CoRDAGe : Un service générique de co-déploiement et redéploiement d'applications sur grilles

Loïc Cudennec

Numéro d'ordre : 3869

THÈSE

présentée devant

L'UNIVERSITÉ DE RENNES 1

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention INFORMATIQUE

PAR

Monsieur Loïc Cudennec

Équipe d'accueil : équipe PARIS, IRISA, INRIA Rennes

École doctorale MATISSE

Composante universitaire : IFSIC

CoRDAGe : un service générique de co-déploiement et de redéploiement d'applications sur grilles.

soutenue le 15 janvier 2009 devant la commission d'Examen

Composition du jury

M. Gabriel ANTONIU, Chargé de Recherche, INRIA (Rennes)	Directeur de Thèse
M. Luc BOUGÉ, Professeur, ENS Cachan/Antenne de Bretagne	Directeur de Thèse
M. Yves DENNEULIN, Professeur, INPG (Grenoble)	Rapporteur
M. Frédéric DESPREZ, Directeur de Recherche HDR, INRIA (Lyon)	Rapporteur
M. Stéphane GANÇARSKI, Maître de Conférence HDR, Université Paris 6	Examinateur
M. Emmanuel JEANNOT, Chargé de Recherche HDR, INRIA (Nancy)	Examinateur
M. Thierry PRIOL, Directeur de Recherche HDR, INRIA (Rennes)	Examinateur

Remerciements

Une *thèse de doctorat*. Il y a une dizaine d'années, j'étais bien loin d'imaginer pouvoir prétendre un jour au titre de docteur. Dès lors, si il y a un remerciement avec lequel je désire commencer, c'est bien celui adressé au système universitaire français. La *fac* m'a accueilli, à la sortie du baccalauréat, sans jugement sur mon classement, ni refus lié à un *numerus clausus*.

Je remercie les sept membres de mon jury de thèse, Gabriel Antoniu, Luc Bougé, Yves Denneulin, Frédéric Desprez, Stéphane Gançarski, Emmanuel Jeannot et Thierry Priol, d'avoir accepté d'évaluer la qualité de mes travaux et de sanctionner ces trois années de formation au métier de chercheur. Les remarques sur le manuscrit ainsi que la qualité de leurs questions lors de la soutenance m'ont été très profitables. Un grand merci à Thierry de m'avoir accueilli au sein de l'équipe PARIS.

Je remercie Luc Bougé de m'avoir proposé, à l'issu de mon projet de fin d'études INSA, de continuer l'aventure dans l'équipe en qualité de doctorant. Merci Luc de m'avoir accordé votre confiance et de m'avoir appris a respecter une certaine rigueur dans le travail. Merci aussi Gabriel pour l'encadrement exceptionnel que tu m'as offert, pour ton professionnalisme et ta disponibilité. Plus que cela, j'ai su apprécier les nombreuses missions que nous avons partagées, de Roscoff à Tsukuba ! Ces missions n'auraient pas été aussi enrichissantes sans les personnes avec lesquelles j'ai pu travailler. Merci à l'équipe JXTA de m'avoir accueilli plusieurs mois à Santa Clara, et tout particulièrement à Mohamed Abdelaziz, Mike Duigou, Henry Jen et Bernard Traversat pour leur gentillesse. Merci aussi à Osamu Tatebe pour sa sympathie et sa disponibilité à faire découvrir la culture japonaise. Si ces missions se sont bien déroulées c'est aussi grâce à Maryse et au personnel du service mission.

Les travaux de thèse ont été largement inspirés et facilités par les nombreuses discussions entre collègues. Je remercie notamment Landry Breuil, David Margery, Pascal Morillon, Guillaume Mornet, Yvon Jégou et Christian Pérez pour leur aide significative tout au long de ces trois années ! Au delà de l'aspect professionnel, je tiens à remercier tous ceux qui ont contribué à ce que travailler dans l'équipe PARIS soit synonyme de convivialité : Adrien, Boris, Hinde, Julien, Matthieu, Oscar, Raul, Thomas et les ex-PARIS : Aline, Étienne, Marin et Yann... j'en oublie certainement, désolé de ne pas pouvoir tous vous citer ici. J'offre tous mes encouragements à ceux qui suivent, notamment à Alexandra, Bogdan et Diana.

Si j'ai choisi de poursuivre en thèse dans cette équipe, c'est en grande partie grâce à Mathieu Jan et Sébastien Monnet. Je vous remercie tous les deux, aussi bien pour les travaux que nous avons menés ensemble dans le *E207 vert*, que pour les moments partagés en mission et à coté. Très bonne continuation à vous deux, ainsi qu'à Caroline et Hélène.

J'ai aussi apprécié retrouver des insaliens à l'IRISA. Merci à mes camarades de promo et compagnons de thèse : Aurélie, Fabien, Peggy, Sébastien, Stéphanie et Xavier. C'est toujours un plaisir de vous croiser à la cafétéria ou dans un film. Je conserve aussi une pensée particulière pour Marie qui nous a quitté trop tôt.

J'aimerais citer mes amis proches et de longue date, qui ont dû se faire à l'idée que j'étais un éternel étudiant. Merci Antoine, Denis, Esther, Gwendal, Mathieu, Matthieu, Nathalie, Sébastien, Vongsone et Sushi. Enfin, j'aimerais remercier ma famille, mon frère Erwann et mes parents Yolande et Yannick qui m'ont encouragé à poursuivre aussi loin les études, même dans le *binaire*, avec tout le soutien dont on peut rêver.


CoRDAGe

Table des matières

Liste des définitions

Chapitre 1

Introduction

L'EXAMEN raisonné et méthodique du monde et de ses *nécessités* a largement profité du développement de l'*informatique*. Cette *science formelle*, dédiée à l'automatisation du traitement de l'information, livre des outils, toujours plus puissants, particulièrement adaptés à la validation et au raffinement de modèles scientifiques abstraits. Par exemple, la simulation informatique permet d'étudier des phénomènes physiques complexes comme la propagation d'ondes radio en milieu urbain ou la résistance d'une plate-forme pétrolière soumise à l'influence de la houle. Un autre exemple est l'automatisation du traitement de grandes masses de données en vue de leur analyse. C'est notamment le cas pour le projet LHC (*grand collisionneur de hadrons*), mis en opération le 10 septembre 2008 au CERN. Lors des expérimentations effectuées sur cet accélérateur de particules, les capteurs des six détecteurs pourront générer jusqu'à 15 péta-octets de données par an. Dans ce contexte, la qualité des résultats dépend en grande partie de la puissance de traitement des ressources informatiques : elles conditionnent le nombre d'itérations effectuées en un temps donné pour affiner le résultat de l'expérience.

Face à ces besoins non bornés en puissance de calcul et de stockage, les *grilles informatiques* semblent offrir une réponse des plus adaptées. Le principe est de mutualiser les ressources informatiques de plusieurs universités, instituts ou entreprises, pour profiter de l'agrégation des puissances de calcul et des espaces de stockage. Ces infrastructures souffrent cependant d'un problème majeur : leur utilisation s'avère être des plus compliquée, même pour les spécialistes du domaine. Ceci est dû en partie à l'échelle des grilles, composées de plusieurs milliers de ressources hétérogènes et réparties au sein de sites géographiquement distants. Ces sites font partie de multiples organisations, empêchant toute gestion des ressources à l'échelle globale. Déployer des applications distribuées sur de telles architectures requiert beaucoup d'efforts de la part des utilisateurs pour découvrir et sélectionner les ressources, installer et configurer les programmes, transférer les données, lancer les calculs, surveiller l'exécution puis, enfin, récupérer les résultats.

Aujourd'hui, des solutions existent pour automatiser la phase de déploiement d'une ap-

plication. Cependant, la plupart de ces solutions ne considèrent le problème que d'une manière *statique* : une fois l'application déployée, plus aucun support pour accompagner son exécution n'est assuré. Or, les grilles de calcul sont caractérisées par un comportement *dynamique* : de nouvelles ressources peuvent joindre l'infrastructure et d'autres peuvent disparaître, de manière volontaire ou suite à une panne. Afin d'utiliser au mieux les grilles ou, tout simplement, ne pas perdre le résultat d'un calcul à la moindre panne d'une ressource, les applications doivent pouvoir profiter d'un support pour prendre en compte la dynamicité de l'infrastructure.

Objectifs de la thèse

L'objectif de cette thèse est d'étudier les mécanismes liés au déploiement d'applications distribuées sur des infrastructures dynamiques de type grilles de calcul. Ces mécanismes doivent s'appuyer sur des systèmes existants et simplifier l'utilisation des grilles. Deux problématiques liées au déploiement ont été traitées.

Le redéploiement concerne l'évolution de la topologie de l'application en cours d'exécution. Celle-ci doit évoluer en fonction des besoins de l'application et de l'état des ressources. Nous ne considérons, dans ces travaux, que les problèmes d'adaptation en terme de topologie de déploiement.

Le co-déploiement provient du besoin de déployer, de manière coordonnée, plusieurs applications de types différents. Ceci est par exemple motivé dans le cadre d'une collaboration entre différentes équipes de recherche, chacune apportant une brique au prototype final.

Ces problématiques de redéploiement et de co-déploiement sont abordées avec comme objectif de respecter les trois propriétés de transparence, de spécificité et de non-intrusivité.

Transparence. Les interactions avec les outils et services de gestion des ressources de la grille doivent être rendues transparentes pour les applications, et encore plus pour l'utilisateur.

Spécificité. Les applications doivent être prises en charge de manière spécifique, c'est-à-dire que le support à la dynamicité doit s'effectuer en proposant des actions adaptées aux besoins de chaque type d'application.

Non-intrusivité. Le support pour la dynamicité ne doit pas entraîner de contraintes fortes sur le modèle de programmation de l'application. Il doit pouvoir être mis en place, indépendamment de la conception initiale.

Dans ce contexte, cette thèse s'intéresse plus particulièrement à offrir un modèle pour prendre en compte la dynamicité de l'infrastructure lors du déploiement d'application. La section suivante détaille nos contributions.

Contributions et publications

Contributions sur le déploiement dynamique

Les trois contributions suivantes concernent le déploiement dynamique. Elles visent à offrir une nouvelle approche pour intégrer l'aspect dynamique de l'infrastructure et prendre en considération les besoins de l'application.

Un modèle pour le déploiement dynamique d'applications.
Le modèle proposé dans ce manuscrit répond aux deux propriétés énoncées ci-dessus : la transparence et la spécificité. Ce modèle a deux fonctionnalités principales : une première fonctionnalité est la traduction d'actions de haut niveau, spécifiques à l'application, en opérations génériques de bas niveau, portant sur les outils de réservation et de déploiement. La deuxième fonctionnalité est la pré-planification du déploiement, en sélectionnant un ensemble de ressources physiques pertinent pour supporter l'exécution de l'application. Pour cela, le modèle est décomposé en cinq phases : elles permettent de décrire, de configurer et de construire une représentation logique de l'application, de construire une représentation logique des ressources physiques, puis de projeter la représentation de l'application sur celle des ressources [CAB08]. Ce modèle permet le co-déploiement et le redéploiement d'applications, de manière générique, grâce à un ensemble d'opérations sur les représentations logiques.

CoRDAGe : proposition d'architecture.
La validation du modèle pour le déploiement dynamique est effectuée grâce à une architecture respectant autant que possible la propriété de non-intrusivité énoncée ci-dessus. Cette architecture est basée sur le modèle client-serveur. Une instance du serveur CoRDAGe est lancée pour s'interfacer avec un ensemble d'applications clientes. Le serveur offre des procédures pouvant être appelées à distance. Les procédures prennent en paramètre une action de haut niveau. Ces actions donnent lieu à des modifications de la représentation logique de l'application ainsi qu'à des appels aux outils de réservation et de déploiement. L'implémentation a permis de démontrer la validité de l'approche dans le contexte de la grille expérimentale Grid'5000 et des outils Oar et ADAGE.

Support pour le déploiement d'applications distribuées.
Le concept de déploiement dynamique a été appliqué à plusieurs types d'applications distribuées : la plate-forme pair-à-pair JXTA [119], le service de partage de données JuxMem [10] et le système de fichiers distribué Gfarm [117]. Ce travail a été effectué en deux étapes : la première étape vise à prendre en charge la partie statique du déploiement. Ceci a nécessité le développement, pour chaque type d'application, d'un greffon spécifique pour l'outil de déploiement ADAGE. La deuxième étape consiste à prendre en charge la partie dynamique du déploiement, en ajoutant le support de ces applications dans CoRDAGe. La notion de co-déploiement est mise en pratique grâce au déploiement coordonné de JuxMem et Gfarm [ACGT08]. Enfin, CoRDAGe a été retenu comme une brique ayant la charge du déploiement dynamique dans le projet ANR LEGO [149].

Contributions annexes

Les quatre contributions suivantes ont été effectuées dans le cadre de cette thèse mais ne figurent pas dans ce manuscrit. Ce choix a été fait pour des raisons de cohérence thématique et d'espace de rédaction. Elles s'inscrivent cependant dans la problématique générale de l'utilisation des grilles informatiques et se révèlent être d'excellents cas d'études pour motiver le déploiement dynamique d'applications distribuées. Le lecteur intéressé par ces contributions peut se référer aux publications citées dans cette section.

Un service de partage de données hiérarchique.
Les besoins grandissants en terme d'espace de stockage requis par les applications scientifiques (en physique des particules, cosmologie, génétique, etc.) motivent la recherche de solutions adaptées pour la gestion des données à grande échelle. Il devient primordial d'offrir un accès efficace et fiable à de grandes quantités de données partagées. Contrairement à l'approche actuelle basée sur le transfert explicite des données, l'utilisation d'un modèle d'accès transparent aux données permet de simplifier le modèle de programmation. Une telle approche est illustrée à travers les notions de système de fichiers distribué et de service de partage de données à l'échelle de la grille. Notre travail [ACGT08, C08], mené en collaboration avec l'AIST et l'université de Tsukuba au Japon, propose un système de stockage hiérarchique pour grille. Ce système tire parti de la rapidité des accès en mémoire physique et de la persistance d'un stockage sur disque. Cette proposition a été validée par un prototype couplant le système de fichiers GFARM avec le service de partage de données JUXMEM. Le prototype a été évalué sur la plate-forme GRID'5000.

Une base de données distribuée sur un service de partage de données
Les systèmes de gestion de bases de données (*Database Management Systems*, DBMS en anglais) sont conçus pour permettre aux applications de stocker des données *structurées*. Ils masquent toutes les considérations liées à la représentation des données en mémoire ainsi que le choix des ressources physiques. Dans notre travail [AABCG07], nous proposons de construire un DBMS au dessus d'un service de partage de données. Le service de partage de données offre en effet des accès transparents, persistants et tolérants aux défaillances à des données stockées dans un environnement dynamique. Les accès portent cependant sur des données *non-structurées* en mémoire physique. Le DBMS apporte donc une interface de plus haut niveau, sans se soucier du stockage physique des données. Une étude de faisabilité est effectuée sur la grille expérimentale GRID'5000, en utilisant le service de partage de données JUXMEM.

Caractérisation du comportement d'un système pair-à-pair.
Les propriétés du modèle pair-à-pair (P2P), comme le passage à l'échelle et la tolérance à la volatilité, ont largement motivé son utilisation dans les systèmes distribués. Plusieurs bibliothèques P2P génériques ont été proposées pour construire des applications distribuées. Cependant, très peu d'évaluations expérimentales de ces infrastructures ont été conduites à grande échelle. Ce type d'expérimentation est pourtant important pour aider les concepteurs de systèmes P2P à optimiser leurs protocoles et ainsi mieux appréhender les bénéfices du modèle P2P. Ceci est particulièrement vrai dans le cas d'utilisation du modèle P2P dans le contexte du calcul sur grille. Nos travaux [ACDJ07, ACDJ06] ont été menés en étroite collaboration avec l'équipe JXTA de Sun Microsystems, à Santa Clara, Californie. Ils s'intéressent à l'aptitude du pas-

sage à l'échelle de deux protocoles proposés par la plate-forme P2P JXTA. Dans un premier temps, nous effectuons une description détaillée des mécanismes utilisés par JXTA pour constituer son réseau logique et y propager des messages : le protocole de rendez-vous. Dans un second temps, nous décrivons le protocole de découverte des ressources. Enfin, ces protocoles font l'objet d'une évaluation expérimentale détaillée à grande échelle utilisant les neuf sites de la grille expérimentale GRID'5000.

Un protocole hiérarchique pour la cohérence des données.
Ce travail porte sur le problème de la visualisation des données partagées dans les applications à base de couplage de codes sur les grilles de calcul. Nous proposons d'améliorer l'efficacité de la visualisation en intervenant sur les mécanismes de gestion des données répliquées et plus particulièrement au niveau du protocole de cohérence. La notion de lecture relâchée [CM05, ACM06c] est alors introduite comme une extension du modèle de cohérence à l'entrée (*entry consistency*). Ce nouveau type d'opération peut être réalisé sans prise de verrou, en parallèle avec des écritures. En revanche, l'utilisateur relâche les contraintes sur la fraîcheur de la donnée et accepte de lire des versions légèrement anciennes, dont le retard est néanmoins contrôlé. L'implémentation de cette approche au sein du service de partage de données pour grilles JUXMEM montre des gains considérables par rapport à une implémentation classique basée sur des lectures avec prise de verrou. Des expérimentations multi-site [ACM06b, ACM06] ont été menées sur la plate-forme expérimentale GRID'5000 alors en cours de construction.

Publications

Conférences internationales avec comité de lecture

[ACGT08] Gabriel Antoniu, Loïc Cudennec, Majd Ghareeb, and Osamu Tatebe. **Building hierarchical grid storage using the gfarm global file system and the juxmem grid data-sharing service.** In *14th International Euro-Par Conference Euro-Par 2008 Parallel Processing*, Lecture Notes in Computer Science, volume 5168, pages 456–465, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[CAB08] Loïc Cudennec, Gabriel Antoniu and Luc Bougé. **CoRDAGe: towards transparent management of interactions between applications and ressources.** In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008), held in conjunction with the International Conference on Supercomputing (ICS 2008)*, pages 13–24, Kos, Greece, June 2008.

[AABCG07] Abdullah Almousa Almaksour, Gabriel Antoniu, Luc Bougé, Loïc Cudennec and Stéphane Gançarski. **Building a DBMS on top of the JuxMem Grid Data-Sharing Service.** In Proceedings of the HiPerGRID Workshop, held in conjunction with Parallel Architectures and Compilation Techniques 2007 (PACT 2007). Brasov, Romania, September 2007.

[ACDJ07] Gabriel Antoniu, Loïc Cudennec, Mike Duigou, and Mathieu Jan. **Performance scalability of the JXTA P2P framework.** In *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2007)*, page 108, Long Beach, CA, USA, 2007.

[ACM06] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. **Extending the entry consistency model to enable efficient visualization for code-coupling grid applications.** In *6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, pages 552–555, Singapore, May 2006.

[ACM06b] Gabriel Antoniu, Loïc Cudennec, and Sébastien Monnet. **A practical evaluation of a data consistency protocol for efficient visualization in grid applications.** In *International Workshop on High-Performance Data Management in Grid Environment (HPDGrid 2006), held in conjunction with VECPAR'06*, Rio de Janeiro, Brazil, July 2006. Selected for publication in the post-conference book.

Conférences francophones avec comité de lecture

[C08] Loïc Cudennec. **Un service hiérarchique distribué de partage de données pour grille.** In *Rencontres francophones du Parallélisme (RenPar '18)*, Fribourg, Suisse, Février 2008.

[CM05] Loïc Cudennec and Sébastien Monnet. **Extension du modèle de cohérence à l'entrée pour la visualisation dans les applications de couplage de code sur grilles.** In *Actes des Journées francophones sur la cohérence des données en univers réparti (CDUR 2005)*, Paris, Novembre 2005.

Présentations invité et rapports de recherche

[C08b] Loïc Cudennec. **CoRDAGe : un outil de co-déploiement et de re-déploiement d'applications sur grilles.** In *Action d'animation sur la reconfiguration dynamique des logiciels (ADAPT)*, présentation invitée, Fribourg, Suisse, Février 2008.

[C07] Loïc Cudennec. **JuxMem: A Data-Sharing Service for the Grid.** In *ACM SIGOPS, Journées Thèmes Emergents - HPC File Systems: From Cluster to Grid*, invited talk, IRISA, Rennes, France, October 2007.

[C07b] Loïc Cudennec. **The JuxMem-Gfarm collaboration.** Invited talk, Dipartimento di Elettronica Informatica e Sistemistica, Università della Calabria, Rende, Italy, July 2007.

[ACDJ06] Gabriel Antoniu, Loïc Cudennec, Mike Duigou, and Mathieu Jan. **Performance scalability of the JXTA P2P framework.** Rapport de Recherche INRIA RR-6064, IRISA, Rennes, France, Décembre 2006.

[ACM06c] Gabriel Antoniu, Loïc Cudennec and Sébastien Monnet. **Extending the entry consistency model to enable efficient visualization for code-coupling grid applications.** Rapport de Recherche INRIA RR-5813, IRISA, Rennes, France, Janvier 2006.

Organisation du manuscrit

Ce manuscrit est composé de sept chapitres. Les deux premiers chapitres apportent un cadre d'étude pour le déploiement dynamique. Le chapitre 2 présente les applications distribuées, les grilles de calcul ainsi qu'un état de l'art sur la gestion dynamique des applications.

Le chapitre 3 introduit la plate-forme expérimentale GRID'5000 ainsi que les outils de réservations de ressources OAR et de déploiement d'applications ADAGE. Il montre comment, dans ce contexte, les applications pair-à-pair JXTA et le service de partage de données JUXMEM sont déployés, de manière statique. Ce chapitre se conclut en indiquant les difficultés rencontrées par les utilisateurs et en mettant en valeur le besoin pour un outil de déploiement dynamique.

Le chapitre 4 introduit la contribution principale de cette thèse : un modèle pour le codéploiement et le redéploiement d'applications. Le modèle est accompagné d'une proposition d'architecture, appelée CORDAGE, dont l'implémentation est présentée dans le chapitre 5. Les chapitres 6 et 7 proposent, respectivement, la validation de l'architecture et l'évaluation du prototype de recherche. La validation s'effectue en montrant comment ajouter le support CORDAGE aux applications JUXMEM et GFARM. Elle est aussi effectuée à travers l'intégration dans le projet ANR LEGO. Les expérimentations sont effectuées sur les neuf sites de la plate-forme expérimentale GRID'5000, en interaction avec OAR et ADAGE. Enfin, le chapitre 8 correspond à la conclusion et présente les perspectives offertes par nos travaux.

Chapitre 2

Contexte d'étude

Sommaire

Avec la démocratisation du matériel informatique, des accumulations de ressources se sont formées dans les universités, les entreprises et même chez le particulier. Les applications pouvant profiter d'une telle puissance de calcul répartie – et aussi d'un grand espace de stockage – ne manquent pas, à commencer par les applications scientifiques. Cette approche pose cependant de nombreux problèmes de mise en œuvre. Premièrement, les applications doivent être conçues pour profiter de ressources géographiquement distribuées. Deuxièmement, la gestion et l'organisation de ces ressources doivent s'effectuer avec un minimum de synchronisation et de cohérence, ne serait-ce que pour en connaître l'existence et pouvoir les contacter.

Dans ce manuscrit, nous considérons principalement les *applications distribuées* dédiées au *calcul scientifique*. Nous nous plaçons dans le cas où ces applications sont conçues pour être exécutées sur des *grilles de calculateurs*. L'un des principaux défis dans ce contexte est de rendre l'exécution aussi simple que possible, malgré la nature complexe des infrastructures. En effet, pour pouvoir utiliser les ressources d'une grille, de nombreuses étapes très

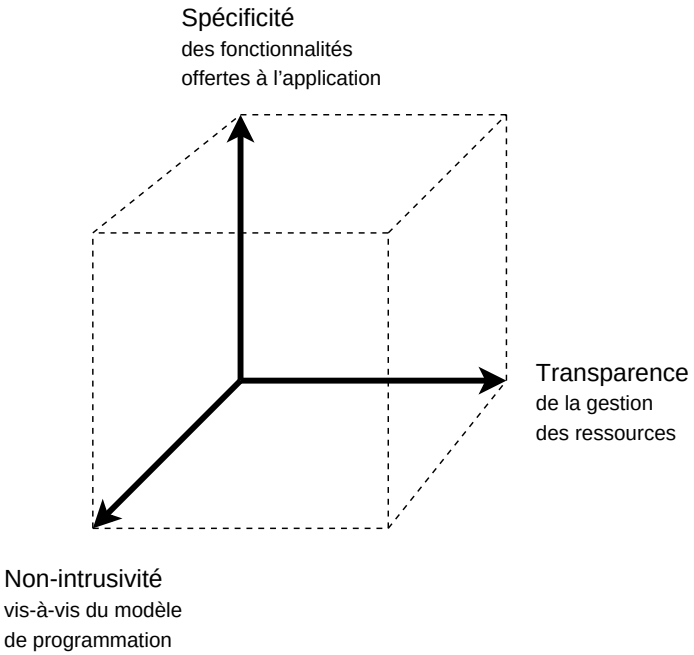

FIG. 2.1 – Les trois propriétés devant être satisfaites pour rendre les applications autonomes : 1) transparence de la gestion des ressources, 2) spécificité des fonctionnalités offertes à l'application et 3) non-intrusivité dans le programme.

techniques sont nécessaires : elles ont trait à la réservation de ressources, au déploiement et à la surveillance des applications ou encore à la sécurité des opérations. La *sélection*, la *réservation des ressources* ainsi que le *déploiement de l'application* constituent des étapes nécessaires à l'utilisation d'une grille. Ce sont ces étapes que nous étudions dans ce manuscrit afin de les rendre les plus transparentes possibles pour l'utilisateur. Elles doivent prendre en compte la *dynamicité* des applications et de l'environnement d'exécution. Ceci se traduit par des actions d'adaptation de l'application pendant son exécution. Des travaux de recherche tentent de donner un cadre conceptuel pour rendre les applications *autonomes* et ainsi limiter l'intervention des utilisateurs.

Cet objectif de rendre les applications autonomes doit satisfaire les trois propriétés suivantes (voir la figure 2.1).

Transparence : masquer les interactions avec les outils de gestion des ressources et les outils de déploiement. Cela est désormais incontournable pour garantir une utilisation simple et efficace d'une grille de calcul. Il doit être possible d'exprimer la dynamicité en terme d'actions de *haut niveau*. Le pilotage des outils de réservation et de déploiement ne doit pas se faire dans leur langage de commande de bas niveau.

Spécificité : proposer des actions spécifiques à l'application. Des besoins particulier en adaptation ne peuvent être pris en compte uniquement grâce à des actions génériques. L'équilibrage de charge et la migration de processus ne sont pas les seules reconfigurations possibles.

Non-intrusivité : laisser le développeur libre du choix du modèle de programmation. Rendre les applications adaptables ne doit pas imposer des choix technologiques. Ceci

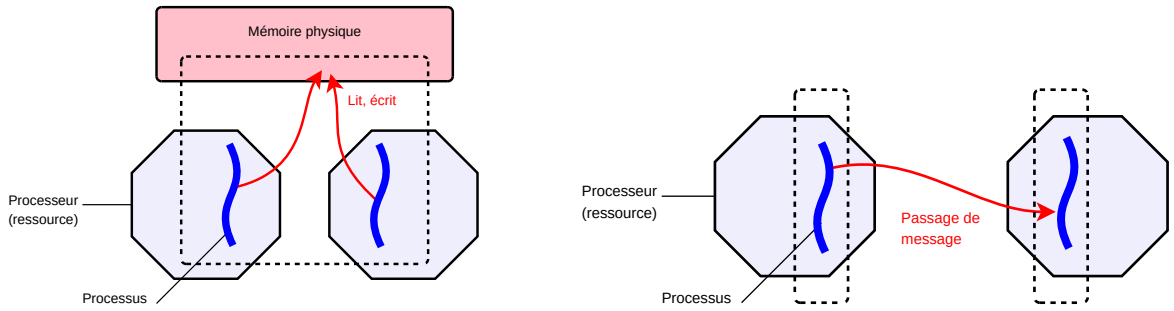

FIG. 2.2 – Deux types de programmation parallèle. (a) Partage de mémoire. (b) Passage de message.

a pour avantage de pouvoir s'appliquer à des applications déjà développées.

Dans ce chapitre, nous présentons, dans un premier temps, les applications distribuées pour le calcul scientifique ainsi que les grilles de calculateurs. Dans un deuxième temps, nous montrons comment les utilisateurs font usage de la grille et comment des solutions d'aide à l'autonomie des applications peuvent émerger à partir d'outils et de services de bas niveau. Nous présentons ces différents systèmes en les positionnant sur les axes de la figure 2.1.

2.1 Des applications pour le calcul scientifique et des grilles de calculateurs

Les applications scientifiques sont réputées pour leurs besoins importants en puissance de calcul et en espace de stockage. Par exemple, la précision des résultats d'une simulation météorologique dépend directement du nombre d'itérations effectuées et de la finesse des mailles du domaine. Ainsi, l'objectif est d'effectuer le maximum d'itérations dans un temps imparti pour obtenir la prévision la plus précise possible. Dans cette section, nous définissons ce qu'est une *application* et plus précisément ce qu'est une *application distribuée*. Nous présentons ensuite les *grilles de calculateurs*, architectures naturelles pour l'exécution d'un certain nombre d'applications scientifiques.

2.1.1 Applications distribuées

Le *programme informatique* est une suite d'opérations compréhensibles par un humain et écrite par un développeur d'applications. Ce programme est compilé en un *exécutable*[1], une suite d'instructions dépendantes de la *ressource informatique* cible (voir la définition 2.6). L'exécutable est lancé sur une ressource avec un ensemble de paramètres. Le résultat est un *processus*. Il est défini par l'ensemble des instructions à exécuter ainsi qu'un espace mémoire pour stocker les données de travail. Plusieurs processus peuvent cohabiter sur une même ressource physique. Nous définissons notre vision d'une *application*.

[1] À l'exception des langages de programmation directement interprétés par une machine virtuelle.

Définition 2.1 : ***application*** — Une application est définie comme un ensemble de programmes qui, une fois instanciés en processus, participent à un but commun, comme par exemple la résolution d'un problème ou la mise en place d'un service.

Lors de l'exécution d'une application, plusieurs processus peuvent interagir entre eux. Ceci est possible selon deux méthodes principales.

1. Les flots d'exécution attachés à un processus se partagent le même espace d'adressage en mémoire physique. Il est donc possible d'écrire et lire, de manière concurrente, les données en mémoire (voir figure 2.2(a)).
2. Les flots d'exécution peuvent s'échanger des données en communiquant de diverses manières, par exemple en s'envoyant des messages ou encore en faisant appel à des procédures distantes (voir figure 2.2(b)).

Définition 2.2 : ***application distribuée*** — Une application distribuée est une application dont les processus communiquent entre eux sans partager le même espace d'adressage en mémoire physique.

2.1.1.1 Généralités

La conception d'une application parallèle ou distribuée est motivée par deux principales raisons.

Performance. Exécuter une application sous forme de plusieurs flots d'exécution concurrents permet de tirer parti de la puissance d'un ensemble de ressources physiques. L'une des approches consiste à adapter les applications pour s'exécuter sur des infrastructures distribuées. Certains problèmes comme le traitement de grandes matrices peuvent être résolus par des applications massivement distribuées : la matrice est découpée en sous-matrices, chacune prise en charge par un processus exécuté sur une ressource physique différente. Le gain de performance n'est cependant pas garanti : il faut s'assurer que le coût des synchronisations ne dégrade pas trop les performances du calcul global.

Disponibilité. Distribuer des processus sur différentes ressources physiques permet de faire face à des défaillances : si l'un des processus est sujet à une défaillance, les autres peuvent prendre le relais, sans risquer la perte du calcul global. Dans le cas d'un service rendu à des utilisateurs, la multiplication des processus pouvant assurer ce service permet de distribuer les requêtes entre plusieurs ressources. Cela permet aussi de choisir la ressource susceptible de répondre au mieux aux attentes de l'utilisateur.

Dans la suite de ce manuscrit, nous nous focalisons sur les applications distribuées dédiées au calcul scientifique.

2.1.1.2 Différents modèles de programmation distribuée

L'une des caractéristiques offertes par la programmation d'applications distribuées est la possibilité de faire communiquer les différents flots d'exécution entre eux. Ceci se traduit, dans les programmes, par l'utilisation de primitives dédiées à la gestion des communications, de la synchronisation des flots d'exécution ou encore du traitement des messages.

Afin de faciliter la conception de telles applications, différents *modèles de communication* ont été proposés et ce, dès la fin des années 70 [71, 39].

Passage de messages. Le modèle du passage de messages est l'un des plus répandus en programmation distribuée. Il consiste à faire communiquer les différents flots d'exécution sur le principe des communications synchrones. Parmi les langages et protocoles de communication permettant de faciliter l'envoi et la réception de messages, nous pouvons citer MPI [54] (*Message Passing Interface*) et PVM [113] (*Parallel Virtual Machine*).

Appel de procédure à distance. Le modèle basé sur l'appel de procédure à distance (*Remote Procedure Call* en anglais) permet d'exécuter des instructions sur une ressource distante. Le nom de la procédure à exécuter ainsi que les paramètres associés sont envoyés à un serveur, qui effectue le traitement et retourne le résultat au client. Les canevas JavaRMI [128] et GridRPC [111] aident au développement d'applications basées sur l'appel de procédure à distance.

Composants distribués. Les modèles de composants permettent de représenter les programmes sous forme d'un assemblage de boîtes noires. Le développeur ne peut accéder qu'aux entrées et sorties de ces boîtes. Ces modèles se prêtent bien à la conception d'applications distribuées : le principe est de lancer l'exécution de différentes instances des composants sur des ressources géographiquement distribuées. Parmi ces modèles nous pouvons citer CORBA [101] et Fractal [24].

Ces différents modèles de communication peuvent être mis en œuvre au sein de deux principales architectures réseau.

Client-serveur. Cette architecture distingue deux rôles : le serveur est un processus passif qui écoute et attend les requêtes des clients. Ces requêtes sont traitées et le résultat est produit en retour. Un serveur peut souvent traiter plusieurs requêtes de clients différents en même temps. Le client est un processus actif, à l'initiative de toute communication. Un exemple d'architecture client-serveur est le terminal X [110], dans lequel le terminal utilisateur envoie à un serveur des commandes pour exécuter ses applications à distance. Les programmes et les données sont stockées sur le serveur. Ce modèle a pour avantage de centraliser les données sur le serveur et donc d'en faciliter la gestion. La centralisation en fait aussi son plus gros inconvénient : lorsque le nombre de clients devient trop important, le serveur ne peut traiter les requêtes dans un temps raisonnable, ce qui se traduit par des délais supplémentaires pour les utilisateurs. Si le serveur tombe en panne, l'architecture complète est paralysée.

Pair-à-pair. L'architecture pair-à-pair (*Peer-to-Peer* – P2P – en anglais) s'oppose au modèle client-serveur dans le sens où chaque processus est client *et* serveur à la fois. L'intérêt est ici de pallier les défaillances de processus en s'appuyant sur les autres instances. Cette architecture a démontré sa capacité à passer à l'échelle. Aujourd'hui, de nombreux systèmes pair-à-pair sont déployés sur Internet et permettent à des millions d'internautes de s'échanger des fichiers. Parmi les systèmes pair-à-pair nous pouvons citer Kazaa [147] et eDonkey [136].

Le choix du modèle de communication et de l'architecture réseau font partie des points clés de la conception d'une application distribuée.

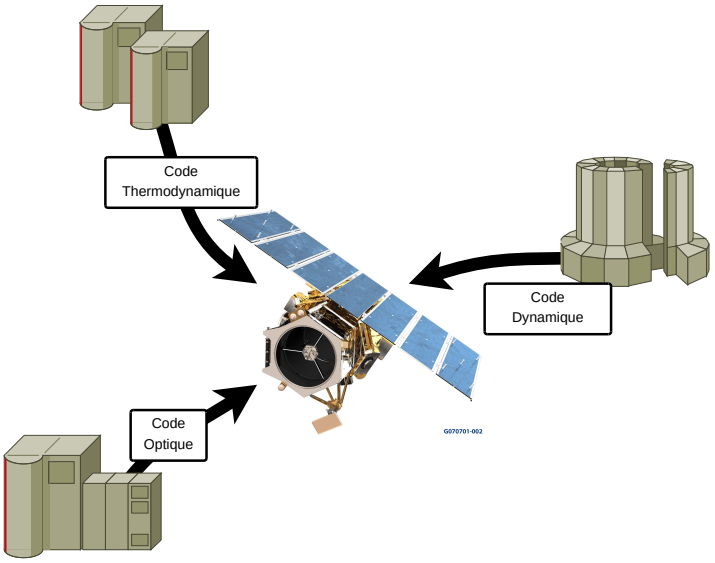

FIG. 2.3 – Couplage de codes lors de la conception d'un satellite.

2.1.1.3 Le couplage de codes

L'une des classes d'applications distribuées utilisée par le calcul scientifique est basée sur le couplage de codes. Dans cette approche, le concepteur de l'application décompose le problème en un ensemble de sous-problèmes, travaillant sur des données disjointes. Ces sous-problèmes ne nécessitent des échanges de données que de manière épisodique. Un exemple est donné en figure 2.3. L'application en charge de la conception collaborative d'un satellite est distribuée sur trois ensembles de ressources physiques. Cette distribution est effectuée de manière thématique : différents sites supportent les calculs liés à la thermodynamique, à l'optique ou encore à la dynamique. Plusieurs exemples d'applications basées sur le couplage de codes sont donnés dans [19, 143, 124].

OASIS [124, 123] est un exemple de l'utilisation du couplage de codes dans le calcul scientifique. Ce projet est mené au sein de l'équipe *Climate Modelling and Global Change* du CERFACS. L'objectif est de proposer un couplage des modèles représentant les différentes composantes du système climatique. OASIS a été appliqué à la modélisation des échanges entre l'océan et l'atmosphère. Le choix du couplage de codes est motivé par 1) le regroupement au sein d'une application complexe des applications déjà existantes, sans en modifier le code et 2) des contraintes géographiques d'exécution de certains codes qui sont dépendants d'un environnement spécifique. OASIS est utilisé comme cas d'étude dans la section 6.3.

2.1.1.4 Le déploiement d'applications

Le cycle de vie d'une application comprend plusieurs étapes, de l'étude d'un besoin jusqu'à la terminaison de son exécution. La figure 2.4 montre les principales étapes, en développant celles qui concernent directement l'utilisateur. En effet, une fois l'application publiée[2]

[2]La publication consiste en la mise à disposition du code compilé aux utilisateurs.

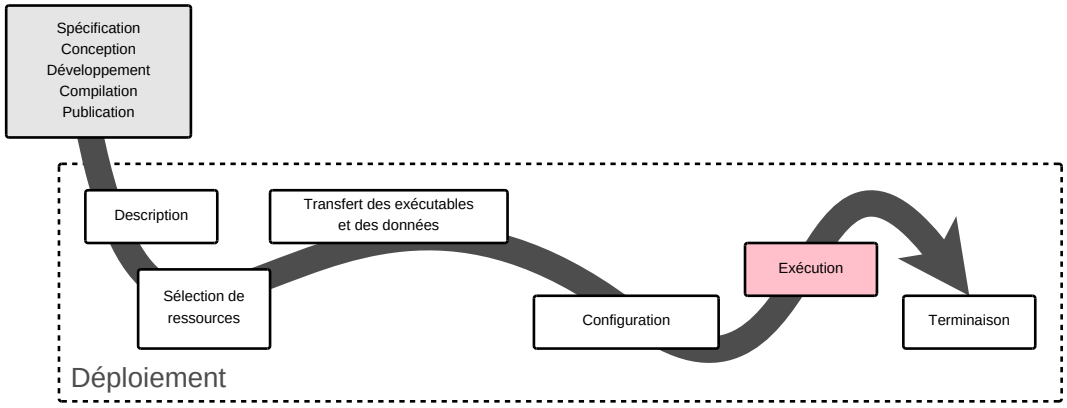

FIG. 2.4 – Le cycle de vie d'une application, de la spécification à la terminaison.

elle est bien souvent *déployée* par l'utilisateur lui-même. Nous définissons la notion de déploiement comme suit.

Définition 2.3 : ***déploiement d'application*** — Le déploiement d'une application est constitué de l'ensemble des étapes situées après la *publication* et permettant le fonctionnement effectif de l'application.

Le déploiement comprend plusieurs phases. Elles sont souvent court-circuitées ou effectuées à la main par l'utilisateur dans le cadre d'applications simples. Pour les applications plus complexes, comme les applications de couplage de codes, ces phases nécessitent des aides logicielles.

- La description permet d'exprimer l'architecture de l'application dans un formalisme spécifique. Cette description contient des informations sur les différents constituants de l'application : leur nombre, la manière dont ils sont assemblés, leurs interconnexions, les paramètres à utiliser lors du lancement.
- La sélection des ressources établit une liste des ressources physiques et logicielles allant supporter l'exécution des différents constituants de l'application. Cette étape peut s'accompagner d'une demande de réservation si les ressources sont sous le contrôle d'un gestionnaire.
- Les exécutables et les données sont transférés sur les ressources physiques sélectionnées auparavant. Ceci est effectué pour chaque constituant de l'application.
- Les différents constituants de l'application sont configurés. Ceci passe notamment par la création de fichiers de configuration, la création de variables d'environnement, le changement de permissions pour l'accès à certains fichiers.
- L'exécution est orchestrée en fonction des contraintes temporelles établies entre les différents constituants. Si l'on prend l'exemple d'une architecture client-serveur, le serveur devra être opérationnel *avant* le lancement du client afin d'être prêt à recevoir les requêtes.
- La terminaison de l'application est définie par la terminaison correcte de tous les processus de l'application.

Cette vision est basée sur une approche *statique*. Une fois l'application lancée, plus aucun contrôle n'est effectué pour s'assurer que les conditions d'exécution sont correctes. Nous

pouvons imaginer divers scénarios dans lesquels l'application est, lors de la phase de description, mal dimensionnée. Par exemple, dans le cas d'une architecture client-serveur, la ressource physique supportant l'exécution du serveur peut ne pas être suffisamment puissante pour prendre en charge tous les clients. Il s'avère alors nécessaire de déployer de nouveaux serveurs permettant d'équilibrer la charge sur un ensemble de ressources. L'adaptation *dynamique* d'une application peut passer par son *redéploiement* qui peut être défini comme suit.

Définition 2.4 : ***redéploiement d'application*** — Le redéploiement est une modification de l'application en cours d'exécution par ajout ou suppression d'éléments qui la constituent.

Une autre problématique liée au déploiement provient des dépendances fonctionnelles qui peuvent apparaître entre plusieurs applications. Cette situation se rencontre notamment lorsqu'une application repose sur une bibliothèque logicielle ou un service extérieur. Nous pouvons imaginer le cas d'une application de couplage de codes qui lit et écrit ses données dans un service distribué de partage de données. Cet exemple pose deux problèmes en terme de déploiement. Le premier problème est de s'assurer que le service de partage de données est correctement déployé et opérationnel *avant* de lancer l'application de couplage de codes. Le deuxième problème est d'effectuer le déploiement de manière coordonnée pour satisfaire les *contraintes de placement* entre l'application et le service. Par exemple, les processus de l'application de couplage de codes qui lisent et écrivent dans le service doivent être placés sur une ressource physique supportant l'exécution d'au moins un des processus du service. Déployer des applications de manière coordonnée fait appel à la notion de *co-déploiement* que nous définissons comme suit.

Définition 2.5 : ***co-déploiement d'application*** — Le co-déploiement consiste à déployer, de manière coordonnée, plusieurs applications dont les développements ont été effectués de manière indépendante.

Le déploiement d'application est un domaine qui prend de l'importance suite à la complexification des applications et des infrastructures d'exécution. Contrairement à des phases comme la spécification et la conception, le déploiement peut largement être automatisé grâce à l'utilisation d'aides logicielles. Nous allons voir dans les sections suivantes qu'il existe des solutions pour traiter l'aspect statique du déploiement. L'objectif de ce manuscrit est de contribuer au support dynamique des applications en terme de déploiement.

2.1.2 Grilles de calculateurs

La fin du vingtième siècle a vu la généralisation des réseaux longue distance permettant d'interconnecter des centres de calculs appartenant à différentes institutions. Les performances de ces réseaux sont devenues suffisamment intéressantes d'un point de vue latence et bande passante pour envisager l'exécution d'applications distribuées sur des calculateurs géographiquement éloignés. L'idée de profiter de la puissance de traitement de plusieurs centres de calcul a été expérimentée en 1995 aux États-Unis, à travers le projet I-Way [45, 55] (*Information Wide Area Year*).

2.1.2.1 De la ressource à la grille informatique

L'agrégation de calculateurs n'est pas une idée nouvelle : à partir des années 80, avec l'arrivée massive de l'ordinateur personnel, les universités et entreprises se dotent de grappes de calculateurs (*clusters* en anglais). Les machines, à peine plus puissantes que les ordinateurs vendus aux particuliers, sont interconnectées par des réseaux à haute performance. Elles sont empilées dans une même salle afin de construire des super-calculateurs distribués. Cette approche apporte une solution alternative aux super-calculateurs centralisés, extrêmement coûteux à concevoir et voués à une production limitée à quelques exemplaires dans le monde. Les grappes de calculateurs ont comme particularité d'être homogènes, c'est-à-dire composées de machines identiques. Elles ne regroupent que quelques centaines de machines. Les grappes de calculateurs vont, elles aussi, profiter de l'interconnexion des universités et institutions, donnant naissance aux *fédérations de grappes*.

Le terme de *grille* (*grid* en anglais) apparaît en 1998, dans le livre [58], édité par Foster et Kesselman. Il est choisi d'après l'analogie qui peut être faite entre ce type d'infrastructure de calcul et le réseau de distribution d'électricité (*Power Grid* en anglais). L'idée est ici de rendre l'utilisation des grilles informatiques aussi simple que celle du réseau électrique : les appareils sont branchés à une prise, sans pour autant connaître les modalités de production de l'énergie. Dans le contexte des grilles informatiques, l'utilisateur doit pouvoir soumettre une application sans se soucier de la gestion de son exécution, ni même être conscient des ressources qui sont impliquées dans le calcul.

La définition des grilles informatiques laisse, aujourd'hui encore, une grande place à l'interprétation lorsqu'il s'agit de décider si une plate-forme peut prétendre à cette appellation. Néanmoins, plusieurs publications [59, 57] s'attachent à caractériser les grilles informatiques. Afin de donner notre propre définition, nous introduisons les concepts de *ressources* et de *site d'une grille*. Nous reprenons les définitions proposées dans [81].

Définition 2.6 : ***ressource informatique*** — Une ressource informatique est un élément matériel ou logiciel qui permet de générer, traiter, stocker ou échanger des données informatiques automatiquement.

Définition 2.7 : ***site d'une grille*** — Un site de grille est un ensemble de ressources informatiques localisées géographiquement dans le même institut, campus universitaire, centre de calcul, entreprise ou chez un individu et qui forment un domaine d'administration autonome, uniforme et coordonné.

Ces deux définitions nous permettent de proposer une définition de la grille informatique qui sera utilisée ce manuscrit.

Définition 2.8 : ***grille informatique*** — Une grille informatique est un ensemble de sites de grilles.

Nous pouvons, accessoirement, caractériser les utilisateurs des grilles informatiques à travers la notion d'*organisation virtuelle*.

Définition 2.9 : ***organisation virtuelle*** — Une organisation virtuelle est définie par un ensemble d'acteurs, qu'ils soient représentés par une personne ou une entreprise, ayant un intérêt commun à partager des ressources informatiques. Les organisations virtuelles sont dynamiques : elles se font et se défont en fonction des besoins des acteurs.

Les sites de grilles peuvent alors définir leurs politiques d'accès de manière indépendante pour accueillir ou non des organisations virtuelles. L'utilisation des sites s'effectue sur la base de contrats à durée déterminée passés entre les sites et les organisations virtuelles.

Il existe plusieurs types de grilles. Elles diffèrent selon leur but ou leur organisation. L'une des propriétés est d'être *multi-utilisateur*, c'est-à-dire que plusieurs utilisateurs peuvent y accéder pour utiliser des ressources. Une deuxième propriété est d'être *multi-application*, c'est-à-dire qu'il est possible d'y soumettre des applications de types différents.

Les grilles de données sont spécialisées dans le stockage de grands volumes de données. Elles doivent garantir la persistance et la disponibilité des données tout en offrant des accès rapides. Les grilles de données sont multi-utilisateur mais mono-application. Par exemple, le projet LHC (*Large Hadron Collider*), dirigé par le CERN et mis en service fin 2008 [36], devrait dépasser la production de 15 péta-octets par an lors de ces expériences. Cette avalanche de données est gérée par le projet MONARC [95] en s'appuyant de l'infrastructure LCG [150] (LHC *Computing Grid*).

Les grilles de récupération de cycles sont composées d'ordinateurs dont le rôle n'est pas à l'origine de participer à des calculs scientifiques. Ce sont typiquement des ordinateurs de bureau connectés à Internet, dont les capacités sont largement sous-utilisées. Les grilles de récupération de cycles sont mono-application et mono-utilisateur. Des projets comme BOINC [7] et le très populaire SETI@Home [8] démontrent le succès de l'utilisation de millions de machines, sur la base du volontariat, pour effectuer un calcul global. La nature de cette architecture basée sur le modèle client-serveur se prête bien aux applications de calcul paramétrique.

Les grilles de calcul sont conçues pour supporter des applications demandant une grande puissance de calcul. Pour cela, l'accent est mis sur les performances des ressources de la grille ainsi que sur les réseaux qui les interconnectent. Les grilles de calcul sont, pour la plupart, multi-application et multi-utilisateur.

Dans ce manuscrit, nous étudions les mécanismes liés au lancement d'applications distribuées sur les grilles informatiques. En ce qui concerne le déploiement, les grilles de données ne constituent pas un exemple pertinent : les utilisateurs n'interagissent avec elles que pour accéder à des données. Aucune application supplémentaire ne peut y est déployée. Les grilles de récupération de cycles ne sont pas, non plus, caractéristiques de la problématique du déploiement. Elles sont conçues sur le principe que chaque propriétaire de ressource a effectué l'installation de l'application ou d'un environnement d'exécution avant de participer au calcul global. En revanche, la plupart des grilles de calcul sont non spécialisées et doivent être capables de supporter le déploiement et l'exécution d'un grand nombre d'applications distribuées. C'est pourquoi nous ne considérons par la suite que les grilles informatiques dédiées au calcul scientifique.

2.1.2.2 Quelques exemples représentatifs de grilles de calcul

Il existe de nombreuses grilles de calcul dans le monde. Leurs vocations sont diverses : de la grille de production à la grille expérimentale.

Application domain	Number of active VOs	Number of users
Computer Science and Mathematics	3	21
Multidisciplinary VOs	17	944
Astronomy, Astrophysics and Astro-Particle Physics	19	226
Life Sciences	7	261
Computational Chemistry	5	316
Earth Sciences	4	124
Fusion	2	46
High-Energy Physics	35	7146
Infrastructure	21	2991
Others	24	1888
TOTAL	**137**	**13963**

FIG. 2.5 – Nombre d'organisations virtuelles (VO) et d'utilisateurs, classés par disciplines de recherche, au sein de la grille de production européenne EGEE. Cette table montre que les applications issues de la recherche sur la physique des particules sont les mieux représentées. Ceci s'explique par le besoin non borné en puissance de calcul de telles applications et aussi par l'utilisation historique des calculateurs par les physiciens depuis les tout débuts de l'informatique. *Cette table est issue du site Internet d'EGEE/IN2P3, le 26 septembre 2008.*

La grille de production. Elle est avant tout dédiée au support d'applications scientifiques ou commerciales telles que la simulation, la conception participative ou le support de services pour des sites Internet. Elle est administrée avec pour objectif de rendre un service de calcul fiable et efficace. Nous pouvons citer comme exemples de grilles de production, les grilles norvégienne NorGrid [153], suédoise SweGrid [160] ou encore européennes DEISA [100] et EGEE [137].

La grille expérimentale. Elle a pour but de mettre en valeur les prochains défis de l'informatique distribuée. Elle est conçue suivant les besoins des domaines de recherche qui doivent y être étudiés, avec une tendance pour les architectures hautement reconfigurables, c'est-à-dire qui permettent à l'utilisateur de contrôler l'environnement matériel et logiciel. Nous pouvons citer, à titre d'exemples de grilles expérimentales, la grille américaine TeraGrid [104], la grille européenne EUROGRID [84] ou encore le projet français GRID'5000 [30] que nous présentons plus en détail dans la section 3.1.1.

2.1.2.3 Les nouveaux défis sur l'utilisation des grilles de calcul

Les grilles de calcul représentent une alternative aux super-calculateurs centralisés. Les coûts de mise en réseau d'ordinateurs non spécialisés sont faibles. Ils ont permis à de nombreux scientifiques, issus de disciplines de recherche diverses, d'accéder à une grande puissance de calcul. Ces infrastructures distribuées s'avèrent cependant être beaucoup plus complexes à utiliser qu'un super-calculateur classique, pour plusieurs raisons.

Une architecture distribuée. Le super-calculateur centralisé a comme principal avantage d'être conçu comme une solution tout-en-un. Le manuel qui l'accompagne en spécifie les usages : lancer une application ne s'avère pas tellement plus compliqué que sur un ordinateur de bureau. Par sa nature distribuée, fédérant des sites administrés par des institutions différentes, la grille soulève des questions d'un autre ordre : quelles

sont les ressources disponibles ? À qui doit-on demander l'autorisation pour les utiliser ? Comment y transférer les programmes et les données ? Autant d'interrogations qui détournent l'utilisateur de son domaine d'expertise d'origine.

Une architecture hétérogène et évolutive. L'aspect multi-administration des grilles de calcul se traduit par des divergences sur les choix matériels et logiciels au sein de chaque site. Ces divergences s'expliquent par un manque de concertation entre les domaines d'administration, mais aussi par des raisons budgétaires ou techniques. Pour de nombreuses grilles de calcul, il en résulte une configuration hétérogène. Dans le cas d'un super-calculateur centralisé, les applications sont conçues et compilées pour une solution matérielle et logicielle fixée. Dans le cadre des grilles hétérogènes, cette approche n'est plus possible. Il faut prendre en compte les spécificités des différentes ressources qui peuvent être assignées au calcul.

Un système massivement multi-utilisateur. L'aspect centralisé des super-calculateurs permet d'avoir une approche globale de la gestion des requêtes : un point d'accès unique permet aux utilisateurs de soumettre leur travaux qui sont alors ordonnancés selon un algorithme fixé et connu. Dans le cas des grilles informatiques, selon la définition donnée par Foster dans [57], il n'y a pas de point d'entrée centralisé. Ceci est principalement dû à la multiplication des utilisateurs : un point d'entrée centralisé risque de devenir une ressource critique pour le reste du système. Une autre raison provient du côté multi-administration : chaque site offre un point d'entrée pour ses utilisateurs. L'ordonnancement devient lui aussi distribué, avec toute la problématique de la synchronisation entre les sites. Enfin, des questions de sécurité d'accès aux ressources et aux données personnelles viennent s'ajouter à l'aspect massivement multi-utilisateur des grilles de calcul.

Des infrastructures à grande échelle. Enfin, certaines grilles atteignent plusieurs dizaines de milliers de processeurs et autant d'utilisateurs, ce qui ajoute à la complexité de compréhension du système. À cette échelle, l'infrastructure est considérée comme dynamique : les ressources, voire des sites entiers, apparaissent et disparaissent au gré des défaillances ou des contrats de participation. Il devient difficile de fixer une image globale stable du système, ce qui rend chaque exécution différente, non-reproductible. Les grilles sont des architectures situées à mi-chemin entre les grappes de calculateurs et le réseau Internet. Mais comment concilier la facilité d'utilisation de petites infrastructures distribuées avec le potentiel de calcul d'un océan de ressources ?

L'une des approches visant à simplifier l'utilisation des grilles de calcul est la mise au point d'une aide logicielle. Cette aide logicielle se traduit par la notion de services de grille. Ils permettent l'installation et le lancement des applications distribuées sur la grille, de manière transparente pour l'utilisateur.

Définition 2.10 : ***service d'une grille*** — Un service d'une grille est une application installée sur un ou plusieurs sites de la grille. Il permet d'offrir des fonctionnalités visant à simplifier l'utilisation des ressources. Il repose sur des outils spécialisés dans l'information, la gestion des ressources, l'ordonnancement des tâches ou encore le déploiement d'applications.

