Architecture d'un service de partage de données modifiables sur une infrastructure pair-à-pair

Mathieu Jan

Architecture d'un service de partage de données modifiables sur une infrastructure pair-à-pair

Mathieu Jan
Mathieu.Jan@irisa.fr

Superviseurs : **Gabriel Antoniu, Luc Bougé, Thierry Priol**
{Gabriel.Antoniu,Luc.Bouge,Thierry.Priol}@irisa.fr

IRISA, Projet Paris
Février — juin 2003

Rapport de stage de DEA d'Informatique de l'IFSIC, université de Rennes I.

Table des matières

Chapitre 1

Introduction

1.1 Le contexte : calcul sur grille

Le fort développement d'Internet durant les dernières années a mené à l'apparition de nombreuses initiatives ayant pour objectif l'exploitation des ressources de calcul et de stockage disponibles sur ce gigantesque réseau. Ces initiatives ont souvent en commun l'idée que les traitements de calcul intensif réalisés il y a 20 ans sur des super-calculateurs spécialisés, et plus récemment sur des grappes à haute performance, se feront de plus en plus de manière transparente sur des ressources distribuées à une grande échelle, typiquement à l'échelle d'un pays. Dans ce contexte, un effort de recherche particulièrement importante a été mené autour du *calcul sur grille* (*Grid computing*) [22].

Le terme *grille* a été choisi vers le milieu des années 1990 pour désigner une proposition d'infrastructure pour le calcul distribué à très grande échelle, typiquement des simulations numériques de plusieurs heures ou jours, en utilisant les ressources lourdes de plusieurs centres de calcul distribués d'un pays. Les ressources utilisées sont la puissance de calcul, mais aussi la capacité de stockage (disques, tours, robots, etc.), d'acquisition de données (accélérateurs de particules, microscopes électroniques, satellites, sismographes, etc.) et surtout d'exploitation des données (visualisation immersive en réalité virtuelle, etc.). Un grand nombre des projets de recherche visant à développer ce type d'infrastructure de calcul se sont développés, notamment Condor [52], Legion [63] et Globus [54].

L'un des principaux apports des environnements de calcul sur grille développés jusqu'à présent est d'avoir : d'une part *découplé les calculs ; et de l'autre les traitements de déploiement nécessaires pour ces calculs*. Dans cette approche effet, le déploiement est vu comme un service proposé par l'infrastructure. Le programmeur n'a pas à le programmer lui-même dans les applications, c'est l'infrastructure qui se charge de localiser et d'interagir avec les ressources physiques et d'y ordonnancer les traitements. En revanche, on peut constater qu'aujourd'hui il n'existe pas de service de ce type pour la gestion des données sur la grille. Paradoxalement, on dispose d'infrastructures complexes permettant d'ordonnancer de manière transparente des calculs répartis sur plusieurs sites, alors que le stockage et le transfert des données nécessaires aux applications est laissé à la charge de l'utilisateur. Or la complexité d'une gestion explicite de grandes masses de données par les applications devient un facteur limitant majeur dans l'exploitation efficace des grilles de calcul.

1.2 Le besoin : gestion de données

Les utilisateurs de grilles de calculs sont des physiciens, chimistes, biologistes, etc. souhaitant simuler des systèmes de plus en plus réalistes. Par exemple, les météorologistes ont besoin d'une puissance de calcul et de stockage de plus en plus grande afin de prendre en compte le maximum de paramètres pour le calcul de prévisions météo plusieurs jours à l'avance. Plusieurs phénomènes physiques sont donc simulés sur plusieurs sites et en parallèle en raison des capacités de calculs nécessaires. Le volume de données à partager et donc à échanger entre les différents sites pour mener à bien de telles simulations, peut représenter plusieurs dizaines de gigaoctets voire quelques dizaines de téraoctets. C'est ce type d'applications qui motive la conception d'un service de partage de données modifiables à l'échelle de la grille permettant une gestion efficace des données entre les différents sites de calculs d'une grille.

Plusieurs environnements de calculs numériques sur grille existent : les projets NetSolve [66], Ninf [67], MetaNEOS [64] et DIET [53]. L'objectif de la réalisation d'un service de partage de données modifiables pour la grille est de l'intégrer à un environnement de calcul numérique, par exemple DIET, permettant ainsi de *découpler l'application de calcul de la gestion des données associées*. Ce service vise donc à fournir la *persistance* et la *transparence de la localisation* des données au niveau applicatif. Toutefois la réalisation de ces deux objectifs doit être effectuée dans un environnement soumis à des contraintes d'*extensibilité* en nombre de nœuds présents dans le système afin de viser l'échelle de la grille, typiquement plusieurs milliers ou dizaines de milliers de nœuds, de *cohérence* des données qui peuvent être modifiées de manière simultanées par plusieurs sites et de *volatilité* des ressources disponibles. À la première vue, la volatilité semble une contrainte non nécessaire pour l'architecture visée, puisque les grappes de machines interconnectées constituant une grille sont relativement stables. Toutefois, l'hypothèse d'un arrêt brutal et d'un redémarrage différé de nœuds appartenant à une grappe suite à un problème technique est tout à fait réaliste. La volatilité des nœuds constituant une grappe est donc une contrainte à prendre en compte dans la conception d'un tel service.

1.3 Notre contribution : service de partage de données

L'objectif de ce rapport est de définir un service de partage de données modifiables pour la grille. Ce service prend en compte les propriétés de persistance et de transparence de la localisation des données et les contraintes d'extensibilité, de cohérence des données et de volatilité. La contribution résultante est la définition de l'architecture d'un service de partage de données modifiables sur une infrastructure pair-à-pair, illustrée par une plate-forme logicielle appelée JuxMem, pour *Juxtaposed Memory*. Cette architecture est hiérarchique afin d'être extensible et de tirer parti des caractéristiques physiques de l'architecture sous-jacente. Elle est dite basée sur une infrastructure pair-à-pair puisque chaque nœud peut à la fois fournir et utiliser le service. La transparence de la localisation des données au niveau applicatif est assurée et cette volatilité des nœuds est prise en compte pour assurer la persistance des données. Enfin, ces mêmes données sont modifiables par différents clients.

Après un bref état de l'art montrant l'absence de systèmes répondant à ces besoins (chapitre 2), le service proposé est détaillé (chapitre 3). L'implémentation de JuxMem au-dessus de la plate-forme JXTA[59] est ensuite présentée (chapitre 4). JXTA est un environnement

générique qui permet de bâtir des applications utilisant le modèle pair-à-pair. Enfin, l'implémentation de JuxMem au-dessus de JXTA est évaluée en terme de consommation mémoire et de résistance à la volatilité des pairs hébergeant des données (chapitre 5). La conclusion (chapitre 6) présente les contributions apportées par JuxMem, les différents problèmes ouverts ainsi que les perspectives pour la mise en œuvre d'un service partage de données modifiables sur une infrastructure pair-à-pair.

Chapitre 2

Positionnement

2.1 Gestion de données sur la grille

Le service de gestion de données sur la grille que l'on se propose d'élaborer vise deux objectifs.

Persistance. Il s'agit de pouvoir stocker des données sur la grille, afin de permettre le partage et le transfert efficaces de ces données, ainsi qu'un meilleur ordonnancement des calculs, compte tenu de la localisation des données.

Transparence. Il s'agit de décharger les applications de la gestion explicite de la localisation et du transfert des données entre les sites en ayant besoin.

Toutefois, la réalisation de ces deux objectifs est soumise à trois contraintes que l'on se fixe et qui sont les suivantes.

Extensibilité. Alors que l'algorithmique du calcul parallèle a été bien étudiée pour des configurations avec un nombre relativement réduit de machines, l'architecture visée dans notre cas est de l'ordre de plusieurs milliers ou dizaines de milliers de nœuds fédérés en grappes. Les algorithmes doivent donc intégrer dès le départ cette hypothèse.

Cohérence. Dans les applications de calcul visées, les données sont généralement partagées et peuvent donc être modifiées par plusieurs sites. Un problème de cohérence des données à grande échelle se pose donc.

Volatilité. La disponibilité des ressources sur l'architecture visée n'est pas garantie. Une grille de calcul est en effet une fédération de grappes de machines gérée par différentes entités et mises à la disposition de la communauté. Les nœuds constituant une grappe peuvent donc être arrêtés puis redémarrés suite à des problèmes techniques ou tout simplement à cause de l'indisponibilité temporaire de la ressource, par exemple dans le cas d'une mise à disposition des nœuds seulement lorsqu'ils sont inactifs.

Le service proposé se rapproche de plusieurs types de systèmes de gestion de données existants mais qui ne prennent en compte que partiellement les objectifs et les contraintes ci-dessus. La présentation de ces différents systèmes qui suit, vise surtout à montrer sur quels points ils ne répondent pas à ces objectifs et contraintes.

2.1.1 Gestion des données modifiables sans transparence

Globus. Actuellement, l'approche la plus utilisée pour la gestion des données nécessaires aux calculs répartis sur différentes machines d'une grille repose sur des *transferts explicites* des données entre les clients et les serveurs de calcul. La plate-forme Globus [21] est l'environnement le plus utilisé pour la gestion de ressources disponibles sur une grille. Cette plate-forme fournit en outre, pour ce qui nous intéresse, des mécanismes d'accès aux données, *Globus Access to Secondary Storage* [7] (GASS), basés sur le protocole *Grid File Transfer Protocol* [2] (GridFTP), un protocole de type *File Tansfer Protocol* (FTP) enrichi de quelques fonctionnalités telles que : gestion de l'authentification, gestion de l'intégrité des données, transferts parallèles, gestion des reprises en cas d'échec, etc. Bien que plus évolués que le protocole FTP de base, ces mécanismes exigent toujours une gestion explicite de la localisation des fichiers par le programmeur. Globus intègre également des catalogues permettant l'enregistrement de plusieurs copies d'une donnée (*Globus Replica Catalog* et *Globus Replica Management* [42]). Toutefois, la gestion de ces catalogues reste *manuelle*, puisque l'enregistrement des copies dans les différents catalogues présents est à la charge de l'utilisateur. Ce dernier doit garantir lui-même la cohérence des différentes copies enregistrées, aucune garantie n'étant fournie en ce sens par Globus.

MPI. Une solution alternative pour la gestion des communications sur la grille consiste à utiliser la bibliothèque MPI-2, qui fournit des mécanismes performants pour gérer des communications collectives et des synchronisations par groupes de machines. MPI-2 a déjà montré son efficacité dans des expériences de déploiement sur plusieurs milliers de nœuds. Le principal point faible reste néanmoins la complexité de la programmation, car la gestion des communications reste à la charge de l'utilisateur. L'extension MPI-IO [16] de MPI permet de gérer des fichiers à l'échelle d'une grille, la modification d'un fichier pouvant être modélisée par l'envoi d'un message. Le transfert de fichiers de données entre différents sites de calculs est de ce fait possible et cela en s'affranchissant de l'hétérogénéité des différentes interfaces des systèmes de fichiers disponibles sur les nœuds des différentes grappes d'une grille. Évidemment, la gestion des communications reste toujours à la charge de l'utilisateur ; elle est d'une complexité croissante avec le nombre de nœuds impliqués.

2.1.2 Stockage de données modifiables sans transparence

Le projet *Internet Backplane Protocol* [6, 37] (IBP) de l'Université du Tennessee propose un système de stockage pour des applications réparties à grande échelle qui fournit des mécanismes de gestion d'un ensemble de tampons présents sur Internet. L'utilisateur a la possibilité de *louer* ces espaces de stockage et de les utiliser pour des transferts de données. Il faut voir ces espaces de stockage comme des dépôts où l'on peut accumuler des données. IBP a été utilisé par l'environnement de calcul numérique NetSolve pour implémenter un service de gestion de données persistantes. La gestion des transferts reste toutefois encore à la charge de l'utilisateur. De plus, IBP ne fournit pas de mécanismes permettant de prendre en compte dynamiquement l'évolution de la configuration comme l'ajout et le retrait de ressources de stockage. Enfin, aucun mécanisme de gestion de la cohérence n'est disponible. En effet, IBP ne stocke qu'une suite d'octets dans ses tampons sans tenir compte des relations existant entre des copies d'une même donnée situées dans deux tampons présents sur Internet.

2.1.3 Partage transparent de données à petite échelle

Mémoires virtuellement partagéees. Le partage transparent de données à travers l'abstraction d'un espace d'adressage unique accessible par des machines physiquement distribuées a fait l'objet de nombreux efforts de recherche dans le domaine des systèmes à *mémoire virtuellement partagée* (MVP). Utilisés essentiellement pour exécuter des applications de calcul parallèle sur des grappes de machines, les MVP présentent l'avantage d'offrir la transparence des accès aux données : tous les nœuds y accèdent de manière uniforme à partir d'un identifiant ou d'une adresse virtuelle et c'est la MVP qui se charge de la localisation, du transfert, de la réplication éventuelle ainsi que de la gestion de la cohérence des copies.

Dans ce cadre, un nombre important de modèles et de protocoles de cohérence ont été définis pour permettre la gestion efficace des données répliquées, comme le modèle de la *cohérence séquentielle* [30] et le modèle de la *cohérence à la libération* [25]. Dans le premier modèle, tous les processeurs « voient » toutes les lectures et toutes les écritures de tous les nœuds dans le même ordre. Le second modèle assure que la mémoire est cohérente entre deux points de synchronisation (acquisition et libération de verrous protégeant les sections critiques).

Un protocole de cohérence est une implémentation particulière d'un modèle de cohérence. D'autre part des protocoles efficaces ont été conçus pour permettre de multiples modifications concurrentes des données par des nœuds différents. Les systèmes *TreadMarks* [3] et *Munin* [12] autorisent ces écrivains multiples sur des parties différentes d'un objet par l'utilisation d'un mécanisme de *twinning-diffing*. Lors de la première écriture, une copie jumelle (*twin*) de l'objet est créée ; au moment de la libération du verrou, la différence mot à mot (*diff*) entre la copie jumelle et l'objet modifié est calculée, puis envoyée à l'ensemble des nœuds qui disposent d'un copie pour la mise à jour de l'objet. Une version améliorée de ce protocole consiste à envoyer les modifications à un nœud hôte de la page, qui sert de copie de référence pour les autres nœuds, puis invalider les copies répliquées. Ce protocole se nomme le protocole de cohérence à la libération avec nœud hôte [51, 26]. Tout autre nœud qui a besoin d'accéder à une page partagée la rapatrie dans son intégralité depuis le nœud hôte de cette page.

Les MVP ont montré une efficacité satisfaisante uniquement lorsque le nombre de machines est faible, jusqu'à quelques dizaines. Ceci représente une limitation majeure qui ne permet pas l'utilisation pratique de ces systèmes à plus large échelle.

