Gestion des bases de données biologiques sur grilles de calculs

Gaël Le Mahec

To cite this version:

Gaël Le Mahec. Gestion des bases de données biologiques sur grilles de calculs. Modélisation et simulation. Université Blaise Pascal - Clermont-Ferrand II, 2008. Français. NNT : . tel-00462306

HAL Id: tel-00462306
https://theses.hal.science/tel-00462306

Submitted on 9 Mar 2010

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

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

N° d'ordre D.U : 1891
EDSPIC : 424

PCCF T 0805

Laboratoire de Physique Corpusculaire

THÈSE

présentée à

L'UNIVERSITÉ BLAISE PASCAL, CLERMONT-FERRAND

pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITÉ BLAISE PASCAL, CLERMONT-FERRAND

Spécialité INFORMATIQUE

Gestion des bases de données biologiques sur grilles de calcul

par

Gaël Le Mahec

Équipe d'accueil : PCSV, IN2P3, Clermont-Ferrand
École doctorale des Sciences Pour l'Ingénieur, Université Blaise Pascal

Soutenue le 3 décembre 2008 devant la Commission d'Examen

COMPOSITION DU JURY

Monsieur Frédéric Desprez, Directeur de recherche, INRIA	Directeur de thèse
Monsieur Vincent Breton, Directeur de recherche, IN2P3	Directeur de thèse
Monsieur Thierry Priol, Directeur de recherche, INRIA	Rapporteur
Monsieur Pierre Sens, Professeur, Université de Paris 6	Rapporteur
Monsieur Lionel Brunie, Professeur, INSA Lyon	Examinateur
Monsieur David Hill, Professeur, ISIMA	Examinateur

Résumé : Depuis le début des années 80, les bases de données biologiques n'ont cessé de gagner en volume. Une recherche sur ces bases qui ne prenait que quelques minutes peut désormais nécessiter plusieurs jours. En parallèle, la communauté de recherche en bioinformatique s'est développée et des laboratoires spécialisés sont nés partout dans le monde. La collaboration et l'échange de données entre équipes de recherche parfois géographiquement très éloignées a conduit à considérer la grille comme un moyen adapté à la fois aux nouveaux besoins en terme de puissance de calcul mais aussi comme outil de partage et de distribution des données biologiques entre chercheurs.

L'utilisation de la grille pour la recherche en biologie et bioinformatique est un atout considérable, cependant de nouvelles problématiques apparaissent quant à la gestion des données ainsi que dans l'ordonnancement des tâches qui doit prendre en compte la taille et la disponibilité des données. Cette thèse aborde ces problématiques nouvelles en prenant en compte les spécificités des bases de données biologiques pour une utilisation efficace de la grille.

Nous montrons l'intérêt des approches semi-statiques joignant réplications de données et ordonnancement des tâches. Pour cela, nous avons procédé en trois étapes : une analyse théorique, une première validation par simulation et enfin une implantation sur plateforme réelle.

La mise en place de la plateforme a mené à la conception d'un nouveau gestionnaire de données pour l'intergiciel DIET : DAGDA. Au-delà des applications de bioinformatique, ce gestionnaire de données peut répondre aux besoins de nombreuses applications portées sur les grilles de calcul.

Abstract : Since the beginning of the eighties, biological databases have considerably grown. A request on the bases which took only a few minutes can now need several days. Meanwhile, the research community in bioinformatics developped itself and specialised laboratories were created all around the world. Collaboration and exchanges between teams that were sometimes geographically very far from each other led to consider the grid as an adapted solution to the needs in computation power but also to share and distribute the biological data between the researchers.

We show the interest of semi-static approaches joining data replications and task scheduling. To this end, we proceeded in three steps : a theoretical analysis, a first validation through simulations, and an implementation on a real platform.

The development of this platform led to the conception of a new data manager for the DIET middleware : DAGDA. Beyond its biological applications, this data manager can meet the needs of numerous applications on computational grids.

Remerciements

Je tiens tout d'abord à remercier les membres du jury qui ont accepté d'évaluer mon travail de thèse.

Merci à Thierry Priol, directeur de recherche à l'INRIA Bretagne-Atlantique et à Pierre Sens professeur à l'Université de Paris 6, d'avoir accepté d'être rapporteurs de ce manuscrit. Merci également à Lionel Brunie, professeur à l'INSA de Lyon et à David Hill, professeur à l'ISIMA de Clermont-Ferrand pour avoir accepté d'être examinateurs et membres de mon jury de thèse.

Je remercie Frédéric Desprez, tout d'abord pour son rôle de directeur de thèse, ses conseils, les orientations qu'il m'a données, ses encouragements et tout le travail qu'on a réalisé ensemble et qu'il a toujours su mettre en valeur. Mais aussi pour son humour, sa simplicité et sa patience.

Je remercie bien entendu Vincent Breton, pour la co-direction de cette thèse, pour sa gentillesse et son professionnalisme.

Durant ces trois années, j'ai eu la chance de travailler avec Eddy Caron. Je le remercie tout particulièrement pour ses conseils, ses encouragements, ses idées, son dynamisme et toute l'aide qu'il m'a apportée. Sa capacité à envoyer des mails à trois heure ET à huit heure du matin, en étant toujours aussi efficace et de bonne humeur pendant la journée même avec douze heures de décalage horaire (et ceux qui connaissent Eddy savent que j'exagère à peine) reste un grand mystère pour moi.

Merci également à Yves Caniou pour ses conseils, sa gentillesse, pour les moments sympas passés à Seattle, à Boston, à Barcelone et bien sûr à Lyon.

Cette thèse, entre Clermont-Ferrand et Lyon, m'a permis de travailler avec deux équipes formidables. Je remercie toute l'équipe PCSV pour l'ambiance dans laquelle j'ai pu travailler. Je pense particulièrement à Jean, Arnaud, Vincent, Vinod, Joe, Ziad, Yannick, Géraldine, Ana, Manu, Matthieu. Et j'en oublie forcément... J'en oublierai sûrement autant dans l'équipe GRAAL de l'ENS de Lyon, mais un grand merci à Cédric, Ben, Véronika, Ghislain, David, Raphaël, Aurélien et tous les autres. Je remercie Frédéric Vivien de m'avoir accueilli dans son équipe durant cette dernière année.

J'ai une pensée toute particulière pour Jean-Sébastien (J.-S.) que je ne remercierai jamais assez pour son amitié et tout simplement pour ce qu'il est. On attend tous ton retour, et si tu ne peux malheureusement pas assister à ma soutenance, je ne raterai la tienne pour rien au monde.

D'un point de vue plus personnel, merci à Florence pour ses encouragements, pour ses conseils, pour ces milliers de kilomètres en train, pour ces week-ends trop courts, pour ces vacances (parfois), pour ces merveilleux moments passés ensemble (toujours) et pour tout le reste. Sans toi, l'achèvement de cette thèse n'aurait pas été possible.

Merci enfin à ma mère, à ma sœur et à mes amis qui ont toujours été là quand j'ai eu besoin de me détendre.

Table des matières

1 Introduction **17**

I Contexte : les bases de données biologiques et les grilles de calcul **23**

2 La gestion des données biologiques et leur utilisation sur la grille **25**
2.1 Les données biologiques 27
2.2 Les applications de bio-informatique 28
2.3 Basic Local Alignment Search Tool 29
2.3.1 L'algorithme de BLAST 30
2.4 Parallélisations de BLAST 33
2.4.1 Parallélisation du cœur de l'algorithme BLAST 34
2.4.2 Parallélisation des séquences de la base de données 35
2.4.3 Parallélisation des séquences de la requête 37
2.4.4 Conjonction des trois parallélisations possibles de BLAST 37
2.5 Problématiques générales de la gestion de données sur les grilles 39
2.5.1 L'approche par catalogues de données/meta-données 41
2.5.2 L'approche des "dépôts logistiques" de données 41
2.5.3 L'approche pair-à-pair 41
2.5.4 L'approche par système de fichiers 42
2.6 Les problématiques spécifiques à la gestion des données biologiques 42
2.7 L'intergiciel de la grille EGEE : gLite 43
2.7.1 Le Workload Management System (WMS) 44
2.7.2 R-GMA : Le système d'information de gLite 45
2.7.3 La gestion de données dans gLite 45
2.8 L'approche Grid-RPC proposée par DIET 48
2.8.1 Architecture de DIET 48
2.8.2 L'ordonnancement dans DIET 49
2.9 La gestion des données dans DIET 52
2.9.1 Data Tree Manager - DTM 52
2.9.2 La gestion des données avec JuxMem 56
2.10 Conclusion 60

3 Ordonnancement des tâches bio-informatiques sur plateformes hétérogènes **63**
3.1 Contexte 65
3.2 Problématique générale 67

3.3 Ordonnancement “Temps d’Accomplissement Minimum” 67
3.4 L’algorithme de Réplication et d’Ordonnancement : Schedule and Replication Algorithm 70
3.5 Conclusion 72

II Contribution à la gestion des bases de données biologiques sur les grilles de calcul 73

4 Un gestionnaire de données sur la grille : DAGDA 77
4.1 L’architecture de DAGDA 79
4.2 La gestion de données dans DIET avec DAGDA 81
4.2.1 La gestion des données dans les appels de procédures distantes de DIET 82
Les algorithmes de remplacement de données proposés par DAGDA . 83
4.2.2 La gestion directe par l’intermédiaire de l’API DAGDA 84
Ajout de données sur la grille 84
Récupération de données sur la grille 87
Partage des données 91
Sauvegarde de la plateforme (Checkpointing) 91
La réplication explicite des données 91
4.3 Utilisation de DAGDA pour la gestion de données 93
4.3.1 Utilisation d’ensembles de données 93
4.3.2 Réplication des données 93
4.3.3 Recouvrement des tâches de calcul et de transferts 95
4.3.4 Pré-chargement des données 97
4.4 Expérimentations 99
4.4.1 Performances des transferts 99
4.4.2 Persistance des données. 100
4.4.3 Réplication des données. 102
4.5 Conclusion 103

5 Le portage d’une application bio-informatique sur la grille 105
5.1 DIET-BLAST : Le portage de l’application BLAST sur l’intergiciel DIET . . . 107
5.1.1 L’application cliente 107
5.1.2 L’application serveur 112
5.2 Ordonnancement des tâches et placement des données 115
5.2.1 Objectifs 116
5.2.2 Algorithmes d’ordonnancement des tâches. 116
5.2.3 Évaluation des performances par simulations 118
5.2.4 Les algorithmes testés. 119
5.2.5 Configuration des simulations 120
5.2.6 Les simulations 120
Sans variation des fréquences des tâches 121
Variation des fréquences 123
Influence du choix du seuil 127
5.3 Implantation des algorithmes pour DIET-BLAST 127
5.4 Expérimentations sur une plateforme réelle 129

5.4.1 Les expérimentations . . . 130
Deux stratégies de découpage des requêtes. . . . 130
Comparaison des performances de MCT et de dynamique SRA pour un déploiement de 1000 nœuds . . . 133
5.5 Conclusion . . . 134

6 Conclusion et perspectives **135**
6.1 Contributions . . . 137
6.1.1 Ordonnancement des tâches bio-informatiques sur les grilles de calcul 137
6.1.2 DAGDA : Un gestionnaire de données pour l'intergiciel DIET . . . 137
6.1.3 DIET-BLAST . . . 138
6.2 Perspectives . . . 138

A Proposition d'API de gestion de données dans le GridRPC **149**

Table des figures

1.1 Évolution des capacités matérielles 19

2.1 Croissance de la base EMBL depuis 1982 27
2.2 Le format FASTA pour les données bio-informatiques. 28
2.3 Le format Staden pour les données bio-informatiques. 29
2.4 Alignement global de 2 séquences. 30
2.5 Plusieurs types d'alignements de séquences. 30
2.6 Modifications provoquées sur la protéine codée par un gène suite à une mutation sur un seul nucléotide. 31
2.7 Principe de l'algorithme de BLAST 32
2.8 Parallélisation des séquences de la base. 34
2.9 Parallélisation des séquences de la requête. 35
2.10 Temps d'exécution d'une requête BLAST sur une base de 150 Mo, avec mpiBLAST en fonction du nombre de nœuds et de la taille de la requête. 36
2.11 Architecture d'une plateforme de soumission BLAST pour la grille. 38
2.12 Architecture de LCG. 44
2.13 Organisation globale des composants de gLite 45
2.14 Le Workload Management System 46
2.15 Interactions entre les différents services de gestion de données dans gLite . . 47
2.16 Architecture de DIET 49
2.17 Étapes de soumission d'une tâche dans DIET 50
2.18 Organisation des classes d'ordonnancement dans DIET. 51
2.19 Politiques d'ordonnancement au niveau des agents. 52
2.20 Organisation de DTM au sein de DIET. 53
2.21 Deux cas d'utilisation de données persistantes. 54
2.22 Cas d'utilisation d'une donnée persistante avec retour. 55
2.23 Cas d'utilisation d'une donnée "sticky". 56
2.24 Exemple de problèmes résultant des limitations de DTM. 57
2.25 Interactions DIET-JuxMem 59

3.1 Modèle de soumission des tâches 66
3.2 Diagrammes des différents ordonnancements. 69

4.1 Architecture générale de DAGDA 80
4.2 Architecture interne des objets DAGDA 81
4.3 Gestion avec DAGDA 83

4.4 Ajout d'une donnée par l'intermédiaire de l'API au niveau du client. 85
4.5 Ajout d'une donnée par l'intermédiaire de l'API au niveau d'un agent. 85
4.6 Ajout d'une donnée par l'intermédiaire de l'API au niveau d'un SeD. 85
4.7 Récupération d'une donnée avec l'API DAGDA. 89
4.8 Exemple d'utilisation de la fonction de réplication de l'API DAGDA. 93
4.9 Utilisation d'un ensemble de données dont le nombre est déterminé dynamiquement avec DIET. 94
4.10 Utilisation d'un ensemble de données dont le nombre est déterminé dynamiquement avec DAGDA. 94
4.11 Réplication des données et ordonnancement Round-Robin. 95
4.12 Recouvrement temps de calcul / temps de transfert avec DAGDA. Utilisation de deux SeDs. 96
4.13 Recouvrement temps de calcul / temps de transfert avec DAGDA. Un seul SeD enchaîne les deux calculs. 97
4.14 Pré-chargement systèmatique sur l'ensemble des nœuds offrant le service. . . 98
4.15 Pré-chargement sur les nœuds sélectionnés par le plugin de l'agent. 99
4.16 Pré-chargement sur l'agent afin d'approcher la donnée des nœuds de calcul. . 100
4.17 Moyenne des temps de transfert de données de 10 Mo à 1 Go avec DAGDA, DTM et scp. 101

5.1 Principe du découpage en fichiers de taille choisie 109
5.2 Organisation générale de l'application cliente de DIET-BLAST. 112
5.3 Organisation générale de l'application serveur de DIET-BLAST 115
5.4 Variations locales de fréquences conservant les proportions globales. 116
5.5 Architecture simulée par OptorSim 118
5.6 Architecture simulée par OptorSim après nos modifications 119
5.7 Configuration de la plateforme simulée. 121
5.8 Greedy sans réplication. 122
5.9 MCT. 122
5.10 SRA avec réplications synchrones. 123
5.11 SRA avec réplications asynchrones. 124
5.12 Approche hybride - Distribution des données SRA / Ordonnancement MCT. 124
5.13 Sans correction des placements. 125
5.14 Variation continuelle des fréquences. 126
5.15 Variation ponctuelle des fréquences. 126
5.16 Seuil fixé à 5% 127
5.17 Seuil fixé à 10% 128
5.18 La répartition géographique des différents sites de Grid'5000 et leur réseau d'interconnexion. 130
5.19 Temps moyen d'exécution pour quatre algorithmes d'ordonnancement différents avec la stratégie de découpage maximal des requêtes. 132
5.20 Temps moyen d'exécution pour quatre algorithmes d'ordonnancement différents avec la stratégie de découpage des requêtes en le nombre de nœuds disponibles. 132
5.21 Temps d'exécution moyen pour l'ensemble des requêtes de chaque type sur 1000 nœuds en utilisant l'algorithme MCT. 133

5.22 Temps d’exécution moyen pour l’ensemble des requêtes de chaque type sur 1000 nœuds en utilisant l’algorithme SRA dynamique. 134

A.1 Data locations in the GridRPC model 152
A.2 Simple RPC call with input and output data. 165
A.3 Simple RPC call with input and output data using external storage resources. 166
A.4 GridRPC call with data management using persistence through the GRPC_STICKY mode. 167
A.5 GridRPC call with data prefetching using the API. 168
A.6 Three RPC call with data management using persistence. 169

Chapitre 1

Introduction

Il est généralement admis que l'évolution de la puissance de calcul des processeurs est régie par la célèbre loi de Moore, promettant le doublement des capacités des processeurs tous les dix-huit mois. Deux autres "lois" d'évolution des capacités du matériel informatique correspondent aux constatations effectuées ces deux dernières décennies : La loi de stockage (Storage Law) qui estime la croissance des capacités de stockage comme étant multipliées par seize tous les quatre ans et la loi de Gilder (Gilder's Law) qui veut une multiplication par trente-deux de la bande passante réseau maximale sur la même période (voir la figure 1.1). Derrière ces trois lois, d'usage très général se cache l'intérêt même du concept de grille de calcul pour le calcul scientifique. En effet, si on dispose d'une capacité de calcul qui croît moins vite que la capacité de stockage des données à traiter, on se destine à des temps d'attente de plus en plus longs si tant est qu'on soit capable de produire suffisamment de données à analyser. Or, si par ailleurs, les capacités de transmission des données par les réseaux augmentent encore plus vite, l'exécution des tâches de calcul à distance et en paralléle permet d'envisager des gains de performances très importants pour des tailles de données toujours plus grandes.

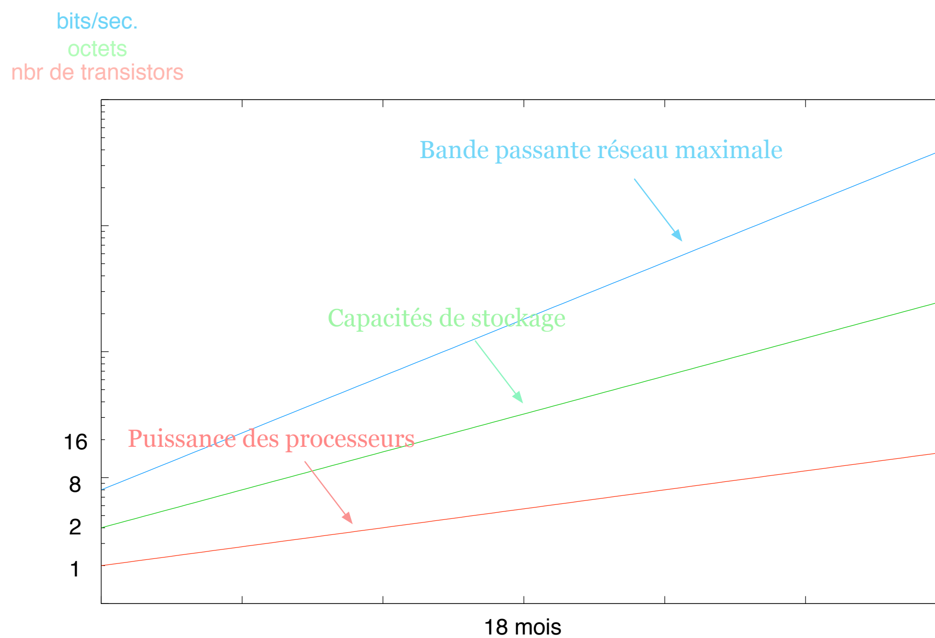

FIG. 1.1 – Évolution des capacités matérielles

Depuis le début des années quatre-vingts, les programmes d'acquisition de données biologiques, tels que les projets de séquençage de génomes complets se sont multipliés tout autour du monde. On recensait une dizaine de bases de données biologiques il y a vingt-cinq ans. De nos jours, c'est plus de mille bases qui sont produites et mises à jour par de nombreux laboratoires. Par ailleurs, les moyens de découverte/production des données biologiques n'ont cessé de s'améliorer et la taille de ces bases évolue de manière exponentielle. Par exemple, la base EMBL [60] a aujourd'hui une taille supérieure à 230 giga-octets. Ainsi, l'exécution d'un programme d'analyse qui ne nécessitait que quelques minutes sur une machine locale aux débuts des projets de bio-informatique peut aujourd'hui demander plu-

sieurs jours de calcul. De plus, la multiplication des sources de données conduit à une gestion de plus en plus complexe des bases notamment pour ce qui est de leur mise à jour et l'accès aux dernières versions des outils d'analyse permettant de confronter les différents résultats obtenus est une tâche indispensable qui peut vite s'avérer fastidieuse et contraignante.

Comme pour la physique des hautes énergies qui partage ces caractéristiques de volume (à une échelle qui est même supérieure) et de diversité des données à traiter, la grille présente un intérêt tout particulier pour la recherche en biologie. Plusieurs projets autour du monde ont ainsi pour objectif de fournir l'infrastructure logicielle à la conception de grilles pour la bio-informatique. Ces projets reposent généralement sur les standards "de facto" de la grille que sont Globus [47] et Condor [43]. L'objectif principal pour l'analyse des données en sciences de la vie est d'offrir une interface d'accès transparent aux ressources de la grille quelles soient des ressources physiques (stockage, processeur etc.) ou logiques (les données elles-mêmes) aux biologistes, souvent peu familiers des intergiciels de grilles et des notions avancées d'informatique qu'ils nécessitent. Des projets tels que le Canadian BioGrid [91], EuroGrid BioGrid [13], Asia Pacific BioGrid [101], GenoGrid [65], le Cancer biomedical informatics Grid [34, 82, 24], le projet Simdat [7] et le Biomedical Informatics Research Networks [49] partagent capacités de calcul et de stockage par l'intermédiaire de pages web dédiées offrant un accès publique aux bases de données et aux outils d'analyse. En Europe, EGEE vise à déployer une infrastructure de grille sécurisée offrant des ressources aux différentes communautés de chercheurs et notamment la communauté de recherche en sciences de la vie [55]. Des projets comme BioinfoGRID [69] l'utilisent pour la production de données ou pour leur analyse dans le domaine de la biologie moléculaire.

D'autres projets comme MyGrid [92], Proteus [25] et ProGenGrid [2] développent des environnement de travail personnalisables permettant l'integration de données distribuées et le management de workflows. Ils reposent sur des technologies de web sémantique permettant la manipulation des objets biologiques et des concepts tout en cachant à l'utilisateur les opérations techniques sous-jacentes. La réalisation de tels environements est facilitée par l'utilisation d'interfaces standards pour l'accès aux outils bioinformatiques et aux bases de données biologiques. Le réseau d'excellence Embrace [41] travaille à l'integration des données de la biologie moléculaire par l'intermédiaire d'accès standardisés aux ressources bioinformatiques à travers l'Europe. La standardisation devrait également permettre de faciliter l'accès aux données, offrant aux fournisseurs de celles-ci des standards bien définis permettant aux biologistes de tirer le meilleur parti de données distribuées sur la grille.

Pour ce qui est de l'aspect plus "technique" des grilles, l'Open Grid Forum [42] s'efforce de proposer une convergence vers des standards les plus ouverts possible et couvrant les besoins des différents "types" de grilles afin de fédérer les différentes approches et les différents projets.

Le chapitre suivant propose un aperçu de la gestion de données sur grilles de calcul et les efforts réalisés pour la parallélisation d'une application bio-informatique très populaire : le Basic Local Alignment Search Tool (BLAST). Nous verrons ensuite les nouvelles problématiques posées par l'utilisation des grilles de calcul pour la gestion et l'utilisation des bases de données biologiques. Le chapitre 4 sera consacré à la présentation de Data Arrangement for the Grid and Distributed Applications (DAGDA), le nouveau gestionnaire de données développé durant ma thèse et désormais intégré à DIET. Le chapitre 5 présentera notre implantation de l'outil BLAST utilisant l'intergiciel de grille Distributed Interactive

Engineering Toolbox (DIET) et les performances obtenues par celui-ci en utilisant différents algorithmes d'ordonnancement des tâches. Nous aborderons ensuite le travail entrepris pour l'ordonnancement des tâches bio-informatiques et la réplication des bases de données biologiques sur la grille.

Nous conclurons enfin et présenterons les perspectives du travail réalisé. Une annexe supplémentaire présentera enfin le travail de standardisation proposé à l'OGF dont une partie de la réflexion repose sur le travail entrepris pour la réalisation de DAGDA.

Première partie

Contexte : les bases de données biologiques et les grilles de calcul

Chapitre 2

La gestion des données biologiques et leur utilisation sur la grille

2.1 Les données biologiques

L'extraction et la collecte des données biologiques a connu des progrès énormes depuis les années soixante-dix. En effet, des découvertes comme la méthode de Sanger pour le séquençage de l'ADN [87] ont permis d'automatiser une partie de l'extraction de connaissances des entités biologiques et ont conduit à la croissance exponentielle des banques de données qui y sont consacrées. La figure 2.1 présente la croissance de la base EMBL de 1982 jusqu'à aujourd'hui. De nombreux détails statistiques sur la base EMBL sont disponibles sur le site internet de l'EBI (European Bioinformatics Institute) [54].

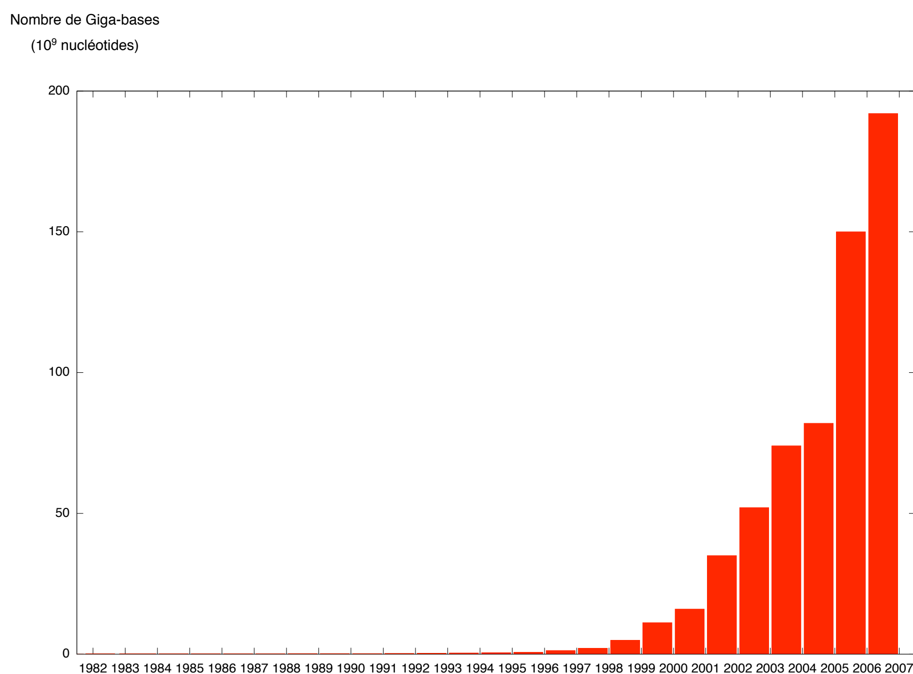

FIG. 2.1 – Croissance de la base EMBL depuis 1982

On distingue généralement deux types de bases de données biologiques :

- Les bases dites "généralistes" : Ce sont des bases publiques internationales qui recensent des séquences d'ADN ou de protéines déterminées quelque soit l'organisme ou la méthode utilisée pour les acquérir. La quantité de données annotées qu'elles contiennent, le nombre de séquences, dont certaines ne sont plus disponibles ailleurs, leur grande diversité et la qualité des annotations en font des outils indispensables pour la communauté scientifique. Elles présentent cependant quelques défauts qui ont conduit à la création de bases plus spécifiques : Les données qu'elles contiennent n'ont pas toujours été vérifiées, elle sont parfois trop diverses et nombreuses pour être exploitées efficacement et sont parfois mises à jour avec un certain retard.
- Les bases dites "spécialisées" : Ce sont des bases dont les données ont été classées suivant une caractéristique biologique particulière. Il s'agit par exemple des bases recensant toutes les séquences d'un même génome.

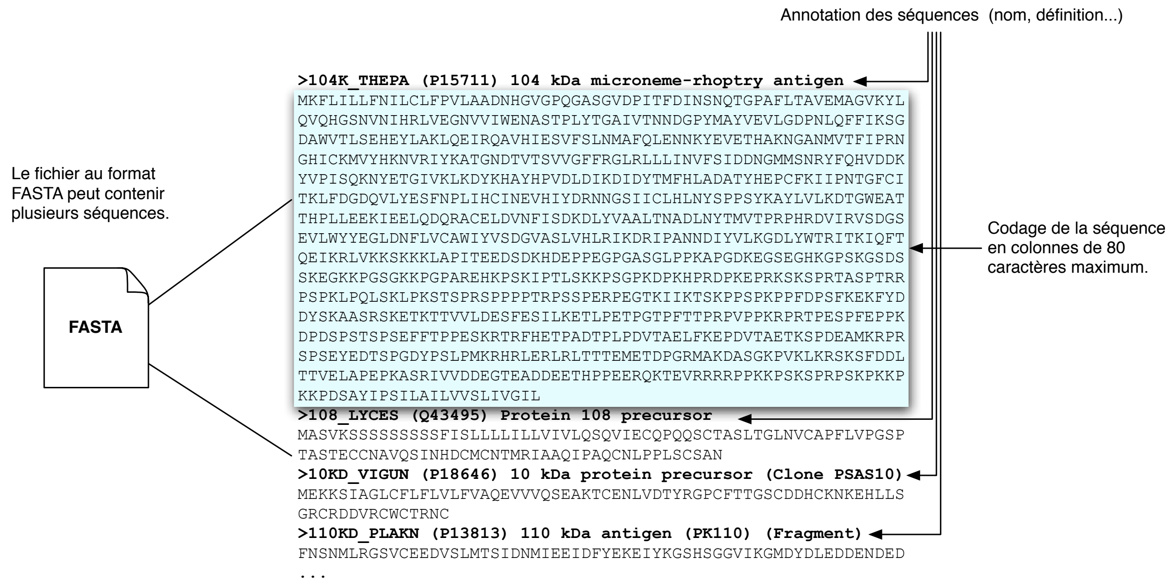

FIG. 2.2 – Le format FASTA pour les données bio-informatiques.

On citera notamment les bases EMBL (European Molecular Biology Laboratory) [60], GenBank [19] ou encore Swissprot [12] au rang des bases généralistes et la base de structure PDB (Protein Data Bank) [20] et la base Kegg (Kyoto Encyclopedia of Genes and Genomes) [71] comme bases spécialisées parmi les plus connues.

L'essor d'Internet a par ailleurs grandement amélioré la diffusion de ces bases qui étaient transmises par courrier jusqu'au début des années quatre-vingt-dix et permis un accès simplifié et immédiat à celles-ci. Aujourd'hui les bases les plus connues comptent des dizaines de milliers d'utilisateurs et des mises à jour quasi-quotidiennes.

Ces bases étant généralement stockées sous forme de fichiers plats, des logiciels d'interrogation spécialisés ont vu le jour. Le logiciel ACNUC [48] est par exemple utilisé pour des requêtes multi-critères sur le contenu de ces bases.
Les figures 2.2 et 2.3 présentent deux formats de fichiers utilisés pour les bases de données biologiques. Le format "historique" Staden et le format FASTA plus largement utilisé de nos jours. Le format Staden ne permet de stocker qu'une seule séquence sans annotation. Le format FASTA, permet quant à lui de regrouper plusieurs séquences dans un même fichier et d'annoter chacune d'entre elles.

2.2 Les applications de bio-informatique

L'analyse et l'utilisation des données biologiques font l'objet de nombreuses recherches depuis plusieurs décennies. Ainsi, de nombreuses applications permettant de tirer profit des masses de données accumulées ont vu le jour répondant chacune à différents besoins exprimés par les biologistes. On distinguera plusieurs types d'applications visant des

FIG. 2.3 – Le format Staden pour les données bio-informatiques.

objectifs bien distincts :

- L'analyse et la comparaison des données acquises par expériences ou utilisation de matériel spécifique. Les applications de type BLAST [4, 3], FASTA [73] ou encore Clustal [51] sont utilisées pour tirer de nouvelles connaissances des données accumulées.
- La simulation *in-silico*. Il s'agit ici de simuler des phénomènes biologiques et leurs conséquences par ordinateur. Par exemple, on pourra simuler le matériel de radiothérapie afin de proposer les configurations physiques les plus adaptées en terme d'efficacité du traitement tout en limitant au maximum les effets indésirables de ce type de traitement. En biologie moléculaire, on pourra sélectionner des molécules en simulant les interactions qu'elles peuvent avoir avec des protéines spécifiques des parties actives d'un virus. Les logiciels de docking comme FlexX [80] ou autodock [70] permettent d'effectuer de telles simulations et ont été utilisés avec succès dans le cadre de data-challenges réalisés sur la grille EGEE [56].
- Le croisement et l'extraction de connaissances des documents scientifiques et résultats d'expériences afin d'en tirer de nouvelles connaissances.

Dans ces trois cas, la répartition géographique des sources de données, la masse des données à stocker et la puissance de calcul nécessaire à leur analyse font de la grille un moyen particulièrement adapté au déploiement à grande échelle de ces applications.

2.3 Basic Local Alignment Search Tool

BLAST [4, 3] est certainement l'application de bio-informatique la plus utilisée par les biologistes autour du monde. Utilisée pour la recherche de similitudes entre séquences de nucléotides ou d'acides aminés, elle a pour rôle de tenter de déterminer la fonction d'un gène ou d'une protéine par comparaison avec des séquences dont le rôle est bien connu. Comme son nom l'indique, BLAST détecte les similarités locales entre deux séquences, similarités

qui ont en général plus de sens d'un point de vue biologique que des similarités globales qui peuvent apparaître entre des séquences aux rôles forts différents. La figure 2.4 illustre une telle situation où un alignement complet peut être trouvé entre deux séquences d'un point de vue global et inexistant d'un point de vue local. Ce genre de situation est particulièrement

```
...ACAATGGTACGTAATCCCACCACCATTGGACTCCATGGTTGCTGCGGCTACCCAAGTCACGTGGCTTTGACTACTATCCTAGGACTGAG...
   A   T   A  TA T            G   T    G T          A      T A  T       A T         G
```

FIG. 2.4 – Alignement global de 2 séquences.

courant dans l'utilisation habituelle des applications de recherche d'alignement, à savoir la recherche d'alignement entre une nouvelle séquence, plutôt courte et une base de donnée complète. Par ailleurs, la recherche d'alignements locaux permet de détecter des rôles communs à deux protéines/gènes par ailleurs très différents en se concentrant sur leurs points communs qui peuvent être le codage d'un site actif particulier d'une protéine ou d'un gène codant une protéine. Le principe sur lequel est fondé BLAST est une méthode heuristique qui *quantifie* les similarités entre séquences en utilisant une matrice de score de similarité adaptée aux constatations biologiques (fréquence de mutation d'un acide aminé en un autre, impossibilité de certaines mutations etc). BLAST permet par ailleurs à ses utilisateurs de définir des paramètres de sensibilité, comme les seuils au-delà desquels l'algorithme renoncera à aligner deux séquences.

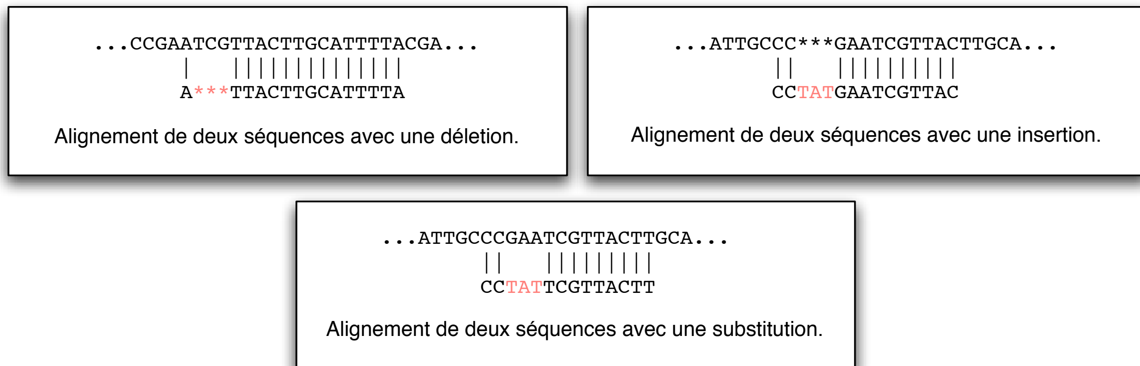

FIG. 2.5 – Plusieurs types d'alignements de séquences.

La figure 2.5 présente différents exemples d'alignements de séquences. L'un avec *délétion*, c'est-à-dire la disparition d'un certain nombre de nucléotides dans la séquence. Un autre avec *insertion*, c'est à dire avec l'ajout d'un certain nombre de nucléotides dans la séquence. Et enfin, un alignement avec *substitution*, c'est à dire que des nucléotides de la séquence se sont vus remplacés par d'autres.

2.3.1 L'algorithme de BLAST

L'algorithme de BLAST repose sur deux "intuitions" biologiques :

- Une forte similarité entre séquences traduit *l'homologie* entre ces séquences. On suppose qu'à partir d'un certain seuil de ressemblance, celle-ci ne peut être dûe au hasard

seul et les protéines ou les gènes qui codent ces protéines ont un ancêtre commun. Leurs différences ne sont alors que le résultat de mutations, insertions ou délétions apparues au cours du temps.
– L'homologie entre deux séquences peut traduire, sans toutefois le prouver, un rôle identique ou proche de la séquence dans la protéine ou dans le gène.

Il est toutefois à noter qu'il arrive qu'une forte similarité ne traduise pas une homologie. BLAST ne doit donc être utilisé que comme un outil de recherche d'indice de fonction d'une protéine ou d'un gène et une détection de similarité même forte entre deux séquences n'est en aucun cas une preuve d'homologie entre celles-ci. L'interprétation des résultats obtenus par BLAST est un travail à part entière nécessitant de solides connaissances en biologie et en biologie moléculaire.

Partant de ces deux idées, les auteurs de BLAST ont développé une méthode heuristique permettant de détecter de telles similarités entre séquences. BLAST quantifie (i.e. attribue une valeur) la similarité entre séquences en calculant le nombre d'appariements pénalisé par le nombre de mésappariements pour la position d'alignement optimal trouvée entre celles-ci. Le calcul de ce score est paramétrable pour pouvoir s'adapter aux spécificités de la biologie. Par exemple, la disparition ou l'ajout d'un nucléotide dans une séquence ADN peut avoir plus d'importance que le même événement pour trois nucléotides, représentant alors l'ajout ou la suppression d'un acide aminé dans la protéine codée. Dans un cas, le codage de la protéine sera très différent dans l'autre, seul un acide aminé est ajouté ou supprimé dans la protéine. La figure 2.6 présente un tel cas de figure. On y voit un exemple où la mutation d'un seul nucléotide transforme le codon de l'acide aminé *lysine* en un "codon stop" qui arrête la transcription de la protéine.

Le gène original est transcrit en une protéine.

TTTGCTCACTTGAAATTTTTTTCTTGTTCTTTTGCTTTG

TTT GCT CAC TTG AAA TTT TTT TCT TGT TCT TTT GCT TTG

Transcription ADN ⇒ protéine

F A H L K F F S C S F A L

FAHLKFFSCSFAL

Une mutation se produit sur un seul nucléotide de la séquence originale.

TTTGCTCACTTGTAATTTTTTTCTTGTTCTTTTGCTTTG

TTT GCT CAC TTG **TAA** TTT TTT TCT TGT TCT TTT GCT TTG

Transcription ADN ⇒ protéine

F A H L **STOP** ~~F F S C S F A L~~

FAHL

La protéine produite est très différente de celle issue du gène original.

FIG. 2.6 – Modifications provoquées sur la protéine codée par un gène suite à une mutation sur un seul nucléotide.

Dans le cadre d'une recherche d'alignement entre séquences protéiniques, le choix et la conception de la matrice de substitution a un rôle très important. En effet, d'un point de vue biologique, certaines mutations n'ont aucune conséquence sur la fonction de la protéine. On parle alors de mutations conservatives et elles sont jugées comme probables et sans grande importance. D'autres mutations ont au contraire un effet particulièrement "néfaste" sur le phénotype et sont donc jugées comme plus pénalisantes pour le calcul du score.

Ces spécificités biologiques sont prises en compte par l'algorithme de BLAST qui permet de définir différents paramètres de sensibilité. La figure 2.7 présente le fonctionnement de l'algorithme.

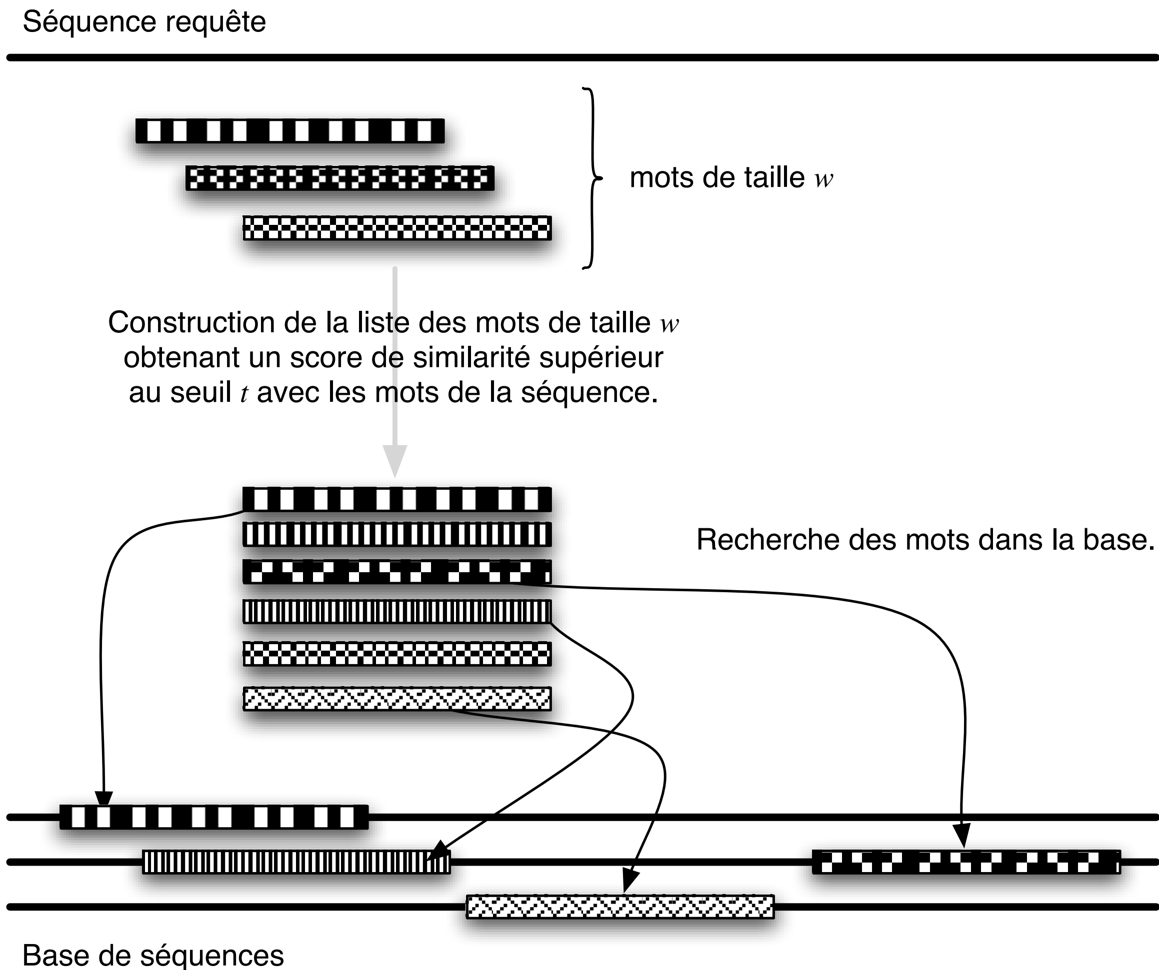

FIG. 2.7 – Principe de l'algorithme de BLAST

L'algorithme de BLAST se déroule en trois phases principales :

- La recherche des alignements maximums entre paires de séquences de taille w. C'est à dire les sous-segments de taille w qui présentent les meilleurs scores d'alignement parmi tous les alignements possibles entre toutes les pairs de sous-séquences de taille w possibles. Une liste de mots de taille w est ainsi construite pour servir de points d'ancrage aux extensions des alignements effectuées dans la suite de l'algorithme. Il s'agit ici pour l'algorithme d'éviter de procéder à des extensions d'alignement courts qui ont peu de chance d'atteindre la valeur seuil à partir de laquelle un alignement est jugé comme "bon".
- La liste des mots de taille w est alors cherchée dans la base et servira comme point de départ aux extensions procédées à l'étape suivante.

– L'extension des alignements de taille w trouvés dans la base et la sélection de ceux qui présentent un score supérieur à une valeur seuil choisie par l'utilisateur.

L'idée exploitée par BLAST est donc qu'un "bon alignement" doit contenir des segments strictement identiques, points d'ancrage des extensions des alignements exécutées dans la suite de l'algorithme. La première étape de l'algorithme se déroule différemment lorsqu'on compare des séquences ADN et lorsqu'on compare des séquences protéiniques. Dans le premier cas, tous les mots de taille w tirés de la séquence "requête" sont utilisés, dans le second cas, seuls ceux dont le score dépasse une valeur seuil T sont retenus pour la construction de la liste.

BLAST se décline en cinq versions qui permettent chacune de réaliser un type de recherche différent :

– BLASTN : Procède à la comparaison d'une séquence nucléique avec une base de séquences nucléiques. (ADN contre ADN)
– BLASTP : Procède à la comparaison d'une séquence protéique avec une base de séquences protéiques. (Protéine contre Protéine)
– TBLASTN : Procède à la comparaison d'une séquence protéinique avec la traduction en protéines des séquences d'une base nucléique. (Protéines contre ADN)
– BLASTX : Procède à la comparaison d'une séquence nucléique avec une banque de séquences protèiques. (ADN contre Protéine)
– TBLASTX : Procède à la comparaison d'une séquence nucléique traduite en séquence protéique avec les séquences d'une base de séquences nucléiques elles-même traduites en séquences protéiques.

Chacune des versions de BLAST présentées ci-dessus est d'une complexité différente, en fonction de l'alphabet d'expression des séquences (quatre lettres pour les séquences nucléiques, vingt pour les séquences protéiques), et des traductions à effectuer. Ainsi, une recherche d'alignement avec BLASTN qui s'applique sur des séquences toutes codées sur un alphabet de quatre lettres s'exécute beaucoup plus rapidement qu'une recherche avec TBLASTX, qui nécessite à la fois une traduction de la séquence soumise et des séquences de la base pour effectuer une recherche sur un alphabet de vingt lettres.

2.4 Parallélisations de BLAST

Étant données la croissance en taille des bases de données utilisées pour les recherches d'alignement et la popularité chez les biologistes de l'outil BLAST, de nombreux travaux ont été entrepris pour accélérer l'exécution de l'algorithme. Dans ce cadre plusieurs approches de parallélisation ont été proposées [30, 74, 99]. On identifie trois grandes approches dans la parallélisation de BLAST qui peuvent être appliquées de manière complémentaire afin de tirer le meilleur parti des ressources mises à disposition par les grilles de calcul :

– La parallélisation du cœur de l'algorithme lui-même. Chaque étape de l'algorithme est parallélisée en utilisant plusieurs processus simultanément sur une machine SMP. Cette solution est mise en œuvre directement dans les dernières versions des implantations de BLAST et tire parti des systèmes multi-processeurs.

– La parallélisation des séquences de la base cible. Chaque recherche d'alignement est effectuée en parallèle sur des portions différentes de la base. La figure 2.8 présente ce principe de parallélisation de l'algorithme détaillé dans la section 2.4.2.
– La parallélisation des séquences de la requête. Une requête BLAST peut contenir plusieurs séquences à aligner avec celles de la même base. On procède alors aux recherches d'alignements de sous ensembles de la requête avec la base complète. La figure 2.9 présente ce principe de parallélisation de l'algorithme détaillé dans la section 2.4.3.

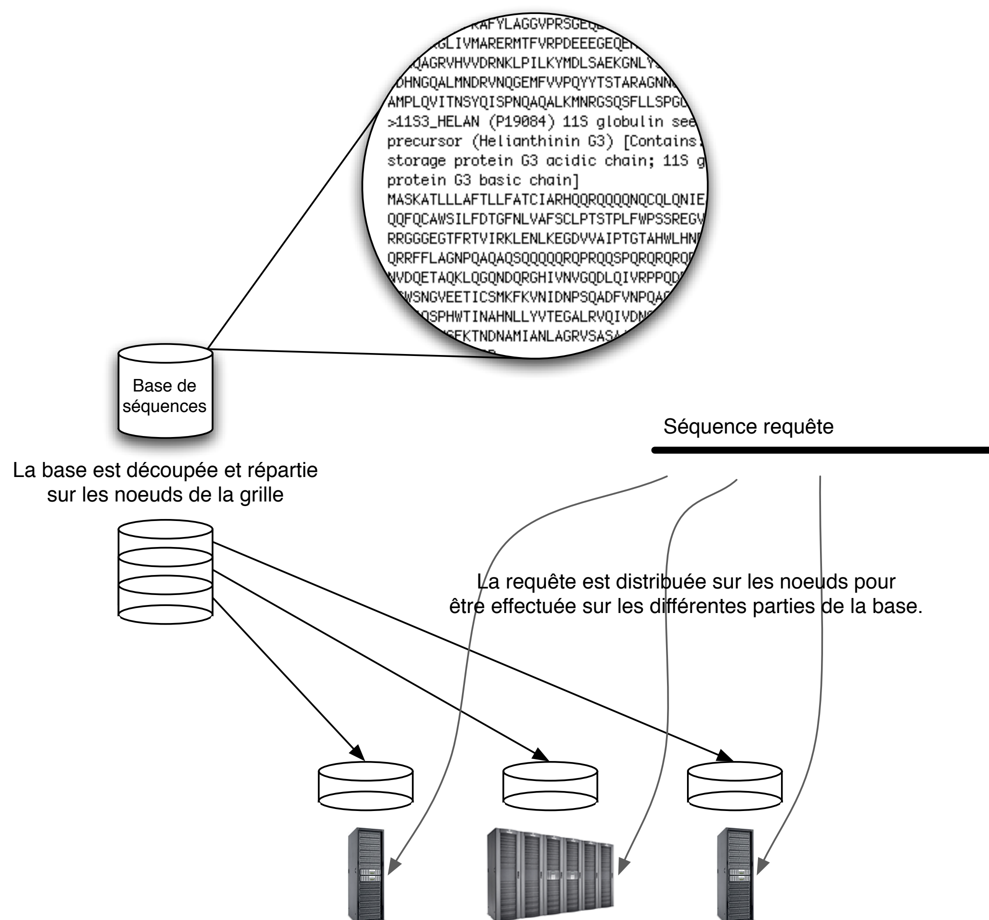

FIG. 2.8 – Parallélisation des séquences de la base.