Nous allons voir dans les sections suivantes qu'aujourd'hui il est possible d'utiliser les grilles de manière relativement simple et transparente. L'utilisateur peut, par exemple, faire totalement abstraction des ressources de calcul lors de la soumission de son application,

à charge ensuite aux services de la grille de sélectionner les ressources, d'y transférer les programmes et les données et de retourner le résultat final. Des travaux tentent de faire converger les services offerts par les grilles vers un standard unique.

Les nouveaux défis. Ils se situent dans une étape ultérieure. La prise en charge des applications sur les grilles se limite bien souvent à une approche *statique* : une fois l'application déployée sur un nombre fixe de ressources, les services attendent sa terminaison pour en retourner le résultat. L'aspect dynamique de l'infrastructure, mais aussi le comportement imprévisible des utilisateurs, rend cette approche peu efficace. Un effort important porte désormais sur la gestion *dynamique* des applications, notamment pour pallier les cas de défaillances de ressources ou, inversement, pour profiter de ressources inutilisées.

2.2 Une gestion transparente et autonome des applications

La prise en compte de la dynamicité dans les applications s'avère être une tâche encore plus complexe que l'utilisation statique des grilles. Il faut en effet savoir détecter et analyser les signes d'un possible dysfonctionnement, prendre les décisions adaptées, planifier les actions à effectuer puis les exécuter. Il n'est pas réaliste de laisser ces tâches à la charge de l'utilisateur. Tout d'abord parce qu'elles semblent *automatisables*. Et ensuite car elles sont fastidieuses et doivent être effectuées tout au long de l'exécution de l'application. Nous touchons ici au domaine de l'*informatique autonome* qui vise à proposer des solutions pour concevoir des applications dynamiques et réactives à l'environnement d'exécution.

2.2.1 Externaliser la réalisation des calculs

Une première approche visant à rendre les applications dynamiques est directement issue de l'analogie établie entre les grilles et le réseau de distribution d'électricité. Elle consiste à externaliser la réalisation des calculs. Le principe est, lors de la conception de l'application, de ne pas se soucier de l'aspect dynamique. Ceci est possible en s'appuyant sur un environnement d'exécution extérieur offert par les infrastructures de calcul. La particularité de cet environnement réside dans la dynamicité : il s'adapte à l'infrastructure mais aussi aux besoins des applications.

Les environnements d'exécution sont pré-déployés sur un ensemble de ressources physiques par l'administrateur de l'infrastructure. Les applications et les calculs sont soumis par le biais d'une interface utilisateur. L'environnement prend en charge la sélection des ressources et toutes les autres étapes du déploiement. Nous sommes donc en présence d'un *déploiement dans le déploiement*. L'environnement peut aussi détecter les défaillances ou les problèmes d'équilibrage de charge. Dans ces cas, il est en mesure de redéployer certaines parties de l'application, de relancer des calculs ou de migrer des processus d'une ressource physique à une autre.

Différentes approches pour externaliser les calculs ont été proposées. Une première approche reprend le principe des *systèmes à image unique* qui donnent l'illusion que l'ensemble des ressources forment une seule ressource. Elle se situe au niveau du système d'exploitation.

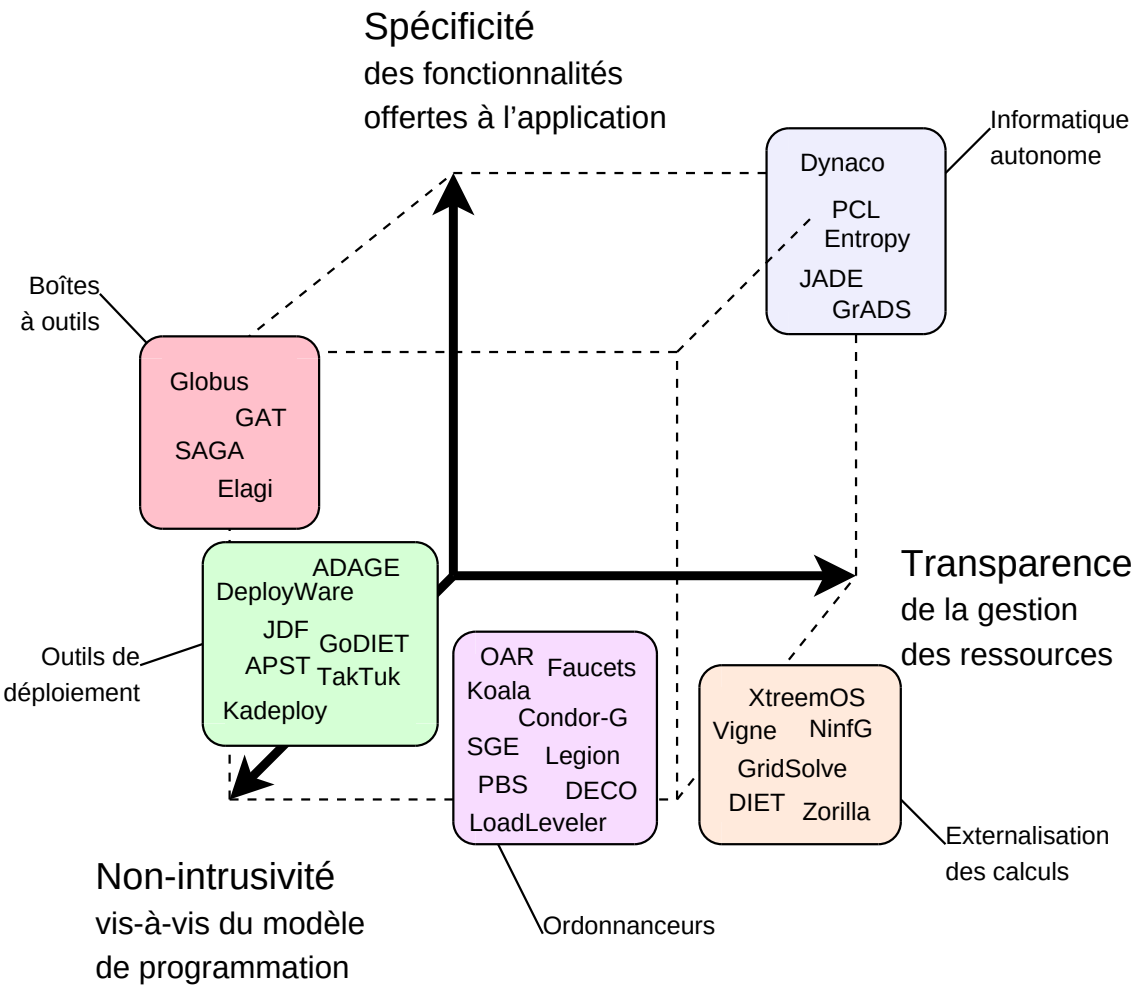

FIG. 2.6 – Les trois propriétés devant être satisfaites pour rendre les applications autonomes : 1) transparence de la gestion des ressources, 2) spécificité des fonctionnalités offertes à l'application et 3) non-intrusivité dans le programme. Positionnement de quelques-uns des différents systèmes cités dans ce manuscrit.

Le projet européen XTREEMOS [38] propose un système d'exploitation pour grilles. Le système d'exploitation doit être installé sur toutes les ressources de l'infrastructure. Cette approche est pertinente si l'on considère l'utilisation de machines virtuelles. Celles-ci permettent d'installer et de migrer facilement les images système d'une ressource à l'autre.

VIGNE [74, 75, 76, 105, 106, 107] est un environnement d'exécution pour applications distribuées reposant sur une architecture pair-à-pair. VIGNE doit être initialement déployé sur un ensemble de ressources physiques. Si nécessaire, il est possible d'ajouter dynamiquement de nouveaux pairs dans le réseau logique ou d'en retirer. Ceci est effectué manuellement en réservant de nouvelles ressources physiques sur lesquelles les pairs VIGNE peuvent être déployés.

Une seconde approche consiste à offrir des services au-dessus des systèmes d'exploitation. C'est le cas des *Web Services* ou autres variantes d'appels de procédure à distance (RPC). Par exemple, la spécification GRIDRPC API [111] reprend les concepts du RPC classique mais avec des notions propres aux grilles de calcul, comme le temps de traitement des requêtes, l'aspect asynchrone et massivement parallèle des appels ou encore la gestion de la sécurité. Différentes implémentations de cette spécification ont été proposées dans les projets NINF [98, 115, 97], GRIDSOLVE [131] ou encore DIET [33].

GRIDSOLVE [131, 15] nécessite l'installation locale, sur les stations de travail ou nœuds de calcul, d'environnements scientifiques tels que MATLAB [94]. Les programmes doivent donc être déjà installés et configurés sur les ressources. Ceci ne semble pas réaliste dans le contexte d'une infrastructure avec plusieurs domaines d'administration et ne facilite pas l'ajout d'une application par un utilisateur.

DIET [33, 34] (*Distributed Interactive Engineering Toolbox*) distribue les tâches sur plusieurs *agents*. Les agents sont organisés de manière hiérarchique : les clients soumettent leurs tâches à un ou plusieurs agents maîtres (MA), qui les transmettent à des agents locaux (LA) puis enfin à des travailleurs (SeD). Cette organisation suppose un déploiement initial des agents et travailleurs en un réseau logique prêt à recevoir des requêtes GRIDRPC. Ce déploiement est effectué de manière statique grâce aux outils GODIET [32] et ADAGE [81] présentés dans la section 2.3.2.

Enfin, nous pouvons citer l'intergiciel d'exécution Zorilla [49, 48, 125, 5], calqué sur l'organisation des systèmes pair-à-pair. Les pairs participant au réseau logique Zorilla peuvent supporter l'exécution de tâches soumises par des clients. L'approche retenue nécessite le déploiement initial et statique d'un réseau logique de pairs avant de pouvoir soumettre des tâches.

L'externalisation des calculs permet de masquer aux utilisateurs la gestion des ressources physiques. Tous ces systèmes doivent cependant être déployés de manière statique avant de pouvoir soumettre des applications. Le nombre de ressources allouées à une application utilisateur est adapté en fonction de ses besoins, mais seulement dans la limite des ressources utilisées par l'environnement de calcul. De plus, cette adaptation n'est autorisée que dans les cas les plus simples. Un tel cas est par exemple le redéploiement d'une partie de l'application qui a subi une défaillance. Un autre cas simple est la migration d'un processus sur une autre ressource pour des raisons d'équilibrage de charge. Ces opérations sont effectuées de manière *générique* par rapport à l'application. Celle-ci n'est pas en mesure d'indiquer des

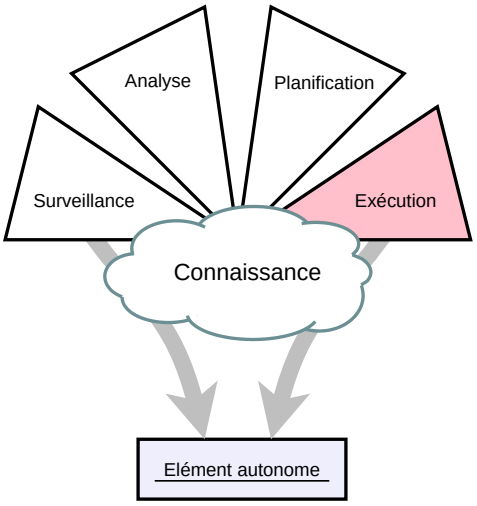

FIG. 2.7 – Architecture pour la gestion des applications autonomes.

besoins plus *spécifiques* : il n'est pas possible d'ajouter des contraintes sur le redéploiement ou d'effectuer un co-déploiement. Comme on le voit sur la figure 2.6, l'externalisation des calculs offre les propriétés de transparence et de non-intrusivité, mais pas celle de spécificité.

2.2.2 L'informatique autonome

Externaliser les calculs et reposer sur des services génériques limite les possibilités en terme d'adaptation d'application. L'*informatique autonome* tend à recentrer les procédures d'adaptation autour des applications pour mieux prendre en compte les besoins plus spécifiques.

2.2.2.1 Propriétés générales

Dans [72] et [52], IBM décrit sa vision de l'informatique autonome. L'autonomie des systèmes informatiques y est comparée à celle du système nerveux : la gestion des activités vitales de bas niveau, comme la pompe cardiaque ou encore la température du corps, est effectuée de manière transparente pour la partie consciente du cerveau. En informatique, cette approche doit permettre de décharger les administrateurs systèmes des tâches les plus ingrates pour se concentrer sur de la gestion et de la conception de plus haut niveau. La notion clé de l'informatique autonome réside dans l'*autogestion*. Cette notion se décompose en quatre sous-notions.

L'auto-configuration concerne le déploiement, l'installation, la configuration et l'intégration de grands systèmes. Ces phases, préliminaires à la mise en exploitation, se révèlent très complexes. Ceci est principalement dû à la multiplication des matériels et logiciels offrant bien souvent des interfaces complètement incompatibles. L'auto-configuration vise à automatiser ces processus d'intégration de nouvelles ressources. Celles-ci doivent être capables de se déclarer et d'annoncer les services qu'elles peuvent offrir et ceux nécessaires à leur bon fonctionnement.

L'auto-optimisation s'effectue pendant la phase d'exploitation. Les grands systèmes dépendent d'une multitude de paramètres qui impactent les performances d'exécution. Il est d'autant plus difficile d'optimiser ces systèmes qu'ils sont souvent formés de plusieurs sous-systèmes. Les performances sont alors dépendantes d'une combinaison de facteurs et de paramètres. L'auto-configuration doit donc permettre, grâce à un processus d'apprentissage, de définir quelles sont les meilleures valeurs pour les paramètres d'exécution à chaque instant.

L'auto-réparation vise à déclencher des procédures de remise en route des systèmes lors de la détection d'une défaillance. Le processus ne s'arrête pas là : un système autonome doit être capable d'identifier clairement la source de cette défaillance, par exemple en analysant des journaux et traces d'exécution, puis de proposer des modifications du système pour y remédier.

L'auto-protection permet de garantir l'intégrité d'un système face à des intrusions malveillantes ou tout simplement maladroites. Cette propriété inclut la détection des activités malicieuses et aussi la mise en place d'une stratégie de défense.

Dans ce manuscrit, nous nous intéressons plus particulièrement à la notion d'*auto-configuration*. Cette notion recouvre en effet l'ensemble des étapes nécessaires à la réalisation d'un déploiement d'application. Dans [80], des propositions d'architecture sont données afin de réaliser des systèmes autonomes. Le principe est illustré par la figure 2.7. Les éléments autonomes peuvent être des programmes ou des ressources physiques. Chacun d'entre eux est en relation avec un gestionnaire d'autonomie. Les gestionnaires sont divisés en cinq parties.

La surveillance permet de détecter les modifications de comportement ou signaux extérieurs envoyés par l'élément autonome.

L'analyse permet de filtrer les signaux reçus par la partie surveillance et de ne garder que les plus pertinents.

La planification d'un certain nombre d'opérations est effectuée à partir des signaux fournis par l'analyse. Ces opérations peuvent découler de règles stockées dans la base de connaissance.

L'exécution de la suite d'opérations planifiées lors de la phase précédente : ces opérations s'appliquent directement ou indirectement sur l'élément autonome.

La connaissance est partagée entre les différentes phases. Elle peut contenir aussi un historique du comportement de l'élément afin d'affiner les phases de planification.

Cette architecture décrit un cadre conceptuel repris par de nombreux environnements de programmation pour applications autonomes comme DYNACO [25] et JADE [103] présentés ci-après. Le travail effectué dans le cadre de cette thèse se positionne dans la phase d'*exécution*.

2.2.2.2 Quelques systèmes autonomes

La vision de l'informatique autonome introduite par IBM a inspiré plusieurs projets visant à offrir des canevas et des implémentations pour réaliser des systèmes autonomes. Parmi ces projets, nous pouvons citer DYNACO [25], JADE [103], Entropy [69], GRADS [17] et PCL [51].

Listing 2.8 – Exemple de plan produit par le planificateur de DYNACO.

```
1 déployer le programme sur la machine 'machine-19'
2 créer un processus sur la machine 'machine-19'
3 transférer le processus de la machine 'machine-13'
4           à celui de la machine 'machine-19'
5 mettre à jour les connexions réseau en remplaçant 'machine-13' par 'machine-19'
6 terminer le processus de la machine 'machine-13'
7 nettoyer la machine 'machine-13' du programme
```

JADE [103, 23] est un canevas permettant de rendre autonome l'administration d'applications complexes à déployer. Il repose sur un modèle de composants : les composants sont appelés *composants applicatifs* et ne requièrent *aucune modification du code de l'application*. JADE ne permet pas de prendre en compte des contraintes de co-localisation entre les composants, ni de prendre en compte des exigences matérielles ou logicielles pour le choix des ressources. L'aspect dynamique du déploiement n'est pas à l'initiative de l'application. Il est basé sur les notifications d'un système de surveillance extérieur. Cette approche limite la finesse et la pertinence des notifications. JADE semble ici moins adapté pour prendre en charge des besoins en déploiement plus spécifiques au type d'application. Par exemple, certains déploiements additionnels peuvent être motivés par des évolutions de l'application qui ne peuvent être observées par un système externe. Pour prendre en compte ces évolutions, l'application doit pouvoir directement interagir avec le système autonome. Pour cela, une modification du code de l'application est nécessaire, ce qui va à l'encontre de la politique d'encapsulation des applications proposée dans JADE. Ces limitations proviennent notamment d'un manque d'intégration de JADE avec les ordonnanceurs et les systèmes d'informations mis en place sur les plates-formes d'exécution. Elles proviennent aussi de l'absence d'interaction avec les applications pour prendre en compte des besoins plus spécifiques.

DYNACO [25, 26] est un canevas conçu pour mettre en œuvre des techniques d'adaptation au sein des logiciels. Dans ce projet, l'adaptation se définit comme la possibilité donnée à une application de choisir entre différentes implémentations d'une même fonctionnalité. Contrairement à JADE, DYNACO ne se limite pas qu'aux modifications de la topologie de l'application à l'aide de mécanismes extérieurs. Chaque application offre plusieurs implémentations dont l'exécution est adaptée à des situations différentes. DYNACO permet choisir dynamiquement la meilleure implémentation en fonction de l'environnement d'exécution. L'exemple de plan donné dans le listing 2.8 montre une suite d'actions de bas niveau. Dans le contexte d'un déploiement d'application, cette liste peut devenir rapidement longue et complexe, avec beaucoup de notions spécifiques à la plate-forme physique ainsi qu'aux outils de gestion des ressources. Il semble plus naturel d'exprimer dans la phase de planification de DYNACO des actions uniquement *spécifiques* à l'application. Ces actions doivent être de plus *haut niveau*. Dans le contexte d'un système de fichiers distribué, il serait intéressant d'offrir une action correspondant à l'augmentation de l'espace de stockage global et non pas d'indiquer le déploiement d'une entité en particulier. Dès lors, un intermédiaire doit se charger de traduire ces actions de haut niveau en une suite d'opérations de bas niveau pour les outils de gestion des ressources de l'infrastructure.

Entropy [69, 70] est un environnement pour applications autonomes lui aussi inspiré du manifeste d'IBM. Il repose sur la technologie de virtualisation des grilles de calculateurs [86, 16]. L'adaptabilité selon Entropy concerne le placement et la migration de ces machines virtuelles. Pour cela, des contraintes sont exprimées de la part des administrateurs et des utilisateurs du système. Un solveur de contraintes est utilisé pour trouver des solutions au problème d'affectation des machines virtuelles. Entropy est spécialisé dans l'adaptation au regard de l'état des ressources et non pas en fonction de l'exécution de l'application.

D'autres systèmes permettent aux applications de s'auto-adapter. Ils sont principalement axés sur l'équilibrage de charge et la migration de tâches. À cet effet, nous pouvons citer des systèmes tels que GRADS [17, 122, 79], Violin [108], *Adaptable Resource Broker* [102], PCL [51, 50] et Océano [14].

2.2.3 Conclusion

Les canevas et systèmes de conception pour applications autonomes sont très prometteurs : ils permettent d'avoir une vision globale et le contrôle total de toutes les étapes nécessaires pour rendre les applications autonomes. Pour le développeur, ces systèmes restent cependant très contraignants à mettre en œuvre : ils impliquent bien souvent un modèle de programmation très strict, souvent à base de composants. Ceci entraîne des contraintes sur la conception de l'application qui peuvent remonter jusqu'aux phases de spécifications des fonctionnalités. L'approche est donc ici très *intrusive* dans le processus de développement. De plus, les canevas ne permettent pas toujours de prendre en compte les besoins spécifiques des applications. Ces notions sont reprises dans la figure 2.6. Dans la suite de ce chapitre, nous regardons plus en détail comment apporter de la dynamicité aux applications en utilisant des technologies *peu intrusives* dans la conception de l'application. Ce sont notamment des technologies largement représentées sur les grilles de calculs.

2.3 Concilier la dynamicité des applications avec les services de la grille

Les grilles de calculs ont rapidement suscité l'intérêt d'un grand nombre d'utilisateurs plus ou moins spécialisés dans le domaine de l'informatique. Afin d'en faciliter l'accès, des solutions logicielles ont été mises en place. Elles concernent toutes les étapes du déploiement d'applications distribuées. Ainsi, des systèmes d'information permettent d'obtenir une image globale du statut des ressources de la plate-forme. Des boîtes à outils logicielles offrent un ensemble de services pour transférer les exécutables et les données, ainsi que démarrer les tâches. Des ordonnanceurs organisent le flot des requêtes et distribuent les travaux sur les ressources. Enfin, des outils dédiés au déploiement permettent de prendre en charge tout un ensemble d'applications distribuées de manière spécifique. Dans cette section, nous offrons un panorama des services et des outils représentés dans les grilles de calcul.

2.3.1 Interactions avec la plate-forme d'exécution

Avec la mise en place de grilles informatiques de grande taille, il n'est plus réaliste de laisser la gestion des ressources à la charge des utilisateurs, ni même à celle des adminis-

trateurs. Différents services de gestion des ressources physiques et des tâches ont donc été proposés.

2.3.1.1 Outils de gestion des ressources : systèmes d'information

Les systèmes d'information ont la charge de maintenir une vision locale ou globale des ressources physiques de l'infrastructure. Ces informations contiennent le statut de chaque ressource : libre, réservée, suspectée de panne ou éteinte. Elles devraient contenir aussi toutes les caractéristiques pouvant aider au choix de la ressource la plus appropriée pour supporter l'exécution d'une tâche. Ces caractéristiques concernent le type de matériel présent (processeur, mémoire, disques), le logiciel (système d'exploitation, bibliothèques) mais aussi le réseau (technologies) et les connexions établies avec les autres ressources. Parmi les principaux systèmes d'information nous pouvons citer NWS [130], GANGLIA [91] et MonALISA [99]. Nous présentons les deux premiers.

NWS [130, 114] est un système de surveillance de ressources physiques. Il est distribué et permet d'effectuer des prévisions de performance à court terme en se basant sur l'historique des mesures.

GANGLIA [91] est un système de surveillance hiérarchique dédié aux fédérations de grappes. Il se base sur un protocole de diffusion multi-point.

Les systèmes d'informations sont un élément déterminant dans la gestion des ressources : ils permettent de connaître en temps réel l'état de l'infrastructure. Cette information est utilisée pour effectuer le choix des ressources mais aussi pour anticiper l'adaptation des applications. Ces systèmes font partie de l'étape de surveillance dans le modèle de l'informatique autonome.

2.3.1.2 Les boîtes à outils pour grilles

Les boîtes à outils offrent un ensemble d'utilitaires pour améliorer l'utilisation des grilles. Ces utilitaires peuvent être intégrés sous forme de services de grille, d'exécutables ou de bibliothèques pour les applications. Ils concernent principalement la phase d'exécution dans le schéma de l'informatique autonome. Parmi ces boîtes à outils, nous pouvons citer Globus [56], GRIDLAB GAT [4], ELAGI [138] et SAGA [66].

Globus [56, 60] est une boîte à outils logicielle largement utilisée dans les systèmes actuels : plates-formes expérimentales, plates-formes de production ou instituts de calculs comme TERAGRID [104], le CERN [150] ou le *Earth System Grid* [18]. Globus est composé de multiples services et bibliothèques [63, 3, 64, 41] : ils permettent de découvrir, contrôler et surveiller les ressources, de prendre en charge les procédures de sécurité ou encore d'accéder à distance à des fichiers. Depuis quelques versions, la boîte à outils repose sur une interface de type *services en ligne* (*Web Services*). La soumission des tâches auprès du méta-ordonnanceur s'effectue de manière interactive pour l'utilisateur à l'aide d'un fichier de description dans lequel doivent être spécifiés le chemin vers l'exécutable ainsi que les paramètres à employer.

Le *Grid Application Toolkit* GAT [4, 5] vise à offrir une interface unifiée pour simplifier et rendre transparent l'utilisation d'intergiciels comme Globus, CONDOR-G [118] ou

Unicore [37]. Avec un tel outil, la gestion d'une application dans un environnement dynamique reste à la charge du développeur de cette application qui doit lui-même décider de la création de nouveaux processus et gérer leur déploiement. De plus, l'utilisation du GAT implique la présence physique de l'utilisateur, notamment pour la surveillance et le contrôle de son application à travers une interface de type *Web Services*.

Le groupe de recherche SAGA [66, 159] (*Simple* API *for Grid Application*) propose une interface unifiée d'accès aux intergiciels de la grille. Ce projet est issu d'une collaboration menée par les équipes de GRIDLAB, Globus et REALITYGRID [22, 83]. SAGA est reconnu comme une spécification standard de l'*Open Grid Forum* [155]. Il est retenu comme interface de programmation dans le système d'exploitation pour grilles du projet XTREEMOS [38].

Les boîtes à outils logicielles sont largement répandues sur les plates-formes expérimentales et de production. Elles font appel à des notions simples, inspirées par les fonctionnalités offertes sur un ordinateur personnel, mais adaptées à l'environnement grille. Par exemple, la copie d'un fichier entre deux ressources est facilitée par des protocoles de transferts tels que GRIDFTP [3]. Ces services restent de bas niveau : l'automatisation complète du cycle de déploiement d'une application demande un effort de programmation pour interagir avec les systèmes d'information, les outils de transferts et les ordonnanceurs. Cet effort doit être effectué pour chaque application et chaque environnement d'exécution. Les boîtes à outils n'offrent que rarement la possibilité de gérer les ressources de manière transparente. Elles permettent cependant de s'adapter aux besoins des applications sans pour autant imposer un modèle de programmation. Ces propriétés permettent de positionner ces systèmes sur la figure 2.6.

2.3.1.3 Ordonnanceurs

Le rôle d'un ordonnanceur est de décider du meilleur moment pour l'exécution d'une tâche. Ceci doit permettre un partage juste et efficace des ressources de la grille entre les utilisateurs. L'ordonnanceur joue un rôle d'autant plus important que les utilisateurs sont nombreux. Les principales qualités d'un ordonnanceur sont : 1) une interface de soumission simple qui permet d'exprimer des contraintes sur l'exécution ; 2) une prise en charge automatique et transparente de la soumission ; 3) une interface permettant de surveiller l'exécution ; et enfin 4) la possibilité d'établir des priorités dans l'exécution des tâches.

De nombreux ordonnanceurs ont été proposés pour les grilles, souvent issus d'infrastructures plus modestes, comme un calculateur ou une grappe de calculateurs. De nouveaux problèmes se posent à l'échelle des grilles : il ne semble pas réaliste de concevoir un ordonnanceur global pour une telle plate-forme. En effet, cette approche pourrait mener à une surcharge du système en fonction du nombre d'utilisateurs, du nombre de soumissions, du nombre de ressources à gérer et de la finesse des contraintes à prendre en compte. Enfin, l'aspect multi-administration des grilles condamne rapidement l'usage d'un ordonnanceur global centralisé. Parmi les ordonnanceurs largement utilisés sur les infrastructures de calcul, nous pouvons citer SGE [61], PBS [68], Nimrod/G [1, 28], LOADLEVELER [151], DECO [2] et Legion [67]. Nous choisissons de présenter succinctement Condor [87], KOALA [93] et Faucets [77] pour les différentes approches qu'ils proposent. OAR [29] sera présenté en détail dans la section 3.1.2.

Condor-G [87, 118] profite de la standardisation des accès aux différents ordonnanceurs de tâches offert par Globus. Cela permet d'interconnecter Condor-G avec des domaines gérés par d'autres ordonnanceurs, ce qui en fait un ordonnanceur adapté pour les grilles de calculateurs. Condor-G n'est cependant pas adapté pour le déploiement d'applications complexes comme des applications construites par couplage de codes. Il ne permet pas non plus, à l'exception de la migration et du redémarrage de tâches, de prendre en charge le redéploiement d'une partie de l'application.

KOALA [93] est un ordonnanceur spécialisé dans le problème de la co-allocation des tâches et des données. La politique de placement *close-to-file* y est utilisée afin de minimiser la distance entre les tâches et les données. Ceci est particulièrement intéressant dans le cadre de systèmes multi-grappe, où les transferts réseaux deviennent coûteux. KOALA repose sur les outils offerts par Globus, notamment pour la description des tâches, le service d'information et le module de gestion d'allocation des ressources.

Faucets [77] est spécialisé dans l'ordonnancement de tâches issues de programmes écrits en *Adaptive* MPI [20]. AMPI est une implémentation de MPI [54] qui supporte dynamiquement l'équilibrage de charge. Cette technologie d'application permet notamment de faire évoluer le nombre de processus communicants attachés à une même tâche. Il est donc possible d'étendre la topologie d'une application ou de la réduire, de manière dynamique, en fonction de la disponibilité des ressources. Cependant, l'utilisation de la technologie AMPI réduit le spectre des applications concernées par cette stratégie, même si la conception d'un convertisseur d'applications MPI vers AMPI est prévue.

Il est acquis que les ordonnanceurs sont la pièce maîtresse dans la bonne gestion des travaux soumis à une plate-forme de calcul. La taille et l'architecture des grilles impliquent la mise en place de plusieurs ordonnanceurs, répartis au sein de chaque site. L'ordonnancement y est donc distribué et peut être pris en charge par des systèmes différents. Par exemple, nous pouvons imaginer qu'un site utilise SGE alors qu'un autre site utilise CONDOR. Des projets tentent d'unifier les interfaces offertes par les ordonnanceurs pour simplifier l'accès aux utilisateurs. L'idée est d'offrir un protocole commun pour piloter les différents systèmes rencontrés dans la grille : c'est la notion de *méta-ordonnancement*, introduite par des projets comme GridWay [141] et DRMAA [120]. Le méta-ordonnancement permet aussi d'effectuer des réservations sur plusieurs sites en une seule demande.

Dans le contexte du déploiement d'applications, les ordonnanceurs peuvent prendre en charge toutes les étapes. Ils sont en effet en relation avec les systèmes d'information, ils effectuent la sélection des ressources et lancent les exécutables à distance. Certaines phases posent cependant problème, comme la phase de description de l'application ou la phase de configuration. Le point commun de ces deux phases est de porter sur des propriétés *spécifiques* à l'application déployée. Les interfaces utilisateur offertes par la plupart des ordonnanceurs ne permettent de prendre en charge que des configurations simples de déploiement. Ainsi, en pratique, l'utilisateur contourne ces limitations en ne soumettant pas directement son application, mais *un programme de déploiement spécifique à son application*, comme un script shell. Ce programme a la charge d'effectuer toutes les phases de configuration, de transfert des données et de lancement de l'application.

La plupart des ordonnanceurs prennent en charge la dynamicité de l'application en se focalisant sur l'équilibrage de charge et le redémarrage de tâches lors d'une défaillance. Très peu de mécanismes sont offerts pour permettre à l'application de s'adapter en prenant en

compte des contraintes de co-localisation par exemple. Les interactions entre applications et ordonnanceurs sont limitées à la surveillance des processus. Les ordonnanceurs sont, à l'exception de Faucets, aucunement intrusifs sur le modèle de programmation des applications. Ils masquent partiellement la gestion des ressources et ne sont pas spécifiques à une application, comme illustré sur la figure 2.6.

2.3.1.4 Aide pour la réservation de ressources physiques

Les ordonnanceurs ne sélectionnent pas toujours des ressources pertinentes pour l'exécution d'une application. L'utilisateur peut avoir intérêt à privilégier certaines ressources par rapport à d'autres. Par exemple, l'ordonnanceur devrait prendre en compte la présence d'une bibliothèque logicielle sur un site en particulier ou alors la présence de données qui ne peuvent être facilement transférées. Dans ce cas, des outils permettent à l'utilisateur de choisir lui-même les ressources physiques. Ils reposent sur les systèmes d'information pour présenter l'état des ressources.

GRUDU [142] est un utilitaire d'aide à la réservation et à la surveillance des ressources physiques. Il met l'accent sur l'interface avec l'utilisateur qui est très intuitive. Elle représente les ressources réservées sur une carte topologique de la grille de calcul. Un module basé sur JFTP [144] est aussi disponible pour transférer, manuellement, des données. GRUDU facilite les interactions avec les outils de gestion des ressources pour les utilisateurs qui ne souhaitent pas utiliser une interface en ligne de commande.

KATAPULT [146] est un script propre à l'environnement logiciel de GRID'5000 : il permet d'automatiser le déploiement d'une image système grâce à KADEPLOY [62], de vérifier le bon fonctionnement des ressources après le déploiement et de lancer un script de post-configuration sur les ressources ainsi configurées. KATAPULT s'utilise comme un programme à exécuter lors de la réservation des ressources avec l'ordonnanceur de tâches OAR [29].

Les outils d'aide à la sélection de ressources soulagent l'utilisateur désireux de plus de liberté dans le choix des ressources. Ils ont cependant comme désavantage principal de replacer l'utilisateur au centre du processus de déploiement, alors que la tendance voudrait que toutes ces étapes soient opérées de manière transparente.

2.3.2 Outils de déploiement

Les outils de déploiement sont directement issus du besoin de lancer les applications de manière *automatique* et *répétée*. Ils sont particulièrement appréciés par les scientifiques désirant effectuer des expériences. Celles-ci nécessitent en effet souvent de déployer plusieurs fois la même application en faisant varier des paramètres à chaque fois. Parmi les outils de déploiement nous pouvons citer DeployWare [53], OpenCCM [90, 154], Pegasus [44, 43], SmartFrog [65], Concerto [88], TAKTUK [89], APST [35], et XtremWeb [31]. Dans cette section, nous présentons plus en détail les outils faisant partie de la thématique du déploiement d'applications sur grilles : JDF [11], GODIET [32], ADAGE [81] et KADEPLOY [62]. Ces outils sont particulièrement bien adaptés au déploiement d'applications et de services distribués comme JXTA [119], JUXMEM [10], GFARM [117] et DIET [33]. Ces applications

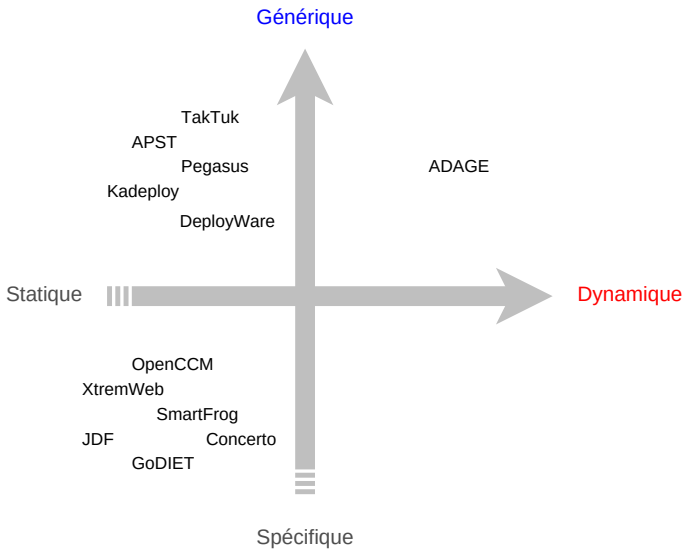

FIG. 2.9 – Classification des outils de déploiement selon leur niveau de généricité et de dynamicité.

et services pour grille présentent des particularités topologiques et de déploiement très variées. Elles sont utilisées comme cas d'étude, dans le contexte de la plate-forme expérimentale GRID'5000. Ces applications sont présentées plus en détail dans la suite de ce manuscrit.

JDF [11] (JXTA *Distributed Framework*) est un outil de déploiement proposé par Sun Microsystems et dédié aux applications basées sur la plate-forme pair-à-pair JXTA [119]. Cet outil de déploiement reste cependant statique, c'est-à-dire qu'il ne permet pas facilement d'ajouter, à la demande, des pairs dans un réseau logique déjà déployé. Il ne permet pas non plus de prendre en compte des contraintes avancées de placement, comme la co-localisation de deux pairs sur la même ressource physique ou la même grappe. Enfin, la création d'une description du réseau logique par l'utilisateur, effectuée dans un formalisme proposé par Sun Microsystems, reste pénible pour des configurations complexes.

GODIET [32] est un outil dédié au déploiement des applications basées sur la plate-forme d'exécution distribuée DIET [33]. Il est statique et ne prend en charge ni le redéploiement d'une application ni le co-déploiement d'un ensemble d'applications. Il reste par ailleurs dédié aux applications basées sur DIET. Son utilisation repose sur la création manuelle d'une description de l'application à déployer. Ce point a été récemment modifié pour prendre en compte l'aspect récursif de certaines organisations logiques [47]. Par ailleurs un utilitaire nommé XMLGoDIETGenerator [161] simplifie la génération de la description de l'application en ne demandant qu'un minimum d'informations sur le déploiement à effectuer.

ADAGE [81] est un outil de déploiement *générique* pour applications parallèles et distribuées. Il permet cependant de mettre en place des mécanismes pour prendre en compte les besoins *spécifiques* de chaque application. Le but est d'automatiser le processus de déploiement en prenant en charge les phases de sélection des ressources, de planification du déploiement, de transfert de fichiers, de configuration des ressources, de lancement des processus et enfin de post-configuration. L'outil de déploiement est

aussi capable d'effectuer du redéploiement d'application. Ceci est possible en modifiant la description initiale d'une application. Ces modifications soumises à ADAGE se traduisent alors en un déploiement additionnel des constituants ajoutés, en prenant en compte ceux déjà lancés. ADAGE supporte le co-déploiement, c'est-à-dire qu'il est possible de déployer, en une seule fois, plusieurs applications, éventuellement de types différents. ADAGE est présenté de manière plus complète dans la section 3.1.3.

KADEPLOY [62, 145] est un outil de déploiement d'environnements logiciels. Il permet d'installer une image système sur les partitions disques de plusieurs ressources. Cette approche demande un travail pointu dans la conception d'environnements logiciels, notamment des compétences système pour la réalisation d'une image à déployer.

L'existence de nombreux outils de déploiement montre l'ampleur du problème rencontré par les utilisateurs : avec le développement d'infrastructures à grande échelle, le lancement des applications devient complexe, au point de nécessiter l'automatisation des étapes de déploiement. Nous pouvons classer ces outils suivant deux axes de conception comme cela est montré sur la figure 2.9. Le premier axe permet de faire la distinction entre les outils de déploiement dédiés à certaines applications ou génériques. Les outils de déploiements les plus évolués sont génériques, c'est à dire qu'ils ne se limitent pas à un type d'application. Cette caractéristique ne doit cependant pas empêcher l'outil de prendre en charge des besoins spécifiques aux applications. Le deuxième axe permet de les classer selon leur degré de dynamicité.

À notre connaissance, seul l'outil ADAGE [81] offre des fonctionnalités permettant la reconfiguration en ligne d'une application générique sans en modifier le programme. ADAGE n'offre cependant pas une interface très simple pour l'utilisateur : les interactions nécessitent la manipulation de concepts bas niveau, comme la description explicite de la topologie de l'application et des contraintes internes à prendre en compte pour le déploiement.

2.4 Conclusion

Les applications scientifiques profitent des avancées technologiques effectuées dans les infrastructures de calcul à grande échelle. Les grilles de calcul s'avèrent être des plates-formes tout à fait adaptées pour des applications massivement distribuées comme le couplage de codes ou la conception collaborative. Malheureusement, l'accumulation de ressources hétérogènes réparties dans différents domaines d'administration rend l'utilisation des grilles informatiques difficile et *peu efficace*. La taille des infrastructures ainsi que l'utilisation de ressources physiques grand public entraîne un nombre de défaillances non négligeable. Ces paramètres sont désormais à prendre en compte par les utilisateurs afin de tirer parti des grilles. Ceci est d'autant plus vrai lorsqu'il s'agit du *déploiement* d'application, une étape souvent mésestimée, qui devient critique dans un environnement dynamique à grande échelle.

Des solutions logicielles ont été proposées afin de simplifier l'accès aux grilles. Elles tentent de rendre transparent toute la gestion des ressources et l'exécution des applications. Une première approche consiste à externaliser la gestion de l'exécution grâce à l'utilisation d'environnements d'exécution. Ces environnements sont le plus souvent limités à des considérations d'équilibrage de charge ou de migration des processus en cas de défaillance. Il

est difficile de prendre en compte des besoins en déploiement très spécifiques aux applications, comme le co-déploiement et la co-localisation de processus sur une même ressource physique.

Une approche prometteuse met en avant la notion d'*informatique autonome*. Le principe fondateur est l'*autogestion* de l'application. Il se décompose en les quatre phases de surveillance, de décision, de planification et d'exécution. L'application est intégrée dans un canevas d'*adaptabilité* prenant en charge ces quatre phases. Si cette approche permet de rendre les applications autonomes, elles nécessitent pour la plupart la formation des développeurs à un modèle de programmation à base de composants, ainsi que l'utilisation d'un moteur d'exécution spécifique (*runtime*).

L'approche radicalement opposée consiste à regarder comment apporter de la dynamicité dans les applications en intégrant les technologies présentes sur les grilles de calcul. De nombreux outils, services et bibliothèques pour applications distribuées sont largement acceptés par la communauté scientifique. Ils permettent de surveiller les ressources, d'effectuer des réservations, de transférer les données, d'ordonnancer et lancer les tâches. Des outils spécialisés dans le déploiement sont aussi disponibles, notamment des outils prenant en charge de manière *générique* et *dynamique* des applications distribuées. Une première difficulté liée à cette approche provient de la multitude des systèmes existants et du manque d'unification des interfaces. Une solution pour rendre une application dynamique sur telle grille ne sera plus valable sur une autre plate-forme de calcul. Une deuxième difficulté provient des interfaces de *bas niveau* proposées par les outils de déploiement : l'utilisateur est trop souvent obligé de fournir des descriptions lourdes de son application et des contraintes de déploiement associées. Les systèmes présentés sur la figure 2.6 illustrent l'absence d'une approche pour le déploiement dynamique qui satisfait les trois propriétés de *transparence*, *spécificité* et *non-intrusivité*.

Afin d'essayer de répondre à cette attente, nous effectuons, dans la suite de ce manuscrit, une étude de cas autour de la grille expérimentale GRID'5000. Nous présentons les outils mis à disposition des utilisateurs et montrons comment les déploiements d'application sont aujourd'hui effectués de manière statique. L'application motivante est un service distribué de partage de données en mémoire physique. Ce service permet des accès transparents à des données répliquées sur différentes ressources physiques. Il a été initialement mis en œuvre sans prendre en compte l'aspect dynamique de la topologie. La topologie détermine l'espace de stockage global et les performances d'accès aux données. Ces caractéristiques sont principalement tributaires du nombre de ressources physiques utilisées par le service pour stocker les données ainsi que de leur localisation dans le réseau. Or, les besoins en espace de stockage et en performance peuvent évoluer pendant la durée de vie des clients et il est difficile de prévoir ces besoins avant le déploiement initial du service.

L'objectif est donc de rendre dynamique le service de partage de données, en fonction des besoins en stockage des clients et des évolutions de la plate-forme d'exécution. Pour cela, le service doit être capable de réagir à un dysfonctionnement ou à la dégradation des performances en demandant la modification de sa topologie par expansion ou rétraction. La prise en charge de la dynamicité doit cependant satisfaire les trois propriétés énoncées en introduction de ce chapitre.

– Les actions offertes au service doivent être *spécifiques* : par exemple pouvoir étendre la topologie sur une ressource proche d'un client en particulier. De plus, ces actions doivent être de *haut niveau* avec une sémantique faisant abstraction des couches basses.

- La gestion des ressources doit être *transparente* : le service ne doit pas interagir avec des outils de réservation ou de déploiement. Il ne doit pas non plus avoir connaissance des ressources physiques qui supportent son exécution.
- La prise en charge de la dynamicité doit se faire de manière *non intrusive* dans le code du service : les modifications doivent se limiter à l'ajout dans le code existant de quelques instructions pour lancer les actions spécifiques.

Chapitre 3

Le déploiement d'applications sur GRID'5000

Sommaire

Le développement des infrastructures de type grille de calculateurs s'accompagne d'une intensification des activités de recherche. Ces recherches visent à améliorer les performances des applications scientifiques. Elles sont menées à plusieurs niveaux dans les couches logicielles, comme cela est montré sur la figure 3.1. Ces niveaux comprennent les couches basses optimisées pour les communications haute performance, les systèmes d'exploitation, les intergiciels de gestion de l'exécution de l'application et les nouveaux paradigmes de programmation pour grilles de calculateurs. La mise en place et l'expérimentation de nouvelles technologies logicielles dédiées à la grille peuvent être effectuées selon les schémas classiques et complémentaires de simulation, d'émulation et d'expérimentation en conditions réelles. La figure 3.2 présente les différentes méthodologies utilisées pour l'étude des systèmes distribués, en les classant selon un critère allant du monde de l'abstrait au monde du réel. La complexité des systèmes distribués à grande échelle - principalement due aux nombreuses interactions établies entre les ressources - rend l'utilisation du modèle analytique, anecdotique. Des approches plus simples à mettre en œuvre résident en la simulation et

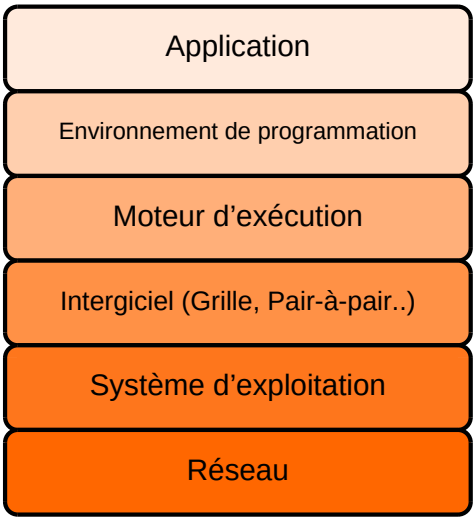

FIG. 3.1 – Les différentes couches logicielles étudiées au sein du projet GRID'5000.

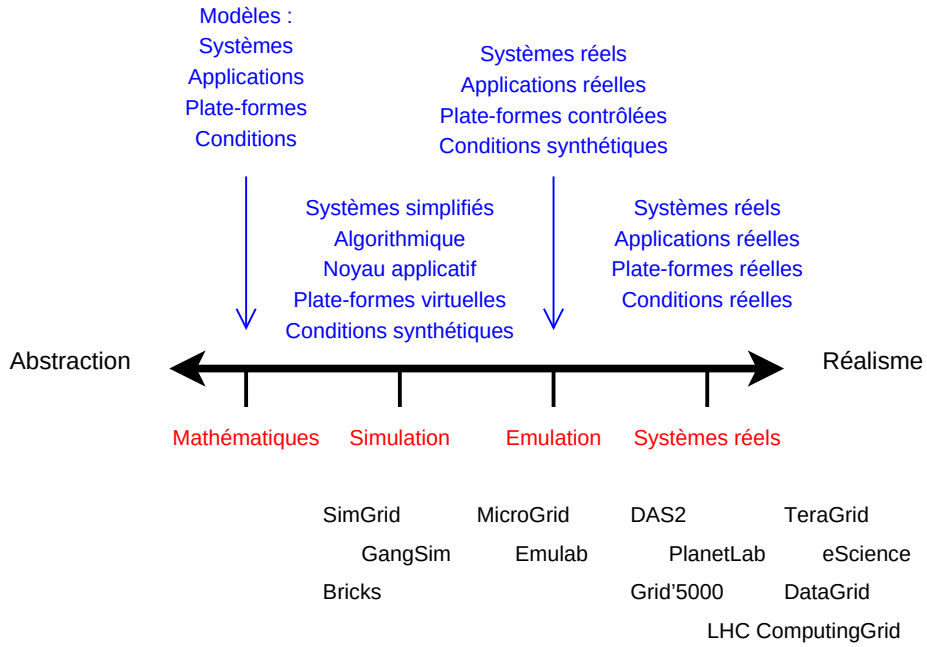

FIG. 3.2 – Méthodologies utilisées pour l'étude de systèmes distribués.

l'émulation. Elles permettent de contrôler tous les paramètres et conditions d'expérimentation de l'application. Elles requièrent cependant un modèle de mise en œuvre des ressources physiques suffisamment pertinent pour refléter le comportement de la plate-forme d'exécution et ainsi se rapprocher des expérimentations réelles. La taille des infrastructures de type grille et la complexité des applications qui y sont exécutées rendent l'étude de ces systèmes difficile par des approches abstraites.

Dans ce chapitre nous présentons l'approche retenue sur GRID'5000, une plate-forme expérimentale hautement reconfigurable. Cette plate-forme est mise en place pour étudier les infrastructures de type grilles de calcul. Nous nous focalisons ensuite sur deux outils dédiés à la réservation de ressources et au déploiement d'applications. Enfin, nous revenons à la problématique du déploiement à travers un exemple complet, mettant en scène divers prototypes de recherche.

3.1 L'approche du déploiement sur la grille GRID'5000

C'est dans ce contexte de forte demande en plate-forme expérimentale à grande échelle que le projet GRID'5000 [30, 139] a été proposé. Cette grille de calculateurs doit permettre d'effectuer des expérimentations dans des conditions réelles tout en restant hautement configurable. Elle doit de plus permettre la reproductibilité des expériences. Dans ce chapitre nous présentons la plate-forme GRID'5000 ainsi que les outils mis à la disposition des utilisateurs. Nous nous focalisons ensuite sur deux outils de gestion des ressources et du déploiement d'application : OAR et ADAGE.

3.1.1 Présentation de la plate-forme GRID'5000

GRID'5000 est une initiative de recherche lancée par le programme de l'Action Concertée Incitative GRID, le CNRS, l'INRIA et RENATER. Son but est de proposer une grille de calcul hautement configurable et contrôlable. L'architecture GRID'5000 est distribuée sur neuf sites en France, comme cela est montré sur la figure 3.3. Les sites partenaires sont Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. À terme, la grille regroupera jusqu'à cinq mille processeurs en s'appuyant sur les infrastructures de 17 laboratoires, spécialisés en informatique, mathématiques, télécommunications, modélisation et simulation.

3.1.1.1 Topologie réseau

L'architecture réseau générale de GRID'5000 est dictée par l'organisation hiérarchique des ressources. Celles-ci sont groupées au sein de grappes de calculateurs, eux-mêmes groupés dans des sites. L'interconnexion des différents sites est assurée par les réseaux à haut débit fournis par RENATER[1] [158, 157]. L'ossature du réseau suit celle de RENATER, permettant d'allouer jusqu'à 10 Gb/s de bande passante entre les sites grâce à l'utilisation d'une *fibre optique noire virtuelle*. Cette fibre est caractérisée par un multiplexage, au niveau physique, sur les fréquences et propose plusieurs canaux appelés *lambdas*. Le réseau RENATER

[1] Réseau national de télécommunications pour la technologie, l'enseignement et la recherche.

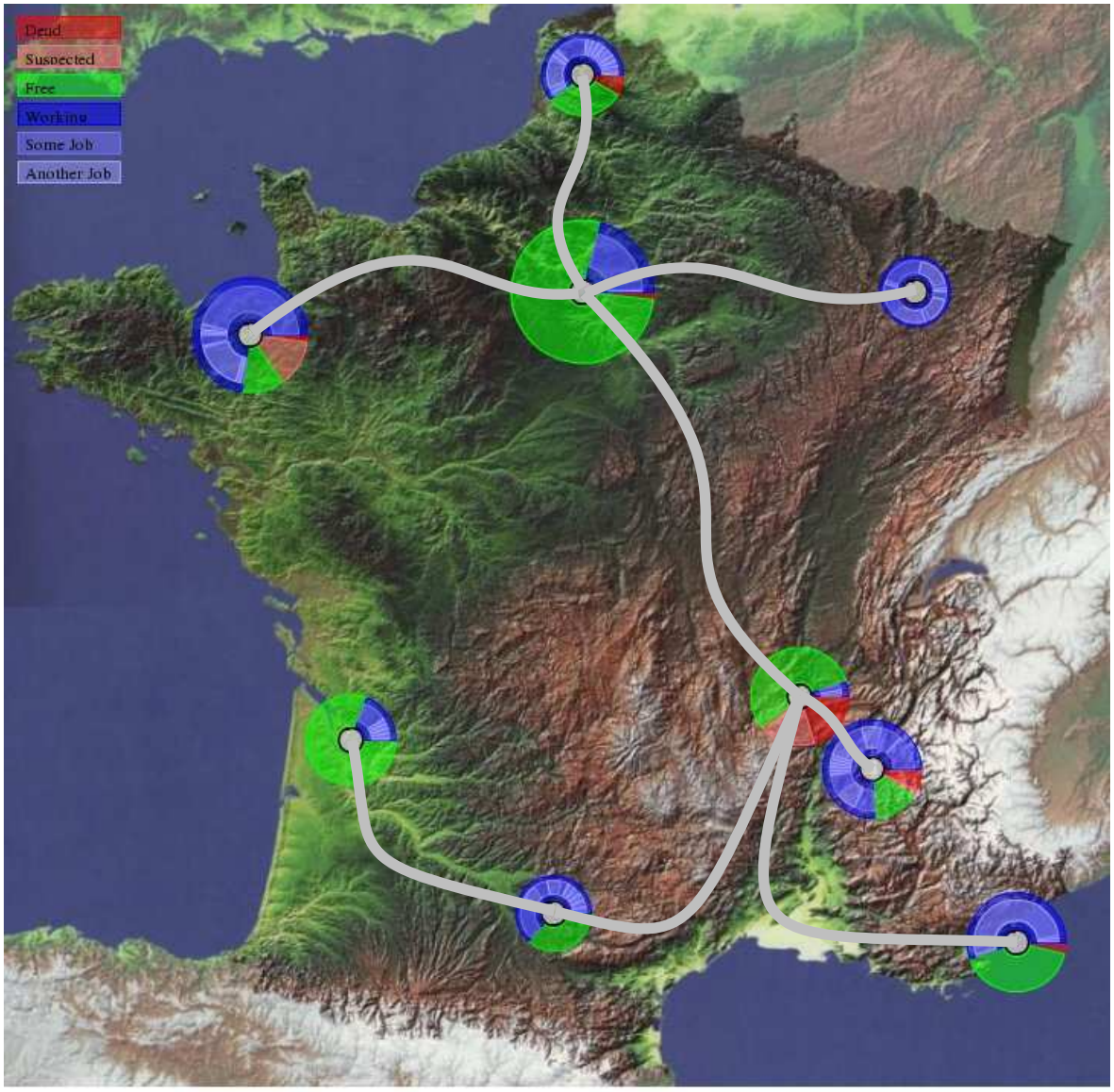

FIG. 3.3 – Carte des sites de la grille expérimentale GRID'5000 vue par l'outil de visualisation des ressources développé par Thomas Hérault (LRI/Orsay). Les sites sont reliés par un réseau 10 Gb/s reposant sur l'infrastructure RENATER.

alloue un lambda de 10 Gb/s au projet GRID'5000. La latence est de l'ordre de 10 millisecondes entre les sites. Elle dépend du chemin emprunté et du type de matériel traversé entre les différentes ressources. Les sections en fibre optique noire sont représentées sur la figure 3.3. Les ressources sont inter-connectées, au sein de chaque site, par un routeur ou une hiérarchie de routeurs. Plusieurs technologies sont utilisées : Ethernet, Myrinet ou encore InfiniBand, assurant des latences de l'ordre de la centaine de microsecondes pour l'Ethernet et de l'ordre de la microseconde pour les réseaux haute performance. Ces latences sont faibles en comparaison de celles observées pour les communications inter-sites. Le réseau GRID'5000 peut donc être considéré comme un réseau hiérarchique à deux niveaux. Le premier niveau concerne les communications intra-sites et le deuxième niveau concerne les communications inter-sites. Ces deux niveaux doivent être pris en compte par les applications lorsqu'elles sont distribuées sur plusieurs sites de la grille. Ils conditionnent largement les performances des applications lors de l'envoi de messages.