JavaSpace. La technologie JavaSpace [57], basée sur le langage Linda [11] de David Gelertner, propose un espace de stockage permettant le partage d'objets Java. La persistance des objets partagés est assurée. Cette technologie est un service proposé au sein de l'architecture Jini [58] de Sun Microsystems. Jini est un système distribué permettant la communication de différents matériels au sein d'un réseau local afin d'offrir des services réseaux à des applications. Toutefois, cette architecture est limitée à des réseaux locaux et utilise un gestionnaire centralisé pour répertorier les différents services disponibles. Le service JavaSpace repose, pour les communications, sur la technologie RMI de Java très limitée pour la réalisation d'un service de partage à grande échelle et performant. En effet, elle est lourde, coûteuse et ne tolère pas les défaillances. Elle est également dépendante de Java pour la méthode de stockage des objets, qui repose sur la sérialisation. Aucun mécanisme de réplication des données n'est disponible et la gestion de la cohérence reste à la charge de l'utilisateur. En effet, JavaSpace ne propose qu'un simple stockage d'objets sans tenir compte des relations existant

entre différentes copies d'un même objet Java. Toutefois, la transparence de la localisation et du transfert des données est assurée, la récupération de données présentes dans l'espace de stockage se faisant par l'utilisation de filtres sur le type et les attributs des objets.

2.1.4 Stockage de données non-modifiables à grande échelle

En parallèle avec les développements institutionnels centrés sur le *grid computing*, un autre paradigme de calcul global a récemment focalisé l'intérêt de la communauté scientifique : le calcul *pair-à-pair* (*peer-to-peer computing* : P2P) [32, 45]. Ce modèle complémente le modèle classique *client-serveur* qui est aujourd'hui à la base de la plupart des traitements sur Internet, en *symétrisant* la relation des machines qui interagissent : chaque machine peut être client dans une transaction et serveur dans une autre [1]. Popularisé par Napster [65], Gnutella [35], et aujourd'hui KaZaA [62], ce paradigme a été centré dès le départ sur la *gestion de larges masses de données réparties à très grande échelle* et il a montré qu'il offre une véritable solution. À titre d'exemple, le réseau *KaZaA*, l'un des réseaux les plus récents de ce type, centré sur le partage de fichiers à grande échelle, regroupe en moyenne, à chaque instant, 4.500.000 machines, 900.000.000 fichiers contenant 9 péta-octets de données. Les machines constituant le réseau sont typiquement des PC connectés à Internet par intermittence et disposant d'une adresse IP temporaire.

Le principal problème dans ces systèmes est de trouver une donnée, selon certains critères comme par exemple le nom du fichier, et de l'acheminer vers le pair qui en fait la demande. C'est l'objectif des techniques de localisation qui sont donc le point central dans un système pair-à-pair de gestion de données à grande échelle. Il existe trois techniques de localisation.

Par répertoire centralisé. La localisation repose sur un un serveur central qui regroupe, dans un répertoire, l'ensemble des couples (pair, ressource) des ressources présentes sur l'ensemble des pairs formant le système : Napster en est le parfait exemple. Lorsqu'une donnée a été localisée dans le système par un pair, c'est-à-dire qu'il a récupéré du serveur central l'adresse d'un autre pair qui héberge cette donnée, le routage se fait alors directement entre ces deux pairs. Toutefois, la défaillance du serveur central suffit à entraîner l'indisponibilité du service rendu par l'application basée sur cette technique.

Par inondation. La localisation repose sur la diffusion de la requête (*broadcast*) à l'ensemble des pairs qui sont à l'intérieur d'un certain voisinage. Ce voisinage est l'ensemble des pairs du système qu'un pair donné connaît, et définit l'horizon que le pair peut atteindre. Gnutella utilise cette technique de localisation. Une variante utilise la notion de *super-pair*. Un super-pair est un pair qui agit comme un répertoire centralisé pour le compte d'un ensemble fini de pairs. La diffusion des requêtes par *broadcast* se fait alors entre ces super-pairs. KaZaA est l'un des systèmes qui se basent sur cette technique. Dans les deux approches, le routage vers le pair qui a émis la requête se fait alors, depuis un pair hébergeant une donnée recherchée, en suivant en sens inverse le chemin pris par la requête. Le point faible de cette technique est le nombre de messages importants émis pour une recherche, gaspillant inutilement la bande passante du système.

Par table de hachage distribuée. La localisation fonctionne indépendamment des réseaux physiques sous-jacents et constitue un réseau logique, nécessitant un mécanisme de

nommage, tant pour les données que pour les pairs. Une fonction de hachage (par exemple SHA1) est appliquée sur l'adresse IP du nœud afin de générer les identifiants des pairs, notés *nodeID*. Les identifiants des données, notés *fileID*, sont générés à partir de la donnée elle-même en utilisant cette même fonction. Un pair est dit *responsable* d'une donnée si son *nodeID* est le plus proche du *fileID* de la donnée. Pour localiser une donnée, il suffit alors de router les requêtes vers le pair qui est responsable de la donnée recherchée en utilisant une *table de hachage distribuée* (DHT, pour *Distributed Hash Table*). Les principaux systèmes utilisant cette technique sont CFS [17], PAST [41] et OceanStore [28] qui s'appuient sur respectivement la couche de localisation et de routage Chord [47], Pastry [40] et Tapestry [50]. Ces mécanismes ont fait récemment l'objet d'améliorations basées sur la notion de graphe de De Bruijn. Ils ont abouti à l'élaboration de systèmes comme Koorde [27] et D2B [23]. Cette technique de localisation et de routage permet de garantir de trouver une donnée dans le système de manière efficace.

Ces techniques de localisation et de routage dans les systèmes pair-à-pair ont fait l'objet de l'étude bibliographique de ce stage. Les avancées permises par ces systèmes garantissent des propriétés intéressantes pour le service que l'on propose dans ce rapport, comme le support d'un très grand nombre de nœuds très volatiles ou la garantie de retrouver une donnée de manière efficace.

2.1.5 Stockage de données modifiables à grande échelle

Les systèmes pair-à-pair, présentés à la section précédente, ont été étudiés essentiellement pour des applications de *partage de fichiers répliqués en lecture seule*. Toutefois, des systèmes comme OceanStore de l'Université de Berkeley et Ivy [34] du MIT, proposent des mécanismes de partage de données modifiables visant la grande échelle.

Dans Ivy, basé sur le service de localisation Chord, chaque utilisateur possède un fichier appelé *log* dans lequel il écrit ses modifications sur l'ensemble des fichiers du système. Pour avoir une vue cohérente du système de fichiers il doit parcourir tous les fichiers *log* des autres utilisateurs. Toutefois, l'utilisation de caches évite le parcours de la totalité de ces fichiers. Malgré tout, ce système ne peut supporter en pratique qu'un nombre très faible d'écrivains, ce qui constitue donc un frein à l'extensibilité du système et ne permet pas d'atteindre en pratique la grande échelle visée.

Concernant OceanStore, les mises à jour concurrentes utilisent un modèle basé sur la résolution des conflits. Les tentatives de modifications sont transmises aux serveurs qui décident de l'ordre à appliquer pour ces modifications. Parmi tous les serveurs qui hébergent des copies de la donnée, seul un petit nombre d'entre eux décident de cet ordre puis transmettent leur choix vers l'ensemble des serveurs possédant une copie. Le nombre de modifications simultanées est de ce fait faible afin de ne pas être coûteux en nombre de messages. Malgré tout, un nombre réduit de modifications simultanées suffit à diminuer les performances d'un tel système. Enfin, aucune garantie sur les mises à jour n'est assurée : les tentatives de modifications peuvent échouer.

En pratique, les résultats obtenus pour ces systèmes sont très loin d'atteindre la grande échelle visée. Les tests ont en effet été réalisés sur des systèmes constitués de seulement une dizaine de nœuds, ce qui ne peut pas être considéré comme de la grande échelle.

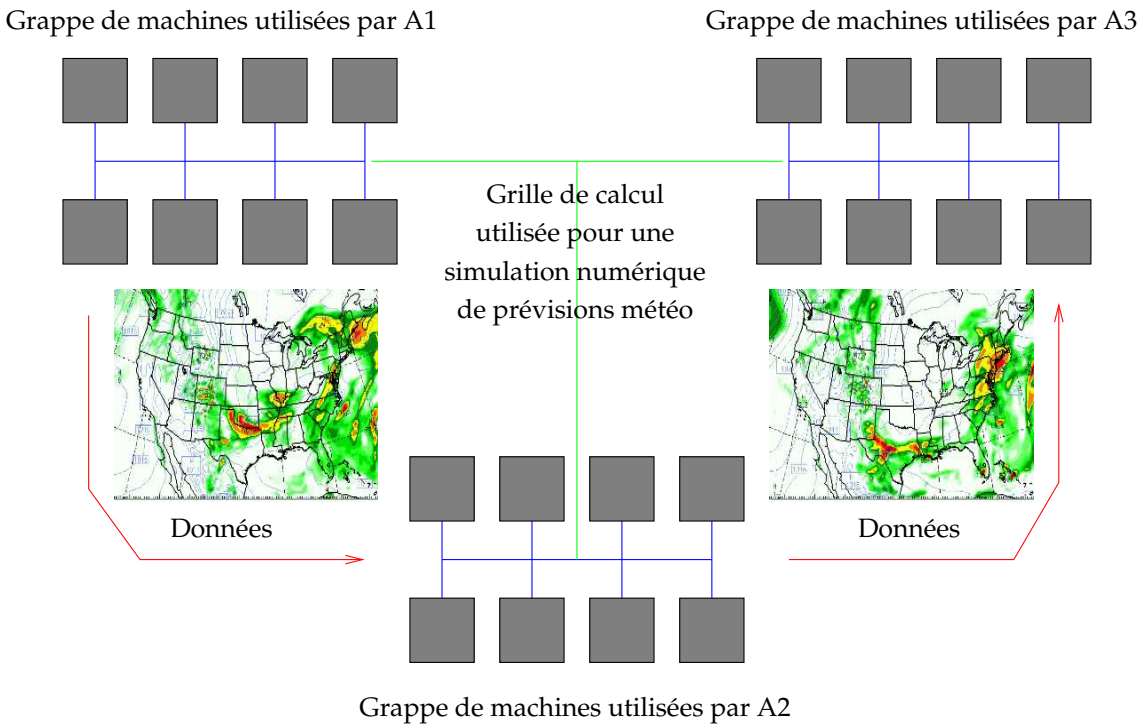

FIG. 2.1 – Simulation numérique de prévisions météo sur trois grappes de machines selon un schéma de type pipeline.

2.2 Proposition : service de partage de données

2.2.1 Scénarios typiques

Soit A1, A2 et A3 trois applications distribuées qui coopèrent sur un large ensemble de données suivant un schéma de type *pipe-line* comme le montre la figure 2.1.

Pour reprendre l'exemple d'une simulation météo pour plusieurs pays et plusieurs jours à l'avance, l'application A1 calcule les prévisions météo du jour, l'application A2 les prévisions météo du lendemain et l'application A3 celles du surlendemain pour chaque pays. Ainsi, les données produites par A1 sont utilisées par A2 pour le calcul des prévisions du lendemain et celles produites par A2 sont utilisées par A3 pour celles du surlendemain. Chaque application s'exécute sur un site différent, typiquement une grappe de machines en raison des besoins en puissance de calcul nécessaire. L'avantage d'utiliser une grille est que A1 peut simuler les prévisions météo d'un autre pays pendant que A2 simule celle du lendemain sur le pays précédent.

L'ensemble des systèmes de gestion de données sur la grille mentionnés à la section précédente ne permettent pas de résoudre de manière simple ce problème. En effet, pour transférer des données depuis l'application A1 vers l'application A2, if faut écrire les données sur un disque de A1, puis *explicitement* utiliser un service de transfert de fichiers afin de les déposer sur un disque de A2. Il est donc souhaitable de disposer au niveau applicatif des mécanismes permettant de rendre transparents la localisation et le transfert des données au sein du système, afin de découpler l'application utilisant les ressources de la grille de la

gestion des données qu'elle souhaite utiliser.

Supposons maintenant que ces trois mêmes applications coopèrent sur un schéma qui n'est plus de type *pipe-line* mais de type *écrivains multiples*. Par exemple, chaque application simule la météo d'un pays toujours plusieurs jours à l'avance, chacune s'occupant individuellement de la pluie, du vent et des nuages. Chaque site a donc besoin des données des autres sites pour avancer dans sa simulation et réciproquement. La gestion par le service de partage de données de cette localisation et de ce transfert des données permettrait, pour les applications, d'effectuer des lectures mais également des écritures dans une zone mémoire globale de manière transparente au niveau de l'application. Des mécanismes plus efficaces qu'un transfert intégral des données entre chaque écrivain sont également envisageables. Toutefois, le problème de la cohérence entre les différentes copies d'une même donnée se pose alors, mais à une échelle supérieure de celle traditionnellement abordée pour ce problème, qui était typiquement de l'ordre de la dizaine de nœuds. Enfin, si des nœuds du site sur lequel tourne l'application A2 deviennent indisponibles, des données nécessaires à l'application A3 peuvent être perdues. Afin d'assurer la persistance et permettre ainsi le partage de données, un mécanisme de gestion de la dynamicité est donc nécessaire.

2.2.2 Convergence entre pair-à-pair et mémoires virtuellement partagées

Les systèmes MVP et pair-à-pair ont des hypothèses et des résultats complémentaires. En effet, la MVP a donné des résultats intéressants concernant les modèles et les protocoles de cohérence permettant de partager des données de manière transparente, mais supposent un environnement statique, généralement homogène et avec peu de nœuds, entre une dizaine et une centaine. Les systèmes pair-à-pair eux ont donné des résultats intéressants concernant le partage de données non-modifiables de manière non transparente dans un environnement hétérogène, très volatile et à grande échelle, entre des milliers et des millions de machines.

Du fait de l'échelle, le contrôle des ressources disponibles sur les nœuds et le degré de confiance accordé à ces nœuds pour l'accès à ces ressources sont importants dans une MVP. En effet, toutes les machines composant une MVP sont administrées par la même entité physique. En revanche, les systèmes pair-à-pair sont une agrégation des machines situées à « la périphérie » d'Internet, c'est à dire chez des particuliers donc sans garantie de confiance. Dans des tels systèmes, un contrôle des ressources est donc difficile à mettre en œuvre. De plus, du fait de l'échelle de ces systèmes, les ressources sont hétérogènes notamment au niveau des machines, des systèmes d'exploitation et des liens réseaux contrairement aux MVP dans lesquelles les machines et leurs configurations sont le plus souvent identiques. Les applications visées sont également différentes. Les MVP sont utilisées par des applications de calculs numériques autorisant la modification des données par plusieurs nœuds en parallèle et donc complexes. Les systèmes pair-à-pair visent simplement le partage et le stockage de fichiers non modifiables. Ces caractéristiques sont résumées sur la première et la troisième colonne du tableau 2.1.

L'idée est de proposer un service de partage de données modifiables pour le calcul scientifique sur grille, *en s'inspirant essentiellement des points forts des approches MVP et pair-à-pair*, c'est à dire l'échelle atteinte pour les systèmes pair-à-pair et les modèles et protocoles de cohérence développés pour les MVP.

	MVP	Service de données pour la grille	P2P
Échelle	10^1–10^2	10^3	10^4–10^6
Contrôle des ressources	Fort	Moyen	Nul
Degré de confiance	Fort	Moyen	Nul
Degré de volatilité	Nul	Moyen	Fort
Homogénéité des ressources	Homogènes (grappes)	Assez hétérogènes (grappes de grappes)	Hétérogènes (Internet)
Type de données gérées	Modifiables	Modifiables	Non modifiables
Complexité des applications	Complexes	Complexes	Simples
Applications typiques	Calcul numérique	Calcul numérique et stockage de données	Partage et stockage de fichiers

TAB. 2.1 – Caractéristiques physiques d'un service de gestion de données inspiré des MVP et du P2P.