2.4.1 Parallélisation du cœur de l'algorithme BLAST

La première approche de parallélisation de BLAST consiste en la parallélisation des différentes étapes de l'algorithme lui-même. Il s'agit ici de paralléliser l'algorithme de telle sorte qu'une recherche d'alignement entre deux séquences utilise plusieurs processeurs contrairement aux autres approches qui consistent à effectuer les recherches d'alignements entre plusieurs séquences, qu'elles soient tirées de la base ou de la requête, en parallèle. Dans [61], les auteurs analysent les résultats obtenus par ce type de parallélisation sur

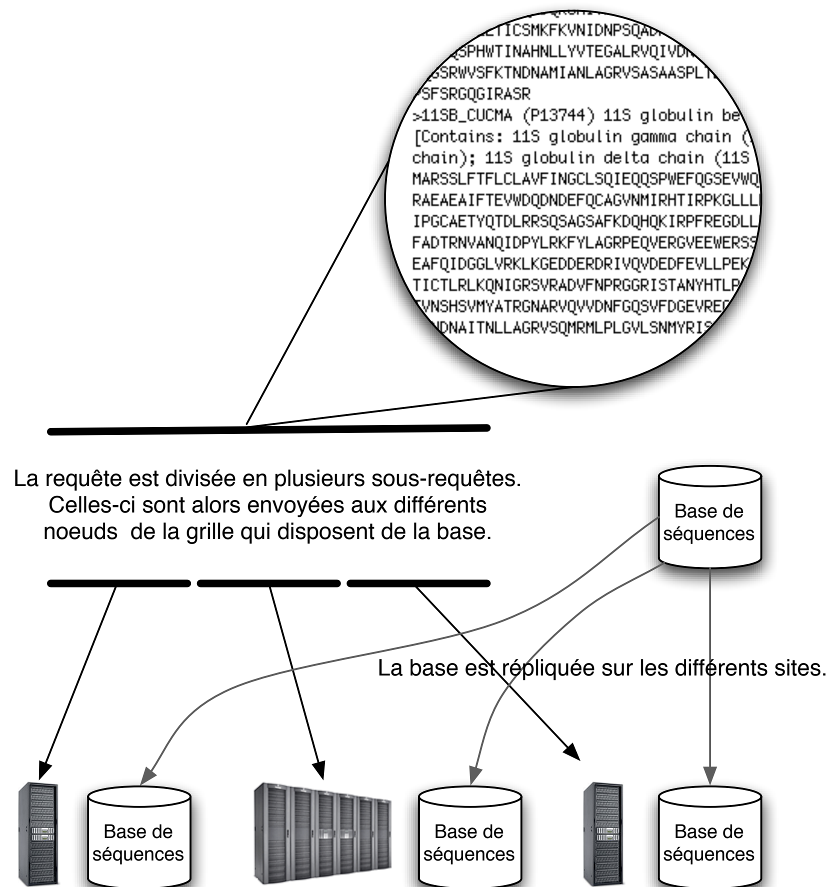

FIG. 2.9 – Parallélisation des séquences de la requête.

une machine disposant de 8 processeurs. Lorsque les séquences à traiter sont suffisament longues, on obtient par ce biais une accélération linéaire en fonction du nombre de processeurs utilisés. L'utilisation de matériel spécifique comme les machines IBM blue/gene permet également d'obtenir d'excellentes performances avec une accélération presque linéaire en fonction du nombre de processeurs mis en œuvre. Dans [79], 2000 processeurs sont utilisés et le temps de calcul est considérablement réduit pour la recherche d'alignement sur une longue séquence. Plus récemment, des chercheurs ont proposé une implantation de l'algorithme de recherche du meilleur alignement global (l'algorithme de Smith-Waterman) utilisant les capacités de parallélisme massifs des GPUs récents [68]. Les algorithmes de recherche d'alignements correspondent bien à ce type de parallélisation et les auteurs ont montré qu'une bonne implantation donne de très bons résultats.

2.4.2 Parallélisation des séquences de la base de données

La seconde approche consiste à distribuer la bases de données elle-même sur plusieurs machines [1, 23, 30, 74]. Chaque requête est alors distribuée sur des parties différentes de la base sur laquelle elle s'applique. Cependant, les résultats d'alignements donnés par l'algorithme BLAST sont classés suivant une valeur d'espérance à la fin du processus. Cette valeur

appelée "e-value" est l'expression du nombre d'alignements que l'on peut attendre d'une séquence aléatoire obtenant un score supérieur au score obtenu dans la banque considérée. Ainsi, cette valeur dépend de la base ciblée et n'est pas comparable entre deux banques de données différentes. C'est le principal frein à la parallélisation des recherches d'alignements en parallélisant les séquences d'une base. En effet, le fait de "découper" la base de données modifie irrémédiablement l'e-value des résultats d'alignements avec la séquence requête. On ne peut donc se contenter de diviser la base d'entrée, effectuer les recherches en parallèle et concaténer les résultats. L'implantation mpiBLAST de l'algorithme utilisant cette parallélisation sur plusieurs machines en utilisant l'interface MPI résoud le problème en effectuant des pseudos requêtes sur la base complète permettant de corriger les e-values obtenues sur les portions de base. C'est à notre connaissance l'unique implantation de ce principe pour BLAST et elle ne permet d'effectuer un partitionnement de la base que d'une manière homogène. C'est à dire que la base considérée est divisée en portions de tailles identiques. Dans le cadre d'une plateforme hétérogène, cette limitation peut s'avérer problématique. De plus, il s'avère que cette implantation utilisant MPI pour les communications entre les différents processus ne passe pas bien à l'échelle et que les performances se dégradent passé un certain nombre de machines.

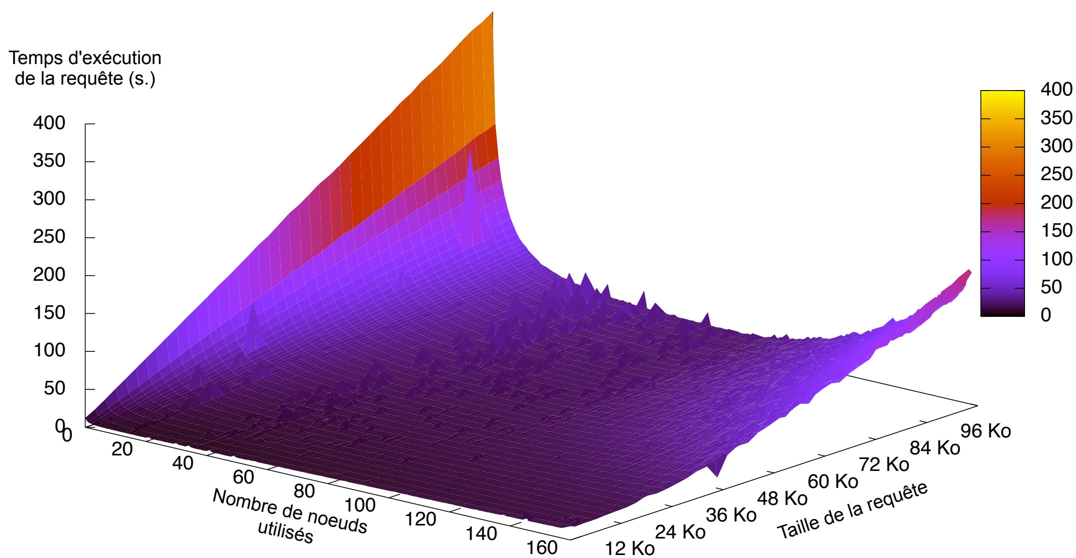

FIG. 2.10 – Temps d'exécution d'une requête BLAST sur une base de 150 Mo, avec mpiBLAST en fonction du nombre de nœuds et de la taille de la requête.

La figure 2.10 présente les résultats obtenus pour un nombre croissant de machines et pour une taille croissante de requête. On constate que passé soixantes machines environ, pour des requêtes de cinquante kilo-octets et plus, les performances se dégradent. On constate également que l'utilisation de quatre machines ou moins donne des résultats moins bon qu'une exécution d'un simple BLAST sur une seule machine. Néanmoins, en divisant

la base de données en portions plus petites, on peut s'assurer que celles-ci peuvent entrer complètement en mémoire de chacune des machines utilisées. On évite ainsi de coûteux accès disque accélérant d'autant la recherche d'alignements. En appliquant ces principes, mpiBLAST apporte des améliorations très importantes aux recherches d'alignements sur de grosses bases en utilisant un cluster de taille moyenne.

2.4.3 Parallélisation des séquences de la requête

Très couramment, les biologistes cherchent les alignements d'un grand nombre de séquences avec une ou plusieurs bases de données. Les recherches de similarités ainsi effectuées sont totalement indépendantes et on peut envisager de les effectuer en parallèle sans aucun problème. Le résultat final consistant en une simple fusion des résultats individuels. Cette fusion, qui peut consister en une simple concaténation ordonnée dans le cas le plus simple (l'ordre étant dicté par l'ordre d'apparition dans le fichier de requête, aucun tri n'est nécessaire à la réalisation de cette tâche) consiste au pire en la fusion de plusieurs fichiers XML. Dans tous les cas, tout comme la division du fichier d'entrée, le temps de calcul nécessaire à cette tâche peut être considéré comme négligeable vis-à-vis du temps nécessaire à l'exécution de l'algorithme BLAST lui-même. Plusieurs implantations de ce principe ont été réalisées ([74], [23], [21], [97]). Dans toutes ces implantations, les fichiers de requêtes sont divisés en parts de taille égale. Sur une plateforme hétérogène, lorsque le nombre de séquences contenues dans la requête est suffisamment important et que celle-ci est divisée au maximum, l'algorithme d'ordonnancement peut équilibrer la charge sur les différentes machines au seul coût de l'overhead réseau provoqué par la transmission de nombreux petits messages. Cette parallélisation nécessite une gestion des réplications des bases de données et doit prendre en compte l'espace disponible sur les différentes machines, le coût des transferts nécessaires à la replication des bases. De plus, les mises à jour des différentes bases peuvent devenir des tâches complexes et sources potentielles d'erreurs. À notre connaissance, toutes les implantations publiques de ce principe nécessitent une gestion des données par l'utilisateur qui décidera où et en quelle quantité répliquer les données sur les différents serveur disponibles.

L'implantation de BLAST pour DIET réalisée dans le cadre de ce travail utilise ce principe et plusieurs résultats sont présentés dans le chapitre 5. De plus DIET-BLAST introduit la gestion automatique des réplications et distributions des bases grâce au gestionnaire de données DAGDA développé en parallèle.

2.4.4 Conjonction des trois parallélisations possibles de BLAST

La fédération de ressources hétérogènes effectuée par les intergiciels de grilles permet d'envisager de tirer profit des trois parallélisations présentées ci-dessus. Ainsi, chaque machine utilisée disposant de plusieurs processeurs doit pouvoir en tirer parti en utilisant la parallélisation du cœur de l'algorithme lui-même. Ces mêmes machines devraient également pouvoir être utilisées en parallèle sur des portions de la base, pas forcément de tailles identiques et les requêtes découpées en sous-requêtes réparties par l'ordonnanceur en fonction de leur nature, de leur taille et de la puissance des machines disponibles. La figure 2.11 présente l'architecture d'un tel système qui n'a jusqu'à aujourd'hui jamais été réalisé.

L'ordonnancement des tâches, la distribution et la replication des bases de données dépendent alors de plusieurs paramètres :

– Le nombre de séquences contenues dans la requête.
– Le nombre de séquences contenues dans la base de données.
– Le nombre de nœuds de calculs disponibles.
– L'accessibilité de la base visée pour chacun des nœuds considérés.
– Le nombre de processeurs utilisables sur chaque nœud et leur puissance de calcul.
– La mémoire disponible sur chaque nœud.
– Les capacités des disques utilisés pour accéder à la base. (capacité, disque local rapide, disque réseau etc).

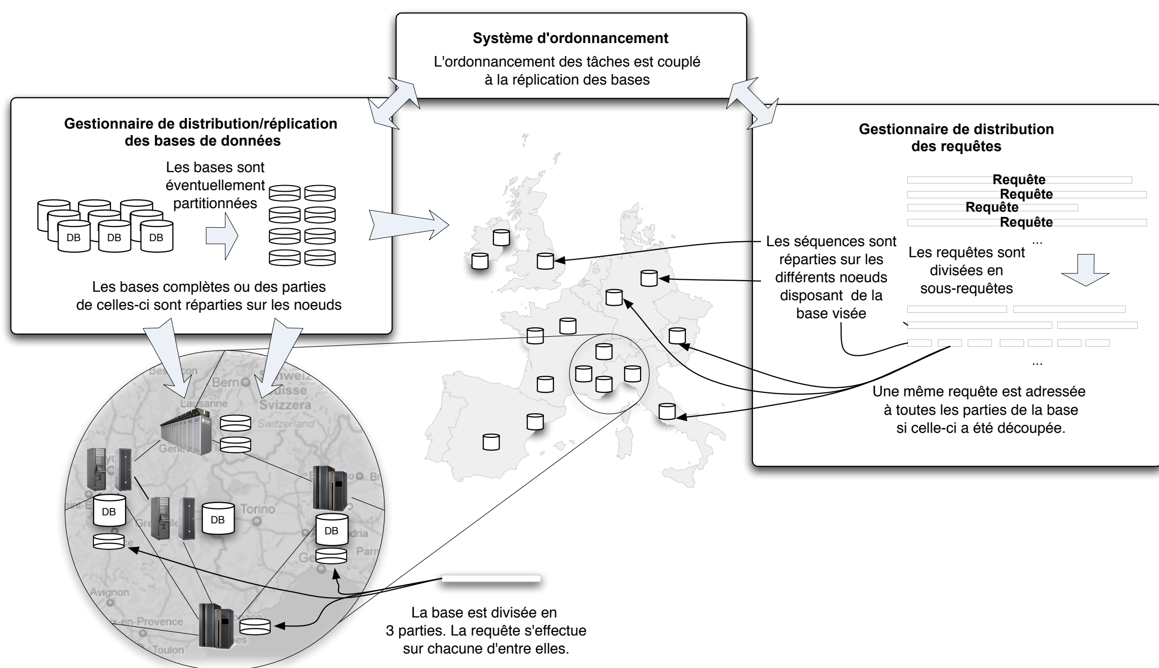

FIG. 2.11 – Architecture d'une plateforme de soumission BLAST pour la grille.

Si le nombre de séquences à l'intérieur d'une requête est petit par rapport à celui des séquences contenues dans la base, il est probable que la distribution de la base sur les différents nœuds sera plus efficace qu'une distribution des séquences de la requête. On a vu que mpiBLAST fournit d'excellents résultats pour des clusters de taille moyenne mais qu'il ne passe pas à l'échelle de la grille. Cette constatation devrait être prise en compte dans la conjonction des différents niveaux de parallélisme. Par ailleurs, si la parallélisation sur des machines SMP peut être utilisée de manière transparente en utilisant les implantations courantes de BLAST telle NCBI-BLAST [3], il est important de prendre en compte le nombre de processeurs disponibles sur un nœud pour choisir quelles requêtes ou sous-requêtes doivent y être soumises. Lorsque de nombreux utilisateurs soumettent des requêtes utilisant la même base de données, il peut être intéressant de répliquer celle-ci plus fréquemment qu'une base moins "populaire".

On notera également que les parallélisations par distribution des séquences sur plusieurs

nœuds impliquent différents besoins en terme de réplication de données :

– Partition des bases : Les différentes parties des bases doivent être répliquées sur chaque site que l'on souhaite utiliser pour la recherche d'alignements. Il est possible d'adapter la taille de chacune de ces parties à l'espace disque disponible ou à la mémoire utilisable pour améliorer les performances et optimiser l'utilisation des espaces de stockage de la grille.
– Partition des requêtes : Les bases complètes doivent être répliquées sur les différents nœuds. Cependant, à défaut de pouvoir équilibrer la charge provoquée par une recherche sur les bases complètes, il est possible d'ajuster la taille ou le nombre des requêtes soumises sur chaque nœud en fonction de la puissance processeur ou de la mémoire disponible.
– Partition des bases et des requêtes : Les bases doivent être répliquées sur les différents sites. Il est alors possible de répliquer les bases complètes ou seulement des parties de celles-ci.

Clairement, si la conjonction des différents niveaux de parallélisme possibles pour le développement d'une interface de soumission de requêtes de recherches d'alignements sur la grille peut apporter de très importants gains de performances, il s'agit d'une opération complexe ouvrant de nombreuses nouvelles problématiques.

Le tableau 2.1 présente les avantages et les inconvénients des deux méthodes de parallélisations par distribution des bases ou des requêtes.

La mise en place d'une telle plateforme nécessitera donc :
– Une gestion des réplications des bases partielles ou complètes.
– Un ordonnancement des tâches centralisé.
– Le référencement des bases et de leurs différents réplicats.
– Des possibilités d'interactions fortes entre l'ordonnanceur et le gestionnaire de données.

2.5 Problématiques générales de la gestion de données sur les grilles

Les problématiques posées par la gestion de données sur les grilles diffèrent largement suivant le type de grille, le type de données et l'usage qui en est fait. Ainsi, du simple partage de fichiers statiques qui ne nécessite qu'un accès en écriture unique et de nombreux accès en lecture jusqu'à un partage d'espace de stockage avec de multiples accès à des données très dynamiques, les contraintes des gestionnaires de données vont s'exprimer à des niveaux et des échelles très différents. On peut distinguer six contraintes principales :

– La pérennité : Les données stockées ne doivent pas être supprimées.
– La transparence d'accès : Les données doivent être accessible de manière transparente pour l'utilisateur.
– L'accessibilité : On doit pouvoir accéder aux données de n'importe quel point de la grille.
– La cohérence : Une donnée partagée doit être identique pour tous les utilisateurs.
– Les performances : L'accès aux données doit être rapide et efficace.

<table>
<tr><th>Partitionnement des bases</th><th>Partitionnement des requêtes</th></tr>
<tr><td>✔ Chaque partie de la base peut tenir en mémoire, évitant ainsi de coûteux accès disque.
✔ Les performances sont améliorées, même lorsque la requête contient peu de séquences.
✔ On peut ne répliquer que des “morceaux” de la base, évitant ainsi de copier les bases complètes sur les nœuds.
✘ Ne passe pas aisément à l'échelle de la grille. Comme nous l'avons vu avec mpiBLAST à la section 2.4.2, les gains de performances ne sont pas linéaires en fonction du nombre de nœuds utilisés. Ce comportement est à prendre en compte pour le découpage et la distribution des bases.
✘ Nécessite une correction des e-values dans les résultats obtenus pour que ceux-ci soient corrects.</td><td>✔ Un très grand nombre de séquences peut être traité simultanément.
✔ Le passage à l'échelle est assuré : Plus on dispose de ressources de calcul, plus le nombre de séquences traitées simultanément est grand.
✘ Les petits ensembles de séquences ne tire pas profit de l'ensemble des ressources de calcul disponibles.
✘ Nécessite des accès au disque lorsque la base de données est trop grande pour entrer entièrement en mémoire.
✘ Il est nécessaire de répliquer les bases complètes sur tous les nœuds de calcul.</td></tr>
<tr><th colspan="2">Partitionnement des bases de données et des requêtes</th></tr>
<tr><td colspan="2">✔ Tire profit de toutes les ressources de calcul.
✔ Passe à l'échelle de la grille.
✔ On évite les accès disque en découpant les bases de telle manière qu'elles puissent tenir en mémoire.
✘ Nécessite des stratégies d'ordonnancement et de réplication complexes.</td></tr>
</table>

TAB. 2.1 – Avantages et inconvénients des parallélisations de BLAST pour la grille.

– La sécurité : L'accès en écriture et en lecture d'une donnée ou d'un espace de stockage doit être réglementé et limité.

Chacune de ces contraintes s'exprimera de manière plus ou moins forte en fonction de l'intergiciel. Plusieurs approches différentes s'adaptent plus ou moins bien à l'usage souhaité de la grille et des données qu'elle stocke.

2.5.1 L'approche par catalogues de données/meta-données

Le Globus Toolkit, "boîte à outils" logicielle permettant la réalisation d'intergiciels de grilles, fournit des services de gestion de données pour la grille. Il est notamment utilisé comme base de développement de l'intergiciel gLite dont les services de gestion de données sont présentées à la section 2.7. Pour fournir un accès aux données répondant aux contraintes de la grille, il s'appuie sur des services de référencement de données sous forme de catalogues. Le Metadata Catalog Service [88] se charge de conserver les meta-données concernant les données stockées sur la grille comme leurs tailles, les droits d'accès à celles-ci, mais aussi leur nature, le moment où elle ont été collectées etc. Globus propose également un système d'identification et de localisation des réplicats d'une même donnée par l'intermédiaire du "Replica Location Service" [33] qui met en relation les identifiants des données avec ses différents réplicats "physique". Il prend en compte des contraintes fortes de sécurité grâce à GSI (Grid Security Infrastructure), l'infrastructure de sécurité de Globus basé sur la cryptographie à clé publique, mais ne permet qu'une gestion des données explicite laissée à la charge de l'utilisateur. Les systèmes utilisant une telle approche sont en général destinés à des grilles "fermées" de clusters administrés et suffisament fiables. Ainsi, la pérennité des données n'est pas assurée par le système mais par l'administration de la grille. Aucun dispositif de contrôle ou de maintien de la cohérence des données n'est implanté et c'est à l'utilisateur de prendre garde aux données qu'il utilise. Les performances d'accès dépendent également de la gestion effectuée par l'utilisateur.

2.5.2 L'approche des "dépôts logistiques" de données

Cette approche notamment retenue par "Internet Backplane Protocol" (IBP) [76, 15] consiste à distribuer les données sur différents "dépots" distribués sur la grille. Il s'agit d'une approche au niveau protocole, soit une approche de très bas niveau permettant de gérer très finement les transferts et le stockage des données mais qui nécessite d'être largement étendue par des développements utilisateurs pour pouvoir assurer, cohérence et persistance des données. Cependant, cette approche permet d'ordonnancer les transferts et dans un contexte de grille d'associer cet ordonnancement à celui des tâches de calcul. Ce type d'ordonnancement conjoint a été implanté dans l'intergiciel NetSolve grâce à IBP [10].

2.5.3 L'approche pair-à-pair

Les applications de partage de fichiers en pair-à-pair connaissent une forte popularité auprès du grand public depuis le début des années 2000. Des logiciels comme Napster, Gnutella, Kazaa, Edonkey et plus récemment BitTorrent ont probablement contribué à la forte croissance des connexions qu'a connu Internet ces dernières années. Ces applications à usage ludique et popularisées par une utilisation souvent illicite sont les prémices de la gestion de

données en pair-à-pair et à grande échelle. Ne permettant qu'une gestion des données trop sommaire (ajout et partage d'une donnée) pour les exigences de la plupart des intergiciels de grille, ils sont pour l'essentiel resté dans la sphère grand public. Cependant, plusieurs systèmes de gestion de données pour la grille ont repris l'approche pair-à-pair utilisée par ceux-ci. Le service JuxMem (**Jux**taposed **Mem**ory) [9] propose par exemple un système de partage de données transparent, gérant la cohérence, la persistance et tolérant aux pannes fondé sur une approche pair-à-pair (voir la section 2.8). OceanStore [64], vise quant à lui, à offrir un système de partage de données accessibles en lecture et en écriture à très large échelle tout en maintenant une disponibilité et une pérennité forte des données.

2.5.4 L'approche par système de fichiers

D'autres approches ont consisté à développer des systèmes de fichiers pour la grille. GFarm [93] et GridNFS [52] sont deux exemples de tels systèmes de fichiers proposant d'étendre aux plateformes de grande échelle les services offert par les systèmes de fichiers réseau comme NFS [86]. GFarm répartis les données sur les différents nœuds de calcul et est surtout dédié à la gestion de données non modifiables. Il gère les réplications de manière transparente lors de l'accès aux données. GridNFS est quant à lui basé sur le protocole NFS [85], étendant ses capacités à l'échelle de la grille. Dans GridNFS, la sécurité est gérée par Globus ce qui signifie qu'il offre les fonctionnalités d'authentification, d'autorisation, de gestion des organisations virtuelles et de contrôle d'accès aux ressources offertes par ce système.

2.6 Les problématiques spécifiques à la gestion des données biologiques

Les bases de données biologiques sont en général d'une taille comprise entre quelques dizaines de méga-octets et une centaine de giga-octets. Si une telle quantité de données peut demander beaucoup de temps de calcul pour être analysée ou comparée, elle ne représente pas une masse de donnée suffisamment importante pour nécessiter des systèmes complexes de partage d'espace sur la grille. En effet, si la somme des tailles de toutes les bases utiles à un laboratoire peut dépasser la capacité de stockage d'un nœud de la grille, chaque base peut en général être individuellement placée sur celui-ci. De plus, les mises à jour de ces bases, si elles sont indispensables et régulières sont de nature ponctuelle. Il n'est en effet pas question dans leur utilisation d'écriture continue et concurrente sur celles-ci. On constate par ailleurs que l'utilisation des bases consiste en général en la soumission d'un grand nombre de tâches courtes. L'application BLAST décrite dans la section 2.3 est à cet égard caractéristique des applications bio-informatiques. Une gestion efficace des données biologiques consistera donc en trois points principaux :

– La pérennité des données : Les bases peuvent être distribuées sur l'ensemble de la grille, on doit néanmoins s'assurer de la présence d'au moins un réplicat de chacune d'entre elles.
– La disponibilité des données : Pour assurer les meilleures performances possibles aux tâches bio-informatiques soumises à la grille, les bases de données doivent être répliquées afin de permettre la parallélisation des tâches élémentaires. Le temps d'une

tâche individuelle étant assez court, il est préférable que la donnée soit présente sur le nœud de calcul au moment de son lancement.
– La cohérence des données : Les mises à jour des données ne sont pas continues cependant il est indispensable de s'assurer que tous les réplicats d'une même donnée sont identiques.

Ces différents points sont à mettre en relation avec les besoins d'une plateforme bio-informatique analysés dans la section 2.4.4. L'accès aux données, les besoins en terme de réplication explicite, les interactions nécessaires entre le gestionnaire de données et l'ordonnanceur centralisé nous ont conduit à retenir l'approche par catalogue de données.

Nous avons retenus deux intergiciels de grille offrant ce type de gestion de données et permettant le développement d'applications bio-informatiques pour la grille : L'intergiciel gLite utilisé par la grille EGEE et l'intergiciel Grid-RPC DIET. Les sections suivantes présentent ces intergiciels et les mécanismes de gestion de données qu'ils offrent.

2.7 L'intergiciel de la grille EGEE : gLite

La grille EGEE (Enabling Grid for E-sciencE) [44] fédère des ressources réparties sur 250 sites dans 48 pays. Deux-cents Organisations Virtuelles regroupent l'ensemble de ses utilisateurs. Plus de 68000 processeurs sont mis à disposition des chercheurs et des centaines de téra-octets sont destinés à stocker les données des expériences scientifiques. Elle utilise l'intergiciel de grille gLite basé sur l'intergiciel LCG [66] conçu par le CERN pour permettre aux chercheurs d'exploiter les données qui seront produites par le Large Hadron Collider [11]. L'architecture de LCG qui ressemble aux architectures des systèmes de batch est présentée dans la figure 2.12. Sa sécurité est basée sur une infrastructure à clé publique et un service centralisé (le *Resource Broker*) assure la soumission des tâches sur les différents nœuds de la grille. Pour ce faire, LCG fait appel à un système d'information qui recense les caractéristiques des différents nœuds (disponibilité des processeurs, système d'exploitation, mémoire etc.)

gLite est un intergiciel "orienté service" qui évolue peu à peu vers une architecture basée sur les web-services. Ses composants principaux sont les suivants :
– L'interface utilisateur (User Interface ou UI) : C'est le point d'entrée à la grille des utilisateurs. Pour se connecter, un utilisateur doit disposer d'un certificat de type X.509 qui l'identifie et lui attribue un rôle, des droits et une appartenance à une ou plusieurs Organisation Virtuelle. L'accès s'effectue en ligne de commande ou par l'intermédiaire d'une API spécifique.
– Le gestionnaire de ressources (Resource Broker) : C'est le "chef d'orchestre" de la grille. Il attribue les ressources en fonction de leur disponibilité et des besoins des utilisateurs.
– Le système d'information (Information System) : Il recense le status et les différentes caractéristiques des ressources de la grille.
– Les éléments de calcul (Computing Elements ou CEs) : Il s'agit d'une abstraction des différentes queues de batchs auxquelles seront soumises les tâches des utilisateurs.
– Les éléments de stockage (Storage Elements ou SEs) : Ce sont les ressources de stockage disponibles sur la grille.

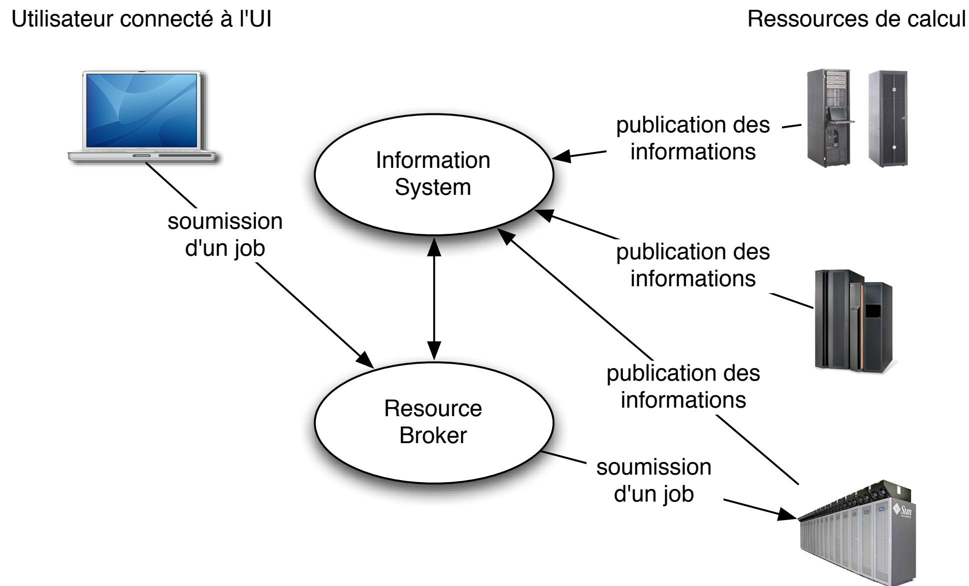

FIG. 2.12 – Architecture de LCG.

La figure 2.13 présente l'organisation et les interactions des différents composants de gLite.

Les sections suivantes présentent les différents services mis en œuvre dans gLite pour, la gestion des tâches, la gestion de données et son système d'information.

2.7.1 Le Workload Management System (WMS)

La soumission des tâches avec gLite repose sur le "Workload Management System", qui gère les tâches de leur soumission jusqu'à la fin de leur exécution et la transmission des données produites. Il interagit avec les services de gestion de données afin de permettre aux utilisateurs de définir simplement les transferts et les accès aux données, nécessaires à la bonne exécution de leurs tâches décrites au format JSDL (Job Submission Description Language) [8]. En effet, dans cette description, l'utilisateur peut demander à ce qu'une donnée stockée localement soit transmise avec la tâche, qu'une donnée présente sur la grille soit transférée dans l'espace d'exécution de celle-ci avant son lancement et que les données produites soient renvoyées, soit directement sur la grille, soit en attente de téléchargement vers l'interface d'accès à la grille de l'utilisateur (User Interface - UI). De plus, le JDL permet d'effectuer des redirections des entrées/sorties du programme lancé vers des fichiers récupérés ou stockés sur la grille avant et après l'exécution.

Le Workload Management System gère une file d'attente des tâches soumise à la disponibilité des ressources demandées par l'utilisateur. Pour cela, le composant "Match Maker" interroge un système d'information (l'Information Supermarket) qui est mis à jour à la fois de manière passive (en recevant les informations depuis les ressources) et pro-active (en allant chercher l'information directement sur les ressources). Lorsqu'un ressource de calcul conforme aux exigences définies par l'utilisateur est disponible, il soumet la tâche au CE correspondant. Le Workload Management System interagit avec les autres services de gLite pour contrôler les droits accordés aux utilisateurs, pour effectuer la gestion de

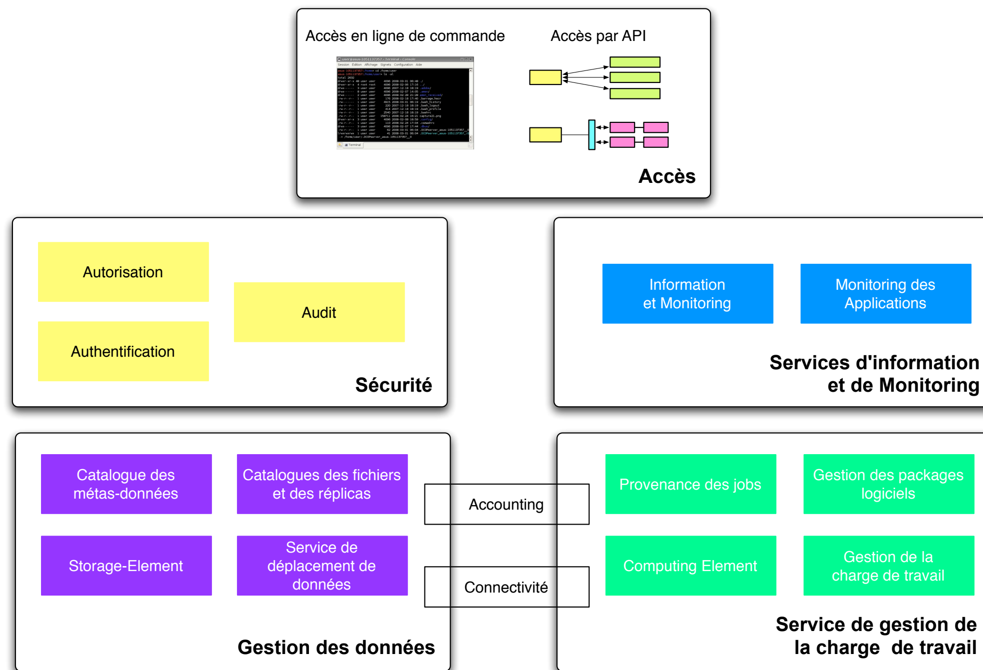

FIG. 2.13 – Organisation globale des composants de gLite

données précisée dans la description de la tâche et pour monitorer et conserver une trace des événements produits par l'exécution de celle-ci. La figure 2.14 présente l'organisation générale de ce service de gLite.

2.7.2 R-GMA : Le système d'information de gLite

Le "Relational Grid Monitoring Architecture" (R-GMA) fournit et maintient à jour les informations sur les ressources de la grille. Ces informations peuvent être aussi bien à propos du matériel disponible que des logiciels installés ou encore des utilisateurs des différentes Organisations Virtuelles.

R-GMA fournit une description des ressources sous forme de modèle relationnel. Il s'agit d'une base de données "virtuelle". Pour fournir les données, il fait appel à des "registres" qui sont constitués de listes de "producteurs de données" qui se sont enregistrés. Ainsi les informations peuvent être obtenues, soit en interrogeant le producteur, soit en "l'écoutant" afin d'obtenir de nouvelles données.

2.7.3 La gestion de données dans gLite

En ce qui concerne la gestion des données, gLite a été conçu afin de permettre la gestion de ressources de stockage hétérogènes (disques durs, sauvegardes sur bandes, etc.) accessibles par l'intermédiaires de différents protocoles. Il permet d'effectuer des replications et distributions des données entre les différents nœuds et des déplacement de données entre différents sites.

Pour réaliser ces tâches, gLite repose sur plusieurs systèmes distincts dont les interactions sont présentées dans la figure 2.15

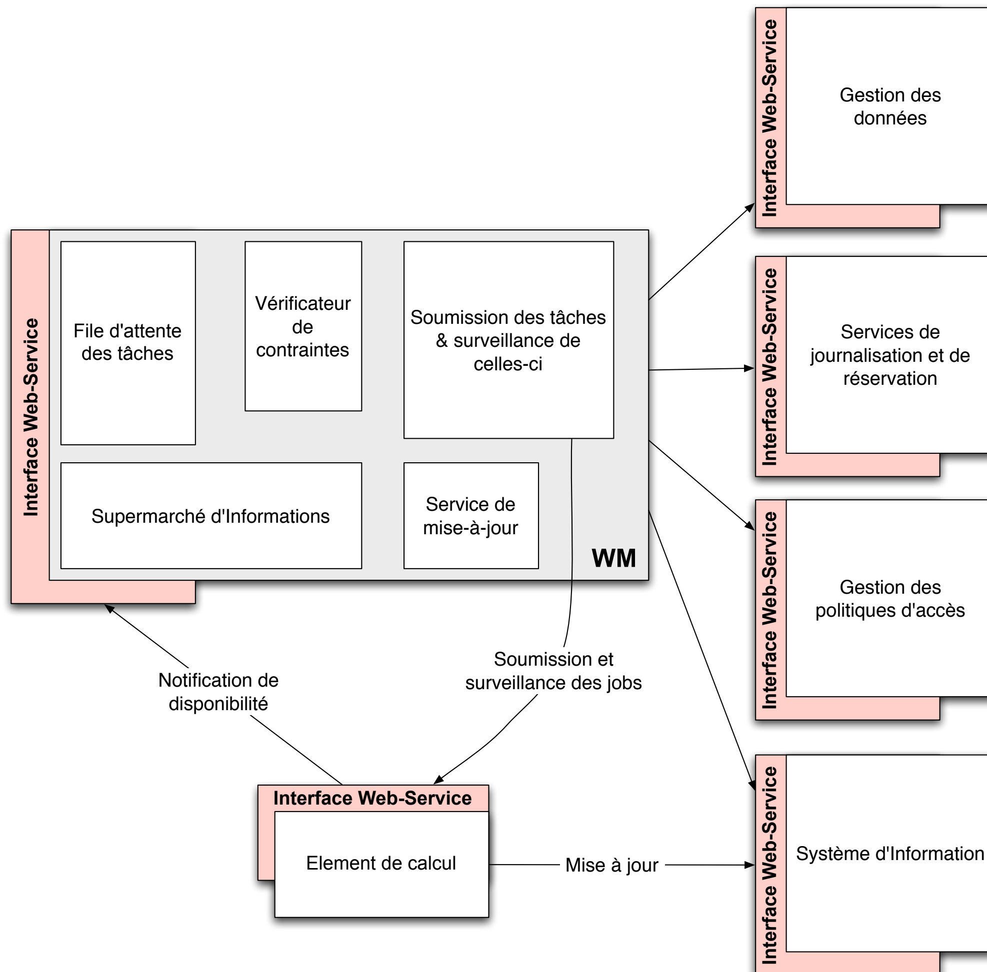

FIG. 2.14 – Le Workload Management System

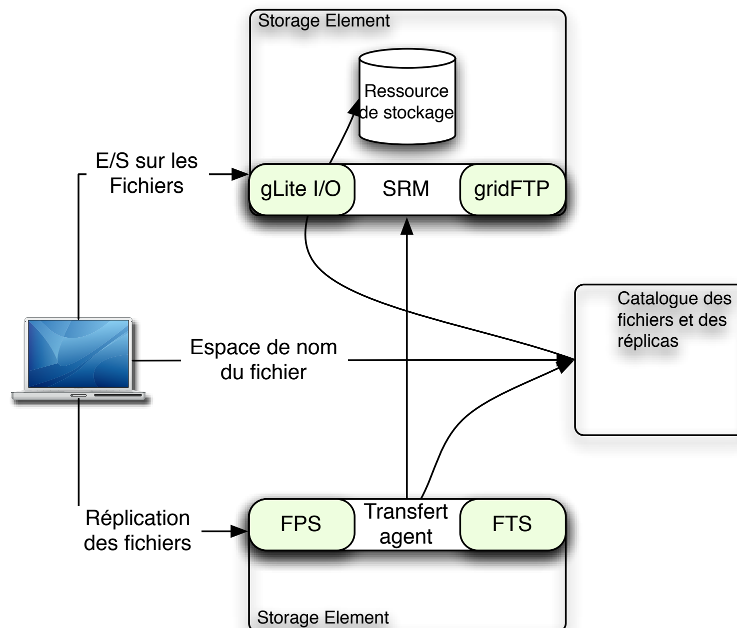

FIG. 2.15 – Interactions entre les différents services de gestion de données dans gLite

Le Storage Resource Manager (SRM) fournit une interface d'accès unique aux ressources physiques quel que soit leur nature. Ainsi, SRM donne un accès transparents aux données, permet d'effectuer des réservations d'espace de stockage, retourne des notifications quant aux status des différents fichiers stockés et gère la durée de vie des données. Il interagit par ailleurs avec différents services de la grille comme gLite I/O, un service d'accès direct aux données sur la grille semblable aux accès POSIX implantés dans la plupart des langages de programmation.

Les catalogues de fichiers et de replicats utilisés pour trouver les données et leurs replicats à partir d'un nom lisible ou d'un identifiant unique. Les catalogues permettent de mettre en relation une donnée physiquement stockée sur la grille et un nom de fichier utilisable comme sur les systèmes de fichiers traditionnels *NIX (/chemin/.../nomdefichier). Ils mettent pour cela en relation un nom utilisable facilement par l'utilisateur, le Logical File Name (LFN) avec un identifiant unique, le Global Unique Identifier (GUID) qui est lui-même associé à une ou plusieurs Site URL (SURL ou Physical File Name - PFN).

Le service de transfert des fichiers (FTS) gère quant à lui les mouvements de données sur la grille. Il s'occupe de la gestion du réseau et du stockage, de la source d'une donnée jusqu'à sa destination en définissant un concept de canaux établis entre deux Storage Elements (SE). Il optimise l'utilisation de la bande passante sur les différents canaux ainsi créés et permet de définir des priorités entre les transferts à effectuer. *Le service de placement des fichiers* (FPS) est un service de transfert qui interagit avec les catalogues pour retrouver une donnée à partir

de son identifiant et enregistrer les nouvelles copies d'une donnée dans ceux-ci.

2.8 L'approche Grid-RPC proposée par DIET

Data Interactive Engineering Toolbox (DIET) [29, 5] est un intergiciel de grille reposant sur le paradigme d'appels à distance Grid-RPC. Il propose une interface de création de programmes client/serveur adapté à la grille en rendant transparent la complexité de celle-ci. Il permet toutefois aux utilisateurs de définir des politiques d'ordonnancement avancées grâce à un système de plugins d'ordonnancement, au niveau du serveur, ou même au niveau de l'intergiciel depuis sa dernière version.

2.8.1 Architecture de DIET

Une architecture DIET est composée de trois éléments principaux :
- Les clients : Ce sont des applications qui font l'interface entre un utilisateur désirant effectuer un appel de service sur la grille et l'intergiciel.
- Les "Server Daemons"(SeD) : Ce sont les serveurs qui fournissent les services à la grille. Un SeD encapsule un serveur de calcul. Il peut être par exemple placé sur le point d'entrée d'une machine parallèle (super-calculateur, cluster etc.) ou simplement sur une machine individuelle du réseau. Un SeD stocke la liste des services qu'il offre aux utilisateurs, mais aussi des informations sur sa charge et ses capacité matérielles (nombre de processeurs, charges des processeurs, quantité de mémoire installée etc.)
- Les agents : Ce sont les éléments de DIET qui effectuent la sélection du ou des serveurs utilisés pour l'exécution d'un service. Ils sont interconnectés suivant une topologie d'arbre dont le sommet est un "agent maître" (Master Agent ou MA) qui sera seul contacté par les applications clientes.

La figure 2.16 présente l'architecture globale d'un déploiement de DIET. Il est à noter que plusieurs architectures DIET peuvent être reliées par des liaisons pair-à-pair réalisées entre plusieurs MA. Cette architecture permet de distribuer la charge des différentes tâches de l'intergiciel sur plusieurs machines, assurant ainsi le passage à l'échelle de la grille des applications développées avec DIET.

La soumission d'une requête dans DIET se déroule en plusieurs étapes :

L'étape "requête" : Le client soumet une requête pour un service au Master Agent. Un service dans DIET est identifié par une chaîne de caractère (par exemple "addition"), mais aussi par le type et le nombre de ses paramètre (par exemple ["int", "int", "int"]). Ainsi, plusieurs services de même nom, mais prenant des paramètres différents peuvent cohabiter dans une même hiérarchie. Par exemple, les services *("addition", integer, integer, integer)* et *("addition", float, float, float)* permettront d'effectuer l'addition de deux entiers ou de deux flottants, sans qu'il y ait confusion au moment de la requête.

L'étape de recherche du service : Le Master Agent fait suivre la requête à l'ensemble des Local Agents et/ou SeDs dont il a la charge. Lorsqu'un Local Agent reçoit une requête, il interroge à son tour les SeDs et agents auxquels il est connecté. Lorsqu'un SeD reçoit une requête, deux cas de figure se présentent :

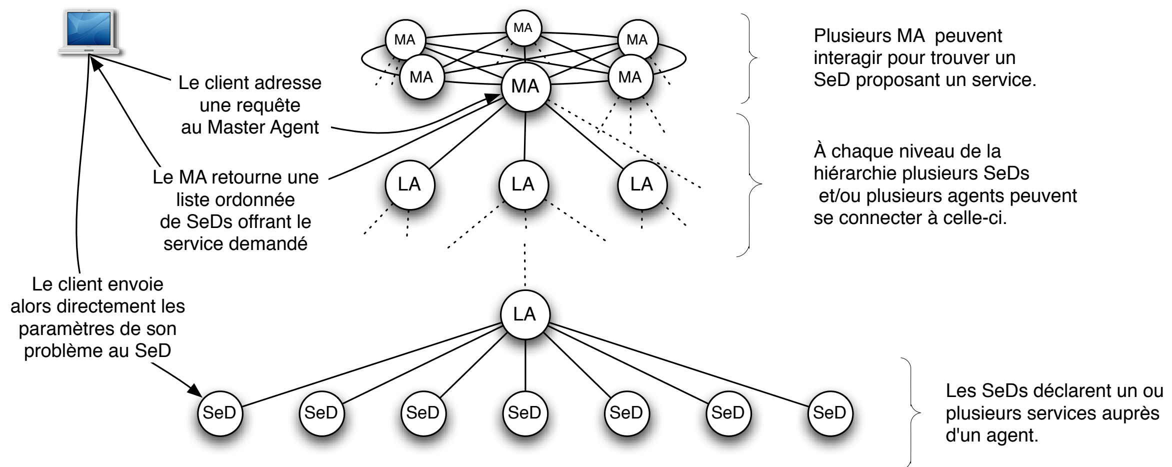

FIG. 2.16 – Architecture de DIET

– Le SeD ne propose pas le service demandé : Il retourne une réponse négative à l'agent qui l'a interrogé.
– Le SeD propose le service demandé : Il retourne une réponse positive accompagnée d'un certain nombre de caractéristiques qui le concerne afin de permettre à l'agent d'effectuer la sélection du "meilleur" serveur pouvant traiter la requête.

L'étape d'ordonnancement : Les agents reçoivent les liste des réponses positives à la requête obtenues depuis le niveau inférieur de la hiérarchie. Il procède alors à la sélection des "meilleurs" serveurs parmis tous ceux présent dans une de ces listes. Ils renvoient alors cette nouvelle sélection au niveau supérieur de la hiérarchie. Cette étape est détaillée dans la section 2.8.2 consacrée aux différents modes d'ordonnancement proposés par DIET.

L'étape de résolution : Lorsque le Master Agent reçoit les listes des serveurs sélectionnés pour la résolution du problème, il effectue une dernière sélection parmi elles et retourne la liste obtenue au client qui a soumis la requête. Le client peut alors s'adresser directement au meilleur SeD sélectionné pour lui soumettre les paramètres du problème à résoudre. Le SeD procède alors à l'exécution du service en utilisant ces paramètres et transmet le résultat au client. Cette dernière étape peut être réalisée de manière asynchrone. Ainsi, pendant la durée d'exécution de la tâche soumise au SeD, le client peut contacter d'autres serveurs pour la résolution d'autres problèmes ou du même problème avec d'autres paramètres.

2.8.2 L'ordonnancement dans DIET

Dans DIET, ce sont les agents qui gèrent les tâches d'ordonnancement. Plusieurs étapes se déroulent depuis la soumission d'une requête jusqu'à l'exécution de la tâche. La figure 2.17 illustre le fonctionnement de ces différentes étapes.

– Le client soumet une requête pour un service auprès du Master Agent. (étape 1)
– Le Master Agent fait suivre la requête à l'ensemble des Local Agents auxquels il est relié. Chacun d'entre eux, procède de la même manière avec les agents auxquels il est relié jusqu'à atteindre un Server Daemon. (étape 2)

– Les Servers Daemons qui offrent le service demandé par la requête collectent alors des informations sur leur état actuel (charge processeur, mémoire etc.) en fonction du type d'ordonnancement choisi. Ils retournent alors un vecteur de valeurs à l'agent appelant. Suivant les services choisis lors de la compilation de DIET, des estimations de performances peuvent également être transmises par l'intermédiaire de ce vecteur. (étape 3)
– En fonction du type d'ordonnancement défini par le SeD ou par l'agent, les réponses obtenus sont aggrégées en une liste de réponse ordonnée et de taille limitée. La liste ainsi construite ne contient alors que les "meilleurs" serveurs choisis en fonction de la métrique d'ordonnancement choisie. Cette liste est alors renvoyée à l'agent de niveau supérieur dans la hierarchie DIET qui procédera à la même sélection des serveurs. (étape 4)
– Lorsque la liste des serveurs atteint le Master Agent, celui-ci procède à la dernière aggrégation/sélection des serveurs et retourne celle-ci au client ayant soumis la requête. (étape 5)
– Le client sélectionne alors un ou plusieurs serveurs parmi la liste obtenue et soumet directement les tâches à ceux-ci. (étape 6)

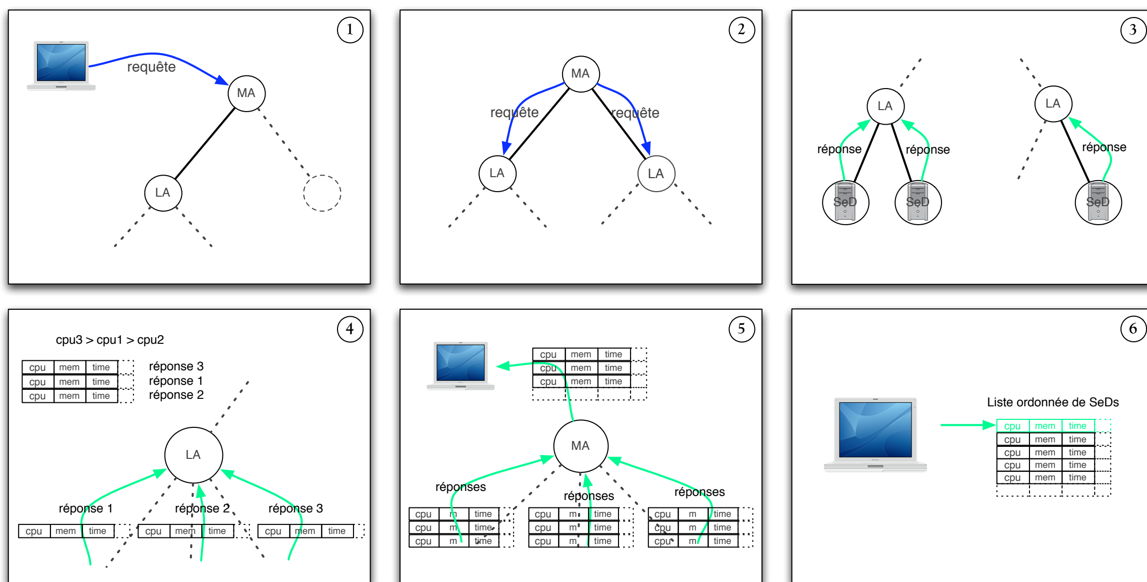

FIG. 2.17 – Étapes de soumission d'une tâche dans DIET

Le mécanisme d'ordonnancement ainsi effectué diffère suivant le type d'ordonnancement choisi. En effet, DIET dispose de politiques d'ordonnancement par défaut utilisées lorsque le développeur du SeD n'a pas défini de politique spécifique à son service et que les agents n'ont pas été configurés pour utiliser un module d'ordonnancement externe. Dans le deuxième cas, le SeD adresse aux agents des priorités quant au classement des réponses. Par exemple, on peut choisir de classer celles-ci en fonction de la puissance du processeur par ordre décroissant de valeur puis en fonction de la charge actuelle du système par ordre décroissant de valeur. Pour ce faire, DIET propose par l'intermédiaire de l'extension CoRI (Collectors of Resource Information) [27] toute une liste de caractéristiques utilisables

comme métrique pour l'ordonnancement :
- Le pourcentage d'occupation processeur.
- La quantité de mémoire disponible.
- Le nombre de processeurs installés sur la machine.
- La fréquence de fonctionnement de chacun des processeurs de la machine.
- La quantité de mémoire installée sur la machine.
- Le pourcentage moyen d'occupation des processeurs.
- La "puissance" de chacun des processeurs exprimée en BogoMips.
- La taille de la mémoire cache de chaque processeur.
- La taille totale du disque de la machine.
- La quantité d'espace disque libre.
- Les temps d'accès au disque en lecture et en écriture.