3.1.1.2 Hétérogénéité des ressources

Issue de la fédération de domaines administratifs indépendants, la grille GRID'5000 est, à l'image de quelques autres grilles expérimentales et de production, composée de ressources physiques hétérogènes. Cette hétérogénéité concerne, à l'origine, un tiers des ressources. Cette caractéristique est, d'une part, la conséquence de l'autonomie dont jouissent les sites pour gérer leur parc informatique. Elle est, d'une autre part, issue de la volonté scientifique de refléter l'architecture matérielle des grilles de production. L'hétérogénéité des ressources intervient à plusieurs niveaux, du matériel au logiciel. Les principales caractéristiques attachées aux ressources concernent notamment le processeur, avec des architectures allant de 32 bits à 64 bits, du mono processeur au quadri processeur, du simple cœur aux processeurs à huit cœurs. Les types de processeurs représentés sont majoritairement basés sur les constructeurs Intel et AMD, avec quelques PowerPC aujourd'hui inutilisés. La taille de la mémoire physique des nœuds varie entre 1 Go et 32 Go. Les interfaces réseau utilisent des cartes gigabit Ethernet 1G, InfiniBand 10G ou encore Myrinet 10G. Les disques durs offrent des espaces de stockage de 30 Go à 600 Go. D'un point de vue logiciel, les systèmes d'exploitation déployés par défaut sur chacune des ressources sont mis en place par les responsables des sites. Si l'utilisation des systèmes à base d'Unix sont généralisés, il en résulte tout de même un large éventail de distributions, impliquant des bibliothèques logicielles aux versions très différentes. Ces caractéristiques sont à prendre en compte lors du déploiement d'une application sur les ressources, notamment lorsque celle-ci utilise des bibliothèques particulières.

3.1.1.3 Gestion des données

Le stockage des données dans GRID'5000 est effectué à deux niveaux : au niveau des ressources physiques et au niveau de systèmes de fichiers distribués de type NFS [109]. Sur chacune des ressources physiques un disque local est mis à la disposition des utilisateurs. Ce disque est partitionné pour permettre notamment de stocker des données temporaires pendant un calcul. Cet espace est réinitialisé à chaque utilisation de la ressource. Il ne permet donc pas de conserver des données sur le long terme. Pour cela, un système NFS est proposé sur chaque site de la grille. Il permet de conserver des données entre deux utilisations

Listing 3.4 – Exemple de soumission d'une tâche à OAR.

```
paramount:~> oarsub -r "2008-09-03␣18:54:00"
                    -l nodes=4/cpu=2,walltime=2:30:00
                    /chemin/vers/mon-programme.sh
```

de la plate-forme. Le NFS d'un site est accessible à partir de toutes les ressources appartenant à ce site. Cette particularité permet notamment, lors du déploiement d'une application sur un même site, de ne pas transférer explicitement les programmes et les données sur les ressources. Il est en effet possible d'atteindre un fichier en utilisant son adresse unique dans l'arbre du système de fichiers. En revanche, l'espace NFS d'un site est difficilement accessible à partir de ressources localisées sur un site différent. Il faut pour cela connaître l'adresse d'une machine appartenant au site distant pour y effectuer une demande de transfert. De plus, les NFS de chaque site ne sont pas synchronisés. Les données stockées sur un site ne sont pas mises à jour sur les autres sites. La synchronisation des différents sites reste donc à la charge des utilisateurs. Enfin, les espaces NFS ne sont pas répliqués ou sauvegardés sur un support externe. Ils ne peuvent donc pas être utilisés comme du stockage fiable, mais seulement comme du stockage temporaire pour le code des applications en cours de développement ou pour les données résultant d'une expérience.

3.1.2 Réserver des nœuds avec OAR

Le système OAR [29] est un gestionnaire de ressources, aussi présenté comme un ordonnanceur de tâches, conçu pour les grandes grappes de calculateurs. Les utilisateurs peuvent soit soumettre des tâches - qui sont alors planifiées et démarrées sur les ressources sélectionnées - soit demander des ressources en particulier pour une certaine plage horaire. OAR est actuellement utilisé pour gérer les ressources mises à disposition par la plate-forme GRID'5000.

Interface utilisateur. Les interactions avec OAR s'effectuent grâce à différentes commandes indépendantes permettant de soumettre des tâches, d'annuler et de surveiller des réservations. Les trois principales commandes sont `oarsub` pour la soumission de tâches, `oardel` pour l'annulation et `oarstat` pour la surveillance de l'état courant des réservations. Ces commandes permettent de modifier la base de données de manière valide, c'est-à-dire en respectant son schéma et les contraintes associées. Dans la terminologie OAR, une tâche est définie par un programme et un ensemble de ressources requises pour son exécution. Lors de la soumission, il est possible de spécifier le programme à lancer, l'heure de démarrage, le temps total ainsi que des contraintes sur les ressources à sélectionner dans le site. Un exemple de soumission est donné dans le listing 3.4 dans lequel un utilisateur demande au site rennais l'exécution de son programme `mon-programme.sh` en utilisant les ressources d'une réservation de 4 nœuds bi-processeurs le 3 septembre 2008 à 18 heures 54 et pour une durée de 2 heures et 30 minutes. Le résultat d'une telle requête est soit positif avec un numéro de réservation, soit négatif avec la suggestion d'une nouvelle date de démarrage lorsque les ressources deviennent disponibles. Lorsque l'heure de démarrage est atteinte, OAR configure les ressources sélectionnées pour permettre l'accès à l'utilisateur et lance le

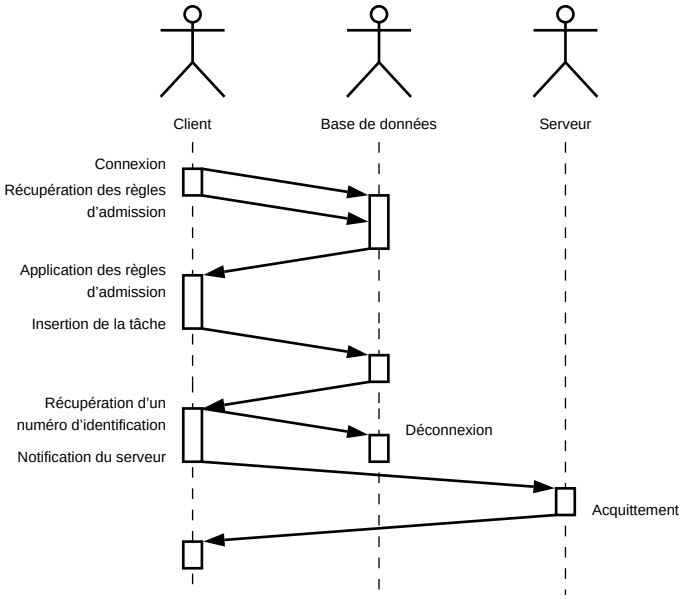

FIG. 3.5 – Diagramme de séquence des interactions entre le client, la base de données et le serveur OAR lors de la soumission d'une tâche.

programme spécifié grâce à l'outil de déploiement TAKTUK [89]. À la fin de la période de réservation, tous les processus de l'utilisateur sont terminés sur les ressources, qui sont alors libérées. Une variante consiste en l'utilisation de l'outil de déploiement d'images système KADEPLOY [62, 145] à la place de TAKTUK.

OAR offre aussi un mode dit de *best effort*. Les ressources inutilisées sont automatiquement allouées par l'ordonnanceur à des utilisateurs ayant soumis leur application dans ce mode. Il n'y a aucune garantie sur la disponibilité des ressources. OAR a un pouvoir de préemption sur les tâches s'exécutant en *best effort*. Elles peuvent être interrompues à tout moment pour laisser la place à une réservation classique. Le choix des tâches à stopper n'est pas effectué en accord avec l'application : si l'on considère une application distribuée sur le modèle du client-serveur, aucune information ne permet d'aider OAR à stopper des tâches de moindre importance comme les clients, avant de stopper le serveur.

Architecture du gestionnaire. Son originalité est d'être conçue à partir d'une base de données relationnelle MYSQL [152], chargée de stocker les informations sur les utilisateurs, les ressources et les réservations. Un ensemble de scripts PERL permet d'accéder aux données. OAR est entièrement spécifié par la sémantique des tables et des relations contenues dans le schéma de la base de données. Son exécution repose donc sur le moteur d'exécution de la base de données relationnelle. Un serveur OAR appelé *Central Module* est par ailleurs lancé pour chaque base de données et est responsable du bon déroulement de l'ordonnancement, de l'exécution ou encore de la surveillance des tâches. Le but d'une telle conception est de montrer qu'il est possible d'offrir des fonctionnalités et des performances équivalentes aux ordonnanceurs de tâches classiques tout en se basant sur des solutions logicielles de haut niveau. Son architecture modulaire en fait aussi un candidat privilégié pour l'expérimentation de nouvelles stratégies d'ordonnancement.

Interactions entre utilisateur, gestionnaire et ressources. La soumission d'une tâche entraîne des interactions entre le client ayant lancé la commande, la base de données et le serveur OAR, comme cela est montré sur la figure 3.5. Toute soumission requiert l'application de *règles d'admission* permettant de définir si la requête peut être honorée. Ces règles sont récupérées sur la base de données à chaque soumission. Dans leur forme la plus simple, elles permettent de filtrer les soumissions en empêchant plus de trois réservations dans le futur pour un même utilisateur ou encore des réservations considérées comme trop longues. Elles découlent directement de la charte d'utilisation de la plate-forme sur laquelle OAR est déployé et contribuent à une utilisation équitable des ressources entre les utilisateurs.

Pendant le temps d'une réservation, il est possible d'accéder aux ressources sélectionnées grâce à des méthodes classiques de connexion basées sur le SSH [2]. Cette connexion comprend l'utilisation d'un couple de clés publiques et privées, générées par OAR pour chaque réservation. La clé publique est transférée sur chaque ressource sélectionnée pour cette réservation alors que la clé privée est mise à disposition de l'utilisateur.

Vers un gestionnaire de niveau grille. Afin de coller au mieux à la topologie de la plate-forme GRID'5000, OAR a été déployé, de manière indépendante, sur chacun des sites de la grille. Chacune de ces instances a donc la charge des ressources mises à disposition par le site dont il fait partie et il n'est pas possible de réserver des ressources d'un autre site. Ces différentes instances d'OAR sont administrées et exécutées de manière indépendante. Cette approche permet d'éviter de tomber dans un schéma centralisé, avec un gestionnaire de ressources au niveau de la grille devant traiter les requêtes de tous les utilisateurs. Néanmoins, le caractère multi-site d'un nombre croissant d'expérimentations, pousse les utilisateurs à contacter chacune des instances d'OAR à la main pour effectuer du co-ordonnancement. Cela correspond à la réservation en parallèle de ressources situées sur des sites différents afin d'exécuter une même application. Le script OARGRID permet d'automatiser, de manière séquentielle, la suite des réservations à effectuer sur chacun des sites. Si l'un des sites ne peut satisfaire la requête, l'ensemble des réservations déjà effectuées sont annulées et la requête globale échoue. Cette approche reste éloignée des fonctionnalités offertes par des gestionnaires spécialisés dans le co-ordonnancement comme KOALA [93] ou DECO [2].

Si OAR permet de gérer de manière efficace les réservations à l'échelle de quelques grappes de calculateurs, son déploiement au sein du projet GRID'5000, sous forme d'une instance par site, reste inadapté dans le cadre d'expérimentations multi-site.

3.1.3 Déployer une application avec ADAGE

L'outil ADAGE [3] [81, 82, 132] a été initialement proposé dans le cadre du projet Padico [46] afin de prendre en charge toutes les considérations liées au déploiement d'applications sur grille de calculateurs. Son but est de rendre le déploiement aussi simple que possible, c'est-à-dire transparent pour les utilisateurs, tout en se révélant assez générique pour supporter des applications de types variés : applications parallèles, distribuées ou encore mixtes comme les composants parallèles. Cette généricité a permis à ADAGE d'être utilisé au sein de projets contemporains et de poursuivre son développement. Plusieurs types

[2] *Secure Shell*.

[3] *Automatic Deployment of Applications in a Grid Environment*.

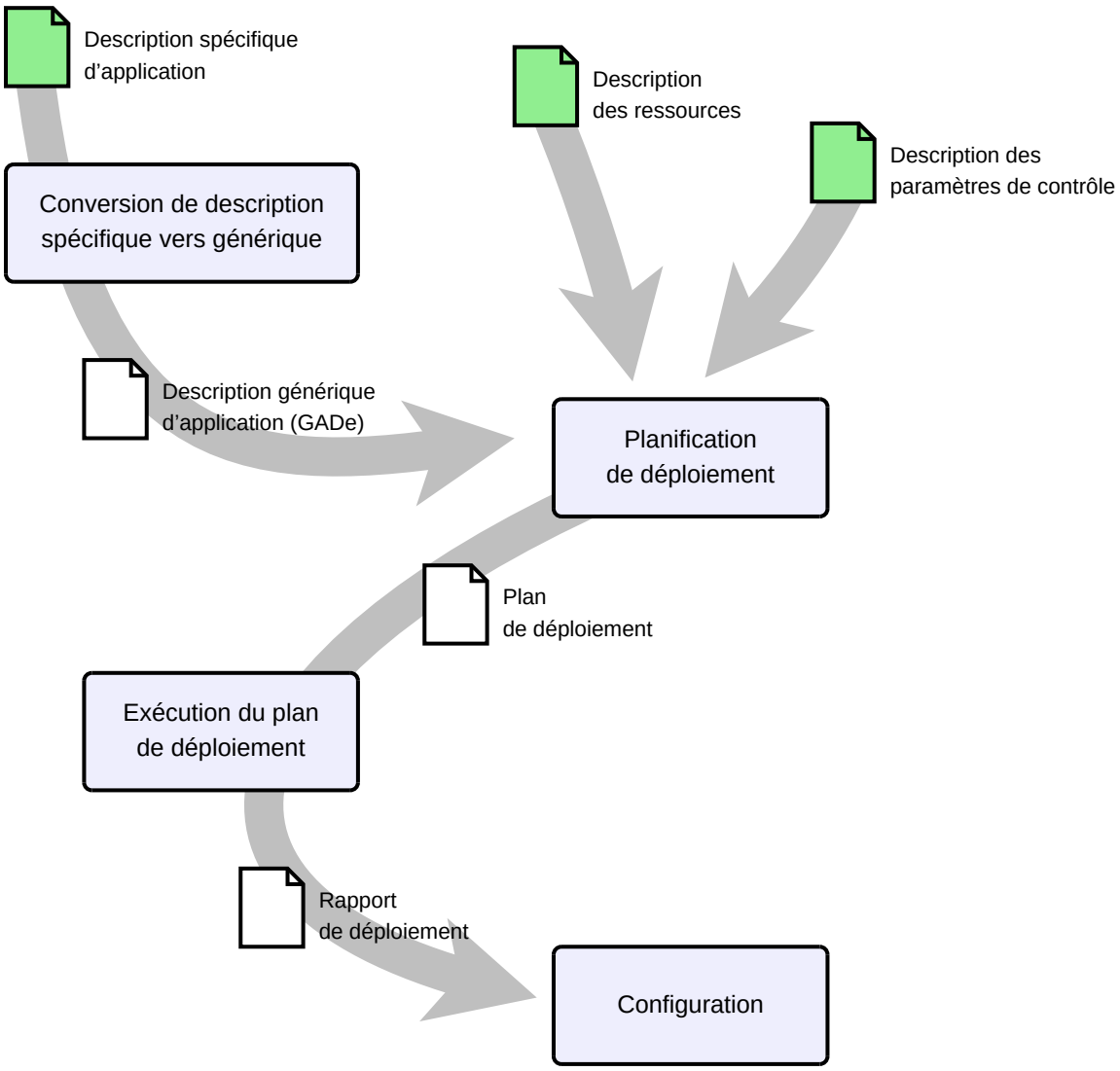

FIG. 3.6 – Architecture générale du déploiement automatique d'applications selon ADAGE.

Listing 3.7 – Exemple de description spécifique d'application pour ADAGE.

```
1 <!DOCTYPE HEURE_application>
2 <affiche-heure id="p1" cardinality="1"/>
```

Listing 3.8 – Exemple très simplifié de description générique d'application pour ADAGE.

```
1 <processes>
2   <process>
3     <id>p1</id>
4     <cardinality>1</cardinality>
5     <binary_location>/usr/bin/uptime</binary_location>
6   </process>
7 </processes>
```

d'applications et intergiciels sont d'ores et déjà supportés comme MPI [78], CORBA [101] , DIET [33], GFARM [116, 117], JUXMEM [10], JXTA [119] et autres environnements pair-à-pair. ADAGE a été largement utilisé pour la mise en place d'expérimentations à grande échelle sur la plate-forme GRID'5000. Ainsi il a été possible de déployer jusqu'à 29000 pairs (processus) JXTA sur 400 ressources [12] ou encore 4000 composants sur 1000 ressources réparties dans 7 sites de GRID'5000 [21]. D'autres déploiements ont été menés sur les grilles DAS-2 aux Pays-Bas et OMNIRPC au Japon. L'utilisation d'ADAGE ne requiert aucune modification des applications ni des outils de gestion des ressources de la grille. Une vue globale de l'architecture d'ADAGE est donnée dans la figure 3.6. Cette architecture montre le modèle en quatre phases, introduit par l'outil de déploiement :

Phase 1 : conversion d'une description spécifique au type d'application vers une description générique utilisée par le planificateur d'ADAGE. Un greffon spécifique est fourni pour chaque type d'application supporté.

Phase 2 : planification du déploiement : c'est-à-dire la sélection des ressources supportant l'exécution des différents constituants de l'application. C'est dans cette étape que se situe toute l'intelligence de l'outil. Plusieurs planificateurs sont disponibles comme le *round-robin* ou encore le *random*. Si un planificateur échoue, un second est sélectionné. Cette phase génère un plan de déploiement.

Phase 3 : exécution du plan de déploiement. Cette phase comprend le transfert des fichiers vers les ressources cibles, le lancement des processus de l'application à distance et la récupération d'un identifiant de tâche permettant de surveiller ou de terminer leur exécution. Cette phase génère un rapport de déploiement contenant notamment des informations sur le placement des processus et leur identifiant.

Phase 4 : configuration de l'application. Cette phase est de nouveau spécifique au type de l'application et est fournie par le greffon associé. Elle peut être utilisée par exemple pour inter-connecter des composants après leur démarrage. À ce niveau, il est possible d'utiliser les informations du rapport de déploiement créé précédemment.

L'utilisation de l'outil de déploiement nécessite de spécifier : - 1) une description spécifique de l'application, - 2) une description des ressources et - 3) une description des paramètres de contrôle. Ces informations sont utilisées afin d'effectuer les phases de planification, d'exécution du déploiement et de configuration des ressources. Elles sont transmises à l'outil soit directement en paramètre lors de son invocation, soit à l'aide d'un protocole de communication.

La description spécifique de l'application est une description de l'application à déployer, rédigée dans un formalisme qui est propre au type de l'application. Son but est de permettre la description de l'application à déployer de la manière la plus simple possible. Le format de cette description est libre de choix. Pour des raisons de facilité de traitement, le format conseillé est le XML. Il permet de donner un cadre d'écriture plus strict grâce à la spécification d'une grammaire au format XSD. Un exemple de description spécifique d'application est donné dans le listing 3.7. Dans cet exemple, une application de type `heure` est décrite avec comme information un élément de type `affiche-heure` dont l'identifiant est `p1` et la cardinalité[4] est `1`. Ces informations sont utilisées pour construire la description générique de l'application, écrite dans un langage,

[4]Le nombre d'instances de cet élément.

nommé GADE [5] [82] et indépendant du type de l'application. Cette description est obtenue à partir de la traduction de la description spécifique, grâce à un greffon qui est lui aussi spécialisé au type de l'application. Nous pouvons imaginer que l'exemple précédent génère la description générique proposée dans le listing 3.8. Ici, le greffon associé au type d'application `heure` traduit chaque élément `affiche-heure` en un processus au sens ADAGE dont le programme à exécuter est `uptime`.

La description des ressources définit l'ensemble des ressources pouvant être utilisées lors de la phase de planification pour supporter l'exécution des processus. La description proposée par ADAGE permet d'exprimer de nombreuses caractéristiques sur les ressources. Elles portent sur le matériel (processeurs, mémoire, disques, réseaux) et sur le logiciel (système d'exploitation, bibliothèques). Les ressources sont décrites dans un format spécifique à ADAGE, mettant en valeur les liens réseaux existant entre elles. Elles sont regroupées par appartenance à un même sous-réseau. Cette approche permet notamment le passage à l'échelle en simplifiant le travail du planificateur et en factorisant l'information. La représentation de ces ressources est effectuée à l'aide d'une description au format XML qu'il est possible de spécifier dans les paramètres de l'outil. Cette description ne doit contenir que des ressources réservées, il est donc nécessaire de la modifier en conséquence. Afin de faciliter le travail de l'utilisateur, il est possible d'indiquer des numéros de réservation. La récupération des ressources est à la charge d'ADAGE qui va s'interfacer avec le système d'informations de la grille. Il est possible, dans l'implémentation actuelle, de fournir des numéros de réservation OAR et OARGRID. La prise en charge des interactions avec le MDS Globus [56] et les intergiciels du projet Unicore [37] est aussi prévue bien que non-implémentée.

La description des paramètres de contrôle expriment des contraintes fournies par l'utilisateur sur une exécution particulière de l'application. Ces contraintes peuvent donner des informations sur la qualité de service désirée pour ce déploiement. Mais aussi sur les caractéristiques des ressources pouvant être utilisées ou encore privilégier un algorithme de planification plutôt qu'un autre. Les deux informations qui vont nous intéresser plus particulièrement dans le cadre de ces travaux sont les *contraintes de placement* et les *contraintes temporelles*. Les contraintes de placement permettent d'indiquer si deux constituants de l'application doivent être déployés ou non, sur une même ressource physique. Les contraintes temporelles indiquent que le lancement d'un processus ne doit s'effectuer qu'après la réception du message d'acquittement d'un autre processus.

Le support d'un type d'application ou d'intergiciel dans ADAGE nécessite la définition d'une grammaire régissant le format de la description spécifique. Cette grammaire doit tenir compte des particularités de l'application et doit être la plus simple possible pour l'utilisateur final. Malgré cela, l'écriture de la description spécifique d'une application reste une tâche rédhibitoire, bien souvent verbeuse et répétitive, en particulier lors du déploiement d'applications distribuées à grande échelle. Le support de l'application s'accompagne aussi du développement d'un greffon spécialisé dans la traduction de la description spécifique vers la description générique. Une fonction de conversion doit être écrite ; Elle consiste principalement en la transformation d'un arbre XML spécifique en un second arbre XML générique correspondant à la description GADE. L'écriture de cette fonction de conversion

[5] *Generic Application Description*.

nécessite la pleine compréhension du langage GADE, sous peine de ne pas obtenir un résultat cohérent lors de la phase de planification. Le greffon propose par ailleurs des méthodes de configuration lancées par ADAGE après l'exécution du plan de déploiement.

Discussion. L'outil de déploiement ADAGE permet, dans son implémentation actuelle, de prendre en charge la plupart des déploiements pour l'expérimentation et la production à petite, moyenne et grande échelle. Son efficacité, sa robustesse et sa spécialisation dans un nombre croissant d'applications en font l'outil idéal pour les déploiements automatisés complexes. ADAGE reste cependant un outil dédié au déploiement statique des applications. c'est-à-dire que le support pour le suivi de l'exécution ainsi que pour effectuer des opérations de redéploiement et de rétraction est encore en phase de validation. De plus, l'écriture d'une description spécifique d'application reste une tâche pénible pour les utilisateurs. Dans le contexte des applications dites autonomes, les interactions avec ADAGE ne devraient pas être du ressort de l'utilisateur : l'application déployée devrait elle même, par le biais d'un service de gestion de l'autonomie, être capable de spécifier ses besoins, sous forme d'une action de haut niveau d'abstraction ou d'une description spécifique.

3.2 Étude de cas : un service de partage de données pour la grille

Dans cette section nous effectuons une étude de cas autour d'un service de partage de données pour la grille nommé JUXMEM [10, 73, 96]. JUXMEM signifie *Juxtaposed Memory*, en référence à la plate-forme pair-à-pair JXTA [119] (*Juxtaposed* en anglais) sur laquelle il reposait à l'origine. Le déploiement de JUXMEM requiert une certaine expertise sur le fonctionnement des différents composants du système. C'est donc un excellent exemple d'application nécessitant une aide pour le déploiement à grande échelle. Nous présentons donc dans un premier temps JUXMEM afin de bien appréhender les spécificités liées à son déploiement et mettre par la suite en valeur les limitations de l'approche actuelle.

Le but d'un service de partage de données est d'offrir des accès efficaces et transparents à des données partagées stockées dans les mémoires physiques de plusieurs ordinateurs. Ce modèle d'accès aux données a été largement étudié au sein des mémoires virtuellement partagées [85] (*Distributed Shared Memory*, DSM *Systems* en anglais) il y a plus de trente ans maintenant. Cependant, de tels systèmes ont été conçus pour s'exécuter sur des grappes de calculateurs, à des échelles de l'ordre de quelques dizaines de nœuds. Avec l'arrivée d'infrastructures d'exécution fédérant plusieurs milliers de ressources, les DSM ont montré leurs limites en terme de passage à l'échelle. De nouvelles approches ont dû être explorées. C'est dans ce contexte que le concept de service de partage de données pour grilles de calculateurs a été proposé. Il s'inspire de plusieurs approches.

- Les systèmes à mémoire virtuellement partagée offrent des accès transparents aux données ainsi que des modèles et protocoles pour la gestion de la cohérence.
- Les systèmes distribués proposent des algorithmes de tolérance aux défaillances.
- Les systèmes pair-à-pair (*peer-to-peer*, *P2P* systems en anglais) sont reconnus pour leur capacité à passer à l'échelle et à supporter la volatilité des ressources.

D'un point de vue utilisateur, le modèle utilisé par JUXMEM s'oppose au modèle d'accès explicite aux données, car il masque à l'utilisateur toute la gestion de la localisation et du

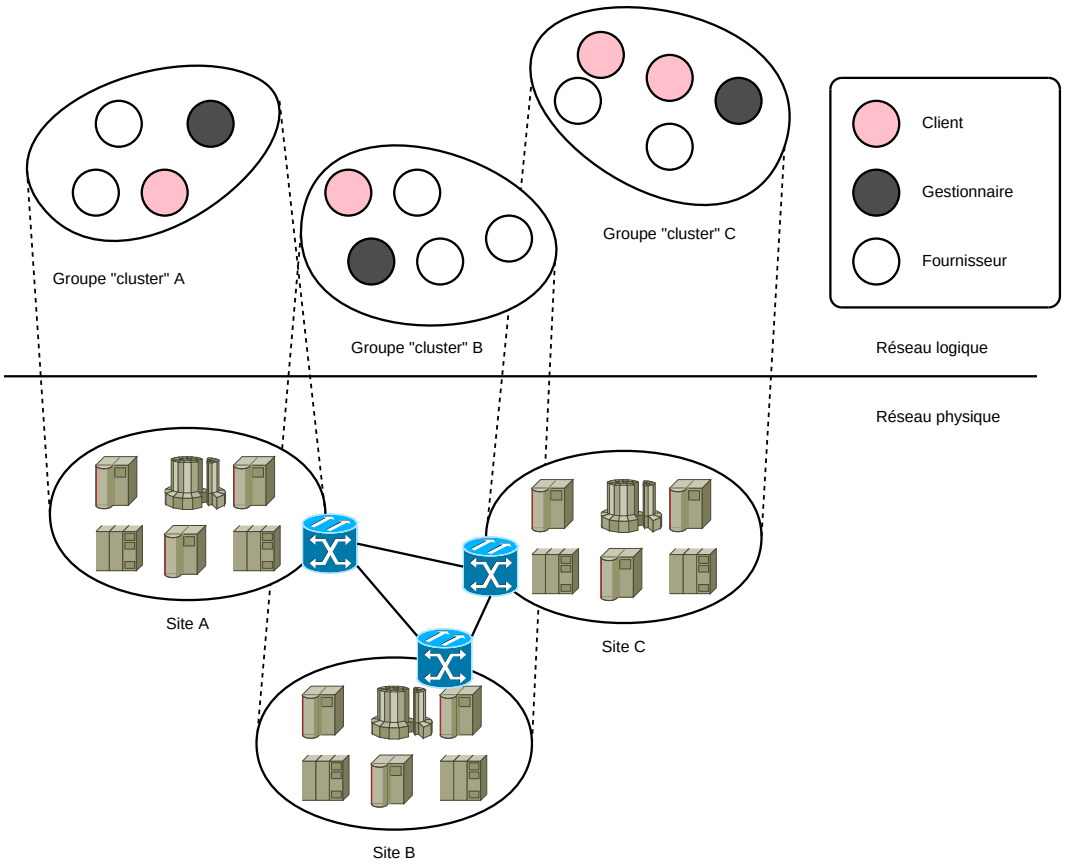

FIG. 3.9 – Projection de l'architecture logique de JUXMEM sur le réseau physique.

transfert des données. L'interface de programmation est inspirée des DSM et permet d'effectuer des requêtes d'allocation mémoire, d'accéder à une donnée grâce à un identifiant global unique et de synchroniser les accès des différents clients grâce à un système de verrous. L'utilisateur spécifie, pour chaque requête d'allocation, le nombre de copies de la donnée souhaité. La requête d'allocation retourne ensuite un identifiant qui peut être utilisé par les autres nœuds afin d'accéder à la donnée.

3.2.1 Architecture de JUXMEM

L'une des différences principale avec les DSM est que l'architecture de JUXMEM est conçue à l'image de l'architecture des grilles, c'est-à-dire que les entités sont organisées au sein de groupes hiérarchiques, comme cela est montré en figure 3.9. Cette architecture a été mise en place pour prendre en compte les spécificités en terme de connexion réseau de la plate-forme d'exécution et permettre un passage à l'échelle dans de bonnes conditions. Il existe trois types d'entités dans le service.

Le client utilise le service pour effectuer des accès aux données partagées. Cette entité peut être directement définie par un code applicatif ou, dans des configurations plus élaborées, faire partie d'une autre application.

Le fournisseur met à disposition du service de l'espace de stockage. La capacité totale de stockage est calculée en cumulant les capacités de tous les fournisseurs déployés.

Le gestionnaire met en place les liens entre les entités. Il organise structurellement le réseau logique en jouant le rôle de point de rendez-vous pour les nouvelles entités. Il aiguille les messages de publication d'espace disponible et de recherche d'espace de stockage.

Un scénario simple mettant en jeu ces trois entités est le suivant : le gestionnaire est déployé en premier. Le fournisseur est à son tour déployé et s'enregistre auprès du gestionnaire. Il indique par ailleurs qu'il met à disposition un espace de stockage d'une certaine taille. Un client est alors déployé, il s'enregistre auprès du gestionnaire et effectue une requête d'allocation dans le service. Cette requête est reçue par le gestionnaire qui met en relation le client et le fournisseur.

Les entités JUXMEM sont organisées sous forme de groupes hiérarchiques. Le groupe principal est appelé le *groupe JuxMem* et contient toutes les entités déployées. Ce groupe est décomposé en sous-groupes appelés *groupes cluster*. Lors d'une requête d'allocation, le client spécifie dans combien de *groupes cluster* et sur combien de fournisseurs la donnée doit être répliquée. Chaque *groupe cluster* contient au plus une entité gestionnaire en charge de son organisation. Les clients et les fournisseurs sont tous attachés à un unique *groupe cluster*. Le choix du groupe est laissé à la charge de l'utilisateur mais il doit s'effectuer en fonction des affinités entre entités en terme d'accès aux données partagées. En effet, les *groupes cluster* logiques sont conçus afin d'être déployés sur des sites physiques différents. Cette organisation a été choisie pour tenir compte des différences de performances réseau, principalement la latence que l'on peut observer entre les communications intra-sites et inter-sites. Ainsi, les protocoles utilisés dans JUXMEM pour gérer la synchronisation des accès ou la cohérence des données favorisent les échanges de messages entre les entités situées dans le même *groupe cluster* et donc dans le même site physique. Cela montre l'importance du choix du groupe pour chaque entité.

Il existe deux implémentations de JUXMEM, basées sur des technologies différentes de gestion du réseau logique. La première implémentation, aujourd'hui gelée, est basée sur la plate-forme *P2P* JXTA. Elle a permis d'effectuer la validation de l'approche JUXMEM en profitant des algorithmes de publication et de découverte des ressources de JXTA. Des tests détaillés ont été effectués avec cette version pour mesurer le coût des communications, l'efficacité des algorithmes de tolérance aux défaillances et l'efficacité des protocoles de cohérence des données. C'est aussi avec cette version basée sur JXTA que des interactions avec le système de fichiers global GFARM [116] ont été mises en place. La seconde implémentation se base sur une couche de communication générique et indépendante du moyen de transport. Cette implémentation diverge de la première, d'un point de vue conceptuel, en ne définissant qu'un seul *gestionnaire* pour chaque déploiement de l'application JUXMEM et non plus un *gestionnaire* par *groupe cluster* comme cela est utilisé dans la version basée sur JXTA.

Par la suite, nous nous intéressons à la problématique du déploiement de ces deux versions de JUXMEM. Afin de prendre en charge la première version du service de partage de données, nous introduisons la technologie pair-à-pair JXTA avec une approche orientée organisation logique, configuration et déploiement.

3.2.2 Héritage pair-à-pair avec la plate-forme JXTA

La spécification JXTA, développée par Sun Microsystems depuis 2001, permet l'élaboration de systèmes distribués sur le modèle pair-à-pair. L'acronyme JXTA signifie *juxtaposed* (juxtaposés) et fait référence aux complémentarités que l'on peut trouver entre les modèles pair-à-pair et client-serveur. JXTA est une *spécification*, elle propose des concepts pour la construction des applications *P2P*. Ces concepts consistent en des spécifications de protocoles et de services devant être mis en place pour permettre l'interconnexion des pairs ou encore le format des messages échangés. Aucune implémentation, ni même d'algorithme n'est fixé par cette spécification. De ce fait, JXTA n'est pas dépendant d'une technologie ou d'un matériel en particulier. Plusieurs implémentations se référant à la spécification sont proposées pour mettre en œuvre JXTA sur diverses plates-formes d'exécution, de l'ordinateur classique à l'assistant personnel. Parmi les plus connues, on trouve la version Java, historiquement la première à être proposée et la version *C*. Toutes deux sont utilisées comme support pour différentes implémentations de JUXMEM.

La plate-forme JXTA permet de mettre en place un *réseau logique* composé de *pairs*. Ce réseau logique autorise le routage des messages, de manière transparente, par adressage du pair de destination. Un identifiant unique est calculé pour chaque nouveau pair désirant joindre le réseau. Il permet aussi de publier et de rechercher des informations. Ces fonctionnalités sont utilisées par JUXMEM pour mettre en place son propre réseau logique, interconnecter les différentes entités, publier les annonces d'espace de stockage et les demandes d'allocation de données.

Les implémentations de la spécification JXTA considérées par la suite introduisent trois principaux types de pairs.

Le pair standard (*edge peer*) peut émettre et traiter des requêtes JXTA. ceci constitue le service minimal devant être offert par tout pair du réseau.

Le pair de rendez-vous (*rendez-vous peer*) ajoute au pair standard la possibilité d'aiguiller les requêtes. Pour cela, il participe à l'indexation des ressources présentes dans le réseau. Chaque pair de rendez-vous connaît une vue locale de ces ressources.

Le pair de relais (*relay peer*) peut faire office de boîte aux lettres en stockant temporairement des messages à destination d'autres pairs. Cette fonctionnalité est utilisée pour contourner des obstacles en terme de connectivité comme un pare-feu ou un système de traduction d'adresses (NAT).

La composition de ces trois types de pairs permet de constituer un réseau logique *P2P semi-structuré*. En effet, certains pairs comme les pairs de rendez-vous, jouent le rôle de *super pairs*, établissant des liens privilégiés dans le réseau.

Lors du déploiement d'un réseau logique JXTA, la spécialisation des pairs est effectuée selon un fichier de configuration appelé `PlatformConfig` lu pendant la phase d'initialisation des services. Ce fichier de configuration doit être fourni pour chaque pair. Il contient principalement l'identifiant du pair concerné, le port de communication sur lequel écouter et les adresses physiques des ressources supportant l'exécution d'un ou plusieurs pairs de rendez-vous. Ces pairs de rendez-vous particuliers sont appelés *seeds* en anglais, à l'image de graines permettant de faire *germer* le réseau. Le fichier de configuration contient par ailleurs de nombreuses informations sur le comportement, le type de protocole de transport à utiliser ou

encore l'appartenance à un groupe de pairs. Ce fichier de configuration doit être généré et personnalisé pour chaque pair voulant joindre le réseau JXTA.

Le prototype JUXMEM, basé sur la couche de communication JXTA, utilise les pairs standard pour supporter les entités de type client et de type fournisseur. Ces entités ne nécessitent en effet qu'une connectivité au réseau logique et ne font que demander, accéder ou mettre à disposition de l'espace de stockage. Des pairs de rendez-vous sont nécessaires pour aiguiller les requêtes et annonces. Il sont utilisés pour supporter l'exécution des gestionnaires JUXMEM, à raison d'un pair par *groupe cluster*. Cette organisation forme un super-réseau de pairs de rendez-vous gestionnaires, un gestionnaire par site physique, sur lesquels sont attachés des pairs standards clients et fournisseurs au sein de chaque site.

3.2.3 Déploiement de JUXMEM avec ADAGE

La problématique du déploiement de JUXMEM s'est posée dès les premières évaluations expérimentales du service. À l'origine, le service repose sur les implémentations Java et C de la plate-forme pair-à-pair JXTA. Les solutions pour déployer JUXMEM se résument à l'utilisation de scripts spécifiques avec, dans leur version la plus élaborée, le système JDF [11] (pour JXTA *Distributed Framework*, outil présenté dans la section 2.3.1.4). Outre la difficulté d'utilisation reconnue par quelques malheureux aventuriers, ce système a montré de nombreuses limites, notamment en terme d'expressivité du langage de description de l'application, mais aussi lors de son utilisation dans des configurations à grande échelle. Un langage de description des applications basées sur JXTA, appelé JDL (pour JXTA *Description Language*) a alors été proposé dans [11, 73]. Il s'est accompagné de la mise en place d'un greffon[6] pour ADAGE permettant de prendre en charge le déploiement de JXTA. Avec l'arrivée de la nouvelle version de JUXMEM basée sur des communications génériques, un nouveau langage de description ainsi qu'un nouveau greffon ADAGE ont été conçus. Dans une première sous-section nous présentons le fonctionnement et les améliorations apportées à la gestion du déploiement des applications basées sur JXTA avec ADAGE. Puis nous montrons comment la nouvelle version de JUXMEM a été prise en charge dans ADAGE.

3.2.3.1 Prise en charge de la version de JUXMEM basée sur JXTA

Le problème du déploiement de la première version de JUXMEM s'apparente au problème du déploiement d'un réseau *P2P* classique. Les travaux réalisés pour effectuer de tels déploiements ne concernent d'ailleurs pas JUXMEM en particulier mais bien l'ensemble des applications basées sur JXTA. Ainsi, le langage de description JDL ne contient aucune notion spécifique à JUXMEM mais décrit, de manière générique, un réseau JXTA.

Un format de description pour les applications basées sur JXTA. La description est effectuée en deux parties, comme cela est montré dans le listing 3.10. Une première partie présente différents `profils` pouvant être instanciés en pairs. Ces profils sont identifiés par un nom et contiennent les informations importantes devant être placées dans le fichier de

[6] Un greffon (*plugin* en anglais) est un bout de code additionnel utilisé pour étendre les fonctionnalités de l'application qui le reçoit.

Listing 3.10 – Exemple de description d'application JXTA, au format JDL.

```
<JXTA_application>

  <profiles>
    <profile name="p_rdv">
      <services rdv="true" relay="false" />
      <behavior binding="c" filename="gestionnaire" args="1800" />
    </profile>
    <profile name="p_edge">
      <services rdv="false" relay="false" />
      <behavior binding="c" filename="fournisseur" args="10000␣1800" />
    </profile>

    <overlay>
      <rdvs>
        <rdv id="r1" profile_name="p_rdv" cardinality="1" />
        <rdv id="r2" profile_name="p_rdv" cardinality="1">
          <rpv>r1</rpv>
        </rdv>
      </rdvs>
      <edges>
        <edge id="e1" profile_name="p_edge" cardinality="3" rdv="r2" />
      </edges>
    </overlay>

</JXTA_application>
```

configuration `PlatformConfig` présenté en section 3.2.2. Parmi ces informations nous retenons particulièrement le champ `rdv` indiquant si le profil définit un pair de rendez-vous ou un pair standard (*edge*) ainsi que les champs `filename` et `args` indiquant respectivement le programme exécutable et les arguments associés.

Dans une deuxième partie, le format JDL décrit la topologie du réseau logique JXTA devant être déployé. Cette topologie est composée d'une partie dédiée aux pairs de rendez-vous ainsi que d'une partie dédiée aux pairs standards. Chaque pair est caractérisé par un identifiant interne (qui n'est pas l'identifiant JXTA), un type de profil et une cardinalité. La cardinalité indique le nombre d'instances à créer pour ce pair. Il est aussi possible de faire référence à un autre pair dans la description : celui-ci sera utilisé comme graine (*seed*) pour se connecter au réseau logique.

Un greffon pour traduire la description des applications. Le greffon JXTA pour ADAGE joue différents rôles, de l'interprétation du langage JDL à la gestion dynamique de la configuration des entités et ce, pendant toute la durée de la phase de déploiement. Une vue d'ensemble est donnée par le diagramme de séquence proposé en figure 3.11. Point de départ du déploiement, le fichier de description spécifique écrit au format JDL est traduit, à la demande d'ADAGE, par le greffon JXTA dans le format générique GADE. L'idée est de créer, pour chaque pair déclaré dans le fichier JDL, un processus au sens ADAGE dans la description GADE.

Des informations sont renseignées pour chaque pair avec, par exemple, les contraintes temporelles liant les différents pairs entre eux. Ces contraintes sont exprimées de manière implicite dans la description JDL : ADAGE s'assure que les pairs de rendez-vous soient démarrés *avant* que les pairs qui les utilisent comme des graines soient déployés. Nous verrons dans le paragraphe suivant qu'ADAGE s'assure que les pairs de rendez-vous soient démarrés et *actifs*, c'est-à-dire prêts à recevoir des messages. Une autre information importante est l'identifiant du pair. Cet identifiant est calculé par le greffon lors de la phase de traduction. Comme cette étape est centralisée, il est possible de garantir l'unicité de tous les identifiants. Enfin, le port de communication sur lequel chaque pair doit écouter est soit récupéré à partir de la description spécifique, soit établi avec une valeur par défaut. L'ensemble de ces informations forme la *configuration statique* de l'application.

Une configuration dynamique des pairs. La description générique est ensuite utilisée par ADAGE pour préparer le plan de déploiement. Le plan de déploiement associe pour chaque entité une ressource physique particulière. Ce plan est exécuté en s'assurant que tous les pairs soient déployés en respectant les contraintes temporelles calculées auparavant. L'exécution comprend une première phase d'interrogation du greffon afin de récupérer les informations de configuration statique. Un *script de lancement* est alors exécuté sur la ressource physique choisie pour ce pair. Ce script a la charge de générer le fichier de configuration *PlatformConfig* du pair à partir des informations de configuration statique. Selon l'algorithme de planification sélectionné pour ce déploiement, plusieurs entités peuvent être placées sur une même ressource physique.

Dans ces conditions, le choix du port de communication, effectué de manière statique, peut se révéler problématique. En effet, pour diverses raisons, le port en question peut ne pas être libre sur cette ressource physique. La plupart du temps cela arrive lorsque toutes les

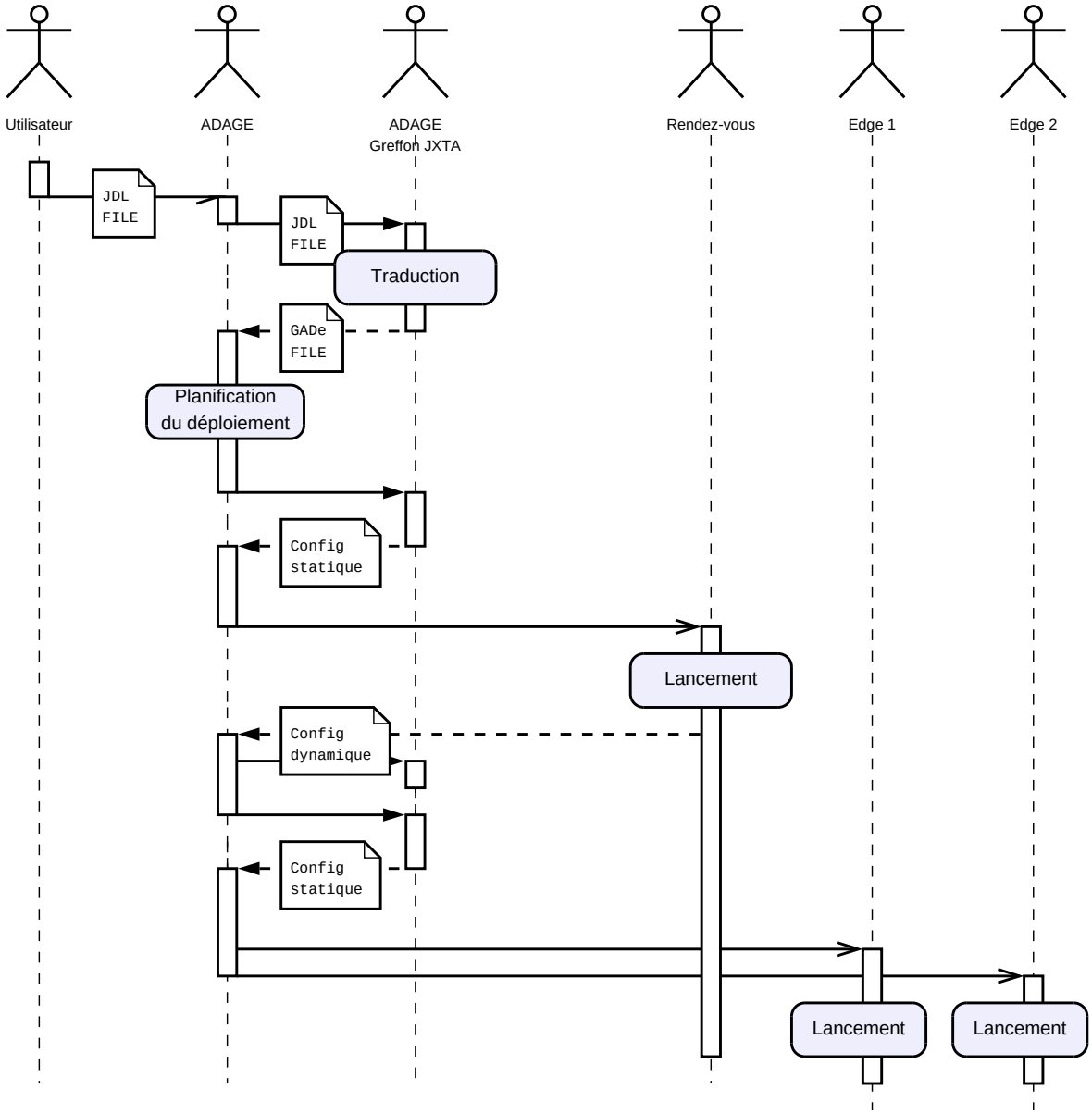

FIG. 3.11 – Diagramme de séquence du déploiement d'un réseau logique JXTA avec ADAGE.

entités utilisent la valeur par défaut et qu'une entité est déjà lancée sur la ressource physique. Le script de lancement effectue donc une vérification de disponibilité du port et si celui-ci n'est pas disponible, il en choisit un nouveau. Ce choix est effectué en additionnant la valeur du port courant avec un nombre tiré aléatoirement. Pendant cette phase, un verrou est pris sur la ressource afin d'empêcher une autre entité d'effectuer la même opération en même temps, ce qui pourrait entraîner le choix du même port. Le nouveau port de communication ainsi choisi de manière dynamique est retourné à l'outil de déploiement grâce à une fonction de rappel (*callback* en anglais). Cette fonction met à jour les informations de configuration statique, utilisées par la suite pour le déploiement des pairs suivants.

Une synchronisation du déploiement des pairs. La synchronisation du déploiement des différents pairs est un point important dans le déploiement d'une application comportant des contraintes temporelles. Dans le contexte de JXTA, il est primordial qu'un pair de rendez-vous soit lancé et prêt à recevoir des communications entrantes avant de démarrer d'autres pairs qui doivent s'y connecter. Cette synchronisation est effectuée grâce à une fonctionnalité particulière offerte par ADAGE : il est possible d'endormir le script de lancement juste après avoir démarré l'exécution du pair de rendez-vous. Le script endormi ne retourne donc pas à ADAGE de message indiquant que le pair est démarré. Le script est endormi jusqu'à ce qu'un signal soit envoyé par le pair. Ce signal consiste, dans l'implémentation actuelle, en la création d'un fichier. Cette méthode demande cependant la modification du code de JXTA ou de l'application basée sur JXTA afin de déclencher l'envoi du signal. Dans le contexte de JUXMEM, le code de création du fichier est inséré dans JUXMEM lorsque les services JXTA ont été initialisés avec succès.

Contribution. Les travaux préliminaires effectués consistent en la réécriture complète du greffon JXTA afin de prendre en compte la nouvelle version de l'interface de programmation offerte par ADAGE. Le nouveau greffon prend aussi en compte les nouvelles versions de JXTA, caractérisées par des fichiers de configuration légèrement différents. Cette nouvelle version a aussi permis de mettre en place toute la partie dynamique de la configuration avec, notamment, le choix du port de communication résolu directement sur la ressource. Enfin, les contraintes temporelles sont désormais prises en compte grâce à l'utilisation du fichier de synchronisation. L'ensemble de ces modifications représentent environ 800 lignes de code écrites en langage C++ et SHELL. Le greffon JXTA a de plus servi de base pour la création d'un greffon générique dédié au déploiement de plusieurs systèmes *P2P* comme CHIMERA [133], KHASHMIR [148] et I3/CHORD [112].

Suite à ce travail, il est possible de déployer un service JUXMEM basé sur la couche de communication JXTA. Cependant, pour un utilisateur en charge du déploiement, cette approche requiert une connaissance des notions introduites par JXTA. Il est en effet nécessaire de comprendre le rôle des différents types de pairs JXTA. Ceci ne contribue pas à rendre le déploiement plus facile. Par ailleurs, les options de configuration propres à chaque entité JUXMEM, comme le temps total d'exécution, la taille de la mémoire mise à disposition ou tout autre paramètre utilisé en entrée du programme, doivent être spécifiés dans le champ `args` des profils de la description JDL. Il en résulte une description peu lisible et mélangeant des paramètres propres à JUXMEM et des paramètres propres au programme utilisant JUXMEM.

Listing 3.12 – Exemple de description d'application JUXMEM.

```
<JUXMEM_application>

  <juxmem_cluster id="groupe1">
    <managers>
      <manager id="gestionnaire" port="9876" uptime="60" />
    </managers>
    <providers />
    <clients />
  </juxmem_cluster>

  <juxmem_cluster id="groupe2" rdv="groupe1">
    <managers/>
    <providers>
      <provider id="fournisseur" port="9876" uptime="60" mem="10000" card="3" />
    </providers>
    <clients>
      <client id="ecrivain" port="9871" bin="juxmem-writer"
              args="@managerhostname␣4" card="1" />
    </clients>
  </juxmem_cluster>

</JUXMEM_application>
```

3.2.3.2 Prise en charge de la nouvelle version de JUXMEM : avancées et limitations

Avec l'arrivée de la nouvelle version de JUXMEM basée sur une couche de communication générique, il a été possible de proposer un greffon plus spécifique et ainsi de se concentrer sur les points importants du service. Le déploiement de cette version reprend les grandes lignes de l'ancienne approche, à savoir une description spécifique à l'application et un greffon en charge de la traduction de la description et de la gestion des informations de configuration. Une nouvelle description spécifique a été proposée, reprenant la topologie caractéristique de JUXMEM et visant à simplifier l'interface utilisateur. Un exemple de description est donné dans le listing 3.12, il décrit la même topologie que la description JDL du listing 3.10.

La description est organisée autour des *groupes cluster*. Chaque groupe peut contenir des gestionnaires, fournisseurs et clients. Dans la version actuelle, le service JUXMEM ne contient qu'un seul gestionnaire pour l'ensemble du réseau déployé. Dans cet exemple, le gestionnaire est placé dans le `groupe1`. Un deuxième *cluster* JUXMEM est défini comme le `groupe2`. Son point de rendez-vous est le `groupe1`, ce qui signifie que toutes les entités de ce groupe doivent se déclarer auprès du gestionnaire du `groupe1`. Cette description de l'application permet de bien séparer les informations de configuration propres à JUXMEM et les informations spécifiques à ce déploiement. L'exemple le plus clair se situe au niveau du client du `groupe2`. Tous les champs, à l'exception de `args`, contiennent des informations propres à la technologie JUXMEM. Le champ `args` est mixte, c'est-à-dire qu'il contient des paramètres spécifiques à ce déploiement, comme la valeur `4`, mais aussi des paramètres propres à JUXMEM, comme `@managerhostname`. Ces paramètres un peu particuliers sont identifiés grâce au caractère `@` qui

les précède. Ils font référence à des informations de configuration dynamique et sont instanciés directement sur la ressource physique au moment du déploiement. Dans cet exemple, `@managerhostname` sera remplacé par le nom de la ressource physique supportant l'exécution du gestionnaire. Une telle information ne peut en effet pas être connue lors de l'écriture de cette description.

Le greffon spécialisé dans le support de cette nouvelle version de JUXMEM offre les fonctionnalités introduites dans la version pour JXTA : la résolution dynamique des ports de communication et la gestion de la synchronisation du déploiement des entités. À ces fonctionnalités s'ajoute la possibilité de spécifier de l'information de configuration dynamique directement dans la description de l'application. La description a été modifiée pour prendre en compte la spécificité topologique de JUXMEM et se focaliser uniquement sur la technologie utilisée. Cela contribue à la clarté de la description et facilite leur utilisation.

Limitations. Malgré ces efforts, le greffon conçu pour JUXMEM ne résout par encore les deux problèmes suivants.

- L'écriture d'une description d'application reste toujours une tâche pénible pour l'utilisateur.
- Aucune solution n'est prévue pour rendre la gestion du déploiement dynamique pendant l'exécution de l'application.

Ces limitations sont à l'origine des travaux de recherche présentés dans le cadre de cette thèse.

3.3 Conclusion

Le projet GRID'5000 permet de mettre en valeur toute la problématique du déploiement d'applications sur les grilles de calculateurs. Bien souvent négligé par les utilisateurs de la plate-forme, le déploiement est pourtant un élément clé dans la réussite d'une expérimentation et qui plus est à grande échelle. L'approche statique est la plus couramment utilisée, notamment avec des outils développés de manière spécifique au type de l'application. Dans ce contexte, ADAGE se révèle être, à notre connaissance, l'outil le plus avancé dans le domaine du déploiement générique d'applications. En plus du simple déploiement statique, ADAGE permet déjà d'effectuer des co-déploiements ainsi que des redéploiements. Il pose donc les bases nécessaires à la gestion dynamique des applications. Des applications distribuées comme le service de partage de données JUXMEM, la plate-forme pair-à-pair JXTA ou encore le système de fichiers global GFARM peuvent tirer parti de telles fonctionnalités.

La prise en charge de ces applications par ADAGE, de manière dynamique, implique de nouvelles interactions entre l'application et l'outil de déploiement. Ces interactions doivent permettre d'effectuer des modifications topologiques pendant le temps d'exécution. Cette approche rend l'application *dépendante* de la technologie introduite par ADAGE. Elle doit exprimer ses besoins en terme de description d'application, de paramètres de contrôle ou de ressources physiques. Ce constat met en lumière la nécessité de simplifier drastiquement ces interactions et de permettre aux applications d'exprimer leurs besoins en terme d'*actions de haut niveau*. Ces actions, spécifiques au type de l'application, doivent ensuite être traduites en une liste d'*opérations de bas niveau* utilisées pour le choix des ressources et le déploiement de composantes supplémentaires.

Chapitre 4

Contribution : CoRDAGe, un service de co-déploiement et redéploiement

Sommaire

Avec la complexification des applications distribuées et des infrastructures d'exécution, les outils d'accompagnement ou de gestion du déploiement deviennent une aide précieuse pour les utilisateurs en charge de la maintenance ou des expérimentations. Bien souvent limités à une aide ponctuelle pour l'utilisateur, ces outils ne traitent pas des aspects dynamiques de l'exécution de l'application. Entre outils de bas niveau et environnements lourds pour applications autonomes, nous proposons CoRDAGe [9], un outil de gestion du déploiement dynamique ayant la charge d'interfacer les applications avec les outils bas niveau

de réservation et de déploiement. Cet outil permet de décharger l'utilisateur de toutes les interactions avec les outils bas niveau de la grille nécessaires à la vie de son application. Dans ce chapitre nous introduisons les besoins des utilisateurs sous forme de scénarios puis nous proposons un modèle orienté déploiement permettant de représenter les applications et les ressources. Enfin, nous montrons comment utiliser ce modèle pour effectuer une pré-planification du déploiement et suivre le comportement de l'application, de manière dynamique, tout au long de son exécution.