2.2.3 Caractéristiques de l'architecture physique cible

L'architecture physique visée pour ce service de partage de données présente des caractéristiques intermédiaires situées entre celles des architectures visées par les systèmes MVP et par les systèmes pair-à-pair. Ainsi, l'échelle visée est de l'ordre du milliers de nœuds regroupés en grappes inter-connectées. Les ressources sont donc hétérogènes. Le degré de dynamicité et de confiance est également intermédiaire entre celui des MVP et des systèmes pair-à-pair. En effet, nous considérons une grille de calcul formé d'un ensemble de grappes de machines administrées par des entités différentes mais liées par des accords pour cette mise à disposition des ressources de calculs. Les applications visées sont des applications de calcul numérique générant un volume de données important et nécessitant un partage de ces dernières entres les différentes grappes d'une grille. Ceci est résumé dans le tableau 2.1.

Les données manipulées dans le service visé sont de plusieurs gigaoctects à plusieurs téraoctets avec des nœuds ayant un temps moyen entre chaque indisponibilité de l'ordre de quelques heures. Les données doivent malgré tout avoir un niveau de persistance de l'ordre de quelques heures ou quelques jours. Enfin, la topologie réseau est hiérarchique. En effet, des réseaux locaux à haut débit (Go/s) avec des latences de l'ordre de quelques microsecondes (LAN) relient des nœuds d'une même grappe, et de réseaux haut débit avec des latences plus importantes de l'ordre de dizaines de millisecondes (WAN) comme VTHD ou GRID 5000, voire de secondes comme l'Internet transatlantique relient les grappes d'une grille.

2.2.4 Plan de travail

La principale difficulté dès que l'on se propose de gérer des données modifiables dans un système pair-à-pair est due à la nécessité d'intégrer des mécanismes de gestion de la *cohérence* des données répliquées. Pour ce problème, il n'y a pas encore à ce jour de solution adaptée à la grande échelle. Dans les systèmes comme Gnutella ou KaZaA, le problème est de retrouver la majorité ou toute les copies d'une même donnée disséminées sur les différents pairs constituant le système. Dans les systèmes comme CFS, PAST ou OceanStore qui s'appuient sur une DHT et sur des mécanismes de caches, lorsque le contenu d'une donnée est modifiée, il faut recalculer l'identifiant de la donnée, qui est souvent obtenu par hachage sur la donnée, afin de déterminer le nouveau pair responsable de cette donnée. De plus, il faut informer les clients possédant une copie de la donnée de la nouvelle valeur de la clé s'ils souhaitent accéder à la donnée modifiée. Pour la conception d'un système de partage de données modifiables de type MVP, l'utilisation d'un réseau pair-à-pair comme support nécessite de revisiter les hypothèses traditionnelles des MVP et d'en repenser les mécanismes afin de prendre en compte de nouvelles problématiques spécifiques à ce type de support comme la dynamicité du réseau, le facteur d'échelle et l'hétérogénéité. Comment augmenter le nombre de nœuds dans de tels systèmes est sans aucun doute un facteur limitant très important.

L'objectif visé dans ce rapport est de définir une architecture pour un service de partage de données sur une infrastructure pair-à-pair, donc volatile, hétérogène et à l'échelle de la grille, qui autorise la modification des données. Cette architecture est illustrée par la mise en œuvre de la plate-forme logicielle JuxMem, qui permet de répondre aux besoins des scénarios décrits à la section 2.2.1.

Chapitre 3

La plate-forme JuxMem : support pour un service de partage de données modifiables

3.1 Présentation générale

L'architecture que nous proposons pour un service de partage de données modifiables est composée de différentes entités. Cette section définit le rôle de chaque entité et présente la hiérarchie ainsi que les interactions entre les différentes entités. Cette architecture est illustrée par une plate-forme logicielle appelée JuxMem (pour *Juxtaposed Memory*).

3.1.1 Les pairs

L'entité de base de l'architecture est le pair. Un pair est une entité capable de communiquer. Il existe trois type de pairs : *fournisseur*, *gestionnaire*, et *client*, ayant chacun un rôle spécifique à jouer.

Le pair fournisseur offre une zone mémoire afin d'accueillir d'éventuels bloc de données. Les pairs fournisseurs sont organisés en groupes, comme expliqué dans la section 3.1.2. Ces groupes correspondent généralement à des grappes de machines.

Le pair gestionnaire est responsable de l'espace mémoire fourni par l'ensemble des pairs fournisseurs d'un groupe.

Le pair client fournit l'interface d'accès au service, permettant d'allouer des zones mémoires dans l'espace offert par les pairs fournisseurs, de repérer et de manipuler les bloc de données introduits dans le système.

Cette séparation précise des rôles entre les différents pairs vise à clarifier la conception du service, mais également à faciliter son implémentation. Il faut distinguer un nœud d'un pair : un nœud peut jouer différents rôles et ce de manière simultanée. Ainsi, on notera dans la suite du rapport un nœud qui joue le rôle de fournisseur un *pair fournisseur* et un nœud qui joue le rôle de fournisseur et de gestionnaire un *pair fournisseur-gestionnaire*.

3.1.2 Les groupes

La définition d'un groupe a pour intérêt principal de rassembler un ensemble de pairs qui partagent une entité commune : l'accès à un service, à une ressource, etc. Chaque instance d'un groupe dispose d'un identifiant unique permettant de le distinguer des autres groupes. L'architecture du service définit trois types de groupes. De la même manière que pour les pairs, chaque groupe a un rôle précis.

Le groupe `juxmem` inclut tous les types de pairs présentés à la section 3.1.1. C'est le groupe principal du service, il est unique. Les pairs clients ne sont membres que de ce groupe mais peuvent communiquer avec les autres groupes.

Les groupes `cluster` regroupent les pairs fournisseurs d'une grappe gérés par un pair gestionnaire.

Les groupes `data` regroupent les pairs fournisseurs qui hébergent un même bloc de données. En effet, un bloc de données peut être répliqué sur de multiples pairs fournisseurs. Un des pairs est choisi pour être le responsable du groupe, typiquement le premier à héberger le bloc de données.

Chaque nœud peut être membre d'un ou plusieurs groupes selon les rôles qu'il joue et dispose d'un identifiant de pair pour chaque groupe auquel il appartient, permettant de le distinguer au sein du groupe. L'intérêt des groupes est tout d'abord de gagner en lisibilité vis-à-vis de la conception du service. D'autre part, ils permettent également de disposer d'une unique entité responsable des communications vis-à-vis de l'extérieur du groupe : un canal de communication identifié, au bout duquel le responsable du groupe écoute. Cela permet un changement de responsable de manière transparente vis-à-vis de l'extérieur du groupe. De plus, cela permet de définir un espace protégé et sécurisé ainsi que de restreindre les communications au sein d'un groupe. La gestion du groupe est de ce fait transparente vis-à-vis de l'extérieur du groupe. Enfin, la perte d'un des pairs membre du groupe n'entraîne pas forcément l'indisponibilité du service rendu par le groupe.

3.1.3 Hiérarchie entre les entités

L'architecture du service reflète l'architecture *hiérarchique* de la grille : le groupe `juxmem`, auquel il faut appartenir pour fournir et avoir accès au service, englobe les groupes de type `data` et `cluster`. La figure 3.1 présente la hiérarchie des entités dans le réseau virtuel défini par le service, au-dessus du réseau physique. Ce réseau physique est une grille composée de trois grappes de machines A, B et C inter-connectées. Au niveau logique ces grappes sont représentées par les groupes `cluster` A, B et C. Un bloc de données est hébergé par le système, formant ainsi un groupe `data`. Il est à noter qu'un nœud peut être à la fois un pair gestionnaire, client et fournisseur ; toutefois ici pour plus de clarté, chaque nœud ne joue qu'un seul rôle.

Du point de vue des pairs clients seuls les groupes `cluster` et les groupes `data` sont visibles. Ainsi, un pair client membre du groupe `juxmem` ne voit pas l'ensemble des pairs fournisseurs des différentes grappes de la grille. Les groupes `data` et `cluster` sont situés sur le même niveau de la hiérarchie des groupes puisque le groupe `data` recouvre une partie des groupes `cluster` A et C. En effet, un bloc de données n'est pas spécifique à une grappe de machines et peut comme ici être réparti sur deux grappes, en l'occurrence A et C. De plus, rien n'oblige à lier un groupe de type `cluster` à une unique grappe physique.

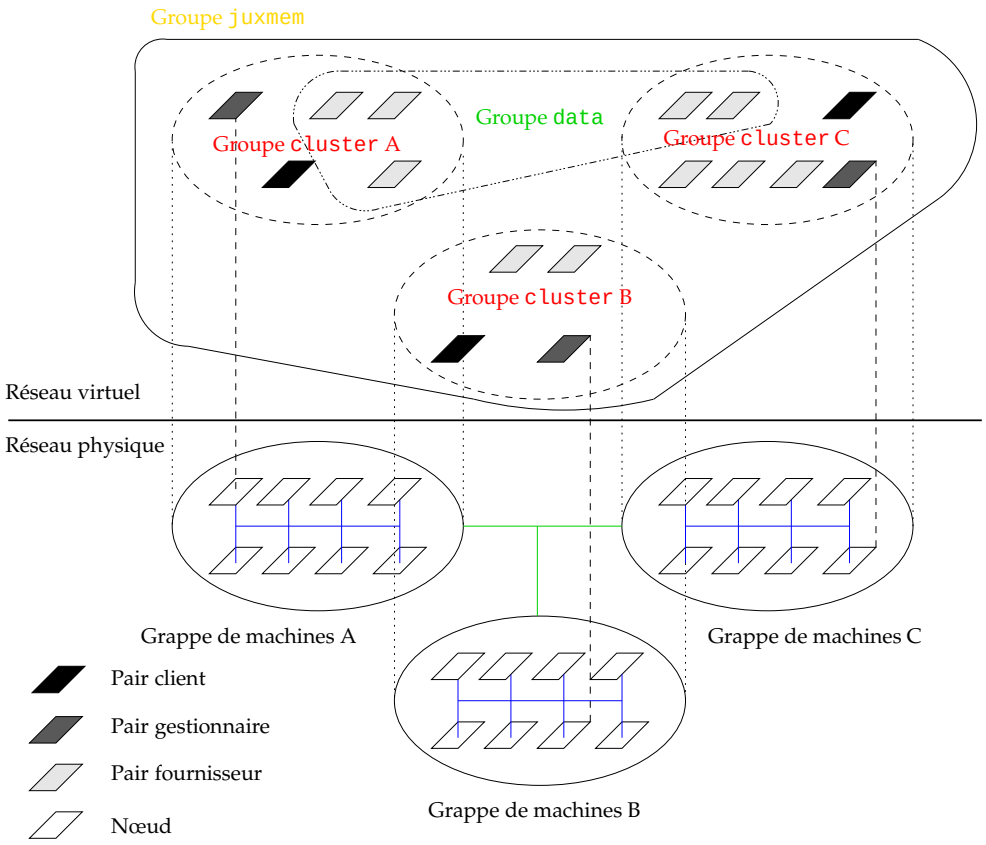

FIG. 3.1 – Hiérarchie des entités dans le réseau virtuel défini par le service.

Ainsi, un groupe de type `cluster` peut déjà représenter une fédération de grappe. On peut également envisager de diviser une unique grappe physique en deux grappes virtuelles. Enfin, l'architecture du service est dynamique puisque les groupes de type `cluster` et `data` peuvent être créés à l'exécution. Ainsi, pour chaque bloc de données introduits dans le système, un groupe de type `data` est automatiquement instancié.

3.1.4 Interface du service

Le service proposé offre au niveau applicatif une *interface* (API, pour *application programming interface*) composée de 8 opérations. La signature et le rôle de chaque méthode de l'API sont les suivants.

`alloc (size, options)` permet d'allouer une zone mémoire d'une taille de `size` sur une grappe. Le paramètre `options` permet de spécifier le niveau de réplication et le protocole de cohérence utilisé par défaut pour la gestion des copies des bloc de données à stocker. Dans la version courante, ce paramètre n'est pas encore utilisé. La primitive retourne un identifiant qui peut être assimilé au niveau applicatif à un identifiant de bloc de données, mais qui correspond à l'identifiant de la zone mémoire allouée (section 3.2).

`map (id, options)` permet de récupérer l'annonce du canal de communications per-

mettant de manipuler la zone mémoire identifiée par `id`. Le paramètre `options` permet de spécifier des paramètres pour la "vue" du bloc de données que souhaite avoir un pair client, comme par exemple le degré de cohérence. Dans la version courante, ce paramètre n'est pas encore utilisé.

`lock (id)` permet de verrouiller la zone mémoire spécifiée par l'identifiant `id`. Il est à noter que pour chaque identifiant, donc pour chaque bloc de données, il existe un verrou. Le pair client qui a appelé la primitive `lock` est bloqué jusqu'à ce que l'opération réussisse.

`unlock (id)` permet de déverrouiller la zone mémoire spécifiée par l'identifiant `id`. Le pair client est bloqué jusqu'à ce que l'opération réussisse. Cette attente est nécessaire afin de s'assurer que le verrou a bien été relâché.

`put (id, value)` permet de modifier la valeur de la zone mémoire spécifiée par l'identifiant `id`. La nouvelle valeur du bloc de la donnée est alors `value`. Cette méthode retourne un valeur attestant de la prise en compte de la modification du bloc de la donnée par le service.

`get (id)` permet d'obtenir la valeur courante de la zone mémoire spécifiée par l'identifiant `id`. Le pair client qui a utilisé cette opération n'attend que la première réponse qu'il reçoit afin de tirer parti de la localité physique.

`delete (id)` permet de demander la destruction de la zone mémoire spécifiée par l'identifiant `id`. Il est à noter que cette destruction n'est pas garantie.

`reconfigure (cluster, client, size)` permet de reconfigurer un nœud. Le paramètre `cluster` permet d'indiquer si le pair physique va jouer le rôle de pair gestionnaire, le paramètre `client` permet d'indiquer si le pair physique joue le rôle de pair client et enfin le paramètre `size` permet d'indiquer la tialle de la zone mémoire que le pair fournisseur va offrir. Si cette valeur est à 0 le nœud ne va pas jouer le rôle de pair fournisseur.

3.2 Gestion des ressources mémoires

L'objectif du service est de partager des zones de mémoire à l'échelle de la grille. Il faut donc proposer un mécanisme de gestion de ces ressources mémoires compatible avec l'architecture logicielle hiérarchique du service et tenant compte des contraintes fixées, notamment la volatilité des entités constituant le système.

3.2.1 Publication des annonces de ressources

Les ressources mémoires sont gérées par l'utilisation d'*annonces* qui ont pour objectif d'informer sur la disponibilité d'une zone mémoire. Afin d'être compatible avec l'architecture proposée à la section 3.1, il existe deux types d'annonces :

Annonce de type fournisseur. Chaque pair fournisseur informe le pair gestionnaire auquel il est connecté, de l'espace mémoire dont il dispose, par l'intermédiaire de son champ `memory`. Cette annonce est publiée au sein du groupe `cluster` dont est membre le pair fournisseur en question.

