temporairement déconnecté de la grille. Il pourrait décider d'éteindre son ter-

minal ou de le redémarrer pour terminer une installation. Si à ce moment précis, une de ses tâches est complétée, le peer responsable de l'utilisateur jettera les résultats car il n'est pas capable de contacter ce dernier, lequel ne sera donc jamais averti (même à sa reconnexion) que sa tâche a réussi, et n'en aura jamais les résultats. Une amélioration intéressante serait donc de réessayer de contacter plusieurs fois l'utilisateur avant d'« oublier » ses tâches.

10.2 Sécurité

Le middleware a été conçu et testé dans un environnement de laboratoire. L'accès aux programmes de lancement des nœuds étant restreint, les questions de sécurité n'ont été abordées qu'en surface. Pourtant, avant de placer ce programme dans un environnement sans aucun contrôle, il est important de protéger les nœuds contre les attaques élémentaires.

10.2.1 Cryptographie

Un problème dans l'architecture actuelle est qu'il est aisé d'usurper l'identité des peers car leur reconnaissance n'est basée que sur des identifiants. Si un attaquant est capable de rendre indisponible un peer renommé assez longtemps pour que les moteurs de recherche suppriment sa handle de leurs registres, par exemple par une attaque de type déni de service, il pourra prendre sa place en envoyant une handle contenant l'ID du peer renommé et l'adresse de l'attaquant. L'attaquant profite alors de la réputation du peer légitime et peut exécuter ses tâches en son nom.

Ce problème est renforcé par le fait qu'il suffit de posséder une handle pour pouvoir effectuer un appel. Créer une telle handle est aisé, et un attaquant pourrait donc mener l'attaque suivante :

- L'attaquant négocie avec un peer B en se faisant passer pour un peer A et lui envoie une tâche ;
- B exécute la tâche et crédite A ;
- B renvoie les résultats à A qui ne les reconnaît pas. B est capable de contacter le vrai A car l'attaquant n'a pas remplacé la handle de A par la sienne et que A et B sont sur le même moteur de recherche ;
- A est alors considéré comme ayant consommé les ressources de B **alors qu'il n'a jamais fait de demande**.

Une solution à ce problème serait d'utiliser un systèmes de certificats, par exemple en sécurisant le protocole au niveau de la couche transport avec TLS [47]. L'avantage

de cette solution est qu'on pourrait en plus profiter des services de chiffrement que le protocole offre.

10.2.2 Protection des ressources

Le GridDataManager ainsi que les classes de lancement de tâches (LiveTaskHandler et LiveTaskRunner) protègent les ressources de certaines attaques mais celles-ci restent vulnérables à certaines tâches malveillantes.

Par exemple, un utilisateur pourrait soumettre une tâche dépendant de beaucoup de données de faible taille. La tâche se contenterait de faire quelques opérations rapides (voire aucune opération) et se terminerait, consommant peu de ressources au niveau du peer fournisseur (le peer consommateur ne serait donc que faiblement débiteur). Par contre, la multitude de données de faible taille aura éjecté du cache des données qui pourraient être de taille plus importante et d'utilisation fréquente, forçant la ressource à les retélécharger au prix d'une perte de temps et de bande passante.

Une tâche pourrait aussi saturer une ressource et/ou son peer en retournant des données d'une taille trop importante. Conservées sous la forme d'un tableau d'octets (voir la section 4.7), elles pourraient consommer toute la mémoire du nœud et le rendre indisponible. Pour résoudre ce problème, on pourrait fournir à la tâche un `java.io.FileOutputStream` dans lequel elle placerait les données de sortie. Conservées sur disque, elle ne risqueraient plus de saturer la mémoire des nœuds. Après l'exécution de la tâche, les données seraient partagées grâce à un protocole non distribué, par exemple FTP. Les données de sortie étant a priori uniques, il n'est pas possible de partager la charge avec un protocole peer-to-peer. Cette solution n'a pas été implémentée car elle implique de gérer un nouveau cache de données au niveau de la ressource. Une entrée en cache ne pourrait alors être effacée que lorsque l'utilisateur l'a téléchargée, une information qu'il n'est pas aisé d'obtenir de manière fiable.