4.1 Vers une gestion plus transparente des interactions entre l'application et les ressources

Pour des applications scientifiques de conception participative ou encore les applications de simulation, les besoins en nombre de ressources physiques nécessaires au support de l'exécution sont considérés comme variables et non prévisibles. C'est notamment le cas pour un ensemble d'applications conçues pour les grilles de calculateurs et reposant sur des services tels que le stockage persistant de données partagées (JUXMEM [10], GFARM [116]) ou le support aux calculs distribués (DIET [33], zorilla [49]). Le déploiement initial de ces services s'avère problématique. Il doit être dimensionné de manière à supporter, en terme de charge de calcul, de stockage ou de bande passante, les sollicitations du ou des applications clientes. Un dimensionnement trop important entraîne la sous-utilisation de ressources physiques qui, si elles ne sont partagées entre les utilisateurs, ne peuvent servir à d'autres tâches. À l'inverse, un dimensionnement insuffisant entraîne une dégradation du service rendu qui se répercute sur les performances de l'application.

4.1.1 Un scénario de déploiement additionnel

Nous nous intéressons à des applications pour lesquelles il n'est pas possible de prévoir le comportement et les besoins. Dans ce contexte, une approche *manuelle* consiste tout d'abord à évaluer les besoins de l'application à déployer avant d'effectuer la réservation des ressources. Cette étape nécessite que l'opérateur *humain* ait une connaissance approfondie du fonctionnement de l'application pour effectuer un dimensionnement adapté. La sélection des ressources se détermine en effet en partie à partir du nombre d'*entités*[1] à déployer pour que l'application fonctionne, mais aussi par des contraintes de placement. Par exemple, deux entités amenées à interagir ensemble de manière fréquente ont intérêt à se trouver sur deux ressources situées dans une même grappe de calculateurs pour ainsi profiter de communications haute performance. La connaissance du fonctionnement de l'application permet à l'opérateur de proposer une topologie minimale pour le déploiement en indiquant quels types d'entités doivent être déployées et sous quelles conditions de placement. Une fois cette topologie minimale établie, le dimensionnement s'effectue de manière à prendre en compte de façon quantitative et qualitative les sollicitations qui vont être effectuées par les clients, engendrant l'utilisation des ressources physiques (processeur, mémoire physique, disque, réseau, etc.). Ceci est mis en œuvre notamment en augmentant le nombre de certaines entités. Le nombre de ressources ainsi estimé est réservé, de manière statique, pour le temps total d'exécution de l'application. L'application est alors déployée sur toutes ces ressources.

[1] Une entité est un élément de base à déployer. Cette notion est introduite dans la section 4.2.2.

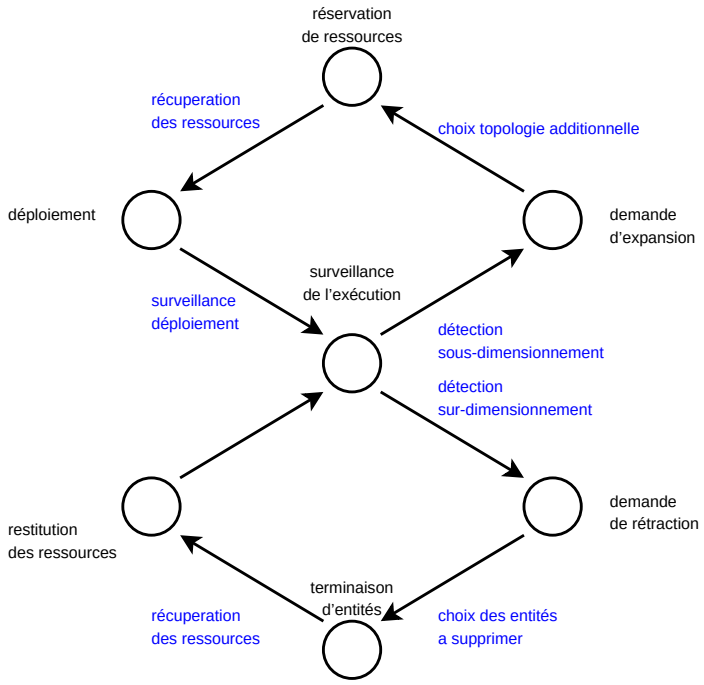

FIG. 4.1 – Déploiement additionnel : expansion et rétraction.

Pendant l'exécution de l'application, l'opérateur peut être averti, par des outils de surveillance, que certaines ressources sont sous-utilisées ou que, à l'inverse, les performances de l'application se dégradent par manque de ressources physiques allouées.

Sous-utilisation des ressources. L'opérateur peut décider de supprimer certaines entités déployées. Il peut utiliser un mécanisme interne à l'application ou faire appel à un mécanisme de base mis à disposition par le système d'exploitation de la ressource. Ce nombre d'entités à stopper reste, là aussi, décidé de manière empirique et fait appel à la connaissance du fonctionnement de l'application. Par ailleurs, l'opérateur peut restituer le sous-ensemble des ressources physiques réservées correspondant à ces entités supprimées avant la date finale de la réservation initiale.

Dégradation des performances. L'opérateur doit, de nouveau, estimer de quel ordre est le redimensionnement de la topologie de l'application. Il doit alors effectuer une nouvelle réservation de ressources, avec des contraintes supplémentaires de placement et de durée, dépendantes du déploiement précédent, avant de pouvoir déployer la partie additionnelle de l'application.

La figure 4.1 montre la suite des actions restant à la charge de l'opérateur pour prendre en compte l'aspect dynamique de la topologie d'une application pendant son exécution. La fréquence des déploiements additionnels de certaines parties d'une application, que ce soit pour une expansion ou une rétraction, dépend principalement des sollicitations extérieures durant son exécution. À chaque fois qu'une modification de la topologie de l'application est demandée, des opérations fastidieuses mettant en jeu les outils de réservation et de déploie-

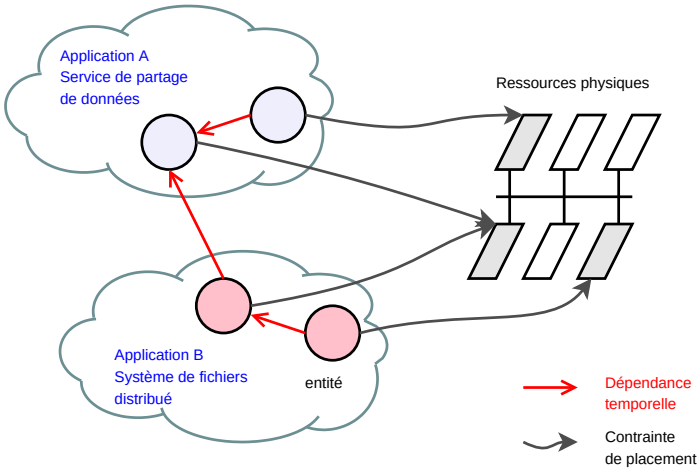

FIG. 4.2 – Co-déploiement : expression de contraintes temporelles et des contraintes de placement entre entités de deux applications.

ment de la plate-forme d'exécution sont réalisées par l'opérateur. Dans ce contexte, il serait intéressant de disposer d'un outil permettant d'automatiser toutes ces opérations et ainsi de rendre transparente la gestion des ressources et des déploiements pour l'opérateur.

4.1.2 Un scénario de co-déploiement

Dans le contexte du calcul scientifique, les applications, qu'elles soient utilisées dans un but expérimental ou pour de l'exploitation, sont bien souvent formées par la composition de plusieurs applications. À titre d'exemple, une application de simulation numérique basée sur le couplage de code met en jeu différentes applications, chacune responsable d'un calcul métier précis. Ces applications vont s'échanger des données de manière épisodique. Un autre exemple, décrit en section 6, consiste en un service hiérarchique distribué de partage de données stockées en mémoire physique et s'appuyant sur un système de fichiers global distribué. Le déploiement de telles *applications composées* ne se limite pas au simple déploiement, de manière indépendante, de chacune de leurs applications. En effet, des contraintes d'ordre temporel ou de placement interviennent non seulement au sein des applications, mais aussi entre les entités de différentes applications. C'est pourquoi la notion de *co-déploiement* fait référence à l'action de déployer de manière coordonnée une application formée de plusieurs applications.

Les contraintes temporelles liant deux entités d'une application se rencontrent principalement lorsqu'il existe une dépendance fonctionnelle d'une entité sur l'autre. Cette remarque est aussi valable à l'échelle des applications, exprimant le fait qu'une application ne peut être déployée avant qu'une autre application ne soit opérationnelle. L'exemple du service hiérarchique de partage de données cité précédemment est illustré par la figure 4.2. Le service de partage de données en mémoire physique (A) ne peut être déployé avant que le système de fichiers global (B) sur lequel il s'appuie ne soit rendu opérationnel. Le co-déploiement s'assurera donc ici que la topologie minimale de l'application B soit déployée avant d'effectuer le déploiement de A. Si l'on se place dans le contexte d'un opérateur humain ayant à effectuer le déploiement de l'application composée globale, celui-ci doit non seulement avoir une

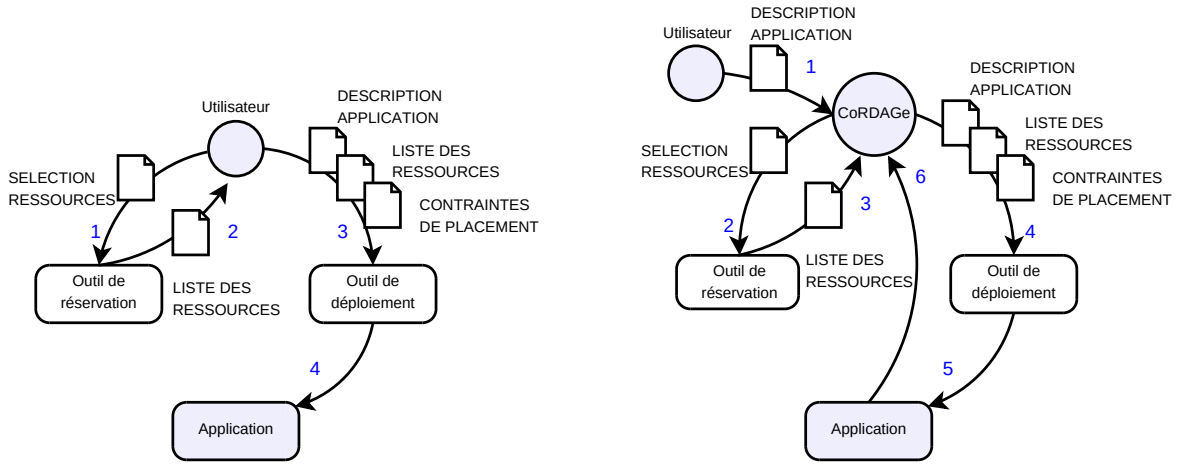

FIG. 4.3 – (a) Déployer une application sans CORDAGE. (b) Déployer une application avec CORDAGE.

connaissance approfondie du fonctionnement de chacune des applications, mais aussi une vision globale lui permettant de démarrer, dans un ordre adapté, les entités des différentes applications.

Le co-déploiement introduit aussi des contraintes de placement entre les applications. Ces contraintes peuvent par exemple exprimer que deux applications doivent être placées sur une même grappe physique pour tirer parti de communications haute performance. À l'inverse, elles peuvent améliorer la tolérance aux défaillances en cas de panne sur un site, en déployant ces applications sur des sites distincts. Elles peuvent aussi exprimer, de manière plus fine, que telle entité de telle application doit être placée sur la même ressource physique qu'une autre entité appartenant à une autre application, comme cela est montré sur la figure 4.2. De plus, dans l'exemple cité précédemment, il peut être intéressant de distribuer l'application A sur plusieurs sites pour se prémunir de la défaillance d'un site. Quant à l'application B, elle doit être déployée sur tous les sites sélectionnés pour A, afin de profiter de communications inter-sites rapides entre A et B. Là encore, le travail de l'opérateur pourrait être simplifié en prenant en charge de manière automatique ces contraintes de placement.

4.1.3 La généricité

Les scénarios de déploiement additionnel et de co-déploiement présentés dans les sections précédentes mettent en valeur un certain nombre d'actions. Elles portent sur la gestion des ressources physiques et des entités logiques laissées à la charge de l'opérateur humain. La figure 4.3(a) montre une suite d'actions à effectuer en cas de modification de la topologie initiale. Ces actions peuvent être décomposées en deux parties : une partie spécifique à l'application et une partie générique. D'un point de vue générique, ces actions se traduisent par le choix du type des entités à ajouter et leurs cardinalité respectives, le choix et la réservation des ressources physiques jusqu'à la date finale et le déploiement des entités sur les ressources choisies. Pour chacune de ces étapes interviennent des paramètres spécifiques aux applications déployées ainsi qu'aux outils de réservation et de déploiement.

Le choix du type des entités à ajouter est conditionné par la nature du sous-dimensionnement détecté. Par exemple, dans un système de fichiers distribué, le sous-dimensionnement peut intervenir au niveau d'un serveur de méta-données ou d'un serveur de stockage, ce qui va déterminer s'il faut redéployer une nouvelle entité pour la gestion des méta-données ou pour les données.

Le choix des ressources est dicté par les besoins spécifiques de chaque entité en terme de communication, d'utilisation du processeur, de mémoire physique ou du disque. Prenons l'exemple du déploiement d'une entité de stockage supplémentaire dans un système de fichiers, la ressource physique choisie devra être munie d'un espace disque suffisant et être localisée, d'un point de vue réseau, à proximité des entités lésées par ce sous-dimensionnement. La nature des interactions avec l'outil de réservation est, elle aussi, très spécifique à l'outil de réservation utilisé, notamment en terme d'interface utilisateur ou de représentation des ressources physiques.

Le choix de l'outil de déploiement introduit des notions et concepts spécifiques et génériques à cet outil. Ces notions peuvent porter sur des contraintes temporelles dans l'ordre de déploiement des entités ou sur des contraintes de placement. Là encore la spécificité provient de l'interface utilisateur et de la représentation des applications à déployer.

Afin de faciliter le travail de l'opérateur lors de la gestion d'un déploiement, l'ensemble des actions génériques précédemment décrites peuvent être prises en charge par un outil auxiliaire, spécialisé dans la gestion des interactions entre les applications et les outils de réservation et de déploiement. Cet outil remplit les objectifs suivants :

Rendre transparent pour l'opérateur la gestion des entités logiques de son application. Ceci vaut pour les opérations liées au déploiement, mais aussi pour la réservation et la libération des ressources physiques. Dans le scénario du déploiement additionnel, l'opérateur ne devra donc décrire son application que d'une manière minimale - ou demander un déploiement par défaut - et laisser à cet outil auxiliaire le soin d'effectuer les réservations et un déploiement initial.

Prendre en charge l'évolution de la topologie de l'application en fonction de ses besoins. Ceci se traduit par des réservations et déploiements supplémentaires orchestrés de manière transparente, suite aux demandes issues de l'application ou d'un service de surveillance. Le rôle de l'utilisateur ne se limite ici qu'à celui d'un observateur.

Nous proposons dans ce travail un tel outil, nommé CORDAGE, pour *co-déploiement et redéploiement d'applications générique*. Il doit offrir un canevas générique pour le déploiement et la gestion des ressources tout en étant spécialisable pour différentes applications, pour différents outils de réservation et pour différents outils de déploiement.

4.2 La vision CORDAGE

CORDAGE propose d'offrir aux applications déployées la possibilité de modifier leur configuration de manière coordonnée et transparente pour l'utilisateur de l'application. Pour cela, CORDAGE s'interface entre les applications déployées et les outils de gestion de la plate-forme physique en charge de la réservation des nœuds et du déploiement,

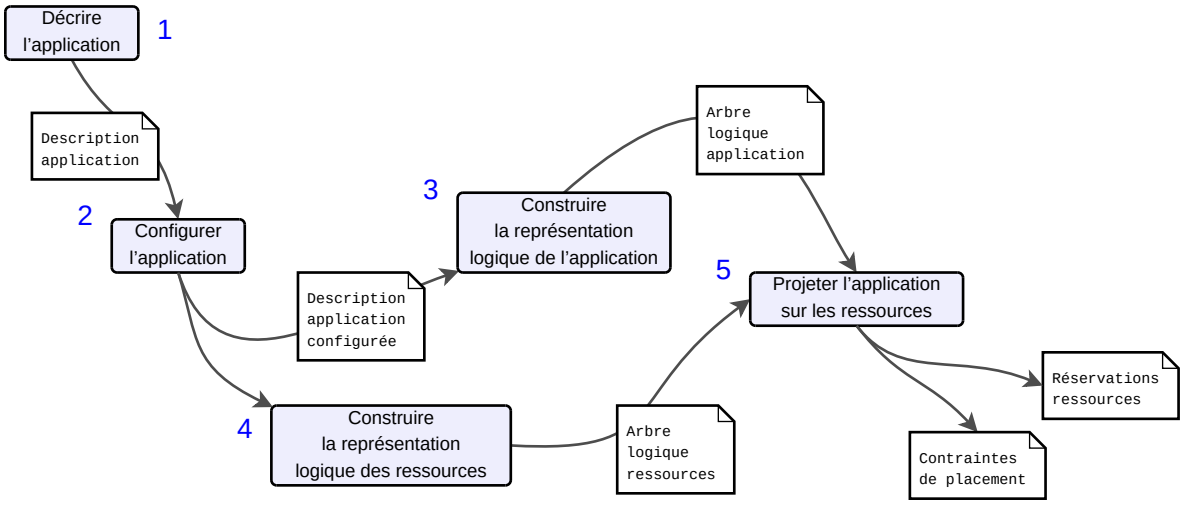

FIG. 4.4 – Les cinq phases du modèle proposé dans CORDAGE.

comme montré sur la figure 4.3(b). Les interactions mises en place entre les applications et CORDAGE s'effectuent sur le principe des appels de procédure distante (RPC). Ces appels contiennent l'identité de l'application appelante, le type d'action à effectuer ainsi que la liste des paramètres associés à cette action. Ils déclenchent l'exécution d'actions génériques offertes par le serveur CORDAGE. Ces actions sont rendues disponibles pour tout type d'application. Lorsqu'une application s'enregistre auprès du serveur, une *fiche de suivi* est créée, contenant une représentation logique de l'application, une représentation des ressources physiques ainsi que d'autres variables d'état auxquelles toutes les actions peuvent accéder. Afin de s'adapter au mieux à chaque type d'application, de nouvelles actions spécifiques sont fournies pour étendre les possibilités de la fiche de suivi. Ces actions spécifiques ont, elles aussi, accès aux variables d'état. L'ensemble de ces actions permet à CORDAGE de proposer deux fonctionnalités principales.

- Une fonctionnalité de traduction des *actions* spécifiques à l'application en une description du déploiement propre au type de l'application.
- Une fonctionnalité de pré-planification du déploiement par projection des entités sur un ensemble de ressources physiques.

Afin d'offrir ces deux fonctionnalités, un modèle décrivant les différentes étapes dans le déploiement d'une application est proposé par CORDAGE. Il est orienté vers la gestion des entités de l'application en terme d'organisation et de placement. Ce modèle propose cinq phases illustrées sur la figure 4.4. Elles permettent, à partir d'une description d'application, de sélectionner des ressources physiques pour le déploiement tout en respectant des contraintes d'affinité entre les différentes entités à déployer. Ces différentes phases sont les suivantes.

La description de l'application permet de caractériser une application à partir d'un ensemble de type d'entités.

La configuration d'une application permet d'instancier cette application à partir de sa description. La configuration porte notamment sur le dimensionnement de l'application à déployer.

La construction de la représentation logique de l'application permet, à partir d'une description spécifique, de construire une description générique pouvant être manipulée par CORDAGE.

La construction de la représentation logique des ressources physiques permet d'exprimer les liens existant entre les ressources pouvant être utilisées pour les déploiements.

La projection de la représentation de l'application sur la représentation des ressources permet d'effectuer une pré-planification du déploiement.

4.2.1 Phase 1 : description de l'application

La description de l'application est la première phase du modèle proposé par CORDAGE. Elle consiste en la représentation des applications sous la forme d'une description minimale. Cette description présente les différents *types d'entités* qui composent une application. Le type d'une entité est défini par le programme à exécuter, indépendamment des paramètres initiaux. À titre d'exemple, dans le cadre d'un système de fichiers distribué, l'application correspondante peut être décrite par un ensemble contenant trois types d'entités : le type serveur de méta-données, le type serveur de stockage et le type client. Cette représentation ne donne cependant aucune information sur la cardinalité de chacun des types d'entités ni sur les propriétés qui les lient. Ces informations sont déterminées lors de la seconde phase du modèle, la *phase de configuration*.

Définition 4.1 : ***type d'entité*** — Le type d'une entité est défini par un programme à exécuter, indépendamment des paramètres initiaux.

Définition 4.2 : ***application (dans le contexte* CORDAGE*)*** — Une application est définie par un ensemble de types d'entités.

4.2.2 Phase 2 : configuration de l'application

La configuration de l'application consiste en l'instanciation des types d'entités qui composent l'application en *entités*. Les entités sont les éléments de base manipulés par CORDAGE et devant être déployés, de manière unique, sur une ressource physique. Cette phase permet de spécifier deux informations sur l'application à déployer.

- La première information est le *dimensionnement* de l'application en terme du nombre d'entités à déployer. Cette phase spécifie donc, pour chaque type d'entité une certaine cardinalité. Plusieurs entités peuvent avoir le même type d'entité et certains types d'entité peuvent ne pas être représentés. Dans le cadre du système de fichiers distribué, nous pouvons par exemple demander la création d'une entité s ayant le type serveur de méta-données, de trois entités $f_{1..3}$, ayant le type serveur de stockage et d'une dernière entité ayant le type client c.
- La deuxième information est issue de la phase de configuration. Cette phase permet d'attacher à chacune des entités créées, de l'*information sémantique*. Cette information peut porter sur les paramètres initiaux des entités mais aussi sur les liens à tisser entre elles. Dans l'exemple du système de fichiers, de l'information sémantique peut être attachée aux entités afin d'indiquer pour chaque f_i la taille de son espace de stockage et pour le client la taille des données à stocker. Ces informations permettent de mettre en valeur des liens privilégiés entre le client et un serveur de stockage en particulier.

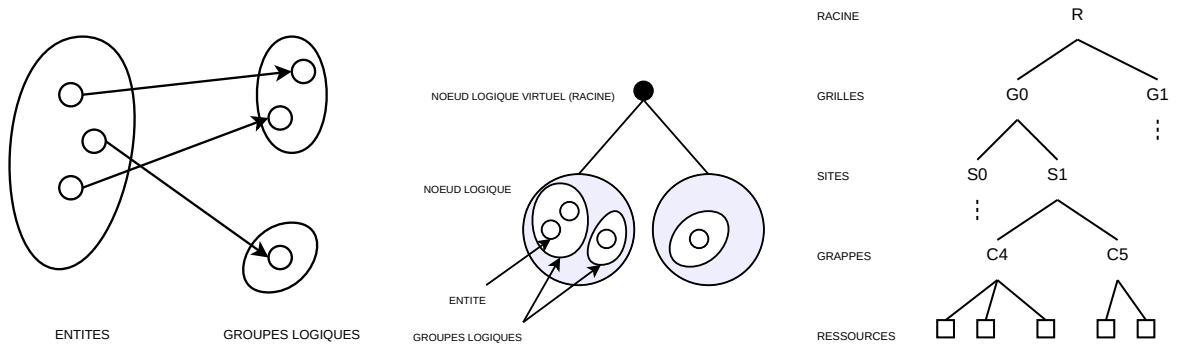

FIG. 4.5 – (a) Représenter l'application en utilisant les groupes logiques. (b) Représenter l'application en utilisant un arbre logique. (c) Représenter les ressources physiques en utilisant un arbre physique.

L'ensemble des entités obtenues à l'issue de la phase de configuration définit une *application configurée*. La notion d'application configurée est illustrée par les *descriptions d'application spécifiques* utilisées par l'outil de déploiement ADAGE.

Définition 4.3 : ***entité*** — Une entité est une instance d'un type d'entité créée dans le but d'être exécutée sur une ressource physique. Elle est caractérisée par son type et par un ensemble d'informations sémantiques permettant de paramétrer son exécution.

Définition 4.4 : ***application configurée*** — Une application configurée est composée d'un ensemble d'entités et des informations sémantiques qui les accompagnent. Ces informations permettent notamment d'*établir des liens* entre les entités.

4.2.3 Phase 3 : construction de la représentation logique de l'application

La *représentation logique* d'une application permet à CORDAGE de manipuler de manière non-spécifique au type de l'application, les différentes entités et les liens établis entre elles. Cette représentation logique est construite à partir de l'application configurée. Deux représentations logiques sont proposées par le modèle CORDAGE : la première est basée sur la notion de groupes logiques et la seconde, plus générique, organise les groupes logiques en un arbre logique.

4.2.3.1 Groupes logiques

Cette représentation vise à décomposer l'ensemble des entités d'une application configurée obtenue à la phase de configuration en un ensemble de *groupes logiques* (voir figure 4.5(a)). Chaque groupe contient les entités appartenant à une même *classe*. Les groupes logiques permettent de mettre en valeur et de factoriser les liens privilégiés existant entre les entités. Le regroupement des entités selon certains critères permet de faciliter la gestion des contraintes de placement lors du déploiement en les attachant à un nombre limité de classes et non plus à chaque entité. À l'issue de la phase de regroupement, chaque entité doit appartenir à une unique classe qui sera plus tard associée à un ensemble de ressources physiques.

Définition 4.5 : ***groupe logique*** — Un groupe logique est un ensemble d'entités.

Cette décomposition est propre à chaque application, qui fournit les critères spécifiques à sa réalisation. La construction de ces classes est basée sur une relation d'affinité $\alpha(e, f)$ indiquant si deux entités e et f *peuvent* être placées dans la même classe. Cette relation exprime le souhait de voir deux entités appartenir à une même classe, ce qui ne se traduit pas par une contrainte forte d'appartenance. À l'opposé, $\neg(\alpha(e, f))$ exprime le fait que e et f *ne doivent pas* appartenir à la même classe, ce qui se traduit par une contrainte forte. La relation d'affinité est symétrique, c'est-à-dire que $\alpha(e, f)$ est équivalent à $\alpha(f, e)$. L'application de la relation d'affinité sur les entités prises deux à deux entraîne la création de nouvelles classes. Le nombre de classes créées est strictement positif et inférieur ou égal au nombre d'entités. L'objectif est donc de minimiser le nombre de classes créées.

Prenons un cas simple composé de deux entités e et f. Si $\neg(\alpha(e, f))$, alors e et f entraînent la création de deux classes. Si, au contraire nous avons $\alpha(e, f)$, une seule classe est créée. Si l'on ajoute maintenant une troisième entité g et que l'on considère le cas $\{\alpha(e, g), \alpha(f, g), \neg(\alpha(e, f))\}$, e et f entraînent la création de deux classes auxquelles g peut appartenir. Le modèle utilisé par CORDAGE interdit à une entité d'appartenir à deux classes différentes. Dans le cas précédent, l'entité g doit donc être placée dans l'une des deux classes. Le choix s'effectue au niveau de l'implémentation des modules CORDAGE spécifiques à l'application.

La relation d'affinité α est propre à chaque application et son évaluation est fonction du type des entités et des informations sémantiques attachées à chacune d'elles lors de la phase de configuration. Considérons l'exemple précédent du système de fichiers, composé d'une entité serveur s, des entités de stockage $f_{1..3}$ et du client c. Lors de la phase de configuration, nous pouvons attacher de l'information sémantique décrivant la taille de l'espace de stockage fournie par les f_i et les besoins en espace de stockage requis par c. Dans ce contexte, considérons une relation d'affinité basée sur 1) le type des entités et 2) la différence entre la taille offerte par les entités de stockage et les besoins de stockage des entités clientes. L'application de cette relation conduit à la création d'une classe pour le serveur de méta-données et d'une classe pour chacun des f_i. L'entité c est placée dans la même classe que le f_i proposant un espace de stockage proche de ses besoins. Si plusieurs f_i sont susceptibles d'accueillir c, le choix de la classe est effectué de manière arbitraire.

4.2.3.2 Extension de la représentation : arbres logiques

Les arbres logiques visent à représenter l'application configurée en une structure en arbre dont les nœuds et les feuilles contiennent des groupes logiques (voir figure 4.5(b)). Cette approche permet d'étendre l'expressivité des groupes logiques en permettant l'organisation des entités de manière hiérarchique et non plus de manière plane. Cette représentation en arbre est dirigée par l'aspect hiérarchique des applications et des ressources que nous considérons dans ces travaux. Cette hiérarchie peut exprimer par exemple des relations de proximité en terme de distance physique ou sémantique entre les entités.

La localisation d'un groupe logique dans l'arbre donne deux informations. La première est la proximité avec les autres groupes logiques qui se traduit plus tard en terme de distance des ressources physiques, selon le critère de la distance réseau par exemple. La deuxième information est la profondeur du groupe logique dans l'arbre. Plus un groupe est situé en

profondeur, plus les ressources physiques sélectionnées pour supporter l'exécution des entités de ce groupe sont proches. Cette notion de proximité entre les ressources physiques est expliquée dans la phase 4.

Lors de la construction de l'arbre il est aussi possible d'insérer des groupes logiques vides. Ce type de nœud un peu particulier est appelé *nœud virtuel*. Il permet de créer une racine virtuelle lorsque tous les groupes logiques doivent être placés sur un même niveau dans l'arbre. Il permet aussi de pousser des groupes logiques vers le bas de l'arbre. Les groupes logiques sont alors situés dans des nœuds plus éloignés de la racine. Ceci se traduit pour les entités contenues par les groupes logiques par un déploiement sur des ressources physiques plus proches. Enfin, il permet d'augmenter la distance entre deux groupes logiques pour qu'ils soient placés sur des ressources plus éloignées.

Définition 4.6 : ***nœud logique*** — Un nœud logique est un ensemble de groupes logiques.

Définition 4.7 : ***nœud logique virtuel*** — Un nœud logique virtuel est un nœud logique vide.

Définition 4.8 : ***arbre logique*** — Par abus de langage, un arbre logique désigne la représentation logique de l'application configurée sous forme d'un arbre, dont les nœuds et les feuilles sont des nœuds logiques.

4.2.4 Phase 4 : construction de la représentation des ressources physiques

Les ressources physiques permettent le support de l'exécution des différentes *entités* qui forment une application configurée. L'un des rôles de CORDAGE consiste à sélectionner pour chaque groupe logique un ensemble de ressources physiques susceptibles de les accueillir.

Ces ressources se distinguent par leurs caractéristiques techniques comme le type d'unité de calcul, la taille de la mémoire physique ou l'espace disque. Elles se distinguent aussi par leur interconnexion, les plaçant à des distances variables en fonction de la latence ou du débit réseau qui les séparent. Ces propriétés sont à prendre en compte pour choisir la ressource physique se prêtant le mieux à l'exécution d'une entité. Pour cela, nous proposons de regrouper les ressources en fonction des caractéristiques les plus pertinentes pour l'application configurée devant être déployée.

La représentation des ressources physiques dans CORDAGE est effectuée sous forme de *groupes physiques* hiérarchiques. Chaque ressource appartient à des groupes de plus en plus englobants et ce, jusqu'au groupe global contenant toutes les ressources de la plate-forme d'exécution. Cette représentation peut aussi être vue comme un arbre dont les feuilles sont les ressources physiques et les nœuds portent les noms des groupes physiques. La construction de ces groupes hiérarchique dépend des critères de sélection des ressources propres à l'application. Par exemple, si l'application est fortement communicante, il peut être préférable de privilégier une organisation hiérarchique des ressources sur le critère de la distance réseau. Cette approche est la plus courante, notamment pour des applications destinées à être exécutées sur des architectures de type grille de calculateurs. Dans ce contexte, les ressources appartiennent à une grappe de ressources, à un site puis à une grille comme cela est illustré par la figure 4.5(c).

Définition 4.9 : ***ressource physique*** — Une ressource physique est un élément matériel permettant l'exécution d'une entité.

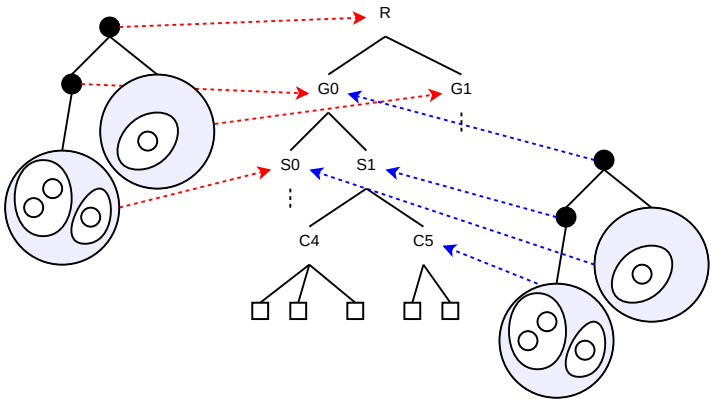

FIG. 4.6 – Deux exemples de projection : le plus haut (gauche) et le plus bas (droite).

Définition 4.10 : ***groupe physique*** — Un groupe physique est un ensemble de ressources physiques.

Définition 4.11 : ***arbre physique*** — L'arbre physique désigne un arbre représentant les ressources physiques de manière logique. Il est formé de nœuds étiquetés par des noms de groupes physiques, ainsi que des ressources physiques en guise de feuilles.

CORDAGE représente les ressources de manière statique. Il n'est pas prévu aujourd'hui de pouvoir modifier l'arbre des ressources pendant l'exécution de l'application. De plus, l'état courant des réservations de ressources par d'autres utilisateurs n'est pas reflété par cette représentation. Les ressources présentes dans l'arbre logique sont donc considérées comme potentiellement utilisables, sans garantie de disponibilité lors d'une demande réservation. Elles sont issues d'une base d'informations sur les ressources mise à disposition par l'infrastructure. Le modèle n'interdit pas pour autant de limiter les ressources présentes dans cette représentation à un sous-ensemble des ressources de la grille. Une présélection pertinente des ressources pour l'application est donc possible à ce niveau. Ceci ne devrait cependant pas s'opposer au travail fourni par les ordonnanceurs, mais au contraire guider la sélection finale des ressources à la lumière des propriétés spécifiques à l'application. Cette construction de la représentation des ressources physiques contribue, avec la phase de projection présentée ci-après, au travail de pré-planification offert dans le modèle CORDAGE.

4.2.5 Phase 5 : projection de l'arbre logique sur l'arbre physique

Une fois les représentations logiques et physiques construites, la phase suivante du modèle consiste en une planification partielle du déploiement. Cette planification partielle est réalisée en projetant l'arbre logique sur un sous-arbre des ressources physiques. Chaque groupe logique doit être associé à un groupe physique en respectant la hiérarchie des deux arbres. Nous considérons l'arbre des ressources physiques plus large et plus profond que l'arbre logique, de telle sorte qu'il est possible de trouver un groupe physique pour chaque groupe logique[2]. Cette étape génère des contraintes préliminaires de placement qui asso-

[2]Dans le cas contraire, un *repliement* de l'arbre logique doit être opéré. Le repliement a pour but de réduire la largeur et la profondeur de l'arbre. Il consiste à migrer des groupes logiques vers des nœuds logiques parents ou encore à fusionner des sous-arbres frères en un seul sous-arbre.

cient un ensemble d'entités à un ensemble de ressources physiques. À la fin de cette étape, toutes les réservations de ressources ont été faites. La projection finale qui associe une ressource physique particulière à chaque entité est cependant laissée à la charge de l'outil de déploiement.

La figure 4.6 présente deux manières de projeter un même arbre logique sur un arbre physique. La projection de gauche est une projection au plus haut qui tente de sélectionner un sous-arbre le plus proche possible de la racine. Cette approche cherche à placer les groupes logiques sur des ressources physiques éloignées. À l'inverse, la projection de droite est une projection au plus bas qui sélectionne un sous-arbre proche des feuilles. L'intérêt est ici de placer les entités d'un même groupe logique sur des ressources physiques très proches. Dans cet exemple, la projection au plus haut se traduit par l'utilisation des deux grilles $G0$ et $G1$ alors que la projection au plus bas ne sélectionne que la grille $G0$.

Cette phase se traduit par la recherche itérative d'une projection valide de l'arbre logique sur un sous-arbre physique. Pour chaque solution, les ressources physiques associées sont réservées. Ce processus itératif se termine si toutes les ressources ont pu être réservées. Nous considérons en effet un environnement partagé entre plusieurs utilisateurs pouvant impliquer des essais de différentes projections avant de trouver une solution avec suffisamment de ressources libres. Si aucune solution n'a été trouvée en balayant toutes les projections possibles sur un même niveau de l'arbre physique, l'algorithme peut sélectionner des sous-arbres placés sur un niveau supérieur dans la hiérarchie de l'arbre physique. Cela se traduit par des contraintes de placement moins fortes et donne accès à un plus grand ensemble de ressources, augmentant les chances d'effectuer toutes les réservations avec succès.

Le résultat de cette phase est la projection de chaque groupe logique sur un groupe physique, en accord avec la hiérarchie des deux arbres. Chaque association se traduit par une contrainte de placement. L'ensemble de ces contraintes de placement, les réservations associées ainsi que la description de l'application sont donnés en entrée de l'outil de déploiement qui peut alors prendre en charge l'application configurée. Le résultat de la phase de déploiement est une *application déployée*. Tous les groupes logiques de l'arbre principal passent alors dans le status *déployé* et ne peuvent plus être modifiés ou déplacés.

Définition 4.12 : ***application déployée*** — Une application déployée est une application configurée dont la représentation logique a été projetée avec succès sur l'arbre physique.

4.3 Fonctionnement du modèle

Le modèle précédemment introduit permet de représenter, d'organiser les entités composant une application puis de les projeter sur des ressources physiques en vue de leur déploiement. Une fois le déploiement effectué, le modèle est utilisé pour exprimer l'aspect dynamique de la topologie de l'application déployée tout au long de son exécution.

4.3.1 Expansion et rétraction

Selon la vision CORDAGE, l'aspect dynamique des applications s'exprime par le besoin de faire évoluer leur topologie par expansion ou rétraction. Le problème de la migration d'entités sur d'autres ressources physiques n'est pas pris en compte par le modèle. Les applications doivent être capable de demander : 1) le déploiement de nouvelles entités et 2) la

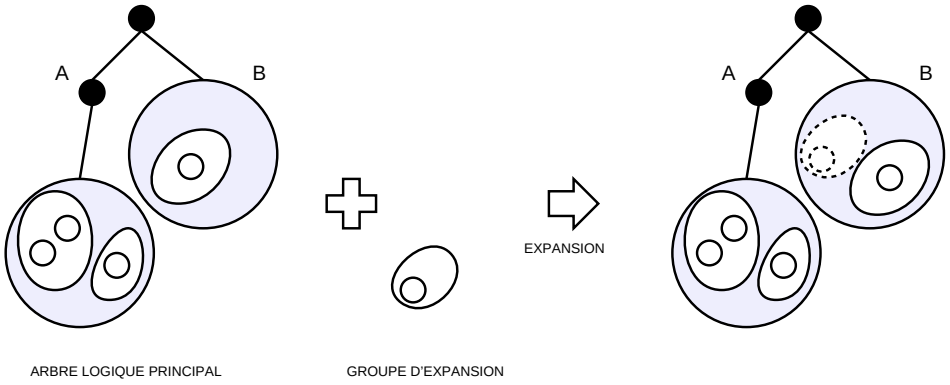

FIG. 4.7 – Expansion : ajout d'un nouveau groupe dans l'arbre principal.

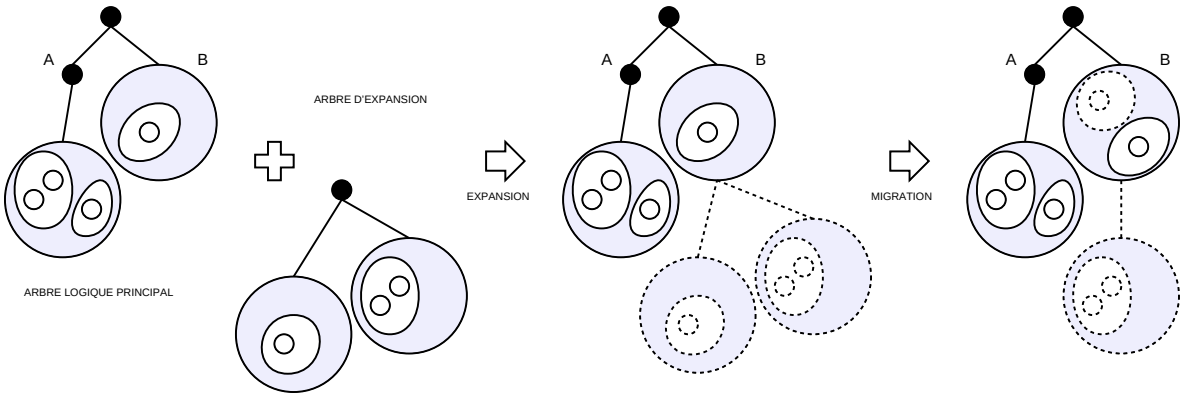

FIG. 4.8 – Expansion : ajout d'un nouveau sous-arbre dans l'arbre principal. Migration : déplacer un groupe logique lors de la phase de projection vers le nœud parent.

suppression d'entités précédemment déployées. Nous définissons deux opérations correspondant à l'expansion et à la rétraction d'une application déployée.

4.3.1.1 Expansion

Étendre une application précédemment déployée consiste à ajouter de nouvelles entités à l'ensemble des entités déployées. Cette opération est effectuée en suivant les différentes phases du modèle présenté en section 4.2. Les nouvelles entités doivent être organisées sous forme d'un ensemble de groupes logiques qui sont par la suite insérés dans l'arbre logique principal. La figure 4.7 montre qu'il est possible d'insérer ces groupes directement au sein de nœuds logiques dans l'arbre principal. Ces nœuds peuvent déjà contenir un ou plusieurs groupes logiques comme c'est le cas pour B ou être des nœuds virtuels comme A. L'ajout d'un groupe logique dans B est motivé par le besoin de déployer l'entité qu'il contient sur une ressource physique appartenant au même groupe physique sélectionné pour la projection de ce nœud logique. L'ajout dans A se traduit par la sélection de ressources physiques différentes de celles retenues pour le déploiement des entités de B.

Il est aussi possible d'organiser une partie des ces nouveaux groupes logiques en un arbre logique, de la même manière que pour l'arbre logique principal. Ce nouvel arbre s'appelle

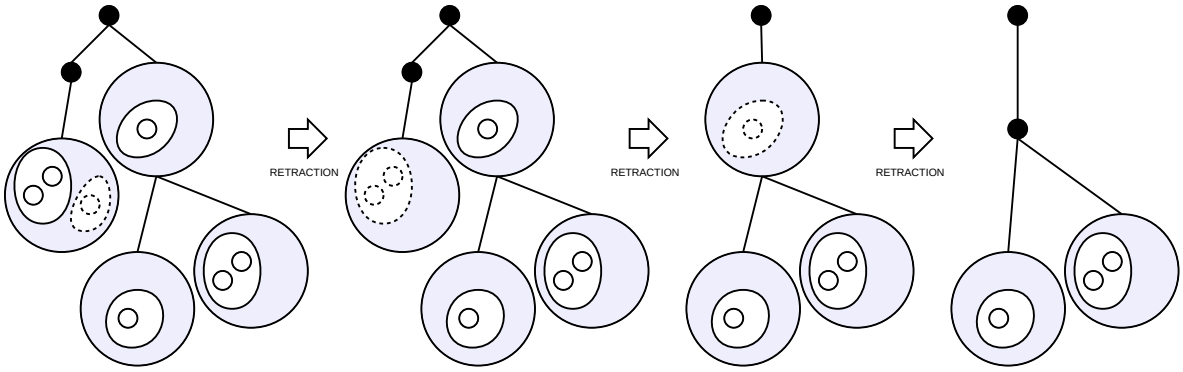

FIG. 4.9 – Rétraction : supprimer des groupes logiques de l'arbre principal.

l'*arbre d'expansion* et doit être attaché comme un sous-arbre d'un nœud ou d'une feuille de l'arbre principal. Ce nœud particulier est appelé *nœud d'expansion* (nœud *B* sur la figure 4.8). Il remplace le nœud virtuel faisant office de racine de l'arbre d'expansion. L'application peut aussi fournir des informations complémentaires afin de déterminer le nœud d'expansion.

La phase de projection est alors effectuée sur les nouveaux groupes logiques insérés, en prenant en compte le fait qu'une partie de l'arbre principal est déjà projetée et déployée. Si, pour un groupe logique, il n'est pas possible de réserver des ressources en suivant ces contraintes de placement, ce groupe peut être migré vers le nœud logique parent dans l'arbre (voir figure 4.8), augmentant le nombre de ressources susceptibles d'êtres réservées et donc les chances de succès. Une fois la projection effectuée, CORDAGE peut commander le déploiement des entités nouvellement créées.

4.3.1.2 Rétraction

Rétracter une application consiste à supprimer des entités de l'ensemble des entités actuellement déployées. L'opération correspondante dans CORDAGE s'effectue en supprimant de l'arbre logique principal un ensemble de groupes logiques déployés. Il n'est pas possible d'enlever seulement une partie des entités d'un groupe logique : le groupe entier doit être supprimé. Ceci est motivé par le fait qu'une réservation de ressources physiques est associée à chaque groupe logique. Rétracter une partie des entités d'un groupe logique ne permet pas de libérer les ressources physiques associées. La figure 4.9 présente différents cas de rétraction. Un nœud logique peut devenir vide suite à la suppression d'un groupe logique. Dans ce cas, ce nœud devient virtuel et, s'il est placé en feuille de l'arbre, le nœud est supprimé.

Supprimer des groupes logiques implique la suppression des entités, de leurs ressources physiques ainsi que la libération des réservations associées. CORDAGE invoque alors les outils classiques de gestion de la grille avec les paramètres adéquats. Cela se traduit par des requêtes à l'outil de réservation pour supprimer les réservations associées à ces ressources. Ou encore des requêtes peuvent à l'outil de déploiement pour stopper l'exécution de ces entités.

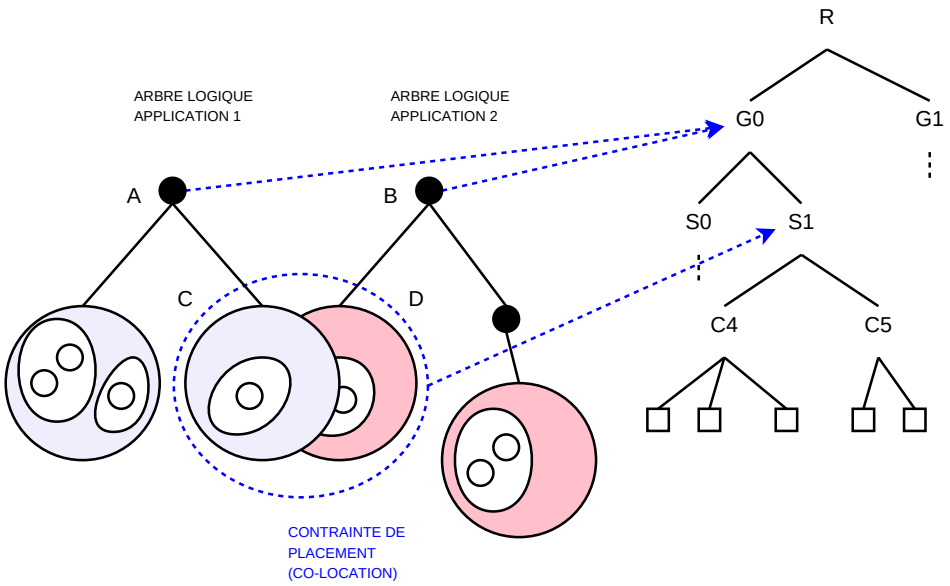

FIG. 4.10 – Co-déploiement : projection de deux arbres logiques.

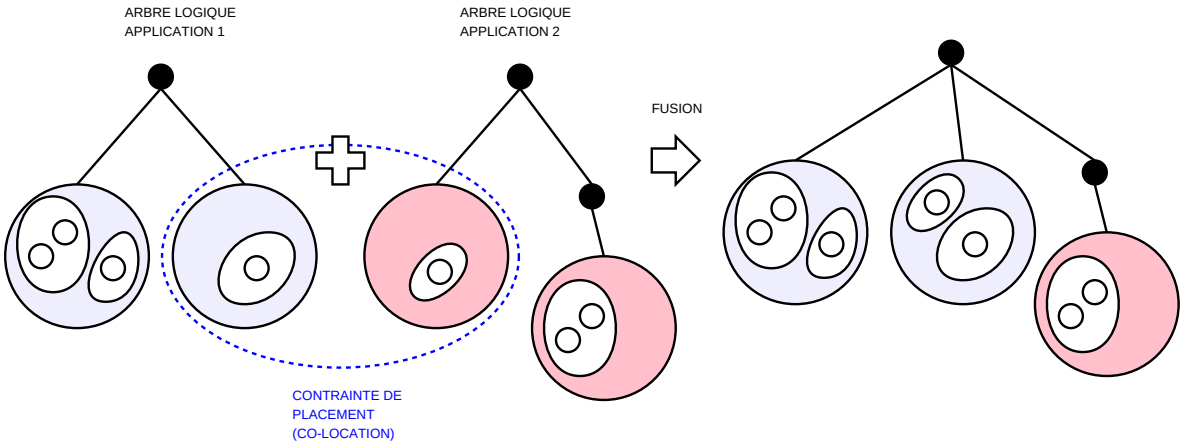

FIG. 4.11 – Co-déploiement : fusion de deux arbres logiques.

4.3.2 Co-déploiement

Le besoin de déployer des applications composées de sous-applications de types différents peut aussi être pris en compte par le modèle CORDAGE. Ce type de déploiement particulier est appelé *co-déploiement*. Il introduit des contraintes supplémentaires entre les sous-applications. CORDAGE prend en charge deux types de contraintes.

Les contraintes temporelles indiquent qu'une entité appartenant à une application X doit être déployée avant de lancer une entité appartenant à une application Y.

Les contraintes de placement indiquent qu'une entité ou un groupe d'entité d'une application X doit être placé sur le même ensemble de ressources qu'une entité appartenant à l'application Y.

Dans le modèle CORDAGE, le co-déploiement se caractérise par un ensemble de contraintes de placement inter-applications. Ces contraintes sont utilisées lors de la phase de projection des arbres logiques sur l'arbre physique. Sur la figure 4.10, cela implique que C et D soient placés sur le même groupe physique $S1$. Cette contrainte de placement se propage aussi par construction à tous les parents de C et D, ce qui explique que A et B soient aussi placés sur un même groupe physique $G0$. Plusieurs contraintes de placement exprimées en même temps peuvent provoquer des conflits lors de la phase de projection. Ceux-ci ne sont pas traités dans ce travail et nous faisons l'hypothèse que l'ensemble des contraintes sont cohérentes.

Une autre approche consiste à fusionner les deux arbres logiques avant la phase de projection. L'arbre résultant est un nouvel arbre logique représentant la fusion des deux applications. La figure 4.11 montre deux arbres logiques correspondant à deux sous-applications. Dans cet exemple, les arbres sont fusionnés en faisant correspondre leurs racines respectives. Mais il est aussi envisageable de fusionner un arbre en alignant sa racine sur un nœud logique quelconque de l'autre arbre. Les contraintes de placement permettent de fusionner certains nœuds logiques en un seul nœud, c'est-à-dire d'ajouter les groupes logiques d'un nœud de la sous-application 2 à un nœud logique de la sous-application 1. Les groupes logiques ainsi ajoutés dans ce nœud seront alors projetés sur un même ensemble de ressources physiques. La fusion de deux arbres logiques garantit, par construction, la propagation des contraintes de placement aux parents des nœuds fusionnés.

Le co-déploiement peut aussi entraîner une contrainte de placement de deux entités sur une même ressource physique. Cela se traduit par.

– La réservation d'une seule ressource pour ces deux entités.
– L'expression d'une contrainte de co-localisation à l'intention de l'outil de déploiement.

Dans ce contexte, la fusion des deux représentations logiques doit être encore plus fine et descendre jusqu'au niveau des entités. Le modèle permet de fusionner deux groupes logiques d'un même nœud logique afin de regrouper des entités appartenant à deux sous-applications différentes.

4.4 Conclusion

La gestion actuelle du déploiement d'applications distribuées reste source de tâches longues et pénibles pour l'utilisateur d'infrastructures de type grilles de calculateurs. Cette perte de temps est particulièrement importante lorsqu'il faut interagir manuellement avec les outils de réservation des ressources de déploiement des applications. Dans de telles conditions, il est difficilement envisageable de traiter de l'aspect dynamique des besoins des applications au cours de leur exécution. CORDAGE est une brique contribuant à la gestion autonome des applications. Il introduit un modèle permettant de transformer des actions de haut niveau en une suite d'opérations de bas niveau ayant trait à la réservation des ressources et au déploiement applicatif. Ce modèle se base sur des représentations logiques de l'application et des ressources physiques utilisées pour refléter l'état courant du déploiement et les objectifs à atteindre. Les actions de haut niveau spécifiques à l'application sont appliquées sur le modèle et profitent d'actions génériques en terme de manipulation de la représentation, de réservation des ressources et de déploiement des entités. Ces actions génériques consistent notamment en une opération de fusion des représentations logiques

pour prendre en compte les applications composées. Elles incluent aussi une opération de projection de l'arbre logique sur l'arbre des ressources physiques. La projection offre par ailleurs une fonctionnalité de pré-planification du déploiement, en associant un groupe de ressources physiques à chaque entité.

Le choix de la représentation des applications sous forme d'un arbre logique est motivé par l'aspect hiérarchique des applications que nous considérons dans ce travail. Cette organisation permet d'exprimer, selon un critère donné, la distance entre deux entités. Ce critère détermine la construction de l'arbre des ressources physiques sur lequel l'arbre logique est projeté. Le choix de l'organisation en arbre peut cependant se révéler être inadapté pour certaines classes d'applications comme, par exemple, les applications à base de flot de données (en anglais *workflow*). Dans ce cas, une représentation plus naturelle consiste en un graphe orienté.

Chapitre 5

Architecture détaillée de CoRDAGe

Sommaire

Le modèle et les notions de déploiement dynamique introduits dans le chapitre précédent ont été mis en œuvre au sein du prototype de recherche illustrant le service CoRDAGe. Ce prototype a été développé et testé dans le contexte de la plate-forme Grid'5000 mais il se veut suffisamment générique pour pouvoir s'adapter à d'autres infrastructures de calcul et d'exploitation. Un travail préliminaire a été effectué en 2006 lors du stage de Master Recherche deuxième année de Voichita Almasan [6] afin de valider la possibilité des interactions entre un programme utilisateur et les outils de réservation et de déploiement de la grille. Le prototype CoRDAGe est une refonte complète de ce premier travail. Il est constitué de plus de 6700 lignes de code [129], majoritairement écrites en C++, mais aussi en *C* et PERL. La procédure d'installation est basée sur l'outil de configuration *Cross-platform*

Make [134]. CORDAGE a été développé entre janvier et juin 2008 sous la forme d'un projet GForge INRIA [135]. Il fait l'objet d'un dépôt à l'Agence pour la Protection des Programmes.

Ce chapitre présente la mise en œuvre de notre contribution dans l'environnement GRID'5000. Dans une première section, nous présentons les hypothèses effectuées sur la plate-forme de développement ainsi que les contraintes devant être respectées par les outils de gestion de la grille et les applications clientes. Dans une deuxième section, nous présentons les interfaces offertes par CORDAGE aux applications clientes. Enfin, la dernière section propose un aperçu de la réalisation du prototype à travers une approche modulaire.

5.1 Hypothèses sur l'environnement de déploiement

L'environnement considéré pour la mise en œuvre de CORDAGE reflète un ensemble d'infrastructures de calcul ainsi que les services qui leur sont associés. Ces infrastructures sont caractérisées, à l'instar de la plate-forme GRID'5000, par une liberté d'utilisation suffisamment grande en terme de réservation des ressources et de déploiement des applications. Des hypothèses portent sur l'infrastructure matérielle et logicielle. D'une manière générale, l'infrastructure visée, de type grille, doit satisfaire les hypothèses suivantes.