Annonce de type grappe. Le pair gestionnaire de chaque grappe est responsable du stockage des différentes annonces de type fournisseur présentes au sein de son groupe publiées par les pairs fournisseurs. Il publie alors une annonce de type grappe à l'extérieur de son groupe, c'est à dire au sein du groupe `juxmem`. Cette annonce a pour objectif de publier l'espace mémoire total disponible sur la grappe par l'intermédiaire de son champ `memory`, ainsi que le niveau maximum de réplication possible pour le bloc de données sur cette même grappe, c'est-à-dire le nombre de pairs fournisseurs disponibles dans cette grappe de machines, par l'intermédiaire de son champ `replication`. Ces annonces sont utilisées au sein du groupe `juxmem` par les pairs client pour demander l'allocation d'une zone mémoire afin d'y stocker un bloc de données.

L'une des contraintes fixées est la volatilité des nœuds constituant le service. Ainsi, les annonces publiées à l'instant t ne sont plus forcément valides à l'instant $t + 1$. En effet, un pair fournisseur offrant une zone mémoire d'une certaine taille peut disparaître à tout moment de l'architecture. Le mécanisme utilisé pour gérer cette volatilité des pairs est de republier l'annonce de type grappe lorsqu'une variation de l'espace global est détectée par le pair gestionnaire responsable de la grappe. Enfin, afin d'éviter que trop d'annonces invalides soient disponibles, celles-ci ont une durée de vie fixée. Ainsi, au bout de cet intervalle de temps (par exemple : 2 heures), les annonces sont détruites du réseau virtuel. Il faut donc également republier périodiquement ces annonces même si aucune modification de l'espace global ou du nombre de pairs fournisseurs qui constituent la grappe n'a été détectée.

3.2.2 Placement et localisation des annonces de ressources

Les annonces de type fournisseur sont publiées sur le pair gestionnaire responsable de la grappe à laquelle les pairs fournisseurs sont rattachés. Cette publication se fait au sein du groupe `cluster` de la grappe (voir figure 3.1). Les annonces de type grappe sont publiées au sein du groupe `juxmem` (voir figure 3.1) et jouent de ce fait un rôle central dans l'architecture. Un pair gestionnaire est chargé de faire le lien entre le groupe `cluster` et le groupe `juxmem`. Ils sont donc utilisés pour constituer un réseau de pairs organisés en utilisant une DHT au niveau du groupe `juxmem`, afin de former l'ossature du service. Cette ossature est représentée par l'anneau de la figure 3.2, les pairs gestionnaires étant représentés par les carrés rouges. Sur cette figure chaque pair gestionnaire G1 à G6 est responsable d'une grappe de machines respectivement A1 à A6 constituée de six nœuds. Le champ mémoire des annonces de type grappe est utilisé pour générer un identifiant, par application d'une fonction de hachage. Cet identifiant est utilisé pour déterminer le pair gestionnaire responsable de l'annonce au niveau du groupe `juxmem`. Le pair gestionnaire responsable de l'annonce n'est pas forcément le pair qui stocke l'annonce ou qui l'a publiée, il a seulement connaissance de la localisation de l'annonce dans le système.

L'utilisation d'une DHT pour la gestion de la localisation des annonces est motivée par le besoin d'effectuer une recherche efficace et précise. Cette approche a fait l'objet d'études lors du rapport bibliographique de ce stage. Cette technique de localisation et de routage semble la technique la plus aboutie et la plus adéquate au problème : chaque client souhaite en effet disposer de toutes les annonces de type grappe répondant à une certaine taille mémoire. Or cette technique permet de garantir de localiser toutes les annonces répondant à un critère et cela avec un coût réduit en nombre de messages. Compte tenu de la contrainte de volatilité, la solution de routage par répertoire centralisé n'est pas envisageable du fait de la présence

d'un serveur central. La solution de routage par inondation génère un trop grand nombre de messages et ne fournit aucune garantie sur l'exactitude des réponses. Cette technique de localisation n'est donc pas envisageable non plus.

3.2.3 Cheminement d'une requête d'allocation

Un pair client fait des requêtes d'allocation en spécifiant la taille souhaitée pour la zone mémoire. Les différentes phases d'une requête d'allocation, numérotées sur la figure 3.2, sont les suivantes.

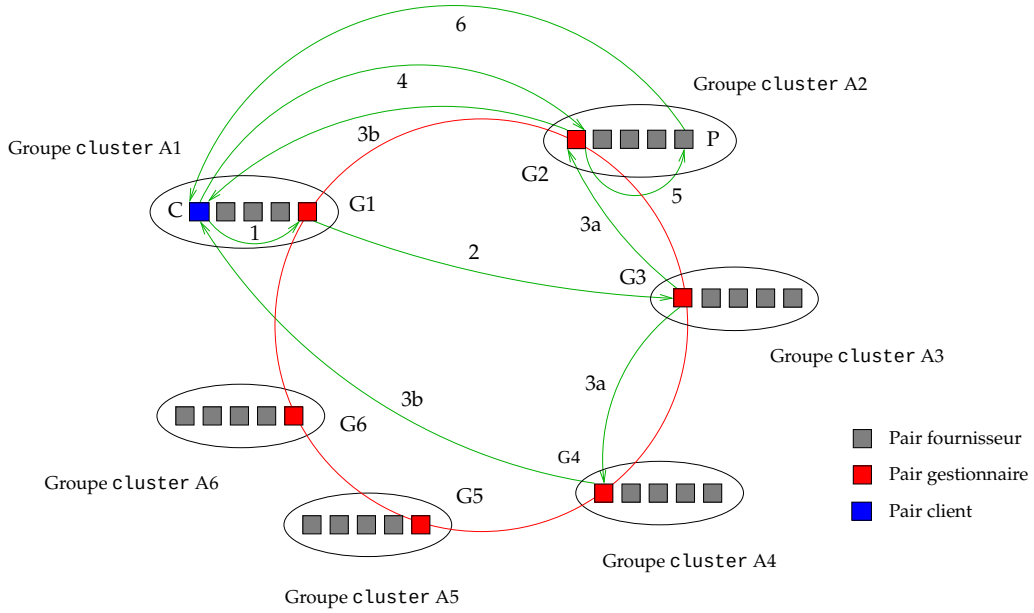

FIG. 3.2 – Étapes d'une requête d'allocation par un pair client.

1. Le pair client choisit la taille et le degré de réplication pour la zone mémoire qu'il souhaite allouer et envoie sa requête au pair gestionnaire auquel il est connecté. Dans l'exemple, le pair client C du groupe `cluster` A1 souhaite faire une requête d'allocation d'une zone mémoire de taille 10 avec un degré de réplication de 2, il soumet donc sa requête au pair gestionnaire G1.
2. Le pair gestionnaire détermine le pair gestionnaire responsable des annonces répondant à la taille spécifiée par la requête. Ce calcul s'effectue par application d'un fonction de hachage sur, notamment, la taille mémoire fournie par la requête. Dans l'exemple, le pair gestionnaire G1 va déterminer que le pair responsable des annonces ayant une taille de taille 10 est le pair gestionnaire G3, et donc lui transmettre la requête.
3. Le pair gestionnaire auquel on a transmis la requête recherche dans son cache local la localisation des annonces répondant à ce critère. Il va ainsi demander aux pairs gestionnaires qu'il a localisé de transmettre leurs annonces au pair client. Dans l'exemple, le pair gestionnaire G3 va alors demander au pair gestionnaire G2 et G4 de transmettre leurs annonces de type grappe au pair client C.

4. Le pair client peut alors choisir la meilleure annonce de type gestionnaire selon ses critères afin de lui soumettre une requête d'allocation. En effet, cette annonce contient l'identifiant du pair gestionnaire responsable de la grappe. La meilleure annonce peut être choisie selon le niveau de réplication offert dans le groupe qui a publié l'annonce. Dans l'exemple, le pair client C choisi le pair gestionnaire G2, la grappe sous-jacente offrant actuellement un degré de réplication supérieur à la grappe dont est responsable le pair gestionnaire G4, et lui soumet sa requête d'allocation de 10.
5. Le pair gestionnaire qui reçoit une requête d'allocation va alors vérifier qu'il peut satisfaire cette requête. S'il peut la satisfaire il va demander à l'un des pairs fournisseurs du groupe gestionnaire dont il est responsable d'allouer une zone mémoire. S'il ne peut pas satisfaire la requête il va en avertir le pair client. Dans notre exemple, le pair gestionnaire G2 reçoit une requête d'allocation de 10 qu'il peut satisfaire et va alors demander à l'un de ses pairs fournisseurs, par exemple P, d'allouer une zone mémoire de 10.
6. Le pair fournisseur sollicité va alors vérifier s'il peut satisfaire cette requête. S'il peut la satisfaire il va directement retourner une annonce au pair client pour la zone allouée. S'il ne peut pas satisfaire cette demande, il va renvoyer au pair gestionnaire responsable de la grappe un message pour indiquer son incapacité à fournir cet espace mémoire. Le pair gestionnaire doit alors chercher un autre fournisseur susceptible de satisfaire la requête d'allocation. Si aucun pair fournisseur ne peut satisfaire cette requête, un message d'erreur est renvoyé au pair client. Dans l'exemple, le pair fournisseur P peut satisfaire cette requête et va donc créer une zone mémoire de taille 10 puis va renvoyer l'annonce de cette zone mémoire au pair client C. Il va également devenir responsable pour le groupe `data` associé à cette zone mémoire, ainsi que chercher à répliquer cette zone.

Dans le cas où le pair gestionnaire ne peut pas satisfaire la requête d'allocation du pair client, ce dernier va parcourir la liste des annonces dont il dispose afin de trouver un pair gestionnaire capable de lui fournir l'espace mémoire demandé. Enfin, si aucun pair gestionnaire n'est en mesure de satisfaire la demande, le pair client va à nouveau exécuter les opérations 1 à 4 mais en augmentant à chaque fois l'espace mémoire demandé. En effet, chaque pair gestionnaire publie seulement l'espace mémoire total dont dispose la grappe. Or, il est probable que l'espace mémoire demandé par un pair client soit de taille inférieure à l'espace dont disposent les grappes. Le pair client augmente donc l'espace recherché selon une loi, par exemple logarithmique, afin de trouver une grappe pouvant satisfaire sa requête. Si au bout de N exécutions (par exemple $N = 3$) le pair client n'a pu obtenir l'allocation d'une zone mémoire, une erreur au niveau applicatif est retournée. Il est à noter que les pairs gestionnaires publiant l'espace total dont dispose la grappe, les bloc de données sont susceptibles d'être découpés et d'être répartis sur plusieurs pairs fournisseurs afin d'y être stockées.

3.3 Gestion des blocs de données partagés

Le service a pour objectif la persistance et la transparence de la localisation des blocs de données afin de permettre le partage et d'éviter la réalisation au niveau applicatif d'une gestion explicite de ces bloc de données.

3.3.1 Stockage et accès transparent

L'allocation d'une zone mémoire par un pair client se traduit par la création d'un groupe pour le bloc de données et par l'envoi d'une annonce au pair client, permettant de communiquer avec le groupe précédemment créé. Cette annonce est stockée au niveau du pair client, mais seul l'identifiant de cette annonce est retourné au niveau applicatif. Cette annonce est également publiée au niveau du groupe `juxmem` sur les pairs gestionnaires toujours par utilisation d'une DHT. L'accès aux blocs de données par d'autres pairs clients est ainsi possible par la seule connaissance, au niveau applicatif, de l'identifiant de l'annonce de la zone mémoire allouée pour le bloc de données auquel on souhaite accéder. La recherche de cette annonce est faite sur le même principe qu'une requête d'allocation. Toutefois ici, une annonce unique est retournée au pair client. Le stockage des blocs de données n'est pas lié à un pair client et de ce fait est persistant. De plus, le service n'offre que des zones mémoires non typées, ainsi tous les types de données peuvent être stockés. Enfin, la destruction des bloc de données est possible mais n'est pas garantie. En effet, un pair fournisseur hébergeant une copie peut devenir indisponible au moment de la demande de destruction de la zone mémoire allouée pour le bloc de données, suite à un problème technique sur le lien réseau le reliant au reste de la grappe. Le bloc de données peut donc toujours être présent dans le système, une fois le problème technique résolu, malgré la demande de destruction émise par un pair client.

La localisation des blocs de données est transparent au niveau applicatif, puisque pour manipuler les blocs de données l'application n'a qu'à fournir l'identifiant de la zone mémoire hébergeant le bloc de données. C'est la mission du service de localiser, à partir de cet identifiant, le bloc de données afin de transmettre au groupe responsable du bloc, les opérations que le pair client souhaite réaliser.

3.3.2 Cohérence et synchronisation

L'une des contraintes du service est le fait que les données soient *modifiables*. Plusieurs pairs clients doivent pouvoir accéder au même bloc de données afin de la modifier. Le service propose donc, en plus de l'opération *get* qui permet de récupérer la valeur d'un bloc de données, l'opération *put* permettant de modifier un bloc de données. Il est à noter que les pairs clients ne stockent pas de copie locale du bloc de données ; le résultat d'une lecture est une valeur qui ne sera pas actualisée lors d'éventuelles mises à jour ultérieures. Ainsi, pour chaque appel à l'opération *get*, le groupe du bloc de la donnée sera sollicité afin d'en fournir la valeur. Concernant cette opération, il est à noter que l'ensemble des pairs fournisseurs qui hébergent une copie du bloc de données répondent à la requête. Mais le pair client ne prend en compte que la première réponse afin de tirer parti de la localité physique. En revanche, chaque bloc de données est répliqué sur un certain nombre de pairs fournisseurs pour une meilleure disponibilité.

Il faut donc s'assurer de la cohérence entre les copies d'un même bloc de données mais ceci concerne uniquement les pairs fournisseurs et non pas les pairs clients. L'emploi d'une solution de type *multicast* permet de résoudre ce problème : les différentes copies d'un même bloc de données sont ainsi simultanément mises à jour lors de toute opération de modification par un pair client. Il faut noter que cette opération de multicast est réalisée au niveau du réseau virtuel défini par les groupes de pairs. Ceci permet de s'affranchir des contraintes physiques et des configurations de certains réseaux, comme l'absence de routine de multi-

cast sur une grappe. Ce problème éventuel étant à la charge des couches de communication de niveau inférieur et reste donc transparent à notre niveau. Enfin, le pair client qui a réalisé l'opération *put* attend une réponse d'un des pairs fournisseurs qui héberge une copie du bloc de données afin de s'assurer de la modification effective du bloc de données.