Les tâches sont supposées s'exécuter sans communication avec l'extérieur (qu'il s'agisse d'un réseau ou du système de fichiers). Cette condition n'est toutefois pas encore vérifiée et les tâches s'exécuteront donc avec les droits de l'utilisateur qui a lancé le middleware ! Cela implique que dans le meilleur des cas (en supposant qu'un utilisateur restreint ait été créé pour le service comme c'est souvent le cas dans les systèmes Unix), une tâche aura accès en écriture aux fichiers du cache, y compris aux éventuels fichiers des tâches précédentes et au répertoire de configuration d'Azureus (voir section 5.1.3). Sun fournit une réponse à ce problème grâce à l'utilisation de politiques de sécurité [48], mais bien les configurer n'est pas trivial (une tâche ne devrait avoir un accès en lecture qu'à ses propres fichiers d'entrée).

Enfin, il est aussi possible de certifier formellement les programmes à exécuter [49]. Cette approche a déjà été testée dans le cas de Java, mais elle est très lourde pour le programmeur et ne correspond donc pas aux objectifs de ce travail.

Le choix du middleware *state of the art* OurGrid [11] se porte vers la virtualisation à l'aide de Xen [10] mais la démarche est coûteuse (cela demande de reconfigurer le système d'exploitation) et bien que prévue depuis plusieurs années, elle commence à peine à être déployée en production.

10.2.3 Protection des tâches

Les tâches peuvent aussi être victimes de ressources malveillantes. Celles-ci pourraient essayer de voler le code source de la tâche (après décompilation) ou d'analyser ses données à des fins d'espionnage industriel. Cette attaque a un sens dans un environnement où l'accès à la grille n'est pas contrôlé (on peut envisager une grille où l'accès est contrôlé mais où les nœuds sont susceptibles de se connecter ou déconnecter à tout moment).

Par exemple, des entreprises pourraient tenter de profiter des progrès de leurs concurrents.

Pour contrer cela, il faut faire appel à des mécanismes de *security of execution* [50, 51].

Pour protéger le code source et les données, on peut envisager plusieurs solutions :

- Faire appel à du matériel certifié et sécurisé. Dans le cadre d'une grille, on ne peut envisager l'installation fiable d'un tel matériel que si la grille est soumise à un contrôle. Dans ce cas, protéger les tâches est peut-être inutile ;
- Modifier le code pour le rendre plus difficile à lire (*obfuscation*). Le procédé est empirique et ne se base pas sur des résultats mesurables. De plus, le code est seulement difficile à lire et une personne entraînée pourra tout de même comprendre le fonctionnement des algorithmes ;
- Les instructions du code compilé peuvent être transformées en un ensemble d'instructions équivalentes qui seront correctement traitées par le système et fourniront le résultat attendu, mais pour lesquelles il sera impossible (ou, du moins, très difficile) de retrouver l'algorithme de départ [50].

En plus de la confidentialité, un middleware de grille peut aussi fournir un service d'intégrité, c'est-à-dire qu'il doit s'assurer que les résultats de la tâche n'ont pas été faussés. Une solution possible est d'utiliser de la réplication [52], c'est-à-dire à exécuter plusieurs fois la même tâche sur des ressources différentes et à comparer ensuite les

résultats des exécutions. Si les résultats diffèrent, alors au moins une ressource a exécuté la tâche de manière incorrecte et on peut en avertir l'utilisateur.

Moyennant le support des tâches, on peut pousser plus loin cette approche en effectuant les vérifications à certains points de l'exécution (*checkpoints*) plutôt qu'à la fin [53]. En cas d'erreur, on est ainsi en mesure de relance l'exécution depuis le dernier *checkpoint* validé plutôt que depuis le début de la tâche.

Notons que cette approche est probabiliste : elle ne garantit pas que les tâches soient correctement exécutées mais permet d'approcher la certitude (à chaque exécution d'une tâche, on diminue les chances qu'une corruption des résultats reste indétectée).

10.2.4 Protection du réseau

Les nœuds individuels ne sont pas les seuls éléments de la grille qui peuvent être attaqués. Le fonctionnement de la grille dans son ensemble doit lui aussi être considéré.