E1 Proposer un ensemble de ressources physiques pouvant supporter l'exécution d'une entité au sens CORDAGE. Ces ressources sont typiquement des nœuds ou, plus finement, des processeurs ou des cœurs dans une machine.

E2 Rendre les ressources accessibles à partir d'au moins une adresse physique par des protocoles de connexion et de transfert.

E3 Proposer des services dédiés à la réservation des ressources et au déploiement des applications.

Ces hypothèses, très générales, suffisent à caractériser l'environnement matériel nécessaire à la mise en œuvre de CORDAGE. Nous définissons ensuite des contraintes portant sur les services offerts par l'infrastructure et les applications supportées par CORDAGE. Nous faisons ici la distinction entre la phase de *réservation des ressources* et la phase de *déploiement de l'application*. Ces deux phases sont fusionnées dans de nombreux systèmes d'ordonnancement de tâches. Par exemple sur GRID'5000, les outils KATAPULT [146] et GRUDU [142] permettent de soumettre à OAR des travaux qui vont donner lieu à une sélection de ressources puis au démarrage des entités à l'heure choisie par l'outil. Une telle approche combinée est séduisante mais ne permet qu'un déploiement limité, loin du raffinement proposé par un outil de déploiement dédié comme ADAGE. C'est pourquoi nous laissons la possibilité de choisir un outil de déploiement indépendant du choix de l'outil de réservation.

5.1.1 Contraintes sur l'outil de réservation

L'outil de réservation proposé par l'infrastructure a la charge de maintenir à jour la liste des ressources disponibles en fonction de la demande des utilisateurs. Dans le contexte de CORDAGE, cet outil doit satisfaire les hypothèses suivantes.

R1 Offrir une interface utilisateur accessible pour un programme client extérieur. Cette interface peut consister en une ligne de commande ou une connexion directe sur un port

de communication. CORDAGE joue donc le rôle d'un client standard pour l'outil de réservation et ne devrait nécessiter aucune modification de son interface.

R2 Permettre la réservation de ressources physiques en spécifiant l'heure de départ, le temps total (*walltime*) et le nombre de ressources. Il doit aussi être possible de spécifier des contraintes sur la proximité physique (en terme de connectivité réseau) ou autres caractéristiques de ces ressources.

R3 Permettre d'effectuer plusieurs réservations par utilisateur sur un même créneau horaire, c'est-à-dire que des ressources issues de réservations différentes peuvent être utilisées en même temps. Cette contrainte est dictée par l'aspect dynamique de CORDAGE : lorsque de nouvelles entités doivent être déployées, de nouvelles réservations doivent être effectuées en parallèle.

R4 Permettre de récupérer un identifiant associé à chaque réservation et une liste d'adresses physiques pour les ressources sélectionnées ou, d'une manière plus générale, tout moyen d'identification et de connexion aux ressources.

R5 Indiquer lorsque les ressources sont prêtes à être utilisées. Cette contrainte permet de prendre en compte le temps, même minime, entre l'heure de début de réservation et la mise à disponibilité effective des ressources.

R6 Pouvoir traiter plusieurs interactions concurrentes en provenance de divers clients. Des mécanismes de synchronisation doivent donc être prévus comme la séquentialisation des requêtes.

R7 [*optionnelle*] Permettre l'annulation d'une réservation. Cette contrainte contribue à la bonne gestion des ressources en permettant à CORDAGE de libérer celles qui ne sont plus utilisées.

5.1.2 Contraintes sur l'outil de déploiement

L'outil de déploiement a la charge de planifier le déploiement, c'est-à-dire associer pour chaque entité une ressource physique, configurer cette ressource en y transférant les fichiers d'exécution et les données ainsi que de démarrer les différentes entités. Afin d'être utilisé par CORDAGE, cet outil doit satisfaire les hypothèses suivantes.

D1 Accepter en paramètre des descriptions génériques ou spécifiques aux applications. Ces descriptions reflètent l'application configurée devant être déployée.

D2 Prendre en paramètre un identifiant de réservation ou une liste de ressources physiques du même type que les informations fournies en sortie de l'outil de réservation (voir contrainte R4).

D3 Permettre d'exprimer et de prendre en compte des contraintes de placement entre les entités de l'application et les identifiants de ressources physiques. Cette contrainte permet de prendre en compte la planification partielle effectuée par CORDAGE.

D4 Permettre la modification de la description de l'application, des contraintes de placement et de la liste des ressources physiques pendant l'exécution de l'application. Ces modifications doivent ensuite être répercutées sur le déploiement effectif de l'application. Cette contrainte permet la prise en charge du déploiement dynamique.

D5 Pouvoir traiter plusieurs interactions concurrentes en provenance de divers clients. Des mécanismes de synchronisation doivent donc être prévus comme la séquentialisation des requêtes.

D6 [*optionnelle*] Indiquer si des ressources réservées n'ont pas été utilisées. Cette contrainte permet une meilleure gestion des ressources en évitant des réservations inutiles.

5.1.3 Contraintes sur l'application

Les applications destinées à utiliser CORDAGE doivent satisfaire les hypothèses suivantes.

A1 Pouvoir être déployées par l'outil de déploiement de manière native ou par le biais d'un greffon (*plugin* en anglais).

A2 Pouvoir être déployées de manière incrémentale : certaines entités peuvent être ajoutées ou retranchées à la volée.

A3 Pouvoir être représentées de manière générique dans au moins un des modèles logiques proposés par CORDAGE de manière native ou à l'aide d'un greffon.

A4 [*optionnelle*] Pouvoir communiquer avec CORDAGE sur le principe de l'appel de procédure à distance. Ceci doit pouvoir se faire en faisant appel à la bibliothèque cliente CORDAGE ou en respectant le langage d'actions de CORDAGE basé sur un protocole dédié. Si aucune modification de l'application n'est possible ou souhaitée, les appels à CORDAGE peuvent être effectués à partir d'un élément auxiliaire, tel un environnement autonome.

5.2 Vue de haut niveau

L'objectif de CORDAGE est de mettre à la disposition des applications un service dédié à la gestion du déploiement. Ce service doit accompagner l'application tout au long de son exécution en prenant en compte l'historique des interactions. D'un point de vue conceptuel, cela se traduit par la mise en place d'un serveur dédié à la gestion du déploiement. Ce serveur centralisé peut avoir la charge d'une ou plusieurs applications. Pour chacune d'entre elles, il crée une *fiche de suivi*[1]. Cette fiche rassemble les informations liées au déploiement, comme la représentation logique de l'application ou encore l'adresse de l'outil de déploiement à utiliser. L'organisation générale est illustrée par la figure 5.1. Dans cet exemple, l'outil de réservation est partagé entre les différentes applications. Cette configuration est dictée par la mise en place d'une interface unique pour la réservation des ressources pour tous les utilisateurs de la grille. A l'inverse, il existe une instance de l'outil de déploiement par application, ce qui a un sens dans le cas où le déploiement est à la charge de l'utilisateur.

Par ailleurs, plusieurs instances du serveur CORDAGE peuvent être exécutées en parallèle, de manière indépendante. Par exemple une instance par application, par utilisateur ou par site de la grille. Ces différentes combinaisons permettent de répartir les applications sur les serveurs afin d'éviter toute surcharge de travail. Le fonctionnement général de CORDAGE peut être décomposé en plusieurs étapes.

[1] aussi appelée *assignment* dans la terminologie CORDAGE.

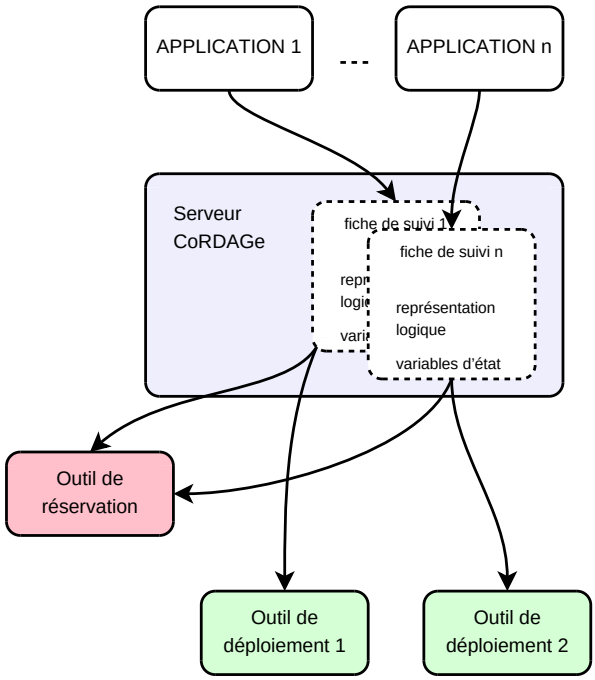

FIG. 5.1 – Une vue de haut niveau.

1. La première étape est à l'initiative d'une application. Celle-ci effectue une requête au serveur en spécifiant son identité, le type d'action ainsi que les paramètres associés à cette action.
2. La deuxième étape consiste en un traitement de cette requête sur le serveur. L'identifiant de l'application permet de sélectionner la *fiche de suivi* correspondante. L'action demandée est alors effectuée dans le contexte de cette fiche et peut entraîner une modification de la représentation logique, des autres variables d'état ainsi que des interactions avec les outils de réservation et de déploiement.
3. Enfin, le résultat de ce traitement est retourné à l'application.

Par la suite nous détaillons chacune des ces étapes du point de vue de la mise en œuvre en débutant par les interactions entre les applications et le serveur, puis en expliquant les traitements effectués au niveau du serveur.

5.3 Comment utiliser CoRDAGe : interface fournie aux applications clientes

L'activité de CoRDAGe est liée à la nature des interactions mises en place entre les applications et le serveur. Ces interactions sont basées sur le modèle *client-serveur* dans lequel les applications jouent le rôle de clients auprès du serveur CoRDAGe. Elles peuvent effectuer des requêtes en utilisant le principe de l'appel de procédure distant (*Remote Procedure Call*, RPC en anglais) comme cela est montré en figure 5.2. Chaque requête donne lieu à un traitement particulier sur le serveur et retourne, de manière synchrone, le résultat du traitement. Les requêtes contiennent un identifiant unique propre à l'application, un type

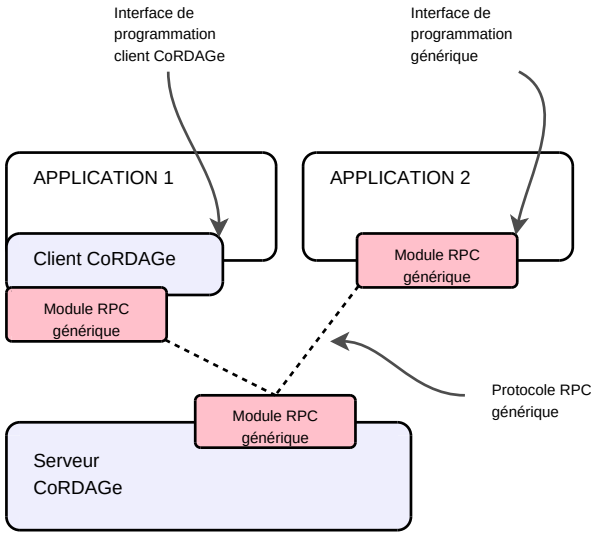

FIG. 5.2 – Interactions entre les applications et le serveur CoRDAGe.

Listing 5.3 – Exemple de requête XML-RPC.

```
<?xml version="1.0"?>
<methodCall>
  <methodName>cordage.execute</methodName>
  <params>
    <param>
      <value>
        <string>mon-parametre</string>
      </value>
    </param>
  </params>
</methodCall>
```

Listing 5.4 – Interface pour l'appel de procédure distante avec XML-RPC C++.

```
monClient.call(adresse_serveur,
               "execute",
               "s",
               &resultat,
               "mon-parametre");
```

d'action à effectuer ainsi que les paramètres associés. Nous présentons maintenant plus finement comment intégrer le client CoRDAGe dans une application.

5.3.1 Format des requêtes

Les interactions avec le serveur CoRDAGe s'effectuent sur le principe des appels de procédure distants (RPC). La couche de communication RPC prise en charge par des modules

Listing 5.5 – Utilisation de l'interface pour l'appel de procédure distante XML-RPC C++ dans le cadre de CORDAGE. La chaîne "sis" renseigne sur le type des paramètres successifs transmis à la procédure cordage.execute : s pour une chaîne de caractères et i pour un entier.

```
1 monClient.call(adresse_serveur,
2                "cordage.execute",
3                "sis",
4                &resultat,
5                "identifiant", 8, "foo␣2");
```

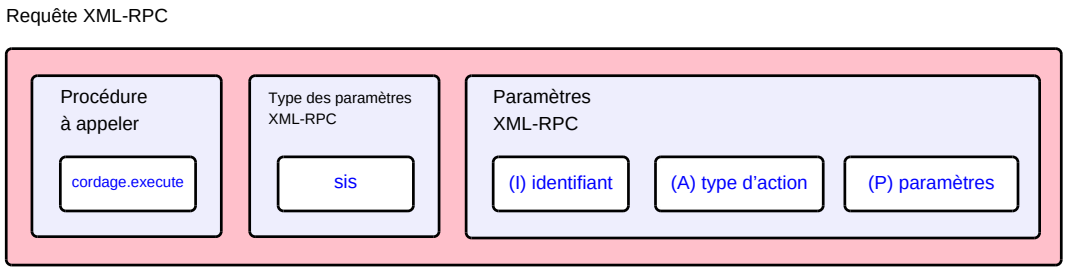

FIG. 5.6 – Encapsulation des requêtes CORDAGE au format XML-RPC. La seule procédure distante offerte par le serveur CORDAGE est cordage.execute. Elle attend une suite de trois paramètres : un identifiant, une numéro d'action et les paramètres associés.

génériques sur la figure 5.2 permet de masquer :

- Les appels à des procédures offertes par le serveur.
- L'envoi de données structurées et typées.

Plusieurs environnements de programmation ont été proposés pour offrir de tels services. C'est le cas notamment de CORBA [101], DCOM [92], XML-RPC [121] ou encore SOAP [127]. Les interactions avec le serveur CORDAGE ne nécessitent pas de concepts avancés. Nous avons donc choisi de baser les communications sur le système XML-RPC pour des raisons de simplicité [163].

XML-RPC est défini par une spécification permettant à des programmes distribués sur des ressources hétérogènes en terme de matériel et de logiciel, d'effectuer des appels de procédure distants. L'une des particularités de cet environnement est d'utiliser le format XML pour l'encapsulation des données et le protocole HTTP pour leur transport. Le listing 5.3 propose un exemple de requête XML-RPC générée pour demander l'exécution de la procédure cordage.execute avec un paramètre de type chaîne de caractères dont la valeur est 'mon-parametre'.

Différentes implémentations de XML-RPC permettent de rendre transparent la manipulation de ces requêtes XML. L'une des plus répandues d'entre elles, XML-RPC pour le langage C++ [162], offre une interface proposant une méthode call pour effectuer un appel distant, comme cela est montré dans le listing 5.4. Cette méthode prend en paramètre l'adresse du serveur sur lequel envoyer la requête, le nom de la procédure distante à appeler, le type des paramètres associés[2], un pointeur sur une variable pour stocker le résultat et enfin les paramètres associés (dans cet exemple il n'y en a qu'un).

[2]Ici le type est une chaîne de caractères, le code associé est "s"

Listing 5.7 – Définition des actions génériques.

```
1  #define CORK_BASE_TERMINATE 0
2  #define CORK_BASE_SET_APPDESCPATH -1
3  #define CORK_BASE_SRTI_SET_FRONTEND -2
4  #define CORK_BASE_DTI_SET_FRONTEND -3
5  #define CORK_BASE_SET_CTRLPATH -4
6  #define CORK_BASE_DEPLOY -5
7  #define CORK_BASE_VERSION -6
8  #define CORK_BASE_REGISTER -7
9  #define CORK_BASE_DTI_SET_PORT -8
10 #define CORK_BASE_DISCARD -9
11 #define CORK_BASE_ADD_SUB_APP -10
12 #define CORK_BASE_ENTITY_ACK -11
13 #define CORK_BASE_MONITOR -12
```

Les requêtes XML-RPC utilisées par CORDAGE portent toutes sur la fonction `cordage.execute` implémentée sur le serveur. Elles contiennent obligatoirement trois paramètres : *I*, *A* et *P*.

- `I` est une chaîne de caractères. Il est utilisé comme un identifiant permettant de retrouver la fiche correspondant à l'application appelante sur le serveur.
- `A` est de type entier. Il correspond au numéro d'action à effectuer.
- `P` est de type chaîne de caractères. Il contient la liste des paramètres de cette action CORDAGE, séparés par des espaces.

Le listing 5.5 montre un exemple de requête au format CORDAGE demandant le déclenchement de l'action numéro `8` avec les paramètres `foo` et `2`. La signification des numéros d'action et des paramètres est discutée en section 5.3.2. La figure 5.6 reprend le format global d'une requête XML-RPC pour CORDAGE dans laquelle seuls les champs `I`, `A` et `P` peuvent varier. Cette contrainte forte est assurée par la mise en place d'une sur-couche logicielle dont le rôle est de masquer aux applications l'utilisation de XML-RPC, comme cela est expliqué en section 5.3.3.

5.3.2 Actions et paramètres

Les requêtes CORDAGE effectuées par les applications sur le serveur contiennent chacune un numéro d'action à exécuter ainsi que des paramètres associés à cette action. Par exemple, dans le listing 5.5 le numéro de l'action demandée est `8` et les paramètres associés sont `foo` et `2`. Ces actions peuvent être de deux natures différentes.

Les actions génériques sont définies de telle sorte que leur exécution ait un sens, indépendamment du type de l'application appelante. Ces actions sont définies par CORDAGE et ne peuvent faire l'objet d'une modification sémantique pour s'adapter à l'application.

Les actions spécifiques sont définies par le fait qu'elles n'aient un sens que dans un contexte particulier. Ces actions sont conçues par un développeur en charge de l'adaptation de CORDAGE à une application en particulier. La mise en œuvre des actions spécifiques sur le serveur est présentée en section 5.4.3.

Qu'elles soient génériques ou spécifiques, les actions sont identifiées par des numéros d'action. Ces numéros sont définis par des noms symboliques préfixés par la chaîne "*CORK*[3]". Ce préfixe est l'acronyme du noyau logiciel, il signifie CORDAGE *Kernel* en anglais. Les numéros d'actions sont obligatoirement uniques et doivent être négatifs ou nuls pour les actions génériques. Pour les actions spécifiques, ces numéros sont définis comme strictement positifs et uniques dans le contexte de chaque application. Plusieurs actions spécifiques à différentes applications peuvent donc partager un même numéro. Le listing 5.7 présente la liste des actions génériques offertes par CORDAGE. Elles peuvent être regroupées selon leur utilisation au sein de plusieurs catégories.

5.3.2.1 Déclaration

Lorsqu'une application désire profiter des services offerts par un serveur CORDAGE, celle-ci doit se déclarer afin d'y créer une *fiche de suivi*. Cette étape permet de renseigner deux informations importantes.

- La première information est l'identifiant de l'application, unique à l'instance de cette application et qui servira à retrouver la fiche de suivi correspondante. Cet identifiant peut être proposé par l'application, sous réserve de respecter la propriété d'unicité sur ce serveur ou sur le réseau de serveurs. Il peut aussi être imposé par le serveur s'il n'est pas spécifié. L'identifiant doit par la suite être systématiquement renseigné pour toute requête sur le serveur.
- La deuxième information est le type de l'application. Ce type doit faire partie de la liste des types d'application connus par le serveur contacté[4]. Le type permet d'instancier la fiche de suivi avec les actions spécifiques à l'application concernée.

L'action de déclaration est effectuée en envoyant la requête

```
{identifiant, CORK_BASE_REGISTER, type}
```

avec `identifiant` l'identifiant suggéré pour cet application et `type` le type de cette application. Cette requête retourne l'identifiant retenu ou un code d'erreur sinon.

Le modèle proposé pour CORDAGE permet la gestion d'applications composées de plusieurs sous-applications. Il peut s'agir par exemple d'une organisation multi-intergiciel. La déclaration d'une telle architecture logicielle est effectuée en déclarant de manière indépendante toutes les sous-applications puis en les ajoutant à l'application principale. Il est ainsi possible de construire un arbre de sous-applications en ajoutant des sous-applications aux sous-applications. Des liens orientés sont alors établis entre les fiches de suivi. Ils permettent par la suite de gérer certaines actions comme le déploiement de manière coordonnée. L'action d'ajout d'une sous-application est effectuée en envoyant la requête

```
{id-app, CORK_BASE_ADD_SUB_APP, id-sous-app}
```

avec `id-app` l'identifiant de l'application principale et `id-sous-app` l'identifiant de la sous-application. Cette requête retourne un code de succès.

[3] Le mot *Cork* fait référence à un codage informatique de caractères, nommé d'après une ville en Irlande. C'est aussi la traduction anglaise pour le bouchon à vin en liège.

[4] Actuellement les types d'application supportés sont JUXMEM-JXTA, JUXMEM 2, GFARM et DIET.

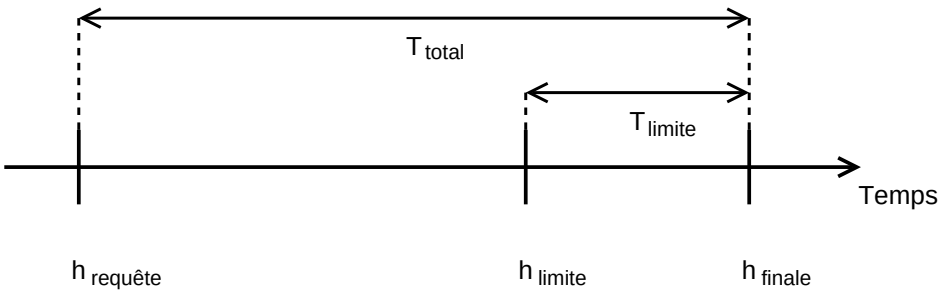

FIG. 5.8 – Calcul de l'heure limite et de l'heure finale à partir du temps total renseigné par l'application et le temps limite renseigné par les outils de réservation et de déploiement.

5.3.2.2 Configuration

Une fois l'application déclarée auprès du serveur, plusieurs interactions sont nécessaires afin de renseigner la fiche de suivi. La plus importante d'entre elles est la spécification du fichier contenant la description de l'application. Ce fichier doit contenir la description initiale de l'application à déployer. S'il n'est pas spécifié, une description minimale sera proposée par le serveur. L'action de spécification du fichier de description de l'application est effectuée en envoyant la requête `{id-app, CORK_BASE_SET_APPDESCPATH, lien-fichier}` où `lien-fichier` est un chemin sur un système de fichiers ou une adresse réseau. Il est aussi prévu de pouvoir envoyer directement la description de l'application en paramètre. Sur le même principe, la requête `{id-app, CORK_BASE_SET_CTRLPATH, lien-fichier}` permet de spécifier le fichier à utiliser pour lister notamment les contraintes de placement des entités sur les ressources physiques.

Les requêtes sur les actions `CORK_BASE_SRTI_SET_FRONTEND`, `CORK_BASE_DTI_SET_FRONTEND` et `CORK_BASE_DTI_SET_PORT`[5] renseignent sur les modalités d'accès aux outils de réservation et de déploiement. Ces informations sont propres à chaque fiche de suivi. Ainsi, même au sein d'une application composée de plusieurs sous-applications, il est possible d'utiliser des outils de réservation et de déploiement différents.

Les informations de configuration sont modifiables pendant toute la durée de vie de la fiche de suivi. Il est donc possible de changer d'outil de réservation entre deux déploiements par exemple.

5.3.2.3 Déploiement

Pour CORDAGE, le déploiement consiste en l'enchaînement de trois phases.

1. Une phase de construction de la représentation de l'application en un arbre logique.
2. Une phase de pré-planification dirigée en accord avec l'outil de réservation des ressources.
3. Une phase de coordination de l'outil de déploiement.

L'enchaînement de ces trois phases est déclenché par la requête de déploiement `{id-app, CORK_BASE_DEPLOY, temps-total}` avec `temps-total` T_{total} le temps pendant lequel

[5]SRTI signifie *Scheduling and Reservation Tool Interface* et DTI signifie *Deployment Tool Interface*

le déploiement doit être effectif. Ce temps, exprimé en minutes, joue un rôle important dans la gestion des réservations : il permet de calculer l'*heure de réservation finale* (h_{finale}) et l'*heure limite de réservation* (h_{limite}).

L'heure h_{finale} est obtenue en additionnant l'heure de réception de la requête initiale de déploiement $h_{requete}$ avec le temps total T_{total}. Toutes les réservations de ressources physiques effectuées dans le contexte de cette fiche de suivi seront valides jusque h_{finale} et ce, même pour les déploiements additionnels effectués par la suite. Il n'est pas prévu aujourd'hui de pouvoir modifier h_{finale} après le premier déploiement, même si cela peut se justifier dans certaines situations comme une mauvaise évaluation du temps total d'exécution de son application.

L'heure h_{limite} est obtenue en retranchant un temps limite T_{limite} à l'heure finale h_{finale}. Au delà de cette heure limite, il n'est plus possible d'effectuer de réservations. Cela se justifie par le fait que lorsque l'on se rapproche de l'heure finale, le temps requis pour effectuer les différentes phases de pré-planification et de déploiement devient trop important au regard du temps restant avant expiration des réservations. Ce temps T_{limite} est donc adapté en fonction des caractéristiques des outils de réservation et de déploiement.

La figure 5.8 récapitule les relations entre les heures limite et finale et les temps limite et total. Le déploiement d'une application composée de sous-applications entraîne le déploiement de toutes les sous-applications et ce, de manière récursive. Le temps total T_{total} est alors transmis à toutes les sous-applications de l'application déployée.

Une fois déployée, l'application ne peut plus être modifiée que par le biais de requêtes portant sur des actions spécifiques à l'application, à l'exception de la requête `{id-app, CORK_BASE_DISCARD, id-entité}` qui permet de supprimer une entité déjà déployée. L'action `CORK_BASE_TERMINATE` entraîne quant à elle la terminaison de toutes les entités ainsi que des sessions de réservation et de déploiement pour l'application qui en fait la demande. La fiche de suivi correspondante est détruite. Les liens avec d'éventuelles fiches représentant une application parente ou des sous-applications sont détruits. Le chaînage est reconstruit en attachant les sous-applications à l'application parente.

5.3.2.4 Surveillance

La surveillance est la quatrième catégorie d'actions génériques proposées par CORDAGE. La fonction principale est la possibilité de récupérer, pour chaque fiche de suivi, la représentation courante de l'application interne au serveur sous forme d'un arbre logique ainsi que la liste des réservations de ressources physiques. La requête associée est `{id-app, CORK_BASE_MONITOR, variable}` avec `variable` le nom de la variable d'état de la fiche de suivi à récupérer. Cette requête retourne la valeur de la variable demandée.

Il est aussi possible de renseigner le serveur sur le déploiement effectif des différentes entités et sur leurs interconnexions. L'information ainsi envoyée permet de mettre à jour la représentation interne de l'application en indiquant que telle entité est non seulement bien déployée et en cours d'exécution, mais aussi connectée à une autre entité. La requête correspondante, `{id-app, CORK_BASE_ENTITY_ACK, entité entité-rdv}` contient deux paramètres : `entité` est l'identifiant de l'entité confirmant son exécution et optionnellement `entité-rdv` l'identifiant de l'entité à laquelle elle est connectée.

Listing 5.9 – Interface offerte par le client CORDAGE.

```
/* ce constructeur prend en parametre un chemin vers un fichier de configuration */
cdgclient(string configuration_file);

/* indique si la configuration de la fiche de suivie est un succès */
bool success();

/* ajoute une sous-application à cette application */
string add_sub_app(cdgclient * sub_app);

/* déploie la description d'application courante */
string deploy(int walltime);

/* supprime une entité dont l'identifiant est donné en paramètre */
string discard(string id);

/* effectue une requête sur le serveur */
string perform_action(int action_type, string parameters);

/* invalide la fiche de cette application sur le serveur */
string terminate();
```

Listing 5.10 – Exemple de fichier de configuration du client CORDAGE.

```
# adresse pour contacter le serveur CoRDAGe
cordage_server_host=frontale.rennes.grid5000.fr
# identifiant proposé pour la fiche de suivie
cordage_assignment=identifiant-application
# type de l'application␣considéré␣codé␣sous␣forme␣d'un entier
cordage_app_type=3
# port de communication à utiliser pour se connecter à l'outil␣de␣déploiement
cordage_dti_port=36000
# temps total en minutes
cordage_walltime=5
```

Un programme utilitaire nommé `cordage-monitor` est livré avec la distribution de CORDAGE. Il permet de visualiser la représentation logique de l'application et la liste des réservations en utilisant un rafraîchissement paramétrable.

5.3.3 Interface de programmation

L'interface client offerte aux applications est la plus simple possible. L'objectif est de minimiser le travail à effectuer pour adapter une application à l'utilisation de CORDAGE tout en conservant une liberté de pilotage acceptable. Il faut notamment masquer l'utilisation des requêtes spécifiques à XML-RPC, pour des raisons de simplicité mais aussi afin d'abstraire cette couche de communication. Le principe de base est d'offrir la possibilité de demander au serveur l'exécution d'une *action* en spécifiant les *paramètres* associés. Le client CORDAGE propose donc une méthode, appelée `perform_action` et prenant en paramètres un numéro

Listing 5.11 – Exemple de client CORDAGE.

```
1  cdgclient* app1 = new cdgclient(conf_app1);
2  cdgclient* app2 = new cdgclient(conf_app2);
3
4  if (app1->success() && app2->success())
5  {
6    app2->perform_action(CORK_BASE_DTI_SET_PORT, "35000");
7
8    app1->add_sub_app(app2);
9
10   app1->deploy(walltime);
11
12   sleep(walltime*60);
13 }
14
15 delete(app2);
16 delete(app1);
```

d'action ainsi que les paramètres associés. Cette méthode retourne un résultat sous forme d'une chaîne de caractères. D'un point de vue technique, cette méthode est accessible en intégrant la bibliothèque cliente de CORDAGE lors de la phase d'édition de liens de la compilation de l'application. Cette bibliothèque est disponible dans les langages C et C++.

L'interface proposée par la bibliothèque cliente C++ est résumée dans le listing 5.9. Celle-ci définit un objet client CORDAGE dont les attributs contiennent toutes les informations indispensables pour se connecter et interagir avec le serveur. Ces informations peuvent être issues d'un fichier de configuration dont la localisation est passée en paramètre lors de la création de l'objet. Un exemple de fichier de configuration est donné dans le listing 5.10. Les informations de configuration ainsi collectées sont transmises au serveur afin de créer et paramétrer la fiche de suivi. Une fois l'objet créé, il est possible de vérifier si les étapes de connexion au serveur ainsi que de configuration de la fiche de suivi se sont bien passées en invoquant la méthode `success`. Les interactions avec le serveur s'effectuent alors en utilisant la méthode `perform_action` ou l'une des méthodes génériques proposées par l'interface.

Le listing 5.11 montre un exemple de code utilisant le client CORDAGE. Ce code est typique de ce que la personne en charge d'adapter une application doit insérer pour mettre en place les interactions avec le serveur. Dans cet exemple, deux clients sont créés, chacun en charge d'une application. L'application numéro 2 est déclarée comme sous-application de l'application numéro 1 en utilisant la méthode `add_sub_app`. L'ensemble est par la suite déployé pour un temps total défini en paramètre de la méthode `deploy`.

Chacun de ces appels rend un résultat sous forme de chaîne de caractères. Ce résultat est soit défini par la spécification fonctionnelle de CORDAGE dans le cadre de requêtes sur des actions génériques, soit, dans le cadre d'actions spécifiques, défini par la personne en charge d'adapter l'application. Le résultat est déterminé sur le serveur CORDAGE suite à l'exécution de l'action demandée, puis transmis en retour au client qui en a fait la demande.

5.4 Mise en œuvre du serveur CORDAGE

Le serveur CORDAGE a la charge de traiter les requêtes en provenance d'une ou plusieurs applications. Son rôle principal est de transformer les actions de haut niveau contenues dans ces requêtes en une liste d'opérations de bas niveau relatives à la gestion des ressources physiques et au déploiement des entités. Ces opérations doivent s'effectuer dans le cadre strict du modèle introduit dans le chapitre 4. Ce modèle prévoit, pour chaque application enregistrée, la mise en place d'une représentation logique utilisée pour caractériser l'état courant de l'application. Les actions peuvent déclencher l'altération de cette représentation qui est alors répercutée sur la topologie de l'application.

Le serveur CORDAGE actuellement développé est un serveur centralisé pouvant traiter en parallèle des requêtes issues de plusieurs clients. Le serveur est dit *à état* (*stateful server*) car il conserve des données relatives aux clients entre chacune de leurs connexions. Dans cette section nous montrons comment l'architecture logicielle du serveur permet de mettre en œuvre le modèle proposé par CORDAGE de manière modulaire et ouverte.

5.4.1 Architecture logicielle

L'architecture globale du serveur CORDAGE est organisée autour de quatre grands axes. Le premier axe est la réception et l'aiguillage des requêtes envoyées par les clients. Le second est la gestion de la représentation interne de l'application. Un troisième a trait à la mise en place des interactions avec les outils de réservations des ressources physique. Enfin, le quatrième se charge des interactions avec les outils de déploiement.

5.4.1.1 Vue de haut niveau

La figure 5.12 présente une vue générale de l'architecture logicielle du serveur. Les requêtes, prises en charge par une interface d'appel de procédures distant (RPC), sont envoyées à un noyau d'exécution pour appliquer l'action demandée sur la fiche de suivi correspondant au client. Cette fiche de suivi contient différentes variables d'état dont, notamment, la représentation logique de l'application. Ces variables peuvent être accédées par des méthodes implémentant les actions génériques communes à toutes les fiches ainsi que des actions spécifiques au type de l'application. La fiche contient aussi des liens vers d'autres fiches afin de former l'arbre des applications dans le cadre d'applications composées de sous-applications. Enfin, des modules interfaçant le serveur avec les outils de réservation et de déploiement sont instanciés au sein de chaque fiche.

5.4.1.2 Cheminement des requêtes

Les requêtes prises en charge par le serveur CORDAGE traversent plusieurs couches logicielles avant d'être aiguillées sur la bonne action. La figure 5.13 montre le cheminement possible d'une requête. Dans un premier temps, l'interface client reçoit les requêtes en provenance de diverses applications clientes. Ces requêtes sont traitées en parallèle, la couche de communication RPC utilisée ayant la possibilité de créer un processus léger pour chacun des appels reçus. La synchronisation est alors prise en charge par le serveur pour gérer la concurrence d'accès sur les variables partagées, notamment les fiches de suivi. Les

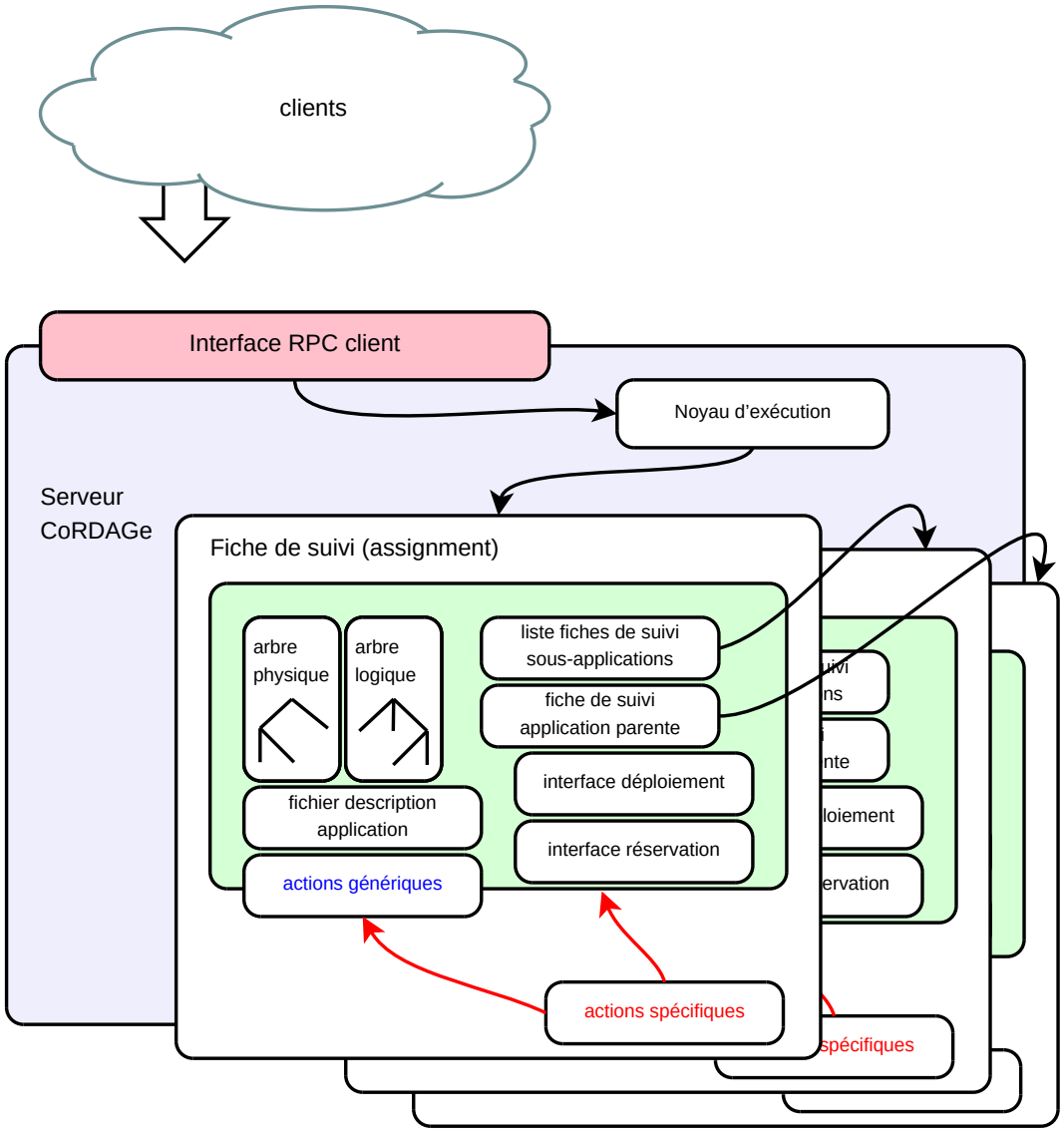

FIG. 5.12 – Une vue de haut niveau du serveur CoRDAGe.

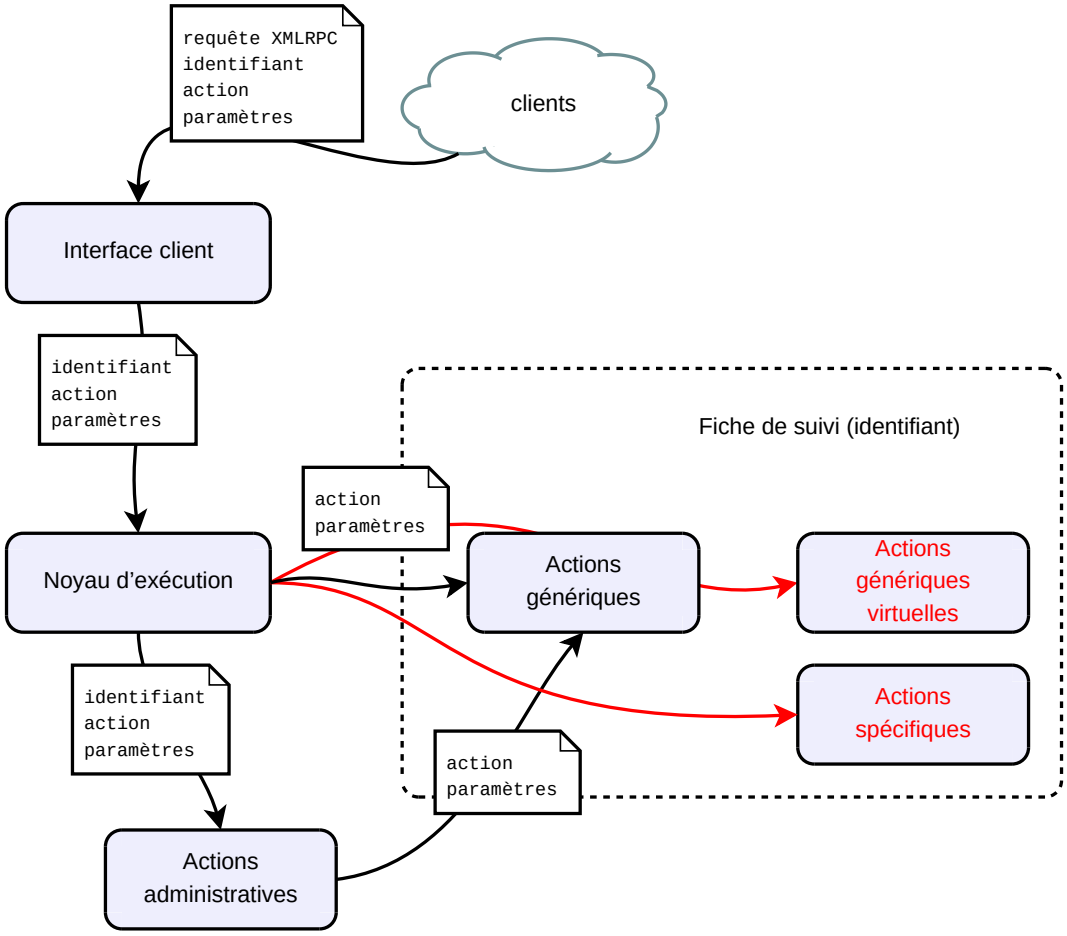

FIG. 5.13 – Cheminement des requêtes au sein du serveur.

requêtes, construites suivant le format présenté en section 5.3.1, sont décomposées par l'interface clients afin d'en extraire les trois informations pertinentes pour le serveur, à savoir 1) l'identifiant de la fiche de suivi sur laquelle travailler, 2) le numéro de l'action à effectuer et 3) les paramètres associés à cette action. Les actions peuvent être classées suivant quatre natures principales.

Les actions administratives sont utilisées pour la gestion des informations générales sur le serveur, pour la création ou la destruction de fiches de suivi. Ces actions peuvent nécessiter un identifiant de fiche si des opérations sont effectuées sur l'une d'entre elles par la suite. Les actions administratives ne peuvent être modifiées pour adapter CORDAGE à la gestion d'une application en particulier.

Les actions génériques sont appliquées sur une fiche de suivi avec une sémantique identique, quel que soit le type de l'application. L'action générique principale est l'action de déploiement de l'application. Les actions génériques n'ont pas besoin d'être modifiées pour adapter CORDAGE à la gestion d'une application en particulier. Elles peuvent par ailleurs effectuer des appels sur les actions génériques virtuelles mais pas sur les actions spécifiques car elles n'en connaissent pas l'existence.

Les actions génériques virtuelles sont des actions génériques ré-écrites par une personne en charge de l'adaptation d'une application à CORDAGE. Il n'est pas possible d'en modifier la sémantique ni la spécification, mais uniquement l'implémentation. Certaines de ces actions doivent être impérativement ré-écrites, comme la procédure de construction de la représentation logique à partir du fichier de description de l'application. Les actions génériques virtuelles peuvent faire appel aux actions génériques et spécifiques.

Les actions spécifiques sont des actions dont la sémantique, la spécification et l'implémentation sont propres à chaque type d'application. Elles sont complètement définies par le programmeur en charge de l'adaptation de CORDAGE à une application. Les actions spécifiques peuvent faire appel aux actions génériques et génériques virtuelles.

Contrairement aux actions administratives, les actions génériques, génériques virtuelles et spécifiques ont toutes accès aux variables d'état de la fiche de suivi sur laquelle elles sont appliquées.

5.4.2 Fiche de suivi

La fiche de suivi, aussi appelée *assignment* dans la terminologie CORDAGE, est un objet créé sur le serveur pour toute demande de prise en charge d'une application. Cette fiche contient des représentations logiques de l'application, des ressources physiques ainsi que des variables indiquant l'état courant de l'application déployée. La définition des fiches doit être suffisamment complète pour pouvoir exprimer toutes les informations contenues dans le modèle et ne permettre que les modifications autorisées. L'accès aux représentations logiques est effectué à l'aide de méthodes dont le but est de simplifier la tâche du programmeur extérieur.

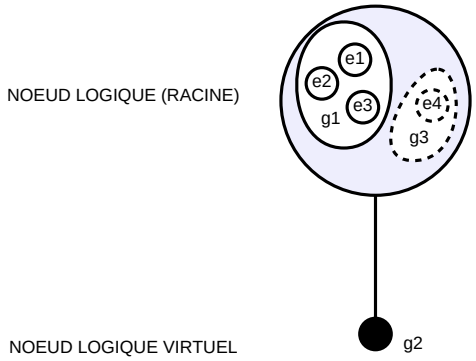

FIG. 5.14 – Exemple d'arbre logique. La version XML est donnée dans le listing 5.15.

Listing 5.15 – Exemple d'arbre logique au format XML.

```
<CORDAGE_LOGICAL_TREE>
  <logicaltree name="t1" physical="rennes">
    <logicalnode>
      <logicalgroup name="g1" jobid="1664" deployed="true" application="app1">
        <entity name="e1" cardinality="1" acked="true" connectedto="none"/>
        <entity name="e2" cardinality="2" acked="true" connectedto="e1"/>
        <entity name="e3" cardinality="1" acked="false" connectedto="none"/>
      </logicalgroup>
      <logicalgroup name="g3" jobid="-1" deployed="false" application="app2">
        <entity name="e4" cardinality="1" acked="false" connectedto="none"/>
      </logicalgroup>
    </logicalnode>
    <children>
      <logicaltree name="t2" physical="paramount">
        <logicalgroup name="g2" jobid="1665" deployed="false" application="app1"/>
      </logicaltree>
    </children>
  </logicaltree>
</CORDAGE_LOGICAL_TREE>
```

Listing 5.16 – Quelques-unes des 37 méthodes offertes pour accéder et modifier l'arbre logique.

```
1  /* creation */
2  void add_node(string name);
3  void add_group(string name);
4  void add_entity(string name, int cardinality);
5
6  /* deplacement dans l'arbre */
7  bool move_down(string name);
8  bool move_up();
9  void move_top();
10 bool is_virtual();
11
12 /* deplacement dans les groupes */
13 bool move_to_first_group();
14 bool move_next_group();
15 bool move_next_undeployed_group();
16 bool move_previous_group();
17 bool move_to_group(string name);
18
19 /* fonctions de haut niveau */
20 set <int> get_jobids_from_undeployed_groups();
21 void merge_tree(cdglogicaltree ** logtree);
```

5.4.2.1 Représentation de l'application

Suivant le modèle proposé par CORDAGE, la représentation de l'application est construite sous la forme d'un arbre logique. Les nœuds de cet arbre contiennent des groupes logiques, qui eux-mêmes contiennent des entités. D'un point de vue technologique, la représentation est mise en œuvre à travers des arbres logiques écrits en XML. Ces arbres sont définis par une grammaire au format XSD [6], présentée en annexe dans le listing A.1. Un exemple d'arbre logique respectant cette grammaire est donné au format XML dans le listing 5.15 et représenté de manière compréhensible pour un humain en figure 5.14.

L'arbre logique `logicaltree` est l'élément principal de la représentation. Il est composé d'une partie nœud logique `logicalnode`, d'une partie pour les enfants de ce nœud `children` et de deux attributs : son nom et son groupe physique. L'attribut `name` est utilisé pour identifier ce sous-arbre logique dans la représentation complète. Le deuxième attribut, `physical`, est un identifiant du groupe de ressources physiques sur lequel cet arbre logique est ou doit être projeté.

Le nœud logique `logicalnode` contient une séquence de groupes logiques `logicalgroup`, chacun de ces groupes ayant pour attribut 1) un identifiant de groupe `name`, 2) un identifiant pour la réservation `jobid` renseigné lors de la phase de projection, 3) un indicateur de déploiement `deployed` ainsi que 4) un identifiant d'application dans le cas où cet arbre est issu d'une fusion de plusieurs applications. L'indicateur de déploiement est placé à `vrai` lorsque l'outil de déploiement confirme que ce groupe a été déployé. Le

[6] XML Schema Definition

Listing 5.17 – Exemple d'arbre physique au format XML. Le champ `used` est utilisé pour indiquer si ce groupe de ressources héberge déjà un groupe logique pour cette application.

```
<CORDAGE_PHYSICAL_TREE>
  <grid name="grid5000" used="true">
    <site name="lille" used="true">
      <cluster name="chinqchinq" used="true"/>
    </site>
    <site name="rennes" used="true">
      <cluster name="paramount" used="false" />
      <cluster name="paraquad" used="true" />
    </site>
  </grid>
</CORDAGE_PHYSICAL_TREE>
```

groupe est alors verrouillé et il n'est alors plus possible d'y effectuer des modifications comme ajouter ou enlever des entités.

L'entité `entity` est l'élément de base du modèle CORDAGE. Dans cette description, elle est caractérisée par quatre éléments : l'identifiant de l'entité `name`, le nombre d'instances `cardinality`, un indicateur d'acquittement `acked` et une information de connectivité `connectedto`. L'indicateur d'acquittement est placé à `vrai` lorsque l'entité ou une entité mandataire a envoyé une confirmation de déploiement et de démarrage au serveur. Ce message peut aussi contenir, à titre informatif, l'identifiant d'une deuxième entité à laquelle elle est connectée. Ces informations sont utilisées à des fins de surveillance.

Les enfants `children` contiennent une séquence d'arbres logiques. Ils permettent de construire de manière récursive l'arbre complet.

La représentation logique de l'application au format XML ne peut être manipulée directement par les actions de la fiche de suivi. Ceci a été rendu impossible pour trois raisons.

- Garantir que chaque opération sur l'arbre rende en résultat un arbre qui respecte la grammaire XSD.
- Masquer et abstraire la mise en œuvre.
- Prendre en compte que la modification d'un arbre XML à partir d'un langage de programmation comme le C++ est terriblement pénible pour tout programmeur normalement constitué. Or les personnes en charge d'adapter une application à CORDAGE doivent pouvoir interagir avec la représentation interne de leur application de la manière la plus simple et la plus intuitive possible.

La représentation logique XML est donc encapsulée dans un objet offrant des accesseurs permettant de naviguer, lire ou modifier l'arbre tout en garantissant le respect de la grammaire XSD. Il y a 37 méthodes d'accès à l'arbre au total. Les principales sont décrites dans le listing 5.16. La séquence d'opérations permettant de construire l'arbre donné en exemple en figure 5.14 est composée d'appels aux méthodes `add_group`, `add_entity` et `add_node`.

5.4.2.2 Représentation des ressources physiques

Chaque instance de la fiche de suivi contient une représentation des ressources physiques. Cette représentation des ressources physiques est utilisée dans la phase de projection

Listing 5.18 – Quelques-unes des méthodes d'accès à la représentation des ressources physiques.

```
1  /* creation */
2  void add_grid(string name);
3  void add_site(string name);
4  void add_cluster(string name);
5
6  /* navigation */
7  void move_top();
8  bool move_down(string name);
9  bool move_up();
10
11 /* consultation */
12 bool is_used();
13 set string get_cluster_list();
```

pour recevoir les groupes logiques de la représentation de l'application.

La représentation des ressources physiques présentée dans la section 4.2.4 prévoit d'organiser les ressources en groupes hiérarchiques. Ces groupes sont obtenus suivant un critère particulier visant à rapprocher les ressources qui le partagent. Le critère retenu pour la mise en œuvre porte sur la distance réseau, en terme de latence et de débit, que l'on peut trouver dans une plate-forme de type grille de calculateurs. Cette distance est principalement tributaire de l'emplacement physique de la ressource dans l'infrastructure. Ce choix est motivé par l'importance du placement des entités communicantes sur une plate-forme d'exécution distribuée et ce, pour deux raisons principales : 1) le besoin de communiquer de manière efficace et 2) le besoin de placer deux entités dans des lieux éloignés pour augmenter la tolérance aux défaillances d'un site.

La technologie employée pour représenter les ressources est la même que celle utilisée pour représenter l'application. Une grammaire au format XSD définit l'ensemble des arbres au format XML valides pour représenter les ressources. La grammaire est donnée en annexes dans le listing A.2. L'exemple d'arbre physique donné dans le listing 5.17 montre clairement l'organisation hiérarchique des ressources, caractéristique des grilles de calculs. Les groupes physiques partent de la grille jusqu'aux grappes de calculateurs. La profondeur de la description a été limitée à la grappe et ne traite pas des noeuds, processeurs et cœurs. Ce choix est motivé par des raisons de simplicité. Il n'est pas pertinent de considérer un niveau de détail plus important. De plus, ce choix conditionne le niveau de détails de la phase de pré-planification, c'est-à-dire la projection de l'arbre logique sur l'arbre physique. Si les ressources physiques sont décrites en profondeur, la pré-planification du déploiement tend à devenir une planification complète.

La représentation physique est construite par le module de gestion des ressources du serveur CORDAGE à partir des informations mises à disposition par l'outil de réservation. Comme pour la représentation de l'application, il n'est pas possible de manipuler directement l'arbre XML. Celui-ci est encapsulé dans un objet proposant des méthodes pour sa modification et sa consultation. Quelques-unes des méthodes d'accès à cet objet sont décrites dans le listing 5.18.

L'implémentation actuelle ne permet pas à cette représentation, une fois construite, d'être modifiée pendant l'exécution de l'application. Une telle propriété serait pourtant intéressante dans le cadre d'architecture dynamique, où des ressources peuvent apparaître et disparaître. Cela demanderait la surveillance ou la notification, par l'outil de réservation ou le système d'information de la grille, des événements matériels survenant sur la plate-forme. Par ailleurs, la mise en œuvre présentée ici fixe la définition des groupes physiques : grille, site et grappe. Là aussi il serait intéressant de considérer une approche de plus haut niveau sous forme d'un arbre générique pouvant s'adapter à d'autres critères d'organisation des ressources physiques. Ces deux pistes n'ont pas été explorées dans ce travail.

5.4.2.3 Autres variables d'état

La fiche de suivi contient, en plus des représentations logiques et physiques, diverses variables d'état non spécifiques aux applications. Ces variables donnent accès à la localisation de la description de l'application, à des pointeurs vers les modules d'interfaçage avec l'outil de réservation et l'outil de déploiement, ou encore à des objets de synchronisation. Il est possible d'attacher des sous-applications à une fiche de suivi en utilisant une liste de pointeurs vers d'autres fiches de suivi. Un pointeur vers la fiche parent est aussi disponible afin de naviguer dans l'arbre des sous-applications.

L'ensemble de ces variables, à l'exception des objets de synchronisation, sont accessibles par toutes les actions de la fiche de suivi, quel que soit le type de l'action. Des accesseurs sont fournis afin de diminuer le travail du programmeur et limiter le risque d'atteindre un état incohérent. Ces accesseurs, ainsi que les méthodes et autres actions offertes par les fiches de suivi sont présentés dans la section suivante.

5.4.3 Actions et fonctions extensibles

Les actions proposées par CORDAGE sont, comme il est indiqué dans la section 5.4.1.2, de quatre natures différentes. Selon leur nature, les actions sont mises en œuvre dans le noyau d'exécution ou directement dans les fiches de suivi : les actions administratives sont implémentées dans le noyau d'exécution, les actions génériques dans la fiche de suivi de base, les actions génériques virtuelles et les actions spécifiques dans les *fiches de suivi dérivées*. La fiche de suivi de base contient les variables d'état ainsi que toutes les méthodes implémentant les actions communes à toutes les fiches de suivi, quel que soit le type de l'application. La fiche de suivi dérivée étend cette fiche de base en ajoutant ou en sur-définissant des variables et des méthodes. Elle est utilisée pour adapter CORDAGE aux spécificités des applications prises en charge. Dans cette section nous présentons les fonctions et méthodes utilisées pour implémenter les différents types d'actions.

5.4.3.1 Actions administratives

Les actions administratives permettent d'effectuer des opérations générales sur le serveur ainsi que de gérer les fiches de suivi. Ces actions sont mises en œuvre dans le noyau d'exécution et implémentées par la classe *CoRK* (pour CORDAGE Kernel).

Une méthode principale nommée `perform_action` permet d'appliquer le traitement adéquat pour chaque requête reçue par le serveur. Cette méthode prend en paramètre un identifiant de fiche de suivi, un numéro d'action ainsi que les paramètres associés à cette action et rend le résultat du traitement. Cette fonction intercepte toutes les demandes d'actions administratives, génériques et génériques virtuelles. Les actions ne tombant pas dans cette catégorie, comme les actions spécifiques, sont transmises à la fiche de suivi concernée. Ceci est effectué en appelant la méthode du même nom, `perform_action`, définie par chaque fiche de suivi.