La gestion de la synchronisation entre les pairs clients est réalisée par l'utilisation d'un mécanisme de type verrou. Ainsi, un pair client souhaitant modifier un bloc de données doit au préalable poser un verrou sur celle-ci en utilisant l'opération *lock*. Les autres pairs clients sont ainsi bloqués dans l'attente de la libération du verrou. Pour ôter un verrou, il doit alors utiliser l'opération *unlock*. Prenons l'exemple de deux requêtes d'écriture en parallèle, émises par les pairs client C1 et C2, sur un même bloc de données D stockée dans une zone mémoire ayant comme identifiant *id*. Les deux pairs clients font donc la séquence suivante, dont les étapes sont schématisées sur la figure 3.3, de manière simultanée :

```
lock(id)
put(id,value)
unlock(id)
```

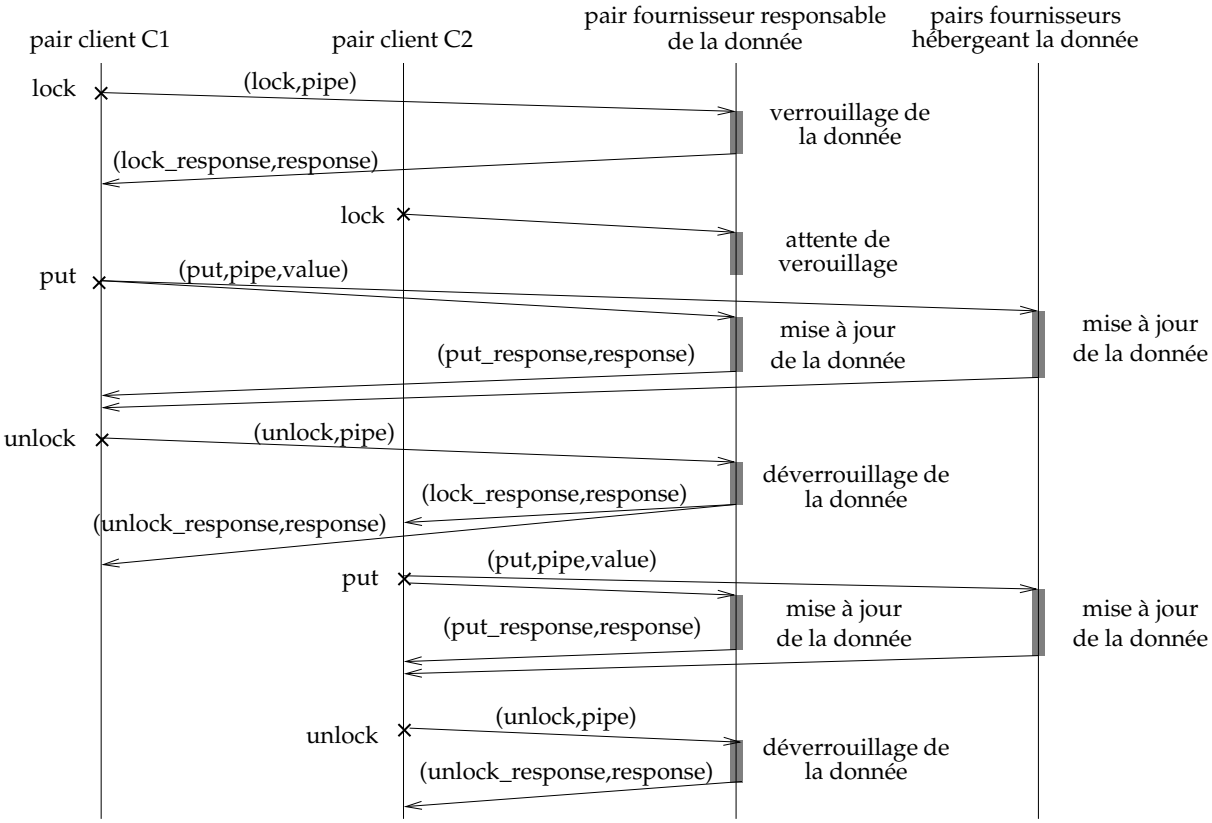

FIG. 3.3 – Étapes d'une séquence *lock*, *put* et *unlock* réalisée par deux pairs clients.

À travers l'utilisation de l'identifiant `id`, les pairs clients s'adressent au groupe `data` responsable du bloc de données D. Toutefois, lors des opérations de type *lock* et *unlock*, seul le responsable du groupe du bloc de données traite la requête. Les pairs clients sont donc

bloqués par ce même responsable en l'attente d'une réponse leur signalant la prise effective du verrou. Supposons que le pair client C1 soit le premier à obtenir le verrou pour le bloc de données (voir figure 3.3). Le responsable du bloc de données D aura mémorisé le fait que le verrou a été positionné par le pair client C1, par l'intermédiaire de son identifiant de pair. Lors de la réception de la demande de verrouillage du pair client C2 sur le bloc de données D, le responsable va ajouter dans la file d'attente l'identifiant du pair client C2 car le verrou est déjà pris. Le pair client C2 va donc se bloquer. Lorsque le pair client C1 aura fini de modifier le bloc de données, il va demander au responsable d'ôter son verrou. Le pair responsable va alors acquitter le pair client C1 et transmettre au pair client C2 le message de prise du verrou autorisant à modifier le bloc de données.

3.4 Gestion de la volatilité

Dans ce rapport, on définit la volatilité d'un pair comme étant le fait qu'il puisse disparaître et éventuellement réapparaître. De notre point de vue, ceci est différent de la dynamicité, que l'on définit comme par exemple étant la possibilité pour un pair fournisseur de modifier la taille de la zone mémoire qu'il offre et plus généralement tout type de reconfiguration possible sur un pair.

3.4.1 Volatilité des pairs gestionnaires

Le pair gestionnaire responsable d'une grappe a des fonctionnalités vitales pour le service comme le traitement des opérations d'allocation d'une zone mémoire sur la grappe sous-jacente. La perte de ce pair peut donc entraîner l'indisponibilité des ressources d'une grappe. Or, dans l'architecture visée la contrainte de tolérance à la volatilité des pairs est présente. Le rôle de pair gestionnaire (noté *responsable principal*) est donc automatiquement dupliqué sur l'un des pairs fournisseurs de la grappe (noté *responsable secondaire*). Le responsable secondaire n'a toutefois pas toutes les fonctionnalités du responsable principal activées. Ainsi, il ne peut pas publier les ressources mémoires de la grappe. En revanche, des pairs fournisseurs peuvent se connecter à lui pour y publier l'espace mémoire dont ils disposent. Un mécanisme d'échange d'annonces entre responsables est donc utilisé pour avertir de la publication de nouvelles annonces. Ainsi, si le responsable principal devient indisponible, le responsable secondaire connaît de manière quasi-exacte l'espace mémoire disponible sur la grappe. La disponibilité d'une grappe est donc maximisée afin de minimiser les perturbations vis-à-vis des pairs clients.

Cependant, cette simple duplication n'est pas suffisante, puisque le responsable secondaire peut lui aussi devenir indisponible. Il existe donc un mécanisme basé sur des échanges périodiques de messages de vie (*heartbeat*) entre responsable principal et responsable secondaire. Ainsi, le responsable principal peut détecter l'indisponibilité du responsable secondaire puis demander à un autre pair fournisseur présent dans la grappe de devenir responsable secondaire. De la même manière, le responsable secondaire peut détecter l'indisponibilité du responsable principal et devenir responsable principal puis demander à un pair fournisseur de la grappe de jouer le rôle de responsable secondaire. Ces changements de responsable au sein du groupe `cluster` d'une grappe ne sont pas visibles depuis l'extérieur du groupe et n'entraînent donc aucune indisponibilité temporaire du service. En effet, le responsable secondaire lorsqu'il devient responsable principal va se connecter, de manière

transparente, sur le canal de communication utilisé par le précédent responsable principal (comme expliqué à la section 3.1.2)

Prenons la configuration suivante : une grappe constituée de cinq nœuds, chaque nœud héberge un pair. Les différents pairs sont : un pair gestionnaire - fournisseur G1, et des pairs fournisseurs G2 à G5. Le pair G1 va automatiquement demander à l'un des pairs fournisseurs, sauf lui-même, de répliquer le rôle de responsable. Par exemple le pair fournisseur G2 va devenir responsable secondaire.

Supposons que le pair G2 tombe. Le pair G1 va le découvrir lors de la prochaine tentative pour échanger un message de vie avec le pair G2. Il va alors automatiquement chercher à répliquer le rôle de responsable secondaire sur un autre pair fournisseur, par exemple le pair G3. Les différentes étapes de cette réplication d'un responsable principal sur un pair fournisseur sont schématisées sur la figure 3.4. Les traits pointillés symbolisent l'absence de pairs ou de messages par rapport à un état sans perte de pairs.

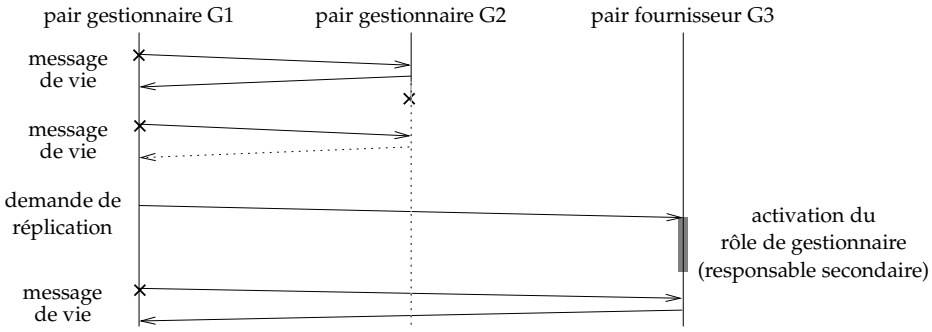

FIG. 3.4 – Étapes d'une réplication d'un responsable principal sur un pair fournisseur.

Si en revanche le pair G1 devient indisponible, de la même manière le pair G2 va le découvrir et automatiquement devenir responsable principal, republier l'espace mémoire disponible de la grappe puis demander, par exemple, au pair fournisseur G4 de jouer le rôle de responsable secondaire. Les différentes étapes de cette réplication d'un responsable secondaire sur un pair fournisseur sont schématisées sur la figure 3.5. Les traits pointillés symbolisent l'absence de pairs ou de messages par rapport à un état sans perte de pairs.

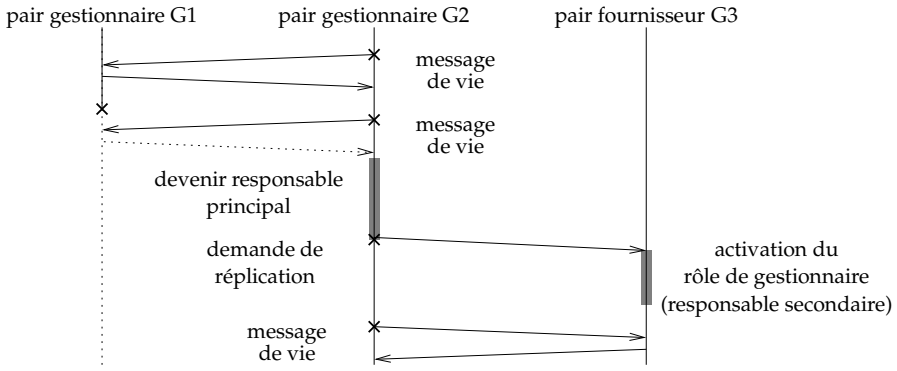

FIG. 3.5 – Étapes d'une réplication d'un responsable secondaire sur un pair fournisseur.

3.4.2 Volatilité des pairs fournisseurs

La volatilité des pairs fournisseurs est traitée par l'utilisation d'un mécanisme d'échange périodique de messages de vie entre le responsable principal du groupe `cluster` et les pairs fournisseurs de la grappe. Reprenons la configuration définie à la section 3.4.1 lorsque le responsable principal s'est répliqué sur le pair fournisseur G2 et supposons que seul le responsable principal teste tous les pairs fournisseurs. Initialement, les pairs fournisseurs G2 et G5 se sont connectés au pair gestionnaire G1. Après son initialisation, le responsable secondaire G2 va récupérer, du responsable principal, l'ensemble des annonces dont ce dernier dispose, grâce au mécanisme d'échange des annonces de type fournisseur. Supposons que le pair fournisseur G3 disparaisse. Lors du prochain échange de message de vie avec les pairs fournisseurs, le pair gestionnaire G1 va détecter cette disparition et donc détruire cette annonce de son cache et republier l'annonce à jour de l'espace mémoire disponible sur la grappe. Toutefois, en raison du mécanisme d'échange des annonces entre responsables, le responsable principal va récupérer du responsable secondaire, le pair G2, l'annonce de l'espace disponible sur le pair G3. Il va donc constater une modification de l'état de la grappe et publier l'annonce à jour de celle-ci, c'est à dire en incluant l'annonce du pair fournisseur G3 dans son calcul de l'espace mémoire dont dispose la grappe. Toute cette séquence d'opérations inutiles, schématisées à la figure 3.6, va se répéter jusqu'à ce que l'annonce du pair fournisseur G3 sur le pair G2 arrive à expiration.

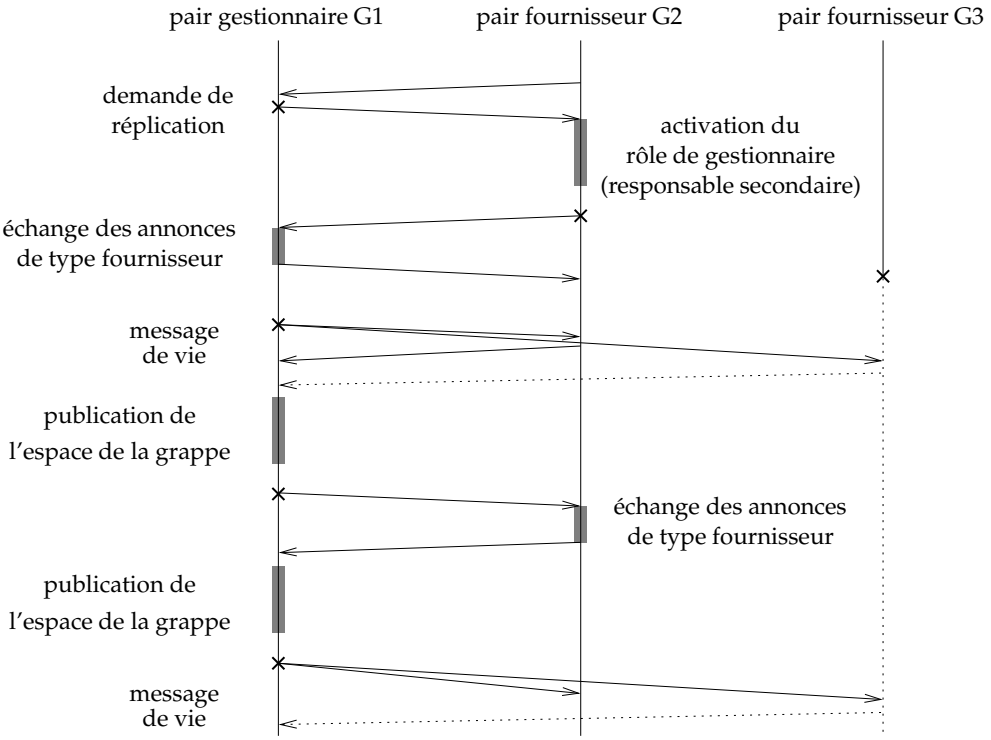

FIG. 3.6 – Échanges d'annonces entre deux responsables d'un groupe `cluster` menant à des opérations inutiles.