Enfin, DIET offre la possibilité d'écrire des modules externes redéfinissant complètement la méthode d'aggregation des agents. Ces modules, seront alors chargés dynamiquement à l'exécution de l'agent si le SeD en fait la demande. Ce dernier mécanisme permet alors de classer les réponses en fonction d'autres paramètres que ceux fournis par les SeDs (nombre d'exécutions sur un cluster, paramètres externes ne pouvant pas être obtenus au niveau du SeD etc). L'organisation des différents ordonnanceurs dans le code de DIET est donnée dans la figure 2.18.

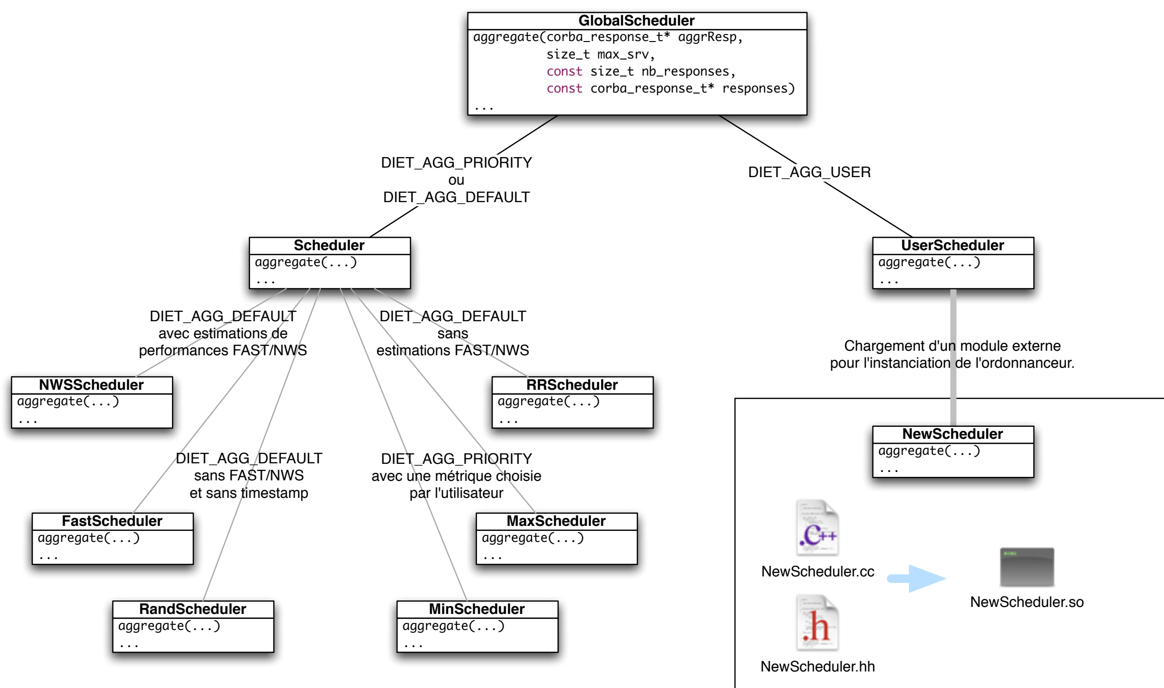

FIG. 2.18 – Organisation des classes d'ordonnancement dans DIET.

Par ailleurs, ce mécanisme permet de définir des politiques d'ordonnancement différentes pour chaque agent. Il sera ainsi possible de choisir un ordonnancement Round-Robin pour chaque cluster géré par un agent et de choisir une autre politique quant au choix

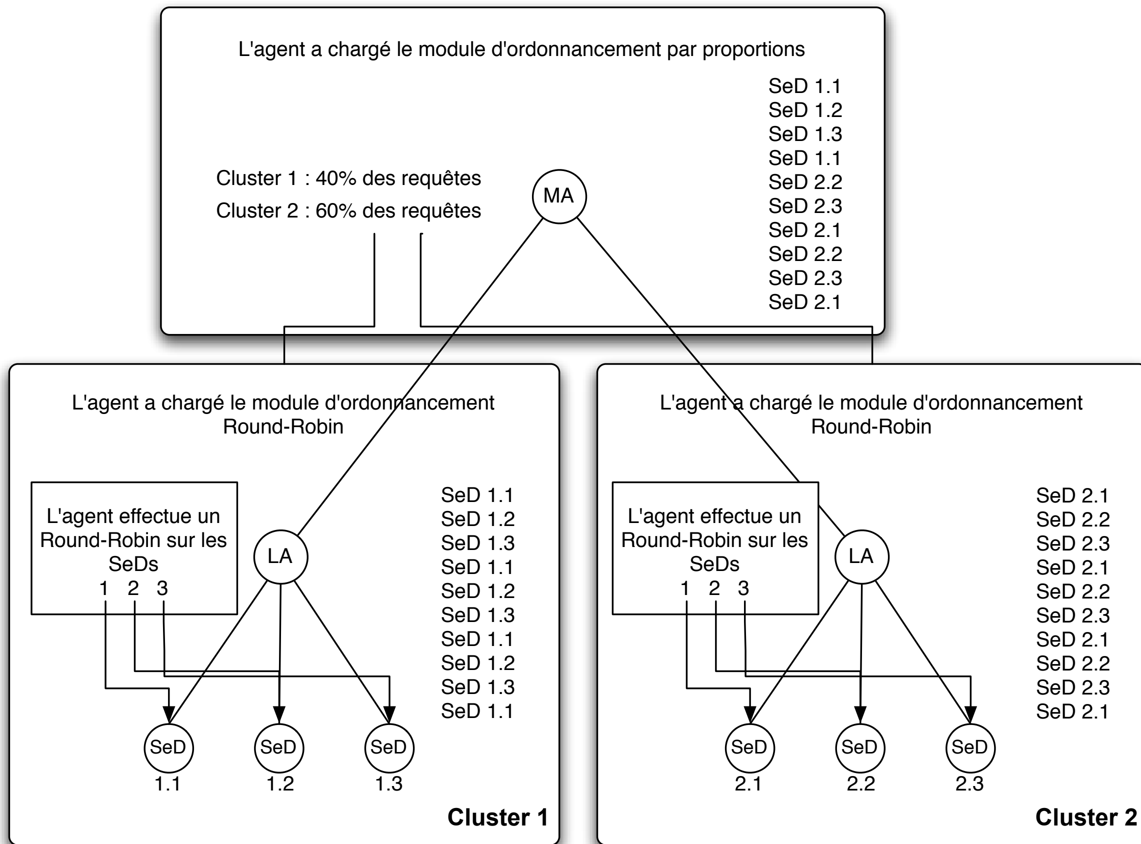

FIG. 2.19 – Politiques d'ordonnancement au niveau des agents.

du cluster au niveau du master agent. La figure 2.19 présente un tel cas de figure.

2.9 La gestion des données dans DIET

DIET propose plusieurs systèmes de gestion de données. Le plus simple et premier implanté dans DIET est le Data Tree Manager [35]. DIET peut également faire appel à JuxMem [9] pour gérer les données qu'il manipule et plus récemment à DAGDA un nouveau gestionnaire de données, développé durant ma thèse, qui permet la réplication explicite ou implicite de celles-ci. Cette section présente les gestionnaires de données DTM et JuxMem. Le chapitre 4 étant entièrement consacré à DAGDA qui a été développé dans le cadre de mon travail de thèse.

2.9.1 Data Tree Manager - DTM

DTM étant une partie intégrante de DIET, sa structure suit la hiérarchie de déploiement de ce dernier. DTM est ainsi une organisation hiérarchique de deux types de composants :

– Les “Data Managers” : Ce sont les composants qui, placés au niveau des SeDs, conservent les données et gèrent les transferts entre les différents serveurs.
– Les “Location Managers” : Au niveau des agents, ils permettent de localiser les données dans l’ensemble des Data Managers à partir de leurs identifiants.

La figure 2.20 présente cette hiérarchie des composants de DTM.

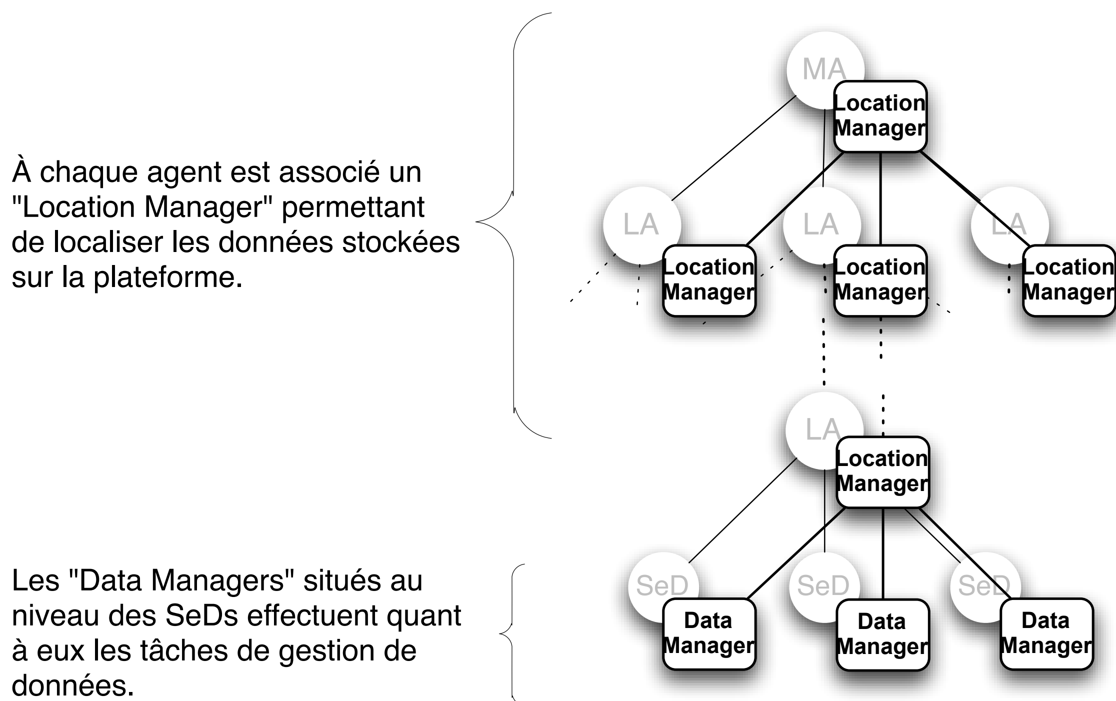

FIG. 2.20 – Organisation de DTM au sein de DIET.

Il est fréquent qu’une même donnée puisse être utilisée par plusieurs services ou qu’une donnée de sortie d’un premier service soit utilisée en entrée d’un second. Maintenir une certaine durée de vie aux données est une approche qui permet d’éviter des transferts inutiles. Cette durée de vie peut être définie en fixant un mode de persistance aux données. On distinguera plusieurs modes possibles :

– Les données persistantes : Une telle donnée utilisée ou produite par un service est maintenue là où elle est permettant de la réutiliser facilement sur place où à proximité. Ce mode de persistance est particulièrement adapté aux données intermédiaires ou aux données de référence souvent utilisée. (voir la figure 2.21)
– Les données persistantes avec retour : Une telle donnée utilisée ou produite par un service est maintenue là où elle est, mais est aussi retournée à l’utilisateur du service. Ce mode est adapté aux résultats de calculs intermédiaires qui ont un intérêt propre. (voir la figure 2.22)
– Les données dites “sticky” : Il s’agit des données persistantes dont on ne souhaite pas qu’elles soient effacées de là où on les a stockées ou produites. Ce mode est adapté aux données dont on sait qu’elles seront utilisées souvent sur les mêmes serveurs. (voir la figure 2.23)

– Les données dites “sticky” avec retour : Ce sont les données “sticky” qui ont un intérêt en elles-mêmes pour l'utilisateur.
– Les données volatiles : Ces données ne sont pas conservées après l'exécution du service. Les données de sorties sont transmises au client du service puis effacées avec les autres paramètres.

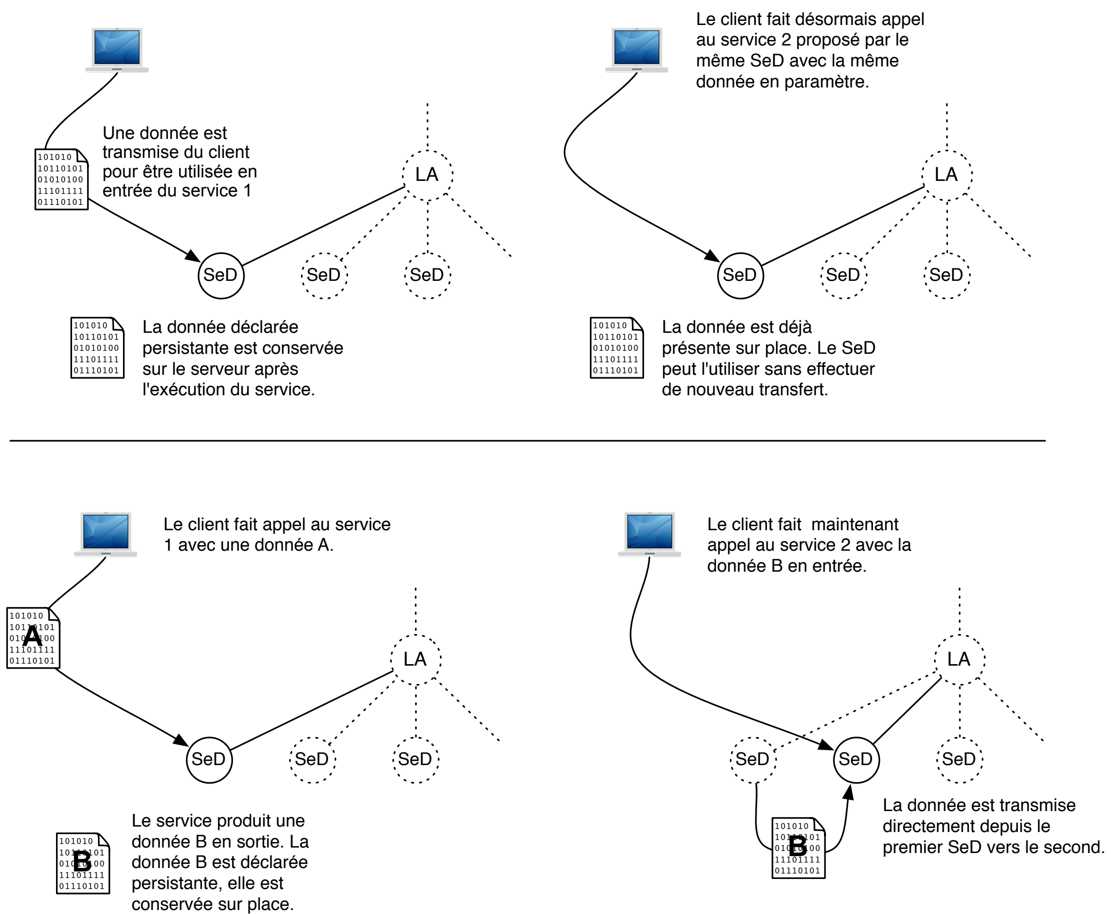

FIG. 2.21 – Deux cas d'utilisation de données persistantes.

Lorsqu'un utilisateur soumet une requête à DIET, il transmet un profil du problème à résoudre. À partir de ce profil, DIET choisira un ou plusieurs SeDs offrant le service demandé. Le client envoie alors les données des paramètres du problème au serveur qui se charge de l'exécution du service. DIET permet aux utilisateurs de définir dans le profil soumis, un mode de persistance des données, qu'elles soient des paramètres d'entrée ou de sortie du problème.

Trois modes de persistance sont gérés par DTM :

– Le mode `DIET_VOLATILE` : La donnée n'est pas conservée par les SeDs à la fin de l'exécution du service.
– Le mode `DIET_PERSISTENT` : Une donnée d'entrée sera conservée telle qu'elle a été

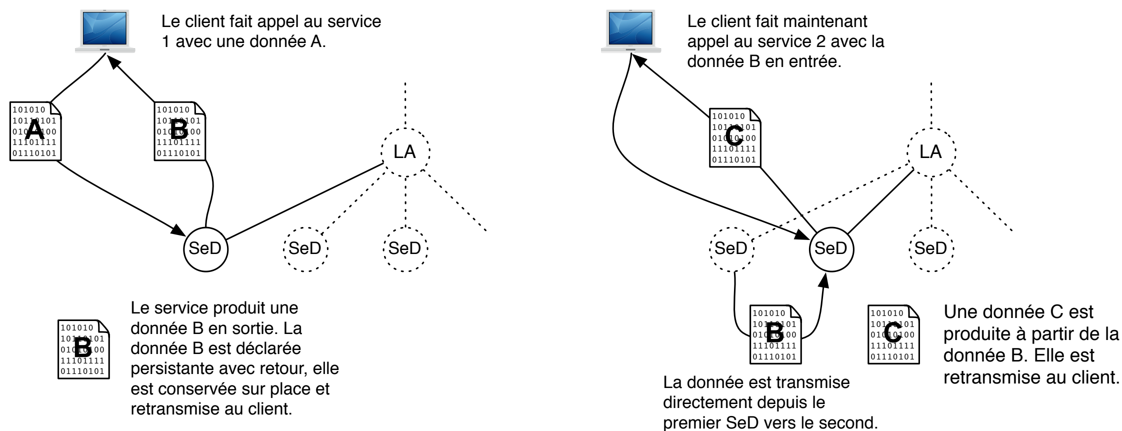

FIG. 2.22 – Cas d'utilisation d'une donnée persistante avec retour.

reçue sur le SeD choisi pour la résolution du problème. Une donnée de sortie sera également conservée sur le SeD mais aussi retransmise au client après l'exécution.
– Le mode `DIET_STICKY` : Le comportement est similaire à une donnée `DIET_PERSISTENT`, mais les données ainsi stockées ne sont plus déplacées des SeDs qui en disposent. Lorsqu'un autre SeD demande à accéder à la donnée, celle-ci lui est transmise et les Location Managers “oublient” leur localisation sur le SeD de départ. Cependant lorsqu'un service situé sur le même SeD demande à utiliser la donnée, aucun transfert n'est effectué et celui-ci utilise la donnée originellement déposée.

Plusieurs problèmes peuvent se poser lorsqu'on utilise les modes de persistance proposés par DTM. En effet, une donnée utilisant le mode `DIET_PERSISTENT` risque d'être constamment déplacée d'un nœud à l'autre si elle est souvent utilisée. A contrario, une donnée utilisant le mode `DIET_STICKY`, pourra être réutilisée sans transfert par le même serveur, mais n'est plus obligatoirement cohérente sur la grille si l'utilisateur n'y prend pas garde. Plusieurs versions d'une même donnée peuvent alors cohabiter dans la hiérarchie sans aucun moyen de savoir quelle est la plus récente. De plus, les Location Managers ne conservent de trace que de la dernière version copiée de la donnée. Ainsi, les transferts suivant ne pourront utiliser que la donnée la plus récemment placée sur un SeD, même si elle est disponible sur un autre SeD plus “proche”, au sens de la proximité réseau. Il est indispensable, lorsqu'on utilise DTM, de prendre garde aux spécificités des modes de persistance qu'il propose. Ces modes de persistance permettant toutefois d'optimiser largement l'utilisation des données sur la grille. Ses principales limitations reposent sur l'impossibilité de gérer la réplication cohérente des données. La figure 2.24 présente une situation où le fonctionnement de DTM n'est pas adapté à une utilisation optimale des ressources de la grille.

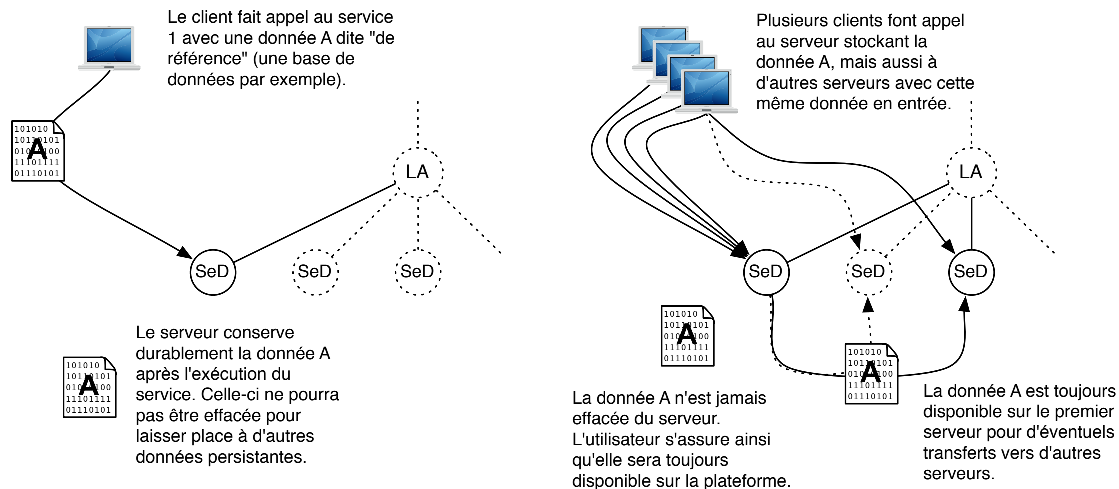

FIG. 2.23 – Cas d'utilisation d'une donnée "sticky".

Il est à noter que lorsque le profil du problème ne définit pas de données persistantes, celles-ci ne sont pas gérées par DTM et sont simplement transmises à l'intérieur du profil au moment de la soumission. Le client doit alors disposer de suffisamment de mémoire pour charger l'intégralité des données envoyées. Ainsi, la transmission de plusieurs gros fichiers dans un même profil nécessitera au mieux l'utilisation de mémoire virtuelle ou au pire provoquera une erreur du client ou du SeD concerné. Outre ces problèmes de mémoire, l'utilisation de DTM pour les transferts de données se heurte aux limitations définies par l'ORB CORBA utilisé. Ainsi, si le total des tailles de données à transmettre dépasse la taille de message autorisée par l'ORB, une erreur se produit et les données ne sont jamais transmises. Il faut alors adapter la configuration de celui-ci à la taille des données transmises par DTM. La configuration de l'ORB devant alors être effectuée en fonction de l'instance du problème soumis.

Par ailleurs, DTM ne permet pas de limiter l'espace de stockage alloué à DIET. Si le nœud ne dispose plus de suffisamment d'espace, une erreur se produit et les nœuds concernés ne sont plus utilisables.

Pour plus de détails quant au gestionnaire DTM, le lecteur peut se référer au manuscrit de thèse de Bruno Del-Fabbro [36].

Dans le cas de la gestion des bases de données biologiques nécessitant de gérer plusieurs réplicas des bases et la mise à jour de celles-ci, DTM s'avère inadapté pour la gestion des données. C'est ce constat qui a conduit à la réalisation de DAGDA, un nouveau gestionnaire de données pour DIET, présenté dans le chapitre 4.

2.9.2 La gestion des données avec JuxMem

JuxMem (**Jux**taposed **Mem**ory est un service de gestion de données pour la grille inspiré des Mémoires Virtuelles Partagées (MVP) et construit sur une approche pair-à-pair. L'approche de JuxMem vise à fournir des services des systèmes de mémoires virtuelles partagées en matière de transparence d'accès et de cohérence de données sur des plateformes à

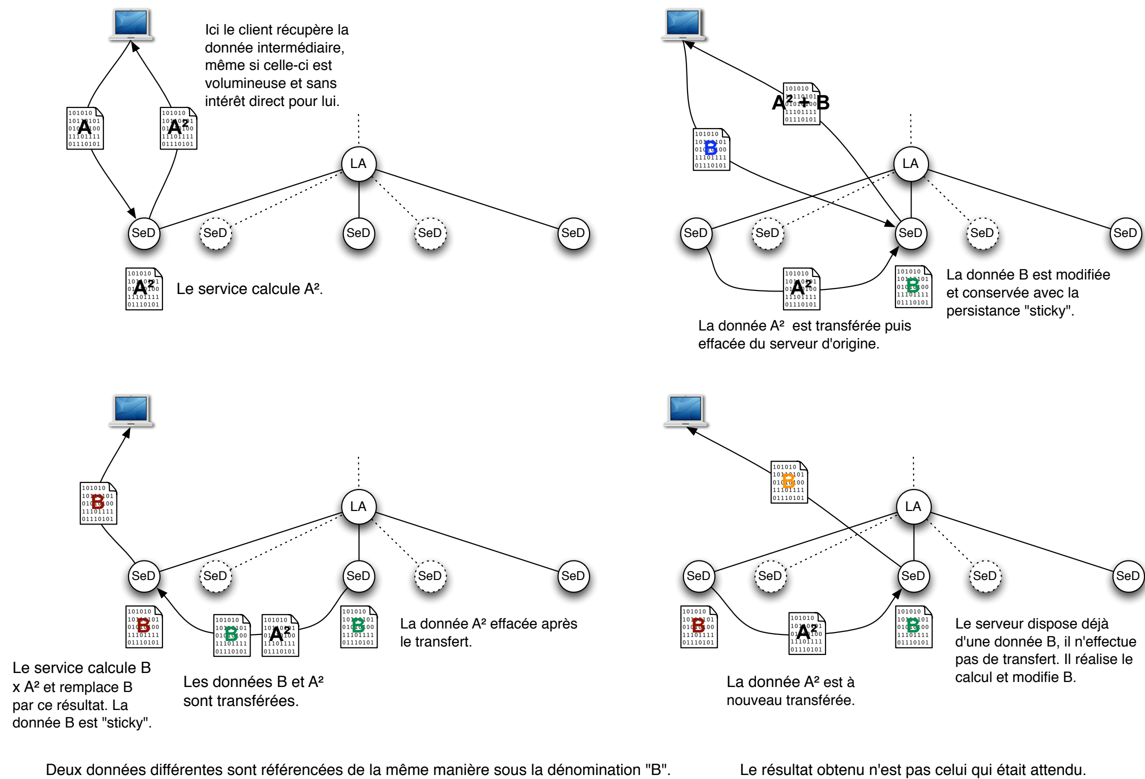

FIG. 2.24 – Exemple de problèmes résultant des limitations de DTM.

l'échelle de la grille. En effet, les MVP s'exécutent en général sur des plateformes très stables regroupant de quelques dizaines à quelques centaines de machines tout au plus. À l'inverse, les systèmes pair-à-pair, permettent une gestion efficace de données non modifiables à une échelle supérieure à celle de la grille (de plusieurs milliers à plusieurs centaines de milliers) avec une volatilité très grande des machines utilisées. En combinant ces deux approches, JuxMem fournit un service permettant d'accéder efficacement et de manière complètement transparente à des données stockées sur une plateforme à l'échelle de la grille (de 1000 à 10000 machines) tout en supportant un degré de volatilité des nœuds de l'ordre de quelques heures.

Les principales caractéristiques de JuxMem sont :

- La transparence d'accès aux données : L'accès aux données s'effectue par l'utilisation d'un identifiant sans aucune gestion de la localisation ou du transfert de celles-ci. Cette propriété de JuxMem est inspirée des systèmes MVP.
- La cohérence des données : Les données sont accessibles en lecture comme en écriture. La gestion de la concurrence d'accès est assurée par le système. Le modèle de cohérence proposé par JuxMem est un modèle de cohérence à l'entrée : Chaque donnée est protégée par un verrou qui lui est propre et les mises à jour ne sont effectuées que lorsqu'elles sont nécessaires. Deux types de verrous sont proposés par JuxMem : Les

verrous d'accès exclusifs pour les données utilisées en écriture et les verrous d'accès partagés pour les données utilisées en lecture seule. JuxMem permet ainsi à plusieurs processus d'accéder simultanément à une même donnée en lecture tandis qu'un seul processus à la fois ne pourra effectuer un accès en écriture.
- La persistance des données : Les données restent accessibles au-delà du temps de leur utilisation sans que JuxMem n'assure toutefois leur pérennité à long terme.
- La tolérance aux pannes : Le service JuxMem supporte la disparition occasionnelle de ressources de l'environnement. La disparition de pairs ne conduit pas à l'arrêt du service et les données sont répliquées de manière à être toujours accessibles si une partie de leurs gestionnaires ne sont plus accessibles.

JuxMem a été implanté à partir l'environnement pair-à-pair JXTA [58] Deux versions ont été conçues basées sur des implantations de JXTA en Java et en langage C (JXTA-C [59]). Pour des raisons de performances et de facilité d'intégration, c'est la version en langage C qui a été utilisée pour la gestion de données dans DIET. Il est cependant à noter que les deux versions sont compatibles et qu'un réseau JuxMem peut faire interagir les deux implantations. Dans JuxMem, trois types d'entités interviennent :

Les clients : Ce sont les "consommateurs" ou "producteurs" des données gérées par JuxMem. Lorsque DIET est compilé pour utiliser JuxMem en tant que gestionnaire de données, les parties clientes et SeDs de l'application se comportent comme des clients JuxMem pour leurs échanges de données. C'est le seule point d'interaction entre JuxMem et DIET, les autres entités réseau de JuxMem étant complétement indépendantes de l'intergiciel.

Les fournisseurs : Il s'agit des éléments de JuxMem qui mettent à disposition les ressources physiques de stockage des données (i.e. une certaine quantité de mémoire). Ils publient les disponibilités des espaces mémoires réservés au partage sur les gestionnaires du réseau JuxMem.

Les gestionnaires : Ils sont chargés de gérer un ensemble de clients et de fournisseurs. Ce sont eux qui fournissent une vue des espaces partagés par les fournisseurs et qui permettent aux clients d'accéder à ceux-ci.

JuxMem supporte une certaine volatilité des gestionnaires et des fournisseurs. La volatilité des clients étant logiquement laissée à la charge des applications qui l'utilise (Dans le cas de DIET, c'est au développeur d'application DIET de gérer les éventuelles reprises sur erreur de son applicatif, JuxMem n'ayant aucun contrôle sur celui-ci). La gestion de la volatilité des fournisseurs repose sur un système "d'abonnement" de ces derniers auprès d'un gestionnaire. Si l'abonnement n'est pas mis à jour comme il doit l'être périodiquement, le gestionnaire considère l'espace comme perdu. La perte d'un gestionnaire est quant à elle gérée par la transformation des fournisseurs en gestionnaire lorsque ceux-ci constatent cette disparition au moment de la mise à jour de leur "abonnement". L'opération de transformation d'un fournisseur est effectuée après un temps aléatoire permettant de limiter l'apparition de plusieurs gestionnaires dans un même groupe. Si malgré tout, plusieurs gestionnaires sont détectés, ils se re-transformeront en fournisseurs, également après un laps de temps de durée aléatoire.

Dans JuxMem, la pérennité des données est assurée par leur réplication. Ainsi, l'utilisateur de JuxMem peut demander la création de m réplicats par site pour une même donnée

permettant de tolérer $\lfloor \frac{m-1}{2} \rfloor$ erreurs simultanées sur ce site.

Les interactions de JuxMem avec DIET s'effectuent à deux niveaux :

Du côté client : Les données persistantes passées en paramètre d'un service sont confiées à la plateforme JuxMem déployée en parallèle de la plateforme DIET. Après l'exécution du service, les données de sortie persistantes sont récupérées depuis la plateforme JuxMem.

Du côté serveur : Les données persistantes sont récupérées depuis la plateforme JuxMem. Les données de sortie persistantes sont à leur tour confiées à JuxMem.

Ainsi, lorsqu'une donnée est confiée à JuxMem, elle reste disponible pour les différents serveurs et peut-être réutilisée en entrée d'autres services. L'utilisation de JuxMem fournit donc un accès transparent aux données et permet une gestion de leur persistance à l'échelle de la grille. Pour l'utilisateur, les interactions avec JuxMem sont complétement transparentes et l'application profite des capacités de stockage et de tolérance aux pannes sans aucune modification. Cependant, tout comme DTM, JuxMem n'effectue aucune gestion des données "volatiles" celles-ci sont toutes transmises en une seule fois, regroupée dans le profil de problème transmis au SeD. On retrouve ainsi les mêmes problèmes de mémoire et de configuration de l'ORB qu'avec DTM.

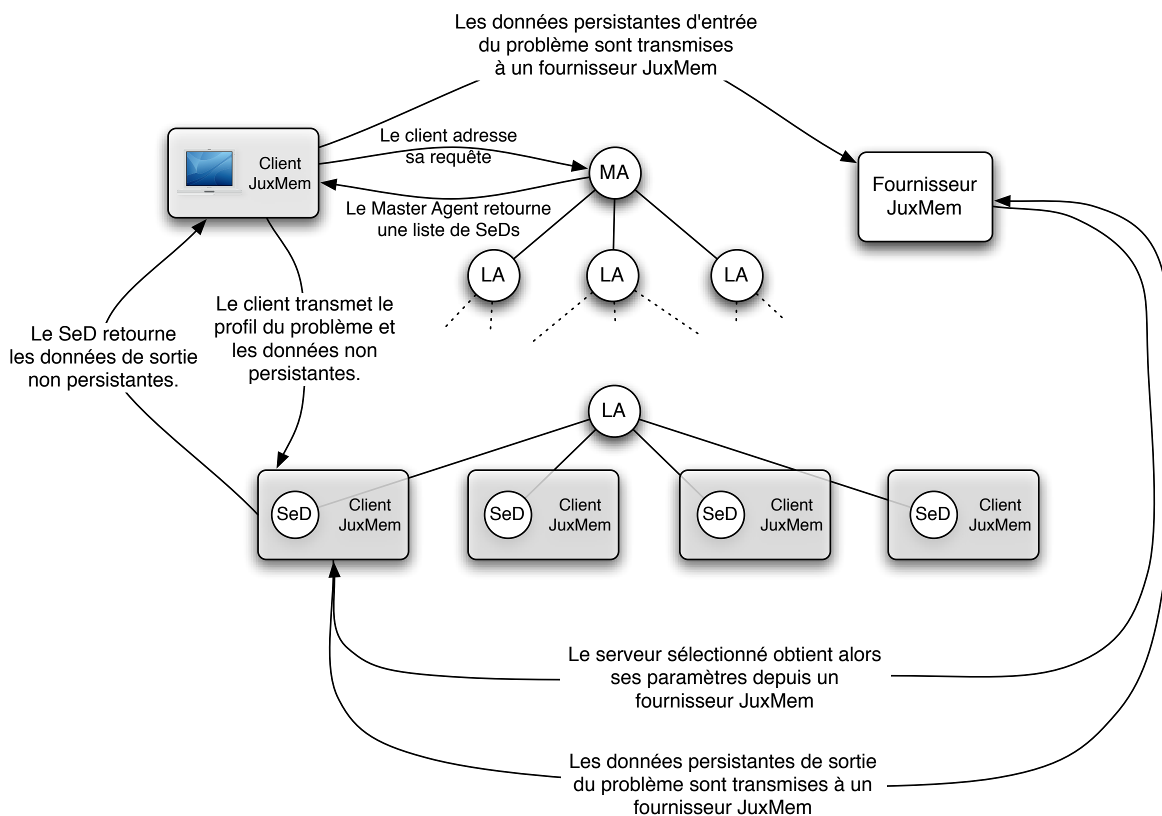

FIG. 2.25 – Interactions DIET-JuxMem

La figure 2.25 présente le fonctionnement de DIET lorsqu'il est compilé en vue d'utiliser JuxMem.

Une description très complète et détaillée du service JuxMem est donnée dans le manuscrit de thèse de Mathieu Jan. Le lecteur intéressé peut se référer à ce document [57].

2.10 Conclusion

La gestion de données est un élément très important dans la réalisation d'une plateforme de grille bio-informatique. Sur de telles plateformes, de nombreuses bases de données doivent être partagées et répliquées par l'intermédiaire des réseaux [95] en utilisant des systèmes de gestion de données [62, 17]. Parmi eux, certains implantent directement la gestion des replications, c'est par exemple le cas de GDMP [45].

Dans [16], les auteurs présentent une plateforme de soumission de recherche d'alignemnents BLAST pour la grille. Cette plateforme est basée sur le Globus Toolkit tout comme GridBLAST [63] un autre système du même type. Ces deux implantations de BLAST pour la grille ne proposent pas de mécanismes de gestion de données. Ainsi, chaque requête commence par le chargement de l'exécutable et de la base de donnée visée. L'ordonnancement proposé n'est alors effectué que sur la base des capacités de calcul des nœuds et la localité des données n'est pas prise en compte.

Dans [83, 84], les bases de données sont mises à jour et répliquées par l'intermédiaire de web-services. Dans ce système, les bases sont considérées comme des ressources de la grille et sont décrites par des fiches XML stockant les informations nécessaire à leur gestion efficace, aussi bien en ce qui concerne les mises à jours qu'en ce qui concerne les moyens de les retrouver sur la grille. Construit pour une infrastructure de grille particulière, ce système est en cours de réécriture afin de rendre son utilisation possible sur la grille européenne EGEE.

On constate que généralement, les plateformes de bio-informatique pour la grille s'appuient largement sur les intergiciels sur lesquels elles ont été construites pour ce qui est de la gestion de données et l'ordonnancement des tâches. Ces plateformes se consacrent avant tout à la mise à disposition des ressources et à la transparence d'accès à celles-ci. Ainsi, les tâches sont généralement ordonnancées en fonction de la disponibilité des ressources et les données répliquées séparément. Ces informations étant elles-même obtenues, gérées et retransmises par l'intergiciel lui-même. Si ces politiques d'ordonnancement et de gestion de données d'application très générale permettent d'utiliser la grille relativement efficacement, nous verrons dans les chapitres suivants que la prise en compte des spécificités de la bio-informatique permet d'améliorer très notablement les performances de la plateforme.

Dans ce premier chapitre nous avons présenté les bases de données biologiques, leur caractéristiques générales et les applications qui les utilisent. Nous avons ensuite présenté la très populaire application BLAST et les travaux déjà réalisés pour sa parallélisation. S'appuyant sur ces travaux, nous avons proposé l'architecture d'une plateforme de soumission de requêtes pour cette application dans un contexte de grilles de calcul. Nous avons vu l'importance de la gestion des données pour la construction d'une telle plateforme et présenté

deux intergiciels de grille permettant sa réalisation.

Chapitre 3

Ordonnancement des tâches bio-informatiques sur plateformes hétérogènes

Dans le chapitre précédent nous avons vu les problématiques de la gestion des données en général ainsi que les possibilités de parallélisation d'une application de bio-informatique très populaire. Nous aborderons dans ce chapitre, les problématiques de l'ordonnancement des tâches bio-informatiques sur plateforme hétérogène en se basant sur l'application BLAST caractéristique de ce type d'applications.

3.1 Contexte

Comme nous l'avons vu au précédent chapitre, les bases de données biologiques utilisées en entrée de l'application BLAST ont pour caractéristiques principales d'être assez volumineuses (de quelques méga-octets à une centaine de giga-octets), mises à jour de manière régulière mais non continue et provenant de plusieurs sources géographiquement distribuées. Par ailleurs, leur intérêt scientifique, la croissance de la communauté des bio-informaticiens et la manière d'utiliser ces bases font qu'un très grand nombre de requêtes doivent être traitées continuellement. Pour que la grille puisse permettre une exploitation efficace de ces bases, il est nécessaire d'allier une bonne gestion de l'ordonnancement des tâches à la gestion efficace des données. Dans ce contexte, un bon ordonnancement des tâches de recherches d'alignements peut permettre des gains non négligeables en terme de performances, permettant d'envisager des recherches dont le coût en puissance de calcul était jusqu'alors prohibitif. La mise à disposition de grandes capacités de stockage et de très nombreuses ressources de calcul offertes par la grille permet par exemple d'envisager la recherche d'homologies entre des génomes complets. Plusieurs travaux traitent de ce type d'utilisation de données par de nombreuses tâches. On notera par exemple des analogies entre la gestion des données du Web et la gestion des bases de données biologiques sur grille de calcul. Les données sur le web sont largement réparties géographiquement et peuvent être énormément solicitées. Pour les plus impressionnantes, on notera le moteur de recherche Google qui avait à traiter de l'ordre de 300 millions de requêtes par jour en 2006 (environ 3500 par seconde) ou encore le site de partage de vidéo youtube qui reçoit environ 100 millions de visites journalières. De plus, les accès aux pages Web et les moments où ceux-ci se produisent sont imprévisibles lorsqu'on les observe individuellement tout comme les requêtes sur les bases de données biologiques. Et si de grandes tendance permettent de prévoir au moins dans quelle mesure un site est visité chaque jour, ces constatations reposent sur des données statistiques. On verra que ce genre d'observations s'applique également aux accès aux bases de données biologiques et que de grandes tendances se dessinent quant à l'utilisation de celles-ci.

L'explosion du nombre de connexion à Internet à conduit depuis déjà quelques années à la mise en place de techniques de réplication et de mise en cache des données pour permettre aux serveurs de supporter la charge provoquée par des utilisateurs toujours plus nombreux. Les articles [98, 77] passent en revue différentes techniques de gestion des accès aux données pour le Web et des méthodes de gestion des caches de données. Par analogie, les techniques de réplications et d'équilibrage de charge mises en œuvre dans ce type de réseaux connus sous le nom de “Content Distribution Network” [75] peuvent être appliqués à la gestion des données et des tâches pour la bio-informatique. Une autre approche, consistant à coupler les réplications et l'ordonnancement des tâches a été étudiée dans [78] et [32] et à fait l'objet des travaux de thèse d'Antoine Vernois [96]. C'est de ces derniers travaux que je suis parti pour améliorer l'approche retenue en introduisant de la

dynamicité dans l'algorithme statique proposé tout en maintenant une forte indépendance de l'ordonnancement et des réplications vis-à-vis des paramètres dynamiques de la grille. En effet, recueillir des informations sur le status des serveurs de calcul nécessite bien souvent la mise en place de services de grille dédiés, complexes à mettre en œuvre et qui n'apportent pas toujours les résultats escomptés notamment en ce qui concerne la "fraîcheur" des données fournies aux utilisateurs. Ce qui handicape d'autant la mise en place de politiques d'ordonnancement reposant sur ce type d'informations.

Dans la suite du chapitre, on considérera le modèle de soumission suivant :
Les clients, répartis sur la grille soumettent des requêtes de taille négligeable par rapport à la taille des bases de données à un ordonnanceur centralisé qui détermine sur quels serveurs celles-ci doivent être réalisées. Chaque serveur dispose d'un certain nombre de ressources et d'une file d'attente gérée suivant le modèle "premier arrivé, premier servi" (FIFO). Les bases de données sont accessibles entre les serveurs et depuis l'extèrieure de la plateforme (depuis des serveurs ftp publics par exemple). Ce modèle correspond bien à ce qui peut être fait sur la grille EGEE ou en utilisant l'intergiciel DIET. La figure 3.1 présente le fonctionnement général de la plateforme.

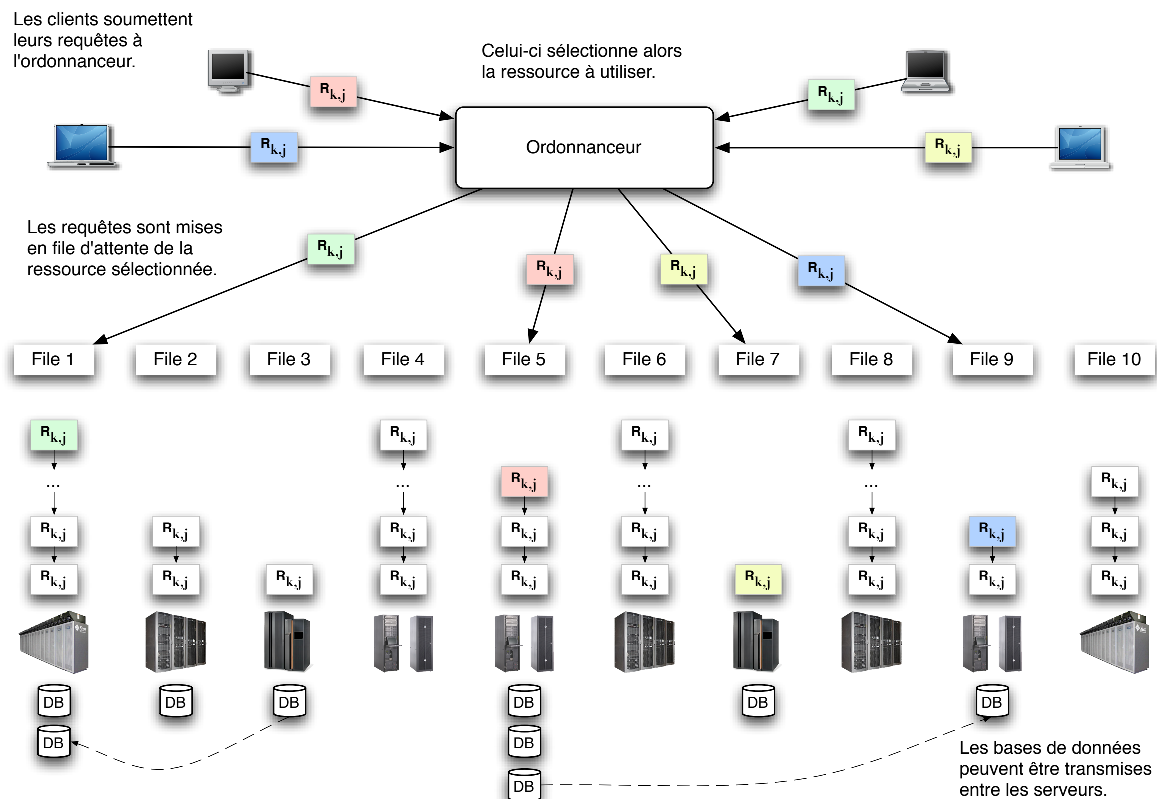

FIG. 3.1 – Modèle de soumission des tâches

3.2 Problématique générale

Le nombre, la variété et la distribution géographique des utilisateurs d'une plateforme bio-informatique sur la grille, font qu'il est difficile de prévoir quand et où une tâche sera soumise. Par ailleurs, les bases gérées par la plateforme sont elles-même réparties et accèdées de manière imprévisible au cours du temps. Sans information supplémentaire, il est compliqué de proposer un ordonnancement des tâches adapté à l'utilisation de la grille, tout du moins un ordonnancement permettant des gains de performance sur la plateforme. L'ordonnanceur peut accéder à plusieurs informations suivant l'intergiciel avec lequel il interagit :

– La distribution des bases de données sur les différents sites.
– Des données statiques sur la configuration de la plateforme tels le type, la puissance, le nombre de processeurs d'un nœud ou encore ses capacités de stockage, de mémoire etc.
– Des données dynamiques : L'espace disque disponible, la charge processeur, la taille des files d'attente des différents sites etc.
– Toutes les informations statiques à propos des tâches qu'on lui soumet, au fur et à mesure qu'on lui soumet. À savoir, leur nombre, le type d'application visé, les données nécessaires, les paramètres transmis etc.

Ces différentes informations peuvent être utilisées pour l'ordonnancement des nouvelles tâches, mais elle ne permettent que d'assurer un ordonnancement des requêtes unes-à-unes sans garantie sur les performances globales de la plateforme. Nous aborderons ici deux approches différentes de l'ordonnancement en ligne des tâches de bio-informatique sur la grille. La première approche faisant appel aux informations dynamiques de la grille : la charge processeur, la bande passante disponible entre les sites etc. Et l'estimation du temps de réalisation des tâches en file d'attente. Il s'agit ici, pour chaque requête de choisir le nœud qui lui permettra d'être réalisée dans le temps minimum à l'instant de sa soumission. C'est l'algorithme "Temps d'Accomplissement Minimum" ou MCT (Minimum Completion Time), présenté dans la section suivante. La seconde approche ne fait plus appel qu'aux données statiques de la grille et à une estimation statistique du nombre de requêtes de chaque type que l'on peut attendre. C'est l'approche retenue par l'algorithme d' "Ordonnancement et de Réplications" ou SRA (Scheduling and Replication Algorithm) présenté à la section 3.4.

3.3 Ordonnancement "Temps d'Accomplissement Minimum"

Une première approche du problème consiste à ordonnancer les tâches bio-informatiques en suivant la politique classique d'ordonnancement "temps d'accomplissement minimum" (Minimum Completion Time ou MCT).

Il s'agit ici, pour chaque tâche adressée à l'ordonnanceur, de choisir la machine qui permettra de la réaliser en le minimum de temps possible. On prendra donc en compte, le temps nécessaire à la réalisation de la tâche, mais aussi le temps nécessaire à la récupération éventuelle de la base de donnée.

On considérera une plateforme de m nœuds de calcul dénommés S_i pour $i \in [1..m]$ ayant

pour caractéristiques respectives :
- w_i, la puissance de calcul en nombre d'opérations réalisables par seconde.
- m_i, la capacité de stockage du nœud en méga-octets.

Les n bases de données seront notées d_j, pour $i \in [1..n]$. Chaque base ayant une taille en méga-octets notée $size_j$ respectivement pour chaque d_j.

On dispose d'un ensemble d'algorithmes à exécuter sur les bases (dans notre cas, les cinq différentes versions de BLAST : BLASTN, BLASTP, TBLASTN, BLASTX et TBLASTX (Voir la section 2.3). dont les complexités en temps seront notées $\alpha_k \times size_j + c_k$ pour leur application sur la base d_j. Les algorithmes considérés seront notés a_k, pour $k \in [1..p]$.
Les requêtes soumises par les utilisateurs seront donc un ensemble de couples (a_k, d_j) notés $R_{k,j}$ signifiant l'application de l'algorithme a_k sur la base d_j. Et le temps nécessaire à la réalisation de la tâche $R_{k,j}$ sur le nœud S_i sera égal à $\frac{\alpha_k \times size_j + c_k}{w_i}$.

La bande passante disponible entre chaque nœud de calcul sera donnée par la matrice b_s^t pour $s \in [1..m]$, $t \in [1..m]$ avec $s \neq t$. Ainsi, le temps nécessaire à la transmission de la base d_j entre le nœud S_s et le nœud S_t est égal à $\frac{size_j}{b_s^t}$. Étant donné la taille des données, les temps de latences sont considérés comme négligeables.

Chacun des serveurs de calcul gère sa propre file d'attente des tâches qu'il a à réaliser suivant une politique du "premier arrivé, premier servi" (FIFO) et peut fournir une estimation Q_i du temps nécessaire à la réalisation de toutes les tâches dans la file.

On notera enfin δ_i^j la matrice de distribution des bases de données sur les nœuds de la grille telle que :

$$\begin{cases} \delta_i^j = 1 & \text{si la base } d_j \text{ est stockée sur le nœud } S_i \\ \delta_i^j = 0 & \text{sinon.} \end{cases}$$

L'algorithme MCT peut alors s'écrire de la manière suivante :

Algorithme MCT

$T \longleftarrow \infty$

TantQue `(l'ordonnanceur peut recevoir de nouvelles tâches` $R_{k,j}$`)` **faire**

  Pour $i \in [1..m]$ **faire**

    Si $\frac{\alpha_k \times size_j + c_k}{w_i} + \underset{s\in[1..m] \text{ avec } \delta_s^j=1}{Min} (\frac{size_j}{b_s^i}) + Q_i < T$ **alors**

      $\mathcal{S} \longleftarrow S_i$

      $T \longleftarrow \frac{\alpha_k \times size_j + c_k}{w_i} + \underset{s\in[1..m]\delta_s^j=1}{Min} (\frac{size_j}{b_s^i}) + Q_i$

    FinSi

  FinPour

  `Ordonnancer la tâche` $R_{k,j}$ `sur le serveur` $\mathcal{S}$

FinTantQue

Pour chacune des tâches adressée à l'ordonnanceur, celui-ci sélectionne le nœud qui nécessitera le moins de temps à sa réalisation. On considère ici des transferts de données synchrones : Lorsqu'une tâche est ordonnancée sur un nœud, celle-ci commence par vérifier si la base de données est présente et si ce n'est pas le cas, effectue le transfert de celle-ci. Si l'espace de stockage vient à manquer, on utilise un algorithme de gestion des caches pour sélectionner une donnée à effacer. L'algorithme doit prendre en compte les possibilités d'effacement d'une donnée pour les calculs de temps de transferts. La matrice δ_i^j dépendant alors du moment de démarrage de la tâche.