L'ajout de fiche de suivi est effectué grâce à la fonction `register_assignment` dont l'appel est déclenché par l'action `CORK_BASE_REGISTER`. Cette fonction prend en paramètre un identifiant de fiche de suivi ainsi que le type de l'application pour laquelle cette fiche doit être créée. La fiche de suivi dérivée correspondant au type de l'application est ainsi instanciée et ajoutée dans la liste des fiches de suivi à gérer par ce serveur. Les fiches peuvent par la suite être retrouvées grâce à une fonction de hachage sur leur identifiant.

La déclaration d'une sous-application est rendue possible par l'utilisation de l'action `CORK_BASE_ADD_SUB_APP`. Cette action prend en paramètre l'identifiant de la fiche de suivi à ajouter en tant que sous-application de la fiche de suivi concernée. Les deux fiches sont recherchées dans la liste des fiches gérées par le serveur et notifiées de leur nouveau lien de parenté.

La suppression d'une fiche de suivi est effectuée grâce à l'action `CORK_BASE_TERMINATE`. Cette action demande la terminaison des sessions de réservation et déploiement auprès des outils utilisés par cette fiche et la supprime de la liste des fiches traitées par le serveur.

Les informations sur le serveur comme son numéro de version ou la valeur des variables d'état des fiches sont obtenues grâce à diverses actions de surveillance.

5.4.3.2 Actions génériques

Les actions génériques sont communes à toutes les fiches de suivi. Elles permettent de consulter ou modifier l'état interne de la fiche avec éventuellement le déclenchement d'interactions avec les outils de réservation et déploiement. Les actions génériques sont mises en œuvre au sein de la fiche de base et implémentées par la classe *CoRKBase*. Les trois principales actions génériques sont la fonction de déploiement, la fonction de rétraction des entités et la fonction de projection.

La fonction de déploiement *deploy* est déclenchée par l'action `CORK_BASE_DEPLOY`. Elle a la charge d'organiser le déploiement, depuis la construction de la représentation logique jusqu'aux interactions avec l'outil de déploiement. Le listing 5.19 décrit le pseudo-code de cette fonction. L'idée principale est de déclencher, récursivement, la génération des représentations logiques des sous-applications en rendant en résultat la fusion de ces représentations. La fonction de génération de la représentation logique est décrite en section 5.4.3.4. La fiche de suivi correspondant à l'application racine ayant déclenché le déploiement se charge ensuite de projeter l'arbre logique obtenu sur l'arbre des ressources physiques avant de demander le déploiement effectif des entités.

Listing 5.19 – Pseudo-code de la fonction de déploiement de la fiche de suivi de base.

```
1  fonction deploiement ()
2  {
3    // generation de la représentation logique
4    // à partir du fichier de description
5    // de l'application
6    genere_representation_logique(description_application);
7
8    // pour toutes les fiches de suivi correspondant aux sous-applications
9    // de cette application
10   pourtout(SA appartenant aux sous-applications) {
11     // demande récursive de construction de la représentation logique
12     SA->deploiement();
13     // fusion avec notre propre représentation logique
14     arbre_logique->fusion(SA->arbre_logique);
15   } // fin de pourtout
16
17   // si nous sommes la racine de l'arbre des sous-applications
18   si (racine) alors {
19     // projection de l'arbre logique sur l'arbre physique
20     projection(arbre_logique, arbre_physique);
21     // ajout des contraintes de placement ainsi obtenues
22     outil_deploiement->ajout_contraintes_placement(arbre_logique);
23     // demande de déploiement
24     outil_deploiement->deploiement();
25   }
26
27 }
```

Listing 5.20 – Pseudo-code de la fonction de projection de la fiche de suivi de base.

```
1  booleen fonction projection(noeud_logique_courant, groupe_physique_courant)
2  {
3    booleen succes = vrai;
4
5    /* recupère tous les noeuds logiques enfants non déployés */
6    noeuds_logiques = recuperer_liste_enfants(noeud_logique_courant);
7
8    tantque (!vide(noeuds_logiques)) {
9
10     /* recupère tous les groupes physiques enfants non-utilisés */
11     groupes_physiques = recuperer_liste_enfants(groupe_physique_courant);
12
13     /* extrait un noeud de la liste */
14     nl = extrait(noeuds_logiques);
15
16     succes = faux;
17     tantque (!succes && !vide(groupes_physiques)) {
18       /* extrait un groupe de la liste */
19       gp = extrait(groupes_physiques);
20       succes = projection(nl, gp);
21     } // fin tantque
22
23   } // fin tantque
24
25   /* pour tous les groupes logiques non déployés
26      appartenant au noeud logique courant.. */
27   pourtout (gl appartenant noeud_logique_courant) {
28     /* reservation des ressources physiques */
29     succes = succes
30       && outil_reservation->demande_ressources(gl->nombre_entites(),
31                                                groupe_physique_courant);
32   } // fin pourtout
33
34   si (succes) {
35     /* mettre à jour les informations de réservation et déploiement
36        pour le noeud logique courant et le groupe physique courant */
37   } sinon {
38     /* défaire les réservations dans la liste noeuds_logiques */
39   } // fsi
40
41   retourne succes;
42 }
```

La fonction de rétraction permet de supprimer une entité précédemment déployée. La suppression est déclenchée par l'action `CORK_BASE_DISCARD` indiquant en paramètre l'identifiant de l'entité à enlever. Celle-ci est supprimée - 1) de la description de l'application en faisant un appel à une fonction générique virtuelle proposée par la fiche de suivi, - 2) de la liste des contraintes de placement, ainsi que - 3) de la représentation logique. Si le groupe logique contenant cette entité devient vide, il est lui aussi supprimé. Sa suppression entraîne l'annulation de la réservation des ressources physiques qui lui sont associées. Si, à la suite de cette suppression, le nœud logique contenant ce groupe devient virtuel, qu'il n'est pas la racine et n'a pas d'enfants, alors ce nœud est supprimé.

La fonction de projection est utilisée par la fonction de déploiement afin d'effectuer une pré-planification. Le but est d'associer à chacun des nœuds logiques de l'arbre un groupe physique de l'arbre des ressources, en respectant l'organisation hiérarchique de ces groupes.

Le listing 5.20 présente un pseudo-code très simplifié de l'algorithme de projection. Cet algorithme prend en paramètre le nœud logique courant dans l'arbre logique de l'application ainsi que le groupe physique courant dans l'arbre des ressources. Il tente de projeter de manière récursive chacun des fils du nœud logique courant sur différents fils du groupe physique courant. La projection du nœud logique courant sur le groupe physique courant est validée si :

- tous les fils du nœud logique courant ont pu être projetés sur les fils du groupe physique courant.
- il est possible de réserver suffisamment de ressources dans le groupe physique courant pour toutes les entités du nœud logique courant.

L'algorithme implémenté dans CORDAGE est beaucoup plus complet et prend en charge l'existence de nœuds virtuels (c'est-à-dire qui ne contiennent pas de groupes logiques), les contraintes de placement forcé sur des groupes physiques particuliers dans le cas d'un déploiement additionnel ou d'un co-déploiement, la libération des ressources réservées dans les sous-arbres lors d'un échec local ainsi que la projection à un niveau supérieur dans l'arbre des ressources physiques en cas d'échec général.

Cet algorithme construit la solution de projection au fur et à mesure que les ressources physiques sont réservées avec succès. Cette approche est dictée par l'aspect dynamique et multi-utilisateur de l'outil de réservation. Nous prenons en compte qu'il n'est pas possible de récupérer une vue globale de la disponibilité des ressources qui soit valable pendant tout le temps de la projection. Une telle possibilité impliquerait d'empêcher la création de nouvelles réservations, c'est-à-dire de verrouiller l'outil de réservation, ce qui n'est pas envisageable dans ce contexte.

Des actions génériques sont aussi proposées pour générer des identifiants pour les fiches de suivi ou les entités, masquer les interactions avec les outils de réservation et de déploiement ou encore mettre à jour la représentation logique de l'application, dans le cadre d'un acquittement d'entité par exemple.

5.4.3.3 Limites de la fonction de projection

Bien que opérationnel, l'algorithme de projection présente de nombreuses limites en terme d'optimalité de la solution ainsi qu'en terme de performance. Si l'objectif de ce travail n'est pas de fournir le meilleur algorithme de projection, nous pouvons tout de même

proposer quelques pistes pour son amélioration. La première piste part du constat que le temps écoulé dans l'algorithme provient en grande partie des interactions avec l'outil de réservation et plus spécifiquement de l'attente de la disponibilité effective des ressources. L'idée est donc de rendre cet algorithme parallèle en profitant de la récursivité sur les fils du nœud logique courant. Cette technique ne devrait pas surcharger l'outil de déploiement, déjà soumis à la concurrence des accès dans un environnement multi-utilisateur. Le gain devrait être obtenu en parallélisant les phases de réservations et celles d'attente de disponibilité effective des ressources.

Une deuxième piste est la définition d'une meilleure technique de sélection du groupe physique pour un nœud logique. L'algorithme actuel ne tient pas compte de la taille des nœuds logiques en terme du nombre d'entités. Il ne tient pas compte non plus de la taille des groupes physiques en terme du nombre de ressources physiques disponibles. Cette approche peut entraîner la projection d'un *petit* nœud logique sur un *grand* groupe physique et empêcher par la suite de trouver une solution pour la projection d'un *grand* nœud logique. Une solution simple consiste à trier les groupes logiques et les groupes physiques selon leur taille. La projection est alors effectuée en commençant par les groupes logiques de grande taille.

5.4.3.4 Actions génériques virtuelles

Les actions génériques virtuelles sont des actions génériques dont la mise en œuvre peut changer à condition de ne pas modifier la spécification. Cette mise en œuvre est spécifique à l'application et doit être fournie pour chaque type d'application supporté. Elles sont implémentées dans la fiche de suivi dérivée associée à l'application. Les trois actions génériques virtuelles portent sur la génération de la représentation logique de l'application, sur la gestion des requêtes spécifiques et sur la rétraction des entités.

La génération de la représentation logique est une fonction appelée par la fonction de déploiement présentée en section 5.4.3.2. Cette fonction produit un arbre logique représentant l'application à déployer à partir de la description de l'application contenue dans la fiche de suivi. La construction de l'arbre s'effectue grâce à l'utilisation de l'objet mettant en œuvre l'arbre logique (voir la section 5.4.2.1). Cet objet est fourni par la fiche de suivi.

La gestion des requêtes spécifiques consiste en la fonction `perform_action` ayant la charge de déclencher les traitements spécifiques correspondant au numéro d'action passé en paramètre. Ces actions sont obligatoirement des actions spécifiques, les autres types d'actions ayant déjà été pris en charge par le noyau d'exécution ou la fiche de base. La fonction `perform_action` recense donc toutes les actions spécifiques offertes pour ce type d'application et fait le lien avec les fonctions spécifiques introduites par la suite. Elle rend le résultat défini par le programmeur de l'action spécifique.

La rétraction d'une entité doit être en partie prise en charge de manière à modifier la description de l'application en supprimant l'entité choisie. Cette modification sert ensuite à notifier l'outil de déploiement que cette entité n'est plus déployée. La fonction associée s'appelle `remove_entity`. Elle est déclenchée par l'action générique de rétraction.

Les trois actions génériques virtuelles représentent le travail minimum à effectuer pour adapter CORDAGE à la gestion d'une application. Afin d'obtenir une plus large gamme de

comportements possibles, la personne en charge de l'adaptation doit spécifier des actions et des fonctions plus spécifiques à l'application.

5.4.3.5 Actions spécifiques

Les actions spécifiques sont des actions dont la sémantique et la mise en œuvre sont laissées à la charge du programmeur. Une action spécifique est caractérisée par un numéro d'action, par le format de ses paramètres associés ainsi que le format de son résultat. L'implémentation des actions spécifiques est effectuée par des fonctions ajoutées dans la fiche de suivi dérivée associée à l'application. Le processus d'ajout d'une action spécifique est constitué des étapes suivantes.

L'ajout d'un numéro d'action dans la liste des numéros d'action déjà déclarés au serveur. L'action doit être prise en charge par la fonction générique virtuelle `perform_action` qui va décomposer les paramètres et invoquer les méthodes spécifiques. Par exemple une action spécifique au contexte pair-à-pair : `CORK_APP_AJOUTE_PAIR` avec comme paramètre un entier n, ayant pour sémantique l'ajout de n pairs dans le réseau actuel et comme résultat le nombre de pairs effectivement ajoutés.

L'ajout de méthodes spécifiques afin de réaliser les actions. Par exemple une fonction qui crée un nouveau groupe logique dans l'arbre logique courant et y ajoute n entités. Cette fonction traduit l'action spécifique en un résultat générique sur l'arbre logique.

Les exemples donnés ci-dessus restent cependant très abstraits. C'est pourquoi une étude de cas mettant en jeu des applications réelles est proposée dans le chapitre 6.

5.4.4 Interactions avec les couches basses

Les modules de gestion des couches basses de CORDAGE ont la charge de masquer toutes les interactions avec les outils de réservation et de déploiement. Ces modules peuvent s'interfacer avec différents types d'outils respectant les contraintes présentées dans la section 5.1. Ils permettent à CORDAGE d'être utilisé sur différentes plates-formes d'exécution. L'interface offerte par ces modules est définie par la spécification de CORDAGE, seules les implémentations peuvent changer en fonction de l'outil visé. L'implémentation actuelle repose sur l'outil de réservation OAR et sur l'outil de déploiement ADAGE, mis en place sur GRID'5000. Un module de réservation ainsi qu'un module de déploiement sont instanciés au sein de chaque fiche de suivi. Ils sont pilotés à partir des actions génériques, principalement les fonctions de projection et de déploiement. Ils peuvent, si nécessaire, être accédés par des actions spécifiques. Ce comportement n'est pas souhaité, notamment parce qu'il compromet l'objectif d'abstraction de la gestion des ressources et du déploiement.

5.4.4.1 Module de gestion des ressources

Le module de gestion des ressources offre des primitives simples adaptées aux besoins de CORDAGE. Les principales primitives sont proposées dans le listing 5.21. La réservation s'effectue grâce à la méthode `reserve` prenant en paramètre le nombre de ressources ainsi que l'identifiant du groupe physique auquel elles doivent appartenir. Le temps de réservation n'est pas renseigné car il est calculé lors de la demande de déploiement, comme cela

Listing 5.21 – Interface outil de réservation.

```
1  /* effectue une reservation et rend le numéro de reservation */
2  int reserve(int nb_ressources, string nom_groupe_physique);
3
4  /* supprime une reservation */
5  void drop(int num_reservation);
6
7  /* attend que les ressources soient disponibles (synchrone) */
8  bool wait_for_job_ready(int num_reservation);
9
10 /* génère l'arbre des ressources physiques */
11 void generate_topology(physicaltree ** arbre_physique);
```

est expliqué en section 5.3.2.3. Cette méthode rend un numéro de réservation utilisé par la suite pour caractériser cet ensemble de ressources. La méthode `drop` permet d'annuler une réservation à partir de ce numéro. Celle-ci est notamment utilisée lors de la phase de projection, lorsqu'une solution en cours de construction s'avère être une impasse ou encore, à la demande de l'application lors de la terminaison de la fiche de suivi.

Dans des outils de réservation tels que OAR, il existe un délai significatif entre la date de lancement de la procédure de réservation des ressources et la date effective de mise à disposition des ressources. Ce délai s'explique principalement par des opérations de configuration des ressources choisies avant d'en permettre l'accès à l'utilisateur. Cette particularité est prise en compte par la méthode `wait_for_job_ready` permettant, de manière synchrone, d'attendre la mise à disposition effective des ressources. Cette méthode est basée sur le principe de l'attente active : le module demande périodiquement si les ressources sont prêtes puis se place dans la file des processus endormis pendant un temps donné. Après un certain nombre d'essais infructueux, cette méthode retourne un résultat indiquant que le temps d'attente a expiré. Le temps d'attente entre deux essais ainsi que le nombre d'essais sont des constantes attachées au module de gestion des ressources.

Enfin, une méthode est fournie par le module afin de construire l'arbre des ressources physiques à partir des informations fournies par la plate-forme d'exécution. Ces informations peuvent être données par l'outil de gestion des ressources, par un système d'information de la grille ou par une description statique contenue dans un fichier. L'arbre est construit en utilisant l'interface proposée en section 5.4.2.2. La méthode `generate_topology` est invoquée lors de la création de la fiche de suivi et donc de l'instanciation du module de gestion des ressources.

L'implémentation courante basée sur l'outil de réservation OAR utilise des scripts écrits dans le langage PERL. Ces scripts ont la charge d'effectuer l'analyse syntaxique de la sortie de l'interface de commande OAR afin d'extraire les informations pertinentes comme le numéro de réservation ou l'état des ressources. L'interaction avec OAR s'effectue grâce à une connexion de type SSH sur la machine frontale spécifiée par l'utilisateur.

Limites du module de gestion des ressources. Le module de gestion des ressources permet à CORDAGE de manipuler les réservations de manière transparente grâce à l'utilisation de primitives simples. Cependant, des interactions plus fines avec les outils de réservation

Listing 5.22 – Interface outil de déploiement.

```
1 /* méthode générique pour intéragir avec l'outil */
2 string set_param(int action, string paramètres);
3
4 /* retourne le nom de la ressource sur laquelle
5    une entité donnée est déployée */
6 string get_host_from_entity(string entité);
7
8 /* ajoute une contrainte de placement */
9 void add_placement_constraint(string entité, string groupe_physique);
```

sont souhaitables. Une première piste est l'élimination du problème de l'attente active en permettant à CORDAGE d'être notifié par l'outil de réservation de la disponibilité des ressources. Cette notification peut s'effectuer sous forme de requêtes CORDAGE d'une certaine classe, dédiées aux interactions avec l'outil de réservation. Une telle conception demande cependant d'adapter ces outils à l'utilisation de CORDAGE pour l'envoi des requêtes, ce qui n'est pas forcément envisageable dans tous les environnements. Des travaux préliminaires sont actuellement en cours avec la conception d'une interface de type *Web Services* permettant d'interroger l'état de la base de données utilisée par OAR. Ces travaux sont menés en collaboration avec Pascal Morillon[7] et David Margery[8] de l'équipe GRID'5000.

Une autre piste d'amélioration est le traitement des réservations par lot : l'algorithme de projection actuellement implémenté associe pour chaque groupe logique un numéro de réservation de ressources. Ceci est aussi valable pour les groupes logiques appartenant à un même nœud logique. Il y a donc au minimum autant d'interactions avec l'outil de réservation qu'il y a de groupes logiques à projeter. Afin de limiter le nombre des interactions à l'échelle du nœud logique, il serait intéressant de pouvoir exprimer une demande de réservation portant sur un même groupe physique, mais composée de plusieurs groupes de ressources. Ces groupes de ressources doivent être caractérisés par des numéros de réservation différents. Cette propriété demande là encore l'extension des contraintes liées au fonctionnement des outils de réservation, ce qui n'est pas forcément envisageable.

5.4.4.2 Module de gestion du déploiement

Le module de gestion du déploiement est, comme le module de gestion des ressources, une interface visant à masquer les interactions avec les outils offerts par la plate-forme d'exécution. Le module est instancié au sein de chaque fiche de suivi et est accessible par toutes les méthodes génériques et spécifiques. Son interface offre des primitives permettant de s'adapter au mieux à différents outils de déploiement. L'implémentation actuelle est basée sur ADAGE. Le listing 5.22 présente quelques-unes de ces primitives d'accès. Ces primitives conservent leur sémantique quel que soit l'outil de déploiement, seule la mise en œuvre change.

La méthode `set_param` permet de laisser une grande liberté d'expression dans le choix des actions à effectuer. La liste des actions est définie pour chaque type d'outil de déploie-

[7] Ingénieur d'études à l'Université de Rennes 1.
[8] Ingénieur de recherche à l'INRIA.

ment. Dans le contexte d'ADAGE, cette méthode est utilisée pour configurer l'environnement de déploiement en spécifiant les fichiers de description de l'application, les paramètres de contrôle, les ressources physiques disponibles, récupérer des informations sur le statut de l'application ou encore donner des ordres de déploiement.

La méthode `get_host_from_entity` permet de connaître la ressource physique sur laquelle une entité donnée est effectivement déployée. Cette information n'est pas connue par CORDAGE car la phase de projection n'effectue qu'une planification partielle. Cette méthode permet donc d'obtenir, de manière plus fine, la planification précise. Elle est utilisée dans le cadre du déploiement additionnel imposant des contraintes de placement en terme de proximité avec une entité déjà déployée. Enfin, la méthode `add_placement_constraint` permet de spécifier une contrainte de placement associant une entité particulière avec un groupe physique.

L'implémentation fournie pour le support d'ADAGE se base sur un ensemble de scripts écrits dans le langage PERL effectuant des appels locaux au client ADAGE et analysant le résultat retourné par ce client. Le client ADAGE a ensuite la charge d'interagir avec le serveur ADAGE distant.

5.5 Conclusion

L'architecture proposée dans ce chapitre se traduit par un prototype de recherche. Ce prototype est utilisé pour la validation du modèle introduit dans le chapitre 4. Cette mise en œuvre reprend les grandes lignes de la vision CORDAGE, à savoir la possibilité donnée aux applications déployées de modifier leur configuration de manière coordonnée et transparente pour l'utilisateur. Pour cela, une architecture client-serveur a été proposée pour interfacer les applications avec les outils de gestion des ressources et du déploiement. L'interface se veut la moins intrusive possible afin de limiter la complexité d'adaptation des applications à CORDAGE. Les deux fonctionnalités principales offertes par le modèle sont :

- La traduction d'actions de haut niveau en une suite d'opérations de bas niveau. Cette fonctionnalité est assurée par la mise en place de requêtes génériques et spécifiques aux applications. Ces requêtes sont prises en charge par le serveur et entraînent des interactions avec les outils de réservation et de déploiement de la grille.
- La pré-planification du déploiement. Cette fonctionnalité est rendue possible grâce à l'implémentation des représentations en arbre de l'application et des ressources. Ces représentations sont accompagnées de méthodes de construction, de fusion et de projection afin de suivre au mieux le modèle CORDAGE.

L'implémentation actuelle reste cependant incomplète et ne propose pas encore toutes les possibilités offertes par le modèle CORDAGE. La plupart des améliorations souhaitables portent sur la gestion des représentations des applications et des ressources. Ainsi, les algorithmes de fusion et projection des arbres logiques gagneraient à être optimisés et rendus plus adaptables aux besoins de l'application. Dans le cas de la fusion de deux applications, il n'est pour l'instant pas possible de prendre en compte des contraintes de co-localisation, c'est-à-dire la possibilité de fusionner deux nœuds logiques en particulier. Il n'est pas non plus possible de spécifier à quel niveau dans l'arbre principal doit être fusionné l'arbre secondaire, bien que cette limitation puisse être contournée par l'ajout de nœuds virtuels.

Dans le cas de l'algorithme de projection, des améliorations en terme de performance grâce, notamment, à la parallélisation des réservations, sont à mettre en place. Une projection plus intelligente prenant en compte les caractéristiques des groupes logiques et des groupes physiques est aussi à introduire dans l'implémentation. Enfin, l'aspect dynamique de l'algorithme de projection reste à développer, notamment la possibilité de migrer des groupes logiques non déployés vers les nœuds logiques parents dans le cas d'un échec local, comme prévu dans le modèle.

L'implémentation actuelle guide la construction de la représentation physique suivant le critère de la proximité réseau. Une mise en œuvre plus ouverte doit laisser libre la construction de l'arbre physique en se basant sur les spécificités et les besoins de l'application. Enfin, les interactions avec les outils de gestion des ressources de la grille peuvent être améliorées en permettant 1) de faciliter la construction de la représentation des ressources physiques à partir des systèmes d'information de la plate-forme et 2) de rendre cette représentation modifiable pendant un déploiement pour prendre en compte la dynamicité de l'infrastructure.

Ces remarques sont autant de pistes explorées actuellement pour améliorer la mise en œuvre du modèle CORDAGE. L'implémentation est cependant suffisamment complète pour permettre l'utilisation et la validation du prototype avec différentes applications scientifiques issues de projets de recherche dans le domaine des grilles, comme JUXMEM et GFARM. Le chapitre suivant montre comment notre approche a pu être appliquée dans le cas concret du scénario introduit en section 3.2.

Chapitre 6

Validation : un service de partage de données et un système de fichiers distribué

Sommaire

La validation de l'outil de gestion du déploiement dynamique CORDAGE est effectuée dans le cadre du projet de recherche LEGO [1] [149]. Ce projet a pour objectif de proposer des solutions algorithmiques pour le calcul haute performance sur des infrastructures à grande échelle. Ces solutions sont exprimées dans un modèle de programmation multi-paradigme pour applications parallèles, prenant en compte les spécificités des grilles de calculateurs. Ces paradigmes de programmation sont aussi divers que le modèle de composant, le modèle d'accès transparent aux données, le paradigme maître-travailleur ou encore le modèle de *workflow*. Les applications clientes de ce projet sont issues de domaines de recherche variés, tels que la modélisation des échanges entre océan et atmosphère, la cosmologie ou encore

[1] En anglais *League for Efficient Grid Operation*, projet ANR-05-CIGC-11

la manipulation de matrices creuses. De telles applications clientes sont caractérisées par des besoins importants en terme de puissance de calcul, mais aussi en terme de stockage des données. En effet, la taille des données générées par ces applications peut varier du gigaoctet au téraoctet en fonction du type de simulation effectué et de la résolution de calcul demandée. Les données peuvent être transitoires, c'est-à-dire produites par un sous-calcul et consommées par le suivant. Elles doivent donc pouvoir être stockées et accédées de manière rapide pour ne pas ralentir le calcul global. Dans un premier temps nous nous focalisons sur le problème du stockage des données comme un exemple motivant pour valider l'approche CORDAGE.

Dans ce contexte, le stockage des données générées par les applications clientes est assuré par JUXMEM [2], un service de partage de données pour la grille introduit dans [10]. Ce service offre de l'espace de stockage distribué en mutualisant les mémoires physiques des différentes ressources du système. L'espace global est donc la somme des espaces physiques locaux. Il dépend donc directement du nombre de ressources impliquées dans le service. Ce nombre est difficilement prévisible car il n'est pas possible de connaître à l'avance les besoins en espace de stockage pendant l'exécution des applications scientifiques. De plus, avec la génération de données transitoires, ces besoins évoluent de manière significative pendant l'exécution des applications. Il n'est pas raisonnable de dimensionner JUXMEM de manière statique pour stocker l'ensemble de ces données transitoires pendant la durée totale de l'exécution. Une telle stratégie entraînerait la réservation de nombreuses ressources largement sous-utilisées.

La problématique du dimensionnement de JUXMEM s'avère donc être un cas d'étude pertinent pour évaluer l'approche proposée par CORDAGE. Celui-ci peut en effet prendre en charge le caractère dynamique du service de partage de données grâce à la reconfiguration de la topologie. Ce premier scénario permet de valider l'intégration de CORDAGE dans une application ainsi que son aptitude à gérer le déploiement additionnel.

Dans un deuxième scénario, le système de fichiers distribué GFARM [116, 117] est ajouté à JUXMEM afin d'en augmenter les capacités de stockage. L'idée est ici de construire une *mémoire hiérarchique distribuée* composée d'un service de partage de données en mémoire physique pour les accès rapides et d'un système de fichiers distribué pour le stockage à plus long terme. Du point de vue du déploiement, l'application GFARM doit être déployée de manière coordonnée avec JUXMEM. Ce deuxième scénario est utilisé pour valider la fonctionnalité de co-déploiement offerte par CORDAGE.

6.1 Vers un JUXMEM dynamique grâce à CORDAGE

Dans le chapitre 3 nous avons montré toute la complexité de la mise en œuvre du déploiement de JUXMEM avec ADAGE et ce, malgré les efforts effectués pour rendre le pilotage d'ADAGE plus aisé. Les besoins en terme de dynamicité topologique du service de partage de données sont une motivation supplémentaire à l'utilisation de l'outil de gestion du déploiement CORDAGE. Dans le chapitre 5, nous avons montré que l'adaptation de CORDAGE à un type d'application se faisait en définissant des fonctions génériques virtuelles et des fonctions spécifiques. Dans cette section, nous montrons comment définir la fonction ayant la charge de la construction de la représentation logique puis comment ajouter des

[2] En anglais *Juxtaposed Memory*

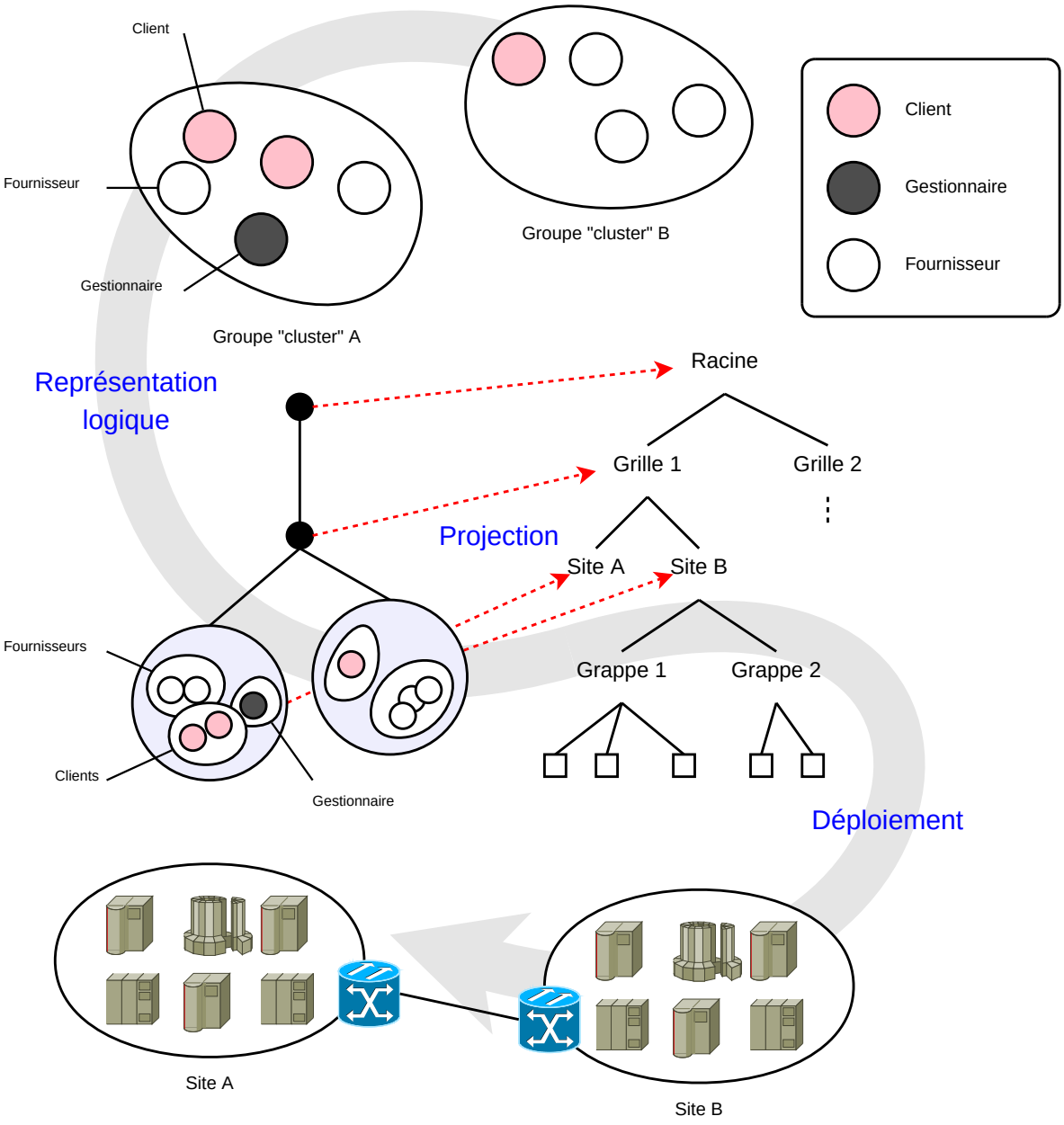

FIG. 6.1 – Vue générale des transformations, de la description de la topologie JUXMEM jusqu'au déploiement sur les machines, en passant par la représentation logique et la projection sur les ressources physiques.

fonctions spécifiques pertinentes pour la gestion du système JUXMEM. Le travail suivant a été effectué pour les deux versions de JUXMEM, la version basée sur JXTA et la version générique. Pour des raisons de clarté et parce que le principe reste le même, nous ne présentons ici que la deuxième version.

6.1.1 Génération de la représentation logique

La première étape de l'adaptation de CORDAGE à JUXMEM est la définition des stratégies de construction de la représentation logique à partir de la description de l'application[3]. Le travail principal consiste à établir les liens entre la topologie de l'application, composée de *groupes cluster* et les nœuds et groupes de l'arbre logique. Pour cela, il faut proposer le critère pertinent pour le placement des entités lors du déploiement. Dans le contexte de JUXMEM, le critère le plus intuitif est la proximité réseau en terme de latence et débit. Le choix de ce critère est basé sur la connaissance fine du fonctionnement de l'application. Il doit permettre 1) d'optimiser les communications entre les clients et les gestionnaires pour garantir un service de partage de données efficace et 2) de distribuer les *groupes cluster* sur des sites multiples pour tolérer les pannes sur un site complet.

La figure 6.1 montre une vue générale des transformations effectuées depuis la description de l'application jusqu'au déploiement. L'idée est ici de placer chaque *groupe cluster* de la description de l'application dans un nœud logique différent. Cette première contrainte permet d'exprimer que les *groupes cluster* ne soient pas déployés sur un même groupe de ressources physiques. De plus, JUXMEM ne définit aucun lien hiérarchique entre les groupes en terme de topologie ou d'affinité pour les interactions. Nous faisons donc le choix de placer les différents nœuds logiques sur un même niveau dans l'arbre. Cette deuxième contrainte garantit de déployer les *groupes cluster* sur des groupes de ressources physiques différents et de même niveau dans la représentation physique.

Les *groupes cluster* peuvent contenir trois types d'entités : les clients, les fournisseurs et le gestionnaire. La question est maintenant de savoir comment organiser ces différentes entités au sein de chaque nœud logique. Une première solution est de créer un groupe logique pour toutes les entités appartenant à ce nœud. Cette solution n'est cependant pas adaptée car ces entités ont des comportements différents, en particulier quant à la durée d'exécution. Ainsi le gestionnaire est-il amené à rester opérationnel pendant toute la période de disponibilité du service. Les fournisseurs doivent être déployés suivant les besoins en espace de stockage. Les clients doivent être déployés uniquement le temps d'effectuer leurs calculs. Le choix a donc été fait de placer ces trois types d'entités dans des groupes logiques différents. Les groupes logiques appartenant à un même nœud sont bien sûr tous projetés sur le même groupe physique ; ils bénéficient d'identifiants de réservation différents, ce qui permet plus de modularité dans la gestion des réservations. Ceci est notamment intéressant lors de la rétraction de fournisseurs ou de clients : il suffit alors de supprimer la réservation attachée à leur groupe logique.

La représentation logique ainsi fixée est mise en œuvre dans la fonction générique virtuelle `generate_logical_topology` (voir section 5.4.3.4), en utilisant l'interface de gestion de la représentation décrite en section 5.4.2.1. Cette fonction est lancée lors du déploiement initial. La topologie adoptée peut être spécifiée par l'utilisateur à l'aide d'un fichier de description de l'application. Si aucun fichier n'est renseigné, la topologie déployée est minimale : elle consiste en un unique *groupe cluster* contenant un gestionnaire. L'aspect dynamique s'exprime dans JUXMEM, par l'ajout et le retrait de *groupes cluster*, de fournisseurs et de clients. Dans ce choix de représentation logique, ceci se traduit par l'ajout et le retrait de nœuds et de groupes logiques.

[3]Un exemple de description d'application JUXMEM est présenté dans le listing 3.12. L'architecture de JUXMEM est présentée dans la section 3.2.1

6.1.2 Définition d'actions spécifiques

L'adaptation de CORDAGE au service de partage de données nécessite la définition d'actions spécifiques au service. Ces actions doivent permettre de prendre en charge la dynamicité de JUXMEM en terme de topologie. Nous pouvons définir deux types de modifications topologiques correspondant à des cas d'utilisation du service.

- Le premier cas fait suite à une requête d'allocation d'un client ne trouvant pas assez de fournisseurs pour stocker les différentes copies de la donnée. Ce premier cas peut être divisé en deux sous-cas : 1) il n'y a pas assez de fournisseurs dans un *groupe cluster* déjà déployé et 2) il n'y a pas assez de *groupes cluster* déployés.
- Le deuxième cas correspond au déploiement d'un nouveau client.

Ces évolutions de la topologie s'accompagnent d'une modification de la description de l'application ainsi que d'une modification de la représentation logique. La description de l'application est modifiée afin que l'outil de déploiement prenne en compte les changements à effectuer. La représentation logique est elle aussi modifiée afin de refléter dans CORDAGE l'objectif à atteindre par la nouvelle pré-planification. Trois actions spécifiques sont donc définies pour le support de JUXMEM.

Ajout d'un *groupe cluster*. Elle s'effectue grâce à l'action spécifique

`CORK_JUXMEM2_ADD_CLUSTER`.

Un nouveau nœud est créé dans la représentation logique avec un identifiant calculé par le serveur CORDAGE. Cet identifiant est retourné au client CORDAGE, comme résultat de la requête spécifique. Ce nouveau nœud est virtuel, c'est-à-dire qu'il ne contient aucun groupe logique. La description de l'application est modifiée en indiquant que le nouveau *groupe cluster* doit se connecter au groupe dans lequel se situe le gestionnaire.

Ajout d'un fournisseur. Elle s'effectue grâce à l'action spécifique

`CORK_JUXMEM2_ADD_PROVIDER`.

Cette action prend en paramètre le nombre de fournisseurs à ajouter ainsi que l'identifiant du *groupe cluster* dans lequel les déployer. La requête retourne au client CORDAGE les identifiants des fournisseurs.

Ajout d'un client JUXMEM. Elle s'effectue grâce à l'action spécifique

`CORK_JUXMEM2_ADD_CLIENT`.

Cette action prend en paramètre l'identifiant du *groupe cluster* dans lequel déployer le client, le port de communication sur lequel écouter, le programme à exécuter, le nombre d'instance de ce client à créer et enfin la liste des arguments à utiliser en entrée du programme client.

Après chacune des modifications, la fonction générique `redeploy` est appelée. Celle-ci est similaire à la fonction générique de déploiement, à la différence qu'elle ne relance pas la génération de la représentation logique. La pré-planification est alors effectuée, en essayant de projeter les nœuds et groupes nouvellement créés. L'outil de déploiement est informé des modifications qu'il répercute dans le service. Un exemple complet d'utilisation de ces actions spécifiques est donné dans le programme synthétique présenté en listing A.3.

6.1.3 Intégration du code CORDAGE dans JUXMEM

Les actions génériques et spécifiques présentées dans la section précédente doivent être déclenchées par un programme ayant accès à suffisamment d'informations pour prendre la décision d'altérer ou non la topologie de JUXMEM. Dans le cadre d'un système autonome complet comprenant les quatre phases de surveillance, analyse, planification et exécution, le pilotage de CORDAGE doit être effectué au sein de la phase d'exécution. Pour des raisons de simplicité, ces quatre phases sont directement assurées par le service de partage de données. L'entité JUXMEM ayant le plus de connaissance sur l'état du service est le gestionnaire. Il est le point de rendez-vous de toutes les entités et il maintient une liste des fournisseurs disponibles dans le service. De plus, il reçoit les requêtes d'allocation mémoire en provenance des clients et il a la charge de sélectionner la liste des fournisseurs susceptibles de stocker une copie de la donnée. Il est de ce fait le candidat le mieux placé pour décider d'augmenter ou réduire la topologique du service.

La mise en place d'une boucle entre JUXMEM, CORDAGE et ADAGE. L'utilisation de l'entité gestionnaire JUXMEM comme client CORDAGE assure aussi l'aspect autonome du système : il existe désormais une boucle dans le graphe des interactions entre JUXMEM, CORDAGE et ADAGE. En effet, lorsque JUXMEM demande l'ajout d'un fournisseur, une requête contenant l'action `CORK_JUXMEM2_ADD_PROVIDER` est envoyée à CORDAGE. Pour prendre en charge cette requête, CORDAGE modifie la description courante de JUXMEM pour y ajouter le fournisseur. Il demande ensuite à ADAGE de prendre les modifications effectuées. ADAGE compare la nouvelle description de JUXMEM avec la description courante et constate qu'un fournisseur doit être déployé. Ce nouveau fournisseur est configuré pour se déclarer à JUXMEM.

Une question se pose alors : quel élément de la boucle doit être lancé le premier par l'utilisateur ? Plusieurs cas d'utilisation ont motivé que chacune des briques puisse être déployée la première. L'utilisateur peut avoir des raisons de lancer JUXMEM manuellement puis de démarrer un serveur CORDAGE qui fera appel à ADAGE. Dans ce cas, JUXMEM doit renseigner CORDAGE sur sa topologie courante.

Un autre scénario qui nous semble plus intuitif est de ne lancer que le serveur CORDAGE, possiblement avec ADAGE, en lui demandant de mettre en place une topologie minimale pour JUXMEM. Cette demande peut directement être exprimée dans la configuration du serveur ou, grâce à une requête effectuée par un client synthétique. Enfin, il est possible de tout déployer en interagissant directement avec ADAGE.

Par la suite, nous nous plaçons dans le deuxième cas, un serveur CORDAGE est déployé en premier grâce à ADAGE, avec comme but le déploiement d'une topologie minimale de JUXMEM. La fiche de suivi correspondant à cette instance du service est donc déjà créée. L'identifiant de la fiche, ainsi que toutes les informations pour contacter le serveur CORDAGE sont contenus dans un fichier de configuration mis à disposition de chaque entité par l'outil de déploiement ADAGE. La modification du comportement des entités JUXMEM pour supporter CORDAGE intervient ensuite à trois reprises.

– La première modification du code de JUXMEM est commune à toutes les entités. Elle consiste à s'enregistrer auprès du serveur CORDAGE et à envoyer un message d'acquittement comme cela est décrit en section 5.3.2.4. Cet acquittement contient l'identifiant de l'entité ainsi que l'identifiant du gestionnaire auquel elle est connectée.

– La deuxième modification prend place sur le gestionnaire lors de la réception d'une requête d'allocation mémoire. Le code original dans JUXMEM consiste à regarder s'il y a assez de fournisseurs pour stocker les copies de la donnée. Si non, un code d'erreur est retourné au client JUXMEM qui va retenter l'allocation un certain nombre de fois avant d'abandonner. Avec le support CORDAGE, nous modifions le comportement en cas d'échec pour calculer le nombre de fournisseurs à ajouter et envoyer une requête spécifique d'ajout de fournisseurs au serveur CORDAGE. Le nombre P de fournisseurs à ajouter est calculé en fonction du nombre de copies demandées n, du nombre de fournisseurs déclarés au gestionnaire f et du nombre de fournisseurs commandés par CORDAGE c, selon la formule $P = n - (f + c)$. La variable c permet de prendre en compte les fournisseurs en cours de déploiement mais pas encore enregistrés auprès du gestionnaire. Si P est strictement positif, une requête est envoyée à CORDAGE et c est incrémentée de P fournisseurs. Enfin, lors de l'enregistrement d'un fournisseur auprès du gestionnaire, le compteur c des fournisseurs en attente de déploiement est décrémenté.
– La troisième modification du code de JUXMEM est elle aussi commune à toutes les entités. Elle consiste à envoyer un message de suppression d'entité en indiquant son propre identifiant. Cette requête est envoyée lorsque le programme associé à cette entité demande la terminaison de sa participation au service JUXMEM. La requête permet notamment au serveur CORDAGE de libérer si possible la ressource physique réservée pour cette entité. Dans l'implémentation actuelle, cette requête est systématiquement envoyée lorsque le programme termine sa participation au service JUXMEM. Cette approche doit être nuancée car le programme devrait pouvoir continuer son exécution sans pour autant être connecté au service JUXMEM.

L'ajout du code client CORDAGE dans les entités JUXMEM permet un fonctionnement en mode autonome sans pour autant avoir à modifier profondément le code du service de partage de données, ni à avoir recours à un environnement lourd de développement pour applications autonomes.

Les environnements pour applications autonomes restent cependant une piste privilégiée pour la validation de CORDAGE : ils devraient permettre de mieux prendre en charge les systèmes d'information pour la surveillance des ressources et des entités et de mieux formaliser les politiques de décision aboutissant au pilotage de CORDAGE. Des expérimentations portant sur les fonctionnalités et les performances apportées par cette architecture logicielle sont décrites dans le chapitre 7.

6.2 Couplage de JUXMEM avec le système de fichiers GFARM

Le service JuxMem stocke les données en mémoire physique, principalement pour profiter de la rapidité d'accès supérieure par rapport au stockage sur disque. Ce choix limite la taille maximale des données pouvant être accueillies par le service, l'espace de stockage en mémoire physique étant environ cent fois moins important que l'espace disque. Ceci est clairement une limitation pour les applications nécessitant la gestion de grandes masses de données. Une solution est d'adjoindre au service JuxMem un stockage secondaire sur disque afin d'augmenter l'espace de stockage. Ceci a par ailleurs l'avantage d'offrir des propriétés supplémentaires de persistance des données sur disque. C'est l'objectif d'un travail que nous

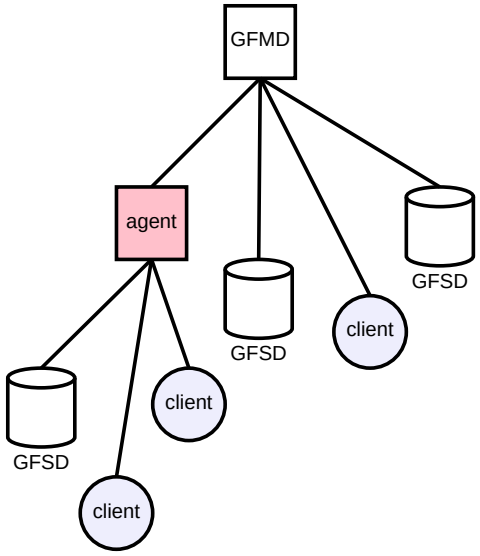

FIG. 6.2 – Architecture de GFARM : le GFMD est le serveur de méta-données, les agents sont des serveurs de cache pour les méta-données, les GFSD sont des nœuds de stockage. Dans la version 2 de GFARM, les nœuds de stockage et les clients sont directement connectés au GFMD.

avons effectué en collaboration avec l'équipe GFARM de l'AIST et l'Université de Tsukuba au Japon. Ce travail a mené à la conception d'un service de partage de données hiérarchique et distribué. Le stockage secondaire est assuré par GFARM, un système de fichiers distribué pour les grilles de calculateurs. Nous nous attachons ici à présenter le problème du point de vue du déploiement. Le lecteur intéressé par ce projet pourra trouver des informations complémentaires dans [13] et [40]. Dans cette section, nous présentons l'architecture de GFARM, les solutions qui ont été employées pour le déployer conjointement avec JUXMEM et enfin comment utiliser ce projet pour valider le concept de sous-applications dans CORDAGE.

6.2.1 GFARM : un système de fichiers distribué pour la grille

Le système de fichiers distribué GFARM vise des applications manipulant des données de l'ordre du péta-octet. L'idée est d'offrir un service inter-organisation pour le partage de données, en fédérant les disques locaux des participants et en décentralisant les accès. Pour des raisons de performance et de tolérance aux fautes, les fichiers sont fragmentés et répliqués sur les différents nœuds de stockage. Si la procédure de réplication reste à la charge de l'utilisateur, les accès en lecture s'effectuent, eux, de manière transparente vis-à-vis de la localisation des données. La bande passante est par ailleurs améliorée et optimisée pour le passage à l'échelle grâce à l'utilisation des entrées-sorties parallèles.

L'architecture de GFARM est composée de 4 types d'entités.

Le serveur de méta-données, aussi appelé *GFMD* pour GFARM *Master Daemon*, est chargé de centraliser toutes les méta-informations sur les données stockées dans le système. Ces méta-informations portent sur le nom, le type, le propriétaire, les droits d'accès, la localisation dans l'arbre logique du système de fichiers ou encore la localisation physique des fragments du fichier. Le serveur enregistre ces méta-informations dans une

base de données.

Les *agents* sont des serveurs de cache pour le serveur de méta-données. Ils permettent de distribuer la charge des requêtes sur plusieurs nœuds.

Les nœuds de stockage, aussi appelés *GFSD* pour GFARM *Storage Daemon*, sont des entités ayant la charge de mettre à disposition de l'espace disque.

Les clients sont des entités faisant appel au système de fichiers, soit par le biais d'une bibliothèque, soit par des programmes prévus à cet effet, soit par un montage de GFARM dans le système de fichiers local.

Les clients et les nœuds de stockage sont attachés soit à un agent, soit directement au serveur de méta-données, comme cela est montré en figure 6.2. Depuis la version 2 de GFARM [117], le rôle *agent* a été supprimé. La redondance des méta-données est tout de même assurée sur le serveur par une base de données pouvant être distribuée sur plusieurs nœuds.

Le déploiement du système de fichiers GFARM est pris en charge par ADAGE. Ce travail a été effectué en étroite collaboration avec l'équipe GFARM, notamment suite à la migration vers la version 2 du système de fichiers. Comme pour le support de JXTA et JUXMEM, il a été nécessaire de définir un langage de description spécifique à l'application ainsi qu'un greffon pour la traduction dans la description générique GADE. Le langage de description spécifique à GFARM n'est pas montré ici car il ne présente aucune difficulté conceptuelle nouvelle. En revanche, l'une des particularités du greffon provient de la gestion des fichiers de configuration de GFARM. Le déploiement de GFARM doit respecter des contraintes temporelles entre les entités. L'ordre logique provient principalement de l'architecture en arbre : le serveur de méta-données est lancé en premier, suivi des agents puis des nœuds de stockage et des clients. À chaque niveau de l'arbre, les entités créent un fichier de configuration contenant les informations transmises par leur parent ainsi que de nouvelles informations permettant à chacune des entités filles de se connecter à elles. Cette configuration contient notamment l'adresse physique ainsi qu'un couple de clés privée et publique pour restreindre l'accès au système de fichiers. En pratique, le fichier de configuration créé par le serveur de méta-données va contenir des informations de connexion au serveur ainsi qu'une clé d'accès. Ce fichier doit être fourni aux agents qui vont chacun créer leur version du fichier par mise à jour de l'original. Et enfin ces fichiers sont transmis aux clients et aux nœuds de stockage qui sont connectés aux agents.

Le transfert des fichiers de configuration est pris en charge dans ADAGE par le greffon GFARM. Chaque entité crée son propre fichier et retourne, par la fonction de rappel `callback`, l'adresse de ce fichier. Cette adresse est transmise en paramètre de configuration dynamique aux entités filles qui vont effectuer un transfert du fichier avant de le modifier avec leurs propres informations. Cette stratégie tire parti de l'aspect hiérarchique de GFARM. Elle offre un comportement satisfaisant pour le passage à l'échelle, à moins cependant que l'arbre GFARM ne soit à la base mal équilibré.

6.2.2 Co-déploiement de JUXMEM et GFARM

L'architecture logicielle proposée dans le cadre de la mémoire hiérarchique distribuée est basée sur l'idée que les fournisseurs JUXMEM puissent écrire dans le système de fichiers GFARM. Les fournisseurs JUXMEM doivent donc pouvoir se comporter comme des clients

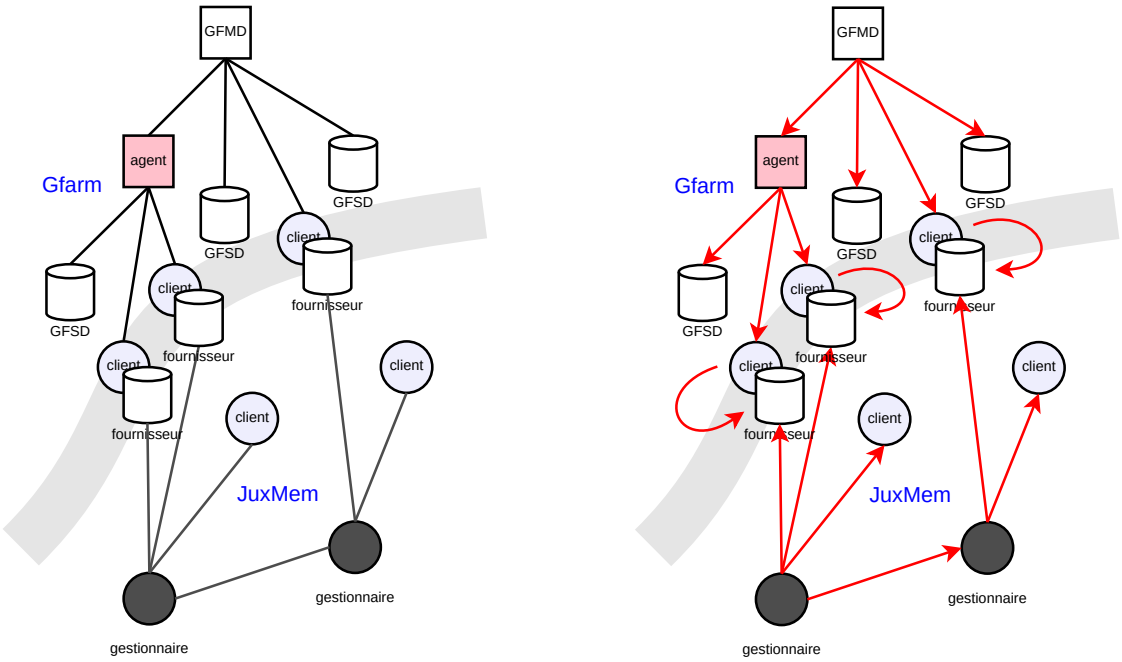

FIG. 6.3 – (a) Architecture conjointe entre GFARM et JUXMEM. Les fournisseurs JUXMEM jouent le rôle de clients GFARM. (b) Dépendances temporelles de déploiement liant les entités.

GFARM. Cette interaction est décrite de manière plus fine dans [13]. D'un point de vue déploiement, cela implique deux types de contraintes.

Contraintes de placement : chaque entité fournisseur JUXMEM doit être déployée sur la même ressource physique qu'une entité cliente GFARM. C'est ce que montre la figure 6.3(a), ces deux types d'entités se trouvant à la frontière des deux applications.

Contraintes temporelles : le système de fichiers doit être opérationnel avant le service de partage de données. Si l'on regarde plus en détail les liens entre entités, les clients GFARM doivent être déployés avant les fournisseurs JUXMEM. C'est ce que montre la figure 6.3(b) : les flèches rouges indiquent une dépendance temporelle entre deux entités. Les dépendances internes, liant deux entités de la même application, sont des dépendances calculées implicitement par le greffon. Ces dépendances s'avèrent être aujourd'hui suffisantes pour déployer les deux applications. L'ajout de nouvelles dépendances implicites n'est cependant pas à exclure, comme par exemple, entre les clients et les fournisseurs JUXMEM ou encore entre les clients et les nœuds de stockage GFARM. L'ajout de nouvelles dépendances réduit le parallélisme lors du déploiement des entités et donc augmente le temps total requis pour déployer l'application.

Si, pour la plupart, ces dépendances sont déjà prises en charge de manière implicite par les deux greffons ADAGE, la mise en place des contraintes inter-application requiert une coordination plus fine des déploiements. Pour prendre en charge ces contraintes, ADAGE propose le concept de *méta-application*. Une méta-application est caractérisée par une description d'application faisant référence à une ou plusieurs sous-descriptions d'applications. Ces sous-descriptions peuvent concerner des types d'applications différents. La description

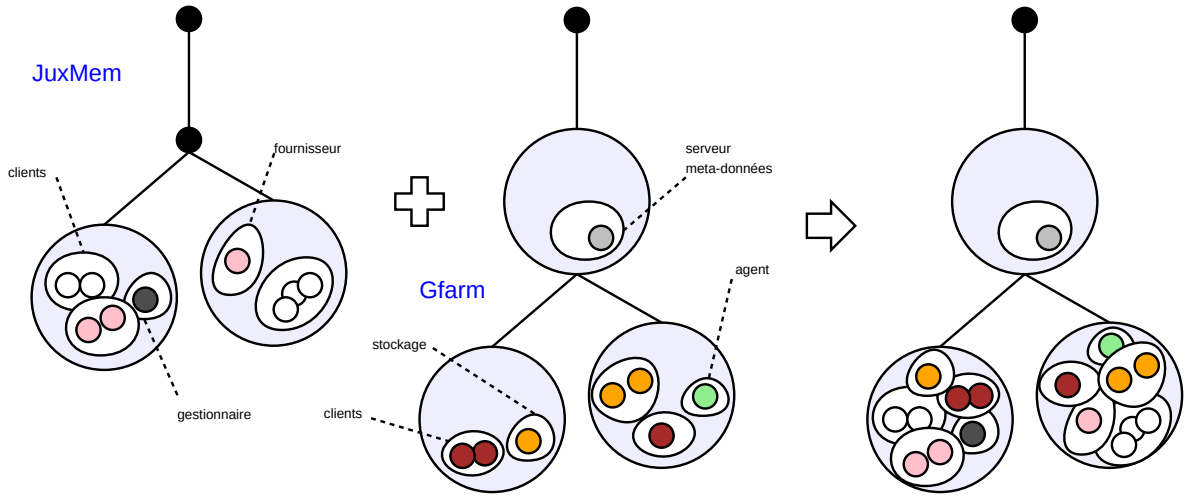

FIG. 6.4 – Représentations logiques de JUXMEM et GFARM puis fusion pour co-déploiement.

de méta-application permet aussi d'exprimer des contraintes temporelles entre les entités appartenant à deux applications différentes. Il en est de même pour les contraintes de co-localisation.