Par exemple, il est possible pour un utilisateur de soumettre des tâches dépendant d'une grande quantité de données de petite taille. Lors de l'exécution de ces tâches sur les ressources, les petites données pourraient remplacer d'autres données de tailles plus importantes dans le cache. Si une telle attaque est menée sur plusieurs peers, c'est l'ensemble de la grille qui est pénalisé :

- Les peers fournisseurs doivent passer plus de temps à retélécharger des données qu'ils auraient sinon en cache quand une tâche légitime est exécutée ;
- Si les données légitimes ont été transférées grâce à BitTorrent, leur disparition réduit l'efficacité de l'essaim. Cela ralentit le téléchargement sur de nouvelles ressources et apporte une charge supplémentaire au peer consommateur ;
- Si les données légitimes ont été transférées via FTP, elles devront être retéléchargées depuis le peer consommateur. Cela réduit sa bande passante disponible et lui apporte une charge encore plus importante que dans le cas BitTorrent.

D'autre part, il a été dit dans la section 5.1 que les identifiants de données avaient la structure `peer.user.filename`. Or, ces identifiants sont générés par l'utilisateur et si un client mal intentionné A modifiait son logiciel de soumission de Jobs pour fournir des identifiants non conformes, il pourrait réduire l'espace de noms ou perturber le téléchargement de données d'un utilisateur légitime B : en prédisant l'identifiant des données légitimes et en plaçant les siennes en cache sur une partie des ressources de la grille avant B, A empêche le peer de B d'accéder à des ressources qui seraient peut-être puissantes. Générer les identifiants à la soumission au niveau du peer est une solution, mais il faudrait tenir compte des optimisations effectuées dans la section 4.1 (l'optimisation du transfert des bytecodes est aussi appliquée pour les GridDatas). Une

autre solution est de vérifier que les identifiants sont conformes avant de télécharger les données.

10.3 Résistance aux facteurs d'échelle

En théorie (cela n'a pas été testé en situation réelle), le middleware résisterait à des grilles de centaines de peers. Certains points pourraient toutefois poser problème.

10.3.1 Gestion des sockets

La politique actuelle pour les appels réseau (voir section 4.2) est de créer une nouvelle connexion TCP pour chaque appel. Tant que les temps entre deux appels impliquant les mêmes nœuds est relativement important, ce comportement est raisonnable car on ne maintient une connexion ouverte que si on en a effectivement besoin. Toutefois, si les appels devenaient plus fréquents, certaines connexions seraient fermées pour être ouvertes à nouveau presque directement après, pénalisant d'autant les performances des appels.

Pour résoudre ce problème, il serait nécessaire de modifier les classes du package `lbg.protocol.rpc` afin de conserver les connexions ouvertes si besoin est. Cette discussion est similaire à celle qui a conduit à l'introduction du *HTTP Pipelining* dans HTTP 1.1 [54].

Une première solution serait de créer un pool de sockets qui aurait la responsabilité de créer les connexions et les conserverait un certain temps avant de les former. Pour effectuer un appel, une instance de RemoteCall ferait appel au SocketPool qui lui renverrait une connexion existante ou en créerait une nouvelle. Si la modification au niveau du client est minime, cela implique toutefois que le serveur conserve des Threads à l'écoute sur toutes les connexions récentes au cas où un nouvel appel arriverait. Il faudrait donc procéder à des tests pour vérifier que cette approche présente bien un avantage par rapport à la politique actuelle.

Une autre solution serait d'utiliser la nouvelle API d'entrées/sorties `java.nio` [55]. Cette API fut introduite dans Java 1.4 et propose des appels non bloquants prévus pour l'implémentation de serveurs performants. Elle est donc plus adaptée au cas présent, mais demanderait sans doute de modifier des classes extérieures au package `lbg.protocol.rpc` et de définir un nouveau protocole de communication en mode texte car elle n'est pas adaptée au transfert d'objets par sérialisation, la méthode choisie pour l'instant.

10.3.2 Découverte de peers

Le système de découverte de peers décrit dans le chapitre 6 est fonctionnel mais basique et supporterait sans doute mal la gestion de plusieurs centaines de peers.

Une amélioration possible serait d'utiliser une hiérarchie similaire à DNS [56, 57] où des serveurs répliqués se transfèreraient l'information à la demande. Un système similaire est utilisé par GRIP [41]. NodeWiz utilise aussi une structure d'arbre étudiée pour fournir un service constant même si les peers composant l'arbre viennent à se déconnecter [58].

Un autre système intéressant est Kademlia [59], une table de hachage distribuée prévue pour des environnements non contrôlés. Kademlia est utilisé dans des réseaux d'échange de fichiers comme Overnet (maintenant disparu) ou le réseau Kad utilisé notamment par eMule [5], et sert de base à l'implémentation de tracking décentralisé d'Azureus (voir section 5.1.3).