On pressent aisément les problèmes qui peuvent apparaître en utilisant cet algorithme. En prenant par exemple deux tâches distinctes à distribuer sur deux machines de puissances différentes (L'une est une fois et demie plus rapide que l'autre) qui disposent chacune des bases nécessaires à leur réalisation, avec la tâche n˚2 deux fois plus longue à réaliser que la tâche 1, on obtient la distribution suivante (la figure 3.2 présente les diagrammes de Gantt des différents ordonnancements qui peuvent être obtenus par l'algorithme et de l'ordonnancement optimal) :

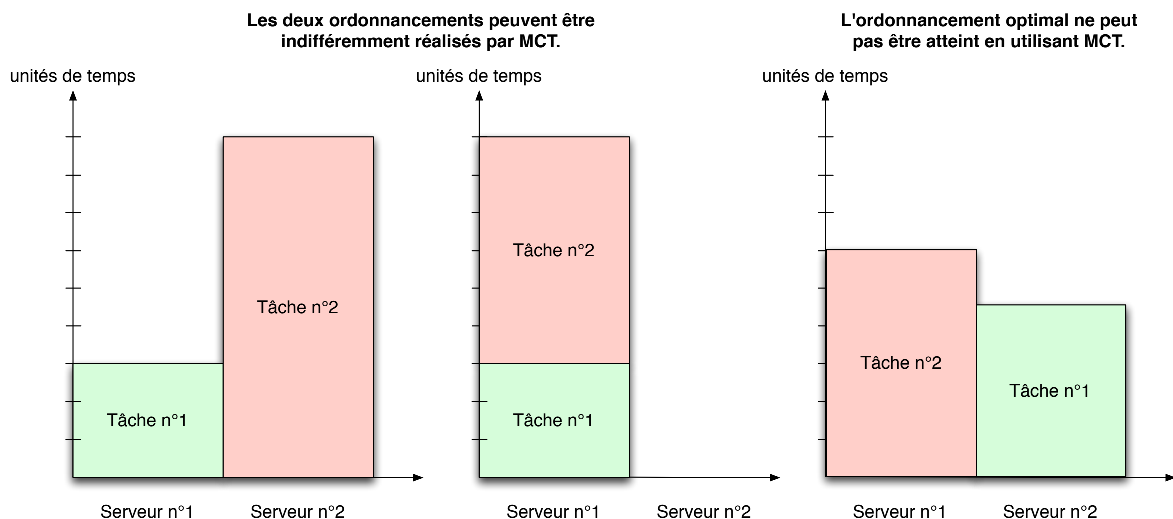

FIG. 3.2 – Diagrammes des différents ordonnancements.

- La tâche n˚1 est soumise à l'ordonnanceur, celui-ci détermine que le temps nécessaire à son achèvement est minimal si on choisit le serveur 1 qui est le plus rapide. Il faudra en effet, 3 unités de temps pour qu'elle soit exécutée.
- L'ordonnanceur adresse donc la tâche n˚1 au serveur 1. La tâche n˚2 est alors soumise à l'ordonnanceur. Celui-ci détermine qu'il faudra 6 unités de temps pour réaliser celle-ci sur le serveur 1 une fois la tâche numéro 1 achevée. L'ordonnancement sur la première machine conduit donc à une attente de $6 + 3 = 9$ unités de temps. Il détermine aussi qu'il faudra 9 unités de temps pour réaliser celle-ci sur le serveur 2. Il choisit donc indifféremment l'un ou l'autre serveur.

Le temps d'attente total pour réaliser les deux tâches est donc de 9 unités de temps. Or si la tâche 1 avait été adressée au serveur n˚2, elle aurait nécessité 4,5 unités de temps. La tâche n˚2 étant alors adressée au serveur 1 et ne nécessitant alors plus que 6 unités de temps. Le

temps total pour réaliser les deux tâches aurait alors été de 6 unités de temps.

Ce type de situation où MCT ne fournira pas l'ordonnancement optimal est inévitable dans notre contexte d'ordonnancement des tâches unes-à-unes sans information supplémentaire quant à l'ordre et les proportions dans lesquelles elles seront soumises. MCT permet toutefois d'obtenir des résultats très convenables dans la plupart des cas.

3.4 L'algorithme de Réplication et d'Ordonnancement : Schedule and Replication Algorithm

Dans [37] les auteurs développent une approche intéressante consistant à combiner réplication/distribution des données sur la grille et ordonnancement des tâches en régime permanent. L'algorithme SRA (**S**cheduling and **R**eplication **A**lgorithm) présenté dans cet article donne de bons résultats en terme d'optimisation de débit de tâches sous certaines conditions. Par l'analyse de l'utilisation de clusters par des bio-informaticiens, les auteurs ont montré qu'il est raisonnable de considérer que les requêtes soumises par les utilisateurs auront une fréquence constante si tant est qu'on choisit un intervalle de temps suffisament long. L'algorithme SRA va tirer profit de ces informations supplémentaires quant aux proportions des différents types de tâches soumis sur la grille. Pour cela, il va utiliser un programme linéaire décrivant les contraintes du problème de l'ordonnancement des tâches et prenant en compte les fréquences attendues des différents types de requêtes. On appelle "type de requête", un couple (k, j), algorithme / base de données et la fréquence de ce type de requête, la proportion moyenne de toutes les requêtes faisant appel à l'algorithme k sur la donnée j. Ainsi, "(BLASTP, Swissprot)" est un type de requête (dont la proportion est en général très élevée dans l'utilisation que font les bio-informaticiens des clusters qui sont mis à leur disposition).

Dans la suite on reprendra une partie des notations utilisées pour la description de l'algorithme MCT à la section précédente, à savoir :
- $\{S_i\}_{i\in[1..m]}$: les m serveurs de calculs de capacités de stockage respectives m_i et de puissances de calcul respectives w_i.
- $\{d_j\}_{j\in[1..n]}$: les n données de tailles respectives $size_j$.
- $\{a_k\}_{k\in[1..p]}$: les p algorithmes de complexités en temps respectives $\alpha_k \times size_j + c_k$.
- $R_{k,j}$ les requêtes utilisant l'algorithme a_k sur la base de données d_j.
- La matrice de distribution des bases sur les nœuds δ_i^j, avec $\delta_i^j = 1$ si la donnée d_j est présente sur le nœud S_i et 0 sinon.

On notera par ailleurs $n_i(k, j)$ le nombre de requêtes de type $R_{k,j}$ effectuées sur le serveur S_i. Pour l'expression des contraintes d'utilisation des bases avec des algorithmes qui peuvent s'y appliquer (il n'y a par exemple pas de sens à appliquer un algorithme prévu pour les bases d'ADN sur une base de protéine), on notera $v_{k,j}$ tel que

$$\begin{cases} v_{k,j} = 1 & \text{si l'application de } a_k \text{ à la base } d_j \text{ a un sens.} \\ v_{k,j} = 0 & \text{sinon.} \end{cases}$$

TP désigne le rendement de la plateforme, c'est la variable à optimiser. Et enfin, $f_{k,j}$ les fréquences des requêtes de chaque type dans le flot de requêtes.

L'algorithme SRA repose sur la résolution du programme linéaire défini par les contraintes suivantes :

- Chaque donnée est présente au moins une fois sur la plateforme.

$$\forall j \sum_{j=1}^{n} \delta_i^j \geq 1$$

- On ne peut pas stocker plus de données sur un nœud que sa capacité totale de stockage.

$$\forall i \sum_{j=1}^{n} \delta_i^j \times size_j \leq m_i$$

- Les requêtes confiées au nœud S_i ne peuvent excéder sa capacité de calcul.

$$\forall i \sum_{k=1}^{p} \sum_{j=1}^{n} n_i(k,j) \times (\alpha_k \times size_j + c_k) \leq w_i$$

- Le nombre de requêtes de type $R_{k,j}$ exécutées sur un serveur S_i est nul si ce type de requête est impossible ou si la donnée d_j n'est pas sur le nœud S_i.

$$\forall i \; \forall j \; \forall k \; n_i(k,j) \leq v_{k,j} \times \delta_i^j \times \frac{w_i}{\alpha_k \times size_j + c_k}$$

- Le nombre de requêtes de chaque type respecte les proportions fixées par les fréquences $f_{k,j}$.

$$\forall j \; \forall k \; \sum_{i=1}^{m} n_i(k,j) = f_{k,j} \times TP$$

En découle le programme linéaire suivant :

Maximiser TP sous les contraintes

$$\begin{cases} \sum_{j=1}^{n} \delta_i^j \geq 1 & 1 \leq i \leq m \\ \sum_{j=1}^{n} \delta_i^j \cdot size_j \leq m_i & 1 \leq i \leq m \\ n_i(k,j) \leq v_{k,j} . \delta_i^j . \frac{w_i}{\alpha_k \cdot size_j + c_k} & 1 \leq i \leq m, 1 \leq j \leq n, 1 \leq k \leq p \\ \sum_{k=1}^{p} \sum_{j=1}^{n} n_i(k,j)(\alpha_k \cdot size_j + c_k) \leq w_i & 1 \leq i \leq m \\ \sum_{i=1}^{m} n_i(k,j) = f_{k,j} \cdot TP & 1 \leq i \leq m, 1 \leq j \leq n \\ \delta_i^j \in 0,1 & 1 \leq i \leq m, 1 \leq j \leq n \end{cases}$$

L'approximation en nombres entiers de ce programme linéaire donne donc à la fois une distribution de données δ_i^j et pour chaque type de requête $R_{k,j}$ le nombre $n_i(k,j)$ de requêtes à exécuter sur le nœud S_i.

Le placement δ_i^j est un placement statique des données, c'est à dire qu'on considère qu'au lancement de la plateforme, les données sont déjà distribuées suivant cette configuration. Ainsi, aucun déplacement de données n'est pris en compte dans la solution proposée. On verra dans le chapitre 5 que la distribution des données après le début des soumissions permet d'obtenir de bons résultats en terme de performances en un temps fini dépendant essentiellement des capacités réseaux de la plateforme et de la taille des données à déplacer.

3.5 Conclusion

Dans ce chapitre nous avons présenté deux approches pour l'ordonnancement en-ligne des requêtes BLAST sur la grille. La première approche qui sélectionne le nœud en fonction des données dynamiques de la grille comme la charge processeur ou la taille des files d'attente, nécessite un système d'information robuste et efficace. De plus, dans cette approche, la distribution des données est inhérente à l'ordonnancement des tâches. Ce dernier point peut s'avérer particulièrement handicapant lorsque les données sont de tailles très hétérogènes comme c'est souvent le cas pour les bases de données biologiques. Nous verrons dans le chapitre 5 que les performances de MCT peuvent en souffrir très largement. La seconde approche, quant à elle, basée sur des estimations statistiques du nombre de tâches de chaque type fournit un placement statique des données et le nombre de tâches à réaliser sur chacun des sites. Le principal problème est alors la validité des proportions utilisées sur des intervalles de temps de longueur acceptable. Dans le chapitre 5, nous présenterons une évolution de cet algorithme tentant de minimiser ce problème et ne nécessitant plus d'informations statistiques préalables, l'algorithme se chargeant lui-même de les recueillir.

Deuxième partie

Contribution à la gestion des bases de données biologiques sur les grilles de calcul

Dans la première partie, nous avons abordé les problématiques de la gestion des données biologiques sur la grille et de l'ordonnancement des tâches bio-informatiques. Nous nous sommes placés dans le contexte de l'utilisation d'une application de bio-informatique : le BLAST. Nous avons vu les différentes possibilités de parallélisation de cette application dans le chapitre 2 section 2.4.

De cette analyse a découlé l'architecture d'une plateforme de soumission des requêtes BLAST qui devra assurer la réplication des données, la localisation de celles-ci et la distribution de très nombreuses requêtes de courte durée d'exécution.

La gestion des données offerte par l'intergiciel gLite permet la réalisation d'une telle plateforme mais le système d'ordonnancement proposé, conçu essentiellement pour des soumissions de tâches de longue durée, s'adapte mal à notre contexte de travail. À l'inverse, l'intergiciel Grid-RPC DIET permet l'implantation efficace des algorithmes d'ordonnancement pour de très grands ensembles de tâches courtes à réaliser en parallèle, mais ne dispose pas d'un gestionnaire de données permettant de réaliser des réplications de données explicites.

Dans cette partie, nous présentons un gestionnaire de données pour la grille conçu pour permettre la réplication explicite des bases de données biologiques avec l'intergiciel DIET. Nous présentons ensuite notre travail sur l'ordonnancement et la réplication conjoints pour les applications de bio-informatique, mis en application dans une implantation pour DIET du logiciel BLAST grâce à ce nouveau gestionnaire de données.

Chapitre 4

Un gestionnaire de données sur la grille : DAGDA

Les chapitres précedents ont montré les besoins en terme de gestion de données pour le portage sur la grille d'applications de bio-informatique telles que BLAST. C'est dans cette optique que nous avons développé un nouveau gestionnaire de données pour la grille que nous avons ensuite intégré à l'intergiciel DIET. En effet, Data Arrangement for the Grid and Distributed Applications (DAGDA) [39] est construit sur le même modèle de déploiement hiérarchique que DIET et a été d'abord conçu pour pallier aux manques du gestionnaire DTM intégré à cet intergiciel. Cependant, DAGDA peut être vu comme une librairie de gestion de données pour la grille dissociée de DIET comme l'est par exemple JuxMem. Il diffère de JuxMem par l'approche retenue. En effet, JuxMem permet un accès efficace et transparent aux données sans que l'utilisateur puisse choisir explicitement les nœuds de stockage pour ses données. DAGDA permet quant à lui de choisir directement où et quand une donnée doit être placée. Il est d'ailleurs prévu de séparer DAGDA en deux entités distinctes dont l'une sera consacrée au stockage de données et l'autre à l'interfaçage entre différents gestionnaires, que ce soit une plateforme JuxMem, un objet DAGDA, un serveur GridFTP etc.

Dans ce chapitre quand nous parlerons de DAGDA, il s'agira toujours de l'intégration de DAGDA à DIET.

4.1 L'architecture de DAGDA

DAGDA utilise une architecture sensiblement similaire à celle utilisée par DIET. CORBA est utilisé comme couche de communication pour DAGDA tout comme il l'est pour DIET et chacun des gestionnaires DAGDA de la plateforme est connecté aux autres suivant une topologie en arborescence. La figure 4.1 présente l'organisation des objets de DAGDA au sein de DIET. Ainsi, le déroulement général de la résolution d'un problème par DIET en utilisant DAGDA s'effectue en 6 étapes principales numérotées de 1 à 6 sur la figure 4.1 :

1. Le client effectue un appel du service.
2. DIET sélectionne un ou plusieurs SeD(s) pouvant réaliser le service demandé.
3. Le client effectue alors la soumission du problème en envoyant la description seule de celui-ci. Les données ne sont pas transmises. (La seule exception étant les données scalaires, qui sont transmises avec le profil, celles-ci étant de taille trés réduite - au maximum 8 octets).
4. Le SeD récupère, les données sur le client, ou directement depuis un autre nœud de la plateforme si celles-ci y sont déjà disponibles.
5. Le SeD exécute la requête avec les paramètres ainsi transmis.
6. Le SeD effectue alors les mises à jour des données de sortie et d'entrée-sortie qu'il a modifiées si celles-ci sont enregistrées sur la plateforme. Il retourne les données de sorties au client qui en a fait la demande en définissant celles-ci comme "volatiles" ou persistantes "avec retour".

Au sein de cette architecture, le composant DAGDA du client a une place particulière. En effet, celui-ci est déconnecté du déploiement et n'interagit avec la plateforme qu'à travers des appels au Master Agent et des transferts vers et depuis les SeDs choisis pour la résolution d'un problème.

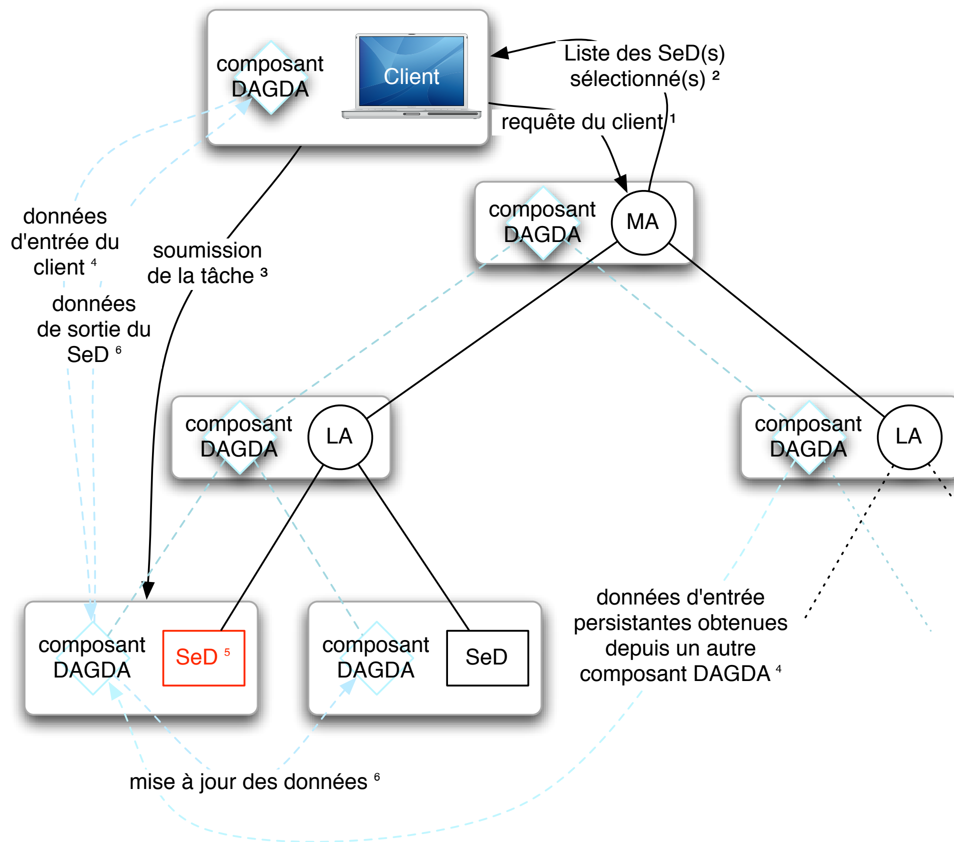

FIG. 4.1 – Architecture générale de DAGDA

Chacun des éléments DAGDA de la plateforme dispose d'une architecture interne décrite dans la figure 4.2. Dans l'architecture interne des objets DAGDA, on notera les éléments principaux suivant :

– L'interface CORBA : C'est la couche de communication de DAGDA avec les autres éléments de la plateforme. Il s'agit de la partie de l'objet qui est accessible depuis l'ensemble du réseau. Elle est obtenue par la définition IDL (Interface Description Language) de l'objet et sa traduction par le compilateur IDL en une entité du langage visé. Dans le cas de DAGDA, il s'agit du C++.
– Le gestionnaire des données locales : Il s'agit de la partie de DAGDA effectuant les contrôles nécessaires à l'accès aux données physiquement stockées sur le nœud et choisissant, le cas échéant, des données à effacer pour libérer de l'espace.
– Le gestionnaire des sauvegardes/restaurations des données sur le nœud : DAGDA permet de réaliser une "image" de l'état des données sur les nœuds qui le permettent. Le gestionnaire se charge d'enregistrer dans un fichier les données mémoires gérées par DAGDA et les chemins d'accès aux fichiers référencés. Lorsque l'option est activée, au démarrage, le fichier est lu par DAGDA afin de remettre en mémoire les données ainsi enregistrées et de les référencer à nouveau ainsi que les fichiers encore disponibles au moment du lancement. Ce mécanisme permet l'arrêt de la plateforme sans perte des données au moment où on relance celle-ci.
– L'implantation de l'objet DAGDA lui-même : C'est la couche logicielle qui fait la liaison

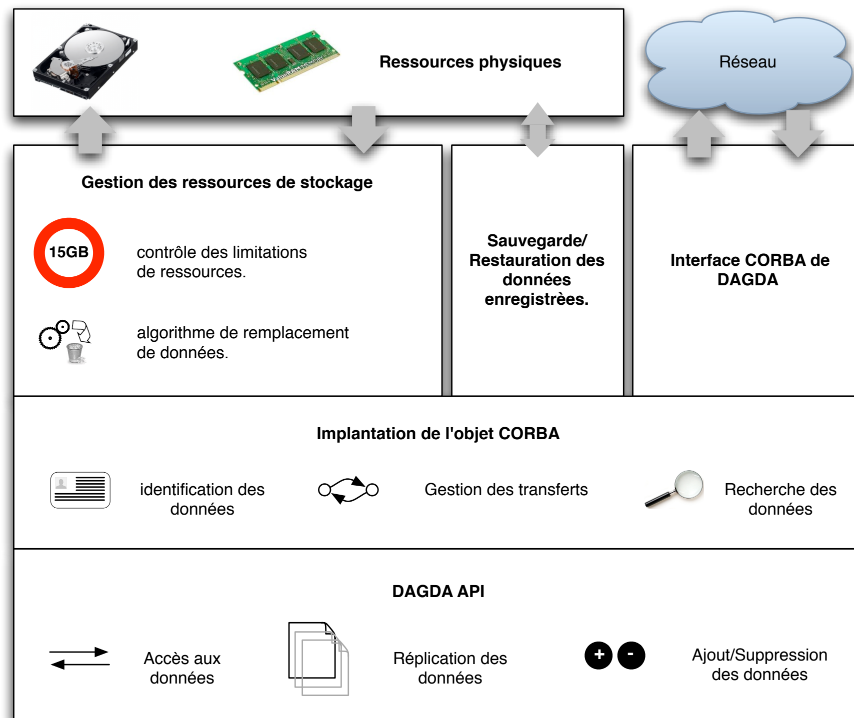

FIG. 4.2 – Architecture interne des objets DAGDA

entre l'interface CORBA, les gestionnaires de ressources physiques et l'API DAGDA.
- L'API DAGDA : DAGDA fournit une API de gestion directe des données utilisables à tout niveau de DIET. L'API est décrite en détail dans la section 4.2.2.

La section suivante décrit en détail la gestion de données opérée par DAGDA au sein de DIET comme par l'intermédiaire de l'API fournie.

4.2 La gestion de données dans DIET avec DAGDA

Le gestionnaire de données DAGDA permet une gestion des données explicite et implicite. C'est à dire que la soumission d'une tâche DIET, en utilisant DAGDA induira une gestion des données transparente pour l'utilisateur. À l'inverse, en utilisant l'extension de l'API DIET fournie par DAGDA, l'utilisateur peut effectuer des opérations sur les données complètement dissociées de toute tâche de soumission. DAGDA introduit ainsi dans DIET de nouvelles fonctions de gestion des données :
- La gestion implicite ou explicite des réplications de données.
- Un système de sauvegarde et de restauration de la configuration des données sur la plateforme.
- Le partage en lecture seule de fichiers accessibles entre plusieurs nœuds ayant par exemple accès à la même partition NFS.
- Le choix d'un algorithme de remplacement de données pour la gestion implicite de celles-ci.
- Une configuration avancée de l'espace mémoire et de l'espace disque utilisable par

DAGDA, pour stocker les données, mais aussi les transmettre.

L'utilisation de DTM comme de JuxMem ne permettant l'ajout de nouvelles données sur la plateforme que par l'intermédiaire d'un appel de service, le système d'identification des données centralisé ne pose pas réellement de problème. En effet, la création des identifiants de données étant réalisée par le seul Master Agent, celui-ci peut s'assurer simplement de l'unicité de l'identifiant attribué. Puisque DAGDA permet l'ajout de données en tout point de la hiérarchie, nous avons dû changer le système de nommage des données. Afin d'éviter de solliciter le Master Agent à chaque création d'un nouvel identifiant, nous avons opté pour la création d'identifiants uniques sur le modèle des *Universally Unique IDentifiers* [67]. Ce système assurant une probabilité de collision de nom extrêmement faible, il nous a permis de décentraliser la création des identifiants sur la grille.

DAGDA autorise une gestion plus fine des données en permettant de définir des espaces de stockage sur tout nœud de la plateforme. En effet comme tout les composants DAGDA de la hiérarchie sont identiques, des données peuvent être stockées sur les différents agents permettant à la fois d'augmenter les capacités de stockage de la plateforme et d'optimiser les placements de données en approchant celles-ci des nœuds de calcul.

Afin d'éviter les problèmes de mémoires posés par les autres gestionnaires de données de DIET, DAGDA permet de fixer une taille maximale de message pour le transfert des données. Les données gérées par DAGDA peuvent ainsi être transmises en plusieurs fois, limitant l'utilisation de la mémoire à la taille choisie par l'utilisateur et cette taille étant automatiquement bornée à la taille maximale permise par l'ORB, l'utilisateur n'a plus à s'inquiéter de la configuration de celui-ci.

Avec DAGDA, l'utilisateur peut définir la taille mémoire et l'espace disque réservés aux données de DIET. Si l'espace disponible n'est pas suffisant, DAGDA tente d'effacer une donnée parmi celles qui sont définies comme persistantes (et pas "sticky"). Si l'espace nécessaire reste supérieur à l'espace disponible, une simple erreur est retournée à l'utilisateur, la stabilité de la plateforme ne souffrant en aucun cas de ce type de problème.

On distinguera deux manières de gérer les données en utilisant DAGDA : La gestion transparente dans les appels de procédures distantes de DIET et la gestion directe par l'intermédiaire de l'API. Ces deux manières de gérer les données sont présentées dans les sections suivantes.

4.2.1 La gestion des données dans les appels de procédures distantes de DIET

Lorsque le Master Agent d'une plateforme DIET transmet au client une liste de SeDs correspondant aux besoins formulés dans un profil de problème, le client sélectionne un SeD. À partir de ce moment, il doit transmettre les données du profil au SeD. Lorsque DIET utilise DAGDA, toutes les données sont gérées par celui-ci : qu'elles soient persistantes ou volatiles, elles sont enregistrées dans le gestionnaire DAGDA du client et les transferts s'effectuent alors entre celui-ci et le gestionnaire du serveur. Pour la réalisation de ces transferts, DAGDA utilise le modèle "pull", c'est à dire que les données sont "réclamées" par leur destination et non transmises d'autorité par leur source. La figure 4.3 présente le fonctionnement de la gestion de données dans DIET avec DAGDA.

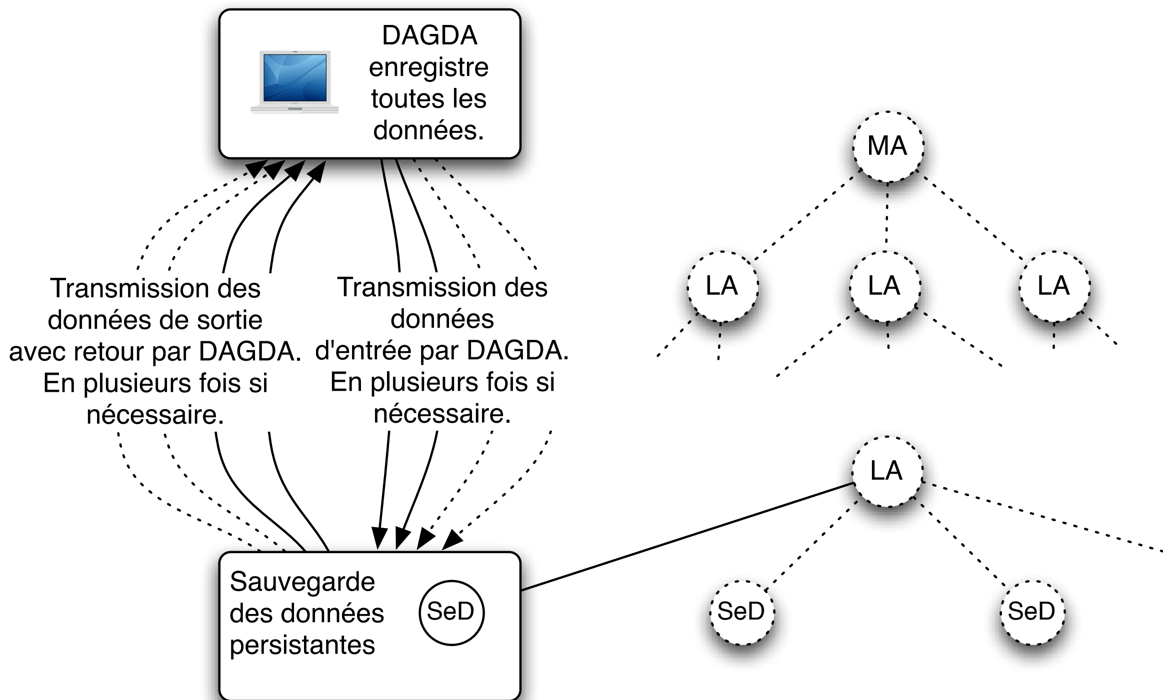

FIG. 4.3 – Gestion avec DAGDA

Les algorithmes de remplacement de données proposés par DAGDA

DAGDA permet de choisir un algorithme de remplacement de données parmi trois actuellement implantés :

Least Recently Used (LRU) : La donnée sélectionnée est celle qui a été utilisée "le moins récemment" parmi l'ensemble des données persistantes et "non-sticky" de taille suffisante. Pour cet algorithme, DAGDA maintient une liste des derniers temps d'utilisation de la donnée.

Least Frequently Used (LFU) : La donnée sélectionnée est celle qui a été la moins utilisée parmi l'ensemble des données persistantes et "non-sticky" de taille sufisante. Pour cet algorithme, DAGDA maintient un historique d'utilisation des données.

First In First Out (FIFO) : La donnée sélectionnée est celle qui a été enregistrée le plus longtemps parmi l'ensemble des données persistantes et "non-sticky" de taille suffisante. Pour cet algorithme, DAGDA maintient une liste des "dates" d'enregistrement des données.

On notera bien que les données concernées par ces algorithmes ne sont que les données "non-sticky" dont la seule libération permet le stockage de la nouvelle donnée. Ainsi, s'il faut plusieurs effacements de données pour libérer suffisamment d'espace, les algorithmes ne trouveront pas de donnée suffisamment grande et l'ajout de la donnée sera impossible. Chacun de ces algorithmes étant implanté séparément, dans une prochaine évolution de DAGDA, il sera possible de développer son propre algorithme de remplacement de données et des politiques prenant en compte la présence de plusieurs données "effaçables" seront

proposées.

4.2.2 La gestion directe par l'intermédiaire de l'API DAGDA

DAGDA définit une extension de l'API DIET permettant l'ajout de données directement sur la plateforme et l'obtention d'une donnée stockée sur celle-ci, et ce, de manière synchrone ou asynchrone. Par ailleurs, DAGDA permet la réplication explicite des données déja présentes et fournit un système d'alias permettant un partage aisé de celles-ci. Enfin, une fonction de l'API permet de provoquer la sauvegarde des données stockées sur les nœuds qui le permettent. Nous allons expliciter les différentes fonctions et macros de l'API puis nous aborderons les différentes utilisations qui peuvent en être faites.

Ajout de données sur la grille

Comme nous l'avons vu précédemment, dans un déploiement DIET, le composant DAGDA du client est dissocié du reste de la hiérarchie. Ainsi, les appels aux fonctions de l'API diffèrent si ils sont effectués depuis un nœud de la hiérarchie ou depuis un client. Pour tous les appels effectué par un client, celui-ci s'adresse directement au composant DAGDA du Master Agent. Par contre, depuis un nœud de la hiérarchie, c'est le composant DAGDA local qui est utilisé. Ce fonctionnement, indispensable si on souhaite permettre une gestion des données depuis les applications clientes, a une incidence sur la manière dont sont ajoutées les données par l'intermédiaire de l'API. En effet, lorsque l'utilisateur dépose une donnée sur la grille, l'intergiciel doit sélectionner quel nœud sera mis à contribution. On peut imaginer plusieurs stratégies de placement des données de départ : Le nœud le plus "proche" dans la hiérarchie, le nœud le moins occupé, le nœud le plus sollicité etc. DAGDA utilise cette première stratégie, c'est à dire qu'il tente d'abord de placer la donnée localement et ne fait récursivement appel aux autres nœuds que lorsque cela est nécessaire. Ainsi, les données déposées par un client seront d'abord placées sur l'espace de stockage du Master Agent, alors même que les données ajoutées depuis un SeD ne pourront l'être que sur son espace local. Les figures 4.4, 4.5 et 4.6 présentent le comportement de DAGDA au moment de l'ajout d'une donnée par l'intermédiaire de l'API.

Ce comportement par défaut peut s'avérer problématique et le choix du nœud pour le placement initial d'une donnée pourrait être effectué par l'utilisateur. Pour l'implantation de DAGDA, nous avons privilégié la transparence d'utilisation de la grille et c'est l'intergiciel qui choisit lui-même où placer les données. Cependant, afin de permettre aux utilisateurs qui souhaitent optimiser ces placements de données initiaux, DAGDA proposera dans l'avenir un mécanisme de sélection du ou des nœuds sur lesquels la donnée doit être installée. Les différentes fonctions et macros de l'API permettant l'ajout direct d'une donnée sur la plateforme sont les suivantes :

Ajout synchrone des données :

```
int dagda_put_data(void* value, diet_data_type_t type,
                   diet_base_type_t base_type,
                   diet_persistence_mode_t mode,
                   size_t nb_r, size_t nb_c,
                   diet_matrix_order_t order,
                   char* path, char** ID);
```

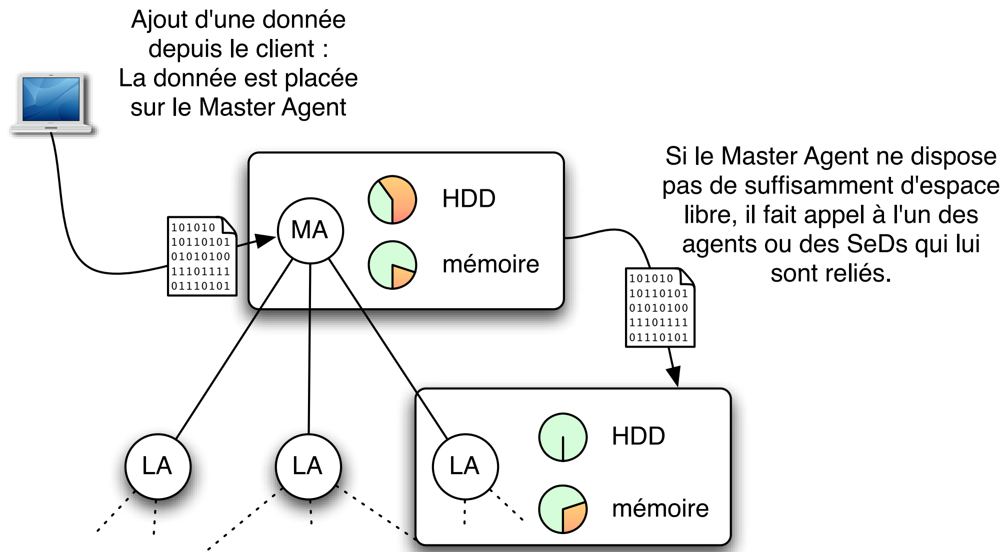

FIG. 4.4 – Ajout d'une donnée par l'intermédiaire de l'API au niveau du client.

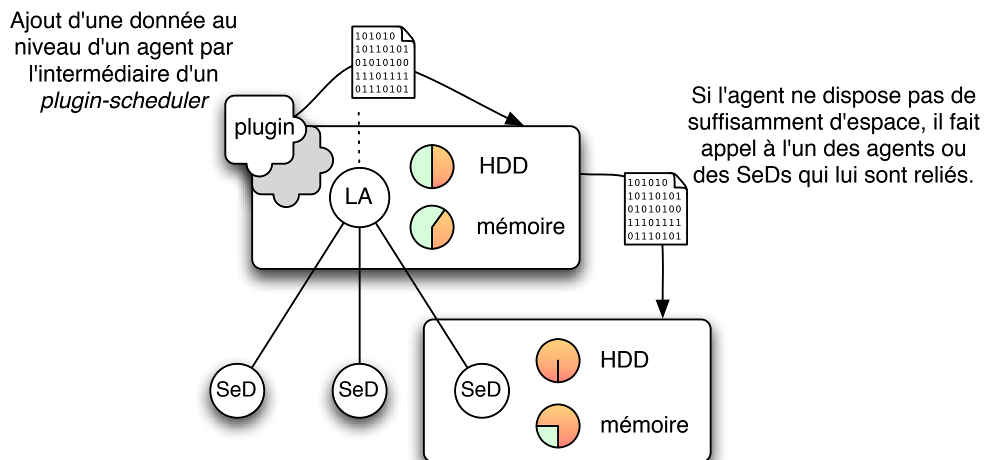

FIG. 4.5 – Ajout d'une donnée par l'intermédiaire de l'API au niveau d'un agent.

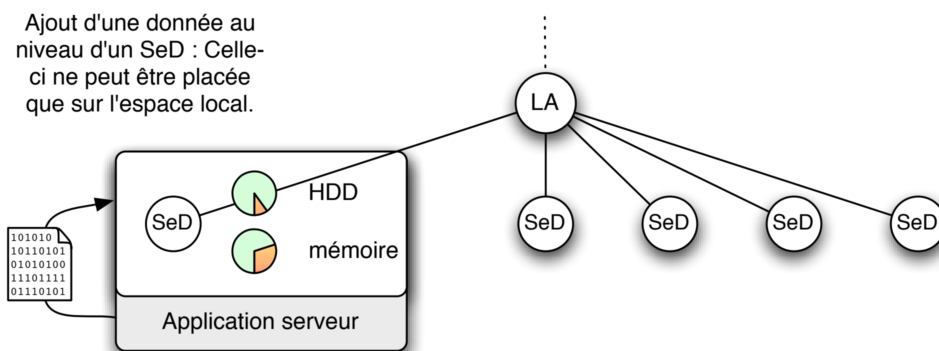

FIG. 4.6 – Ajout d'une donnée par l'intermédiaire de l'API au niveau d'un SeD.

Cette fonction permet d'ajouter à la plateforme une donnée de type `base_type` (DIET_SCALAR, DIET_MATRIX, DIET_VECTOR, DIET_FILE ou DIET_STRING). Le type de base de la donnée étant précisé par le paramètre `base_type` (DIET_CHAR, DIET_SHORT, DIET_INT, DIET_FLOAT, DIET_LONGINT ou DIET_DOUBLE). L'utilisateur doit également préciser le type de persistance souhaité, à savoir DIET_PERSISTENT ou DIET_STICKY. Les paramètres restants dépendent du type de données à ajouter. Ainsi, il faudra préciser les tailles des vecteurs et des matrices ou encore le chemin d'accès à un fichier. Le dernier paramètre étant un pointeur sur une chaîne de caractères qui contiendra l'identifiant de la donnée après le retour de la fonction.

Afin de faciliter l'utilisation de cette fonction et pour conserver une certaine homogénéité dans l'API de DIET, nous avons défini les macros suivantes :

```
dagda_put_scalar(value, base_type, mode, ID)

dagda_put_vector(value, base_type, mode, size, ID)

dagda_put_matrix(value, base_type, mode, nb_rows, nb_cols, \
                 order, ID)

dagda_put_string(value, mode, ID)

dagda_put_file(path, mode, ID)
```

Chacune d'elle ne nécessite que les paramètres spécifiques au type de donnée utilisé.

Ajout asynchrone des données :

Afin de permettre des ajouts de données en parallèle ou d'effectuer du recouvrement calculs/transferts de données, DAGDA propose une version asynchrone de chaque fonction ou macro d'ajout d'une donnée :

```
unsigned int dagda_put_data_async(void* value,
                                  diet_data_type_t type,
                                  diet_base_type_t base_type,
                                  diet_persistence_mode_t mode,
                                  size_t nb_r, size_t nb_c,
                                  diet_matrix_order_t order,
                                  char* path);
```

Les paramètres de cette fonction sont identiques à ceux utilisés pour la fonction synchrone à l'exception près que l'identifiant de la donnée n'est pas initialisé par la fonction. Pour pouvoir obtenir cet identifiant, l'utilisateur récupère au retour de la fonction un identifiant de transfert en valeur de retour de celle-ci. Cet identifiant est alors utilisé comme paramètre de la fonction d'attente de fin de transfert. C'est cette dernière qui initialise l'identifiant de la donnée une fois le transfert effectué.

Les macros spécifiques à l'ajout de chaque type de donnée sont également définies, suivant le même modèle que leur version synchrone :

```
dagda_put_scalar_async(value, base_type, mode)
```

```
dagda_put_vector_async(value, base_type, mode, size)

dagda_put_matrix_async(value, base_type, mode, nb_rows, \
                       nb_cols, order)

dagda_put_string_async(value, mode)

dagda_put_paramstring_async(value, mode)

dagda_put_file_async(path, mode)
```

La récupération de l'identifiant de la donnée ajoutée étant alors obtenu par l'appel à la fonction suivante :

```
int dagda_wait_put(unsigned int transferID, char** ID);
```

On passe en premier paramètre l'identifiant de transfert retourné par la fonction ou la macro d'ajout en asynchrone. La fonction attend la fin du transfert correspondant et initialise alors le paramètre `ID` avec l'identifiant de la donnée concernée.

Récupération de données sur la grille

Une fois les données placées sur la grille, on dispose de deux moyens distincts de les utiliser :

- Comme une donnée persistante dans un appel de service classique. On utilise alors la fonction `diet_use_data(diet_arg_t* arg, char* id)` pour définir le paramètre de l'appel (`arg`) comme étant la donnée d'identifiant `id`. Un simple `diet_call` provoquant alors la récupération de la donnée sur le nœud sélectionné.
- En les récupérant grâce à l'API DAGDA. Plusieurs fonctions sont définies dans l'API permettant de récupérer les données de manière synchrone comme asynchrone.

Il est important de noter que contrairement à DTM, la récupération d'une donnée sur un nœud ne provoque pas sa destruction sur le nœud qui la fournit. Ainsi, chaque utilisation d'une donnée sur un nœud qui n'en dispose pas provoque la création d'un replicat de celle-ci.

La récupération de la donnée nécessite d'abord de localiser ses différents réplicats puis d'en sélectionner un comme source du téléchargement. DAGDA effectue ces tâches de manière transparente pour l'utilisateur en constituant une liste des sources possibles pour une donnée et en tentant de déterminer parmi celles-ci quelle est la plus appropriée. Dans la version de DAGDA fournie avec DIET, DAGDA base ses décisions sur les statistiques de transferts entre le nœud demandeur et les autres nœuds. Ces statistiques sont collectées au fur et à mesure des transferts. Les futures versions de DAGDA permettront l'externalisation de la méthode de choix de la "meilleure source" permettant d'implanter des mécanismes plus adaptés comme des mesures temps réel, des estimations de la bande passante disponible ou encore l'application d'algorithmes prenant en compte la priorité du transfert ou l'ordonnancement de ceux-ci. Par ailleurs, si en pratique le temps de sélection de la source de la donnée est négligeable, on peut légitimement se demander si cette méthode assure le passage à une échelle d'un ordre de grandeur supérieur en nombre de nœuds. Nous avons

pour cela déjà mis en œuvre plusieurs méthodes de construction de la liste des sources, limitant le nombre des résultats transmis à l'algorithme de sélection. Une première méthode qui construit la liste par "proximité" dans la hiérarchie :

1. Le composant DAGDA local commence par contrôler si la donnée n'est pas déjà présente sur le nœud. Dans ce cas, il retourne une liste ne contenant que sa propre référence.
2. Si le nœud n'est pas disponible sur place, il interroge l'ensemble de ses nœuds "fils" dans la hiérarchie par un appel récursif à cette même méthode. Il retourne alors la liste si celle-ci contient au moins un nœud disposant de la donnée.
3. Si la liste obtenue par ces appels récursifs est vide, il fait alors appel à son nœud "père" qui effectuera à son tour la même opération.

Et une seconde méthode calquée sur le fonctionnement de la sélection du nœud de calcul par DIET :

1. Chaque agent aggrège les listes des sources de la donnée retournées par les agents ou les SeDs situés au niveau inférieur de la hiérarchie, ne retenant que les n meilleures sources en fonction des statistiques locales de transfert entre le nœud source et le nœud destination.
2. La sélection de la meilleure source pour la donnée s'effectuant alors sur une liste d'au plus n éléments (n étant paramétrable librement) au niveau du nœud "demandeur".

En pratique, nous n'avons pas pu relever de différence notable entre ces deux méthodes, aussi bien dans les temps de transferts relevés que dans le temps de sélection du nœud, et cela pour des déploiements de plusieurs centaines de nœuds répartis sur plusieurs sites de la plateforme Grid'5000 avec des données de plusieurs giga-octets.

La figure 4.7 présente le fonctionnement de la récupération d'une donnée sur la grille par l'intermédiaire de l'API DAGDA.

Récupération synchrone des données :

```
int dagda_get_data(char* dataID, void** value,
                   diet_data_type_t type,
                   diet_base_type_t* base_type,
                   size_t* nb_r, size_t* nb_c,
                   diet_matrix_order_t* order,
                   char** path);
```

Cette fonction récupère la donnée d'identifiant `dataID` ainsi que toutes les informations nécessaires à son utilisation. Ainsi, si la donnée est une matrice, la fonction initialisera les variables `nb_r`, `nb_c` et `order`, s'il s'agit d'un fichier, la chaîne de caractères `path` contiendra le chemin vers le fichier etc. Le passage d'un pointeur `NULL` permet de ne pas récupérer les informations qui ne sont pas pertinentes. Ceci permet de définir les macros suivantes, qui donnent à l'API une certaine homogénéité d'utilisation :

```
dagda_get_scalar(ID, value, base_type)

dagda_get_vector(ID, value, base_type, size)
```

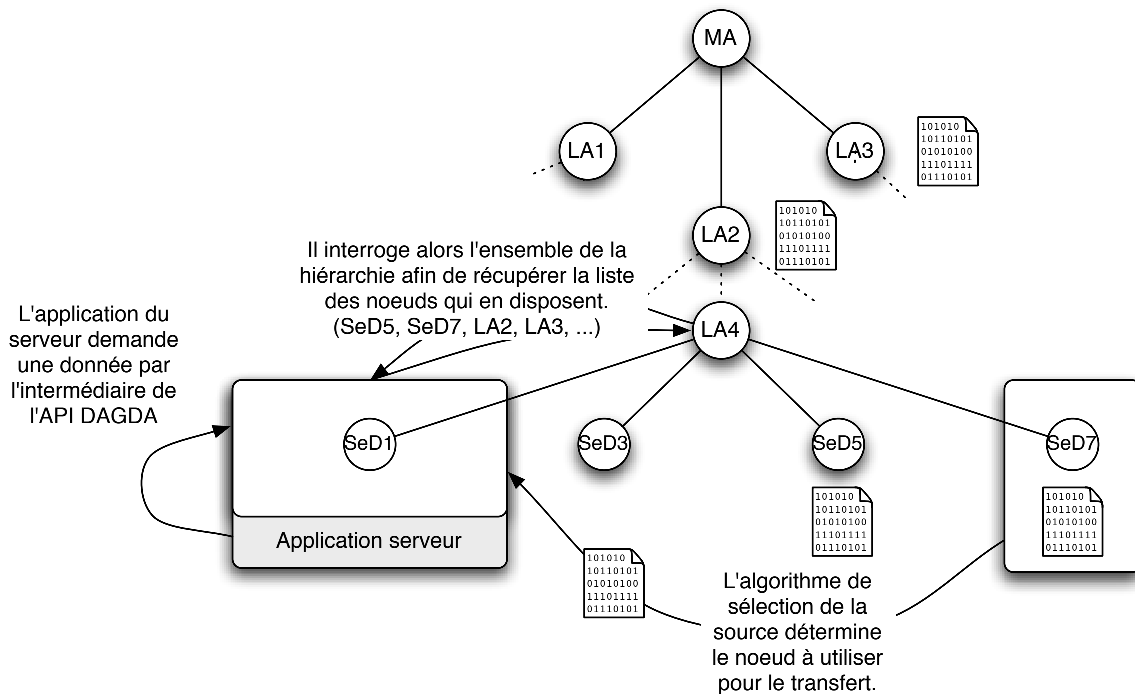

FIG. 4.7 – Récupération d'une donnée avec l'API DAGDA.

```
dagda_get_matrix(ID, value, base_type, nb_r, nb_c, order)
```

```
dagda_get_string(ID, value)
```

```
dagda_get_file(ID, path)
```

Chacune de ces fonctions/macros attend la fin du transfert avant de rendre la main à l'utilisateur. Afin de permettre des transferts en parallèle ou le recouvrement calcul/transferts, l'API DAGDA propose pour chacune d'elles une version asynchrone.

Récupération asynchrone des données :

Le prototype de ces fonctions/macros est calqué le modèle utilisé pour l'ajout de données en asynchrone :

```
unsigned int dagda_get_data_async(char* ID, diet_data_type_t type);
```

La fonction ne prends en paramètre que l'identifiant de la donnée et le type de celle-ci afin de filtrer les informations utiles à récupérer. Tout comme la fonction d'ajout asynchrone elle retourne un identifiant de transfert permettant d'attendre la fin de celui-ci. Les macros définies dans l'API n'ont ici pour seul objectif que l'homogénéité du code utilisant l'API DAGDA. En effet, il ne s'agit ici que de définir la variable `type` en fonction de la donnée à récupérer. Elles sont ainsi définies comme suit :

```
dagda_get_scalar_async(ID)
```

```
dagda_get_vector_async(ID)

dagda_get_matrix_async(ID)

dagda_get_string_async(ID)

dagda_get_file_async(ID)
```

Afin de permettre l'attente de la fin du téléchargement de la donnée et la récupération de celle-ci ainsi que des informations qui l'accompagnent, l'API définit la méthode suivante :

```
int dagda_wait_get(unsigned int transferID, void** value,
                   diet_base_type_t* base_type,
                   size_t* nb_r, size_t* nb_c,
                   diet_matrix_order_t* order,
                   char** path);
```

Un identifiant de transfert est passé comme premier paramètre, il s'agit de la valeur retournée par une des fonctions/macros `dagda_get_*_async`. Au retour de la fonction, si aucune erreur ne s'est produite, la variable `value` contient un pointeur vers la donnée si celle-ci est stockée en mémoire, les variables `base_type`, `nb_r`, `nb_c`, `order` et `path` contiennent quant à elles les informations concernant la donnée en fonction du type de celle-ci.

Chargement asynchrone des données :

Pour effectuer un transfert de données de manière asynchrone, DAGDA crée un "processus léger" (Thread) qui se charge d'effectuer celui-ci, avec la possibilité d'attendre la fin de son exécution. Pour cela, DAGDA garde une référence sur le *thread*. Il n'est alors "détruit" en mémoire que lorsque son exécution est achevée et que l'utilisateur a fait appel à la fonction "wait" correspondante. Si ce système correspond bien à ce que l'on peut souhaiter pour réaliser des transferts simultanés ou du recouvrement transferts/calculs, il pose un problème pour la mise en œuvre de techniques de "pré-chargement" (*prefetching*) sur les nœuds de la grille. En effet, DAGDA permet de lancer un chargement de données depuis un plugin-scheduler, aussi bien au niveau des SeDs qu'au niveau des agents afin de charger par avance une donnée dont on peut supposer qu'elle sera utile. Or, dans ce cas, il devient nécessaire de lancer le chargement et de rendre la main au système sans attendre que celui-ci soit achevé, faute de quoi, le pré-chargement perds tout son sens. Ainsi pour permettre la mise en œuvre de ces techniques dans DIET sans voir le nombre de processus de transfert sans cesse grossir sans possibilité d'en éliminer, DAGDA propose des fonctions et macros équivalentes au chargement des données asynchrones qui libèrent automatiquement la mémoire utilisée par les processus. Il s'agit alors de "charger" les données sur le nœud considéré en vue d'une potentielle utilisation ultérieure. Une fonction de l'API permet de réaliser cette tâche :

```
int dagda_load_data(char* ID);
```

Elle prend en argument l'identifiant de la donnée qui doit être chargée par le composant DAGDA et rend la main immédiatement à l'utilisateur.

Partage des données

Afin de permettre un partage simplifié des données, DAGDA propose un système d'alias permettant l'utilisation de noms plus expressifs pour retrouver l'identifiant unique d'une donnée. Ce mécanisme est centralisé et définitif, un alias défini ne pouvant être réutilisé, et a pour but essentiel le partage d'un petit ensemble de données entre plusieurs clients. Il est notamment utilisé par DIET-BLAST afin de permettre l'utilisation des bases de données par plusieurs clients sans communication entre eux. Il s'agit d'une première approche très simple de l'implantation d'un catalogue de données dans DAGDA. Deux fonctions sont définies dans l'API :

```
int dagda_data_alias(const char* id, const char* alias);
```

Cette fonction tente d'associer l'alias `alias` à l'identifiant `id`. Si l'alias est déjà utilisé, la fonction retourne une valeur différente de 0.

```
int dagda_id_from_alias(const char* alias, char** id);
```

Cette fonction, à l'inverse tente de récupérer l'identifiant de la donnée dont l'alias est `alias`. Si l'alias n'est pas enregistré sur la plateforme, elle retourne une valeur différente de 0.