Dans le contexte de la mémoire hiérarchique distribuée, le concept de méta-application est utilisé pour décrire les applications composées d'une instance de JUXMEM et d'une instance de GFARM. Les contraintes de co-localisation sont définies de telle sorte que chaque gestionnaire JUXMEM soit déployé sur la même ressource physique que le client GFARM associé. De plus les contraintes temporelles expriment le fait que les clients GFARM doivent être déployés avant les gestionnaires JUXMEM, ce qui implique que le système de fichiers soit opérationnel avant le service de partage de données. Le concept de méta-application est proposé dans ADAGE par Christian Perez[4] et Landry Breuil[5]. Bien que fonctionnelle, cette approche reste laborieuse pour l'utilisateur et dédiée aux déploiements statiques.

6.2.3 Vers une gestion coordonnée grâce à CORDAGE

Le modèle introduit par CORDAGE permet de prendre en compte les applications composites, c'est-à-dire formées de plusieurs sous-applications. Ceci est effectué en attachant sur chaque fiche de suivi une liste de fiches de suivi correspondant aux sous-applications. La fonction de déploiement effectue alors, de manière récursive, la génération des représentations logiques pour toutes les sous-applications, puis effectue une fusion en une seule représentation. Le résultat de cette fusion est un arbre logique pouvant être projeté, de manière classique, sur l'arbre des ressources physiques.

La représentation logique de GFARM est construite de manière similaire à celle de JUXMEM. Comme pour JUXMEM, le critère pris en compte est la proximité en terme de connexion réseau. La figure 6.4 présente un exemple d'arbre logique pour GFARM. Cet exemple montre que le serveur de méta-données est placé seul dans un nœud logique en haut de l'arbre. Les nœuds logiques fils contiennent les agents, les nœuds de stockage

[4]Chargé de recherche à l'INRIA.
[5]Ingénieur associé à l'INRIA.

Listing 6.5 – Code client CORDAGE pour le co-déploiement de JUXMEM et GFARM.

```
1  cdgclient* cordage_juxmem = new cdgclient(cordageip, cordageport,
2                                      assignment_juxmem, CORK_TAG_JUXMEM2,
3                                      appl_juxmem, ctrl_juxmem);
4  cdgclient* cordage_gfarm = new cdgclient(cordageip, cordageport,
5                                     assignment_gfarm, CORK_TAG_GFARM,
6                                     appl_gfarm, ctrl_gfarm);
7
8  cordage_juxmem->add_sub_app(cordage_gfarm);
9
10 cordage_juxmem->deploy(walltime);
11
12 cordage_juxmem->terminate();
13
14 delete(cordage_juxmem);
15 delete(cordage_gfarm);
```

(GFSD) et les clients GFARM. Ces entités sont placées dans les nœuds en fonction de leur affinité : en règle générale, il est de préférable de placer un nœud de stockage proche d'un client. En effet, GFARM sélectionne automatiquement le GFSD le plus proche lors d'un accès sur un fichier. Enfin, au sein de chaque nœud logique, les entités sont regroupées selon leur type dans les groupes logiques.

L'arbre logique de GFARM est ensuite fusionné avec l'arbre logique de JUXMEM. Cette étape suppose que les deux arbres soient *alignés*. L'alignement est effectué de telle sorte que lors de la fusion, les groupes logiques contenant les fournisseurs JUXMEM soient à la même profondeur que les groupes logiques contenant les clients GFARM. L'idée est en effet de fusionner, au sein d'un même nœud logique, deux nœuds comportant suffisamment de clients GFARM par rapport au nombre de fournisseurs JUXMEM. Lorsque l'algorithme décide la fusion de deux nœuds logiques, des contraintes de co-localisation entre les clients GFARM et les fournisseurs JUXMEM sont créées

Le support des politiques de fusion n'est pour l'instant pas encore implémenté dans le prototype CORDAGE, c'est-à-dire que les nœuds logiques sont juste ajoutés comme enfants sans fusionner les groupes logiques. La piste étudiée actuellement pour mettre en œuvre une fusion transparente pour l'utilisateur est l'expression des politiques au sein de chaque fiche de suivi : la fiche correspondant à JUXMEM contiendra une politique spécifique à la fusion avec GFARM. Dans cet exemple, la politique de fusion devra donc bien exprimer que deux nœuds logiques doivent être fusionnés s'ils permettent de réunir les clients GFARM et les fournisseurs. Comme pour la projection de l'arbre logique sur l'arbre physique, le travail de fusion touche au domaine de la résolution de contraintes. Dans ce cadre, l'utilisation d'un solveur pourrait apporter une aide appréciable.

Le pilotage de l'application composite s'effectue grâce à un ou plusieurs clients CORDAGE. Ce code client peut être externe ou directement placé dans l'une des applications. Le listing 6.5 propose un code client dans lequel deux clients CORDAGE sont déclarés. Ces clients partagent le même serveur, identifié par son adresse et son port de communication. Ils se différencient par l'identifiant de la fiche de suivi, le type de l'application ainsi que les fichiers de description et de contrôle de l'application. Le client GFARM est ensuite ajouté

comme sous-application du client JUXMEM et l'ensemble est fusionné et déployé. La fonction de terminaison est alors appelée sur le client JUXMEM. Une version récursive de cette fonction de terminaison peut être envisagée pour terminer l'arbre des applications.

L'aspect dynamique du co-déploiement avec CORDAGE est possible grâce à l'implémentation d'actions spécifiques à l'application, mais aussi grâce à une ou plusieurs sous-applications. Dans le contexte du déploiement de la mémoire hiérarchique distribuée, une action d'ajout de fournisseur JUXMEM doit s'accompagner de l'ajout d'un client GFARM. Par construction des fiches de suivi, les actions spécifiques ont un accès à la liste des sous-applications. Dès lors, il est possible de modifier l'action spécifique d'ajout d'un fournisseur, présentée en section 6.1.2, en recherchant si une sous-application GFARM est présente. Si oui, une requête interne au serveur est envoyée pour déclencher l'action spécifique d'ajout d'un client sur la fiche de suivi de GFARM.

Le co-déploiement avec CORDAGE reprend les grandes lignes du déploiement simple avec, en plus, la possibilité d'attacher des sous-applications à son application. La fonction de fusion des représentations logiques est déjà opérationnelle mais ne fournit pas encore le support des politiques de fusion nécessaires pour prendre en compte les contraintes de placement inter-application. Cette fonctionnalité fait partie des travaux à court terme dans CORDAGE.

6.3 Valorisation dans le cadre du projet LEGO

Les travaux de validation du prototype présentés précédemment préparent l'intégration de CORDAGE au sein du projet LEGO. Ce projet vise à proposer une plate-forme pour la programmation multi-paradigme d'applications scientifiques. Ces applications doivent être adaptées pour être exécutées sur des infrastructures de type grilles de calculateurs. LEGO repose sur les domaines d'expertise de différentes équipes de Recherche : le LIP (Lyon), l'IRISA (Rennes), le LaBRI (Bordeaux), l'ENSEEIHT (Toulouse), le CERFACS (Toulouse) ainsi que le CRAL/CEA (Saclay). L'architecture de ce projet est composée de nombreuses briques logicielles développées par ces équipes et appartenant aux catégories des applications métier, des intergiciels, des outils et des services. L'ensemble de la pile logicielle est ici représentée, du code métier aux couches de communications à haute performance.

La figure 6.6 montre un sous-ensemble simplifié des briques présentes dans le prototype du projet LEGO. Elles sont ici classées en trois groupes.

- Les applications métier en vert : couplage de code en cosmologie avec RAMSES [156], couplage de code pour l'étude des échanges océan-atmosphère avec OASIS [124] et la résolution de systèmes linéaires creux avec TLSE [140, 42].
- Les intergiciels en bleu : l'ordonnanceur DIET présenté dans la section 2.2.1, JUXMEM et GFARM.
- Les outils de gestion de déploiement en rouge : ADAGE et CORDAGE.

Ces briques s'auto-organisent, notamment grâce à l'utilisation du paradigme de programmation des composants logiciels CCM [154].

Le scénario d'utilisation d'une telle architecture commence au niveau d'ADAGE suite à la demande de lancement d'un calcul par un utilisateur. ADAGE déploie, de manière statique, l'ordonnanceur DIET paramétré pour lancer l'application choisie, ainsi qu'une ins-

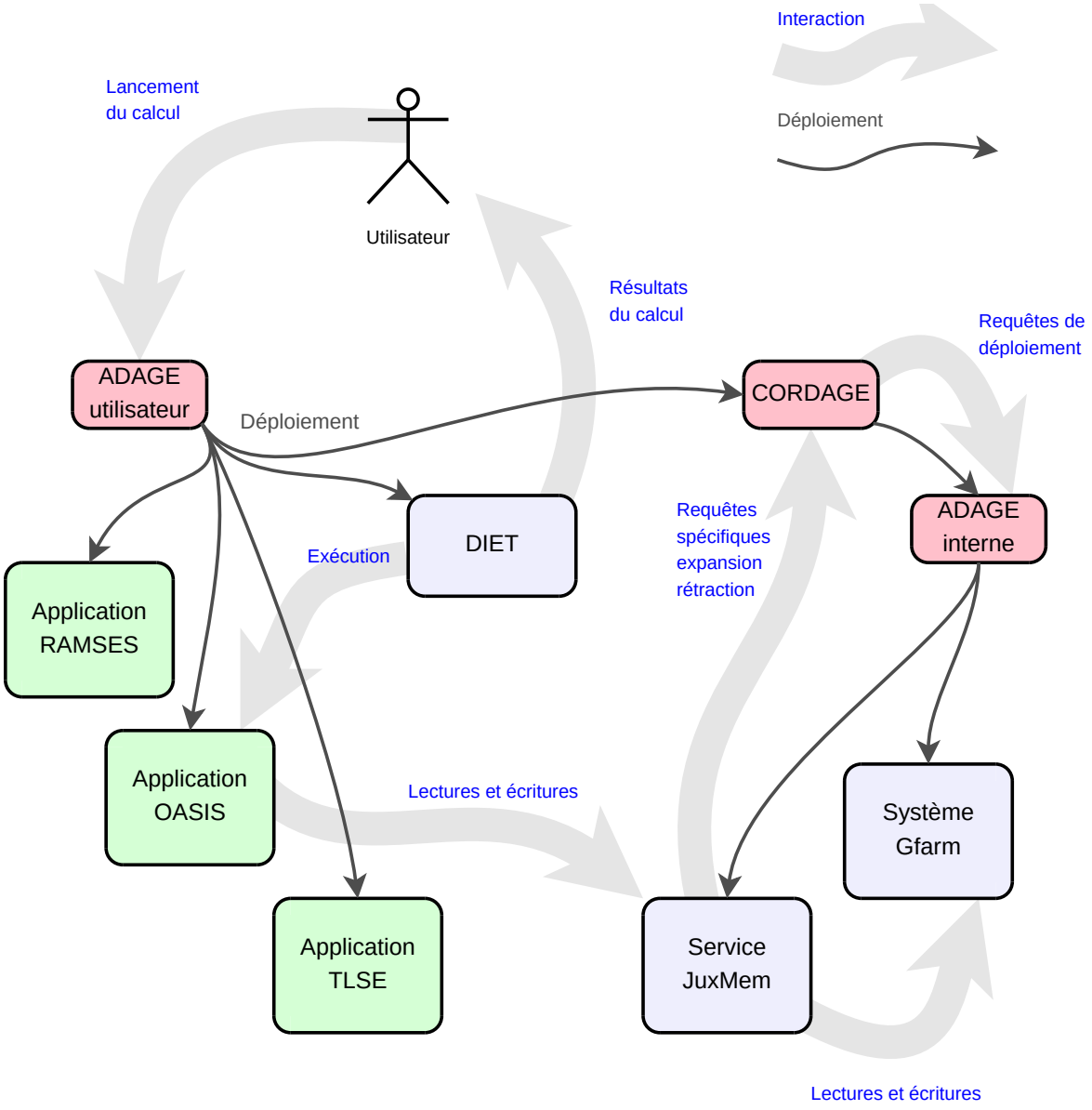

FIG. 6.6 – Validation de CORDAGE dans le cadre du projet LEGO.

tance du serveur CORDAGE. CORDAGE déploie alors une instance privée d'ADAGE et demande la mise en place des topologies minimales pour le service JUXMEM et le système de fichiers GFARM. Ce déploiement en arbre nécessite le transfert d'une information de connexion entre CORDAGE et l'instance utilisateur d'ADAGE afin de permettre aux applications de contacter le service JUXMEM. Une fois cette phase de déploiement terminée, DIET organise l'exécution des tâches pour l'application métier. Cette application effectue des lectures et écritures directement dans JUXMEM, qui sont répercutées dans GFARM. En fonction de ces demandes, le service JUXMEM peut effectuer des requêtes spécifiques d'expansion ou de rétraction. Ces demandes sont traitées par CORDAGE comme expliqué dans les chapitres précédent.

6.4 Conclusion

La validation de CORDAGE a nécessité la conception de nouvelles procédures de déploiement ADAGE pour prendre en charge le service JUXMEM et le système de fichiers GFARM. Ce travail préliminaire a permis d'explorer les mécanismes subtils du déploiement à grande échelle d'applications distribuées basées sur des technologies différentes. La prise en charge de ces applications par CORDAGE nécessite la mise en place d'une représentation logique pertinente et la définition d'un jeu d'actions spécifiques. Ce chapitre montre que CORDAGE, tant par son modèle que par son implémentation, permet d'exprimer facilement les besoins des applications en terme de déploiement. Une proposition d'intégration dans le projet LEGO, mettant en jeu des applications réelles, permet de mettre en valeur l'intérêt d'une telle approche du déploiement dynamique pour les utilisateurs des grilles modernes .

Cette approche peut encore être utilisée dans le cadre des plates-formes de développement pour environnements autonomes, comme le projet DYNACO [25]. Cette plate-forme est présentée plus en détail dans la section 2.2.2.2. Elle offre un canevas pour mettre en œuvre les quatre fonctionnalités des systèmes autonomes que sont la surveillance, l'analyse, la planification et l'exécution. DYNACO peut être utilisé au sein d'un projet comme LEGO pour prendre en charge l'adaptabilité de l'architecture, notamment au niveau du déploiement. Dans ce cas, DYNACO a la charge de surveiller la quantité de mémoire mise à disposition par le service JUXMEM et de décider si il est nécessaire de demander une expansion ou une rétraction. DYNACO implémente toutes les politiques de décision et devient le client principal de CORDAGE. Cette approche est discutée dans les perspectives du chapitre 8.

La validation présentée dans ce chapitre concerne la partie fonctionnelle du prototype. Dans le chapitre suivant, nous nous attachons à mesurer les performances en terme de temps de traitement des requêtes CORDAGE. Le choix de la construction de la représentation logique y est aussi discuté.

Chapitre 7

Évaluation

Sommaire

Dans le chapitre précédent, le prototype proposé pour illustrer le modèle CoRDAGe a été validé par son intégration dans divers projets de recherche. Cette validation porte principalement sur les aspects fonctionnels du prototype. Dans ce chapitre, nous nous attachons à évaluer les performances de CoRDAGe, notamment le surcoût introduit par son utilisation en terme de temps de traitement des requêtes de gestion du déploiement de l'application. Cette évaluation s'effectue dans un premier temps sur des configurations statiques. Les mesures effectuées sur ces configurations doivent montrer l'impact du choix de la représentation logique sur les performances de réservations des ressources et de déploiement. Dans un deuxième temps, nous effectuons une analyse sur des configurations dynamiques pour observer le comportement du prototype au cours de l'exécution de l'application et des besoins pouvant survenir.

7.1 Méthodologie d'expérimentation

Les expérimentations proposées dans ce chapitre se déroulent sur une architecture réelle. Contrairement à une approche basée sur la simulation ou l'émulation, les expérimentations sur des infrastructures réelles caractérisent le comportement du prototype dans les conditions rencontrées par l'utilisateur final. Nous évaluons ici CORDAGE, dont le but est d'assister l'exécution des applications sur les grilles. Il semble donc tout naturel de se placer dans le contexte d'une grille de calcul comme GRID'5000, dédiée à l'expérimentation.

La configuration commune *à toutes les expérimentations* présentées dans ce chapitre comprend les trois points suivants.

1. Le lancement du serveur CORDAGE sur la machine frontale du site de Rennes.
2. L'utilisation de l'outil de réservation global de la grille, OARGRID, installé sur le site de Rennes. OARGRID permet d'effectuer des appels aux différentes instances d'ordonnanceur OAR sur chacun des sites de GRID'5000. OAR est présenté en section 3.1.2.
3. Le démarrage d'une instance de l'outil de déploiement ADAGE sur la frontale pour chaque application gérée par le serveur CORDAGE. ADAGE est présenté en section 3.1.3.

Les requêtes transmises au serveur émanent principalement d'un ou plusieurs clients synthétiques dont le comportement doit mettre en valeur les différentes fonctionnalités du prototype. Quelques requêtes proviennent aussi des entités déployées. Ces requêtes correspondent à des messages d'acquittement : elles indiquent au serveur que les entités ont été correctement lancées. Le traitement de ces requêtes n'est pas pris en compte dans l'évaluation de CORDAGE. Celles-ci sont considérées comme négligeables en comparaison des requêtes de déploiement.

Le code de CORDAGE a été instrumenté pour mesurer le temps passé dans les différents modules du prototype. Une première sonde mesure t_c, le coût de chaque requête mesuré sur le client, c'est-à-dire entre le moment du début de la méthode synchrone `perform_action` et sa fin. Cette mesure concerne directement l'utilisateur de CORDAGE car elle comprend toutes les étapes nécessaires au traitement des requêtes : envoi et traitement de la requête sur le serveur. Le traitement consiste notamment à modifier la représentation logique ou encore à interagir avec les outils de réservation et de déploiement de la grille.

Une deuxième sonde mesure le temps t_s passé sur le serveur entre la réception de la requête et l'envoi du résultat au client. Ce temps est décomposé en plusieurs sous-mesures, les trois principales sont les suivantes.

- Le temps t_r passé à interagir avec l'outil de réservation des ressources.
- Le temps t_d passé à interagir avec l'outil de déploiement.
- Le temps total nécessaire pour projeter l'arbre logique sur l'arbre physique. Lorsque plusieurs interactions avec l'outil de réservation sont nécessaires, les temps sont additionnés pour calculer t_r. Il est alors possible de calculer le surcoût S lié à l'utilisation de CORDAGE pour la gestion des déploiements par la formule : $S = t_s - (t_r + t_d)$.

La première partie de ce chapitre porte sur l'évaluation de CORDAGE dans des configurations statiques. Cette partie met en évidence les coûts d'interaction t_r et t_d avec les outils de réservation et respectivement de déploiement. Dans la deuxième partie, nous évaluons CORDAGE de manière dynamique, en lui soumettant, pendant le temps de l'expérience,

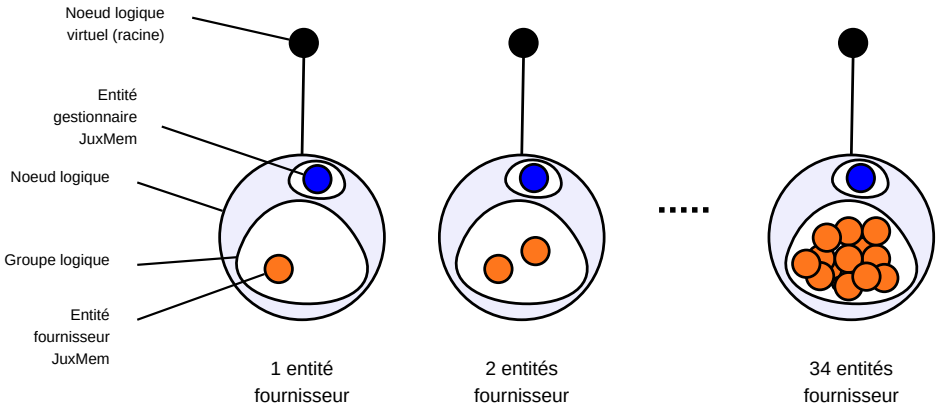

FIG. 7.1 – Évolution de la taille du groupe logique.

une suite de requêtes. Nous nous focalisons alors sur t_s, le temps mesuré sur le serveur et sur t_c, le temps mesuré sur le client.

7.2 Déploiement statique

Dans un premier temps, nous évaluons les performances du prototype lors d'un déploiement simple, c'est-à-dire que nous prenons en compte le traitement d'une seule requête. La prise en charge d'une application par CORDAGE nécessite de définir la manière de construire et modifier la représentation logique. Cette représentation logique, présentée dans le chapitre 4, est construite sous forme d'un arbre logique dont les nœuds contiennent des groupes logiques. Les groupes logiques contiennent les entités à déployer. La phase de projection associe à chaque groupe logique une réservation de ressources physiques. La manière de construire la représentation logique impact donc directement le nombre de réservations à effectuer pour un déploiement. C'est notamment ce type de corrélation entre la représentation logique et le coût du déploiement que nous étudions dans cette section.

Plusieurs configurations de la représentation logique ont été testées : elles se distinguent par l'organisation des entités au sein des nœuds et groupes logiques. Cette étude permet de discuter la manière de construire l'arbre logique en fonction des performances de déploiement attendues et les besoins de chaque type d'application.

7.2.1 Impact de la taille du groupe logique sur le temps de traitement des requêtes

Objectifs de l'expérience. Dans la représentation logique proposée par CORDAGE, les groupes logiques permettent d'imposer que les entités soient déployées sur des ressources appartenant à un même groupe physique. Ceci est assuré par l'association d'un numéro de réservation de ressources physiques à chaque groupe logique, ainsi qu'à la génération contraintes de placement. La taille du groupe logique détermine le nombre de ressources physiques à demander pour la réservation associée ainsi que le nombre d'entités à déployer en une seule commande par l'outil de déploiement. L'objectif de cette expérience est de me-

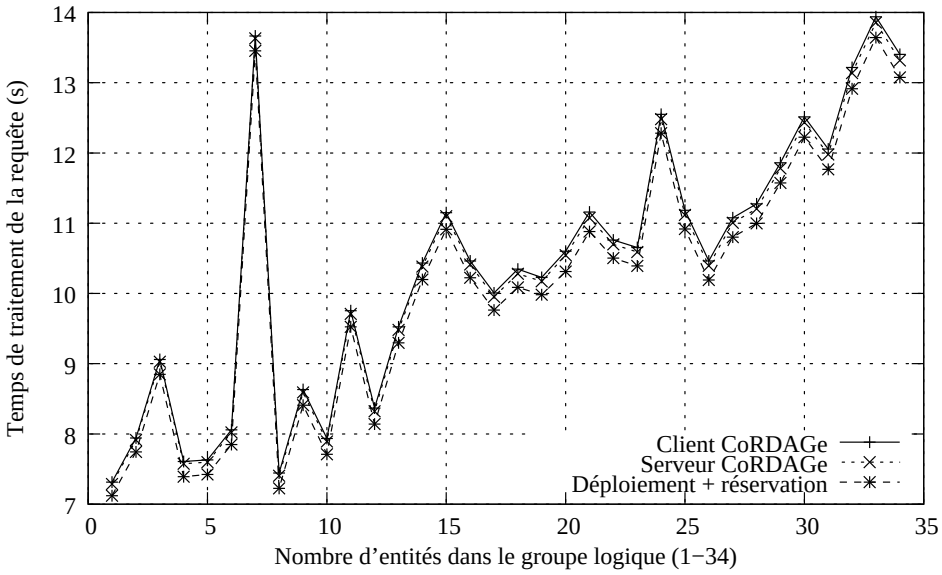

FIG. 7.2 – Impact de la taille du groupe logique sur le temps total de traitement des requêtes.

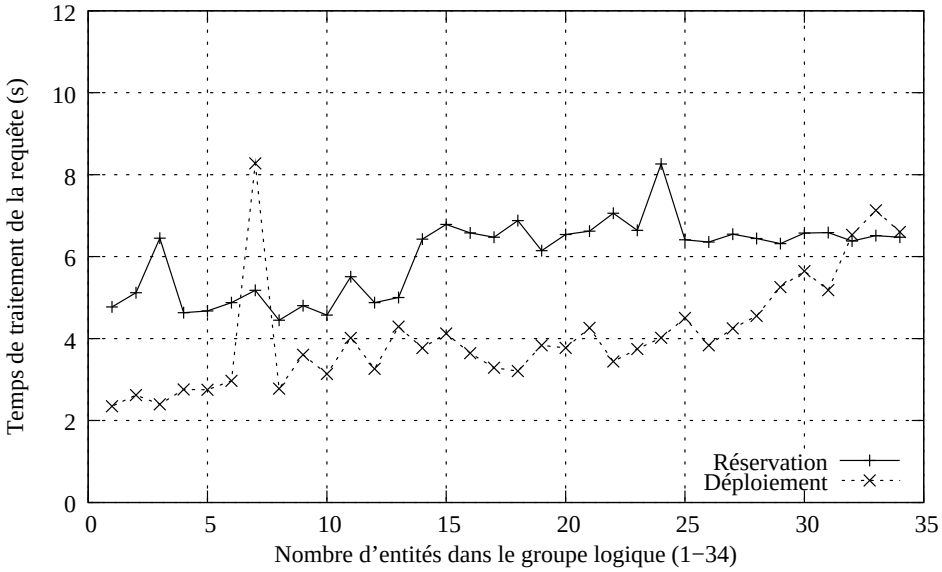

FIG. 7.3 – Impact de la taille du groupe logique sur le temps de réservation et de déploiement.

surer l'impact de la taille d'un groupe logique sur les performances de CORDAGE, des outils de réservation et de déploiement.

Configuration utilisée. Nous choisissons d'effectuer cette mesure par le déploiement d'un nombre croissant de fournisseurs au sein du service JUXMEM. Pour cela, nous utilisons l'action spécifique d'ajout de fournisseurs présentée en section 6.1.2. Cette action prend en paramètre le nom du groupe *cluster* JUXMEM dans lequel déployer les fournisseurs ainsi que le nombre de fournisseurs. Le résultat sur la représentation de l'application est la création d'un groupe logique dans le nœud logique correspondant au *cluster* spécifié. Ce groupe contient autant d'entités fournisseur que demandé.

Le protocole expérimental consiste à lancer un serveur CORDAGE ainsi qu'un client synthétique sur la machine frontale de Rennes. Les ressources physiques réservées par CORDAGE font toutes partie de la grappe de calculateurs *Azur* du site de Sophia. L'algorithme utilisé par ce client est composé d'une boucle portant sur 34 itérations dans lesquelles un nombre croissant de fournisseurs, de 1 à 34, sont déployés. La figure 7.1 montre l'évolution de la représentation logique pour chaque mesure effectuée. Après chaque requête de déploiement, le groupe logique déployé est supprimé avec toutes les entités qu'il contient. Ceci permet 1) de libérer les ressources physiques qui redeviennent disponibles pour la requête suivante et 2) de ne pas accumuler inutilement de groupes logiques déjà déployés dans l'arbre logique. Nous ne mesurons que les requêtes de déploiement. Le gestionnaire est déployé en premier pour tout le temps de l'expérience. Le temps pour effectuer son déploiement n'est pas mesuré.

Toutes les requêtes, qu'elles soient de nature à ajouter des fournisseurs ou à les supprimer, sont espacées de 30 secondes. Ceci permet de s'assurer qu'aucun effet d'accumulation des requêtes n'intervienne sur le serveur et les outils de la grille. Pendant tout le temps de l'expérience nous nous sommes assurés que cette grappe proposait suffisamment de ressources physiques de libres pour déployer jusqu'à 34 entités en une seule fois.

Discussion des résultats. La figure 7.2 illustre le temps passé sur le client et le serveur pour traiter chacune des 34 requêtes de déploiement de fournisseurs. Le temps mesuré est décomposé pour mettre en valeur les temps mesurés du côté du client, du côté du serveur et celui requis par les outils de réservation et de déploiement. La première observation est que le temps global augmente en fonction du nombre d'entités déployées dans le groupe logique. Il passe de 7 secondes pour un fournisseur à 14 secondes pour 34 fournisseurs. L'écart entre les différentes courbes ne varie pas de manière significative, ce qui indique que les temps de traitement de la requête du côté du client et du côté du serveur sont négligeables à cette échelle. L'augmentation du temps de traitement est donc à imputer aux outils de réservation et de déploiement. Les pics observés dans la progression des courbes s'explique par des perturbations extérieures des outils de réservation et de déploiement. Nous pouvons expliquer ces perturbations de trois manières.

- D'autres utilisateurs de la grille interagissent avec les outils OARGRID sur Rennes et OAR sur Sophia.
- Les machines frontales de Rennes et Sophia, utilisées pour supporter OAR, OARGRID et ADAGE, peuvent subir une charge du processeur et du disque.

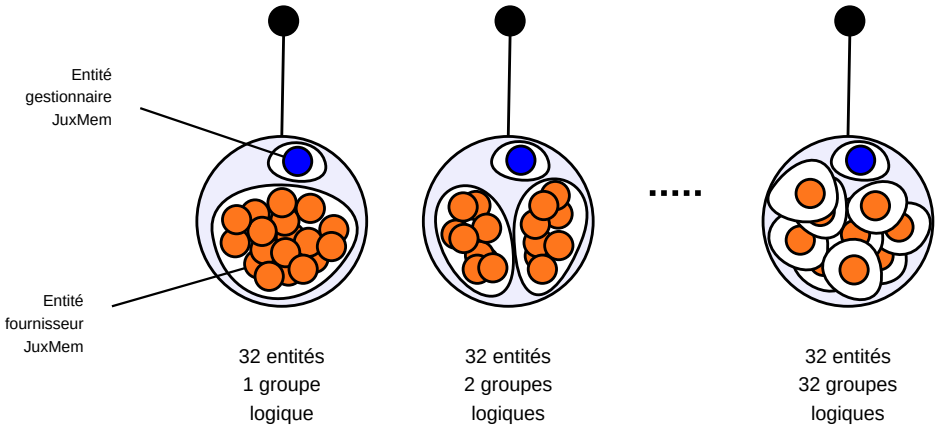

FIG. 7.4 – Évolution du nombre de groupes logiques.

La figure 7.3 correspond aux mêmes expérimentation que la figure 7.2. Elle illustre en détail, pour chacune des requêtes, le temps passé à interagir avec les outils de réservation et de déploiement. Selon la figure 7.3, le temps de réservation est constant au regard du nombre de ressources demandées. Une augmentation de 2 secondes apparaît cependant sur toutes les mesures à partir de la 13^{e} itération. Cette augmentation significative, ne peut être expliquée que par des perturbations extérieures à l'expérience, le comportement de l'outil de réservation restant stable par la suite. Le temps passé dans l'outil de déploiement augmente en fonction du nombre d'entités déployées. Il passe de 2 secondes à 7 secondes dans cette expérience.

Le travail de l'outil de déploiement est composé de deux parties.

- La première partie traite principalement de la création de la description de l'application générique, de la résolution des contraintes de placement et de la planification. Cette partie est effectuée rapidement par ADAGE et ne constitue pas l'essentiel du traitement.
- La seconde partie consiste en l'exécution de la planification. Une connexion est effectuée sur chaque ressource physique pour y transférer les fichiers de binaires et données, configurer l'environnement et attendre l'acquittement du démarrage de l'entité.

Malgré l'aspect parallèle de la deuxième partie - les entités sont déployées de manière concurrente - l'accumulation des transferts de données et l'attente des acquittements restent la principale cause de l'augmentation globale du temps de traitement.

Cette expérimentation montre que l'utilisation de groupes logiques de grande taille s'accompagne d'une augmentation du temps global de traitement de la requête, de manière affine. Ceci est directement lié aux performances de l'outil de déploiement.

Nous tentons maintenant de comparer le déploiement de plusieurs groupes logiques de taille moyenne avec le déploiement d'un groupe logique de grande taille.

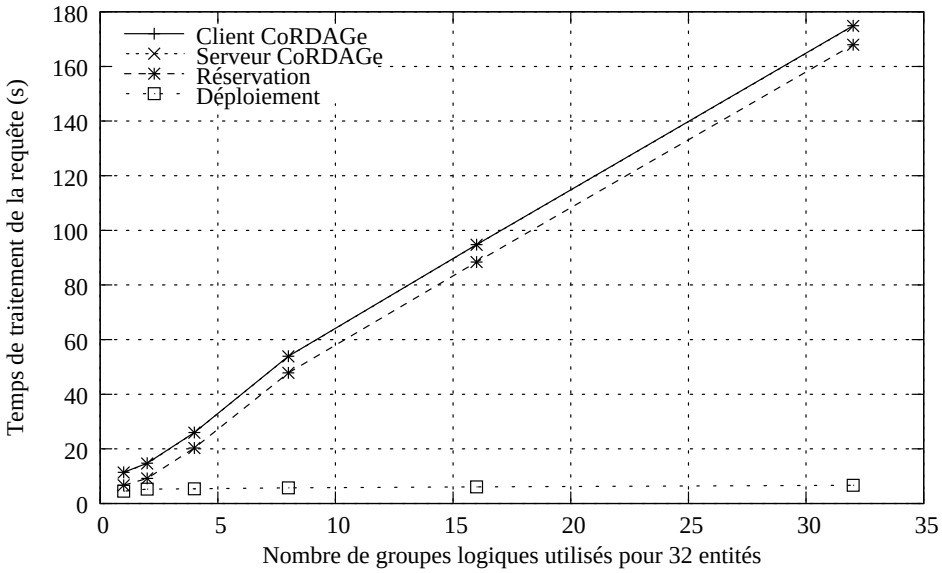

FIG. 7.5 – Impact du nombre de groupes logiques sur le temps de traitement des requêtes.

7.2.2 Impact du nombre de groupes logiques sur le temps de traitement des requêtes

Objectifs de l'expérience. Dans cette expérience nous mesurons l'impact du nombre de groupes logiques utilisés pour représenter une application, sur le temps de traitement de la requête de déploiement. La création d'un nombre important de groupes logiques est motivé par le fait qu'une fois déployé, la seule modification possible sur un groupe logique est la suppression d'une entité. Si le groupe devient vide il est supprimé et la réservation associée est terminée. Dans les systèmes de gestion des ressources comme OAR, il n'est pas possible de libérer un sous-ensemble des ressources physiques associées à une réservation. L'ensemble des ressources de la réservation doit être libéré.

Si après la suppression d'une entité, le groupe logique qui la contenait n'est pas vide, alors la réservation associée n'est pas annulée pour préserver l'exécution des entités restantes. La ressource physique supportant l'exécution de l'entité supprimée reste donc réservée mais devient *inutilisée*. Cette ressource est indisponible pour les déploiements suivants et pour les autres utilisateurs. Pour ne pas générer trop de ressources inutilisées, il est important de répartir les entités dans des groupes logiques en fonction de leur durée de vie. La création d'un seul groupe logique contenant toutes les entités n'est donc pas conseillé. C'est pourquoi dans le contexte de JUXMEM, le gestionnaire - qui doit être déployé pendant toute la durée de vie du service - n'est pas placé dans un groupe avec des fournisseurs, entités susceptibles d'être supprimées à tout moment.

Configuration utilisée. Dans cette expérience, nous mesurons le temps de déploiement de diverses configurations de la représentation logique d'une même topologie du service JUXMEM. Chaque déploiement consiste en 1 gestionnaire situé dans son propre groupe logique et d'un nombre fixe de 32 fournisseurs répartis dans successivement 1, 2, 4, 8, 16 et 32 groupes logiques. Pour cela, l'action d'ajout d'un fournisseur offerte par CORDAGE a

été étendue pour prendre un paramètre supplémentaire. Ce paramètre permet de spécifier le nombre de groupes logiques dans lesquels placer les fournisseurs. La figure 7.4 illustre les différentes topologies correspondant aux requêtes de déploiement émises par le client synthétique. Les ressources réservées par CORDAGE se situent sur la grappe *Azur* du site de Sophia.

Discussion des résultats. La figure 7.5 illustre le temps nécessaire pour traiter les requêtes de déploiement en fonction du nombre de groupes logiques utilisés pour grouper les 32 fournisseurs. Le temps global constaté sur le client augmente linéairement en fonction du nombre de groupes logiques. Il est majoritairement composé par les interactions avec l'outil de réservation. Ceci est expliqué par le fait qu'une seule demande de réservation de ressources est effectuée pour chaque groupe logique à déployer. Ces demandes sont effectuées de manière séquentielle. Dans le cas extrême, 33 réservations ont donc été nécessaires pour déployer les entités, dont une pour le gestionnaire. Le nombre de groupes logiques utilisés se traduit par autant de contraintes de placement avec les entités transmises à l'outil de déploiement. La figure 7.5 montre aussi que l'ajout de ces contraintes ne perturbe pas l'outil de déploiement. Celui-ci offre un temps constant pour les différentes configurations de déploiement de ces 33 entités.

La multiplication des groupes logiques permet plus de souplesse dans la gestion dynamique de la représentation logique. Les ressources physiques peuvent être libérées plus finement, à chaque fois qu'un groupe logique devient vide, en fonction des suppressions d'entités. Cependant, cette stratégie a un coût important en terme d'interaction avec l'outil de réservation et donc de traitement de la requête. Une solution serait la réservation par lot comme discutée dans [27]. L'idée est de demander en une seule requête, g réservations de n ressources, ce qui revient à effectuer une réservation globale ensuite divisée en plusieurs sous-réservations. Pour cela, l'outil de réservation doit offrir une telle fonctionnalité de réservation par lot, ce qui n'est pas le cas avec la version d'OAR mise en place sur GRID'5000.

7.3 Déploiement dynamique

Les expériences présentées dans la section précédente se focalisent sur la configuration statique de l'application à déployer. Dans cette section, nous étudions le comportement du prototype CORDAGE lors d'une utilisation dynamique. L'objectif est d'évaluer le temps de réponse aux requêtes de déploiement dans divers scénarios mettant en jeu plusieurs clients et applications.

7.3.1 Mesure avec expansion et rétraction

L'aspect dynamique du modèle proposé par CORDAGE s'exprime par les opérations d'ajout et de suppression d'entités. Dans cette section nous évaluons le comportement du prototype soumis à des requêtes d'expansion et de rétraction. Une topologie minimale du service JUXMEM contenant un groupe *cluster* avec uniquement un gestionnaire est déployée. Ce déploiement est suivi, pendant une heure, de demandes de déploiement d'un pair fournisseur supplémentaire. Ces demandes sont envoyées par un programme client synthétique. Afin de ne pas être limité par la disponibilité des ressources dans la grappe choisie, nous

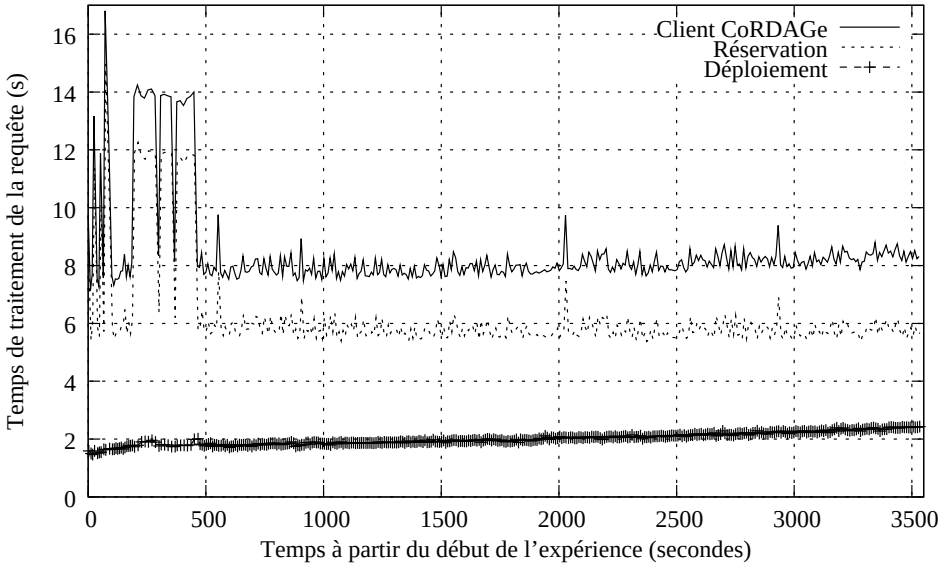

FIG. 7.6 – Comportement dynamique : ajout et suppression d'entités.

fixons le nombre maximum de fournisseurs à 20 pairs. Une fois ce nombre atteint, une requête de suppression du plus ancien fournisseur déployé est envoyée par le client synthétique, en alternance avec les requête de déploiement. Cette expérimentation est menée au sein des grappes du site de Rennes.

La figure 7.6 illustre le temps de traitement des requêtes d'ajout de fournisseurs au cours du déroulement de l'expérience. Pendant une heure, 386 entités ont été ajoutées avec un temps moyen de déploiement constaté sur le serveur d'un peu plus de 8 secondes. Les cinq cent premières secondes de l'expérience mettent en évidence quelques perturbations de l'outil de réservation, entraînant des délais dans le traitement des requêtes. Par la suite, l'allure globale de la courbe du côté du client augmente très légèrement : le temps moyen constaté sur le serveur entre la 500^{e} et la 1000^{e} seconde est de 7.9 secondes. Il augmente à 8.3 secondes entre la 3000^{e} et la 3500^{e} seconde. Cette augmentation est directement liée au comportement de l'outil de déploiement, faisant penser à un effet d'accumulation des requêtes sur l'état interne de l'outil. Les raisons exactes ne sont cependant pas connues à ce jour et demanderaient une analyse plus fine du comportement d'ADAGE.

Une expérience similaire a été conduite sur le site de Sophia. Dans cette expérience nous introduisons un délai de 30 secondes entre les requêtes d'ajout ou de suppression d'entités pour diminuer la fréquence des interactions avec les outils de gestion des ressources et de déploiement. Cette expérimentation mène à des résultats similaires à la version menée sur le site rennais. La fréquence des requêtes n'est donc pas en cause dans la légère augmentation du temps de traitement de l'outil de déploiement.

Les expérimentations présentées dans cette section montrent que CORDAGE est pleinement tributaire des performances des outils de réservation et de déploiement tout au long de son exécution. Aucune augmentation du temps de traitement des requêtes n'a été constaté sur le serveur CORDAGE au cours du déroulement des expérimentations.

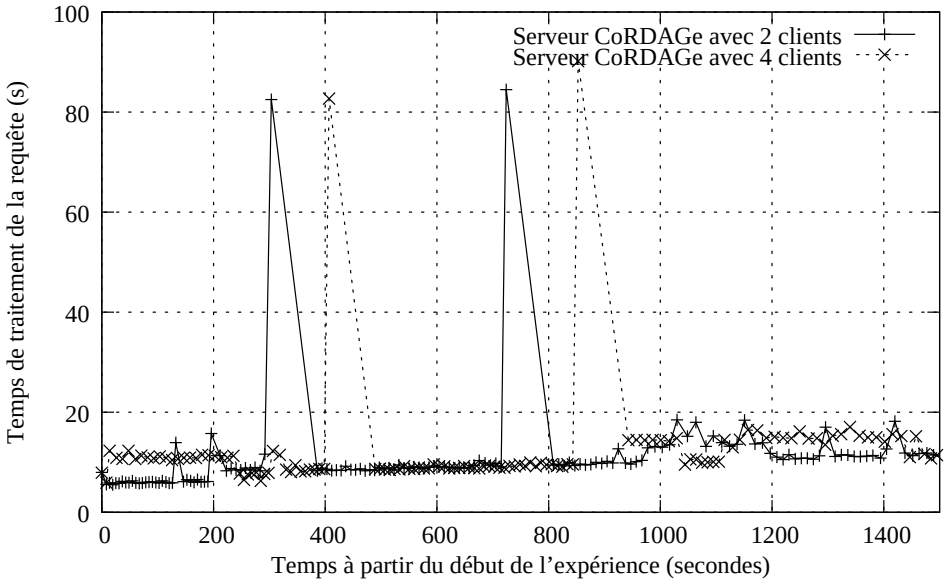

FIG. 7.7 – Comportement dynamique : ajout d'entités avec plusieurs clients sur une même application. Temps sur les serveurs.

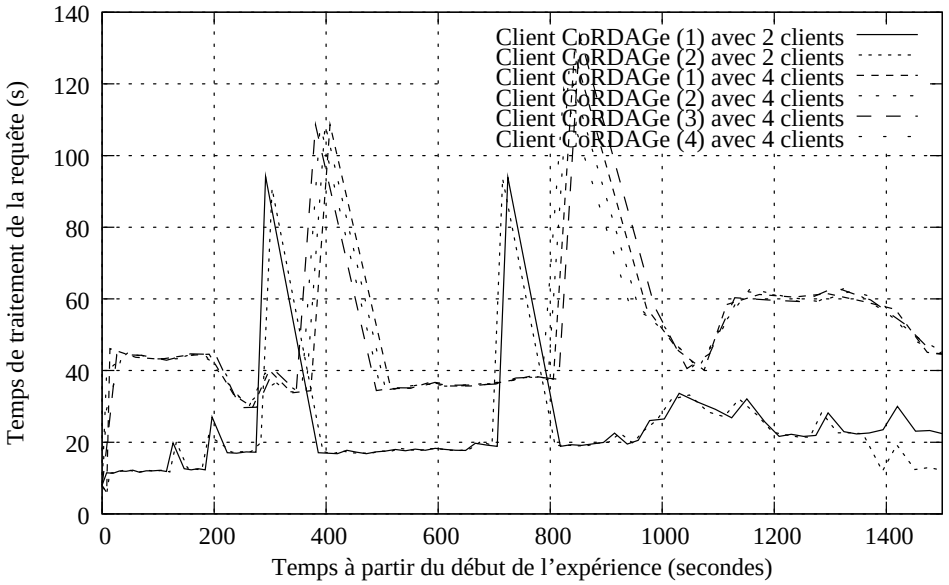

FIG. 7.8 – Comportement dynamique : ajout d'entités avec plusieurs clients sur une même application. Temps sur les clients.

7.3.2 Configuration multi-client : impact de la concurrence

Les expérimentations présentées dans les sections précédentes montrent que le prototype CORDAGE, couplé avec les outils OAR et ADAGE, offre une solution fonctionnelle pour prendre complètement en charge des configurations simples de déploiement. Dans cette section, nous étudions des configurations de déploiement plus avancées. Elles mettent en scène plusieurs clients CORDAGE, plusieurs applications ou encore du déploiement sur

toute la grille.

Objectifs de l'expérience. Une première configuration avancée peut mettre en jeu plusieurs clients CORDAGE effectuant des requêtes portant sur la même application. Ce scénario est par exemple utile dans le cas d'une application distribuée, organisée en plusieurs groupes logiques dans chacun desquels une entité est responsable d'interagir avec le serveur CORDAGE. La dynamicité y est donc gérée à l'échelle locale. Afin d'évaluer le surcoût lié à l'utilisation de plusieurs clients, nous reprenons le principe de l'expérience précédente, c'est-à-dire le déploiement continu de nouveaux fournisseurs dans un service JUXMEM, mais cette fois-ci les requêtes sont émises par deux puis quatre clients synthétiques indépendants. Contrairement à l'expérience précédente, nous n'effectuons aucune suppression d'entités. Les ressources sélectionnées pour cette expérience appartiennent au site de Bordeaux.

Coût mesuré sur les serveurs. La figure 7.7 illustre les temps mesurés pour traiter les requêtes de déploiement *sur le serveur* CORDAGE , dans *les configurations impliquant deux puis quatre clients*. Les mesures affichées concernent toutes les requêtes de déploiement, quel que soit le client qui en est à l'origine. La première observation est que les temps sont sensiblement similaires dans les deux configurations. Le nombre de clients différents n'impacte pas le coût de traitement des requêtes car elles sont traitées de manière séquentielle. La deuxième observation est que l'on retrouve la légère pente liée à l'outil de déploiement et déjà observée dans l'expérience précédente. Cette pente peut aussi ici s'expliquer par l'accumulation des contraintes de placement et l'augmentation de la représentation logique à gérer dans l'outil de déploiement. Le fait ne pas supprimer les fournisseurs entraîne la saturation en terme de réservations des différentes grappes sur le site bordelais : les deux pics observés sur chacune des courbes correspondent aux transitions entre les grappes *Bordemer*, *Bordeplage* et *Bordereau*. L'ampleur de ces pics s'explique par le délai d'attente imposé par CORDAGE avant de déclarer que la grappe courante ne peut pas être utilisée. Ce délai est caractéristique d'une réponse positive de la part de l'outil de réservation mais ne pouvant être honorée immédiatement.

Coût mesuré sur les clients. La figure 7.8 montre les temps mesurés *sur les clients* CORDAGE pour traiter les requêtes de déploiement, dans les configurations avec deux et quatre clients. Le temps moyen observé dans l'expérience précédente mettant en jeu un client est ici doublé puis quadruplé pour, respectivement, deux et quatre clients concurrents. Ceci s'explique par la nécessité de partager la même fiche de suivi pour tous les clients. Afin de conserver des données cohérentes, l'accès à cette fiche nécessite l'obtention d'un verrou. Il en résulte la séquentialisation des requêtes et donc des temps d'attente supplémentaires lorsque les clients sont nombreux.

Ce problème pourrait être minimisé grâce à la mise en place de verrous permettant un contrôle plus fin des accès : un verrou par variable d'état dans la fiche de suivi et des verrous plus ciblés sur certains sous-arbres de la représentation logique. Les deux pics présents sur les mesures du serveur se répercutent sur les courbes des clients. Ces pics présentent un décalage entre les clients et donnent une information sur l'ordre dans lequel les clients sont servis.

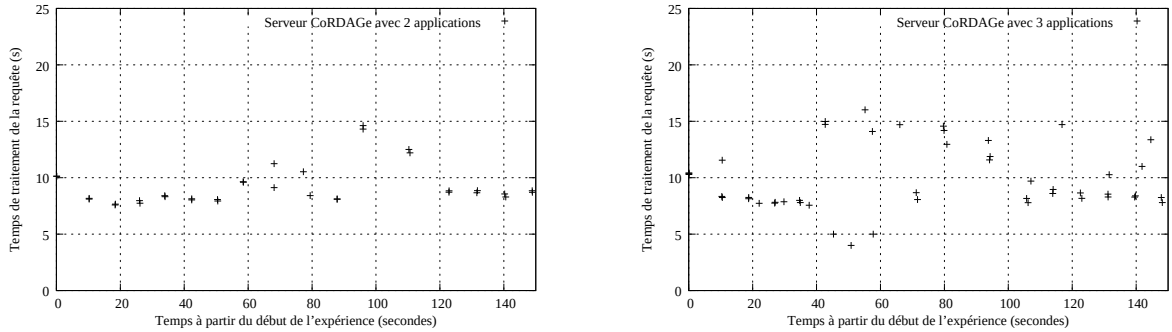

FIG. 7.9 – Temps de traitement des requêtes sur le serveur. (a) Comportement dynamique : ajout d'entités avec 2 applications. (b) Comportement dynamique : ajout d'entités avec 3 applications.

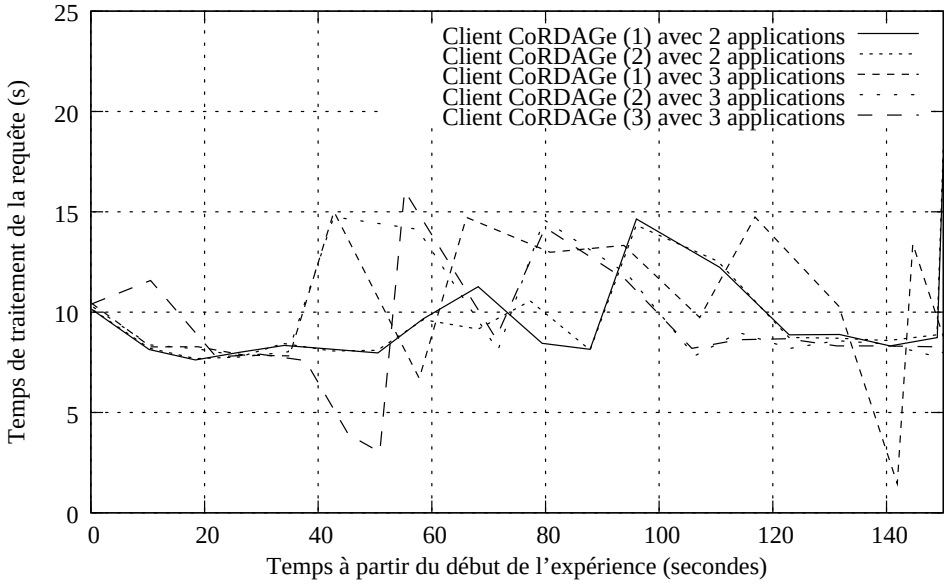

FIG. 7.10 – Comportement dynamique : ajout d'entités avec plusieurs applications. Temps sur les clients.

7.3.3 Configuration multi-application

Le prototype évalué prend en compte l'aspect co-déploiement présenté dans le modèle CORDAGE. Il permet en effet la gestion, au sein de la même instance du serveur, de plusieurs fiches de suivi correspondant à des applications différentes, éventuellement de types différents. Dans cette expérimentation nous observons le comportement de CORDAGE en présence de deux puis trois applications déployées. Nous reprenons le principe de l'expérience d'ajout et de suppression d'entités présentée dans la section 7.3.1. Dans cette expérience, nous lançons plusieurs clients ayant la charge de déployer chacun leur propre service JUXMEM. Autant de fiches de suivi sont donc créées et gérées de manière indépendantes par le serveur. Bien que les fiches de suivi se partagent le même outil de réservation, il est à noter qu'une instance spécifique de l'outil de déploiement est créée pour chacune d'entre elles. Ceci permet des déploiements en parallèle pour les différents services JUXMEM. Les

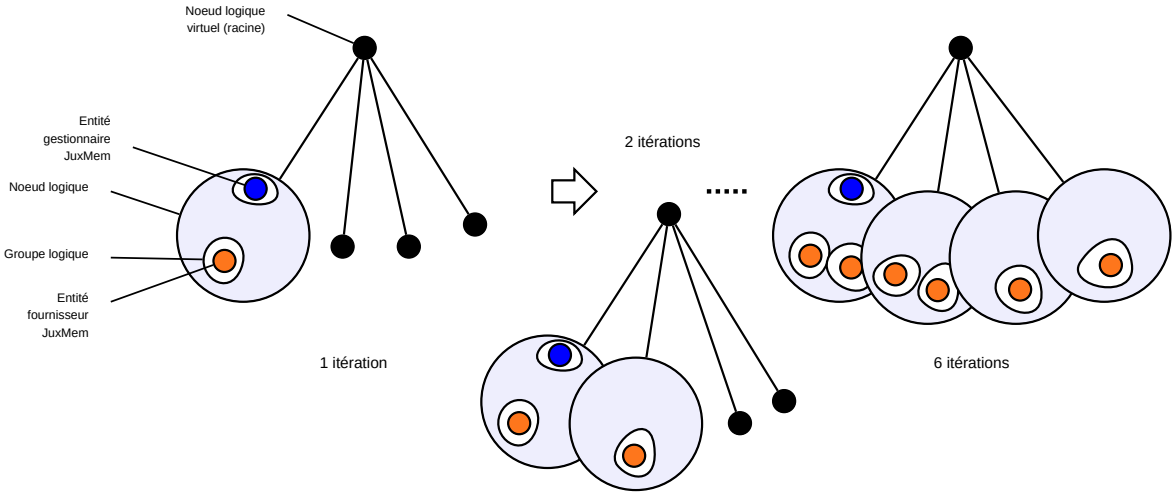

FIG. 7.11 – Évolution de la représentation logique pour les expérimentations multi-site (4 sites dans cet exemple). Chaque nœud logique est projeté sur un site différent.

ressources choisies pour cette expérience sont issues du site de Bordeaux.