La découverte de ressources est un problème récurrent dans les logiciels réseau. Plusieurs solutions existent, avec leurs points forts et leurs points faibles, mais elles souffrent toutes du même problème de *bootstrap* (à des degrés divers) : on ne connaît pour l'instant pas de méthode pour se connecter à un réseau sans connaître au moins un de ses membres [60].

Indépendamment du modèle de découverte choisi, il serait intéressant qu'un peer puisse faire partie de plusieurs grilles à la fois. Tant qu'à présent, comme présenté dans la figure 6.1, une grille possède un seul moteur de recherche (qu'il soit distribué ou non) et un peer ne peut être référencé que par un moteur (donc dans une seule grille) à la fois. En modifiant le peer pour lui permettre de contacter plusieurs services de découverte de peers, on augmenterait sa productivité car il aurait accès à plus de ressources partagées. La figure 10.1 présente une telle possibilité (les différentes grilles sont encadrées, leur intersection n'étant plus vide).

10.4 Accessibilité

Enfin, le middleware pourrait offrir plus de services aux utilisateurs (programmeurs et administrateurs). Ces points ont moins d'impact que les précédents mais pourraient faire la différence dans une mise en production.

L'API pourrait offrir des utilitaires pour des actions communes à l'exécution d'une tâche. Il pourrait aussi être utile de bénéficier des logs d'exécution d'une tâche non seulement quand elle se passe correctement (voir section 4.4.1), mais aussi quand un

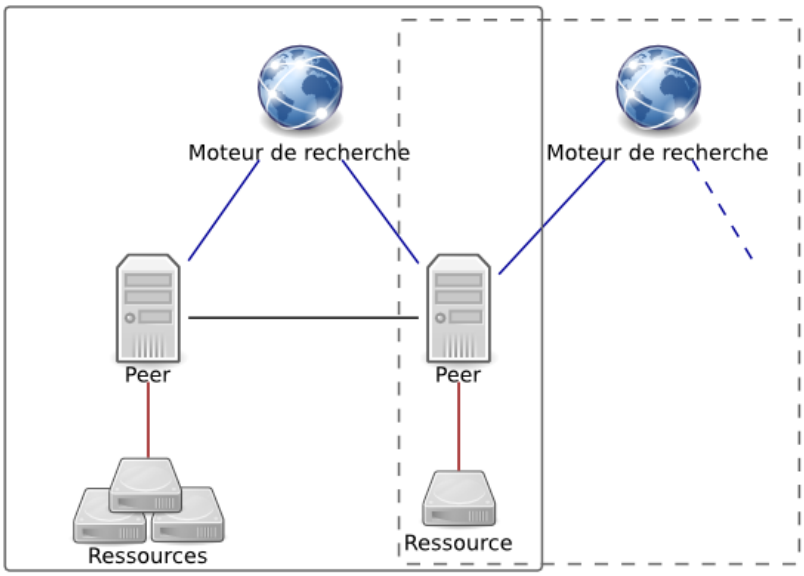


FIG. 10.1 – Moteurs de recherche à intersection non vide

problème survient à l'exécution et que celui-ci n'est pas dû au middleware. On considère que le middleware n'est pas en cause si les opérations préliminaires (téléchargement des données, création du processus LiveTaskRunner, ...) sont effectuées sans erreur, mais que l'un des appels aux méthodes spécifiées par GridApplication échoue. Dans ce cas, il y a de fortes chances que l'exécution échouerait aussi sur une autre ressource, et il serait préférable d'en prévenir l'utilisateur plutôt que de tenter de réexécuter la tâche.

La soumission des tâches de calcul pourrait aussi être simplifiée par la mise à disposition d'une interface Web basée sur un système J2EE. Initialement prévue dans les objectifs, cette fonctionnalité n'a finalement pas été implémentée car elle soulève des problèmes pour lesquels certaines sections du middleware sont encore à implémenter. Par exemple comment remettre des résultats à un utilisateur qui ne s'est connecté que pour soumettre sa tâche et n'est a priori pas joignable dans l'immédiat ?

Enfin, comme cela a été présenté dans la section 8.1, il est nécessaire de paramétrer les noms d'hôte manuellement, en configurant le NAT correctement si nécessaire. Ces concepts ne sont pas toujours bien compris par les utilisateurs et restent peu pratiques à appliquer par les administrateurs. Le middleware pourrait proposer un service de reconnaissance automatique d'adresse et de *NAT Traversal* comme le fait Azureus, ce qui simplifierait la mise en place des nœuds.