Ces opérations peuvent être évitées si tous les responsables testent tous les pairs fournis-

seurs de la grappe. Un mécanisme plus élaboré de mise à jour et d'invalidation des annonces de type fournisseur entre les responsables de grappes permettrait d'éviter que tous les responsables testent tous les pairs fournisseurs de la grappe. Un tel mécanisme sera disponible dans une prochaine version de JuxMem.

3.4.3 Redondance des blocs de données

Les blocs de données sont hébergés par les pairs fournisseurs. La contrainte de volatilité des pairs oblige donc à répliquer les blocs de données pour garantir leur disponibilité et leur persistance afin de rendre possible le partage de blocs de données. Toutefois, une simple réplication statique d'un bloc de données sur un certain nombre de pairs fournisseurs n'est pas suffisante. En effet, l'ensemble des pairs hébergeant une copie du même bloc de données peuvent successivement devenir indisponibles. Un *contrôle dynamique du nombre de copies* d'un bloc de données est donc nécessaire. Le mécanisme retenu est similaire à celui décrit à la section 3.4.1. Au sein du groupe `data`, il existe un responsable principal et un responsable secondaire ayant pour rôle de vérifier la disponibilité des responsables pour le groupe considéré, et, si besoin, de répliquer ce rôle sur un autre pair fournisseur hébergeant un bloc de données. Le responsable principal est également chargé de contrôler le niveau de réplication du bloc de données, et, si ce nombre est inférieur à celui souhaité par les pairs clients, de chercher puis demander à un pair fournisseur, s'il en trouve un, d'héberger une copie du bloc. Lorsque le responsable principal du bloc de données décide de la répliquer, il doit au préalable la verrouiller afin d'assurer la cohérence de la nouvelle copie par rapport aux autres copies. Le pair qui va accueillir cette copie est alors en charge de la déverrouiller. Dans ce cas, le verrou migre donc puisque les clients qui le posent et qui le retirent sont différents. La perte du verrou peut alors se produire si le pair fournisseur qui va héberger un bloc de données devient indisponible alors qu'il n'a pas encore ôté son verrou. La solution retenue est de mettre un *time-out* sur la durée de réplication d'un bloc de données, puis de tester la disponibilité du pair qui devait accueillir une copie du bloc de données. Si le pair est détecté comme indisponible, le responsable principal se charge alors de déverrouiller le bloc de données. Si le pair est détecté comme disponible, le *time-out* est relancé pour une nouvelle tentative.

Les deux responsables traitent en parallèle les requêtes de verrouillage et de déverrouillage pour les blocs de données dont ils sont responsables. Ainsi, si le responsable principal devient indisponible, le responsable secondaire va devenir principal. Il sait quel pair a verrouillé le bloc de données et donc saura interdire l'accès à un bloc de données à tout autre pair client, mais également déverrouiller le bloc de données lorsque le possesseur du verrou en fera la demande. Un responsable secondaire ne fait donc que stocker les informations qui seront utiles pour ôter un verrou d'un bloc de données, lorsqu'il deviendra responsable principal.

Chapitre 4

Implémentation de JuxMem sur la plate-forme JXTA

4.1 La plate-forme générique pair-à-pair JXTA

Afin de bâtir rapidement un prototype de l'architecture logicielle définie au chapitre précédent, nous avons utilisé la plate-forme JXTA[31, 59] qui offre les mécanismes de base pour la gestion de systèmes pair-à-pair. JXTA n'est pas une abréviation mais une contraction de "juxtapose" qui symbolise la juxtaposition du modèle pair-à-pair par rapport au modèle actuel client/serveur d'Internet, et non son opposition.

4.1.1 Motivations et objectifs de JXTA

Les objectifs de cet environnement générique permettant l'élaboration de services et d'applications pair-à-pair sont les suivants.

Factorisation de fonctionnalités identiques par découpage en couches modifiables indépendantes les unes des autres. En effet, les systèmes pair-à-pair utilisent souvent des protocoles différents mais qui, au final, ont les mêmes fonctionnalités, comme la découverte de ressources, la communication entre pairs, le gestion des groupes, etc. Ainsi, afin d'améliorer plus efficacement les fonctionnalités, celles-ci doivent être découpées en couches indépendantes les unes des autres.

Interopérabilité. Les différents systèmes élaborés sont pour l'instant incompatibles entre eux. Or, il serait souhaitable de pouvoir bénéficier de ponts d'échange entre ces systèmes, afin de tirer parti de leur originalité, ces systèmes ayant des objectifs différents. Le développement de services spécialisés basés sur un même environnement générique est nécessaire afin de rendre ces services interopérables.

Indépendance de la plate-forme. L'objectif d'un système pair-à-pair à grande échelle est d'être déployé sur un grand nombre de pairs. Il faut donc disposer d'un environnement générique indépendant de la plate-forme d'accueil du système. Une spécification précise et claire de l'ensemble des protocoles et de leurs formats, à respecter lors de l'implémentation de l'environnement, est nécessaire. Ainsi, le choix du langage ne joue pas sur l'interopérabilité des différentes implémentations existantes.

JXTA est un projet de recherche de Sun Microsystems qui a été rendu public afin de bénéficier des contributions de la communauté du pair-à-pair. JXTA est une spécification [60], en XML,

d'un ensemble de protocoles pair-à-pair génériques répondant aux objectifs précédemment énumérés.

4.1.2 Concepts de base

L'entité de base de JXTA est le pair. Un pair implémente un ou plusieurs protocoles JXTA, pas forcément tous, et désigne n'importe quel équipement disposant d'une connexion réseau comme par exemple un capteur, un téléphone portable, un PDA, un ordinateur, etc. Il existe différents types de pairs qui sont les suivants.

Pair de type *minimal edge*. Ces pairs ne peuvent qu'envoyer et recevoir des messages. Ce type de pair vise les équipements de type Pocket PC, téléphone mobile, etc. disposant de capacités limitées.

Pair de type *full-featured edge*. Par rapport au type de pair précédent, ce pair peut en plus découvrir et stocker des annonces de ressources présentes sur le réseau. C'est le type de pair classique qu'on trouve dans un système pair-à-pair bâti sur JXTA.

Pair de type *rendezvous*. En plus des fonctionnalités du type de pair précédent, ce pair peut propager les requêtes au sein du réseau JXTA, ainsi que mettre à disposition des autres pairs un espace de stockage des annonces de ressources disponibles sur le réseau.

Pair de type *relay*. Ce type de pair vise principalement à passer outre les pare-feux.

Un pair peut être membre d'un ou plusieurs groupes. Un groupe est une entité qui rassemble un ensemble de pairs qui ont un point commun, par exemple le fait de proposer un service de partage de données modifiables. La communication entre pairs est représentée par le concept de canal de communication appelé *pipe* : ce sont des canaux logiques de communications unidirectionnels, asynchrones et non fiables, permettant l'échange de messages entre deux pairs. Il existe également deux types de canaux : point à point et de propagation. Ces derniers permettent l'envoi d'un message d'un pair vers de multiples pairs.

Dans JXTA, il existe deux types de services : le service de pair et le service de groupe. L'avantage d'un service de groupe est qu'il est assuré par un ensemble de pairs et donc est insensible à la perte d'un pair composant le groupe. A l'opposé, la perte du pair qui héberge un service de pair entraîne l'indisponibilité du service. Enfin, toutes les ressources (pairs, groupes, pipes, services, etc.) disponibles sur un réseau JXTA sont représentées par une annonce. Un pair souhaitant utiliser une ressource doit donc au préalable découvrir son annonce. Pour cette recherche, JXTA fournit une DHT dite *loosely-consistent* [48] qui supporte une très grande volatilité des pairs sans avoir un coût de maintenance important contrairement à des systèmes comme Chord, Pastry ou Tapestry. Les techniques de localisation et de routage par répertoire centralisé et par inondation sont également possibles.

4.1.3 Architecture et protocoles

L'architecture de JXTA est découpée en trois couches.

1. La couche de base qui constitue le *noyau* de l'environnement. Cette couche implémente tous les concepts utilisés par JXTA et présentés à la section précédente : pairs, groupes, canaux, mais également la getion de la sécurité, etc.
2. La couche intermédiaire où les développeurs peuvent implémenter des *services*. On y retrouve les services proposés par JXTA, comme la découverte d'annonces, l'envoi de messages, etc. permettant l'accès aux concepts de base de la couche inférieure.

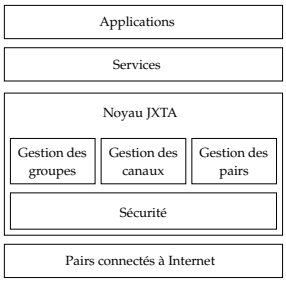

FIG. 4.1 – Architecture de JXTA.

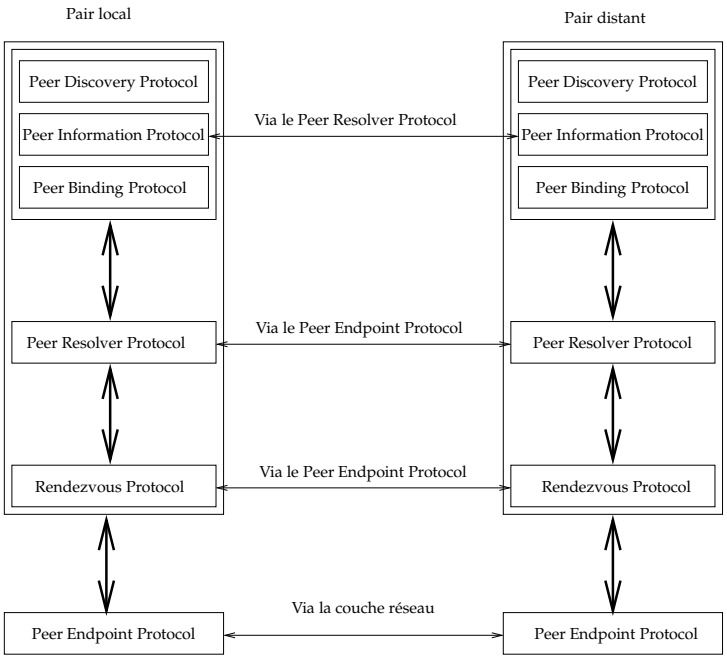

FIG. 4.2 – Couche des protocoles utilisés par JXTA.

3. La couche *applicative* qui s'appuie sur la couche précédente pour proposer à l'utilisateur une application rendant un ou plusieurs services.

JXTA définit un ensemble de protocoles génériques présentés sur la figure 4.2. Cette figure montre également les liens, sous la forme de couches, entre les différents protocoles. Les rôles de ces protocoles sont les suivants.

Le *Endpoint Routing Protocol* (ERP) est une abstraction des différents protocoles de communications utilisables sur le pair physique comme TCP, HTTP, etc.

Le *Rendezvous Protocol* (RP) permet au pair de type *edge* (*minimal* et *full-featured*) de se connecter à un pair de rendezvous. Il permet également aux pairs de type *rendezvous* de s'organiser.

Le *Peer Resolver Protocol* (PRP) permet d'envoyer et de recevoir des requêtes et des réponses génériques sur le réseau. C'est le protocole central dans la couche des proto-

coles JXTA.

Le *Peer Discovery Protocol* (PDP) permet à un pair de découvrir des ressources dans le système, que ce soit des pairs, des groupes, des services, etc.

Le *Peer Information Protocol* (PIP) permet à un pair d'obtenir des informations sur le statut d'autres pairs, comme la charge, le trafic réseau, le temps de connexion, etc.

Le *Pipe Binding Protocol* (PBP) permet à un pair d'établir une connexion physique avec un ou plusieurs autres pairs attachés à un canal de communication donné.

Un certain nombre de services et d'applications ont été bâtis à partir de cet environnement. Le site de JXTA [59] référencie une partie de ces développements.

4.2 Mise en œuvre de JuxMem

L'objectif de cette section est de donner une présentation générale de l'implémentation de JuxMem et de décrire l'utilisation qui a été faite de JXTA pour cette mise en œuvre.

4.2.1 Présentation générale

L'implémentation de JuxMem présentée ici repose sur la version 2.0 des spécifications de JXTA. L'implémentation choisie de JXTA est la version Java, car c'est la seule implémentation de la version 2.0 des spécifications de JXTA. JuxMem est bâti comme un service de groupe dans l'architecture de JXTA et peut donc être utilisé par d'autres services ou applications JXTA. JuxMem est écrit en Java et représente plus de 5 000 lignes de code réparties en 48 classes Java. JuxMem est organisé en différents paquetages visant chacun les différents types de pairs.

Le paquetage `core` qui correspond au *noyau* du service implémenté et est chargé d'initialiser les différents type de pairs pour un même pair physique.

Le paquetage `client` qui correspond au code des différentes opérations qu'un *pair client* peut réaliser : allocation, lecture et écriture d'un bloc de données, etc.

Le paquetage `rdv` qui correspond au code d'un *pair gestionnaire* de grappe, qui reposent sur des pairs JXTA de type *rendezvous*. Il contient les sous-paquetages `cluster` et `juxmem` responsables de traiter respectivement les messages au sein des groupes `cluster` et `juxmem`.

Le paquetage `provider` qui correspond au code d'un *pair fournisseur*. Il contient les sous-paquetages `memory`, `data` et `cluster` responsables de gérer respectivement la mémoire du pair fournisseur, une copie d'un bloc de données et les messages reçus au sein du groupe `cluster`.

En outre, nous avons conçu un outil graphique appelé JuxMemView (voir figure 4.3), dérivé de l'application JxtaView [61], qui permet de visualiser les différents pairs et groupes présents dans le service, mais également d'effectuer toutes les opérations de l'API de JuxMem. Il permet en outre de tuer des pairs afin d'en étudier les effets sur le système.

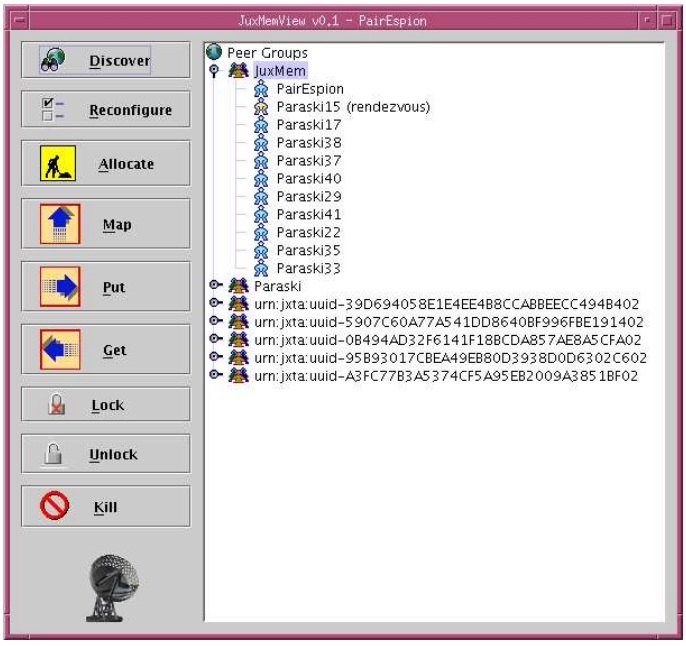

FIG. 4.3 – JuxMemView, un outil graphique pour la plate-forme JuxMem.