On voit clairement qu'un alias ne peut correspondre qu'à une seule donnée, mais que rien n'empêche de définir plusieurs alias pour une même donnée.

Sauvegarde de la plateforme (Checkpointing)

Chaque nœud de la plateforme DIET peut être configuré afin de permettre l'enregistrement sur disque des données enregistrées localement. Ainsi, si la plateforme doit être arrêtée ou qu'un problème intervient sur celle-ci, il est possible de la redémarrer en restaurant les données telles qu'elles étaient au moment du dernier enregistrement (Ce genre de système est connu sous le nom de "Checkpointing"). C'est l'utilisateur par l'intermédiaire de l'API qui décide du moment de l'enregistrement de ces données. Pour cela, l'API DAGDA fournit la fonction suivante :

```
int dagda_save_platform();
```

La restauration des données au démarrage n'est réalisée que lorsque les nœuds ont été configurés dans cette optique. La configuration de chacun des nœuds permet ainsi de limiter le nombre des nœuds permettant une telle sauvegarde aux seuls nœuds choisis. Par ailleurs, les fichiers de sauvegarde ne sont pas spécifiquement attachés à un nœud de la hiérarchie, ce qui permet de modifier la topologie du déploiement ou d'utiliser un même fichier pour un ensemble de nœuds.

La réplication explicite des données

On a vu que la récupération d'une donnée provoque la création d'un réplicat de celle-ci lorsqu'elle n'est pas présente sur le nœud. Qu'il s'agisse de la récupération effectuée par DIET dans le cadre d'un appel de service ou de la récupération par l'intermédiaire de l'API DAGDA, celle-ci s'effectue de manière décentralisée sur les SeDs sélectionnés pour la résolution d'un problème. Cette réplication "implicite" est donc commandée par l'algorithme d'ordonnancement. Or, on a vu qu'il est parfois utile d'inverser cette dépendance et

de lier l'ordonnancement au placement de données et non l'inverse. Pour cela DAGDA propose une fonction de réplication explicite des données. Cette fonction permet de répliquer une donnée déjà présente sur la plateforme en choisissant les nœuds de destination par la définition d'une "règle" de réplication. Dans un premier temps, DAGDA propose de définir ces règles de manière très simple mais permettant de réaliser les réplications nécessaires aux algorithmes étudiés. Il est envisageable de pouvoir définir des règles plus complexes, c'est pourquoi la fonction de réplication proposée prend en paramètre une chaîne de caractères définissant la règle à utiliser, laissant envisager une évolution de celles-ci sans changement dans l'API proposée. La fonction de réplication dans DAGDA est définie comme suis :

```
int dagda_replicate_data(const char* id, const char* rule);
```

Le paramètre `id` représente l'identifiant de la donnée à répliquer et le paramètre `rule`, la règle à appliquer pour la réplication des données. La syntaxe des règles de réplication de la version actuelle de DAGDA est la suivante :
<cible de la description> :<description> :<comportement de remplacement>

- La *cible de la description* est l'identifiant du nœud à utiliser pour la comparaison avec la description. Il peut s'agir du nom d'hôte réseau de la machine ou de l'identifiant du composant DAGDA utilisé par CORBA. On utilisera alors respectivement les mots clés "host" ou "ID" pour désigner quel identifiant utiliser.
- La *description* est un schéma de comparaison à utiliser avec l'identifiant choisi, permettant l'utilisation des caractères "Joker" habituels (`*`, `?` ou `[]`). Ainsi, *"*.lyon.grid5000.fr"*, *"capricorne-[0-5][0-9].*"* ou encore *"*SeD*"* sont des descriptions valides. On peut bien sûr parfaitement utiliser une chaîne sans caractère joker pour désigner un nœud particulier.
- Le *comportement de remplacement*, dernier élément de la description, indique si DAGDA doit utiliser l'algorithme de remplacement de données lorsque l'espace disponible sur le nœud est insuffisant pour recevoir la donnée. *"replace"* indique que DAGDA peut faire appel à l'algorithme de remplacement choisi sur le nœud si nécessaire et *"noreplace"* que si l'espace est insuffisant, on renonce à la réplication sur le nœud considéré.

On pourra par exemple définir les règles de réplication suivantes :
- *"host :*.lyon.grid5000.fr :replace"* : Réplication sur tous les nœuds dont le nom d'hôte fini par *.lyon.grid5000.fr* avec le remplacement éventuel d'une donnée sur les sites considérés.
- *"ID :*LA-[0-5]* :noreplace"* : Réplication sur tous les nœuds dont l'identifiant du composant DAGDA contient *"LA-"* suivi d'un nombre entre 0 et 5 sans faire appel au remplacement de donnée si l'espace disponible est insuffisant.
- *"host :paraquad-55.rennes.grid5000.fr :replace"* : Réplication sur le nœud de nom d'hôte *"paraquad-55.rennes.grid5000.fr"*.

L'appel à cette fonction déclenche la réplication sur les nœuds lors d'un parcours complet de la hiérarchie DIET. Chacune de ces réplications est réalisée en asynchrone. La figure 4.8 présente un exemple de réplications réalisées par l'utilisation de la fonction `dagda_replicate_data`.

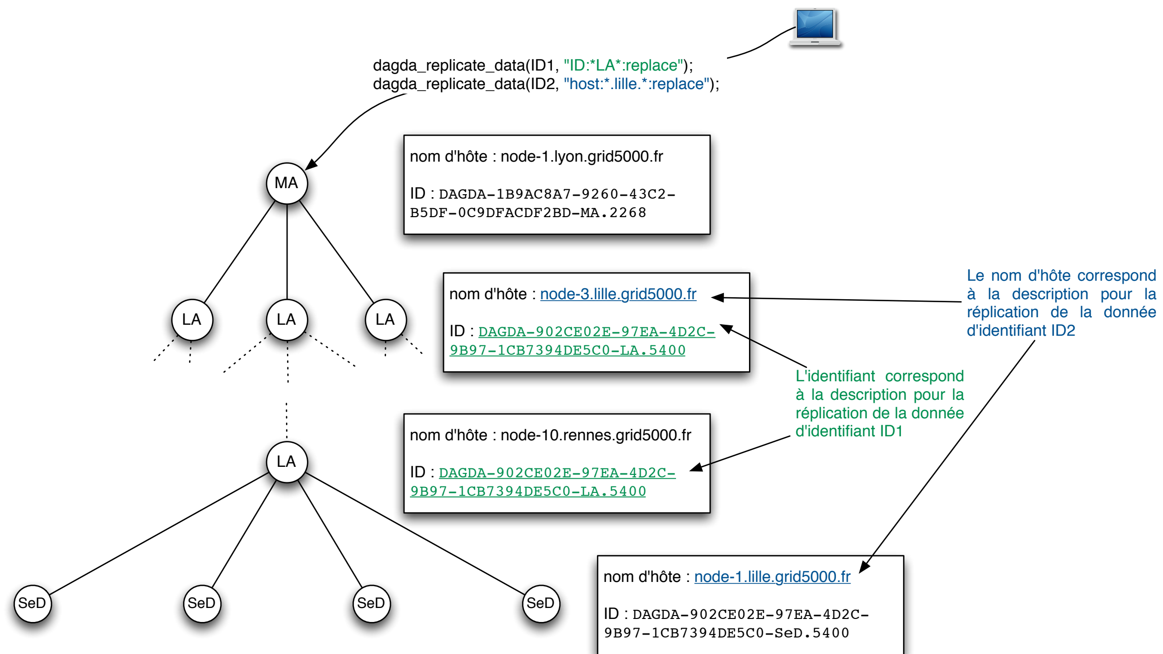

FIG. 4.8 – Exemple d'utilisation de la fonction de réplication de l'API DAGDA.

4.3 Utilisation de DAGDA pour la gestion de données

Les différentes fonctions présentées dans la section précédente permettent une gestion avancée des données sur la grille. Nous présentons ici quelques cas d'utilisations possibles de l'API DAGDA pour la gestion de données et leur utilisation par des services DIET.

4.3.1 Utilisation d'ensembles de données

Dans le fonctionnement classique de DIET, un profil décrit les paramètres d'un problème à résoudre. Cette définition statique du nombre et du type des paramètres limite l'utilisation de DIET aux seuls problèmes qui ont un nombre d'entrées et de sorties constants. Il est toujours possible de regrouper plusieurs entrées et plusieurs sorties dans une même donnée, mais il faut alors effectuer des tâches pas forcément triviales de fusion et d'extraction de données du côté client comme du côté serveur. De plus, les données ainsi regroupées sont obligatoirement gérées comme une unique entité. Il est alors impossible de répliquer une donnée particulière de l'ensemble sans répliquer les autres. L'utilisation de DAGDA permet d'éviter ce genre de problème en ajoutant individuellement des données sur la plateforme et en manipulant chacune d'entre elles séparément. La figure 4.9 présente ce cas de figure avec une gestion des données fusionnées dans une utilisation classique de DIET sans DAGDA et la figure 4.10 présente la même opération réalisée à l'aide de DAGDA.

4.3.2 Réplication des données

Le système de réplication des données offert par DAGDA permet d'améliorer les performances d'exécutions en parallèle sur la plateforme pour le peu que l'algorithme d'ordonnan-

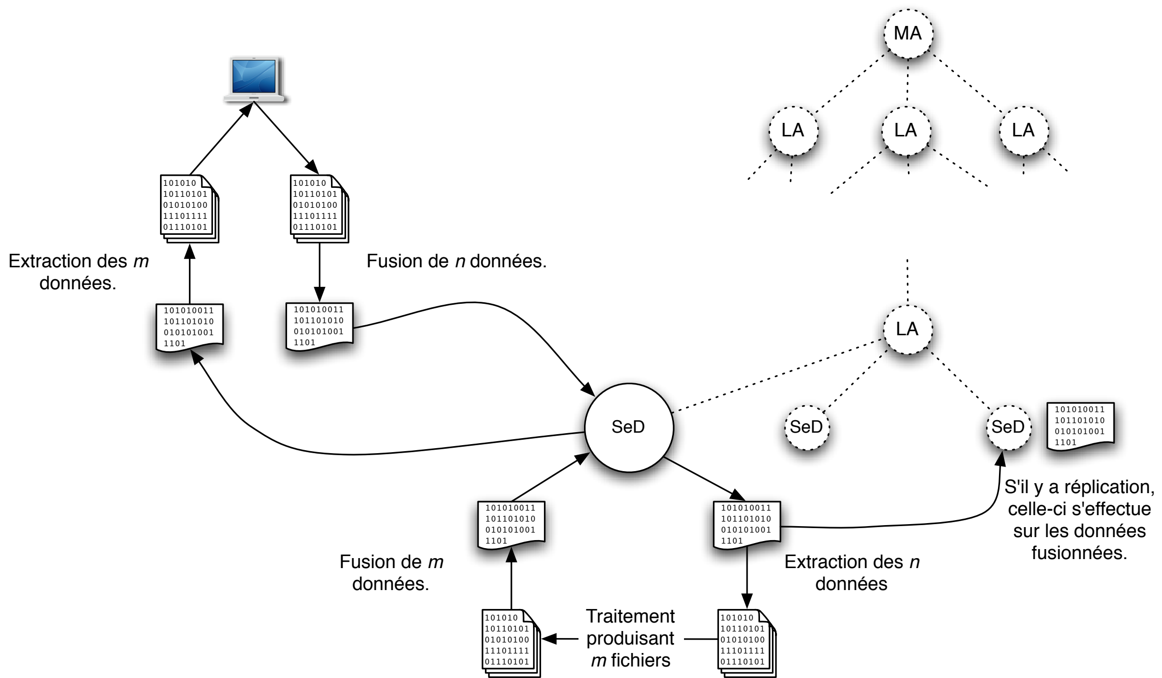

FIG. 4.9 – Utilisation d'un ensemble de données dont le nombre est déterminé dynamiquement avec DIET.

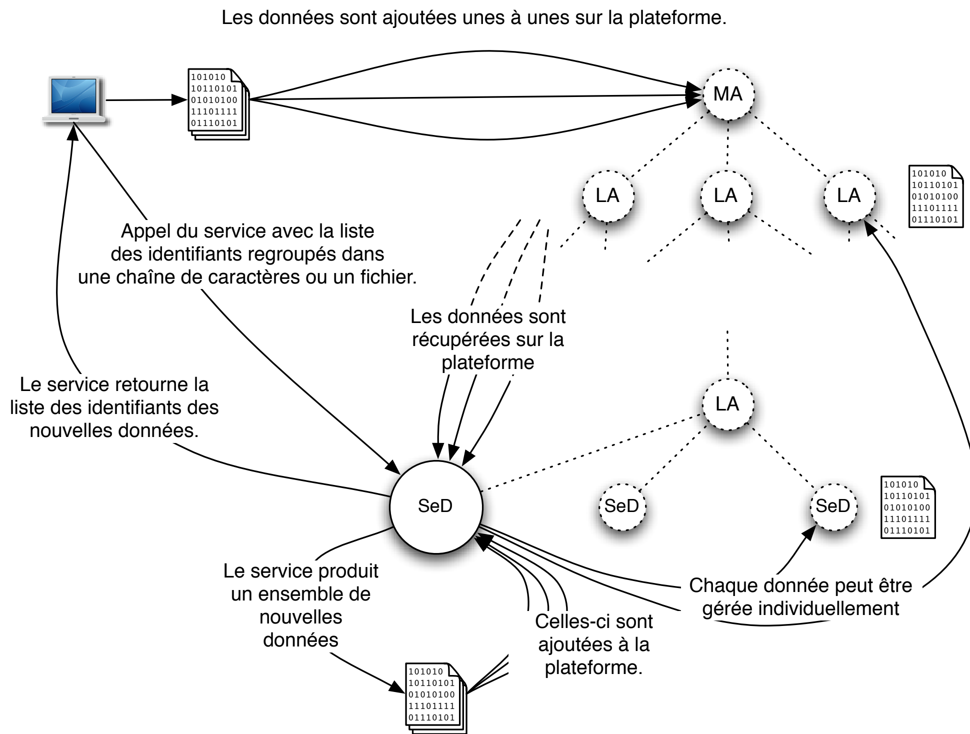

FIG. 4.10 – Utilisation d'un ensemble de données dont le nombre est déterminé dynamiquement avec DAGDA.

cement utilisé prenne en compte la localité des données. La figure 4.11 présente un exemple d'utilisation de la réplication avec un algorithme Round-Robin qui limite la liste des serveurs proposant le service demandé aux seuls disposant déjà de la donnée. Dans l'exemple

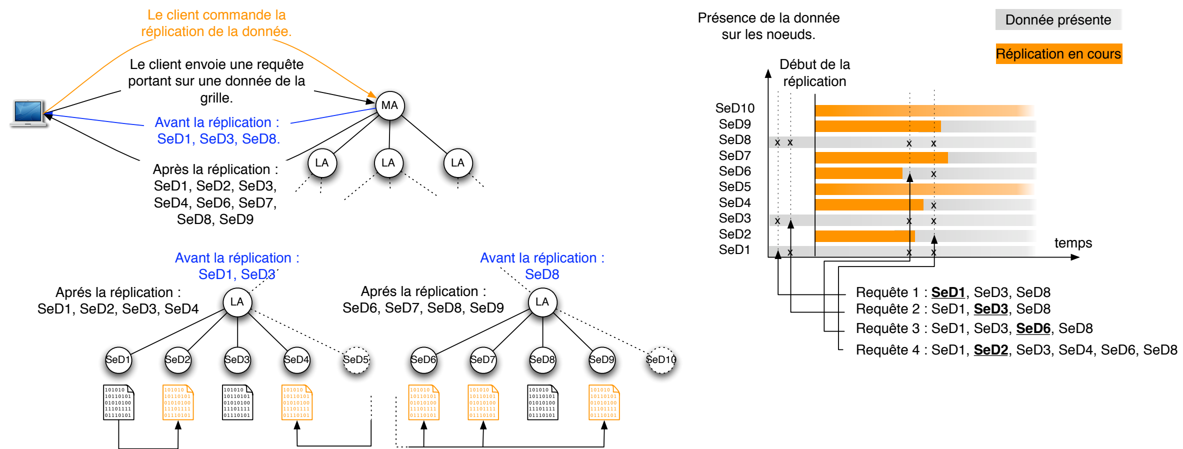

FIG. 4.11 – Réplication des données et ordonnancement Round-Robin.

de la figure 4.11, les données sont répliquées au fur et à mesure du temps, en fonction de la bande passante disponible. Pour les deux premières requêtes, l'ordonnanceur n'a le choix qu'entre trois nœuds (SeD1, SeD3 et SeD8). Il effectue donc l'ordonnancement d'abord sur le serveur SeD1 puis SeD3 suivant la politique Round-Robin. À la troisième requête, le SeD6 vient s'ajouter aux nœuds disposant de la donnée et est donc pris en compte dans l'ordonnancement. Enfin, à la quatrième requête, les serveurs SeD2 et SeD4 ont achevé la réplication de la donnée, c'est le SeD2 qui est donc choisi. En effet, le Round-Robin est basé sur une liste ordonnée des SeDs qui disposent de la donnée en maintenant un compteur du nombre de requêtes traitées sur le nœud.

Dans le cas des applications qui nous intéressent, un grand nombre de tâches nécessitant la même donnée sont soumises à la grille. La réplication des données permet alors d'utiliser plusieurs nœuds pour réaliser les requêtes en parallèle lorsque l'algorithme d'ordonnancement prend en compte la localité des données. C'est le cas du Round-Robin présenté dans l'exemple, mais aussi de l'algorithme de temps de complétion minimum (MCT) ou encore de l'algorithme SRA dynamique présenté plus loin.

4.3.3 Recouvrement des tâches de calcul et de transferts

En proposant une gestion asynchrone des données, DAGDA permet de réaliser du recouvrement entre les tâches de calcul et les tâches de transfert de données. DAGDA permet par exemple de concevoir des services qui enchaînent deux calculs sur des données différentes en profitant du temps du premier calcul pour récupérer la donnée nécessaire au second. Côté client, il est également possible de répliquer une donnée dont on sait qu'elle sera utilisée par la suite. Les figures 4.12 et 4.13 présentent deux exemples de recouvrement des tâches de calcul et des transferts en utilisant DAGDA.

Dans l'exemple de la figure 4.12, c'est le client qui anticipe l'utilisation de la donnée et effectue le transfert de celle-ci sur la plateforme. Le premier service (multiplication matricielle) est appelé en asynchrone et suivi de la réplication de la donnée vers le nœud proposant le

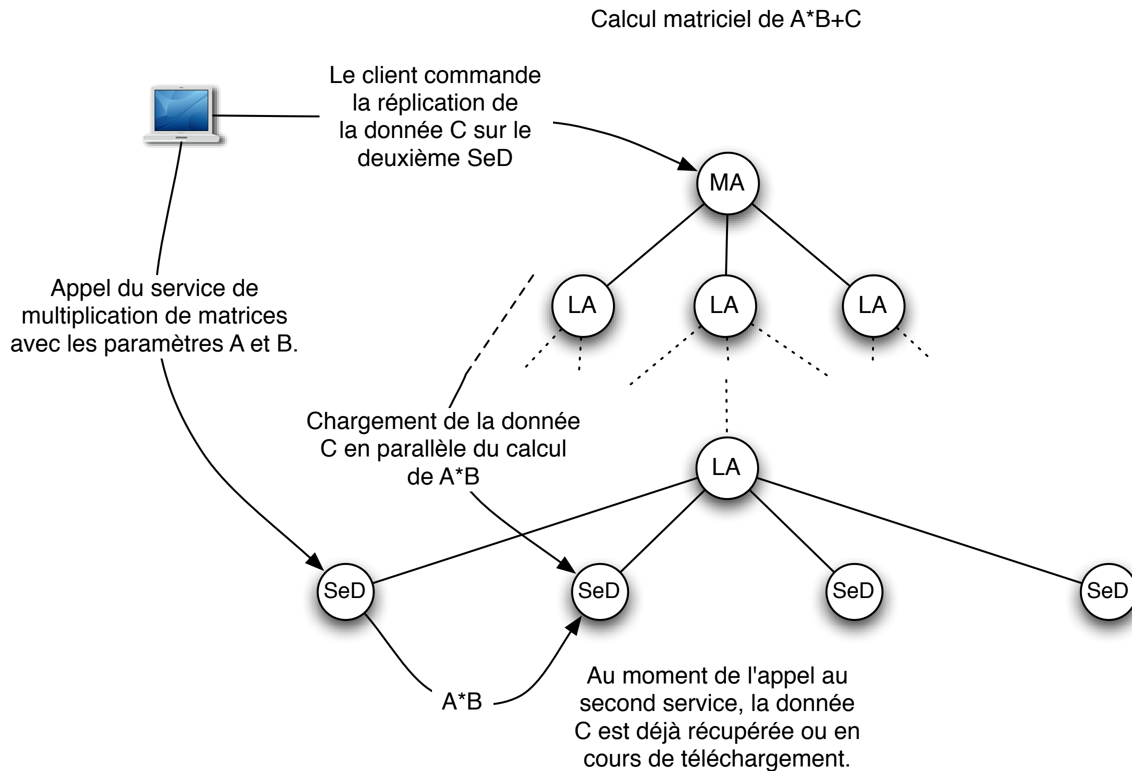

FIG. 4.12 – Recouvrement temps de calcul / temps de transfert avec DAGDA. Utilisation de deux SeDs.

second service (addition matricielle). Une fois le premier service achevé, la donnée est déjà présente ou en cours de chargement sur le second nœud. Ici, c'est le client qui procède à l'optimisation des transferts.

Dans l'exemple de la figure 4.13, le serveur lance le chargement de la donnée nécessaire à la deuxième opération avant de procéder au calcul de la première. En utilisant la récupération des données en asynchrone, le calcul et le téléchargement s'effectuent en asynchrone. À la fin du premier calcul, la donnée est déjà présente, ou au moins en partie déjà récupérée. Dans cet exemple, c'est le serveur qui effectue l'optimisation des transferts.

Le recouvrement des tâches de transferts et de calcul nécessite de connaître par avance le déroulement des opérations. C'est donc, soit le client qui anticipe ses prochains appels aux services, soit le serveur qui effectue plusieurs opérations et peut ainsi anticiper les chargements des données. DAGDA permet ainsi aux développeurs d'applications de réaliser des optimisations spécifiques à leurs applications. Cependant, l'optimisation côté client nécessite des connaissances sur la plateforme, réduisant par la même la transparence d'accès à la grille et l'optimisation côté serveur nécessite la création de services effectuant plusieurs calculs réduisant potentiellement l'efficacité globale de la plateforme. Par ailleurs, le transfert des données ne s'effectue en partie plus par l'intermédiaire d'un appel GridRPC standard. Ce dernier point a conduit a notre proposition d'extension de l'API GridRPC auprès de l'OGF (voir Annexe A) qui, si elle est acceptée permettra de prendre en compte ce type d'optimisations, certes spécifiques aux applications, mais qui doivent néanmoins être au moins possibles pour les développeurs qui choisissent d'utiliser ce standard.

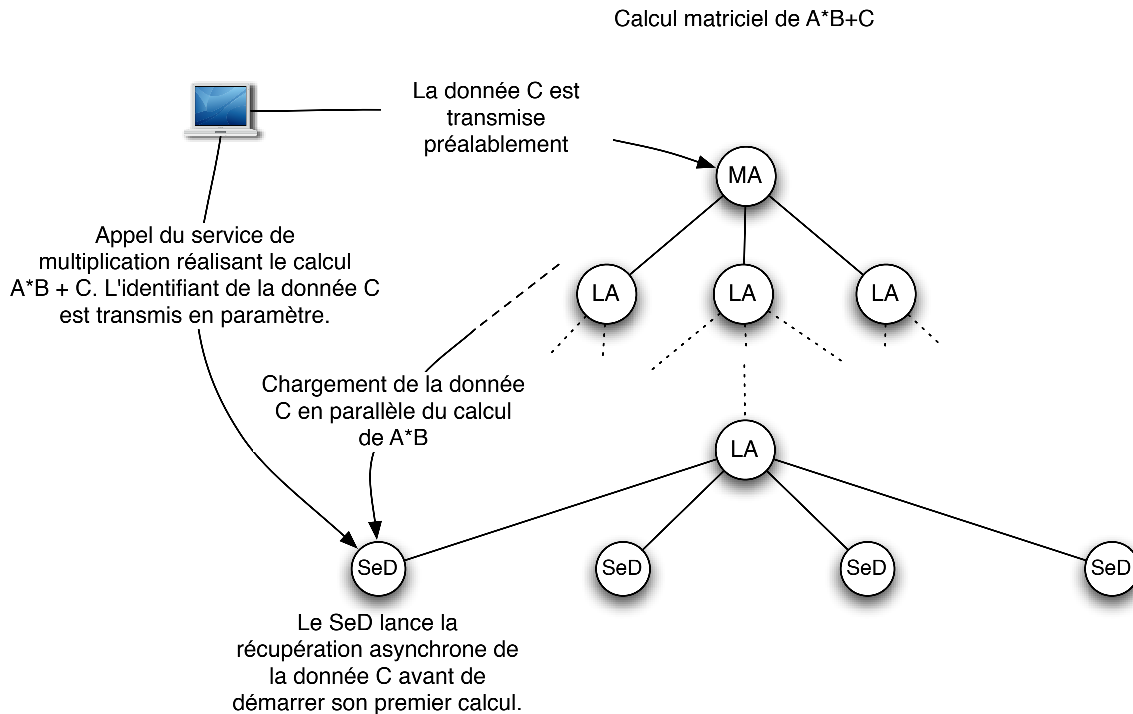

FIG. 4.13 – Recouvrement temps de calcul / temps de transfert avec DAGDA. Un seul SeD enchaîne les deux calculs.

Il est souvent impossible de connaître à l'avance l'enchaînement des services et le nœud qui sera sélectionné pour les exécuter. Ceci est d'autant plus vrai sur une plateforme de grande échelle mise à la disposition de nombreuses personnes qui utilisent les mêmes données. Par exemple, sur une grille dédiée à la bio-informatique, on peut supposer que de nombreux utilisateurs auront besoin des mêmes bases de données de référence et des mêmes services qui pourront donc être appelés dans n'importe quel ordre et à n'importe quel moment. Ainsi, le recouvrement explicite des temps de calculs et des temps de transferts, s'il peut s'avérer utile, peut être difficile à mettre en œuvre de manière efficace. Le mécanisme d'ordonnanceurs *plugin* proposé par DIET, associé à la gestion de données offerte par DAGDA peut permettre de réaliser des optimisations plus systèmatiques et indépendantes de l'application visée. La réplication des données présentée précédemment est une première approche efficace, mais il est également possible d'utiliser des techniques d'optimisations plus fines, déjà utilisées à l'échelle des processeurs (mise en cache de données, ou *pré-chargement*), dans les *Content Distribution Networks* pour l'accès aux pages Web, etc. La section suivante présente les possibilités offertes par DAGDA dans ce domaine.

4.3.4 Pré-chargement des données

Le système d'ordonnanceur "plugin" de DIET allié à l'API DAGDA permet de réaliser du pré-chargement de données sur les nœuds susceptibles d'être sélectionnés par DIET pour la résolution d'un problème. Ce pré-chargement peut-être systèmatique s'il est implanté au niveau d'un SeD ou sélectif si on utilise un ordonnanceur au niveau agent. On peut par exemple décider que les n "meilleurs" nœuds disposant du service peuvent récupérer la

donnée par avance ou décider d'approcher la donnée des SeDs en la récupérant sur un agent. Le système de partage des données entre différents nœuds permet par ailleurs d'effectuer un seul transfert vers un agent de manière à mettre la donnée à disposition de tous les serveurs qu'il prend en charge. Les figures 4.14, 4.15 et 4.16 présentent ces différentes utilisations de DAGDA. La figure 4.14 présente un exemple où les données nécessaires à la résolution du

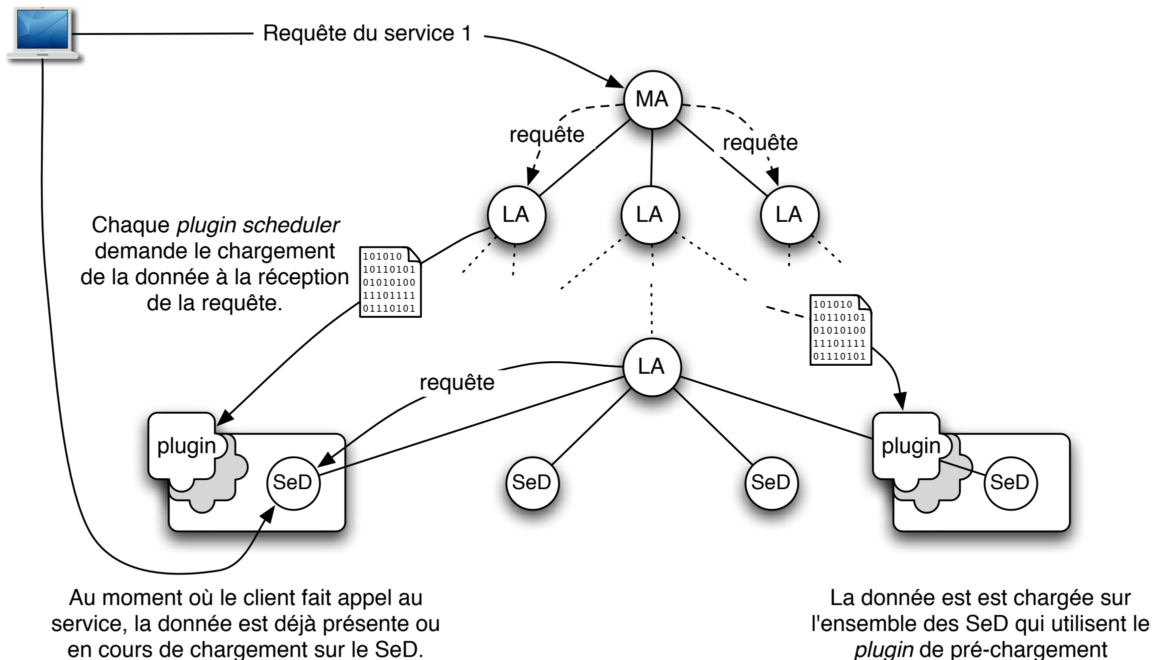

FIG. 4.14 – Pré-chargement systèmatique sur l'ensemble des nœuds offrant le service.

problème sont systèmatiquement chargées sur l'ensemble des SeDs qui dispose du service et qui utilisent l'ordonnanceur de pré-chargement. Lorsque le SeD reçoit la requête il démarre le chargement d'une ou plusieurs données dont l'identifiant a été transmis dans le profil du problème, et ce, sans pouvoir savoir s'il sera sélectionné pour la résolution. Ce mécanisme est particulièrement adapté aux services qui seront appelés de nombreuses fois avec une même donnée et des paramètres différents. Par exemple, un ensemble de séquences à aligner avec une même base de données grâce à l'application BLAST. Ainsi, l'ensemble des serveurs proposant le service BLAST disposeront de la base lors des prochaines requêtes.

Dans l'exemple de la figure 4.15, c'est l'agent qui commande la réplication de la donnée sur un sous-ensemble des nœuds disposant du service demandé. Ce mécanisme permet de limiter les transferts aux nœuds ayant le plus de chances d'être sélectionnés pour la résolution. L'agent ayant une connaissance plus grande de la probabilité du choix d'un nœud pour la résolution, puisque c'est à son niveau que le classement et la sélection des nœuds sont effectués, cette technique de pré-chargement permet d'éviter des transferts inutiles tout en augmentant les chances de présence des données au moment du lancement des services.

L'exemple de la figure 4.16 présente quant à lui une technique qui peut s'apparenter à de la mise en cache de niveau 2. En effet, l'agent récupère la donnée, approchant ainsi celle-ci des SeDs susceptibles d'en demander le chargement pour la résolution du problème. Ce mécanisme suppose bien entendu que la hiérarchie est construite sur un schéma de proximité : Les SeDs sont donnés à la charge d'agents "proches" d'eux, au sens de la proxi-

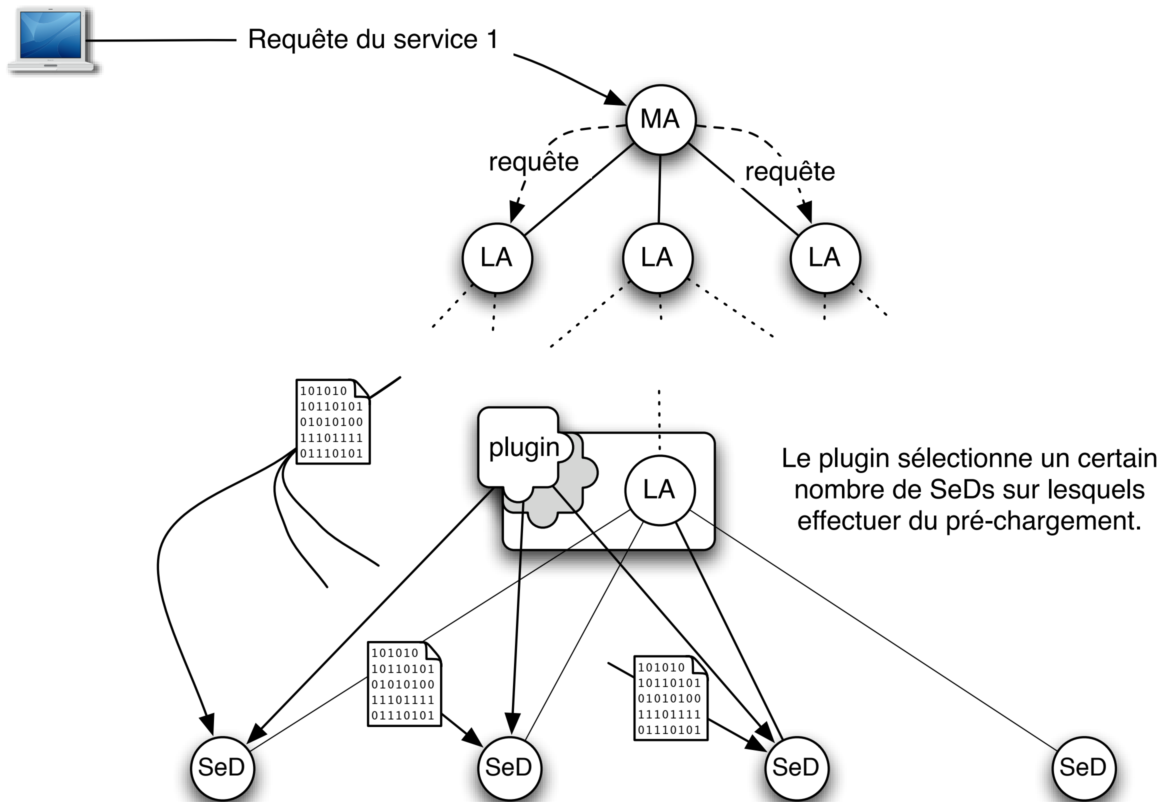

FIG. 4.15 – Pré-chargement sur les nœuds sélectionnés par le plugin de l'agent.

mité réseau en terme de bande passante. DAGDA proposant un mécanisme de partage de données entre les agents et les SeDs qui lui sont reliés, ce mécanisme peut s'avérer très efficace lorsqu'il partage avec les SeDs l'accès à une même partition NFS. Dans ce cas, un seul téléchargement donne l'accès à la donnée à tous les SeDs gérés par l'agent. Il faut cependant s'assurer que le serveur NFS peut supporter la charge des accès concurrents à cette donnée par de nombreux processus en simultané.

4.4 Expérimentations

Dans cette section nous présentons des expérimentations et des mesures de performances réalisées avec DIET en utilisant DAGDA sur la plateforme Grid'5000 [22]. Nous avons ici, testé à la fois les performances "brutes" de DAGDA en terme de vitesse de transmission, mais aussi l'intérêt de la gestion de données offerte par DAGDA grâce à la persistance et la réplication.

4.4.1 Performances des transferts

Dans ces expériences, nous comparons les performances obtenues pour le transfert de fichiers de tailles croissantes avec DIET utilisant le gestionnaire DTM, DIET utilisant le gestionnaire DAGDA et l'outil de transfert sécurisé "scp" (Secure Copy, basé sur le protocole

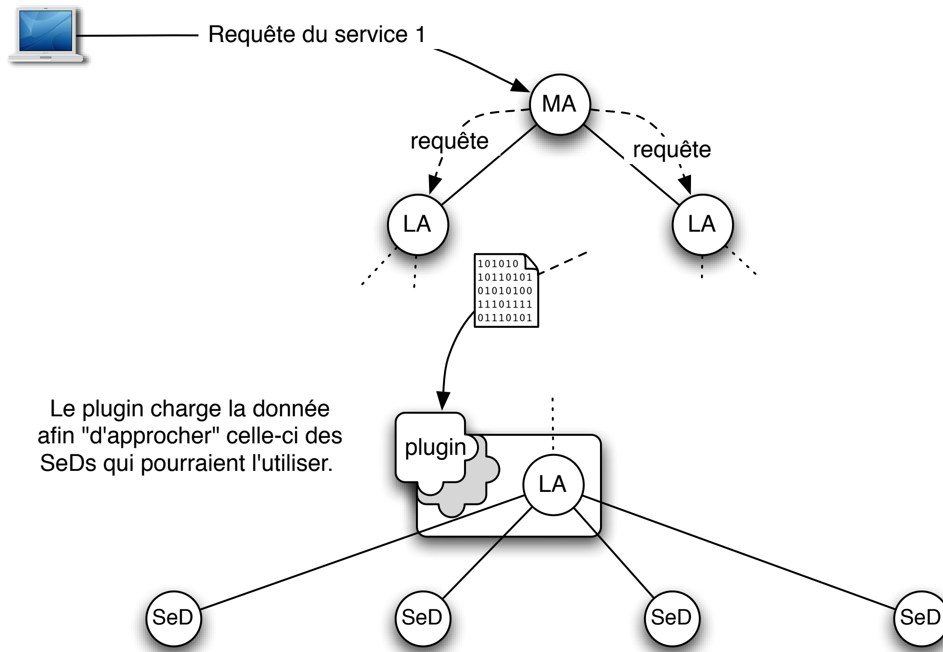

FIG. 4.16 – Pré-chargement sur l'agent afin d'approcher la donnée des nœuds de calcul.

ssh). La figure 4.17 présente les résultats obtenus pour des tailles de fichiers entre 10 méga-octet et 1 giga-octet. Les expériences ont été réalisées entre des nœuds de la plateforme situés à Rennes et à Orsay. Afin de s'assurer que les résultats ne sont pas troublés par d'autres expériences, chacun des transferts à été réalisé dix fois et les résultats présentés sont la moyenne des temps de transfert mesurés.

Nous voyons que les performances obtenues par DAGDA sont très proches de celles du système scp. La différence constatée résidant sûrement dans le temps nécessaire au cryptage des données transmises par scp. Les transferts réalisés par DTM sont fortement pénalisés par le besoin de charger l'intégralité du fichier en mémoire avant l'envoi. L'utilisation de DAGDA permet ainsi d'améliorer les performances globales des applications DIET nécessitant l'échange de fichiers de tailles importantes. On constate ici que les performances de DAGDA sont comparables à celles d'un protocole de transfert très populaire et que l'utilisation de DAGDA peut amener des bénéfices non négligeables aux applications DIET utilisant des données de grande taille.

4.4.2 Persistance des données.

Pour cette expérimentation, nous avons utilisé un service DIET réalisant le comptage d'un caractère dans un fichier. Le service prend ainsi en paramètre, un fichier et un caractère et retourne un entier. Ce service offre l'avantage de pouvoir utiliser une donnée de taille arbitraire, de pouvoir être utilisé plusieurs fois de manière concurrente avec des paramètres différents (le caractère à compter) et d'avoir une complexité exactement linéaire en la taille du fichier transmis. Ainsi, nous pourrons mesurer avec précision l'apport de la persistance des données en minimisant les perturbations dûes aux transmission des paramètres ou d'une complexité variant en fonction du fichier d'entrée. Pour cette expérience,

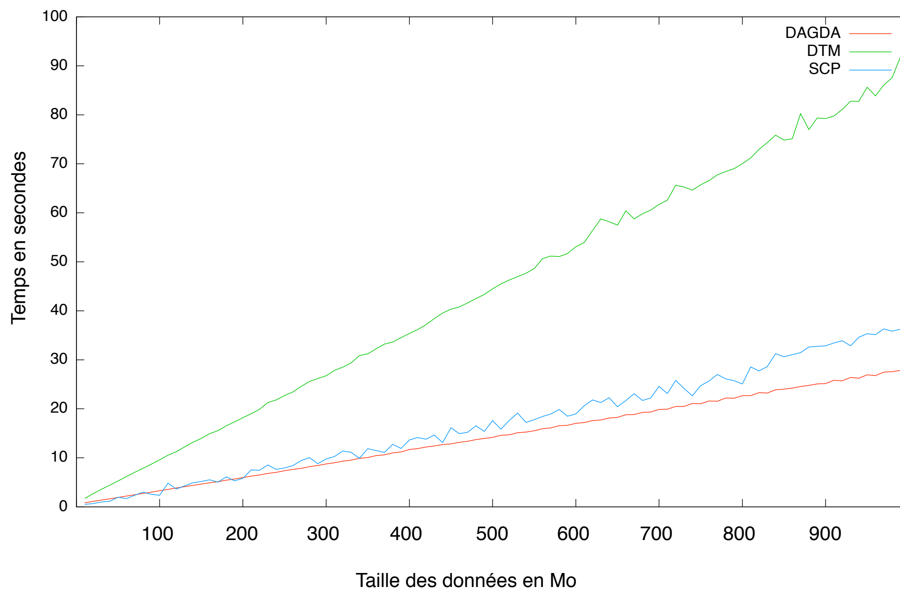

FIG. 4.17 – Moyenne des temps de transfert de données de 10 Mo à 1 Go avec DAGDA, DTM et scp.

nous avons utilisé 10 fichiers de 1 giga-octets et 26 appels consécutifs sur chacun d'entre eux. Deux nœuds homogènes ont été utilisés pour cette expérience très simple visant à comparer les différents modes de persistance proposés par DAGDA avec des espaces de stockage de tailles différentes.

Le client est situé sur un nœud du site de Rennes et le SeD sur un nœud du site d'Orsay. Nous avons mesuré un temps moyen de transmission de 28 secondes pour chaque fichier et un temps moyen de recherche dans le fichier de l'ordre de 17 secondes. Nous procédons à la recherche de la première lettre de l'alphabet sur les dix fichiers puis de la seconde lettre et ainsi de suite. On notera que les trois algorithmes de remplacement de données auront exactement le même comportement avec cet ordre de soumission. En effet, au bout des dix appels pour une même lettre, le premier fichier est à la fois celui qui a été stocké le premier, celui qui a été utilisé il y a le plus de temps et il aura été exactement utilisé le même nombre de fois que les neuf autres (l'algorithme de sélection du fichier le plus fréquemment utilisé choisit le premier fichier de la liste en cas d'égalité). Ainsi, nous distinguerons deux cas :

– Le nœud peut stocker les dix fichiers.
– Le nœud ne peut stocker qu'un sous-ensemble des fichiers.

Espace disponible sur le SeD	Avec persistance	Sans persistance
1 Go	11812,5 secondes	11808,1 secondes
5 Go	11803,2 secondes	11805,2 secondes
10 Go	4905,6 secondes	11807,5 secondes

TAB. 4.1 – Temps d'exécution de l'application avec et sans persistance des données.

Dans le tableau 4.1, on voit, que dans l'ordre de soumission choisit, la persistance n'a d'efficacité que lorsque le nœud peut stocker la totalité des données. Si à l'inverse, nous procédons aux recherches des 26 caractères sur un premier fichier, puis aux recherches des caractères sur le second fichier et ainsi de suite, la persistance est efficace, même lorsque le nœud ne peut stocker qu'un seul fichier. Les résultats obtenus pour cet ordonnancement sont présentés dans le tableau 4.2.

Espace disponible sur le SeD	Avec persistance	Sans persistance
1 Go	4899,3 secondes	11806,1 secondes
10 Go	4912,2 secondes	11823,7 secondes

TAB. 4.2 – Temps d'exécution de l'application avec et sans persistance des données avec un regroupement temporel des requêtes sur un même fichier.

Les résultats relevés correspondent bien aux résultats théoriques que l'on pouvait attendre :
Avec 10 transferts de 1 Go nécessitant environ 28 secondes chacun et 26 recherches de 17 secondes sur chacun des 10 fichiers, le temps minimal requis est de $10 \times 28 + 26 \times \times 17 = 4700$ secondes. Ce qui correspond à peu près au temps mesuré lorsque la persistance agit efficacement. Et lorsque la donnée doit être chargée systèmatiquement, on a 26 recherches nécessitant 10 transferts de 28 secondes environ et 17 secondes de parcours de chaque fichier, soit $(26 \times 10) \times (28 + 17) = 11700$ secondes.

Ces expériences ont montré que la persistance des données peut permettre d'augmenter très sensiblement les performances de la plateforme, à condition que l'ordonnancement soit adapté à l'espace de stockage disponible.

4.4.3 Réplication des données.

Nous évaluons ici les performances obtenues par l'utilisation de la réplication explicite des données pour le même service ce comptage des caractères dans le fichier transmis. Nous comparons les résultats obtenus sans persistance des données, avec persistance mais sans réplication et avec réplication des données. Pour réaliser ces expériences, nous avons utilisé 13 nœuds de calcul situés sur le même cluster et un client lancé depuis un nœud de telle manière que les temps de transfert des données entre le client et les nœuds de calcul soient significativement plus grand que le temps nécessaire aux transferts entre nœuds de calcul. Nous utilisons ici un unique fichier de 1 giga-octets sur lequel nous procéderons à 26 recherches de caractères. Le temps moyen mesuré pour la transmission entre les nœuds et de 28 secondes alors que le temps de moyen de transfert entre le client et un des 13 nœuds est d'environ 157 secondes. Le temps d'exécution du service est en moyenne de 17 secondes pour le parcours du fichier de 1 Go. L'ordonnancement utilisé est un ordonnancement Round-Robin et treize appels sont effectués en simultané sur les treize nœuds. Dans le cas de l'utilisation de la réplication, la donnée est transmise une première fois sur un SeD puis répliquée sur l'ensemble des autres.

Le tableau 4.3 présente les résultats obtenus.

Nous voyons ici que la réplication permet d'augmenter très significativement les performances d'un service lorsque les temps de transferts entre nœuds de calcul sont beaucoup

Sans persistance	Avec persistance, sans réplication	Avec réplication
4832,3 secondes	2180,1 secondes	817,0 secondes

TAB. 4.3 – Temps total d'exécution de l'application avec et sans réplication des données

plus courts que les temps de transfert entre le client et les serveurs. La réplication permet d'utiliser au mieux les nœuds de calcul en permettant la parallélisation immédiate des calculs. La persistance permet par ailleurs d'éviter certains transferts mais ne prend effet que lors d'un deuxième appel sur le même nœud nécessitant la même donnée.

4.5 Conclusion

Dans ce chapitre, nous avons présenté un nouveau gestionnaire de données pour l'intergiciel DIET développé au cours de cette thèse. Il a été conçu en vue de permettre l'implantation d'algorithmes d'ordonnancement et de réplication conjoints et est désormais disponible dans la distribution DIET. Ce gestionnaire de données permet la réplication implicite ou explicite des données par l'intermédiaire d'une nouvelle API et introduit de nouvelles fonctionnalités dans la gestion des données. Nous avons par ailleurs effectué des expérimentations montrant l'intérêt de la réplication et des optimisations introduites par DAGDA. Nous avons également vu que la gestion des données nécessite d'être prise en compte dans l'ordonnancement. En effet, la persistance permet d'éviter des transferts répétitifs d'une même donnée dans le seul cas où on regroupe les calculs temporellement et la réplication n'a d'efficacité que lorsque les calculs sont parallélisables sur la grille.

Cette première version de DAGDA est amenée à évoluer afin de permettre des optimisations plus fines de la gestion des données. Les prochaines versions permettront ainsi de développer et utiliser ses propres algorithmes de remplacement de données. Par ailleurs, la définition des règles de réplication peut être étendue afin de permettre l'implantation de différents algorithmes de gestion de données tels ceux qui sont employés par les Content Distribution Networks pour l'accès aux pages Web sur internet [100, 14, 98]. Grâce à l'API fournie par DAGDA, il est possible d'implanter dans DIET la gestion des données proposée à l'Open Grid Forum pour le standard GridRPC. Cette proposition à laquelle j'ai contribué est fournie en annexe A. Dans cette même optique de standardisation et avec pour objectif l'interopérabilité entre différents intergiciels, le format des descriptions de données utilisé par DAGDA devrait évoluer vers le standard de l'OGF Data Format Description Language [18].

Chapitre 5

Le portage d'une application bio-informatique sur la grille

5.1 DIET-BLAST : Le portage de l'application BLAST sur l'intergiciel DIET

Comme nous l'avons vu au chapitre précédent, le portage pour la grille de l'application BLAST doit prendre en compte différentes contraintes afin de tirer le meilleur parti des ressources mises à disposition. En effet, l'application devra prendre en compte :
- La gestion des bases de données sur les différents nœuds de la grille.
- Le découpage des requêtes en un certain nombre de sous-requêtes.
- L'utilisation éventuelle d'une implantation parallèle de BLAST.
- L'ordonnancement des requêtes qui seront adressées à l'intergiciel.

L'architecture de DIET-BLAST [6] a été conçue afin de laisser beaucoup de latitude quant aux différentes techniques utilisables pour ces différents points. Nous nous sommes cependant attachés à maintenir le maximum de transparence vis-à-vis de la grille et à proposer une application la plus semblable possible à l'application BLAST du NCBI destinée à une utilisation sur une machine locale.

DIET-BLAST se compose d'une application cliente qui simule le fonctionnement de l'application originale et d'une application serveur destinée à être lancée sur une multitude de nœuds de la grille.

5.1.1 L'application cliente

Lancée en ligne de commande, elle prend plusieurs paramètres en entrée :
- Les paramètres "classiques" de l'application BLAST :
  - La "version" de BLAST à utiliser. À savoir BLASTN, BLASTP, TBLASTN, BLASTX ou TBLASTX.
  - Un fichier de requêtes.
  - Un fichier de sortie qui contiendra le résultat des recherches d'alignement dans un format qui peut être choisi par l'utilisateur en utilisant l'option habituelle de BLAST.
  - La base de données à utiliser. Pour cette option, DIET-BLAST propose au choix l'utilisation classique en précisant le nom d'un fichier local, mais aussi le nom d'une base enregistrée sur la grille.
- Des paramètres spécifiques à DIET-BLAST :
  - Le fichier de configuration propres aux clients DIET.
  - Le nombre maximum d'appels parallèles effectués par le client.
  - Une taille de sous-requêtes utilisée pour les découpages du fichier d'entrée.
- Tous les paramètres optionnels de l'application BLAST traditionnelle.

L'architecture proposée se découpe ainsi en plusieurs parties :

La gestion des paramètres : Les paramètres passés en ligne de commande doivent être filtrés afin de séparer les paramètres propres à la recherche d'alignement des paramètres spécifiques à l'application. De plus, pour éviter des appels à la grille comportant des paramètres erronés, DIET-BLAST effectue un contrôle sur l'ensemble de ceux-ci. L'application vérifie que tous les paramètres passés sont acceptés par BLAST, elle vérifie par ailleurs que leur utilisation est valide. Ainsi, une ligne de commande invalide pour une exécution locale de BLAST provoquera une erreur avant même la soumission sur la grille évitant ainsi de

nombreux appels et une éventuelle attente de ressources pour n'obtenir que des messages d'erreur.