Chapitre 11

Conclusion

Le but de ce travail est de fournir une implémentation « réelle » (dite *Live*) à un middleware de Grid Peer-to-Peer déjà utilisé dans un simulateur, lequel a continué à évoluer en parallèle à ce travail de fin d'études pour offrir plus de fonctionnalités, de performances et répondre à certaines contraintes spécifiques aux communications réseau.

Le développement s'est principalement articulé en 4 points : appels de méthodes distantes, exécution effective des tâches, gestion des transferts de données et découverte de peers.

Pour les appels distants (ou *RPC*), un système simple a été préféré aux alternatives comme RMI, plus puissantes mais aussi plus lourdes à déployer. Notre implémentation est relativement flexible puisque l'ajout ultérieur des appels au moteur de recherche s'est fait rapidement et sans nécessiter d'importantes modifications.

L'exécution des tâches était en simulation limitée à une modélisation du temps d'exécution de celles-ci. Ce travail a apporté un canevas d'exécution permettant à des programmeurs d'applications *Grid* de soumettre des tâches réelles en se conformant à une interface simple. Le middleware est capable de charger dynamiquement le code des tâches et de les exécuter dans un environnement confiné pour limiter l'impact qu'une tâche malveillante aurait sur la ressource l'exécutant.

Les tâches exécutées peuvent être des consommatrices de données. L'implémentation d'un système de gestion de données performant représente une partie importante de ce travail. Pour le transfert des données, le service se base sur deux protocoles répandus : FTP et BitTorrent (en utilisant Azureus, dont l'intégration au middleware a demandé un temps considérable). L'efficacité de ces protocoles a été comparée et dépend de la taille et de la répartition des données. Il est donc possible d'utiliser

conjointement les deux protocoles et de repartager dans un protocole des données qui étaient déjà disponibles dans un autre. Des développements futurs pourraient permettre la mise à disposition de plus de protocoles en ne modifiant qu'un nombre restreint de classes. Enfin, pour augmenter les performances, un système de cache est proposé. Il est possible d'en limiter la taille si nécessaire.

Pour permettre aux peers de communiquer sans se connaître à leur lancement, un service de découverte de peers a été implémenté. Il s'agit d'un moteur de recherche centralisé auprès duquel les peers s'enregistrent avant de l'interroger. Ce service représente pour l'instant un *Single Point of Failure* sur la grille mais il pourrait être remplacé par un système répliqué (ou, mieux, distribué) en ne modifiant que le package correspondant. Offrir un service plus fiable sort cependant du cadre de ce travail et l'implémentation actuelle reste parfaitement utilisable.

Étant exclue des objectifs imposés de ce travail, la sécurité n'a pas été considérée en profondeur. Des mécanismes protègent de certaines attaques basiques mais pas de toutes, et les attaques plus recherchées ne sont pas contrées. Des développements futurs pourraient améliorer substantiellement cet aspect du middleware, mais cela requerrait de modifier aussi les classes de base du middleware.

Enfin, d'autres développements pourraient rendre la soumission de tâches de calcul encore plus aisée en limitant les connaissances que les utilisateurs doivent avoir de la grille pour s'en servir et en leur apportant des services qu'ils sont habitués à recevoir d'une exécution locale, comme par exemple la possibilité d'analyser précisément les raisons de l'échec d'une exécution ou d'effectuer le suivi de l'exécution d'une tâche sur la grille.

Annexe : Code source

Le middleware LBG comporte 236 classes Java (sans compter les classes privées) pour un total de $\sim$ 40000 lignes de code.

Dans le cadre de ce travail, j'ai codé ou participé au code de 53 classes Java pour un total de $\sim$ 10000 lignes de code.

L'entièreté du code source est disponible à
`http://www.montefiore.ulg.ac.be/services/ail/tfe/grid.html`.

La documentation de ces classes a été effectuée en Javadoc (compilée en visibilité privée dans le répertoire `doc/api/`) et est aussi lisible par Doxygen [46] (compilée en visibilité privée dans le répertoire `doc/api-doxygen/`).

Bibliographie

[1] Ian Foster. What is the Grid ? - A Three Point Checklist. *GRIDtoday*, Juillet 2002.