4.2.2 Pairs, groupes et canaux

JXTA répond pleinement aux besoins de JuxMem pour sa mise au point. Ainsi les pairs responsables des groupes de type `data` et `cluster` sont des pairs JXTA de type *rendez-vous*. En effet, il est nécessaire pour ces pairs de stocker les annonces de type fournisseur au sein des groupes de type `cluster` et `data` pour respectivement gérer une grappe et un bloc de données. De plus, seul les pairs JXTA de type *rendezvous* peuvent transmettre les requêtes dans un réseau JXTA, cela correspond à l'un des rôles du responsable principal d'une grappe. Les pairs clients et les pairs fournisseurs qui ne sont pas responsables pour un ou plusieurs blocs de données sont des pairs JXTA de type *full-featured edge*. En effet, ils n'ont aucun rôle à jouer dans le contrôle dynamique du nombre de copies dans le système. Les différents groupes définis dans JuxMem sont implémentés par des groupes JXTA. Le groupe `juxmem` implémente un service de groupe constitué par l'API de JuxMem (voir section 4.2.1).

Les canaux de communications de JXTA offrent également le support attendu pour l'implémentation de communications point à point mais également de type multicast par l'utilisation des pipes de propagation pour la mise à jour en parallèle des différentes copies d'un même bloc de données. Le choix des canaux de communications et donc du protocole PBP pour la résolution des canaux par rapport au protocole PRP (voir figure 4.2) est justifié par le niveau d'abstraction supplémentaire offert par le protocole PBP. En effet, le développeur n'a pas à spécifier d'adresse physique pour l'envoi ou la réception de messages, mais simplement un identifiant de canal. C'est le protocole PBP qui se charge de déterminer dynamiquement les pairs connectés sur le canal en mode écoute. De plus, ceci a permis de s'abstraire des modifications subies par le protocole PRP lors du passage de JXTA 1.0 à JXTA 2.0. Ainsi, lors de ce passage une seule demi-journée a été nécessaire pour le portage du service.

4.2.3 Un exemple de gestion de la volatilité

La volatilité des pairs fournisseurs hébergeant des blocs de données est gérée au sein du sous-paquetage `data` du paquetage fournisseur. Sur les responsables d'un groupe `data`, un thread nommé `DataManagerThread` est chargé de vérifier périodiquement le nombre de pairs fournisseurs qui hébergent une copie du bloc de données et la disponibilité des responsables pour ce bloc de données. Les deux responsables doivent chacun vérifier tous les pairs fournisseurs pour les mêmes raisons qu'indiquées à la section 3.4.2. Le diagramme d'action simplifié des différentes opérations réalisées par ce thread sur l'un des responsables, qu'il soit principal ou secondaire, d'un groupe de type `data` est présenté sur la figure 4.4. Sur cette figure, `replication` représente le niveau de réplication spécifié par le pair client, et `n` le nombre de pairs fournisseurs qui hébergent une copie de ce bloc de données.

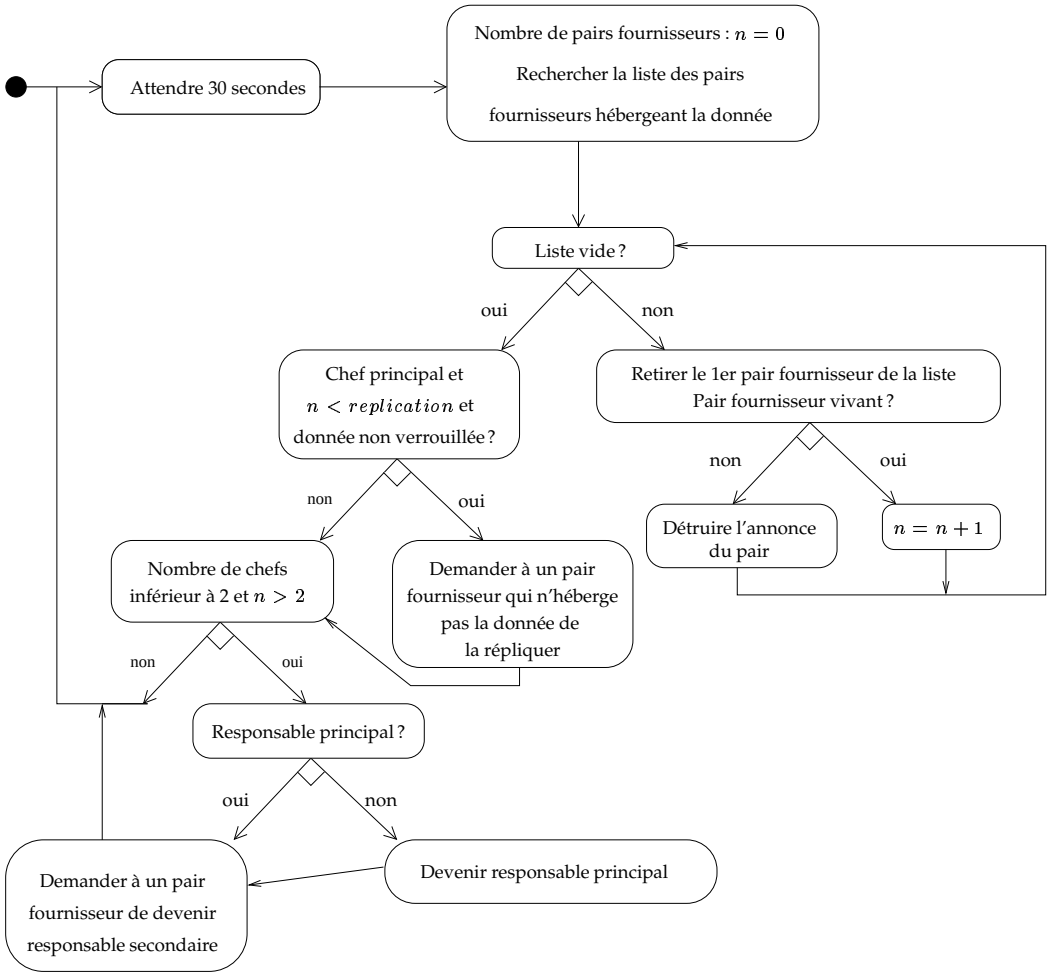

FIG. 4.4 – Diagramme d'activité simplifié pour le thread DataManagerThread exécuté sur un des responsables d'un groupe de type `data`.

4.2.4 Un exemple d'utilisation des blocs de données

La séquence typique d'opérations pour la manipulation d'un bloc de données est :

```
lock(id)
put(id,value)
unlock(id)
```

Cette séquence fait intervenir un pair client et les pairs fournisseurs hébergeant ce bloc de données, regroupés au sein d'une instance d'un groupe de type `data`. Les messages échangés entre les différents acteurs sont représentés à la figure 4.5.

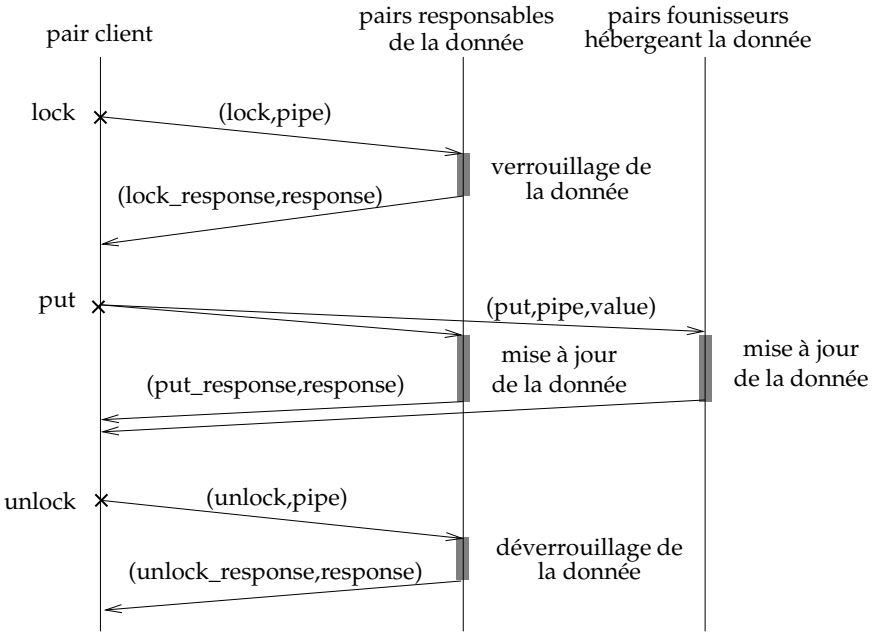

FIG. 4.5 – Messages échangés pour la séquence d'opérations suivante : *lock*, *put* et *unlock*.

Trois types de messages sont utilisés par le pair client pour réaliser ces opérations. Tous ces messages transitent par un pipe de propagation depuis le groupe `juxmem` vers le groupe `data`. Toutefois, les messages de type *lock* et *unlock* ne sont traités que par les responsables du groupe `data`. Le premier message envoyé par le pair client est de type *lock* et contient pour champs :

1. un champ `msg_tag` avec pour valeur `lock` permettant d'identifier le type du message ;
2. un champ `msg_pipe` avec pour valeur l'annonce du pipe point à point sur lxequel écoute le pair client qui a émis la requête.

Lors de la réception de ce message, les pairs fournisseurs hébergeant un bloc de données testent s'ils sont responsables. Les responsables traitent la requête mais seul le responsable principal retourne une réponse au pair client en résolvant le pipe point à point du pair client à l'aide du protocole PBP de JXTA.

Le deuxième message envoyé par le pair client est de type *put* et contient les champs suivants :

1. un champ `msg_tag` avec pour valeur `put` permettant d'identifier le type du message ;
2. un champ `msg_pipe` avec pour valeur l'annonce du pipe point à point sur lequel écoute le pair client qui a émis la requête ;
3. un champ `msg_value` avec pour valeur la nouvelle valeur du bloc de données.

Ce message est traité par tous les pairs fournisseurs de la même manière, responsable ou non, c'est à dire par la mise à jour du bloc de données avec la nouvelle valeur spécifiée par le champ `msg_value`.

Le troisième message envoyé par le pair client est de type *unlock* et contient les champs suivants :

1. un champ `msg_tag` avec pour valeur `unlock` permettant d'identifier le type du message ;
2. un champ `msg_pipe` avec pour valeur l'annonce du pipe point à point sur lequel écoute le pair client qui a émis la requête.

De façon similaire au message *lock*, ce message n'est traité que par les responsables du groupe `data` mais la réponse n'est renvoyée que par le responsable principal. Tous ces messages sont bloquants, les pairs clients attendent en effet la confirmation que l'opération a bien été réalisée. Une amélioration envisageable est de rendre les messages *put* et *unlock* non bloquants. L'amélioration des performances est obtenue au prix de la non-garantie que l'opération a été réalisée. En effet, si une partition réseau s'opère suite à un problème technique sur un lien réseau reliant la grappe A au reste de la grille, l'opération *unlock*, réalisée depuis un pair client de cette même grappe A n'aboutira pas. Or, le pair client considérera avoir ôté son verrou du bloc de données, pourtant celle-ci serait verrouillée à jamais et de ce fait non accessible par d'autres pairs clients.

4.2.5 Déploiement

Un pair peut dynamiquement charger un service, par exemple JuxMem, en téléchargeant le code du service depuis une adresse spécifiée par l'annonce du service. Ce code est en général encapsulé dans un fichier *.jar*, qui est un assemblage de classes Java. Cette fonctionnalité est en cours d'implémentation et sera disponible dans une prochaine version de JuxMem. Toutefois, JXTA n'offre aucun mécanisme de déploiement d'un service. Ce déploiement reste donc à la charge des différents administrateurs des sites qui forment la grille de calcul. Un tel mécanisme serait toutefois souhaitable afin de faciliter l'utilisation de services JXTA, notamment lors de la phase de développement et de test d'un service. Lors de ces phases et dans le cadre de ce stage, le déploiement de JuxMem a donc été réalisé à la main sur une configuration allant jusqu'à vingt machines.

Chapitre 5

Évaluation de JuxMem sur JXTA

5.1 Environnement de développement

Le développement de JuxMem a été réalisé sur la version Java de JXTA 2.0. La machine virtuelle utilisée a été celle de Sun Microsystems en version 1.4.1_01-b01 disponible à l'IRISA. La mise au point d'un système pair-à-pair est un travail délicat. Peu d'outils utilisables sont disponibles pour déboguer de tels systèmes. Toutefois, l'utilisation de la librairie *log4j* m'a permis d'accélérer le développement de JuxMem. *log4j* est un outil permettant de journaliser des informations et de tracer une exécution. Le but de *log4j* est d'éviter, comme c'est souvent le cas, au développeur d'écrire une API de déboguage pour l'application qu'il développe. De plus, cette librairie est bien plus complète qu'une implémentation "maison" et permet par exemple d'enregistrer les informations selon différents niveaux, pour une classe, un paquetage ou une application, de visualiser ces informations sur une console ou à distance. De formater la sortie des erreurs et d'insérer des informations sur le temps, la classe ou la ligne à laquelle a été journalisée l'information. Enfin, cette librairie vise également la performance afin de réduire au minimum le surcoût engendré par l'ajout d'instructions dans le code.

La mise au point de scénarios est également un problème dans de tels systèmes. Toutefois, l'utilisation de *JUnit*, une librairie permettant d'effectuer des tests unitaires et de non régression a permis d'alléger ce processus, sans pour autant entièrement l'automatiser. En effet, les contraintes liées à la réalisation de tests à grande échelle sont un obstacle majeur pour le développement et la mise au point de systèmes pair-à-pair. Preuve en est, le lancement au sein de la communauté JXTA d'un projet appelé JXTA Distributed Framework visant à construire un environnement permettant une automatisation des tests [56].

La démarche adoptée pour la mise au point a été tout d'abord de tester et de déployer différents types de pairs connectés entre eux sur une même machine physique. Puis de déployer ces mêmes pairs sur la grappe « `paraski` » du projet Paris tout en augmentant le nombres de pairs afin d'effectuer des mesures à plus grande échelle. Chaque nœud de la grappe héberge un unique pair. Les nœuds sont des bi-processeurs[1] Pentium2 cadencés à 450 MHz, munis de 256 MB de mémoire vive (RAM). Toutefois, les différents liens réseaux reliant les nœuds de la grappe n'ont pas été utilisés lors de l'évaluation. Celle-ci s'est faite en utilisant le réseau Ethernet 100 Mb/s reliant les nœuds. La raison principale est d'une part que JXTA ne gère pas des liens de type SCI ou Myrinet et que d'autre part l'efficacité ne constitue pas un des objectifs premiers de cette évaluation. En effet, l'objectif principal de

[1] Sur les 2 processeurs de chaque nœud, un seul a été utilisée.

cette évaluation est d'étudier la validité de l'architecture proposée : il s'agit essentiellement d'une étude de faisabilité par la construction d'un prototype.