Pour effectuer ces contrôles, nous avons implanté un mécanisme basé sur trois listes de chaînes de caractères. La liste des paramètres obligatoires, la liste des paramètre autorisés et la liste des paramètres qui doivent être suivis d'une option. Les trois listes utilisées pour DIET-BLAST sont ainsi définies :

```
char* BlastConfiguration::allowed[] =
  {"--dietconfig", "--dbname", "--max-call", "--size",
   "-A", "-B", "-D", "-E", "-F", "-G", "-I", "-J", "-K", "-L",
   "-M", "-O", "-P", "-Q", "-R", "-S", "-T", "-U", "-W", "-X",
   "-Y", "-Z", "-a", "-b", "-d", "-e", "-f", "-g", "-i", "-l",
   "-m", "-n", "-o", "-p", "-q", "-r", "-t", "-v", "-w", "-y",
   "-z", NULL};

char* BlastConfiguration::needed[] = {"--dbname", "-p", "-i",
                                      "-o", NULL};

char* BlastConfiguration::withParams[] =
  {"--dietconfig", "--dbname", "--max-call", "--size",
   "-A", "-B", "-D", "-E", "-F", "-G", "-K", "-L", "-M",
   "-O", "-P", "-Q", "-R", "-W", "-X", "-Y", "-Z", "-a",
   "-b", "-d", "-e", "-f", "-g", "-i", "-l", "-m", "-o",
   "-p", "-q", "-r", "-t", "-v", "-w", "-y",  "-z", NULL};
```

Le découpage du fichier de requêtes : Afin d'expérimenter plusieurs méthodes de découpage du fichier d'entrée, la classe de soumission des requêtes BLAST prends en paramètre l'instance d'une classe de "découpage" de fichier au format FASTA. Il s'agit de la classe `FastaFileCut` dont la définition est la suivante :

```
class FastaFileCut {
public:
  FastaFileCut(std::string inputFileName, std::string baseName);
  FastaFileCut(std::string inputFileName, std::string baseName,
       unsigned long size);
  FastaFileCut(std::string inputFileName, unsigned long size);
  FastaFileCut(std::string inputFileName);
  FastaFileCut(const FastaFileCut &fc);
  virtual ˜FastaFileCut();
  virtual void cutFile(std::string dir);
  int nFile();
  std::list<std::string>::iterator begin();
  std::list<std::string>::iterator end();
};
```

Cette classe permet la redéfinition de la méthode de découpage du fichier `cutFile(std::string dir)` permettant d'implanter facilement plusieurs méthodes de

découpage. Les fichiers produits par ce “découpage” sont alors placés dans le répertoire passé en paramètre. Nous avons implanté deux classes différentes de découpage : le découpage en plusieurs fichiers contenant un nombre choisi de séquences à aligner et un découpage en des sous-requêtes de taille choisie. Ce dernier découpage utilise un algorithme de recherche du découpage le plus proche de la taille choisie. En effet, les fichiers de requêtes ne peuvent être découpés qu'entre deux séquences ce qui ne permet pas de simplement découper le fichier en plusieurs fichiers de taille “ferme”. L'algorithme n'admet cependant qu'une réduction de taille d'un maximum de 10%.P̃our cela, l'algorithme recherche le point de jonction entre deux séquences avec la taille t de la sous-requête telle que $t \geq 0,9 \times n$ et $|t - n|$ soit minimum. La figure 5.1 présente le fonctionnement de l'algorithme utilisé. Lorsque l'utilisateur ne choisit pas de taille explicitement, DIET-BLAST découpe

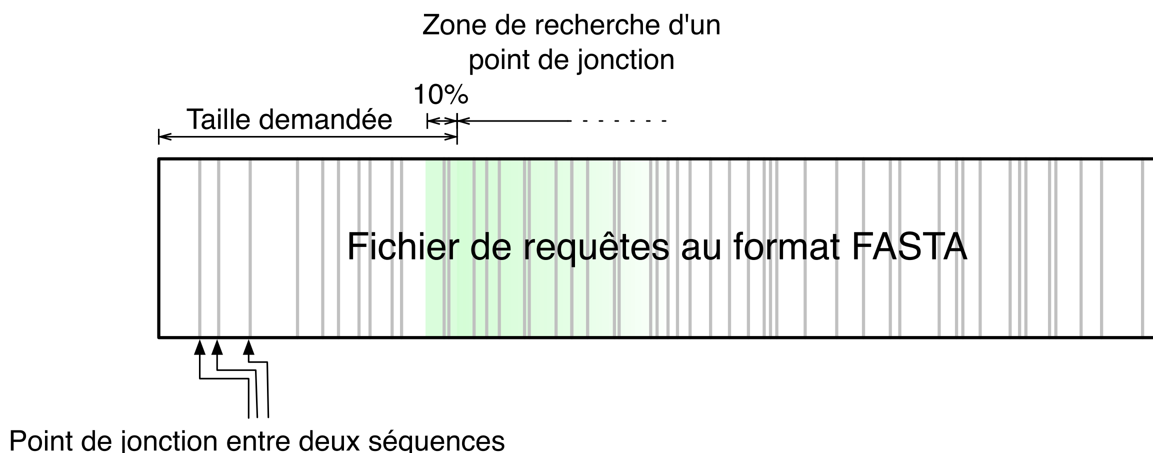

FIG. 5.1 – Principe du découpage en fichiers de taille choisie

le fichier de requête en autant de fichiers qu'il y a de séquences dans le fichier principal. Chacun des fichiers ne contenant alors qu'une seule requête. Si cette méthode permet de laisser l'ordonnanceur gérer au mieux le flux de requêtes, elle peut légèrement ralentir le processus de soumission par le nombre de messages CORBA nécessaire à la soumission de toutes les séquences. C'est donc pour permettre à l'utilisateur de choisir une alternative à cette méthode, notamment dans le cas de soumission d'un fichier contenant énormément de séquences que DIET-BLAST propose de choisir une taille de sous-requête. C'est dans cette même optique que l'algorithme de découpage n'admet qu'une marge inférieure de 10% par rapport à la taille choisie par l'utilisateur. En effet, on considérera que si l'utilisateur choisit d'utiliser cette méthode c'est qu'il a constaté une latence réseau trop importante lors de la soumission d'une requête contenant beaucoup de séquences.

Une fois le découpage effectué, l'objet FastaFileCut instancié permet d'obtenir le nombre de fichiers créés ainsi qu'un itérateur sur les noms de chacun d'eux. Ce sont les méthodes `nFile()`,`begin()` et `end()` qui assurent ces tâches. L'itérateur retourné par `end()` retournant la fin de la liste des noms de fichiers conformément à l'usage courant des itérateurs en C++.

La soumission des requêtes : Pour chaque fichier de séquences retourné par l'itérateur de l'objet `FastaFileCut`, un objet `DietBlastCall` est instancié. C'est cet objet qui permettra

de soumettre la tâche à DIET, de manière synchrone ou asynchrone et qui permettra de garder une référence de l'appel effectué afin de récupérer le résultat de la soumission. Cet objet est défini de la manière suivante :

```
class DietBlastCall {
public:
  DietBlastCall(std::string program, std::string args,
                std::string request, Database db);
  DietBlastCall(const DietBlastCall &dbcall);
  ~DietBlastCall();
  void call();
  void asyncCall();
  std::string getResult();
  int getExecTime();
  diet_reqID_t getReqID();
  void getRemoteResult();
};
```

Le constructeur prend en paramètre la version de BLAST désirée, les arguments précédemment filtrés à passer à BLAST côté serveur, le fichier de séquences et un objet `Database` associé à une base de données qu'elle soit locale ou disponible sur la grille. Cet objet est défini comme suit :

```
class Database {
public:
  Database(std::string name, std::string path="");

  Database(const Database &data) : name(data.name), path(data.path),
   dietID(data.dietID) {}
  void setID(std::string id);
  bool haveID();
  std::string getName();
  std::string getPath();
  std::string getDietID();
};
```

L'objet `Database` utilisé pour les soumissions dans DIET-BLAST associe un nom de base de données à un identifiant de donnée DIET et éventuellement à un fichier du système de fichier de l'utilisateur. Ainsi, si le nom de la base de données correspond à une donnée déjà présente dans DIET, on obtient son identifiant et on utilise la base déjà présente sur la grille. Si par contre, ce nom ne correspond pas à une donnée stockée dans le gestionnaire de données de DIET, on associe le nom à un fichier local qui est alors téléchargé sur la grille obtenant ainsi un identifiant utilisable pour les appels suivant. Ce mécanisme permet à l'utilisateur de DIET-BLAST de faire appel à une base déjà sur la grille ou de déposer sur la grille une nouvelle base à laquelle il pourra accéder grâce à un nom expressif par la suite.

Les méthodes `call()` et `asyncCall()` permettent alors de soumettre la requête référencée par l'objet `DietBlastCall` de manières respectivement synchrone ou asynchrone. La méthode `getResult()` sera alors utilisée pour attendre la notification de fin de

tâche donnée par DIET et de récupérer le résultat dans un fichier dont le chemin d'accès est retourné par la méthode. La classe permet également d'obtenir le temps d'exécution de la requête sur la machine distante et l'identifiant DIET de la requête.

L'ensemble des soumissions de chaque fichier requête est géré par la classe `BlastAsyncDietCall` qui hérite de la classe `BlastParallelCall`. Ces classes sont définies de la manière suivante :

```
class BlastParallelCall {
public:
  BlastParallelCall(std::string program, std::string args,
    std::string inputFile, std::string outputFile,
    Database db, int maxCall);
  virtual ~BlastParallelCall() {}
  virtual void call() {}
  virtual void waitResults() {}
  virtual std::list<std::string> getResults();
  int getCumulatedTime();
  int nJobs();
};
```

Le constructeur de cette classe prend en entrée la version de BLAST à utiliser, les arguments à passer à BLAST, le fichier d'entrée, le fichier de sortie, la base à utiliser et un nombre maximum d'appels simultanés. La méthode `call()` permet de lancer les soumissions, la méthode `waitResults()` d'attendre la fin de celles-ci, `getResults()` d'obtenir la liste des fichiers de résultats, `getCumulatedTime()`, le temps d'exécution cumulé de toutes les exécutions distantes (i.e. la somme des temps d'exécution de chaque tâche) et `nJobs()` le nombre d'appels effectués. Cette classe n'est jamais instantiée directement, c'est la classe `BlastAsyncDietCall` qui en hérite qui est utilisée par DIET-BLAST.

```
class BlastAsyncDietCall : public BlastParallelCall {
public:
  BlastAsyncDietCall(std::string program, std::string args,
     std::string inputFile,
                    std::string outputFile, Database db,
     int maxCall, FastaFileCut* filecut,
                    std::string tmpDir);
  BlastAsyncDietCall(BlastConfiguration cfg, Database db,
                    FastaFileCut* filecut,
     std::string tmpDir);
  ~BlastAsyncDietCall();
  void call();
  void waitResults();
  std::list<std::string> getResults();
  void aggResults();
};
```

Le constructeur de la classe prend les mêmes paramètres que celui de la classe `BlastParallelCall` auxquels on ajoute un pointeur sur un objet de type `FastaFileCut`

décrit précédemment et un répertoire temporaire où les fichiers requêtes résultant du découpage de la requête principale seront stockés pendant l'exécution. Cette classe hérite des méthodes de sa classe parente et y ajoute la méthode d'aggrégation des résultats `aggResults()` qui produira le fichier de résultat final.

La figure 5.2 présente l'organisation générale de l'application client de DIET-BLAST. L'utilisateur passe les paramètres de la recherche d'alignement à l'application cliente, un

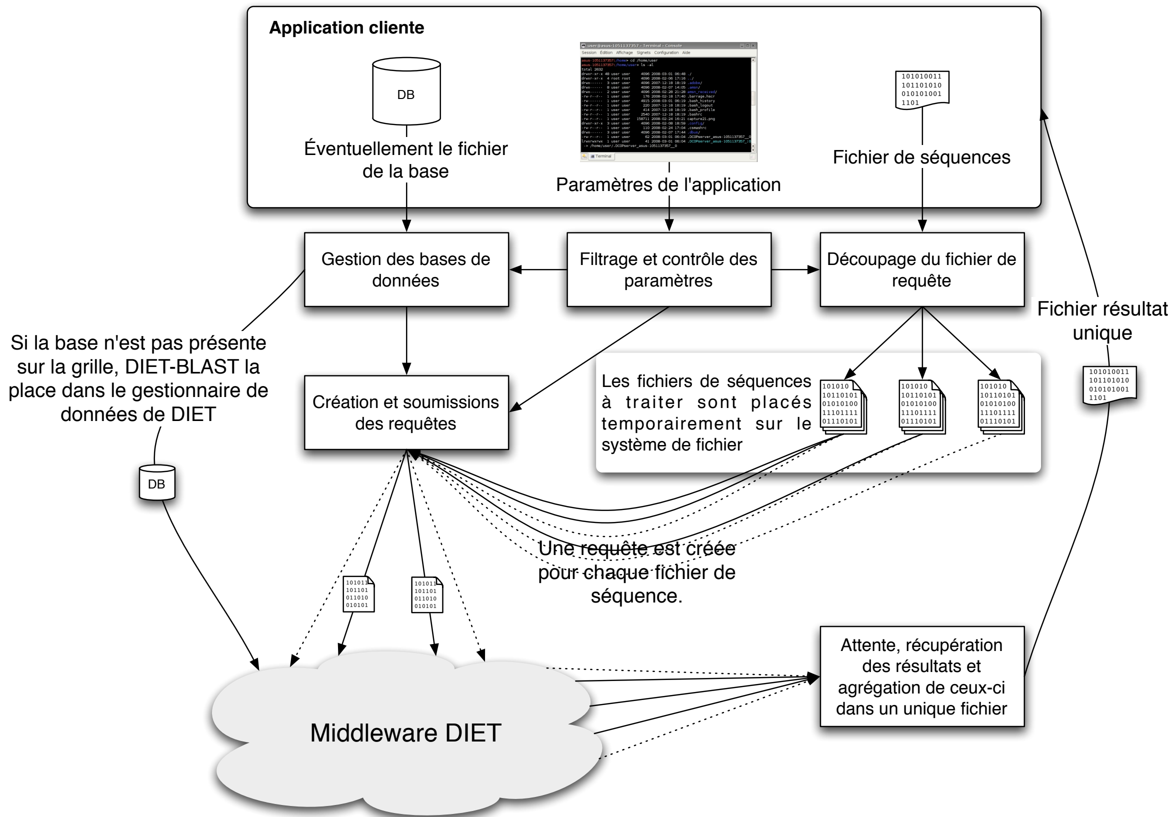

FIG. 5.2 – Organisation générale de l'application cliente de DIET-BLAST.

fichier de séquences et éventuellement la base de données si celle-ci n'est pas déjà stockée sur la grille. L'application filtre les paramètres, découpe le fichier de séquences et copie la base sur la grille si nécessaire. Un ensemble de requêtes est construit grâce aux fichiers de séquences et à l'identifiant de la base sur la grille. À la fin des calculs, l'application récupère les fichiers de résultats qu'elle agrège en un fichier unique.

5.1.2 L'application serveur

Le serveur DIET-BLAST utilise un fichier de configuration qui va permettre de fixer au mieux la manière dont DIET-BLAST va faire appel au programme externe de recherche d'alignements de séquences. En effet, DIET-BLAST a été conçu pour permettre d'utiliser au choix l'application BLAST du NCBI ou sa version parallèle mpiBLAST. Ce fichier de configuration va permettre de définir :

– Le chemin d'accès à l'application BLAST utilisée. (blastall ou mpiBLAST).
– Des paramètres par défaut à passer à l'application (par exemple, le fichier des machines à utiliser pour mpiBLAST).
– Le répertoire temporaire qui servira à stocker les résultats des requêtes.
– Un identifiant de cluster permettant de distinguer les machines de la grille suivant le cluster dans lesquelles elles sont placées.
– Une liste de bases de données à déclarer sur la grille au démarrage de l'application serveur.

On peut ainsi très facilement configurer le serveur pour qu'il utilise mpiBLAST lorsque celui-ci est disponible. La liste des bases de données directement sur le nœud permet d'éviter le transfert de celles-ci sur un nœud qui en dispose déjà. Cela permet notamment de partager une base entre plusieurs SeDs lorsque ceux-ci partagent un espace NFS.

L'application serveur est organisée en différentes parties :

L'interface avec DIET : C'est la partie du logiciel qui va déclarer les différents services offerts par DIET-BLAST. Ici, les services sont les différentes versions de BLAST (BLASTN, BLASTP etc.). Elle permet également de définir un *plugin scheduler* DIET au niveau SeD. Ces tâches sont assurées par une classe singleton qui encapsule les différents services déclarés par DIET-BLAST.

```
class DietBlastServer {
private:
  DietBlastServer(std::string configFile,
  std::list<std::string> programs, char** envp,
  void (*metric)(diet_profile_t*, estVector_t) = NULL);
public:
  static DietBlastServer* getInstance();
  int getCount();
  long getClusterID();
  std::map<std::string,std::string> getDBs();
};
```

Le constructeur de la classe prend en paramètre le nom d'un fichier de configuration, une liste des versions de BLAST assurées par le serveur (BLASTN, BLASTP etc.), un tableau de variables d'environnement et une fonction de métrique à utiliser pour l'ordonnancement (i.e. un plugin scheduler DIET). La méthode `getCount()` permet d'obtenir le nombre d'appels effectués au serveur, la méthode `getClusterID()` l'identifiant du cluster sur lequel est exécutée l'application et la méthode `getDBs()` la liste des bases de données biologiques déclarées au démarrage du serveur.

Le services BLAST : Il s'agit de la partie logicielle gérant les différents services DIET et l'interface entre les requêtes et les appels au programme externe.

```
class BlastServer {
```

```
public:
  BlastServer(std::string &program, char** envp,
              ServerConfiguration &config);
  BlastServer(const BlastServer &server) :
    program(server.program), envp(server.envp),
    config(server.config) { init(); }
  ~BlastServer() { diet_profile_desc_free(desc_profile); }
private:
  int callBack(diet_profile_t* profile);
};
```

Le constructeur de la classe prend en paramètre le nom de la version BLAST offerte par le service, une liste de variables d'environnement à passer au processus fils et une classe de configuration du serveur. La méthode `callBack()` est appelée par DIET lors de la réception d'une nouvelle tâche. C'est dans cette dernière méthode que sont effectués les récupérations des paramètres d'entrée, l'appel au programme externe réalisant la recherche d'alignements, la mesure du temps d'exécution et le renvoi des résultats obtenus.

L'exécution de BLAST : C'est l'encapsulation des appels au programme externe. Il s'agit ici de lancer un processus à partir de la configuration et des paramètres obtenus depuis le profil DIET.

```
class BlastCall {
public:
  BlastCall(std::string &db, std::string &program,
            std::string &params, std::string &request,
            char** envp, ServerConfiguration &config);
  ~BlastCall();
  int call();
  std::string getCommandLine();
  std::string getResult();
};
```

Son constructeur prend en paramètre la base de données visée par l'appel, la version de BLAST à exécuter, les paramètres à passer à l'application externe, le nom du fichier de séquences à traiter, les variables d'environnement pour l'exécution de l'application et une configuration du serveur. La méthode `call()` lance le processus de l'application BLAST externe. La méthode `getCommandLine()` retourne la ligne de commande correspondant à l'appel effectué par la méthode `call()` et la méthode `getResult()` retourne le nom du fichier résultat de l'exécution.

Les plugins d'ordonnancement : DIET-BLAST permet de choisir un plugin d'ordonnancement utilisé côté serveur. Plusieurs possibilités sont offertes à l'utilisateur :
- Un plugin basé sur la présence de la base visée. L'agent choisira en priorité les SeDs disposant de la base.
- Un plugin basé sur la taille de la file d'attente du serveur. L'agent choisira en priorité le SeD qui a le moins de tâches à effectuer avant de pouvoir traiter une nouvelle requête.

– Un plugin basé sur les performances de la machine. L'agent choisira le SeD qui dispose des meilleures performances matérielles.
– Un plugin spécial qui fournit toutes les informations des autres plugins. Ce plugin est destiné à fournir les informations nécessaires à un plugin au niveau agent pour faire le choix du SeD à utiliser.

Par défaut, aucun plugin scheduler n'est déclaré et DIET ordonnancera les tâches comme pour tout autre service (voir le chapitre 2 section 2.8.2).

La figure 5.3 présente l'organisation générale de l'application serveur de DIET-BLAST. À

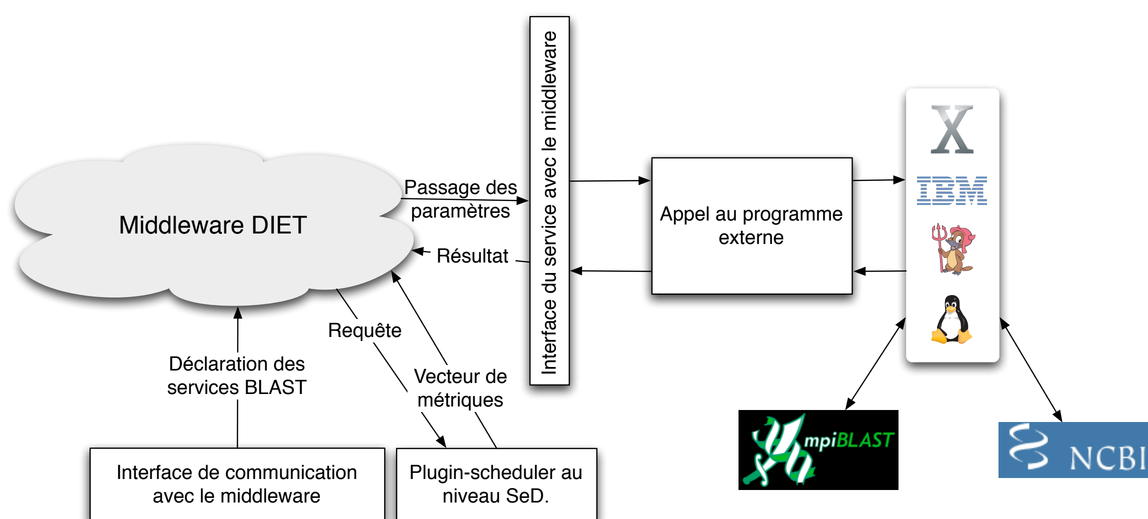

FIG. 5.3 – Organisation générale de l'application serveur de DIET-BLAST

son démarrage, l'application serveur déclare les services BLAST qu'elle propose (BLASTN, BLASTX etc.). À l'arrivée d'une requête, le plug-in d'ordonnancement de l'application retourne un ensemble de valeurs utilisées pour la sélection du nœud. Lorsque celui-ci est sélectionné pour l'exécution de la recherche d'alignement, il récupère les paramètres de la requête de l'intergiciel et lance l'exécution de l'application BLAST par un appel système. L'application retourne alors le résultat à l'intergiciel qui le transmettra à l'application cliente.

5.2 Ordonnancement des tâches et placement des données

Dans la section précédente, nous avons présenté le logiciel DIET-BLAST qui permet de soumettre de grands ensembles de séquences à aligner avec des bases de données biologiques. Dans le chapitre précédent, nous avons vu que la réplication et la distribution des données sont des facteurs importants pour les performances d'une application de bio-informatique de type BLAST sur la grille. Nous présentons ici, le travail effectué quant à l'ordonnancement des requêtes BLAST sur la grille. Notre approche basée sur l'algorithme SRA présenté au chapitre 3, section 3.4, tente d'apporter plus de dynamicité à ce dernier. En effet, si l'utilisation des ressources informatiques par les bio-informaticiens apparaît assez homogène si on choisit un interval de temps assez long, étant données la variété des utilisateurs de la grille et leurs manières d'utiliser les ressources, cet intervalle de temps peut croître

rapidement et fréquemment. Les performances enregistrées par l'utilisation de l'algorithme SRA sont dégradées durant ces périodes d'utilisation spécifique de la grille tout en maintenant de bonnes performances sur un intervalle de temps beaucoup plus long mais respectant la contrainte de constance des fréquences. Autrement dit, les fréquences sont constantes de manière globale tout en étant variables localement. La figure 5.4 présente une telle situation où les fréquences des requêtes des utilisateurs sont constante globalement, mais ne correspondent pas à la réalité pendant un intervalle de temps plus ou moins long.

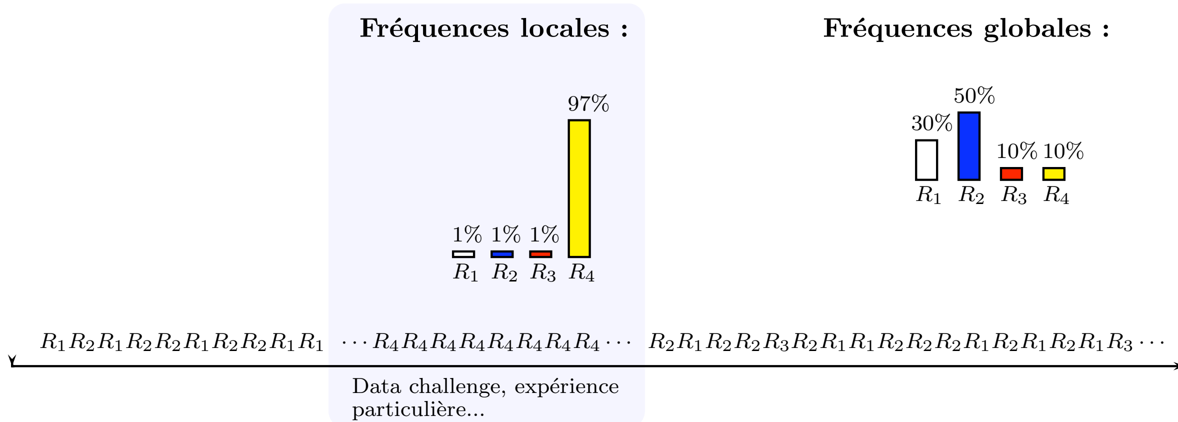

FIG. 5.4 – Variations locales de fréquences conservant les proportions globales.

5.2.1 Objectifs

L'algorithme SRA nous fournit un placement de données δ_i^j et une répartition des tâches sur les différents nœuds de la plateforme $n_i(k, j)$ optimisant le débit des tâches sur une période de temps répondant aux contraintes de fréquences utilisées dans le calcul de son résultat. Ces contraintes peuvent être respectées "globalement" sur une longue période de temps avec toutefois des périodes durant lesquelles elles ne sont pas respectées. Pendant ces intervalles de temps, l'algorithme SRA ordonnance les tâches de manière potentiellement inefficace. Notre objectif est de détecter des changements dans les fréquences de soumission et d'adapter le placement des données ainsi que la répartition des tâches sur les nœuds lorsque cela devient nécessaire au maintien des performances de la plateforme.

5.2.2 Algorithmes d'ordonnancement des tâches.

L'algorithme présenté propose deux variantes suivant le mode de réplication choisi. En effet, on peut décider de synchroniser la redistribution des bases de données à la soumission des tâches ou d'effectuer la redistribution de manière asynchrone parallèlement aux exécutions.

Réplication SRA synchrone : On effectue la réplication d'une donnée suivant le placement obtenu par le programme linéaire, au moment où une tâche nécessitant cette donnée est soumise à l'ordonnanceur. Il est alors nécessaire de choisir, lorsque plusieurs possibilités sont données par l'algorithme SRA, sur quel nœud répliquer la donnée et quelle source choisir pour effectuer le transfert.

Réplication SRA asynchrone : On commence les transferts nécessaires au passage du placement actuel au placement obtenu par le programme linéaire, dès que nécessaire. Comme pour la réplication synchrone, il peut exister plusieurs possibilités dans les choix des transferts à effectuer. En fonction de l'utilisation de la plateforme et du nombre de types de requêtes, on fixera un nombre N de requêtes nécessaires au calcul des fréquences. Par ailleurs, on choisira un seuil ε à partir duquel une différence de fréquence nécessite un nouveau lancement de l'algorithme SRA. Au démarrage de l'ordonnanceur, si on ne dispose pas d'éléments sur les fréquences de chaque type de tâche, on pourra initialiser celles-ci comme étant toutes égales. Rapidement, l'écart des fréquences mesurées avec les fréquences initiales provoquera la redistribution des données.

Algorithme SRA Dynamique

```
init(f_{k,j})
nb ⟵ 0
δ_i^j, n_i(k,j) ⟵ SRA(f_{k,j})
TantQue (l'ordonnanceur peut recevoir de nouvelles tâches T_{k,j}) faire
  Enregistrement de T_{k,j} dans f'_{k,j}
  Si (nb ≥ N ET ∃ k, j tq |f_{k,j} − f'_{k,j}| ≥ ε) alors
    δ_i^j, n_i(k,j) ⟵ SRA(f_{k,j})
    Redistribution(δ_i^j) (si effectuée en asynchrone)
    nb ⟵ 0
    Reinit(f'_{k,j})
  FinSi
  Ordonnancement de la tâche sur un nœud en fonction des n_i(k,j)
  nb ⟵ nb + 1
FinTantQue
```

Le choix du nombre N de requêtes nécessaires à l'évaluation des fréquences et du seuil ε dépendent de la plateforme et de l'utilisation qui en est faite. En effet, si un grand nombre de type de requêtes peut être soumis à l'ordonnanceur, il est nécessaire d'augmenter N afin qu'il soit au moins égal à ce nombre. De plus, afin d'éviter de provoquer sans cesse des transferts de données, il est nécessaire de disposer des fréquences les plus stables possibles. En augmentant N, on augmente la stabilité de ces fréquences (jusqu'à atteindre le fonctionnement initial de l'algorithme SRA quand $N = \infty$). Dans l'utilisation constatée des clusters de bio-informatique, on peut choisir une valeur de N relativement faible sans pour autant provoquer de transferts inutiles (avec N de l'ordre de 20 à 30 fois le nombre de type différents de tâches qui peuvent être soumis). Le choix d'un seuil ε bas, provoquera les transferts de données dès la détection d'un petit changement dans les fréquences de soumission. Les conséquences d'une variation de fréquence, même minime, sur le placement des données peuvent être très importantes, cependant, on constate souvent des variations de placement faibles si la plateforme dispose d'une marge suffisante de puissance et d'espace disque en

regard de la charge qu'on lui soumet. Des expérimentations sur les valeurs du seuil sont présentées dans la partie 5.2.6.

5.2.3 Évaluation des performances par simulations

Les tests des différents algorithmes ont été effectués à l'aide d'une version modifiée d'OptorSim [38], un simulateur mis au point par le CERN dans le cadre du projet DataGrid [89]. Dans sa version originale, OptorSim ne permet de simuler que quelques algorithmes d'ordonnancement implantés directement dans le code source. J'ai travaillé en collaboration avec Antoine Vernois à la conception d'un dispositif permettant de charger dynamiquement un ordonnanceur externe à OptorSim. L'architecture originale d'OptorSim est basée sur les composants de gestion de données de European Data Grid (EDG) [50, 53]. La figure 5.5 présente cette architecture. L'utilisateur soumet un job au Resource Broker qui sélectionne

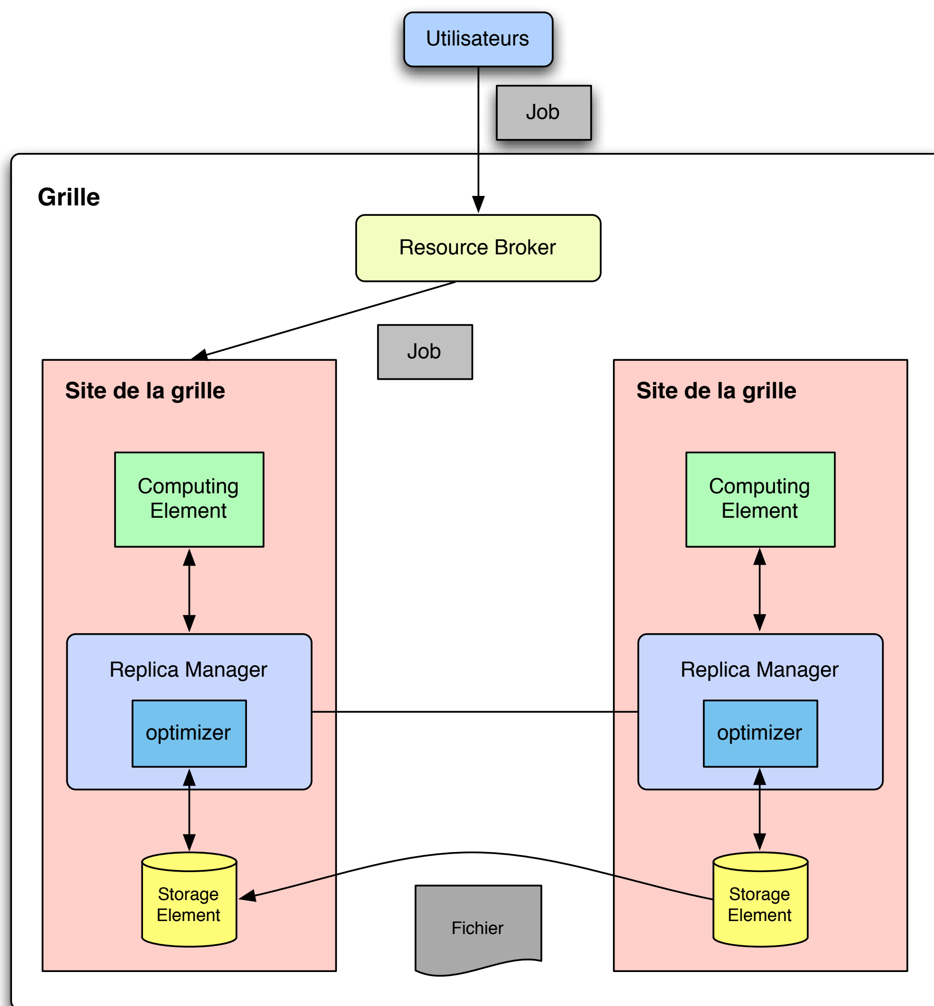

FIG. 5.5 – Architecture simulée par OptorSim

un site pour l'exécution de la tâche. Le gestionnaire de réplication se charge des transferts des données entre les différents nœuds.

Afin de pouvoir tester facilement plusieurs algorithmes d'ordonnancement, nous avons externalisé du code d'OptorSim les classes utilisées pour réaliser cette tâche. L'architecture de cette nouvelle version d'OptorSim reprend l'architecture générale de la version réalisées par Antoine Vernois durant sa thèse [96], à l'exception de l'ordonnanceur qui n'est plus statiquement figé dans le code source du simulateur mais chargé dynamiquement au lancement de celui-ci. La figure 5.6 présente cette nouvelle architecture. Dans cette architecture,

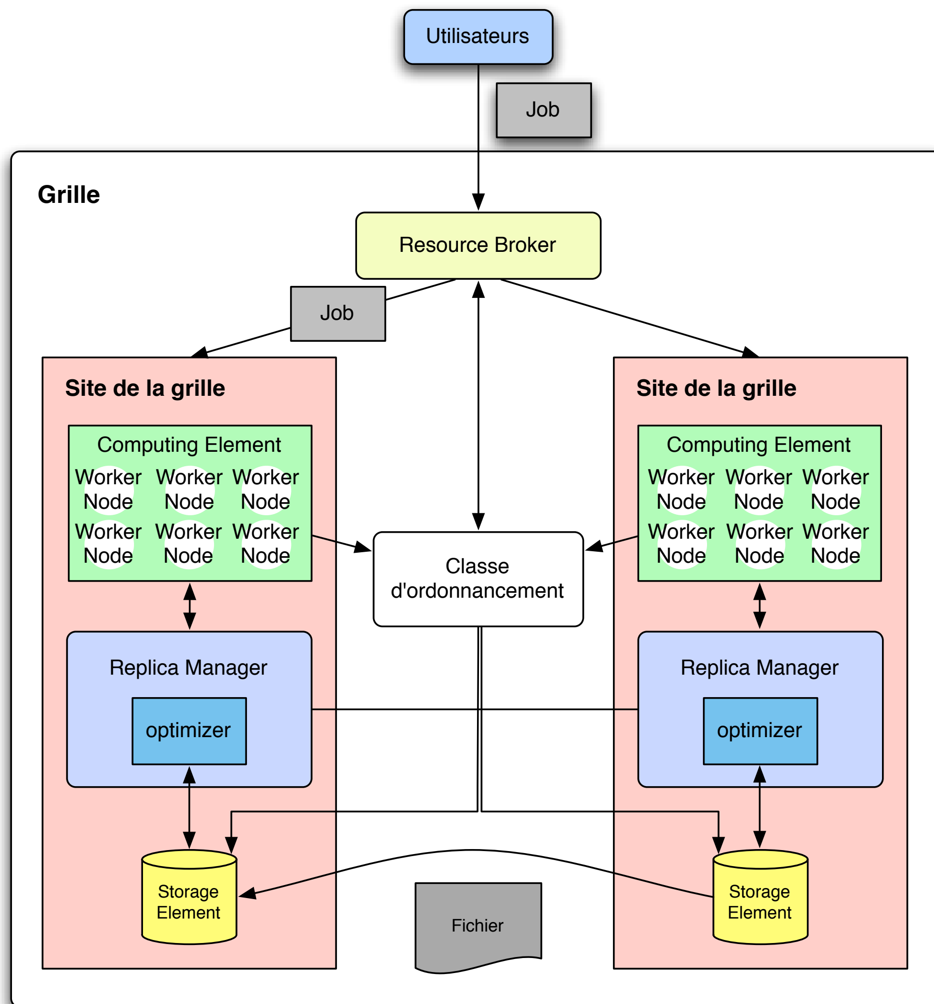

FIG. 5.6 – Architecture simulée par OptorSim après nos modifications

les nœuds implantent les différentes ressources de calcul dont ils disposent. Ce modèle plus réaliste permet de simuler des soumissions de manière plus fidèle à la réalité en prenant en compte les temps de calcul. Une classe d'ordonnancement externe à l'application est chargée et peut interragir avec les différents gestionnaire de la plateforme simulée.

5.2.4 Les algorithmes testés.

Les algorithmes présentés sont des algorithmes d'ordonnancement à la volée des tâches sur les m serveurs. On ne dispose pas d'information sur les tâches à venir et celles-ci sont

ordonnancées une à une. Si une tâche est soumise à un nœud déjà occupé, celle-ci est mise en attente jusqu'à l'instant où elle pourra être exécutée selon la politique de file d'attente FIFO.

Heuristique gloutonne simple : On choisit le nœud qui dispose de la donnée nécessaire et dont la capacité de calcul est la meilleure. Si tous les nœuds sont saturés, on choisit le nœud de capacité maximale sans prendre en compte les tâches déjà soumises sur celui-ci.

Temps d'accomplissement minimum (MCT[1]) : On choisit le nœud qui donne le plus petit temps de réalisation de la tâche. Sont donc pris en compte, la capacité du nœud, les tâches en attente sur celui-ci et le temps de transfert éventuel d'une donnée vers le nœud si la réplication est autorisée ou prévue par l'algorithme.

Ordonnancement SRA : On répartit les tâches sur les nœuds en fonction des valeurs obtenues par l'algorithme SRA. L'ordonnanceur dirige en priorité les tâches sur les nœuds qui disposent de la donnée (si ce n'est pas le cas, le transfert est en cours ou aura lieu dans le futur et la tâche est dirigée au hasard parmi ces nœuds). L'algorithme SRA a défini un placement des données et les proportions de tâches de chaque type à ordonnancer vers chacun des nœuds avant les premières soumissions. Au moment de l'ordonnancement des tâches, il s'agit bien d'un ordonnancement à la volée puisque l'algorithme ordonnance les tâches une à une sans connaître par avance les tâches suivantes.

Ordonnancement SRA dynamique [40] : On utilise l'ordonnancement obtenu par l'algorithme SRA et on enregistre les fréquences des tâches soumises. Si les fréquences des tâches s'écartent de manière significative de celles utilisées pour calculer le placement des données et la répartition des tâches, l'algorithme SRA est exécuté à nouveau avec les valeurs mesurées. Les redistributions de données s'effectueront alors en fonction de l'envoi des tâches sur les nœuds ou bien seront directement commandées par l'ordonnanceur, suivant le mode de réplication choisi.

5.2.5 Configuration des simulations

La plateforme simulée dispose de la configuration suivante (voir également la figure 5.7) :

- 2540 processeurs hétérogènes répartis de manière hétérogène sur 9 sites homogènes reliés entre eux par un réseau de 2 à 10 Gbit/s.
- Sur cette plateforme, sont stockées 5 bases de données de tailles comprises entre 150 Mo et 5 Go.
- 5 algorithmes de complexités linéaires en fonction de la taille de la donnée sur laquelle ils s'appliquent. On considère ici, qu'à un algorithme correspond une unique base (on a donc 5 types de tâches différentes).
- Des utilisateurs soumettent des tâches à l'ordonnanceur au rythme moyen de 1 tâche toutes les 3,5 secondes.

5.2.6 Les simulations

Dans chacun des tests suivant, au démarrage, les données sont réparties aléatoirement sur la plateforme. Des ensembles de tâches sont soumis à l'ordonnanceur qui les dirigera vers les nœuds de calcul en fonction de l'algorithme choisi. Nous mesurerons ici le temps moyen d'exécution d'une tâche en fonction du nombre de tâches soumises sur la plateforme.

[1]Minimum Completion Time

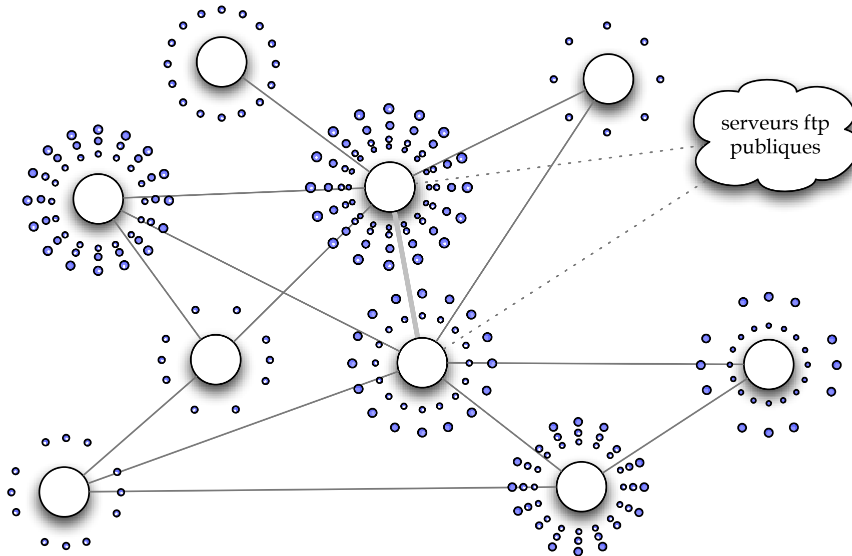

FIG. 5.7 – Configuration de la plateforme simulée.

Sans variation des fréquences des tâches

Nous présentons ici le comportement des algorithmes lorsque les fréquences de chaque type de tâche, déterminées à l'avance ne varient pas au cours du temps. Sans réplication (figure 5.8), les nœuds sont rapidement saturés et les tâches sont de plus en plus longues à être exécutées.

L'algorithme MCT (figure 5.9) se comporte mieux qu'une simple heuristique gloutonne sans réplication, mais lorsque les tâches sont trop nombreuses, la plateforme est saturée et le temps d'attente moyen des tâches croît au fur et à mesure des soumissions. En effet, MCT est conçu pour favoriser la seule exécution de la tâche à l'instant où celle-ci lui est soumise. Ainsi, dans nos simulations où un grand nombre de tâches doivent être ordonnancées, il est fréquent que MCT décide d'ordonnancer l'une d'entre elles sur un nœud rapide, à proximité réseau de la source de la donnée à utiliser (avec donc un temps très court de récupération) même s'il est pour cela nécessaire d'effacer une donnée de grande taille et peu disponible sur la grille. Les prochaines tâches utilisant cette donnée sont alors largement pénalisées. On assiste alors à la concentration des données de petites tailles sur certains nœuds et au grossissement des files d'attentes des nœuds disposant des données de tailles plus importantes.

Plus formellement, en considérant un nœud qui dispose d'une donnée nécessitant au minimum t unités de temps pour être répliquée, il recevra les tâches utilisant cette donnée jusqu'à ce que le temps estimé de réalisation de l'ensemble des tâches en file d'attente Q soit supérieur $t - T$ avec T le temps estimé de réalisation de la nouvelle tâche. Comme les tâches bio-informatiques soumises sont de courtes durées, plus le temps de transfert t est grand, moins la donnée est répliquée. Or, ce temps de transfert dépend directement de la taille de la donnée et des capacités réseaux du nœud sur lequel elle est stockée. MCT conduit donc à des situations où les données les plus petites sont souvent déplacées et répliquées au détriment des données de taille plus importante dont la distribution converge vers l'ensemble des sites à faible capacité réseau, augmentant encore le temps minimum de transfert de la donnée et réduisant par la même ses chances d'être répliquée.

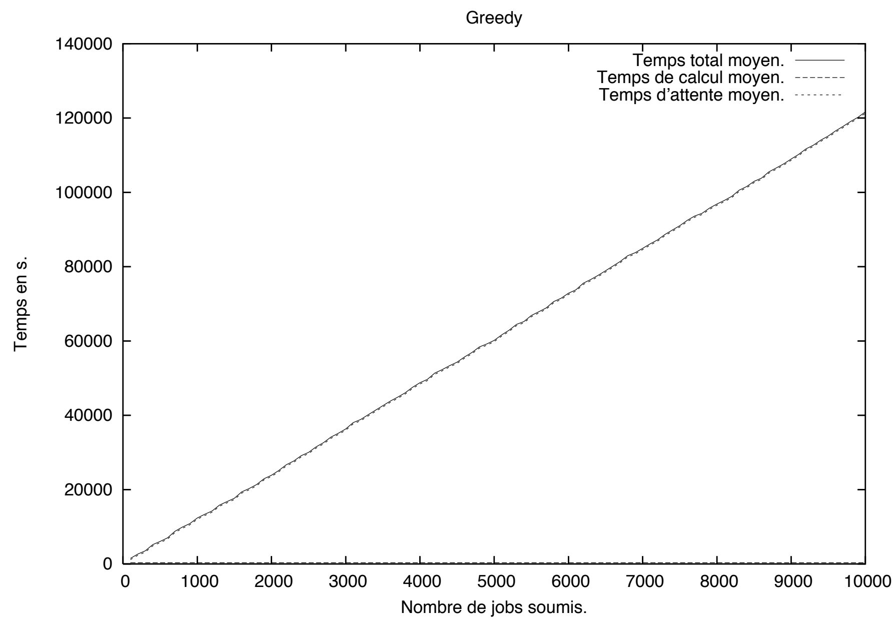

FIG. 5.8 – Greedy sans réplication.

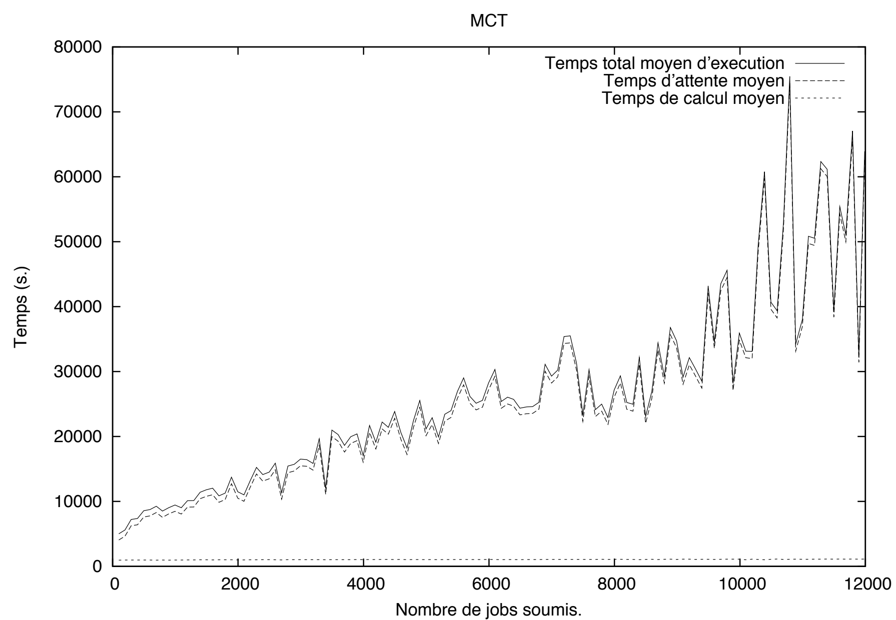

FIG. 5.9 – MCT.

L'utilisation de l'algorithme SRA (figures 5.10 et 5.11) commence à faire décroître les temps d'attente dès que le temps nécessaire aux réplications s'est écoulé. On constate que la réplication asynchrone (figure 5.11) permet d'accélérer le processus d'optimisation de l'algorithme SRA. La primordialité des réplications sur les temps d'exécution des tâches prises individuellement explique pourquoi dans nos expériences les temps moyens de réalisation des tâches sont d'un ordre de grandeur inférieurs aux temps obtenus par l'algorithme MCT. Par ailleurs, l'ordonnancement fourni par l'algorithme SRA est directement couplé à la distribution des bases, ce qui lui assure de bonnes performances.

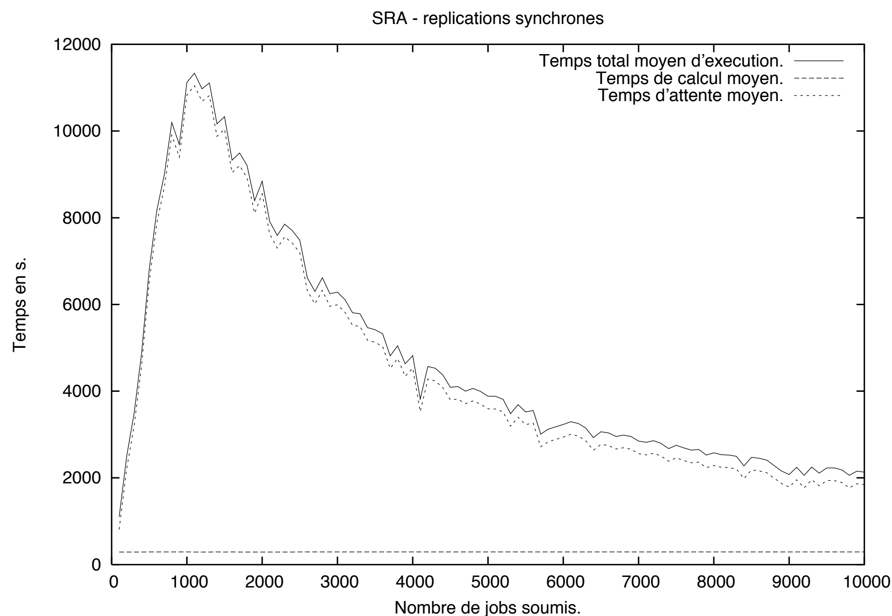

FIG. 5.10 – SRA avec réplications synchrones.

Pour nous assurer de la validité de nos conclusions, nous avons testé une approche hybride utilisant la distribution des données fournie par SRA avec l'ordonnancement des tâches fourni par MCT. MCT sélectionnant alors le nœud offrant le temps d'accomplissement minimum à la tâche soumise parmi les seuls nœuds disposant de la donnée. La figure 5.12 présente les résultats obtenus avec cette approche. On observe dans cette figure que les temps moyens d'exécution restent très supérieurs aux temps obtenus par l'algorithme SRA, mais qu'il restent dans le même ordre de grandeur.