[2] Ian Foster, Carl Kesselman, et Steven Tuecke. The Anatomy of the Grid : Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3), Août 2001.

[3] Cyril Briquet et Pierre-Arnoul de Marneffe. What is the Grid ? Tentative Definitions Beyond Resource Coordination. Technical report, Department of EE and CS, University of Liège, Mai 2006.

[4] Sun Grid Engine ; projet sur `http://gridengine.sunsource.net/`, page commerciale sur `http://www.sun.com/software/gridware/`.

[5] The eMule Project ; disponible sur `http://www.ourgrid.org/`.

[6] Malcolm Atkinson, David DeRoure, Alistair Dunlop, Geoffrey Fox, Peter Henderson, Tony Hey, Norman Paton, Steven Newhouse, Savas Parastatidis, Anne Trefethen, Paul Watson, et Jim Webber. Web Service Grids : an evolutionary approach : Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4), 2005.

[7] Ian Foster et Steven Tuecke. Describing the elephant : The Different Faces of IT as Service. *Queue*, 3(6), Juillet 2005.

[8] SETI@home ; disponible sur `http://setiathome.ssl.berkeley.edu/`.

[9] Folding@Home ; disponible sur `http://folding.stanford.edu/`.

[10] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro B. Costa, et Alisson Andrade. Labs of the World, Unite ! ! !

[11] OurGrid, a free-to-join peer-to-peer grid ; disponible sur `http://www.ourgrid.org/`.

[12] Miranda Mowbray. OurGrid : a Web-based Community Grid, Février 2006.

[13] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro, et Paul Roisenberg. OurGrid : An approach to easily assemble grids with equitable ressource sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, Juin 2003.

[14] Robson Santos, Alisson Andrade, Walfredo Cirne, Francisco Brasileiro, et Nazareno Andrade. Accurate Autonomous Accounting in Peer-to-Peer Grids. In *MGC '05 : Proceedings of the 3rd international workshop on Middleware for grid computing*. ACM Press, 2005.

[15] Baohua Wei, Gilles Fedak, et Franck Cappello. Scheduling Independent Tasks Sharing Large Data Distributed with BitTorrent. In *IEEE/ACM Grid'2005 workshop*, Seattle, USA, Novembre 2005.

[16] Remote Method Invocation, voir `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

[17] Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated ? Voir `http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html`, 1999.

[18] Java Language Specification, disponible sur `http://java.sun.com/docs/books/jls/`.

[19] GNU wget, disponible sur `http://www.gnu.org/software/wget/`.

[20] B. Callaghan, B. Pawlowski, et P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), Juin 1995.

[21] Why Developers Should Not Write Programs That Call 'sun' Packages, voir `http://java.sun.com/products/jdk/faq/faq-sun-packages.html`, 1996.

[22] edtFTPj, disponible sur `http://www.enterprisedt.com/downloads/ftp.html`.

[23] Jakarta Commons Net, disponible sur `http://jakarta.apache.org/commons/net/`.

[24] J. Postel et J. Reynolds. File Transfer Protocol. RFC 959 (Standard), Octobre 1985. Updated by RFCs 2228, 2640, 2773.

[25] Bram Cohen. Incentives Build Robustness in BitTorrent, Mai 2003.

[26] BitTorrent, disponible sur `http://www.bittorrent.com/`.

[27] A. Legout, N. Liogkas, E. Kohler, et L. Zhang. Clustering and Sharing Incentives in BitTorrent Systems. In *ACM Symetrics*. INRIA, Juin 2007.

[28] jTorrent ; disponible sur `http://sourceforge.net/projects/jtorrent`.

[29] The Hunting of the Snark Project - BitTorrent Application Suite ; disponible sur `http://www.klomp.org/snark/`.

[30] GCJ, the GNU Compiler for Java, disponible sur `http://gcc.gnu.org/java/`.

[31] Azureus : Java BitTorrent Client ; disponible sur `http://azureus.sourceforge.net/`.

[32] Baohua Wei, Gilles Fedak, et Franck Cappello. A Case for Efficient Execution of Data-Intense Applications with BitTorrent on Computational Desktop Grid.

Technical report, Laboratoire de Recherche en Informatique / INRIA Futurs, Août 2005.

[33] M. Allman et S. Ostermann. FTP Security Considerations. RFC 2577 (Informational), Mai 1999.