La figure 7.9 montre les temps de traitement des requêtes de déploiement mesurés sur le serveur, dans les configurations comprenant deux et trois applications. Chacun des points correspond donc au temps de traitement d'une demande de déploiement, en fonction du temps depuis le début de l'expérience. C'est pourquoi deux points peuvent se trouver très proches sur l'axe des abscisses. Ils appartiennent à des clients différents. Les figures 7.9(a) et 7.9(b) donnent deux informations sur le temps de traitement.

- Il est stable et comparable aux expériences effectuées avec un seul client (figure 7.6).
- Il n'est pas augmenté de manière significative par l'ajout d'une application supplémentaire à gérer.

Ces deux points s'expliquent par la parallélisation des requêtes portant sur les différentes fiches de suivi et la bonne gestion de ces requêtes parallèles effectuée par l'outil de réservation. La figure 7.10 montre que, contrairement à l'expérience précédente portant sur une seule application, les clients des différentes applications ne souffrent pas des temps d'attente liés à la prise de verrou pour accéder aux fiches de suivi.

7.3.4 Configuration multi-site : impact de l'échelle

Les expériences présentées précédemment ne concernent que des déploiements au sein du même groupe de ressources physiques. Du point de vue de la représentation logique dans le modèle CORDAGE, un seul nœud logique est créé et projeté sur un site de la grille. Dans cette section, nous évaluons le comportement du prototype dans une configuration multi-site. Pour cela, nous prenons comme scénario motivant une topologie du service JUXMEM contenant quatre puis six *groupes cluster*. Chacun de ces groupes est représenté par un nœud logique dans l'arbre logique de CORDAGE. Les nœuds logiques sont ensuite projetés sur des groupes de ressources physiques correspondant à différents sites de la grille. Des fournisseurs sont alors déployés, à tour de rôle, dans les différents *groupes cluster*, à l'initiative d'un client synthétique.

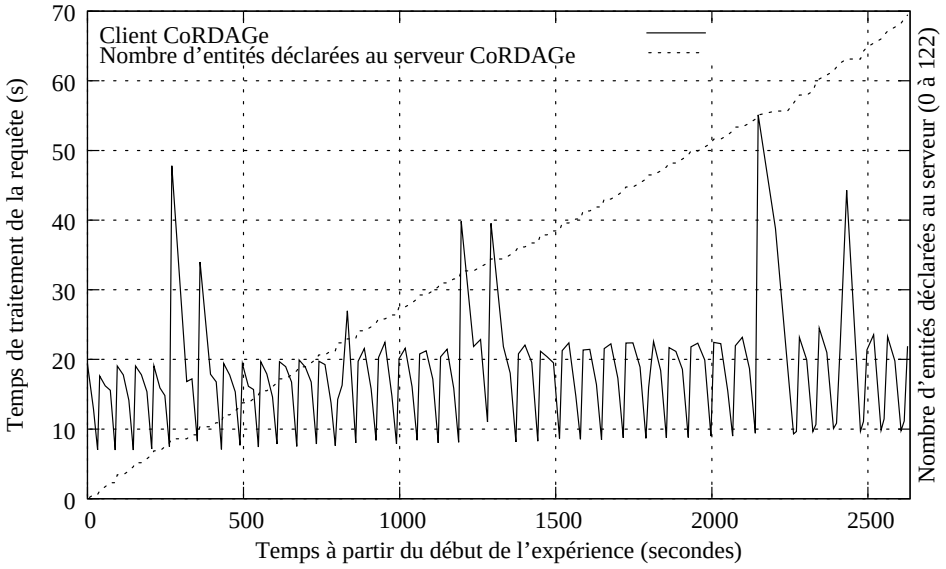

FIG. 7.12 – Comportement dynamique : ajout d'entités dans 4 sites.

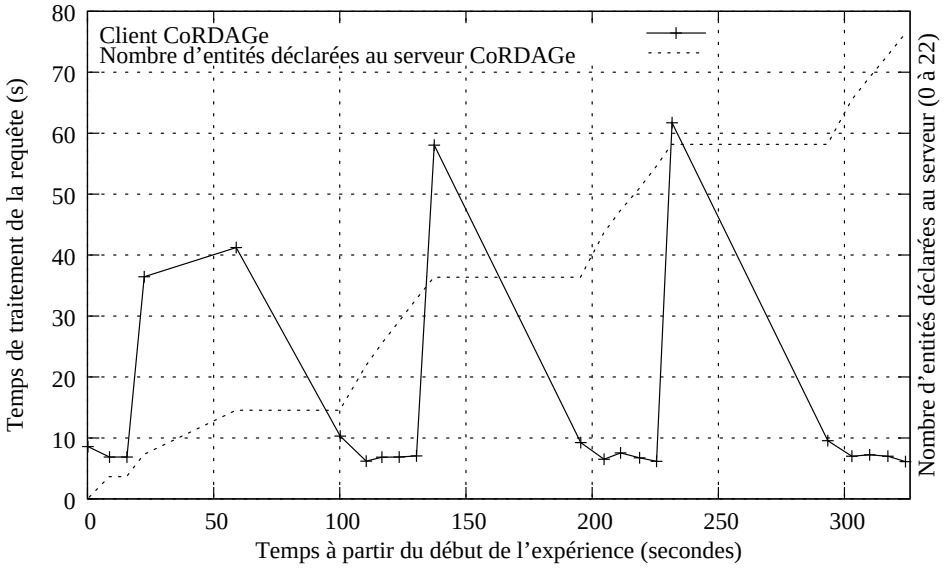

FIG. 7.13 – Comportement dynamique : ajout d'entités dans 6 sites.

La figure 7.12 illustre le temps de traitement des requêtes sur le client pour le déploiement de fournisseurs JUXMEM sur 4 sites de GRID'5000. Elle indique en plus le nombre d'entités ayant envoyé leur message d'acquittement au serveur CORDAGE. Les sites de la grille choisis par CORDAGE sont Bordeaux, Rennes, Sophia et Toulouse. La courbe met en valeur le cycle des réservations sur les différents sites. Chaque site est caractérisé par un temps moyen pour la réservation des ressources. Le site le plus réactif est Rennes, ce que l'on peut expliquer par la présence du serveur CORDAGE sur ce site.

Une vue plus fine du phénomène des cycles de réservation est proposée par la figure 7.13. Dans cette expérience, six *groupes cluster* sont créés pour le service JUXMEM. Les sites retenus par CORDAGE pour le déploiement sont Bordeaux, Lille, Nancy, Rennes, Orsay et Toulouse. Le site ayant le temps de réservation le plus élevé est Orsay, principalement dû à une activité importante et à la difficulté de trouver des ressources disponibles. Ces expériences introduisent des différences de performances de l'outil de réservation en fonction des ressources sélectionnées. Ces performances sont prises en compte par CORDAGE par un système d'expiration de la requête : si les ressources ne sont pas devenues disponibles dans un certain délai, la réservation est annulée et la fonction de projection sélectionne un autre site. Ce délai est aujourd'hui indiqué de manière fixe, mais nous pouvons imaginer le rendre adaptable en fonction du taux de succès des réservations ou de l'intérêt particulier que l'on peut avoir à réserver dans un site.

7.4 Conclusion

L'évaluation du prototype CORDAGE permet de montrer que le surcoût lié à son utilisation est négligeable dans la chaîne complète des opérations à effectuer pour déployer une application. Le temps passé à traiter les requêtes est en grande partie utilisé par les outils de réservation et de déploiement de la grille. En ce sens, CORDAGE ne permet pas aujourd'hui d'améliorer le coût en temps des déploiements : son but est avant tout d'offrir de nouvelles fonctionnalités aux applications. Ces fonctionnalités permettent principalement de masquer l'évolution de la topologie logique de l'application et les interactions avec les outils de réservation et de déploiement installés sur la grille.

La diminution du coût en temps de déploiement nécessite des techniques supplémentaires de parallélisation des demandes de réservation sur différents sites ou de préréservation de ressources sont autant de pistes à explorer pour améliorer les temps d'interaction avec les outils de réservation.

L'étude des configurations statiques met en valeur l'importance du choix de la représentation logique dans CORDAGE. L'utilisation de multiples groupes logiques permet une gestion plus fine de la durée de vie des entités et de la réservation des ressources qui leur sont associées. En contrepartie, ceci s'accompagne d'une augmentation importante du temps de traitement liée aux multiples réservations. La répartition des entités dans les groupes logiques doit donc être effectuée comme un compromis entre les performances et la finesse de gestion du déploiement. Ce compromis est spécifique à chaque type d'application.

L'aspect dynamique met en valeur la stabilité du prototype face aux différentes configurations de déploiement. CORDAGE permet à plusieurs clients de travailler sur une même application, mais aussi de gérer plusieurs applications en même temps.

Chapitre 8

Conclusion et perspectives

Conclusion

Contexte d'étude

Les *applications scientifiques* sont aujourd'hui largement utilisées pour concevoir des systèmes complexes, simuler des modèles ou encore organiser et traiter la connaissance. Elles sont au service de nombreux domaines de recherche, tels que la physique des particules, le génie civil, l'astronomie... La plupart de ces applications n'ont pas de limites en terme de puissance de calcul et d'espace de stockage. Profitant des avancées technologiques effectuées dans les réseaux à longue distance, les *grilles informatiques* se positionnent comme une approche sérieuse pour le support de l'exécution des applications scientifiques. Elles consistent à mutualiser des ressources réparties dans divers instituts, universités ou entreprises. Si ce type d'infrastructure s'avère être séduisant en terme de capacité de calcul et de stockage, les difficultés d'utilisation freinent encore largement son développement. Ces difficultés sont principalement dues à la nature même de la grille : l'agrégation de ressources hétérogènes, volatiles, appartenant à des domaines d'administration différents.

Dans ce contexte, le *déploiement des applications* prend tout son sens : ce qui s'avère être une étape anodine sur des architectures centralisées se révèle désormais être une étape ardue pour les grilles de calcul. Le déploiement demande, de la part de l'utilisateur, un travail supplémentaire et préliminaire à chaque expérimentation ou lancement d'application. Ce travail consiste à découvrir et sélectionner des ressources physiques, à transférer des programmes et des données, à configurer les ressources sélectionnées, à coordonner le lancement des programmes, à surveiller le déroulement de l'exécution et enfin à récupérer les résultats du calcul. L'ensemble de ces opérations est d'autant plus difficile à effectuer qu'il faut prendre en compte les caractéristiques de chaque site de la grille, comme les mécanismes de sécurité, les politiques d'utilisation des ressources ou encore la nature des outils mis à disposition.

Dans ce manuscrit, nous avons considéré deux problèmes plus complexes que le simple

déploiement d'application sur les grilles. Le premier problème provient de la dynamicité de l'infrastructure. Celle-ci entraîne le besoin de gérer l'exécution des applications de manière *dynamique*. Il doit être possible de faire évoluer la topologie de l'application en fonction de ses besoins et de l'état des ressources. Le *redéploiement* permet justement de déployer ou reconfigurer des constituants d'une application en cours d'exécution. Le deuxième problème étudié dans ce manuscrit est relatif au besoin de déployer, de manière coordonnée, des applications de types différents. Ce type de déploiement est appelé *co-déploiement*.

Aujourd'hui, il existe des solutions pour automatiser le déploiement d'applications sur grilles de calcul. Ces solutions sont plus ou moins élaborées : du simple script de conception *artisanale* à l'environnement autonome complexe, en passant par les infrastructures de *cloud computing*. Nous avons défini trois propriétés permettant de caractériser les systèmes prenant en charge le déploiement : 1) la *transparence* vis-à-vis de la gestion des ressources physiques ; 2) le caractère *spécifique* des services offerts aux applications en terme de déploiement ; et enfin 3) la *non-intrusivité* du système dans le modèle de programmation ou dans le code de l'application. À notre connaissance, aucun des systèmes étudiés ne remplit ces trois propriétés.

Contributions

Les résultats de ces travaux nous permettent de proposer un modèle pour le redéploiement et le co-déploiement d'applications distribuées sur grilles de calcul. Ce modèle, ainsi que l'architecture qui l'accompagne, tente de respecter les propriétés de transparence, de spécificité ainsi que de non-intrusivité. La transparence consiste à masquer la gestion des ressources physiques et, en règle générale, toutes les interactions avec les outils et services de la grille. La spécificité consiste à prendre en compte des besoins, en terme de déploiement, très spécifiques à l'application. Enfin, la non-intrusivité garantit de ne pas imposer un modèle de programmation pour l'application, ni d'en modifier exagérément le code. Le modèle que nous proposons est nommé CoRDAGe pour co-déploiement et redéploiement d'applications génériques sur grille.

CoRDAGe offre deux fonctionnalités principales : la première fonctionnalité consiste à traduire des actions de haut niveau, spécifiques à l'application, en des opérations de bas niveau comme la réservation de ressources physiques. La deuxième fonctionnalité consiste en une pré-planification du déploiement, rendue possible grâce à une représentation logique de l'application et des ressources.

Un modèle pour le déploiement dynamique. Le modèle proposé définit plusieurs phases : elles permettent la construction d'une représentation de l'application ainsi qu'une représentation des ressources physiques. Nous avons choisi, pour ces représentations, d'utiliser des arbres logiques. Ce choix est motivé par l'aspect hiérarchique des applications et des ressources considérées dans ce travail. Les arbres logiques sont utilisés pour représenter les évolutions de la topologie des applications par ajout et retrait de sous-arbres. Ils permettent aussi d'effectuer des opérations de fusion dans le cas du co-déploiement et des opérations de projection lors de la phase de pré-planification. Les actions de haut niveau prises en charge par ce modèle entraînent une altération des représentations logiques ainsi que des interactions avec les outils et services de la grille. Ce modèle offre les propriétés de transparence et de spécificité.

CoRDAGe : proposition d'architecture. Le modèle proposé pour le déploiement dynamique s'accompagne d'une proposition d'architecture appelée CoRDAGe. Cette architecture offre la propriété de non-intrusivité. Nous avons choisi, pour des raisons de simplicité, de faire reposer cette architecture sur le principe de l'appel de procédure à distance. Un serveur CoRDAGe a la charge de gérer un ensemble d'applications. Pour chacune d'entre elles, une fiche de suivi est créée, contenant la représentation logique ainsi qu'un jeu d'actions spécifiques à l'application. Les opérations génériques de fusion et de projection sont implémentées et fonctionnelles. Le serveur CoRDAGe offre une interface transparente aux outils de réservation et de déploiement. L'implémentation du prototype s'est déroulée dans le contexte de Grid'5000 avec les outils Oar et ADAGE. Le prototype a permis de conduire des expérimentations sur les neuf sites de la grille et d'évaluer les performances d'une telle approche. Celui-ci montre que le surcoût lié à l'utilisation de CoRDAGe n'est pas significatif face aux délais requis par les outils de réservation et de déploiement. CoRDAGe fait l'objet d'un dépôt à l'agence pour la protection des programmes (APP).

Support pour le déploiement d'applications distribuées. Afin d'appréhender toute la difficulté liée au déploiement d'applications distribuées sur des grilles de calcul, nous avons étudié, dans un premier temps, les mécanismes de déploiement pour différentes applications. Ces applications sont basées sur la plate-forme pair-à-pair Jxta, sur le service de partage de données JuxMem et sur le système de fichiers distribué Gfarm. Cette étude a permis dans un premier temps de concevoir des *plugins* pour chacune de ces applications. Les *plugins* ont été développés dans le cadre de l'outil de déploiement ADAGE et sont dédiés à l'automatisation du déploiement de manière statique. Ils ont été valorisés grâce à des expériences concernant des thématiques de recherche annexe. Dans un deuxième temps, ces applications ont servi d'études de cas pour valider l'architecture CoRDAGe. De nouveaux greffons ont été développés pour CoRDAGe permettant de construire les représentations logiques de ces applications. JuxMem a été légèrement modifié pour interagir avec le serveur CoRDAGe. Cette modification a permis de valider la boucle complète pour rendre une application dynamique et autonome avec CoRDAGe. La notion de co-déploiement a été validée en couplant JuxMem et Gfarm. Ce travail s'est déroulé au sein du projet ANR LEGO.

Le travail effectué dans le cadre de cette thèse a permis de proposer un modèle, ainsi qu'une architecture visant à simplifier la prise en charge du déploiement dynamique. Ces propositions ont été validées et expérimentées au sein d'un projet de recherche ainsi que d'une infrastructure de calcul. Il reste cependant de nombreux points à approfondir, notamment pour démontrer la pertinence de l'approche par rapport aux différents types d'applications. Pour cela, une intégration dans divers projet semble nécessaire. À titre d'exemple, CoRDAGe pourrait être utilisé par les environnements d'exécution tels que zorilla et Vigne pour gérer leur topologie. Des expérimentations inter-grilles pourraient aussi être menées, entre Grid'5000, DAS-3 aux Pays-Bas et Omni-RPC au Japon. Dans la section suivante nous développons des perspectives à plus long terme.

Perspectives

Approche par programmation par contraintes

La représentation des applications et des ressources est effectuée à l'aide d'arbres logiques. Le modèle proposé par CORDAGE se traduit par des opérations sur ces représentations logiques, notamment par des opérations de fusion et de projection. La fusion de deux arbres est commandée lors d'un co-déploiement d'applications. Cette fusion doit être effectuée en respectant des *contraintes* hiérarchiques induites par la structure des arbres. De plus, des contraintes de co-localisation peuvent lier des entités entre elles et imposer la fusion des deux groupes logiques dont elles font partie. L'ensemble de ces contraintes doit être pris en compte lors de la fusion de deux représentations logiques. Des contraintes doivent aussi être prises en compte lors de la phase de projection de l'application sur les ressources physiques. Cette projection doit non seulement respecter les contraintes hiérarchiques induites par les arbres, mais aussi respecter des contraintes relatives à la réservation des ressources physiques : une réservation doit pouvoir être effectuée avec succès pour chaque groupe logique appartenant à la représentation de l'application.

Les algorithmes implémentés aujourd'hui dans CORDAGE ne permettent pas de proposer une solution prenant en compte ces contraintes de manière *optimale* et *efficace*. Ces algorithmes sont restés dans un état préliminaire car l'objectif du prototype présenté dans ce manuscrit est de montrer la validité de l'approche CORDAGE. Cependant, en guise de perspective pour ce travail, nous pouvons imaginer des algorithmes tirant parti de la *programmation par contraintes*. Ce type de programmation permet de poser un problème sous forme de relations logiques entre plusieurs variables. Dans le contexte CORDAGE, ces variables doivent représenter le résultat d'une fusion ou d'une projection entre deux arbres logiques. Cette tâche est d'autant plus difficile que, dans le cas d'une projection, les contraintes évoluent dans le temps. En effet, l'ensemble des ressources physiques libres peut évoluer entre le début et la fin de la phase de projection. Des travaux [126] sont par exemple menés sur la programmation par contraintes en environnement dynamique au sein de l'équipe *Contraintes* du laboratoire Lina à Nantes.

Intégration dans un environnement autonome

La vision de l'informatique autonome introduite par IBM dans [72] s'accompagne d'une proposition d'architecture comprenant les quatre phases de surveillance, de prise de décision, de planification et d'exécution (voir la section 2.2.2). CORDAGE s'inscrit pleinement dans la phase d'exécution. Il n'a *pas* pour vocation d'effectuer la surveillance des applications et des ressources, ni d'intégrer des outils de prise de décision. Son rôle consiste avant tout à interpréter des actions élaborées lors de la phase de planification du modèle autonome. De plus, ces actions sont limitées aux considérations liées au déploiement. Si CORDAGE ne peut être considéré comme un système autonome complet, il peut cependant apporter une contribution à ce modèle, en simplifiant la phase d'exécution : les actions à exécuter deviennent des actions de haut niveau, ayant un sens spécifique à l'application. Le système autonome profite même de la transparence de gestion des ressources offerte par CORDAGE.

L'une des perspectives de ces travaux consiste à intégrer CORDAGE dans un environnement autonome tel que DYNACO (voir section 2.2.2.2) comme cela est suggéré en section 6.4.

Dans cette approche, la phase de planification peut générer des appels au serveur CoRDAGe en demandant le traitement d'une action spécifique à l'application. DYNACO devient donc le principal client de CoRDAGe : les applications n'ont plus à connaître l'existence de CoRDAGe. Elles ne font plus aucun appel au serveur. Cette approche permet de renforcer la propriété de non-intrusivité par rapport au modèle de programmation choisi pour l'application.

Vers un CoRDAGe distribué

L'approche retenue pour l'architecture CoRDAGe est basée sur une instance d'un serveur CoRDAGe par groupe d'applications déployées. En pratique, l'utilisateur ne déploie bien souvent qu'une seule application par expérience : un serveur est donc lancé pour chaque application déployée. Éventuellement, dans le cas d'un co-déploiement, un serveur peut être utilisé pour gérer plusieurs applications lancées par un même utilisateur. Cette approche a pour principal avantage, à l'échelle d'une grille multi-utilisateur, d'éliminer tout problème de congestion sur un serveur centralisé. En effet, la gestion des applications est naturellement répartie sur les serveurs CoRDAGe des utilisateurs. Elle souffre cependant d'un défaut majeur : si un serveur disparaît, toutes les fiches de suivi des applications dont il avait la charge sont perdues. Le support pour la dynamicité n'est plus assuré pour ces applications. Ce problème n'a pas été traité dans le cadre de cette thèse car il existe déjà des solutions, issues de l'algorithmique distribuée, pour pallier la défaillance d'un processus. Nous pouvons cependant donner quelques pistes originales.

Une première approche consiste à stocker les données sur un support considéré comme permanent. Par exemple, nous pouvons envisager de ne plus gérer les fiches de suivi uniquement en mémoire locale, mais de les stocker dans un service de partage de données tel que JUXMEM. Une deuxième approche consiste à profiter des différentes instances des serveurs CoRDAGe déjà déployées, afin d'y stocker des copies des fiches de suivi. Ceci peut être mis en place sur le modèle pair-à-pair, avec le maintien d'une liste de serveurs voisins sur lesquels il est possible de sauvegarder régulièrement les fiches de suivi. En cas de défaillance d'un serveur, l'application doit être capable de demander à un serveur voisin de prendre la relève en attendant le démarrage d'un nouveau serveur.

Cette deuxième approche peut être généralisée pour construire un réseau pair-à-pair entre les serveurs CoRDAGe. Ce réseau sert alors de support pour échanger des informations sur les réservations entre des serveurs voisins. La notion de voisinage peut être construite suivant divers critères : le critère de localité physique permet de sélectionner comme voisins des serveurs gérant des applications déployées sur des ressources proches. Mais on peut imaginer aussi des critères sémantiques, permettant de regrouper des serveurs ayant un intérêt en commun. Nous touchons ici à la notion de communauté CoRDAGe.

Annexe A

Annexes

Sommaire

A.1 Représentations logiques dans CORDAGE

Le listing A.1 propose la grammaire au format XSD utilisée pour définir l'ensemble des arbres au format XML considérés comme valide pour représenter une application dans CORDAGE.

Listing A.1 – Grammaire au format XSD pour la représentation des arbres logiques en XML.

```
1  <?xml version="1.0"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3
4  <xsd:element name="CORDAGE_LOGICAL_TREE">
5   <xsd:complexType>
6    <xsd:sequence>
7     <xsd:element name="logicaltree" type="logicaltreeType"
8                  minOccurs="0" maxOccurs="1" />
9    </xsd:sequence>
10  </xsd:complexType>
11 </xsd:element>
12
13 <xsd:complexType name="logicaltreeType">
14   <xsd:sequence>
15    <xsd:element name="logicalnode" type="logicalnodeType"
16                 minOccurs="1" maxOccurs="1" />
```

```
   <xsd:element name="children" type="childrenType"
                minOccurs="1" maxOccurs="1" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="physical" type="xsd:string" use="required" />
</xsd:complexType>

<xsd:complexType name="childrenType">
  <xsd:sequence>
   <xsd:element name="logicaltree" type="logicaltreeType"
                minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="logicalnodeType">
  <xsd:sequence>
   <xsd:element name="logicalgroup" type="logicalgroupType"
                minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="logicalgroupType">
  <xsd:sequence>
   <xsd:element name="entity" type="entityType"
                minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="jobid" type="xsd:integer" use="required" />
  <xsd:attribute name="deployed" type="xsd:boolean" use="required" />
  <xsd:attribute name="application" type="xsd:string" use="required" />
</xsd:complexType>

<xsd:complexType name="entityType">
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="cardinality" type="xsd:integer" use="required" />
  <xsd:attribute name="acked" type="xsd:boolean" use="required" />
  <xsd:attribute name="connectedto" type="xsd:string" use="required" />
</xsd:complexType>

</xsd:schema>
```

Le listing A.2 propose la grammaire au format XSD utilisée pour définir l'ensemble des arbres au format XML considérés comme valide pour représenter les ressources dans CORDAGE.

Listing A.2 – Grammaire au format XSD pour la représentation des arbres physiques en XML.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="CORDAGE_PHYSICAL_TREE">
 <xsd:complexType>
```

```
  <xsd:sequence>
    <xsd:element name="grid" type="gridType"
                 minOccurs="0" maxOccurs="unbounded" />
  </xsd:sequence>
 </xsd:complexType>
</xsd:element>

<xsd:complexType name="gridType">
  <xsd:sequence>
   <xsd:element name="site" type="siteType"
                minOccurs="1" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="used" type="xsd:boolean" use="required" />
</xsd:complexType>

<xsd:complexType name="siteType">
  <xsd:sequence>
    <xsd:element name="cluster" type="clusterType"
                 minOccurs="1" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="used" type="xsd:boolean" use="required" />
</xsd:complexType>

<xsd:complexType name="clusterType">
  <xsd:attribute name="name" type="xsd:ID" use="required" />
  <xsd:attribute name="used" type="xsd:boolean" use="required" />
</xsd:complexType>

</xsd:schema>
```

A.2 Exemples d'utilisation de CORDAGE

Le listing A.3 propose un exemple complet d'utilisation du greffon JUXMEM pour CORDAGE.

Listing A.3 – Un exemple de client JUXMEM complet utilisant CORDAGE.

```
#include "cdgclient.hh"

#include <string>

using namespace std;

int
main(int argc, char **argv) {

  /* cardinalité, nom du groupe cluster */
  string params_provider = "3␣mycluster";

```

```
13   /* nom du groupe cluster, id client, port, binaire, cardinalité, arguments */
14   /* ceci est l'identifiant du groupe cluster où nous voulons déployer le client*/
15   string params_client = "mycluster";
16   params_client += "␣";
17   /* l'identifiant du client sera généré par le serveur et retourné en résultat */
18   params_client += CORDAGE_NONE;
19   /* port, binaire et cardinalité */
20   params_client += "␣6969␣omnipeer␣1";
21   /* paramètres du programme JuxMem Omnipeer, voir la documentation ADAGE-JuxMem*/
22   params_client += "␣@peerid␣@groupid␣@tcpport␣writer␣10";
23   params_client += "␣@managerhostname␣@managertcpport␣1␣10";
24
25   cdgclient* cordage = new cdgclient();
26
27   if (cordage->_success) {
28
29     cordage->deploy();
30
31     cordage->performaction(CORK_JUXMEM2_ADD_CLIENT, params_client);
32
33     sleep(10);
34
35     cordage->performaction(CORK_JUXMEM2_ADD_CLIENT, params_client);
36
37     cordage->performaction(CORK_JUXMEM2_ADD_PROVIDER, params_provider);
38
39     cordage->performaction(CORK_JUXMEM2_ADD_CLUSTER);
40
41     cordage->terminate();
42
43   }
44
45   delete(cordage);
46
47   return 0;
48 }
```

Références

[1] D. Abramson, R. Sosic, J. Giddy, and B. Hall. Nimrod: a tool for performing parametrised simulations using distributed workstations. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing (HPDC '95)*, page 112, Washington, DC, USA, 1995. IEEE Computer Society.

[2] Vikas Agarwal, Gargi Dasgupta, Koustuv Dasgupta, Amit Purohit, and Balaji Viswanathan. DECO: data replication and execution co-scheduling for utility grids. In *International Conference on Service Oriented Computing (ICSOC 2006)*, volume 4294 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2006.

[3] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, 2002.

[4] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The Grid Application Toolkit: toward generic and easy application programming interfaces for the Grids. *Proceedings of the IEEE*, 93(3):534–550, March 2005.

[5] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, John Shalf, and Ian Taylor. Enabling applications on the Grid: a GridLab overview. *International Journal of High Performance Computing Applications*, 17:449–466, 2003.

[6] Voichita Almasan. A monitoring tool to manage the dynamic resource requirements of a grid data sharing service. Rapport de master de recherche, Université de Rennes 1, 2006.

[7] David P. Anderson. BOINC: a system for public-resource computing and storage. *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (Grid 2004)*, pages 4–10, 2004.

[8] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[9] Gabriel Antoniu, Luc Bougé, and Loïc Cudennec. CoRDAGe: towards transparent management of interactions between applications and ressources. In *International Workshop on Scalable Tools for High-End Computing (STHEC 2008), held in conjunction with the International Conference on Supercomputing (ICS 2008)*, pages 13–24, Kos, Greece, June 2008.

[10] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: an adaptive supportive platform for data-sharing on the grid. *Scalable Computing: Practice and Experience (SCPE)*, 6(3):45–55, November 2005.

[11] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in P2P experiments using the JXTA Distributed Framework. In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par '04)*, volume 3149 of *Lecture Notes in Computer Science*, pages 1038–1047, Pisa, Italy, August 2004. Springer.

[12] Gabriel Antoniu, Loïc Cudennec, Mike Duigou, and Mathieu Jan. Performance scalability of the JXTA P2P framework. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS 2007)*, page 108, Long Beach, CA, USA, 2007.

[13] Gabriel Antoniu, Loïc Cudennec, Majd Ghareeb, and Osamu Tatebe. Building hierarchical Grid storage using the Gfarm global file system and the JuxMem Grid data-sharing service. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing (Euro-Par '08)*, volume 5168 of *Lecture Notes in Computer Science*, pages 456–465, Las Palmas de Gran Canaria, Spain, August 2008. Springer.

[14] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA based management of a computing utility. In *In Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, 2001.

[15] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, USA, June 2002.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, pages 164–177, New York, NY, USA, 2003. ACM.

[17] Francine Berman, Andrew Chien, Keith Cooper, Jack Dongarra, Ian Foster, Dennis Gannon, Lennart Johnsson, Ken Kennedy, Carl Kesselman, John Mellor-Crumme, Dan Reed, Linda Torczon, and Rich Wolski. The GrADS Project: software support for high-level Grid application development. *Int. J. High Perform. Comput. Appl.*, 15(4):327–344, 2001.

[18] D. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. Williams. The Earth System Grid: supporting the next generation of climate modeling research. *Proceedings of the IEEE*, 93(3):485–495, March 2005.

[19] J.-Y. Berthou and C. Caremoli. CALCIUM : un outil pour le couplage de codes : principes, pratique et perspectives. *Calculateurs parallèles*, 11(3):373–396, 1999. Hermès.

[20] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-based adaptive load balancing for MPI programs. In *Proceedings of the International Conference on Computational Science (ICCS 2001)*, volume 2074 of *Lecture Notes in Computer Science*, pages 108–117, San Francisco, CA, USA, May 2001.

[21] Hinde Bouziane. *De l'abstraction des modèles de composants logiciels pour la programmation d'applications scientifiques distribuées*. Thèse de doctorat, Université de Rennes 1, IRISA, IRISA/INRIA, Rennes, France, February 2008.

[22] J. M. Brooke, P. V. Coveney, J. Harting, S. Jha, S. M. Pickles, R. L. Pinning, and A. R. Porter. Computational steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting*, 2003.

[23] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in Tune. In *Proc. Ann. ACM Symp. on Applied Computing (SAC 2008)*, Fortaleza, Ceará, Brazil, March 2008. ACM.

[24] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *International Workshop on Component-Oriented Programming (WCOP '02)*, Malaga, Spain, June 2002.

[25] Jérémy Buisson. *Adaptation dynamique de programmes et composants parallèles*. Thèse de doctorat, INSA de Rennes, IRISA, IRISA/INRIA, September 2006.

[26] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Dynamic adaptation for Grid computing. In *Advances in Grid Computing - European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 538–547, Amsterdam, The Netherlands, June 2005. Springer.

[27] Jérémy Buisson, Ozan Sonmez, Hashim Mohamed, Wouter Lammers, and Dick Epema. Scheduling malleable applications in multicluster systems. In *IEEE International Cluster Conference*, Austin, TX, USA, September 2007.

[28] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: an architecture of a resource management and scheduling system in a global computational Grid. *Computing Research Repository (CoRR)*, 2000.

[29] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high-level components. In *Proc. 5th IEEE/ACM Intl. Symposium on Cluster Computing and the Grid (CCGrid '05)*, pages 776–783, Cardiff, UK, May 2005.

[30] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: a large scale, reconfigurable, controllable and monitorable grid platform. In *Proc. 6th IEEE/ACM Intl. Workshop on Grid Computing (Grid '05)*, pages 99–106, Seattle, WA, USA, November 2005.

[31] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Hérault, Frédéric Magniette, Vincent Néri, and Oleg Lodygensky. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. *Future Generation Comp. Syst.*, 21(3):417–437, 2005.

[32] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDIET: a deployment tool for distributed middleware on Grid'5000. In *Experimental Grid Testbeds for the Assessment of Large-Scale Distributed Applications and Tools (EXPGRID workshop). Held in conjunction with HPDC 15.*, pages 1–8, Paris, France, June 2006.

[33] Eddy Caron and Frédéric Desprez. DIET: a scalable toolbox to build network enabled servers on the Grid. *Intl. J. High-Performance Computing Applications*, 20(3):335–352, 2006.

[34] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Laurent Philippe, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par '02)*, pages 907–910, London, UK, 2002. Springer.

[35] Henri Casanova, Graziano Obertelli, Francine Berman, and Richard Wolski. The AppLeS parameter sweep template: user-level middleware for the Grids. *Sci. Program.*, 8(3):111–126, 2000.

[36] Jorge Cham. PHD Comics: tales from the road - CERN, part 4. http://www.phdcomics.com/comics/archive.php?comicid=1069, September 2008.

[37] Luca Clementi, Michael Rambadt, Roger Menday, and Johannes Reetz. UNICORE deployment within the DEISA supercomputing Grid infrastructure. In *Proceedings of the International Euro-Par Conference on Parallel Processing Workshops (Euro-Par '06)*, volume 4375 of *Lecture Notes in Computer Science*, pages 264–273. Springer, 2006.

[38] Toni Cortes, Carsten Franke, Yvon Jégou, Thilo Kielmann, Domenico Laforenza, Brian Matthews, Christine Morin, Luis Pablo Prieto, and Alexander Reinefeld. XtreemOS: a vision for a grid operating system. White Paper Available at http://www.xtreemos.eu/publications/research-papers/xtreemos-cacm.pdf, May 2008.

[39] J-P. Courtiat, P. Dembinski, R. Groz, and C. Jard. Estelle, un langage ISO pour les algorithmes distribués et les protocoles. *Technique et Science Informatique*, 6(2):311–324, 1987.

[40] Loïc Cudennec. Un service hiérarchique distribué de partage de données pour grille. In *Rencontres francophones du Parallélisme (RenPar '18)*, Fribourg, Suisse, February 2008.

[41] Karl Czajkowski, Carl Kesselman, Steven Fitzgerald, and Ian Foster. Grid information services for distributed resource sharing. *ACM International Symposium on High Performance Distributed Computing (HPDC 2001)*, page 181, 2001.

[42] Michel Daydé, Luc Giraud, Montse Hernandez, Jean-Yves L'Excellent, Chiara Puglisi, and Marc Pantel. An overview of the GRID-TLSE project. In *Poster Session of 6th International Conference High Performance Computing for Computational Science (VECPAR '04)*, volume 3402 of *Lecture Notes in Computer Science*, Valencia, Spain, June 2004. Springer.

[43] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Scott Koranda, Albert Lazzarini, Gaurang Mehta, Maria Alessandra Papa, and Karan Vahi. Pegasus and the pulsar search: from metadata to execution on the Grid. In *Parallel Processing and Applied Mathematics, 5th International Conference (PPAM 2003)*, volume 3019 of *Lecture Notes in Computer Science*, pages 821–830, Czestochowa, Poland, September 2003. Springer.

[44] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[45] Thomas A. DeFanti, Ian Foster, Michael E. Papka, Rick Stevens, and Tim Kuhfuss. Overview of the I-WAY: wide-area visual supercomputing. *Int. J. Supercomput. Appl. High Perfo. Comput.*, 10(2/3):123–131, Summer/Fall 1996.

[46] Alexandre Denis, Christian Pérez, Thierry Priol, and André Ribes. Padico: a component-based software infrastructure for grid computing. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003. IEEE Computer Society.

[47] Benjamin Depardon. Déploiement générique d'applications sur plates-formes hétérogènes distribuées. Research Report RR-6434, INRIA, 2008.

[48] Niels Drost, Elth Ogston, Rob V. van Nieuwpoort, and Henri E. Bal. ARRG: real-world gossiping. In *Proceedings of The 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC 2007)*, Monterey, CA, USA, June 2007.

[49] Niels Drost, Rob van Nieuwpoort, and Henri E. Bal. Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proceedings of the 6th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '06)*, May 2006.

[50] Brian Ensink and Vikram Adve. Coordinating adaptations in distributed systems. *Int'l Conference on Distributed Computing Systems (ICDCS 2004)*, pages 446–455, 2004.

[51] Brian Ensink, Joel Stanley, and Vikram S. Adve. Program control language: a programming language for adaptive distributed applications. *J. Parallel Distrib. Comput.*, 63(11):1082–1104, 2003.

[52] Enterprise Management Associates. Practical automatic computing: roadmap to self managing technology. A White Paper Prepared for IBM, January 2006.

[53] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the Grid with DeployWare. In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGrid '08)*, pages 177–184, Lyon, France, may 2008. IEEE.

[54] Message Passing Interface Forum. Message passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, May 1994.

[55] I. Foster, J. Geisler, B. Nickless, W. Smith, and S. Tuecke. Software infrastructure for the I-WAY high-performance distributed computing experiment. *ACM International Symposium on High Performance Distributed Computing (HPDC '96)*, page 562, 1996.

[56] I. Foster and C. Kesselman. Globus: a metacomputing infrastructure toolkit. *Intl. J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[57] Ian Foster. What is the Grid? - a three point checklist. *GRID Today*, 1(6), July 2002.

[58] Ian Foster and Carl Kesselman, editors. *The Grid: blueprint for a new computing infrastructure.* Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[59] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the Grid: enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.

[60] Ian T. Foster. Globus toolkit version 4: software for service-oriented systems. In *Network and Parallel Computing*, volume 3779 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2005.

[61] W. Gentzsch. Sun Grid Engine: towards creating a compute power Grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGrid '01)*, page 35, Brisbane, Australia, May 2001. IEEE Computer Society. Sun Microsystems.

[62] Yiannis Georgiou, Julien Leduc, Brice Videau, Johann Peyrard, and Olivier Richard. A tool for environment deployment in Clusters and light Grids. In *Second Workshop on System Management Tools for Large-Scale Parallel Systems (SMTPS '06)*, Rhodes Island, Greece, April 2006.

[63] Globus Alliance. GT 4.0 component fact sheet: Web Service Grid Resource Allocation and Management (WS GRAM). http://www.globus.org/toolkit/docs/4.0/execution/wsgram/WS_GRAM_Approach.html.

[64] Globus Alliance. GT 4.0 Reliable File Transfer (RFT) service. http://www.globus.org/toolkit/docs/4.0/data/rft/.

[65] P. Goldsack, J. Guijarro, A. Lain, G. Mecheneau, P. Murray, and P. Toft. SmartFrog: configuration and automatic ignition of distributed applications. Technical report, HP, Genève, Suisse, July 2003. In HP OpenView University Association conference (HP OVUA).

[66] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. SAGA: a simple API for Grid applications - high-level application programming on the Grid. *Computational Methods in Science and Technology*, SC05:8(2), November 2005. special issue "Grid Applications: New Challenges for Computational Methods".

[67] Andrew S. Grimshaw and Wm. A. Wulf. The Legion vision of a worldwide virtual computer. *Commun. ACM*, 40(1):39–45, 1997.

[68] Robert L. Henderson. Job scheduling under the Portable Batch System. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '95)*, pages 279–294, London, UK, 1995. Springer.

[69] Fabien Hermenier, Xavier Lorca, Hadrien Cambazard, Jean-Marc Menaud, and Narendra Jussien. Reconfiguration automatique du placement dans les grilles de

calculs dirigée par des objectifs. In *Actes 6e Conférence Francophone sur les Systèmes d'Exploitation (CFSE 6)*, Fribourg, CH, February 2008.

[70] Fabien Hermenier, Nicolas Loriant, and Jean-Marc Menaud. Power management in Grid computing with Xen. In *Frontiers of High Performance Computing and Networking - ISPA 2006 International Workshops*, volume 4331 of *Lecture Notes in Computer Science*, pages 407–416, Sorrento, Italy, December 2006. Springer.

[71] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[72] Paul Horn. *Autonomic computing: IBM's perspective on the state of information technology*. IBM, 2001.

[73] 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/INRIA, Rennes, France, November 2006.

[74] Emmanuel Jeanvoine. *Intergiciel pour l´exécution efficace et fiable d´applications distribuées dans des grilles dynamiques de très grande taille*. Thèse de doctorat, Université de Rennes 1, IRISA/EDF R&D, Rennes, France, November 2007.

[75] Emmanuel Jeanvoine, Christine Morin, and Daniel Leprince. Vigne: executing easily and efficiently a wide range of distributed applications in Grids. In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par 2007)*, volume 4641 of *Lecture Notes in Computer Science*, pages 394–403, Rennes, France, August 2007. Springer.

[76] Emmanuel Jeanvoine, Louis Rilling, Christine Morin, and Daniel Leprince. Using overlay networks to build operating system services for large scale Grids. In *Proceedings of the 5th International Symposium on Parallel and Distributed Computing (ISPDC 2006)*, pages 191–198, Timisoara, Romania, July 2006.

[77] Laxmikant V. Kale, Sameer Kumar, Mani Potnuru, Jayant DeSouza, and Sindhura Bandhakavi. Faucets: efficient resource allocation on the computational Grid. *International Conference on Parallel Processing (ICPP 2004)*, pages 396–405, 2004.

[78] Nick Karonis, Brian Toonen, and Ian Foster. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[79] Ken Kennedy, Bradley Broom, Arun Chauhan, Rob Fowler, John Garvin, Charles Koelbel, Cheryl McCosh, and John Mellor-Crummey. Telescoping languages: a system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(3):387–408, 2005.

[80] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[81] Sébastien Lacour. *Contribution à l'automatisation du déploiement d'applications sur des grilles de calcul*. Thèse de doctorat, Université de Rennes 1, IRISA/INRIA, Rennes, France, December 2005.

[82] Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic application description model: toward automatic deployment of applications on computational Grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid 2005)*, Seattle, WA, USA, November 2005. Springer.

[83] Gregor Von Laszewski, Jarek Gawor, Peter Lane, Nell Rehn, and Mike Russell. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience, ACM 2000 Java Grande Conference*, 13(8-9):645–662, June 2001.

[84] Bogdan Lesyng, Piotr Bala, and Dietmar Erwin. EUROGRID: European computational grid testbed. *J. Parallel Distrib. Comput.*, 63(5):590–596, 2003.

[85] Kai Li. IVY: a shared virtual memory system for parallel computing. In *Proc. 1988 Intl. Conf. on Parallel Processing*, pages 94–101, University Park, PA, USA, August 1988.

[86] Bin Lin and Peter A. Dinda. VSched: mixing batch and interactive virtual machines using periodic real-time scheduling. *SC: Conference on High Performance Networking and Computing*, page 8, 2005.

[87] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor: a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)*, June 1988.

[88] Yves Maheo, Frédéric Guidec, and Luc Courtrai. Middleware support for the deployment of resource-aware parallel Java components on heterogeneous distributed platforms. *Euromicro*, pages 144–151, 2004.

[89] Cyrille Martin, Olivier Richard, and Guillaume Huard. Déploiement adaptatif d'applications parallèles. *Technique et Science Informatiques (TSI)*, 24(5):547–565, 2005.

[90] Raphaël Marvie, Philippe Merle, Jean-Marc Geib, and Mathieu Vadet. OpenCCM : une plate-forme ouverte pour composants CORBA. In *Actes de la seconde Conférence Francaise sur les Systèmes d'Exploitation (CFSE 2)*, Paris, France, Avril 2001.

[91] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.

[92] Microsoft. DCOM: Distributed Component Object Model Technologies. http://www.microsoft.com/com/default.mspx.

[93] Hashim H. Mohamed and Dick H. J. Epema. The design and implementation of the KOALA co-allocating grid scheduler. In *European Grid Conference (EGC 2005)*, volume 3470 of *Lecture Notes in Computer Science*, pages 640–650, Amsterdam, The Netherlands, 2005. Springer.

[94] Cleve B. Moler. MATLAB: an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science, 1980.

[95] MONARC members. Models of networked analysis at regional centres for LHC experiments: phase 2 report. Technical Report CERN/LCB 2000-001, CERN, 2000.

[96] Sébastien Monnet. *Gestion des données dans les grilles de calcul : support pour la tolérance aux fautes et la cohérence des données*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, November 2006.

[97] Hidemoto Nakada, Satoshi Matsuoka, Yoshio Tanaka, and Satoshi Sekiguchi. The design and implementation of a fault-tolerant RPC system: Ninf-C. In *Proceedings of the High Performance Computing and Grid in Asia Pacific Region, Seventh International Conference (HPCASIA '04)*, pages 9–18, Washington, DC, USA, 2004. IEEE Computer Society.

[98] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *Future Gener. Comput. Syst.*, 15(5-6):649–658, 1999.

[99] Harvey B. Newman, I. C. Legrand, Philippe Galvez, R. Voicu, and C. Cirstoiu. MonALISA : a distributed monitoring service architecture. *Computing Research Repository (CoRR)*, 2003.

[100] Ralph Niederberger and Olaf Mextorf. The DEISA project: network operation and support - first experiences. In *The world of pervasive networking, the 21th Trans-European Research and Education Networking Conference*, Poznan, Poland, June 2005. TERENA.

[101] OMG. CORBA component model specification, v4.0. http://www.omg.org/technology/documents/formal/components.htm.

[102] Abdulla Othman, Peter M. Dew, Karim Djemame, and Iain Gourlay. Adaptive Grid resource brokering. In *2003 IEEE International Conference on Cluster Computing (Cluster 2003)*, pages 172–179, Kowloon, Hong Kong, China, December 2003. IEEE Computer Society.

[103] Noel De Palma, Sylvain Sicard, Sara Bouchenak, Daniel Hagimont, and Fabienne Boyer. Autonomic administration of clustered J2EE applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2005)*, volume 3, pages 1248–1254, Las Vegas, NV, USA, June 2005. CSREA Press.

[104] Rob Pennington. Terascale clusters and the TeraGrid. In *Proceedings of 6th International Conference/Exhibition on High Performance Computing in Asia Pacific Region*, pages 407–413, Bangalore, India, December 2002. Invited talk.

[105] Louis Rilling. Vigne: towards a self-healing Grid operating system. In *Proceedings of the International Euro-Par Conference on Parallel Processing (Euro-Par 2006)*, volume 4128 of *Lecture Notes in Computer Science*, pages 437–447, Dresden, Germany, August 2006. Springer.

[106] Thomas Ropars. Supervision d'applications sur grille de calcul. Rapport de master de recherche, Université de Rennes 1, June 2006.

[107] Thomas Ropars, Emmanuel Jeanvoine, and Christine Morin. GAMoSe: an accurate monitoring service for Grid applications. In *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, pages 295–302, Hagenberg, Austria, July 2007.

[108] P. Ruth, Junghwan Rhee, Dongyan Xu, R. Kennell, and S. Goasguen. Autonomic live adaptation of virtual computational environments in a multi-domain infrastructure. *The International Conference on Autonomic Computing and Communications (ICAC 2006)*, pages 5–14, 2006.

[109] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Summer Technical Conference*, pages 119–130, Portland, OR, USA, June 1985.

[110] Robert Scheifler and Jim Gettys. The X Window system. *Software, Practice and Experience (SPE)*, 20(S2):S2/5–S2/34, 1990.

[111] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: a remote procedure call API for Grid computing. In *Proceedings of the Third International Workshop on Grid Computing (GRID '02)*, pages 274–278, London, UK, 2002. Springer.

[112] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet Indirection Infrastructure. In *Networking, IEEE/ACM Transactions*, volume 12, pages 205–218, Piscataway, NJ, USA, 2004. IEEE Press.

[113] Vaidy Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience (CPE)*, 2(4):315–340, 1990.

[114] Martin Swany and Rich Wolski. Representing dynamic performance information in Grid environments with the Network Weather Service. *Proc. 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (CCGrid '02)*, page 48, 2002.

[115] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G: a reference implementation of RPC-based programming middleware for Grid computing. *J. Grid Computing*, 1(1):41–51, 2003.

[116] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid datafarm architecture for Petascale data intensive computing. In *Proc. 2nd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid (Cluster 2002)*, page 102, Washington, DC, USA, 2002. IEEE Computer Society.

[117] Osamu Tatebe, Noriyuki Soda, Youhei Morita, Satoshi Matsuoka, and Satoshi Sekiguchi. Gfarm v2: a grid file system that supports high-performance distributed and parallel data computing. In *Proceedings of the 2004 Conference on Computing in High Energy and Nuclear Physics (CHEP '04)*, Interlaken, Switzerland, September 2004. Science Press.

[118] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[119] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly a nd Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 super-peer virtual network. http://www.jxta.org/project/www/docs/JXTA2.0protocols1.pdf, May 2003.

[120] Peter Troger, Hrabri Rajic, Andreas Haas, and Piotr Domagalski. Standardization of an API for distributed resource management systems. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*, pages 619–626, Rio de Janeiro, Brazil, May 2007. IEEE Computer Society.

[121] UserLand Software. XML-RPC specification. http://www.xmlrpc.com/spec/.

[122] Sathish Vadhiyar and Jack Dongarra. Self adaptability in Grid computing. *Concurrency and Computation: Practice and Experience (CCPE). Special Issue: Grid Performance*, 17(2–4):235–257, 2005.

[123] S. Valcke, D. Declat, R. Redler, H. Ritzdorf, T. Schoenemeyer, and R. Vogelsang. The PRISM coupling and I/O system. In *Proceedings of the 6th International Meeting, Vol. 1: High performance computing for computational science (VECPAR '04)*, volume 1, Universidad Politecnica de Valencia, Valencia, Spain, 2004.

[124] Sophie Valcke, Arnaud Caubel, Reiner Vogelsang, and Damien Declat. OASIS 3 user's guide. Technical Report TR/CMGC/04/68, CERFACS, Toulouse, France, 2004.

[125] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. User-friendly and reliable Grid computing based on imperfect middleware. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '07)*, November 2007.

[126] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments – a survey. *Constraints*, 10(3), 2005.

[127] W3C. SOAP: Simple Object Access Protocol. http://www.w3.org/TR/soap/.

[128] Jim Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[129] David A. Wheeler. SLOCCount, a set of tools for counting physical source lines of code. http://www.dwheeler.com/sloccount/.

[130] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[131] A. YarKhan, J. Dongarra, and K. Seymour. GridSolve: the evolution of network enabled solver. In *Grid-based problem solving environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments*, pages 215–226, Prescott, AZ, USA, 2006. Innovative Computing Laboratory, University of Tennessee.

[132] ADAGE. http://adage.gforge.inria.fr/.

[133] Chimera DHT. http://current.cs.ucsb.edu/projects/chimera/.

[134] Cross-Platform Make. http://www.cmake.org/.

[135] The CoRDAGe project: co-deployment and re-deployment of generic Grid applications. http://cordage.gforge.inria.fr/.

[136] eDonkey. http://www.edonkey2000.com/.

[137] EGEE: Enabling Grid for E-sciencE. http://www.eu-egee.org/.

[138] Elagi: the easy access to Grid middleware services library. University of California, San Diego (UCSD), Grail Project. Available at http://grail.sdsc.edu/projects/elagi/.

[139] The Grid'5000 project. http://www.grid5000.org/.

[140] GRID-TLSE: tests for large systems of equations. http://www.gridtlse.org/.

[141] GridWay. http://www.gridway.org/.

[142] GRUDU: Grid'5000 reservation utility for deployment usage. http://grudu.gforge.inria.fr/.

[143] Le projet HydroGrid. http://www-rocq.inria.fr/~kern/HydroGrid/HydroGrid.html.

[144] JFtp - the Java network browser. http://j-ftp.sourceforge.net/.

[145] Kadeploy. http://www-id.imag.fr/Logiciels/kadeploy/.

[146] Katapult. http://www-id.imag.fr/~nussbaum/katapult.php.

[147] Kazaa. http://www.kazaa.com/.

[148] Khashmir DHT. http://khashmir.sourceforge.net/.

[149] LEGO: League for efficient grid operation. http://graal.ens-lyon.fr/LEGO/.

[150] Worldwide LHC Computing Grid. http://lcg.web.cern.ch/LCG/.

[151] LoadLeveler. http://www-03.ibm.com/systems/clusters/software/loadleveler/.

[152] MySQL. http://www.mysql.com/.

[153] NorGrid - Norwegian GRID initiative. http://www.norgrid.no/about/.

[154] OpenCCM - the Open CORBA Component Model platform. http://openccm.objectweb.org/.

[155] Open Grid Forum. http://www.ogf.org/.

[156] The RAMSES project. http://www-dapnia.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904.

[157] Mag'Ren : Les dossiers techniques du CIP RENATER. http://www.renater.fr/IMG/pdf/Mag_ren_5-3.pdf, February 2007.

[158] RENATER: Rapport d'activité 2007. http://www.renater.fr/IMG/pdf/RAPPORT2007.pdf, 2007.

[159] SAGA: Simple API for Grid Applications. http://wiki.cct.lsu.edu/saga/space/start/.

[160] SweGrid. http://www.swegrid.se/about/.

[161] XMLGoDIETGenerator. http://graal.ens-lyon.fr/~diet/xmlgodietgenerator.html.

[162] XML-RPC C: XML-RPC for C and C++. http://xmlrpc-c.sourceforge.net/.

[163] XML-RPC vs. other protocols. http://tldp.org/HOWTO/XML-RPC-HOWTO/xmlrpc-howto-competition.html.

Loïc CUDENNEC

CORDAGE : un service générique de co-déploiement et redéploiement d'applications sur grilles.

Mots-clés : grille de calcul, application distribuée, service de co-déploiement, redéploiement, CORDAGE, dynamicité, autonomie.

La mutualisation des ressources physiques réparties dans les universités, les instituts et les entreprises a permis l'émergence des grilles de calcul. Ces infrastructures dynamiques sont bien adaptées aux applications scientifiques ayant de grands besoins en puissance de calcul et en espace de stockage. L'un des défis majeur pour les grilles de calcul reste la simplification de leur utilisation. Contrairement au déploiement d'applications sur une infrastructure centralisée, le déploiement sur une grille nécessite de nombreuses tâches pénibles pour l'utilisateur. La sélection des ressources, le transfert des programmes ainsi que la surveillance de l'exécution sont en effet laissés à sa charge. Aujourd'hui, de nombreux travaux proposent d'automatiser ces étapes dans des cas simples. En revanche très peu permettent de prendre en charge des déploiements plus complexes, comme par exemple le redéploiement d'une partie de l'application pendant son exécution ou encore le déploiement coordonné de plusieurs applications.

Dans cette thèse, nous proposons un modèle pour prendre en charge le déploiement dynamique des applications sur les grilles de calcul. Ce modèle vise à offrir deux fonctionnalités principales. La première consiste en la traduction d'actions de haut niveau, spécifiques aux applications, en opérations de bas niveau, relatives à la gestion des ressources sur la grille. La deuxième consiste en la pré-planification des déploiements, redéploiements et co-déploiements d'applications sur les ressources physiques.

Le modèle satisfait trois propriétés. Il rend transparent la gestion des ressources à l'utilisateur. Il offre des actions spécifiques aux besoins de l'application. Enfin, il est non-intrusif en limitant les contraintes sur le modèle de programmation de l'application.

Une proposition d'architecture nommée CORDAGE vient illustrer ce modèle pour le co-déploiement et le redéploiement d'applications. CORDAGE a été développé en lien avec l'outil de réservation OAR et l'outil de déploiement ADAGE. La validation du prototype s'est effectuée avec la plate-forme pair-à-pair JXTA, le service de partage de données JUXMEM ainsi que le système de fichiers distribué GFARM. Notre approche a été évaluée sur la grille expérimentale GRID'5000.