Variation des fréquences

Nous avons ensuite simulé une plateforme sur laquelle les tâches soumises ne sont plus de fréquence constante. Pour cela, nous avons utilisé deux modèles de variations de fréquences pour tester nos algorithmes :

– La fréquence la plus haute décroît progressivement pour atteindre 0, puis la nouvelle

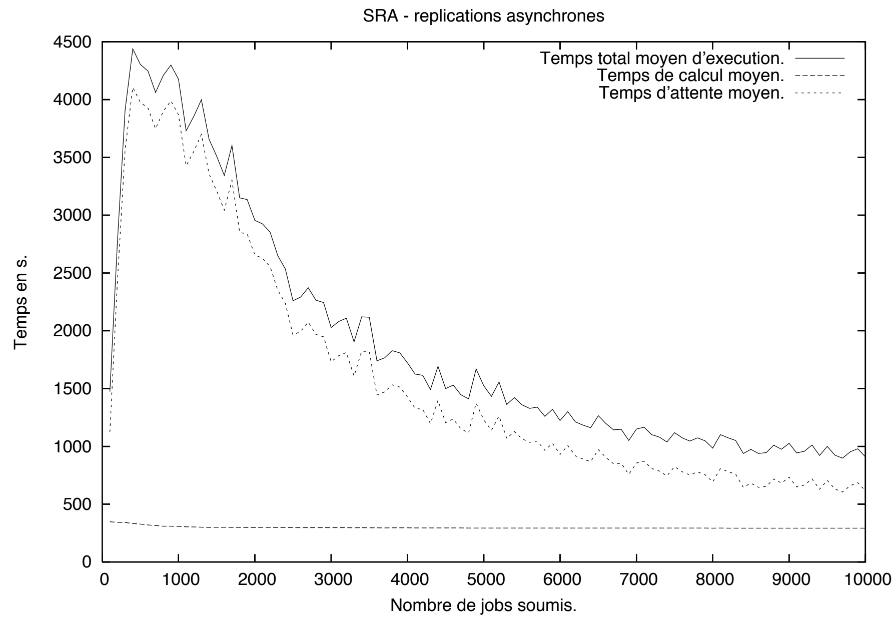

FIG. 5.11 – SRA avec réplications asynchrones.

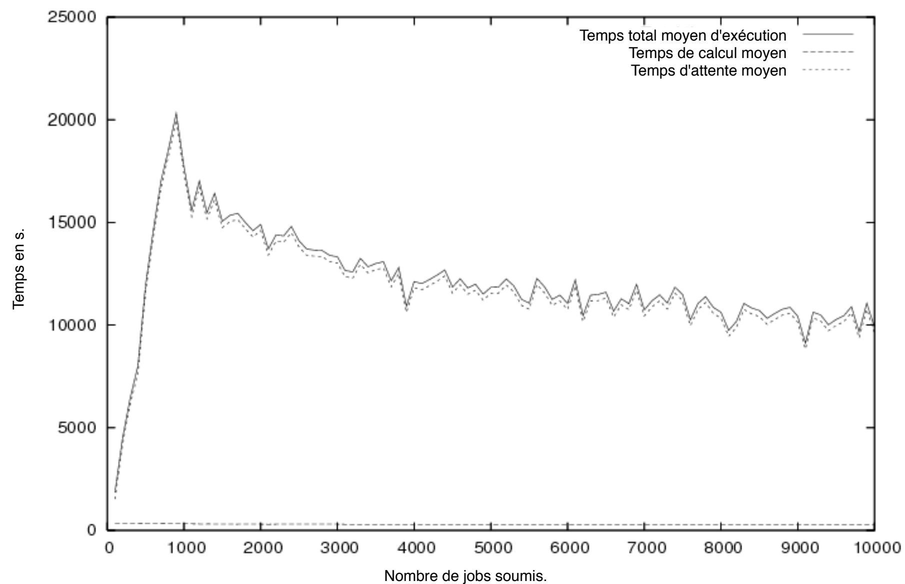

FIG. 5.12 – Approche hybride - Distribution des données SRA / Ordonnancement MCT.

fréquence la plus haute est à son tour choisie. Dans cette situation, les fréquences varient constamment ce qui représente le cas le plus critique pour notre algorithme (figure 5.14). En effet, les intervalles de temps où les fréquences sont constantes sont beaucoup plus courts.
- Toutes les fréquences sont changées à partir de la tâche $n/2$ lorsqu'un ensemble de n tâches est soumis à la plateforme (figure 5.15).

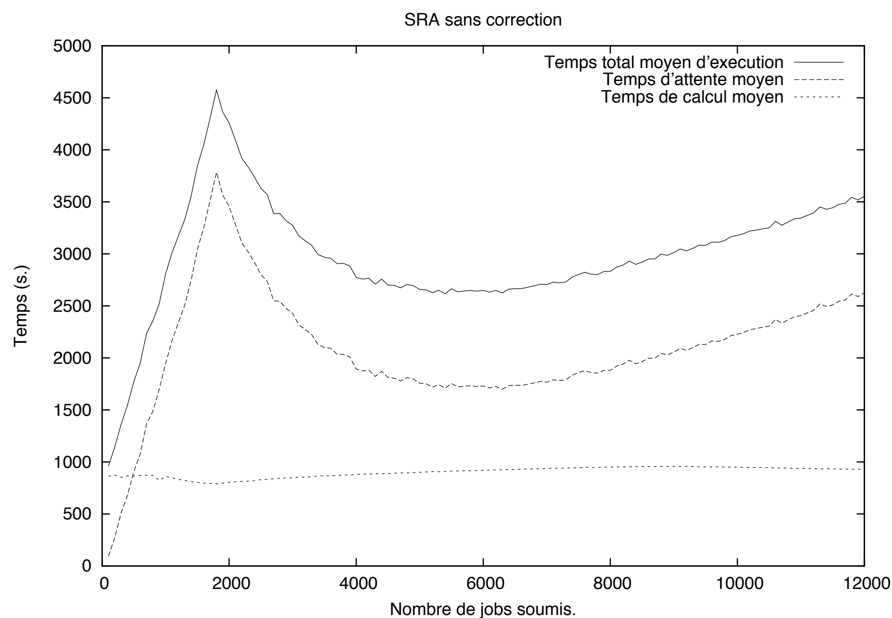

FIG. 5.13 – Sans correction des placements.

Sans correction du placement des données et de la distribution des tâches, les variations de fréquences font progressivement perdre son efficacité à l'algorithme SRA. Pour des ensembles de tâches de petite taille, la variation des fréquences n'a pas le temps d'avoir beaucoup d'effet sur les performances de l'algorithme, mais dès que les tâches sont suffisamment nombreuses, la plateforme commence à saturer malgré les réplications et le temps d'attente moyen des tâches croît régulièrement (figure 5.13).

La figure 5.14 présente le comportement de notre algorithme face un changement continuel des fréquences. On peut constater que les performances sont dégradées, mais que le temps d'attente moyen des tâches ne croît pas aussi vite que lorsqu'aucune correction n'est apportée. Dans la figure 5.15, les fréquences sont changées une fois au milieu du processus de soumission. Notre algorithme apporte alors une correction efficace et on n'assiste plus à une croissance des temps d'attente des tâches.

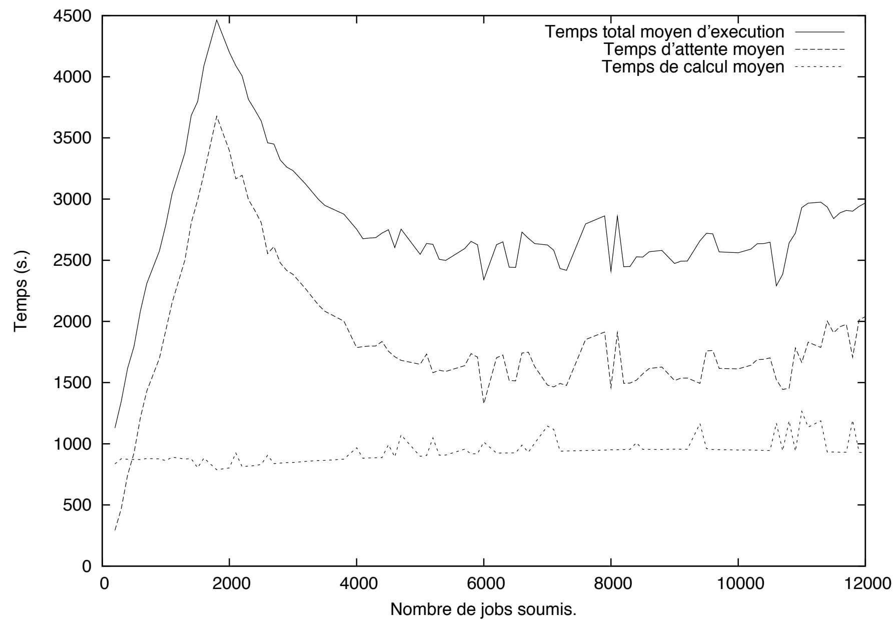

FIG. 5.14 – Variation continuelle des fréquences.

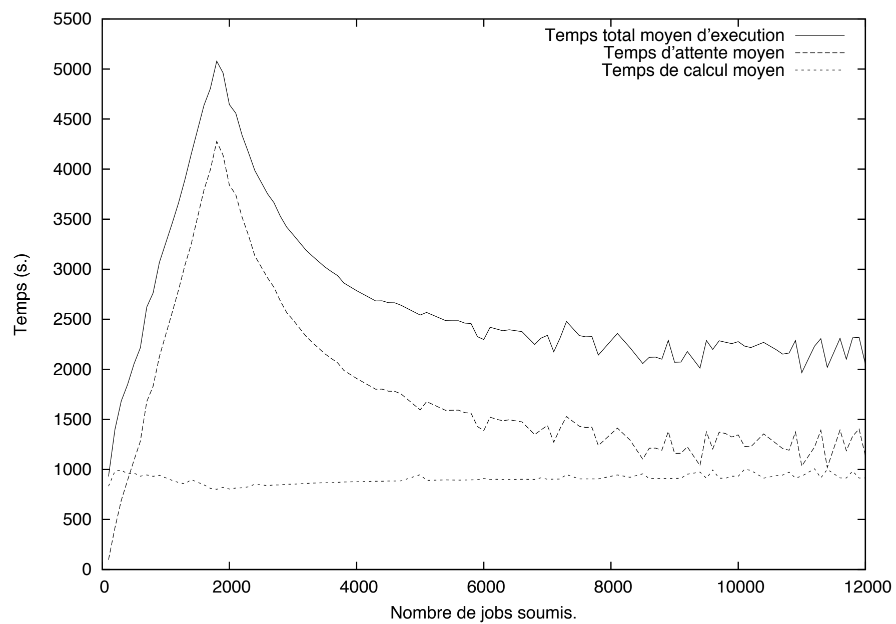

FIG. 5.15 – Variation ponctuelle des fréquences.

Influence du choix du seuil

Nous testons ici deux valeurs de seuil de variation de fréquence au-delà duquel l'algorithme SRA est relancé pour calculer un nouveau placement des données et la répartitions des tâches sur les nœuds. Le modèle de changement de fréquences choisi est le changement continu de toutes les fréquences de tâches. La figure 5.16 présente le comportement de l'algorithme pour un seuil de 5%, la figure 5.17 pour un seuil de 10%. On constate que l'algorithme corrige les effets du changement de fréquences pour un seuil de 5% et qu'aucune amélioration n'est apportée par un seuil trop élevé. Le choix d'un seuil plus faible n'apporte pas d'amélioration malgré un coût élevé en terme de calcul des placements et des proportions d'ordonnancement par l'algorithme SRA.

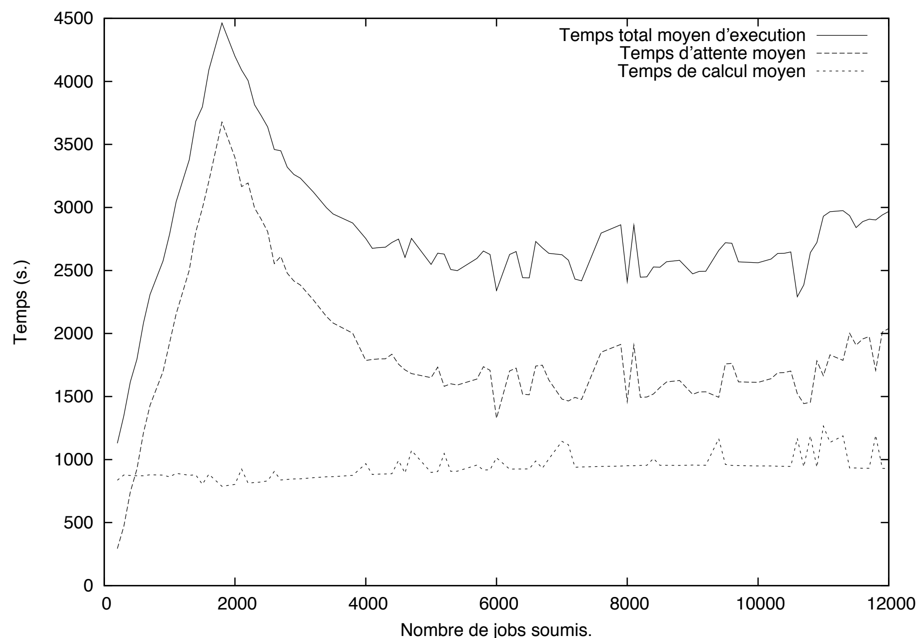

FIG. 5.16 – Seuil fixé à 5%

5.3 Implantation des algorithmes pour DIET-BLAST

Aprés avoir validé notre approche par simulation, nous avons implanté plusieurs algorithmes d'ordonnancement en vue de leur utilisation avec DIET-BLAST. Pour cela nous avons dû étendre les fonctionnalités offertes par DIET. En effet, le système de plugin-scheduler au niveau des SeDs qu'offrait alors l'intergiciel ne permettait pas d'implanter l'algorithme SRA dynamique présenté dans la section précédente. Le gestionnaire de données DTM n'offraient pas de fonctionnalités de réplications ou de gestion directe des données. De plus, ne pouvant obtenir d'estimation de temps de transfert, il était impossible d'implanter un algorithme MCT efficace. Pour pallier à ces manques, nous avons implanté

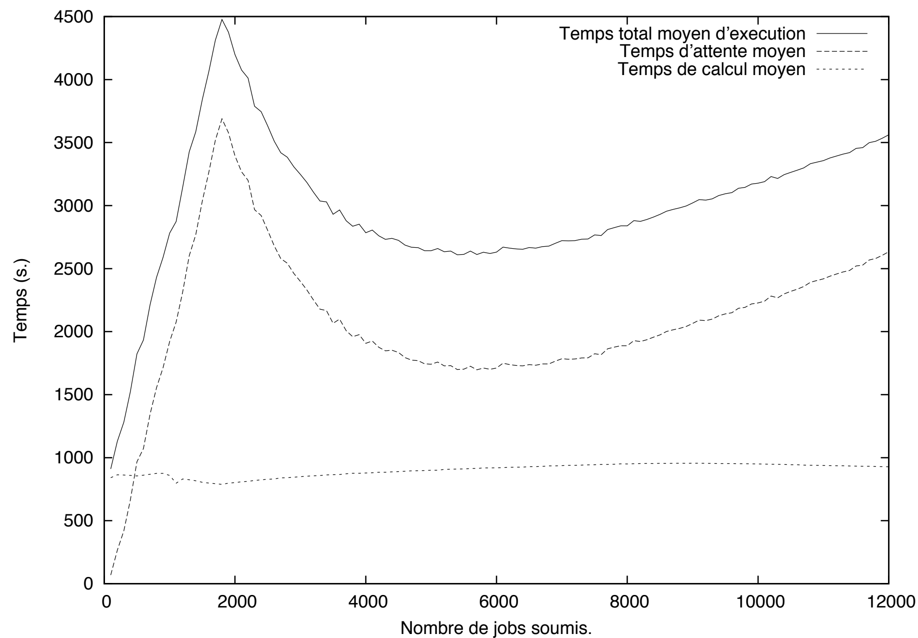

FIG. 5.17 – Seuil fixé à 10%

deux nouveautés dans DIET :

– Un mécanisme d'ordonnancement par chargement dynamique d'un module au niveau des agents.
– Un gestionnaire de données permettant la gestion directe des données, gérant la réplication et permettant le partage de fichiers entre plusieurs nœuds pouvant accéder à une même partition. Le gestionnaire Data Arangement for the Grid and Distributed Applications (DAGDA) est présenté en détail au chapitre 4.

Pour l'algorithme MCT, on peut n'utiliser qu'un plugin d'ordonnancement au niveau SeD "traditionnel". Cependant, il peut s'avérer utile d'utiliser cet ordonnancement sur une partie seulement du déploiement DIET. C'est le cas pour l'ordonnancement SRA dynamique qui réutilise ce plugin pour le choix du "meilleur" nœud d'un cluster particulier. Nous avons donc implanté cet algorithme sous forme de module chargeable dynamiquement par DIET en vue de sa réutilisation.

Pour l'algorithme SRA dynamique, on effectue un déploiement de DIET de telle manière que chaque cluster de la grille soit géré par un Local Agent. Les Local Agents sont tous connectés au Master Agent. Au niveau des Local Agents, on effectue le choix de la machine à utiliser en utilisant l'algorithme d'ordonnancement MCT. Le Master Agent quant à lui fait appel à une classe séparée qui implante l'algorithme SRA dynamique. Ainsi, à chaque nouvelle requête, il enregistre une trace de celle-ci pour maintenir des statistiques sur les fréquences de chaque type de requête et recalcule le cas échéant la distribution des données

sur la grille et la proportion de requêtes de chaque type à attribuer à chaque cluster. Le Master Agent décide alors à quel cluster il attribue la tâche. Pour cela, il entretient un compteur du nombre de requêtes attribuées à chacun d'entre eux. L'algorithme employé par le Master Agent est donc le suivant :

Algorithme de choix du cluster pour SRA dynamique

- $n_i(k,j)$ désigne le nombre de requêtes $R_{k,j}$ a exécuter sur le nœud i.
- $R_{k,j}$ est la requête à ordonnancer.
- n est le nombre de clusters gérés par DIET.
- $CMPT_i(k,j)$ est le compteur du nombre d'exécutions d'une tâche $R_{k,j}$ restant à faire sur le nœud i.

$n_i(k,j) \longleftarrow$ DynSRA($R_{k,j}$) *(La requête est transmise au module dynSRA qui ne met à jour les $n_i(k,j)$ que lorsque cela est nécessaire)*

Si $\forall i \in [1..n]$, $CMPT_i(k,j) = 0$ **alors**

  Pour $i \longleftarrow 1$ **à** n **faire**

    $CMPT_i \longleftarrow n_i(k,j)$

  FinPour

FinSi

Choisir un i tel que $CMPT_i(k,j) > 0$

$CMPT_i(k,j) \longleftarrow CMPT_i(k,j) - 1$

Ordonnancer la tâche $R_{k,j}$ sur le nœud i.

L'implantation de l'algorithme est particulièrement aisée puisqu'elle ne fait appel à aucune donnée dynamique de la plateforme. Dans cette architecture, les Local Agents font office de queues de batch pour les différents clusters et une implantation pour l'intergiciel gLite est parfaitement envisageable.

5.4 Expérimentations sur une plateforme réelle

Nous avons testé notre algorithme sur la plateforme dédiée Grid'5000 [22]. Le projet Grid'5000, devenu aujourd'hui Aladdin-Grid'5000 a pour but de fournir une plateforme pour les recherches sur les grilles informatiques. Son objectif est de déployer une grille de calcul comportant 5000 processeurs répartis sur une dizaine de sites en France. Unique en son genre cette plateforme est à la seule destination de la recherche sur les grilles de calculs, permettant ainsi des expérimentations impossibles à mettre en place sur une plateforme de production. Actuellement Grid'5000 comporte 3216 processeurs répartis sur 9 sites reliés par le réseau RENATER [81]. La figure 5.18 présente la topologie du réseau et l'emplacement géographique des différents sites accueillant un nœud Grid'5000.

Une des caractéristiques de Grid'5000 est d'offrir la possibilité de démarrer l'ensemble des machines réservées en utilisant l'image d'un système d'exploitation choisi, voire construit par l'utilisateur lui-même. Ce mécanisme assuré par le logiciel Kadeploy [46] permet à l'utilisateur d'avoir le plein contrôle sur les machines qu'il utilise et notamment de

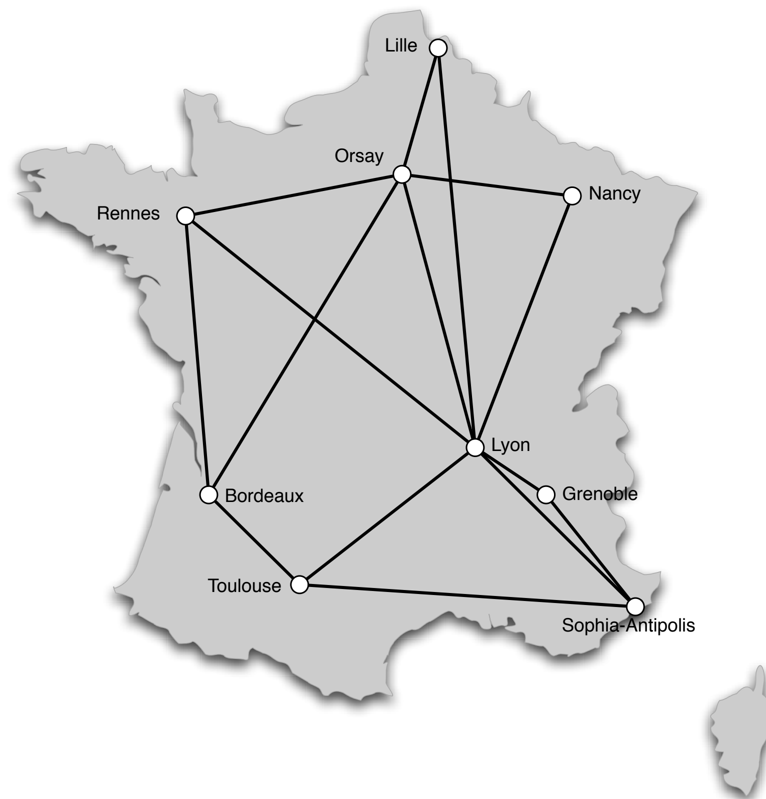

FIG. 5.18 – La répartition géographique des différents sites de Grid'5000 et leur réseau d'interconnexion.

s'y connecter en tant qu'administrateur avec tous les privilèges que cela implique. Pour nos expériences, nous avons systèmatiquement utilisé une telle image système déployée sur l'ensemble des nœuds réservés. Les réservations sur Grid'5000 s'effectuent par l'intermédiaire du gestionnaire de ressources OAR [26]. Pour les déploiements de la hiérarchie DIET sur les machines réservées nous avons utilisé le logiciel de déploiement spécifique à DIET, GoDIET [28]. À partir d'une description en XML de la plateforme à déployer, GoDIET se charge de lancer l'ensemble des éléments de l'intergiciel en respectant l'ordre des lancements. Il est en effet nécessaire de commencer tout déploiement de DIET par le démarrage de l'Object Request Broker (ORB) nécessaire aux communication CORBA utilisées par DIET. Puis le Master Agent auquel se connecteront les Local Agents et ainsi de suite jusqu'au lancement des SeDs. L'emploi de GoDIET nous a permis d'effectuer des expériences sur des déploiements comportant jusqu'à 1000 nœuds.

5.4.1 Les expérimentations

Deux stratégies de découpage des requêtes.

Nous testons ici deux stratégies de découpage des fichiers de requêtes :

- Le découpage du fichier de requêtes en le nombre de séquences qu'il contient. C'est le découpage maximum possible du fichier.

– Le découpage du fichier en le nombre de nœuds déployés sur la plateforme. Chaque fichier ainsi produit contient approximativement le même nombre de requêtes (l'écart maximum constaté est de moins de 10%).

Cette expérimentation a pour but de déterminer dans quelle mesure l'algorithme d'ordonnancement va pouvoir compenser la latence réseau provoquée par la multitude des soumissions. Il s'agit ici essentiellement de valider l'approche de notre algorithme basée sur l'ordonnancement de nombreuses tâches individuelles courtes.

Pour ces expérimentations, nous avons utilisé un déploiement sur 300 nœuds pour la recherche d'alignements de 40000 séquences sur deux bases différentes utilisant quatre versions différentes de BLAST : BLASTN, BLASTP, TBLASTN et BLASTX. Les différentes tâches à réaliser sont réparties équitablement :

– 10000 requêtes de type BLASTN sur la base de séquences ADN.
– 10000 requêtes de type BLASTP sur la base de séquences protéiniques.
– 10000 requêtes de type TBLASTN sur la base de séquences ADN.
– 10000 requêtes de type BLASTX sur la base de séquences protéiniques.

Les nœuds utilisés sont répartis sur quatre sites : Rennes, Lyon, Orsay et Nancy. Chaque expérience a été réalisée dix fois afin d'atténuer les éventuelles interférences provoquées par d'autres expériences en cours sur la grille. Les résultats présentés sont les moyennes des dix temps d'exécution totaux pour l'alignement des 40000 séquences.

Quatre algorithmes d'ordonnancement différents sont utilisés pour l'expérience :

– Random : L'ordonnancement est effectué aléatoirement.
– Temps de Complétion Minimum (MCT).
– Round-Robin : Chaque nœud est utilisé tour à tour.
– SRA Dynamique.

La figure 5.19 présente les résultats obtenus avec la stratégie de découpage maximum des requêtes. La figure 5.20 présente quant à elle les résultats obtenus pour la stratégie de découpage en le nombre de nœuds disponibles.

On constate que pour les algorithmes Random, MCT et Round-Robin, la division de la requête en le nombre de nœuds disponibles est la meilleure stratégie. À l'inverse, pour l'algorithme SRA dynamique, la division en requêtes contenant une seule séquence est le meilleur choix. On constate par ailleurs qu'en général notre algorithme donne de meilleurs résultats en terme de temps d'exécution de l'ensemble des requêtes.

Nous avons ainsi validé notre approche consistant à ordonnancer un grand nombre de requêtes courtes plutôt que de distribuer des portions identiques de requêtes sur chacun des nœuds utilisés.

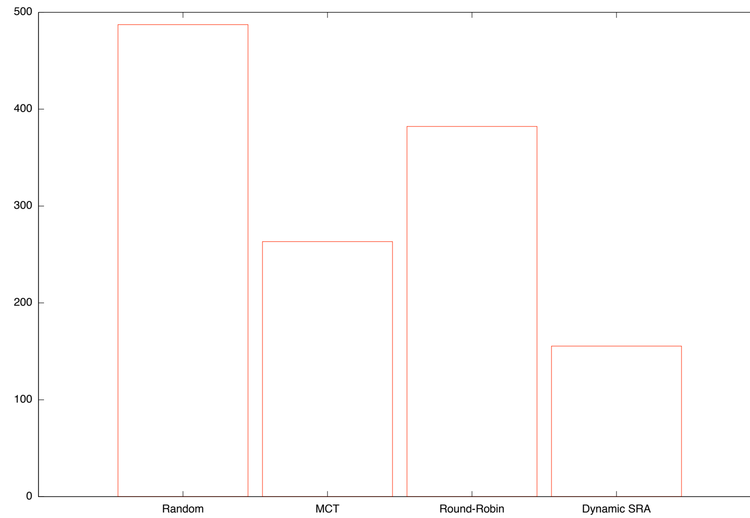

FIG. 5.19 – Temps moyen d'exécution pour quatre algorithmes d'ordonnancement différents avec la stratégie de découpage maximal des requêtes.

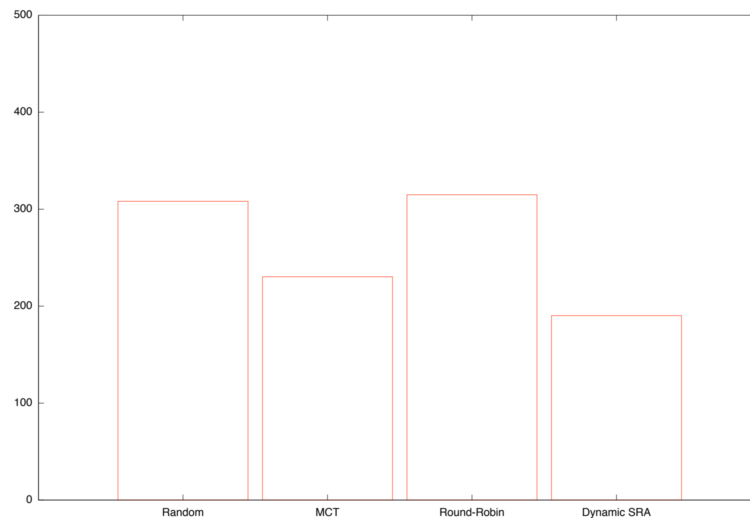

FIG. 5.20 – Temps moyen d'exécution pour quatre algorithmes d'ordonnancement différents avec la stratégie de découpage des requêtes en le nombre de nœuds disponibles.

Comparaison des performances de MCT et de dynamique SRA pour un déploiement de 1000 nœuds

Pour cette expérience à grande échelle (plus de la moitié des machines disponibles sur Grid'5000 à cette époque), nous avons utilisés quatre bases de données biologiques différentes de tailles comprises entre 175 Mo et 6,5 Go. Les fichiers de requêtes contenaient au total 40000 séquences. Les 1000 nœuds étaient répartis sur l'ensemble des sites de la plateforme excepté Grenoble. Les séquences utilisées ont été réparties de telle manière que la mesure de fréquences réalisée par notre algorithme varie au cours du temps :

- Pour les requêtes de 1 à 20000 :
  - BLASTP Vs Base n°1 : 30% des requêtes.
  - BLASTN Vs Base n°2 : 30% des requêtes.
  - BLASTX Vs Base n°3 : 20% des requêtes.
  - TBLASTX Vs Base n°4 : 20% des requêtes.
- Pour les requêtes de 20001 à 40000 :
  - BLASTP Vs Base n°1 : 10% des requêtes.
  - BLASTP Vs Base n°3 : 30% des requêtes.
  - BLASTN Vs Base n°2 : 10% des requêtes.
  - BLASTN Vs Base n°4 : 10% des requêtes.
  - BLASTX Vs Base n°1 : 10% des requêtes.
  - BLASTX Vs Base n°3 : 10% des requêtes.
  - TBLASTX Vs Base n°2 : 20% des requêtes.

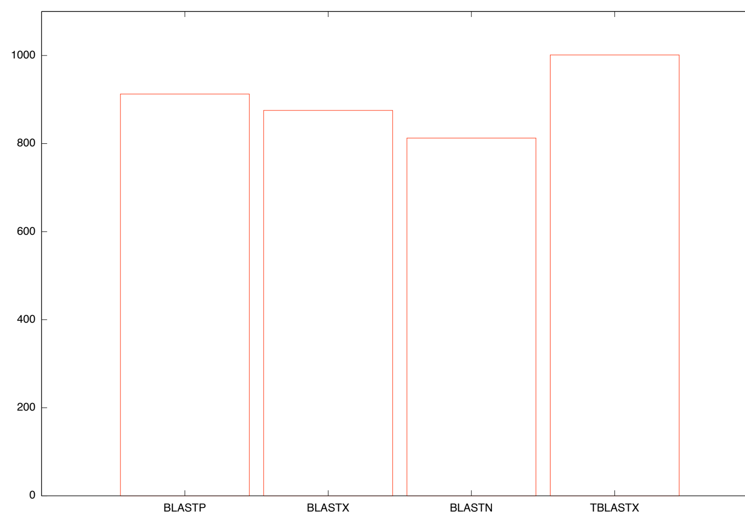

FIG. 5.21 – Temps d'exécution moyen pour l'ensemble des requêtes de chaque type sur 1000 nœuds en utilisant l'algorithme MCT.

Les expériences ont été répétées plusieurs fois afin d'atténuer l'éventuelle influence d'autres expériences en cours sur la grille. Les mesures présentées sont les moyennes des résultats des différentes expériences. Nous avons séparé les résultats de chaque type de soumission (BLASTP, BLASTN, BLASTX, TBLASTX). La figure 5.21 présente les résultats ob-

tenus en utilisant l'algorithme MCT. La figure 5.22 présente les résultats obtenus par notre algorithme.

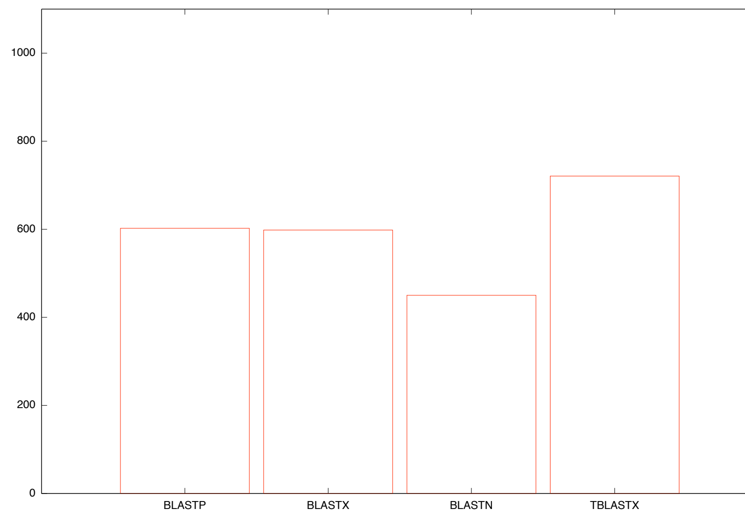

FIG. 5.22 – Temps d'exécution moyen pour l'ensemble des requêtes de chaque type sur 1000 nœuds en utilisant l'algorithme SRA dynamique.

Les résultats obtenus montrent que l'on peut attendre des gains conséquents de l'utilisation de notre algorithme pour l'ordonnancement des requêtes BLAST sur la grille. Ici, l'utilisation de notre algorithme apporte des gains supérieurs à 30% par rapport à l'utilisation de MCT.

5.5 Conclusion

Dans ce chapitre, nous avons présenté notre implantation de l'application BLAST pour la grille utilisant l'intergiciel DIET. Nous avons ensuite présenté notre travail sur l'ordonnancement des tâches bio-informatiques couplé à la réplication des bases de données biologiques. Pour valider notre approche, nous avons d'abord procédé à des simulations grâce au simulateur OptorSim largement modifié par nos soins de manière à pouvoir implanter n'importe quel algorithme d'ordonnancement et n'importe quel algorithme de gestion des données sur le type de grille simulé. L'analyse des résultats obtenus nous a permis d'envisager de nettes améliorations du débit de la plateforme, chose que nous avons pu constater dans des expérimentations réelles sur la grille Grid'5000. Les écarts constatés entre le bénéfice théorique obtenus par notre algorithme lors des simulations et les bénéfices obtenus en expérimentations réelles proviennent du fait que la plateforme Grid'5000 est beaucoup plus homogène que la plateforme que nous avions simulé. Ceci est d'autant plus vrai en terme de connexions réseaux qui sont artificiellement saturées lors des simulations alors qu'il est quasiment impossible de les utiliser à plein dans les expériences réelles. Les gains obtenus supérieurs à 30% restent néanmoins très conséquents, sachant qu'ils représentent plusieurs heures de différence durant nos expériences.

Chapitre 6

Conclusion et perspectives

6.1 Contributions

Durant cette thèse, nous nous sommes intéressés aux problématiques de gestion des données sur grille de calcul en général et plus particulièrement à la gestion des bases de données biologiques et leur utilisation sur des plateformes de ce type. La proposition et l'étude d'un nouvel algorithme semi-statique d'ordonnancement des tâches de bio-informatique fortement lié à la réplication des bases de données biologiques m'ont conduit à développer un nouveau gestionnaire de données pour l'intergiciel DIET utilisé pour les expérimentations sur plateforme réelle. Dans la démarche entreprise, nous nous sommes attachés à d'abord évaluer nos proposition par simulation, puis sur la plateforme Grid'5000 spécialement dédiée à la recherche sur les grilles de calcul. L'ensemble des travaux ici exposés sont l'objet d'un chapitre dans un livre consacré à l'utilisation de la grille pour les sciences de la vie, à paraître prochainement [94].

6.1.1 Ordonnancement des tâches bio-informatiques sur les grilles de calcul

Notre algorithme basé sur le travail précédemment entrepris par Antoine Vernois durant sa thèse a permis d'améliorer les précédents résultats en permettant une plus grande dynamicité de l'usage des grilles. Ceci correspond plus certainement à la réalité de l'utilisation des plateformes de bio-informatique qui voient leur nombre d'utilisateurs grandir continuellement et par la même une plus grande diversité dans la manière de les utiliser. Nous avons obtenus des gains de performances significatifs et montré l'intérêt d'un ordonnancement spécifique par rapport aux techniques plus générales offertes par les intergiciels de grilles courants. Par ailleurs, l'algorithme proposé permet de s'affranchir des systèmes d'informations des grilles pour l'obtention des informations dynamiques sur les nœuds de calcul. En effet, l'obtention de ce type d'informations est en soit un problème complexe et bien souvent, les données obtenues de services dédiés des intergiciels ne sont pas suffisamment récentes pour être utilisées pour l'ordonnancement de tâches très courtes. Dans notre approche, les informations dynamiques nécessaires à l'ordonnancement sont recueillies par l'ordonnanceur lui-même et ne consiste qu'en la trace des soumissions précédemment reçues.

6.1.2 DAGDA : Un gestionnaire de données pour l'intergiciel DIET

La mise en œuvre de l'algorithme d'ordonnancement proposé m'a conduit à développer un nouveau gestionnaire de données pour la grille. En effet, les deux gestionnaires de données précédemment offerts par DIET ne permettaient pas l'implantation de notre algorithme faute de possibilité de réplication avec DTM ou à cause de l'impossibilité d'une gestion explicite des données avec JuxMem. DAGDA, développé intégralement dans le cadre de cette thèse, permet à la fois une gestion directe des données sur la grille, mais il permet également à tous les utilisateurs de DIET de profiter des gains apportés par la réplication sans aucune modification de leurs applications existantes. Diverses optimisations ont également été introduites, améliorant les transferts de données mais aussi la gestion de celles-ci sur la plateforme. Conçu à la base comme un outil pour des expériences sur la grille, DAGDA offre de nombreuses possibilités de configurations qui peuvent s'avérer utiles sur une plateforme de production. On pourra ainsi choisir quel algorithme de remplacement de données utiliser pour une gestion transparente et implicite, la taille maximale d'occupation mémoire pour le stockage des données comme pour la transmission de celles-ci ou encore

l'espace de stockage à utiliser sur les disques des différents nœuds. De manière plus anecdotique, DAGDA permet également de mettre en place des sauvegardes de contextes à l'échelle de la grille permettant l'arrêt et la reprise d'expériences avec restauration des données sur l'ensemble des nœuds qui l'autorisent.

Ce travail de conception d'un gestionnaire de données m'a conduit à participer à la proposition de l'API de gestion de données dans le GridRPC auprès de l'OGF donnéee en annexe de ce document. DAGDA est aujourd'hui suffisamment avancé pour servir de base à l'implémentation des recommendations de ce document pour la conception d'une API GridRPC pour DIET.

6.1.3 DIET-BLAST

L'application DIET-BLAST a concrétisé les travaux théoriques sur l'ordonnancement conjoint des requêtes et des réplications de données sur la grille en permettant leur expérimentation sur une plateforme réelle. La conception de cette application en vue de tester ces algorithmes a nécessité l'ajout à DIET d'un nouveau mode d'ordonnancement : l'ordonnancement au niveau agent réalisé par un module compilé séparément et chargé dynamiquement par celui-ci au moment de la réception d'une requête. Ainsi, grâce à DAGDA et ces ordonnanceurs, nous avons pu tester sur une plateforme réelle les algorithmes précédemment étudiés sur une version modifiée du simulateur OptorSim. Le travail réalisé sur ce simulateur avec Antoine Vernois est désormais disponible en ligne sous le nom "Advanced OptorSim" [72].

6.2 Perspectives

Notre algorithme d'ordonnancement détermine le seuil de différence des fréquences relevées avec les fréquences théoriques au-delà duquel les données doivent être redistribuées de manière empirique. Or, ce seuil dépend essentiellement du temps nécessaire au déplacement des bases de données, c'est à dire de leurs tailles et de la bande passante disponible. L'étude approfondie des conséquences du changement des fréquences sur la distribution des bases permettrait de déterminer ce seuil de manière théorique et potentiellement plus efficace sur des plateformes dont les capacités réseau en terme de bande passante sont moindre. Par ailleurs, ces redistributions s'effectuent en simultané. Si la topologie des grilles de production courantes permet d'obtenir de bons résultats, on peut imaginer des topologies avec des points de contention qui nécessiterait un ordonnancement de ces communications. Nous avons procédé à des tests par simulation, mais le modèle réseau du simulateur OptorSim et la topologie de la plateforme Grid'5000 ne nous ont pas permis d'obtenir de résultats avec des différences suffisament significatives pour être exposés. On pourrait envisager de modifier le modèle réseau du simulateur ou encore d'utiliser un autre simulateur de grille tel SimGrid [31] pour étudier plus avant les conséquences de l'ordonnancement des transferts de données sur les performances de la plateforme. L'accumulation des données dans les domaines scientifiques aussi différents que la physique des hautes énergies, la biologie, la cosmologie et bien d'autres encore, conduiront certainement à devoir optimiser l'ordonnancement et la gestion des données sur les grilles de calcul. Notre approche, basée sur l'analyse statistique du comportement des utilisateurs de la grille pourrait alors être étendue à d'autres domaines dès lors qu'il est possible d'effectuer des préditions sur les soumissions

des différents type de tâches. Quoiqu'il en soit, nos résultats ont montré qu'une approche conjointe de l'ordonnancement des tâches et de la réplication des données peut apporter des améliorations conséquentes en terme de performance des grilles de calcul.

Le gestionnaire de données DAGDA, à la base spécifiquement destiné à la gestion des réplications des données biologiques, est aujourd'hui amené à devenir le gestionnaire par défaut pour tout type d'applications développées avec DIET. Plusieurs évolutions de celui-ci sont d'ores et déjà en cours :

- La possibilité de développer ses propres algorithmes de remplacement de données suivant le modèle des ordonnanceurs dynamiques au niveau agent.
- L'utilisation de plusieurs méthodes de métrologies afin de prédire plus précisément les temps de transfert qu'avec la méthode statistique actuellement utilisée.
- La possibilité de récupérer les données depuis plusieurs sources en simultané.

D'autres fonctionnalités pourront être introduites dans DAGDA telles des techniques de compression et de cryptographie, à la fois pour le stockage et le transfert des données. Toutes ces évolutions peuvent faire l'objet de recherches théoriques avec la possibilité de les mettre en œuvre immédiatement.

Le "Cloud Computing" consiste en l'exécution d'applications par l'intermédiaire d'internet, sans connaissance sur la localisation de celles-ci et sans connaissance nécessaire sur la localisation des données utilisées. Ce concept, comparable au concept des grilles informatiques, parfois même considéré comme identique à celui-ci, peut être apparenté au paradigme du GridRPC. Les technologies existantes comme la technologie Amazon S3 (Simple Storage Service) [90], basées sur les web-services, utilisent la réplication de données et des systèmes d'ordonnancement afin d'assurer la réactivité et la sureté nécessaire aux applications commerciales. Nos travaux peuvent par conséquent, facilement être adaptés à ce type de technologie.

Bibliographie

[1] L. Carey A. E. Darling and W. chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *ClusterWorld 2003*, 2003.

[2] G. Aloisio, M. Cafaro, S. Fiore, and M. Mirto. ProGenGrid : A Workflow Service Infrastructure for Composing and Executing Bioinformatics Grid Services. *Computer-Based Medical Systems, 2005. Proceedings. 18th IEEE Symposium on*, pages 555–560, 2005.

[3] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI–BLAST : a new generation of protein database search programs. *Nucleic Acids Res.*, 25 :3389–3402, 1997.

[4] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol*, 215(3) :403–410, 1990.

[5] Abelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Pushpinder Kaur Chouhan, Andréea Chis, Yves Caniou, Eddy Caron, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, Gaël Le Mahec, and Alan Su. Diet : New developments and recent results. In Lehner et al. (Eds.), editor, *CoreGRID Workshop on Grid Middleware (in conjunction with EuroPar2006)*, number 4375 in LNCS, pages 150–170, Dresden, Germany, August 28-29 2006. Springer.

[6] Abelkader Amar, Raphaël Bolze, Yves Caniou, Eddy Caron, Benjamin Depardon, Jean-Sébastien Gay, Gaël Le Mahec, and David Loureiro. Tunable scheduling in a GridRPC framework. *Concurrency & Computation : Practice & Experience*, 20 :1051–1069, 2007.

[7] J. Angele and M. Weiten. Semantic Technologies in the SIMDAT Grid Project. In Carole Goble, Carl Kesselman, and York Sure, editors, *Semantic Grid : The Convergence of Technologies*, number 05271 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[8] A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. McGough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. *GGF GFD*, 56, 2005.

[9] G. Antoniu, L. Bougé, and M. Jan. JuxMem : An adaptive supportive platform for data sharing on the grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, 2005.

[10] D.C. Arnold, S.S. Vahdiyar, and J.J. Dongarra. On the Convergence of Computational and Data Grids. *Parallel Processing Letters*, 11(2/3) :187–202, 2001.

[11] The Large Hadron Collider Project at CERN. The lhc website. http ://lhc.web.cern.ch/lhc.

[12] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic Acids Research*, 28(1) :45, 2000.

[13] P. Bala, J. Pytlinski, M. Nazaruk, V. Alessandrini, D. Girou, D.W. Erwin, D. Mallmann, J. MacLaren, J. Brooke, and J.F. Myklebust. BioGRID – An European Grid for Molecular Biology. In *Proc. 11th IEEE International Symposium on High Performance Distributed Computing (11th HPDC'02)*, page 412, Edinburgh, Scotland, UK, July 2002. IEEE Computer Society.

[14] G. Barish and K. Obraczke. World Wide Web caching : trends and techniques. *Communications Magazine, IEEE*, 38(5) :178–184, 2000.

[15] A. Bassi, M. Beck, T. Moore, J.S. Plank, M. Swany, R. Wolski, and G. Fagg. The Internet Backplane Protocol : a study in resource sharing. *Future Generation Computer Systems*, 19(4) :551–561, 2003.

[16] M. Bayer, A. Campbell, and D. Virdee. A GT3 based BLAST grid service for biomedical research. In *Proceedings of the UK e-Science All Hands Meeting*, volume 2004, pages 1019–1023, 2004.

[17] M. Beck, T. Moore, and J.S. Plank. An End-To-End Approach to Globally Scalable Network Storage. In *SIGCOMM '02 : Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–346. ACM, 2002.

[18] M. Beckerle and M. Westhead. GGF DFDL Primer. In *Global Grid Forum Data Format Description Language Working Group Memo, May*, 2004.

[19] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, and D.L. Wheeler. GenBank. *Nucleic Acids Research*, 35 :D21, 2007.

[20] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, IN Shindyalov, and PE Bourne. The protein data bank. *logo*, 58(1 Part 6) :899–907.

[21] R.D. Bjornson, A.H. Sherman, S.B. Weston, N. Willard, and J. Wing. TurboBLAST : A Parallel Implementation of BLAST Based on the Turbohub Process Integration Architecture. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS*, pages 183–190. TurboGenomics, Inc., 2002.

[22] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and Touché I. Grid'5000 : A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[23] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A. Roberts. Parallelization of Local BLAST Service on Workstation Clusters. *Future Gener. Comput. Syst.*, 17(6) :745–754, 2001.

[24] K.H. Buetow. Cyberinfrastructure : Empowering a "Third Way" in Biomedical Research. *Science*, 44 :417, 2003.

[25] M. Cannataro, C. Comito, F. Lo Schiavo, and P. Veltri. Proteus, A Grid Based Problem Solving for Bioinformatics : Architecture and Experiments. *IEEE Intelligent Informatics Bulletin*, 3(1) :7–18, 2004.

[26] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, 2005.

[27] E. Caron, A. Chis, F. Desprez, and A. Su. Design of Plug-In Schedulers for a GridRPC Environment. *Future Generation Computer Systems*, 24(1) :46–57, January 2008.

[28] E. Caron, P.K. Chouhan, and H. Dail. GoDIET : A Deployment Tool for Distributed Middleware on Grid'5000. In *EXPGRID workshop. Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Apllications and Tools. In conjunction with HPDC-15*, pages 1–8.

[29] E. Caron and F. Desprez. DIET : A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3) :335–352, 2006.

[30] R.L. De Carvalho and S. Lifschitz. Database Allocation Strategies for Parallel BLAST Evaluation on Clusters. *Dsitributed and Parallel Databases*, 13 :99–127, 2003.

[31] H. Casanova. Simgrid : a toolkit for the simulation of application scheduling. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 430–437, 2001.

[32] A. Chakrabarti, R.A. Dheepak, and S. Sengupta. Integration of Scheduling and Replication in Data Grids. Technical Report TR-0407-001, Infosys Tech. Ltd, July 2004.

[33] A. Chervenak, E. Deelman, I. Foster, W. Hoschek, A. Iamnitchi, C. Kesselman, M. Ripeanu, B. Schwartzkopf, H. Stockinger, and B. Tierney. Giggle : a framework for constructing scalable replica location services. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17. IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

[34] P.A. Covitz, F. Hartel, C. Schaefer, S. De Coronado, G. Fragoso, H. Sahni, S. Gustafson, and K.H. Buetow. caCORE : A Common Infrastructure for Cancer Informatics, 2003.

[35] B. Del Fabbro, D. Laiymani, J.M. Nicod, and L. Philippe. Data management in grid applications providers. In *Proc. of the 1st Intl. Conference on Distributed Frameworks for Multimedia Applications (DFMA'05)*, pages 315–322.

[36] Bruno Del Fabbro. *Contribution à la gestion des données dans les grilles de calcul à la demande : de la conception à la normalisation*. Thèse de doctorat, Université de Franche-Comté, France, novembre 2005.

[37] F. Desprez and A. Vernois. Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid. *Journal Of Grid Computing*, 4(1) :19–31, March 2006.

[38] R. Carvajal-Schiaffino D.G. Cameron, A.P. Millar, C. Nicholson, K. Stockinger, and F. Zini. Evaluating Scheduling and Replica Optimisation Strategies in OptorSim. In *4th International Workshop on Grid Computing (Grid2003)*. IEEE Computer Society Press, November 2003.

[39] F. Desprez E. Caron and G. Le Mahec. Dagda : An advanced data manager for the diet middleware. December, Indianapolis, Indiana USA 2008. To appear.

[40] F. Desprez E. Caron and G. Le Mahec. Parallelization and distribution strategies of large bioinformatics requests over the grid. In S.Q. Zheng Anu G. Bourgeois, editor, *ICA3PP'08 : Proceedings ot the International Conference on Algorithms and Architectures for Parallel Processing*, LNCS, pages 257–261, Ayia Napa, Cyprus, 2008. Springer.

[41] Embrace. Embrace website. http ://www.embracegrid.info.

[42] Open Grid Forum. Open grid forum website. http ://www.ogf.org.

[43] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G : A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3) :237–246, 2002.

[44] F. Gagliardi, B. Jones, F. Grey, M.E. Bégin, and M. Heikkurinen. Building an Infrastructure for Scientific Grid Computing : Status and Goals of the EGEE Project. *Philosophical Transactions : Mathematical, Physical and Engineering Sciences*, 363(1833) :1729–1742, 2005.

[45] GDMP. http ://project-gdmp.web.cern.ch/project-gdmp.

[46] Y. Georgiou, J. Leduc, B. Videau, J. Peyrard, and O. Richard. A Tool for Environment Deployment in Clusters and light Grids. *Second Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS'06)*, April 2006.

[47] Globus. Globus website. http ://www.globus.org.

[48] M. Gouy, C. Gautier, M. Attimonelli, C. Lanave, and G. Paola. ACNUC-a portable retrieval system for nucleic acid sequence databases : logical and physical designs and usage. *Bioinformatics*, 1(3) :167–172, 1985.

[49] J.S. Grethe, C. Baru, A. Gupta, M. James, B. Ludaescher, M.E. Martone, P.M. Papadopoulos, S.T. Peltier, A. Rajasekar, S. Santini, et al. Biomedical Informatics Research Network : Building a National Collaboratory to Hasten the Derivation of New Understanding and Treatment of Disease. *Stud Health Technol Inform*, 112 :100–9, 2005.

[50] L. Guy, P. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. Replica Management in Data Grids. In *Global Grid Forum*, volume 5, 2002.

[51] D.G. Higgins and P.M. Sharp. CLUSTAL : A package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1) :237–244, 1988.

[52] P. Honeyman, WA Adamson, and S. McKee. GridNFS : Global Storage for Global Collaborations. *Local to Global Data Interoperability-Challenges and Technologies, 2005*, pages 111–115, 2005.

[53] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data Management in an International Data Grid Project. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 77–90, 2000.

[54] European Bioinformatics Institute. European bioinformatics institute website. http ://www.ebi.ac.uk.