[34] GridFTP, part of the Globus Toolkit, disponible sur `http://www.globus.org/toolkit/docs/4.0/data/gridftp/`.

[35] William Allcock, John Bresnahan, Rajkumar Kettimuthu, et Michael Link. The Globus Striped GridFTP Framework and Server. In *SC '05 : Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Xiaoyun Wang, Yiqun Lisa Yin, et Hongbo Yu. Finding Collisions in the Full SHA-1. In *CRYPTO*, 2005.

[37] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, et Aliandro Lima. Exploiting Replication and Data Reuse to Efficiently Schedule Data-intensive Applications on Grids. In *Proc. Workshop Job Scheduling Strategies for Parallel Processing*, New York, NY, USA, 2004.

[38] Cyril Briquet, Xavier Dalem, Sébastien Jodogne, et Pierre-Arnoul de Marneffe. Scheduling Data-Intensive Bags of Tasks in P2P Grids with BitTorrent-enabled Data Distribution. In *Proc. UPGRADE-CN'07, HPDC Workshops*, Monterey Bay, CA, USA, June 2007.

[39] Apache FTP Server, disponible sur `http://incubator.apache.org/ftpserver/`.

[40] VSFTPD, disponible sur `http://vsftpd.beasts.org/`.

[41] Karl Czajkowski, Steven Fitzgerald, Ian Foster, et Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In IEEE Press, editor, *Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, Août 2001.

[42] Log4j project, disponible sur `http://logging.apache.org/log4j/`.

[43] Adobe flex, voir `http://www.adobe.com/products/flex/`.

[44] Jakarta Commons, disponible sur `http://jakarta.apache.org/commons/`.

[45] Apache Ant, disponible sur `http://ant.apache.org/`.

[46] Doxygen, Source code documentation generator tool, disponible sur `http://www.doxygen.org/`.

[47] T. Dierks et E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Avril 2006. Updated by RFCs 4366, 4680, 4681.

[48] Default Policy Implementation and Policy File Syntax, voir `http://java.sun.com/j2se/1.5.0/docs/guide/security/PolicyFiles.html`, Septembre 2003.

[49] Proof-Carrying Code, voir `http://raw.cs.berkeley.edu/pcc.html`.

[50] Sergio Loureiro. *Mobile Code Protection.* PhD thesis, Sophia Antipolis, France, Janvier 2001.

[51] Sergio Loureiro et Refik Molva. Privacy for Mobile Code. In *Distributed Object Security Workshop - OOPSLA'99*, pages 37–42, Novembre 1999.

[52] Axel Krings, Jean-Louis Roch, Samir Jafar, , et Sébastien Varrette. A Probabilistic Approach for Task and Result Certification of Large-scale Distributed Applications in Hostile Environments. In *European Grid Conference EGC'2005*, Amsterdam, The Netherlands, Février 2005.

[53] Filipe Araujo, Patricio Domingues, Derrick Kondo, et Luis Moura Silva. Validating Desktop Grid Results By Comparing Intermediate Checkpoints. Technical report, University of Coimbra, Portugal ; Polytechnic Institute of Leiria, Portugal ; INRIA Futurs, France, Octobre 2006.

[54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, et T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Juin 1999. Updated by RFC 2817.

[55] Java New I/O APIs, voir `http://java.sun.com/j2se/1.4.2/docs/guide/nio/`.

[56] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), Novembre 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.

[57] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Novembre 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 2137, 2845, 3425, 3658, 4035, 4033.

[58] Francisco Brasileiro, Lauro Beltrao Costa, Alisson Andrade, Walfredo Cirne, Sujoy Basu, et Sujata Banerjee. A large scale fault-tolerant grid information service. In *MCG '06 : Proceedings of the 4th international workshop on Middleware for grid computing*, New York, NY, USA, 2006. ACM Press.

[59] P. Maymounkov et D. Mazieres. Kademlia : A Peer-to-peer Information System Based on the XOR Metric, 2002.

[60] Sylvain Martin et Guy Leduc. Ephemeral State Assisted Discovery of Peer-to-peer Networks. In *Proc. IEEE ACNM Workshop*, Munich, Germany, Mai 2007.

Les icônes utilisées pour les représentations des peers, ressources et utilisateurs et moteurs de recherche font partie de la bibliothèque Tango, sous licence Creative Commons Attribution Share-Alike : `http://tango.freedesktop.org/`