[55] N. Jacq, C. Blanchet, C. Combet, E. Cornillot, L. Duret, K. Kurata, H. Nakamura, T. Silvestre, and V. Breton. Grid as a Bioinformatic Tool. *Parallel Computing*, 30(9-10) :1093–1107, 2004.

[56] N. Jacq, J. Salzemann, Y. Legre, M. Reichstadt, F. Jacq, M. Zimmermann, A. Maass, M. Sridhar, K. Vinod-Kusam, H. Schwichtenberg, et al. Demonstration of In Silico Docking at a Large Scale on Grid Infrastructure. *STUDIES IN HEALTH TECHNOLOGY AND INFORMATICS*, 120 :155, 2006.

[57] Mathieu Jan. *JuxMem : un service de partage transparent de données pour grilles de calculs fondé sur une approche pair-à-pair*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, Novembre 2006.

[58] JXTA. Jxta website. https ://jxta.dev.java.net.

[59] JXTA-C. Jxta-c website. https ://jxta-c.dev.java.net.

[60] C. Kanz, P. Aldebert, N. Althorpe, W. Baker, A. Baldwin, K. Bates, P. Browne, A. van den Broek, M. Castro, G. Cochrane, et al. The EMBL Nucleotide Sequence Database. *Nucleic Acids Research*, 33 :D29, 2005.

[61] H. Kim, H. Kim, and D. Han. Performance Evaluation of BLAST on SMP Machines. *LECTURE NOTES IN COMPUTER SCIENCE*, 4331 :668, 2006.

[62] T. Kosar and M. Livny. Stork : Making Data Placement a First Class Citizen in the Grid. In *Proceedings of 24th IEEE Int. Conference on Distributed Computing Systems (ICDCS2004)*, Tokyo, Japan, March 2004.

[63] A. Krishnan. GridBLAST : a Globus-based high-throughput implementation of BLAST in a Grid computing framework : Research Articles. *Concurrency and Computation : Practice & Experience*, 17(13) :1607–1623, 2005.

[64] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. OceanStore : an architecture for global-scale persistent storage. *ACM SIGARCH Computer Architecture News*, 28(5) :190–201, 2000.

[65] D. Lavenier, H. Leroy, Mac Wing M., R. Andonov, M. Hurfin, P. Raipin-Parvedy, L. Mouchard, and F. Guinand. GénoGRID : An Experimental Grid for Genomic Applications. In *HealthGrid 2003*, Lyon, January 2003.

[66] The LHC Computing Grid Project (LCG). The lcg website. http ://lcg.web.cern.ch/LCG.

[67] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. *RFC4122, July*, 2005.

[68] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-Sequence Database Scanning on a GPU. *20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)(HICOMB Workshop), Rhode Island, Greece*, 2006.

[69] L. Milanesi and I. Merelli. High Performance GRID Based Implementation for Genomics and Protein Analysis. *Studies in health technology and informatics*, 120 :374–380, 2006.

[70] G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew, and A.J. Olson. Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function. *Journal of Computational Chemistry*, 19(14) :1639–1662, 1998.

[71] H. Ogata, S. Goto, K. Sato, W. Fujibuchi, H. Bono, and M. Kanehisa. KEGG : Kyoto Encyclopedia of Genes and Genomes. *Nucleic Acids Research*, 27(1) :29–34.

[72] Advanced OptorSim. Advanced optorsim website. https ://lipforge.ens-lyon.fr/projects/advancedoptor/.

[73] WR Pearson. Rapid and Sensitive Sequence Comparison with FASTP and FASTA. *Methods in enzymology*, 183 :63–98, 1990.

[74] K.T. Pedretti, T.L. Casavant, R.C. Braun, T.E. Scheetz, C.L. Birkett, and C.A. Roberts. Three Complementary Approaches to Parallelization of Local BLAST Service on Workstation Clusters. In *PaCT '999 : Proceedings of the 5th International Conference on Parallel Computing Technologies*, pages 271–282, London, UK, 1999. Springer-Verlag.

[75] G. Peng. CDN : Content Distribution Network. *Arxiv preprint cs.NI/0411069*, 2004.

[76] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol : Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[77] S. Podlipnig and L. Böszörmenyi. A survey of Web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4) :374–398, 2003.

[78] K. Ranganathan and I. Foster. Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids. *Journal of Grid Computing*, 1(1) :53–62, 2003.

[79] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith, and B. Wallenfelt. Massively Parallel BLAST for the Blue Gene/L. *High Availability and Performance Workshop*, 2005.

[80] M. Rarey, B. Kramer, T. Lengauer, and G. Klebe. A Fast Flexible Docking Method using an Incremental Construction Algorithm. *Journal of Molecular Biology*, 261(3) :470–489, 1996.

[81] Le réseau RENATER. Réseau national de télécommunications pour la technologie l'enseignement et la recherche. http ://www.renater.fr.

[82] J. Saltz, S. Oster, S. Hastings, S. Langella, T. Kurc, W. Sanchez, M. Kher, A. Manisundaram, K. Shanbhag, and P. Covitz. caGrid : Design and Implementation of the Core Architecture of the Cancer Biomedical Informatics Grid. *Bioinformatics*, 22(15) :1910, 2006.

[83] J. Salzemann, N. Jacq, and V. Breton. Replication and Update of Molecular Biology Databases. *IEEE Transactions on Nanobioscience*, 6(2), June 2007.

[84] J. Salzemann, N. Jacq, and G. Le Mahec. Breton, V. Replication and Update of Molecular Biology Databases in a Grid Environment. *Proceedings of the NET-TAB2006 : Santa Margherita*, pages 33–37, 2006.

[85] R. Sandberg. Sun Network Filesystem Protocol Specification. *Sun Microsystems, Inc. Technical Report*, 1985.

[86] R. Sandberg. The Sun Network File System : Design, Implementation and Experience. In *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, 1986.

[87] F. Sanger, S. Nicklen, and AR Coulson. DNA Sequencing with Chain-Terminating Inhibitors. *Proceedings of the National Academy of Sciences*, 74(12) :5463–5467, 1977.

[88] N. Santos and B. Koblitz. Metadata services on the Grid. *Nuclear Inst. and Methods in Physics Research, A*, 559(1) :53–56, 2006.

[89] B. Segal. Grid Computing : The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, pages 15–20. Lyon, 2000.

[90] Amazon S3 (Simple Storage Service). Amazon web services website. http ://aws.amazon.com/s3.

[91] C. Stark, B.-J. Breitkreutz, T. Reguly, L. Boucher, A. Breitkreutz, and M. Tyers. BioGRID : A General Repository for Interaction Datasets. *Nucleic Acids Research*, 34(Database-Issue) :535–539, 2006.

[92] R.D. Stevens, A.J. Robinson, and C.A. Goble. myGrid : Personalised Bioinformatics on the Information Grid, 2003.

[93] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 102–110, 2002.

[94] F. Desprez V. Breton, E. Caron and G. Le Mahec. *Handbook of Research on Computational Grid Technologies for Life Sciences, Biomedicine and Healthcare*, chapter High Performance BLAST over the Grid. IGI Global Press, 2008. To appear.

[95] S. Venugopal, R. Buyya, and K. Ramamohanarao. A Taxonomy of Data Grids for Distributed Data Sharing, Management, And Processing. *ACM Comput. Surv.*, 38(1) :3, 2006.

[96] Antoine Vernois. *Ordonnancement et réplication de données bioinformatiques dans un contexte de grille de calcul*. Thèse de doctorat, École Normale Supérieure de Lyon, Lyon, France, Octobre 2006.

[97] C. Wang, B.A. Alqaralleh, B.B. Zhou, M. Till, and A.Y. Zomaya. A BLAST Service Built on Data Indexed Overlay Network. In *e-Science*, pages 16–23, 2005.

[98] J. Wang. A survey of web caching schemes for the internet. SIGCOMM Comput. *Commun. Rev*, 29(5) :36–46, 1999.

[99] X. Wu and C.W. Tseng. *Parallel Computing for Bioinformatics and Computational Biology*, chapter Searching Sequence Databases Using High-Performance BLASTs, pages 211–232. Parallel and Distributed Computing. Wiley, 2006.

[100] K. Yagoub, D. Florescu, V. Issarny, and P. Valduriez. Caching Strategies for Data-Intensive Web Sites. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 188–199. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2000.

[101] C.-T. Yang, Y.-L. Luo, and C.-L. Lai. Designing Computing Platform for BioGrid. *Int. J. of Computer Applications in Technology*, 22 :3–13, April 15 2005.

Annexe A

Proposition d'API de gestion de données dans le GridRPC

Cette annexe fournit la proposition d'API de gestion de données pour le GridRPC que nous avons proposée à l'Open Grid Forum. La conception d'une telle API doit s'efforcer de ne pas introduire de limitations ou d'obligation de fonctionnement des intergiciels de grille. Elle doit par ailleurs, et tant que possible, assurer le maximum de compatibilité avec l'API GridRPC déjà existante.

Proposal for a Data Management API within the GridRPC

Y. Caniou, E. Caron, F. Desprez, G. Le Mahec, H. Nakada and Y. Tanimura

Status of This Memo

This document (**version 0.8**) provides a recommendation to the Grid community on a proposed model and API for data management to GridRPC. Distribution is unlimited.

Copyright Notice

Copyright © Open Grid Forum (2007). All Rights Reserved.

Abstract

This document follows the document produced by the GridRPC-WG on GridRPC Model and API for End-User applications [1]. This new document aims at completing the GridRPC API with Data Management mechanisms and API.
This document is not intended to provide features and capabilities for building data management middleware. The goal of this document is to complete the GridRPC set of functions and definitions to allow users to manipulate their data. The motivation for this document is to provide explicit functions to manipulate the exchange of data between a GridRPC platform and a client since (1) the size of the data used in Grid applications may be large and useless data transfers must be avoided ; (2) data are not always stored on the client side but may be made available either on a storage resource or within the GridRPC platform.

Introduction

The goal of this document is to define a data management extension to the GridRPC API for End-User applications. As for the GridRPC API document [1], it is out of the scope of this document to discuss the implementation of data management mechanisms inside a GridRPC platform or on a data storage server.

The motivation of the data management extension is to provide explicit functions to handle data exchanges between a data storage service, a GridRPC platform, and the client. The GridRPC API defines a RPC mechanism to access Network Enabled Servers. However, an application needs data to run and generates some output data, which have to be transfered. As the size of the data may be large in grid environments, it is mandatory to optimize the transfers of large data and avoid useless exchanges. Several cases may be considered depending on where data are stored : on an external data storage, inside the GridRPC platform or on the client side. In all these cases, the knowledge of "what to do with these data ?" is owned by the client. Then, the GridRPC API must be extended to provide functions for explicit and simple data management.

We firstly present a motivation for the data management and in Section A, the proposed data management model is introduced. The main contribution of this document is given in Section A where we describe our proposal for a data management API.

Data Management motivation

The main motivation of the data management extension is to provide a way to explicitly manage the data and their placement in the GridRPC model. With the help of this explicit management, the client will avoid useless transfers of large data. However, the client may not want to, or may not know how to manage data. Then, the default behavior of the GridRPC Data Management extension must be in accordance with the GridRPC API document. To illustrate the motivation of data management, we give now some examples describing when it can be used.

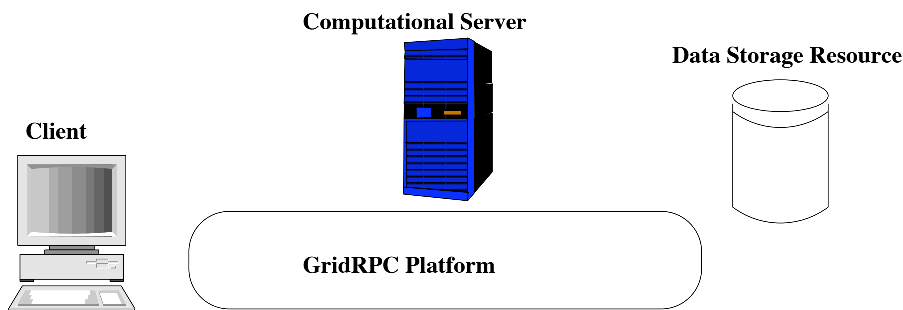

FIG. A.1 – Data locations in the GridRPC model

In a GridRPC environment, data can be stored either on a client host, on a data storage server, on a computational server or inside the GridRPC platform, as shown in Figure A.1. When clients do not need to manage their data, then the basic GridRPC API is sufficient. On each `grpc_call()`, data is transfered between a client and the computational server used. Once the computation performed, results are sent back to the client. However, to minimize

data transfers, clients need data management functions.

Next, we explain two different cases concerning external data and internal data :

– External data are placed on servers, like data repositories. These servers are not registered inside the platform but can be directly accessed to read/write data. The use of such data implies several data transfers if the client uses the basic GridRPC API : the client must download the data and then send it to the GridRPC platform when issuing the call to `grpc_call()`. One of these transfers should be avoided : the client may just give a data reference (or *handle*) to the platform/server and the transfer is completed by the platform/server. Consider a client, with small data storage capacities, that needs to issue a request on large data stored in a data storage server. It is costly, and it may be not possible to send the data first to the client before issuing the call. The client could also need to directly store its results from the computational server to the data storage, or share an input or output data with other users of the GridRPC platform. Examples of such Data Storage servers are IBP [2] or SRB [3]. Among the different available examples of this approach, we can cite : (1) the Distributed Storage Infrastructure of NetSolve [6] ; (2) the utilization of JuxMem in DIET [8]. This approach is well suited for, but not limited to, long life persistent data.
– Internal data are managed inside the GridRPC platform. Their placement depends on computations and it may be transparent to clients : in this case, the GridRPC middleware can manage data. Temporary data, generated by request sequencing [4], are examples of internal data. For instance, a client issues two calls to solve the same problem and the second call use input or output data from the first call. This is the case if we solve $C =^t (A \times B)$, where A and B are matrices. If the client do not find a solver which computes these two operations in one step, then he must issue two calls : $W = A \times B$ and $C =^t W$. But the value of W is of no interest for him. Then, this matrix should not be sent back to the client host. Temporary data should be leaved inside the platform, close to or on the computational server, when clients do not need it. Other cases of useless temporary data occur when the results of a simulation are sent to a graphical viewer as done in most Problem Solving Environments. Among the examples of internal data management, we can cite the Data Tree Management infrastructure used in DIET [5], and the data layer omniStorage in omniRPC [7]. This approach is suitable for, but not limited to, intermediate results to be reused in case of request sequencing.

GridRPC Data Management model

As exposed in the previous section, we consider two different types of data : external and internal data.

In the external data case, data are explicitly stored on a storage depot. Clients manage explicitly their data. When clients invoke a server, they give a data reference to identify the data used for the computation. A client can use already existing data by just providing the data identification given by the storage service.

In the internal data case, the data management service tries to read/write the data inside the GridRPC platform, from the client side or on a computational server. In this case, either the client knows where the data is stored and can manage the transfer (he can use the same calls than the ones to manage external data stored on a storage server), either the data is

transparently managed by the GridRPC middleware, in which case the middleware provides mechanisms for transfers between computational servers.

These two approaches are complementary in the data management model proposed here. The GridRPC platform and the data storage interact to implement data transfers. Note that some additional functionalities which are not addressed in this document, can be designed, such as : reusable data generated by a computation could be stored during a TTL (Time To Leave) on the computational server before being sent to data storage servers ; or when the storage capacity of a computational server is overloaded, it may be sent to another data storage server.

In both cases, it is mandatory to identify each data. All data stored either in the platform or on storage servers will be identified by **Data Handles** and **Storage Information**. Without lack of generality, we define the *GridRPC data type* as either *the data used for a computational problem*, either both a *Data Handle and storage information*. Indeed, when a computational server receives a GridRPC data which does not contain the computational data, it must know the unique name of the data with the Data Handle, and must know its location to get it and where the client wants to save it after the computation. Thus storage information must record the original location of the data and the destination of the data.

Data Management API

In [1], data used as input/output parameters are provided within the `<varargs>` notation of the `grpc_call()` and `grpc_call_async()` functions. Without lack of generality, and in order to propose an API independent of the language of implementation, we refer to `grpc_data_t` as the type of such variables. Thus, in the following, a `grpc_data_t` is any kind of data, or contains a reference on the computational data, which we call a *Data Handle*, as well as some *Storage Information*.

Next, we firstly define some data types for GridRPC data management. We present afterwards the different functions to managed them, composing the proposed API for GridRPC Data Management.

Note that in the following, we refer to "GridRPC data" to designate the generic data which is used in the GridRPC calls and "data" to designate the content data.

GridRPC data types

We introduce here the notion of a GridRPC data which at least includes the data or a data handle, and may contains some information about the data itself (*e.g.,* type, size) as well as information on its location and the protocol used to access it (*e.g.,* the URI of a specific server, a link with a Storage Resource Broker, containing the correct protocol to use). A data handle is essentially a unique reference to a data that may reside anywhere. Data and data handles can be created separately. By managing GridRPC data with data handles, clients do not have to know where data are currently stored.

The grpc_data_t type

A data in a GridRPC middleware is defined by the **grpc_data_t** type. It relies on a data, or on a **grpc_data_handle_t** type and a **grpc_data_storage_info_t** type to access it.

Consequently, the **grpc_data_t** type can be seen as a structure containing the data itself and/or a **grpc_data_handle_t**. The **grpc_data_storage_info_t** type can also be stored in the **grpc_data_t** structure or it can also be stored and managed inside the GridRPC data middleware.

The grpc_data_handle_t type

A variable of this type represents a specific data. It is allocated by the user. After a *data handle* is initialized, it may be used in a server invocation. The lifetime of a *data handle* is determined when the user invalidates it. Data handles are created/allocated by simply creating a variable of this type.

The grpc_data_storage_info_t type

Variables of this type represent information on a specific data which can be local or remote. It is at least composed of :

- Two NULL-terminated lists of URIs, one to access the data and one if the data has to be stored somewhere from this server (for example, an OUT parameter to transfer at the end of a computation).
- Information concerning the mode of management. For example, data management is defaulted to the one of the standard GridRPC paradigm, but it can be noted for example as `GRPC_PERSISTENT`, which corresponds to a transparent management by the GridRPC middleware.
- Information concerning the type of the data, as well as its size.

grpc_data_type_t	Definition
GRPC_INT	integer
GRPC_DOUBLE	double
GRPC_COMPLEX	complex
GRPC_STRING	string
GRPC_FILE	file
GRPC_ARRAY_OF_INT	array of int
GRPC_ARRAY_OF_DOUBLE	array of double
GRPC_ARRAY_OF_COMPLEX	array of complex
GRPC_MATRIX_OF_INT	matrix of int
GRPC_MATRIX_OF_DOUBLE	matrix of double
GRPC_MATRIX_OF_COMPLEX	matrix of complex
GRPC_CONTAINER_OF_GRPC_DATA	container of grpc_data_t

TAB. A.1 – Definition of grpc_data_type_t codes

Details on Storage Information

- URI : it defines the location where a data is stored. It can be built like "protocol ://machine_name :port/path_to_data" and thus, contains at least four fields. Some straightforward examples are given in Section A and several full examples of utilization can be found in Section A.
  - char * protocol : one of the token {"file", "nfs", "memory", "ibp", "local_fs", "middleware", "http"}, and gives some information on how to access the data (the list is not exhaustive).
  - char * hostname : the name of the server on which resides the data.
  - int port : the port to use to access the data.
  - char * path : the full path of the data or an ID.
- The management mode is an enumerated type `grpc_data_mode_t` defined by the client. It is useful to set the behavior of the data on the platform. It is related to the following policy values. If the middleware does not handle the given behavior, it throws an error.
  - GRPC_VOLATILE : used when the data may not be kept inside the platform after a computation (can be considered as the default usage for GridRPC API). Still, the Data Management middleware can keep the data in the system, migrate and replicate it, for later use. Hence, potential coherency issues may arise.
  - GRPC_UNIQUE_VOLATILE, used when the data must not be kept inside the platform after a computation for security reason for example (can be considered as the default usage for GridRPC API).
  - GRPC_PERSISTENT : used when a data has to be kept on the platform. The data is not sent back to the client. Moreover, the data item can migrate or be replicated between servers depending on scheduling decisions, and potential coherency issues may arise.
  - GRPC_STICKY : used when a data is kept inside the platform but cannot be moved between the servers. In that case the data is not given back to the client after computation. This is used if the client needs that data in the platform for a second computation on the same server for example. Note that in this case, the data can be replicated and that potential coherency issues may arise.
  - GRPC_UNIQUE_STICKY : used when a data is kept inside the platform but cannot be moved between the servers. In that case the data is not given back to the client after computation. This is used if the client needs that data in the platform for a second computation on the same server for example. Note that in this case, the data cannot be replicated for security reason for example.
- The type of the data is an enumerated type `grpc_data_type_t` defined by the client : it describes the type of the data, for example GRPC_DOUBLE, GRPC_ARRAY_OF_INT, etc., as exposed in Table A.1. We can note that a special `grpc_data_t` can contain other `grpc_data_t`. That way, the user relies on the GridRPC Data Middleware to transfer a set of `grpc_data_t`. The matter of how to implement it by an array, a list or anything else is GridRPC Data Middleware dependant, then not in the purpose of this document.

Function specific types

In this section, we describe some types that are only used in a given function. They are enumerated types, and are given here for commodity reasons.

The grpc_data_resolution_mode_t type. Used only in the `grpc_data_init()` function, this type is defined by the set { `GRPC_RESULT_RETURN`, `GRPC_NO_RESULT`, `GRPC_RESULT_DEFAULT` }. It lets the user precise if he wants to get the result at the end of a computation like the default GridRPC API behavior of `grpc_call()` and `grpc_call_async()`, or if data are at least momently kept on the computing resource.

The grpc_completion_mode_t type. This type is used in `grpc_data_wait()` and is defined by the enumerated type { `GRPC_WAIT_ALL`, `GRPC_WAIT_ANY` } which can be extended. It is used to detail the behavior of the waiting process : the function can wait for one or all transfers concerning data involved during the call to `grpc_data_wait()`.

The grpc_data_diffusion_t type. This type is only used in the `grpc_data_write()` function. Its purpose it to define some possible diffusion mechanisms when writing a data, if the GridRPC middleware or the data middleware on which it relies has some knowledge about the inner topology, etc.

It can be defined by the set { `GRPC_DEFAULT_DIFFUSION`, `GRPC_UNICAST`, `GRPC_BROADCAST` }, which can be extended.

The grpc_data_info_type_t type. This type is only used with the `grpc_data_getinfo()` function to define the wanted information. It is an enumerated type defined with the following values (which can be extended) :

– GRPC_HANDLE
– GRPC_INPUT_URI
– GRPC_OUTPUT_URI
– GRPC_MANAGEMENT_MODE (used to know if a data is for example `GRPC_VOLATILE`, etc.)
– GRPC_SIZE
– GRPC_TYPE (used to know to which type of the language of implementation the data corresponds)
– GRPC_LOCATIONS_LIST
– GRPC_RESOLUTION_MODE (used to know if a copy of the result is returned after a computation)
– GRPC_STATUS (used to know if a grpc data is “GRPC_IN_PLACE” or “GRPC_TRANSFERING”)
– GRPC_COHERENT (used to know if the considered data is managed by a Data Management middleware that ensures coherency in all replicas (and then, if this replica may or is up-to-date) – should be dependent on the locations)

Note : Information is managed by the GridRPC Data Management API, which relies on at least one Data Management middleware. Then, information concerning a data can be stored within the GridRPC Data Management middleware, and/or within the `grpc_data_t` type. Nonetheless, this document does not focus on implementation.

Examples of use

- A GridRPC data corresponding to an input matrix stored in memory can partly be constructed with the information of `protocol=memory`, `port` is a null string, the machine name is the one of the localhost and path_to_data is the path used to access the data in memory (such as a pointer in C language, or a key that lets the GridRPC API either make the correspondance with the correct input to give to the data middleware, either the key used by the data middleware).
- The URI "http ://myName :/myhome/data/matrix1" corresponds to the location of a file named matrix1, which we can access on the machine named myName, with the `http` protocol. Typically, the data, stored as a file, can be downloaded with a command like "`wget http://myName/myhome/data/matrix1`".

Data Management functions

Data handles are provided by the GridRPC Data Management middleware. They must be unique, and the middleware must record some information about the data, such as the location, the size, etc. The **init** function sets the data handle to the data it identifies, while the user provides needed information concerning the location on where the data is stored and where it has to be stored after the computation. Using this function, all the semantic needed to provide data management and data persistence can be covered.

Data exchanges between client and explicit locations (computational servers or storage servers) are done using the *asynchronous* **read** and *asynchronous* **write** functions. Consequently, the GridRPC data can also be **inspected**, or probed, to get more information about the status of the data or its location. Functions are also given to **wait** after the completion of some transfers. Finally, one can **unbind** the handle and the data, and **free** the GridRPC data.

To provide identification of long lived data, data handles should be **saved** and **restored**, for instance in a file. This will allow two different users to share the same data. Security and data life cycle management issues are not of the API concerns.

Examples of the use of this API are given in Section A.

The init function

The **init** function initializes the *GridRPC data* with a specific data. This data may be available locally or on a remote storage server. Both identifications can be used. GridRPC data referencing input parameters must be initialized with identified data before being used in a `grpc_call()`. GridRPC data referencing output parameters do not have to be initialized.

Function prototype :

```
grpc_error_t grpc_data_init( grpc_data_t * data,
                             char ** list_of_URI_input,
                             char ** list_of_URI_output,
                             grpc_data_type_t variable_type,
                             grpc_data_mode_t storage_mode,
                             grpc_data_resolution_mode_t resolution_mode );
```

list_of_URI_input and **list_of_URI_output** parameters are `NULL`-terminated lists of strings, which give the different locations on where to transfer the data from, and the lo-

cations on where to possibly transfer the data to, as explained previously. Hence, a list describes all the available locations known by the client, in order for the GridRPC Data Management middleware to possibly implement some efficient and fault-tolerant mechanisms to perform a choice among all the proposed selections (and the ones eventually known by the Data Management middleware). In sake of simplicity, one can imagine that the default behavior would be a sequential try until the transfer to/from one of them can be achieved.

If different than `GRPC_RESULT_RETURN`, **resolution_mode** may be used to change the default behavior of the standardized GridRPC API functions `grpc_call()` and `grpc_call_async()`. Indeed, if set to `GRPC_NO_RESULT` and if **data** is used as an output argument for the solve, the result is considered as an intermediate result and then is not sent back to the client but stored on the resource (still, if **List_of_URI_output** is not `NULL`, then result is also transfered to the given locations).

Remarks :

- If `GRPC_UNIQUE_VOLATILE` is used at the same time than a **resolution_mode** equal to `GRPC_NO_RESULT`, then if **List_of_URI_output** is `NULL`, the result is lost.
- **storage_mode** is by default set to `GRPC_VOLATILE`. If another value is given, for example `GRPC_STICKY`, then the storage management is applied to all locations if needed. If a user wants to use the same handle in order to use the same data possibly being managed differently on numerous locations (for example `GRPC_STICKY` on given resources and `GRPC_VOLATILE` on others), then he has to do it when calling the `grpc_data_write()` function.
- Some of the parameters, such as the output parameter, can be unset.
- If the function is called with a grpc_data_t which has been used in a previous call, fields corresponding to information already given are overwritten.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_INVALID_MODE	Specified location is not valid
GRPC_INVALID_RESOLUTION_MODE	Resolution mode is either not valid or not managed
GRPC_OTHER_ERROR_CODE	Internal error detected

Functions specific to a special mode or data

Mappings of memory location to given names.

If he wants to use a data which is in memory, the user must provide some name in the URIs in the input or output fields which has to be understood by the GridRPC Data Management layer in the GridRPC system, in addition of the use of the *memory* protocol. For this reason, we provide here two functions :

Function prototype :

```
grpc_error_t grpc_data_memory_mapping_set( char * key, void * data );
grpc_error_t grpc_data_memory_mapping_get( char * key, void * data );
```

The function `grpc_data_memory_mapping_set()` is used to make the relation between a data stored in memory and a `grpc_data_t` data when the *memory* protocol is

used : the aim is to set a keyword that will be used in the URI used for example during the initialization of the data.

Containers of `grpc_data_t` management functions.

In order to facilitate the use of some special structures like lists or arrays of `grpc_data_t` variables, the two following functions let the user manipulate them at a higher level and without knowing the contents of the structures.

Function prototype :

```
grpc_error_t grpc_data_container_set( grpc_data_t * container, int rank,
                                      grpc_data_t * data );
grpc_error_t grpc_data_container_get( grpc_data_t * container, int rank,
                                      grpc_data_t * data );
```

The variable **container** is necessarily a grpc_data_t of type `GRPC_CONTAINER_OF_GRPC_DATA`. **rank** is a given integer which acts as a key index, and **data** is the data that the user wants to add in or get from the container. Note that getting the data does not remove the data from the container. Furthermore, the container management is free of implementation.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_TYPE	Specified type is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

The write function

This function writes a GridRPC data identified by `data` to the output location set during the init call in the output parameters fields. For commodity reasons, a diffusion mode and a list of additional servers on which the data has to be uploaded can be provided. Some broadcast/multicast mechanisms can then be implemented in the GridRPC data middleware in order to improve performance. The diffusion mode can be used by more intelligent data middleware to diffuse a data in a broadcast manner for example.

A user may want to be able to transfer data while computations are done. For example, if a computation can begin as soon as some data are downloaded but needs all of them to finish, the management of data must use **asynchronous mechanisms** as default behavior. Then, this function initiates the call for the transfers and returns immediately after.

Function prototype :

```
grpc_error_t grpc_data_write( grpc_data_t * data, grpc_data_diffusion_t mode,
                              char ** list_of_URI,
                              grpc_data_mode_t * list_of_management_modes );
```

mode is the mode of diffusion of the data. It can be used if the user may want to benefit from the eventual possibility of the data middleware to manage some kind of point to point protocols (such as using a spanning tree). **list_of_management_modes** is a NULL-terminated list with the same number of items than **list_of_URI**. For each URI describing the hostname,

the protocol used to access the data, etc., a management mode can be specified : a data can be flagged `GRPC_STICKY` on given resources. Hence, **list_of_management_modes** can be used to set different management policies on some resources (for example, set the data as `GRPC_STICKY` to a set of a resources and `GRPC_PERSISTENT` to the others) while possibly benefiting of an "aggressive" write as the data is the same everywhere.

Remarks :
- If **list_of_management_modes** is set to NULL, the management mode of the data is the one specified during the initialization of the data, else the management mode is overridden.
- No information is given as when the transfer will indeed begin.
- If a user needs to know if the transfer is completed on one or another server (or all), he can use the `grpc_data_getinfo()` function.
- If a user wants to wait of the completion of one or more transfers, he can use the `grpc_data_wait()` function.
- If the data middleware (*e.g.,* the GridRPC middleware or the data middleware on which it relies) does not manage coherency between the duplicates on the platform, a correct call to this function can be useful to ensure that all copies are up to date.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

To discuss :
- A special ERROR if protocol not set or not correct should be given ?
- A special ERROR if at least on server does not respond ?

The read function

This function reads a data stored inside the platform or on a specific storage server. A user may want to be able to transfer data while computations are done. For example, if a computation can begin as soon as some data are downloaded but needs all of them to finish, the management of data must use **asynchronous mechanisms** as default behavior. Then, this function initiates the call for the transfers and returns immediately after.

Function prototype :

```
grpc_error_t grpc_data_read( grpc_data_t * data );
```

Remarks :
- No information is given as when the transfer will indeed begin.
- If a user needs to know if the transfer is completed on one or another server (or all), he can use the `grpc_data_getinfo()` function.
- If a user wants to wait of the completion of one or more transfers, he can use the `grpc_data_wait()` function.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

The grpc_data_wait() function

For convenient reasons, the functions `grpc_data_write()` and `grpc_data_read()` are asynchronous. Hence, a user have the possibility to perform overlap transfers with computation and try to realize transfers in parallel. This function can then be used by the user to wait for the completion of one or several transfers.

Function prototype :

```
grpc_error_t grpc_data_wait( grpc_data_t ** list_of_data,
                             grpc_completion_mode_t  mode);
```

Depending on the value of **mode** (`GRPC_WAIT_ALL` or `GRPC_WAIT_ANY`), the call returns when all or one of the data listed in **list_of_data** is transfered, which means that for a given data, all transfers involved for the input *or* output part are finished.

Remarks :
- If **list_of_data** is `NULL`, then either one or all data (depending on the value of **mode**) are transfered since the call to `grpc_initialize()`.
- The use of this function can be done in such a way that the server can test if data are in place (*i.e.*, that transfers involved in the `grpc_data_write()` on the client part have been completed) before doing anything. If the user performs a `grpc_data_read()` of a `grpc_data_t` whose transfer has not been completed, the behavior is depending on the data middleware which manage the data : if the middleware implements some stamps mechanisms, then no problem will occur.
- This function considers only the information that the user is aware of : if the data is shared between different users, then a call to `grpc_data_wait()` returns depending on the input of the user that has performed the call. Hence, the call will not depend on an other user action.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is not valid
GRPC_OTHER_ERROR_CODE	Internal error detected

The unbind function

When the user does not need a handle anymore, but knows that the data may be used by another user for example, he can unbind the handle and the GridRPC data by calling this function without actually freeing the GridRPC data on the remote servers.

Function prototype :

```
grpc_error_t grpc_data_unbind( grpc_data_t data );
```

After calling this function, `data` does not reference the data anymore.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

The free function

This function frees the GridRPC data identified by `data` on a subset or on all the different locations where the data is stored, and unbind the handle and the data. This function may be used to explicitly erase the data on a storage resource.

Function prototype :

```
grpc_error_t grpc_data_free( grpc_data_t * data, char ** URI_locations );
```

If **URI_locations** is `NULL`, then the data is erased on all the locations where it is stored, else it is freed on all the location contained in the list of URI.

After calling this function, **data** does not reference the data anymore.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

A function to get information on a grpc_data_t variable

This function let the user access information about an instanciation of a **grpc_data_t**. It returns information on data characteristics, status, locations, etc.

Function prototype :

```
grpc_error_t grpc_data_getinfo( grpc_data_t * data,
                                grpc_data_info_type_t info_tag,
                                char * server_name,
                                char ** info );
```

The kind of information that the function gets is defined by the **info_tag** parameter. A server name can be given to get some data information dependent on the location of where is the data (like `GRPC_STICKY`). **info** is a `NULL`-terminated list containing the different available information corresponding to the request.

Remarks :

- For values equal to `GRPC_INPUT_URI` and `GRPC_OUTPUT_URI`, the returned list is the considered to be information on the grpc data in the system, not only the information got locally for the handle (or stored in the `grpc_data_t`).
- **server_name** can be set to `NULL` (default behavior). In that case, if the user tries to access the information of the mode (`GRPC_STICKY` for example) and the data has different management mode on the platform, then the value `GRPC_UNDEFINED` may be returned.
- If **info_tag** equals to `GRPC_STATUS`, then **info** can be one of the "GRPC_IN_PLACE" and "GRPC_TRANSFERING" value)

Note that in case of **info_tag** is set to `GRPC_HANDLE`, information is of no use to manage data with the given API : handles are initialized in the init call function, stored in the grpc_data_t.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

The load_data and save_data functions

In order to communicate a reference between Grid users, for example in case of large size data, one should be able to store a GridRPC data. The location can then be shared, for example by mail, and one can be able to load the corresponding information.

Function prototype :

```
grpc_error_t grpc_data_load( grpc_data_t * data, char * URI_input );
grpc_error_t grpc_data_save( grpc_data_t * data, char * URI_output );
```

These functions are used to load/save the data descriptions. Even if the GridRPC `data` contains the data in addition to metadata management informations (data handle, size, type, etc.), only data informations have to be saved in the location. The format used by these functions is let to the developer's choice. The way the informations are shared by different middleware is out of scope of this document and should be discussed in an interoperability recommandation document.

Error code identifier	Meaning
GRPC_NO_ERROR	Success
GRPC_INVALID_HANDLE	Specified handle is invalid
GRPC_INVALID_DATA	Specified data is invalid
GRPC_OTHER_ERROR_CODE	Internal error detected

Appendix 1 : Examples of use of the API

In this section, we give examples of the data management API usage to illustrate its interest. Depending on the passing mode of the arguments (data), we show how to optimize data placement and avoid useless transfers. We do not consider these examples as an exhaustive list but they can help to understand the way to build a data management API in GridRPC middleware.

Basic example

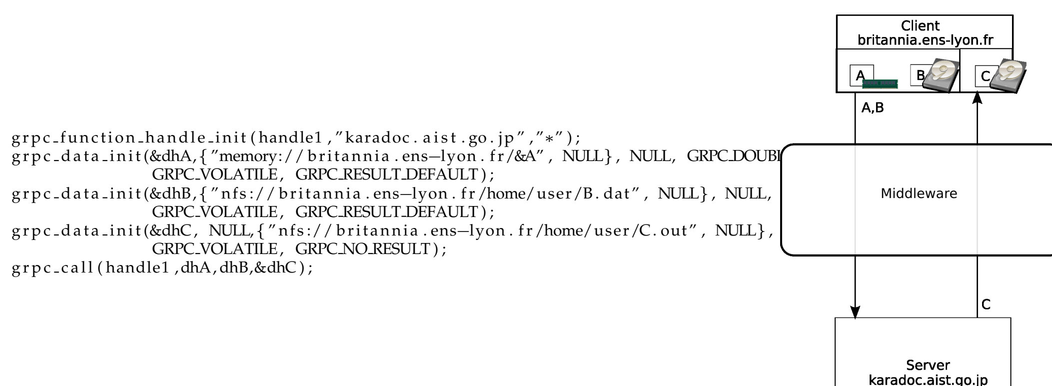

FIG. A.2 – Simple RPC call with input and output data.

In this example (see Figure A.2), we show an example on how to use the GridRPC data management functions when the data does not need to be stored inside the platform or on a storage resource. This example corresponds to the default behavior of the data management performed in the GridRPC paradigm, but conducted by the client with data handles.

Input data

Here, we illustrate the way to send a local data (in memory and on disk) to the GridRPC platform. In this example, the client issues a call with two input data `A` and `B`. `A` and `B` are

local to the client. Note that we use the `&A` notation in the URI for commodity reason, but the real memory address should be given here.

Output data

Output data `C` is sent back to the client because in our case, no data conservation is needed. In this example, the client issues a call with `A` and `B` as input data and `C` as output data. We note that the example uses the `GRPC_NO_RESULT` resolution mode during the initialization of the `grpc_data_t` because the client manage the output data : the given URI is used to manage the data. If `GRPC_RESULT_RETURN` was used, there could be two transfers : one more because of the normal GridRPC behavior of the GridRPC middleware which transfers output data locally on the client (in that case, the client may have to question the GridRPC middleware to get how to access the data).

Example with external storage resources

In this example we show how to use the GridRPC data management when the data is stored on an external data repository.

Figure A.3 shows how to manage external data repository as IBP or SRB.

```
grpc_function_handle_init(handle12,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA,{"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", NULL}, {"NFS://britannia.ens-lyon.fr/home/user/A.dat", NULL}, GRPC_DO
               GRPC_VOLATILE, GRPC_RESULT_DEFAULT);
grpc_data_init(&dhB,{"SRB://carmelide.ens-lyon.fr/COLLECTION/Simulations/B.dat", NULL}, {"IBP://kaamelott.cs.utk.edu/1213#B.dat/WriteKey/WRITE"
               GRPC_DOUBLE, GRPC_VOLATILE, GRPC_RESULT_DEFAULT);
grpc_data_init(&dhC, NULL,{"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL}, GRPC_DOUBLE, GRPC_VOLATILE, GRPC_NO_RESULT);
grpc_call(handle1,dhA,dhB,&dhC);
```

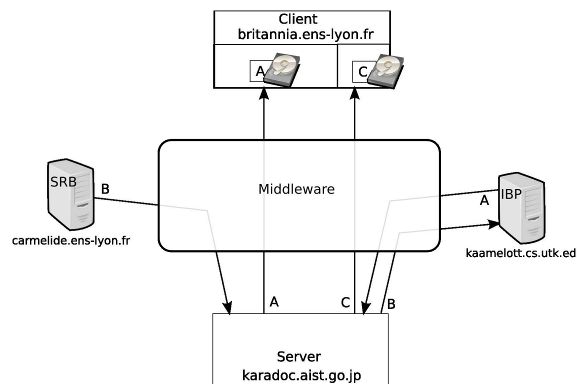

FIG. A.3 – Simple RPC call with input and output data using external storage resources.

Input data

Here, we illustrate the way to send a remote data stored on SRB or IBP server to the GridRPC platform. In this example, the client issues a call with two input data `A` and `B`. `A` is available on SRB repository and `B` is available on IBP repository. With the input and output parameters from the `grpc_data_init()` function we can move the data from a repository to another :

- A is read from SRB server and will be sent to the client.
- B is read from IBP server and will be sent to SRB server.

Output data

Output data C is sent back to the client.

Example with persistence

In this example we show how to re-use data on a specific server without resending them. Client wants to compute $C = C \times A^n$ using the service "*" on server *karadoc*.

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_data_init(&dhA, {"memory://britannia.ens-lyon.fr/&A", NULL}, {"memory://karadoc.aist.go.jp", NULL}, GRPC_DOUBLE, GRPC_STICKY,
                GRPC_RESULT_DEFAULT);
grpc_data_init(&dhC, {"NFS://britannia.ens-lyon.fr/home/user/C.in", NULL}, {"memory://karadoc.aist.go.jp", NULL}, GRPC_DOUBLE, GRPC_STICKY,
                GRPC_NO_RESULT);

for(i=0;i<n+1;i++)
{
   if( i==1 )
      grpc_data_init(&dhC, {"memory://karadoc.aist.go.jp", NULL}, NULL, DOUBLE, GRPC_STICKY, GRPC_NO_RESULT);
   if( i==n )
      grpc_data_init(&dhC, {"memory://karadoc.aist.go.jp", NULL}, {"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL}, GRPC_DOUBLE,
                     GRPC_VOLATILE, GRPC_NO_RESULT);

   grpc_call(handle1,dhA,dhC,&dhC);
}
grpc_data_free(&dhA);
grpc_data_free(&dhC);
```

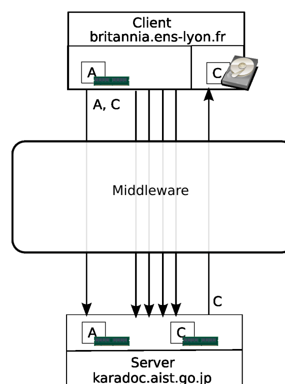

FIG. A.4 – GridRPC call with data management using persistence through the GRPC_STICKY mode.

In this example (see Figure A.4), we show how to use the GridRPC data management functions when the data needs to be stored inside the platform. In this example we consider the client needs to keep the data on the same server. The GRPC_STICKY mode provides this behavior.

Input data

Data A will be used and will remain on server *karadoc*, we can use the GRPC_STICKY parameter to keep the data on server *karadoc*. Data C is an input/output data. The first grpc_data_init for this data requires only an input location and the GRPC_STICKY mode.

Output data

Output data `C` is generated on server `karadoc` but only the last result is useful for the client. Thus, to send the final result to the client we update the output location just before the last `grpc_call()`.

Example with prefetching

In this example we show how the user can deal with the GridRPC to manage the data and thus performs the prefetching of data. Client wants to compute $C = A \times B$ on server *karadoc* and $C = A + C$ on server *perceval*.

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_function_handle_init(handle2,"perceval.rush.aero.org","+");

/* Data initialization */
grpc_data_init(&dhA, {"IBP://kaamelott.cs.utk.edu/1212#A.dat/ReadKey/READ", NULL}, {"NFS://britannia.ens-lyon.fr/home/user/A.dat", NULL}, GRPC_I
        GRPC_VOLATILE, GRPC_RESULT_DEFAULT);

grpc_data_init(&dhB, {"SRB://carmelide.ens-lyon.fr/COLLECTION/Simulations/B.dat", NULL}, {"IBP://kaamelott.cs.utk.edu/1213#B.dat/WriteKey/WRITE"
        GRPC_DOUBLE, GRPC_VOLATILE, GRPC_RESULT_DEFAULT);

grpc_data_init(&dhC, NULL, {"NFS://perceval.rush.aero.org/home/user/C.out", NULL}, GRPC_DOUBLE, GRPC_PERSISTENT, GRPC_NO_RESULT);
grpc_data_init(&dhD, {"NFS://perceval.rush.aero.org/home/user/C.out", NULL}, {"NFS://britannia/home/user/C.out", NULL}, GRPC_DOUBLE, GRPC_PERSIS
        GRPC_NO_RESULT);

/* Write the data using dhA handle on a NFS server. */
grpc_data_write(&dhA, GRPC_BROADCAST, {"NFS://perceval.rush.aero.org/home/user/A2.dat", NULL}, {GRPC_STICKY, NULL});

grpc_call(handle1,dhA,dhB,&dhC);

/* The data transfer of A has been asked previously */
grpc_call(handle2,dhA,dhC,&dhC);

/* Waiting for the end of the C data transfer proceeded after C computation. */
grpc_wait({dhC, NULL}, GRPC_WAIT_ALL);

/* The data is written on its output destination (NFS://britannia/home/user/C.out) */
grpc_data_read(&dhD);
```

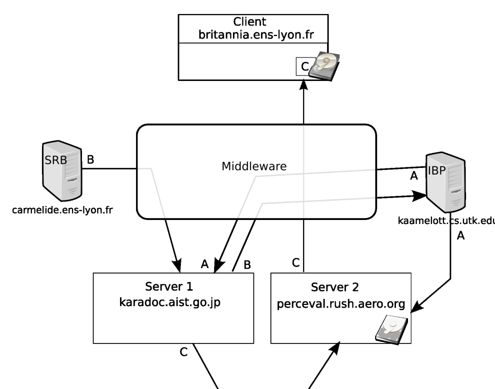

FIG. A.5 – GridRPC call with data prefetching using the API.

In this example (see Figure A.5), we show how to use the GridRPC data management functions to prefetch the data.

Input data

Data `A` is stored on the *Kaamelot IBP* server and will be used on *karadoc* to compute the first operation. This data is also used as an entry to the second operation. Data `B` is located on the *SRB* server and will be used as the second entry of the first operation only. The data prefetching is done by transferring `A` from the *IBP* server to the *perceval* server in parallel of the first computation. The second operation will use the output data `C` as input parameter with `A` that could be prefetched when the computation starts.

Output data

Data `C` is used as output for the two operations. It is first generated on *karadoc* and updated after the call on *perceval*. It is not directly sent back to the client. To obtain it, the client proceed to an explicit data transfer using the `grpc_data_read()` function.

Example with data migration

In this example (see Figure A.6), we show how to use the GridRPC data management functions when the data needs to be stored inside the platform (or on a storage resources, this point depends on the middleware implementation). In this example we consider that the persistence data is kept in memory. Three `grpc_call()` are performed, two on server *karadoc* and one on server *perceval* working on the same data. The goal of the code here is to compute $C = A \times (B + A \times B)$, which is done by doing the steps $C = A \times B$, then $C = B + C$ and finally $C = A \times C$.

```
grpc_function_handle_init(handle1,"karadoc.aist.go.jp","*");
grpc_function_handle_init(handle2,"perceval.rush.aero.org","*");
grpc_function_handle_init(handle3,"karadoc.aist.go.jp","+");
grpc_data_init(&dhA, {"memory://britannia.ens-lyon.fr/&A", NULL}, GRPC_DOUBLE, GRPC_STICKY, GRPC_RESULT_DEFAULT);
grpc_data_init(&dhB, {"NFS://britannia.ens-lyon.fr/home/user/B.dat", NULL}, GRPC_DOUBLE, GRPC_PERSISTENT, GRPC_RESULT_DEFAULT);
grpc_data_init(&dhC, NULL, {"memory://karadoc.aist.go.jp", NULL}, GRPC_DOUBLE, GRPC_PERSISTENT, GRPC_NO_RESULT);
grpc_call(handle1,dhA,dhB,&dhC);
grpc_data_init(&dhC, {"memory://karadoc.aist.go.jp", NULL}, {"memory://perceval.rush.aero.org", NULL}, GRPC_DOUBLE,
               GRPC_PERSISTENT, GRPC_NO_RESULT);
grpc_call(handle2,dhB,dhC,&dhC);
grpc_data_init(&dhC, {"memory://perceval.rush.aero.org", NULL}, {"NFS://britannia.ens-lyon.fr/home/user/C.out", NULL}, GRPC_DOUBLE, GRPC_PERSIST
               GRPC_NO_RESULT);
grpc_call(handle3,dhA,dhC,&dhC);
```

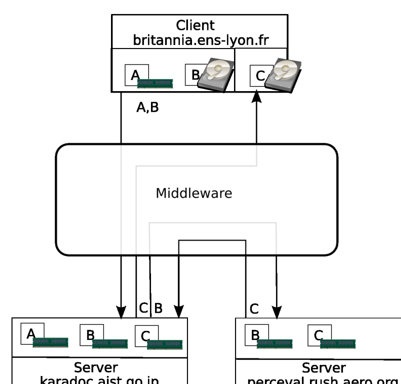

FIG. A.6 – Three RPC call with data management using persistence.

Input data

Data `A` will be used only on server *karadoc*, we can use the GRPC_STICKY parameter to keep the data on server *karadoc* (see Section A). Thus `A` is already available when the third `grpc_call()` is performed. The data `B` is used on two servers. With the GRPC_PERSISTENT mode the second `grpc_call()` implies that the data moves (or is duplicated) from server *karadoc* to server *perceval*.

Output data

Output data `C` is created on server *karadoc*. `C` moves (or is duplicated) from server *karadoc* to server *perceval*. `C` moves (or is duplicated) from server *perceval* to server *karadoc* and at the end, data `C` is sent back to the client.

Author contact information

Yves Caniou
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL
Yves.Caniou@ens-lyon.fr

Eddy Caron
University of Lyon / CNRS / ENS Lyon / INRIA / UCBL
Eddy.Caron@ens-lyon.fr

Hidemoto Nakada
National Institute of Advanced Industrial Science and Technology
hide-nakada@aist.go.jp

Intellectual property statement

The OGF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the OGF Secretariat. The OGF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this recommendation. Please address the information to the OGF Executive Director.

Full copyright notice

Copyright (C) Open Grid Forum (2007). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may

be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the OGF or other organizations, except as needed for the purpose of developing Grid Recommendations in which case the procedures for copyrights defined in the OGF Document process must be followed, or as required to translate it into languages other than English. The limited permissions granted above are perpetual and will not be revoked by the OGF or its successors or assigns. This document and the information contained herein is provided on an "AS IS" basis and THE GLOBAL GRID FORUM DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE."

Bibliographie

[1] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee and H. Casanova. : A GridRPC model and API for End-Users Applications, Global Grid Forum, July 21, 2005, GFD-R.052

[2] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany and R. Wolski : The Internet Backplane Protocol : Storage in the network, Storage in the network. In NetStore '99 : Network Storage Symposium. Internet2, October 1999.

[3] C. Baru, R. Moore, A. Rajasekar and M. Wan : The SDSC Storage Resource Broker, In Procs. of CASCON'98, Toronto, Canada, 1998

[4] D.C. Arnold, D. Bachmann and J. Dongarra : Request Sequencing : Optimizing Communication for the Grid, Lecture Notes in Computer Science 2003, vol 1900, pp 1213

[5] B. Del Fabbro, D. Laiymani, J.-M. Nicod, and L. Philippe : A Data Persistency Approach for the DIET Metacomputing Environment, Int. Conf. on Internet Computing, IC'04, 2004

[6] M. Beck, D.C. Arnold, A. Bassi, F. Berman, H. Casanova, J. Dongarra, T. Moore, G. Obertelli, J. Plank, M. Swany,S. Vadhiyar and R. Wolski : Middleware for the Use of Storage in Communication, IN Parallel Computing, vol 28, number 12, pp 1773-1788, 2002

[7] Y. Aida, Y. Nakajima, M. Sato, T. Sakurai, D. Takahashi and T. Boku : Performance Improvement by Data Management Layer in a Grid RPC System, IN the First International Conference on Grid and Pervasive Computing (GPC2006), pp.324-335, Taiwan, May 3-5, 2006

[8] G. Antoniu, L. Bougé and M. Jan. : JuxMem : An Adaptive Supportive Platform for Data Sharing on the Grid, IN Scalable Computing : Practice and Experience, Vol. 6(3) :45-55, September 2005