5.2 Consommation mémoire des différentes entités

La première mesure effectuée étudie le surcoût engendré en terme de consommation mémoire des différents types de pairs de JuxMem par rapport aux pairs JXTA sous-jacents : un pair client JuxMem a ainsi été comparé à un pair JXTA de type *full-featured edge*, un pair gestionnaire à un pair JXTA de type *rendezvous*, etc.

5.2.1 Pair fournisseur

La première mesure réalisée sur un pair fournisseur est une mesure du surcoût de la consommation mémoire sur un pair n'hébergeant pas de données par rapport à un pair JXTA de type *full-featured edge*. Cette mesure a été réalisée une fois les opérations d'initialisation du pair terminées, sur un pair fournisseur isolé et également sur un pair faisant parti d'une grappe. Le surcoût de la consommation mémoire est de 7 % environ par rapport à un pair de type *full-featured edge*. Le coût des fonctionnalités supplémentaires nécessaires pour offrir une zone mémoire, susceptible d'héberger une donnée, dans une grappe est donc raisonnable.

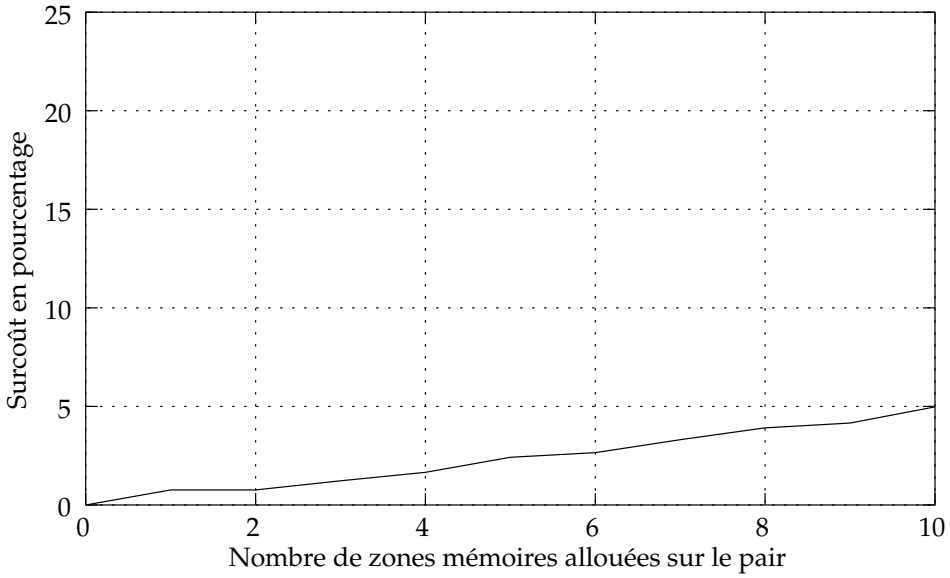

FIG. 5.1 – Surcoût mémoire sur un pair fournisseur selon le nombre de données hébergés sur ce pair.

Une seconde mesure réalisée a été d'étudier l'évolution du surcoût de la consommation mémoire sur le pair en fonction du nombre de données hébergées par rapport à un pair JXTA de type *rendezvous*. En effet, pour chaque groupe `data` associé à une donnée la mesure a été effectuée sur un nœud qui a joué le rôle de responsable. Or, un pair responsable d'un groupe `data` correspond à un pair JXTA de type *rendezvous*. Pour chaque mesure le nœud est le seul

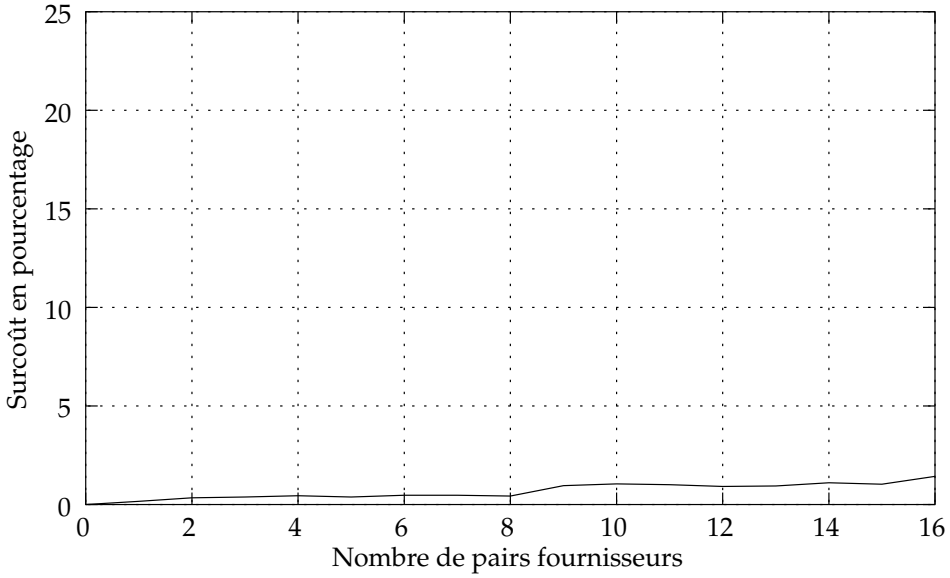

FIG. 5.2 – Surcoût mémoire sur un pair gestionnaire selon le nombre de pairs fournisseurs présents dans la grappe.

pair fournisseur de la grappe. La figure 5.1 présente les résultats obtenus. Il est à noter que le pair sur lequel a été faite la mesure joue le rôle de responsable principal pour toutes les données et que le niveau de réplication a été fixé à 1, c'est-à-dire qu'il y a une unique copie de la donnée dans le système. Aucune opération n'a été réalisée sur les données durant la mesure. De plus, afin de mesurer seulement l'impact du nombre de données, toutes les données introduites sur le pair sont de taille minimale (1 octet). Le surcoût de la consommation mémoire sur un pair fournisseur selon le nombre de données hébergées par rapport à un pair de type *rendezvous* est quasi-linéaire. Cela est normal puisque pour chaque nouvelle donnée hébergée est instancié un même nombre de classes pour en assurer la gestion. De plus, ce surcoût est très raisonnable, ne dépassant pas 5 % pour 10 données hébergées sur le pair, ce qui autorise le support d'un nombre important de données sur un pair fournisseur.

5.2.2 Pair gestionnaire

La première mesure possible sur un pair gestionnaire est la mesure du surcoût de la consommation mémoire par rapport au pair JXTA sous-jacent de type *rendezvous*. La première mesure a été réalisée une fois les opérations d'initialisation du pair terminées, sur un pair gestionnaire isolé. Le surcoût de la consommation mémoire est de 5 %. Le coût des fonctionnalités supplémentaires nécessaires pour la gestion d'une grappe de pair fournisseur est donc raisonnable.

Une seconde mesure réalisée a été d'étudier l'évolution du surcoût en consommation mémoire en fonction du nombre de pairs fournisseurs dont le pair gestionnaire est responsable. Cette mesure a été réalisée sur un pair gestionnaire sur lequel on a désactivé la fonctionnalité de réplication automatique du rôle de responsable (section 3.4.1) afin de cerner seulement ce surcoût. La figure 5.2 présente les résultats obtenus. Aucune donnée n'a été introduite dans le système. Le surcoût maximal est inférieur à 2 %. Ce surcoût est très faible, un pair gestion-

naire peut donc être responsable d'un nombre important de pairs fournisseurs, c'est-à-dire de nœuds dans une grappe.

5.2.3 Pair client

Sur les pairs clients, nous avons évalué le surcoût en terme de consommation mémoire par rapport aux pairs JXTA de type *full-featured edge*. La mesure a été réalisée une fois les opérations d'initialisation du pair terminées, sur un pair client isolé. Le surcoût de consommation mémoire est de 5 % environ. Le coût des fonctionnalités supplémentaires nécessaires pour utiliser la plate-forme JuxMem est, dans ce cas, également raisonnable.

5.3 Perturbation introduite par la volatilité des pairs fournisseurs

5.3.1 Méthodologie des mesures

La mesure réalisée consiste à étudier l'influence du degré de volatilité des pairs fournisseurs sur le temps d'une séquence de `lock-put-unlock` effectuée sur une données par un pair client. Le programme de test consiste à allouer sur une grappe, constituée de 16 pairs fournisseurs et d'un pair gestionnaire, une donnée avec un niveau de réplication de 3. Le pair client réalise une boucle de 100 itérations `lock-put-unlock`. Pendant l'exécution de cette boucle, on tue aléatoirement un pair fournisseur toutes les δ secondes, δ étant le paramètre de la mesure. En effet, chaque pair fournisseur qui accueille une copie de la donnée envoie son annonce à un pair espion. Ce dernier est alors en mesure d'envoyer un message de terminaison immédiate du processus. Un pair fournisseur recevant un tel message s'arrête brutalement, simulant une faute de type crash. Afin de ne mesurer que le surcoût dû à la volatilité des pairs fournisseurs, le pair fournisseur responsable de la donnée n'est jamais tué (les mécanismes de réplication automatique du rôle de responsable étant donc inutiles, ils ont été désactivés). On ne dispose donc que de deux pairs fournisseurs hébergeant une copie de la donnée à "tuer". Il est à noter que la mesure ne s'effectue pas à nombre de pairs constant.

L'objectif de cette mesure est d'évaluer le surcoût, en pourcentage, engendré par les réplications nécessaires au maintien d'un degré de réplication pour la donnée en présence de perte de pairs fournisseurs qui hébergent une copie de la donnée, par rapport à un système stable, c'est-à-dire sans perte de pairs.

5.3.2 Analyse des résultats et discussion

La figure 5.3 présente le surcoût en pourcentage par rapport à un système stable ($\delta = \infty$), c'est-à-dire dans lequel tous les pairs restent connectés durant l'exécution de la boucle. Les tests ont été réalisés avec δ variant de 160 secondes à 30 secondes, par pas de 20 jusqu'à une de valeur de δ égale à 60 puis par pas de 10.

Lorsque le pair responsable de la donnée détecte la disparation d'une copie de la donnée, il cherche à la répliquer sur d'autres pairs fournisseurs. Or, pendant la réplication, le système doit verrouiller la donnée de manière interne : un pair client ne doit pas pouvoir modifier la donnée pendant qu'un pair fournisseur en instancie une nouvelle copie. Ce verrouillage interne entraîne une augmentation de la durée de la séquence `lock-put-unlock` puisque le pair client est alors bloqué en attente de libération du verrou (section 3.3.2). L'évolution de la

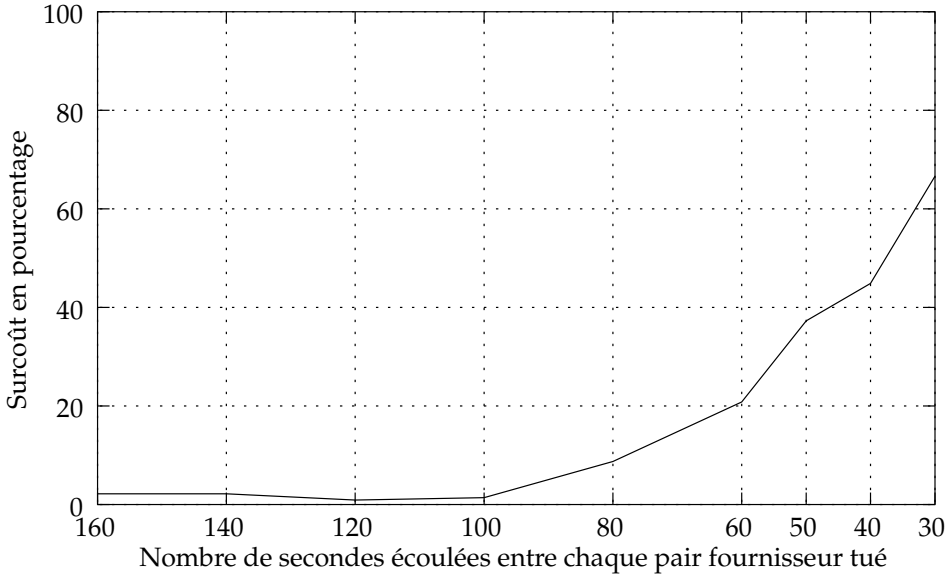

FIG. 5.3 – Perturbation en pourcentage introduite par la volatilité des pairs fournisseur pour une séquence `lock-put-unlock` par rapport à un système stable.

courbe s'explique par la moyenne du nombre de fois que le système réplique la donnée sur des pairs fournisseurs afin de maintenir le nombre de copies souhaité par le pair client, en l'occurrence 3 pour ce test. Pour la durée totale de notre test, cette moyenne est donné dans le tableau suivant.

Secondes	160	140	120	100	80	60	50	40	30
Nombre de réplications déclenchées	1	1	1	1	2	2,5	5	5,5	10

Au-delà de 2 réplications déclenchées ($\delta < 80$ s), le surcoût devient important. Pour $\delta = 30$ s, il atteint plus de 65 % (dix réplications déclenchées). Cependant, dans une situation réaliste, la volatilité des nœuds pour l'architecture physique visée est plus faible ($\delta > 80$ s). Pour de telles valeurs, le surcoût de la reconfiguration du système est très faible, inférieur à 5 %. La plate-forme JuxMem proposé dispose donc d'un mécanisme qui permet de maintenir *dynamiquement* le nombre de copies d'une donnée, afin de maximiser sa disponibilité et autorisant la perte de pairs, *sans un surcoût important* pour ce maintien.

Chapitre 6

Conclusion

Les grilles de calculs sont utilisées pour des simulations numériques de plusieurs heures ou jours, en utilisant les ressources lourdes de plusieurs centres de calcul d'un pays. L'un des principaux apports des environnements de calcul sur grille développés jusqu'à présent est d'avoir *découplé les calculs, des traitements de déploiement nécessaires pour ces calculs*. Ces calculs sur grille génèrent des volumes important de données à partager entre plusieurs sites. Un tel besoin nécessite l'utilisation d'un système de gestion de données sur grille. Or, aucun système qui prenne en compte les propriétés souhaités et les contraintes d'une grille de calcul n'a été élaboré. Ces propriétés sont la persistance et la transparence de la localisation des données, et les contraintes sont l'extensibilité de l'architecture, la cohérence des données partagées et la volatilité des ressources. Ce rapport a défini l'architecture d'un tel service de partage de données modifiables.

6.1 Contributions

6.1.1 Architecture hiérarchique

Ce rapport définit une architecture *hiérarchique* de service de partage de données modifiables pour une grille constituée d'une fédération de grappes. Cette architecture est construite selon une approche pair-à-pair. Elle permet de réduire le nombre de messages pour rechercher une donnée, mais aussi de tirer parti des caractéristiques de l'architecture physique sous-jacente. La politique de gestion d'une grappe peut donc être spécifique à sa configuration, afin d'exploiter au mieux ses capacités notamment en terme de liens disponibles entre les nœuds. L'extensibilité du système est également accrue, puisqu'au niveau supérieur de la hiérarchie une grappe de machines est représentée par une unique entité : le groupe de pairs associé. D'autres systèmes pair-à-pair hiérarchiques ont été proposés très récemment [44, 29, 24]. Ces systèmes définissent au premier niveau de la hiérarchie un réseau de *super-pairs* organisés grâce à une DHT. Au deuxième niveau de la hiérarchie, chaque super-pair est responsable vis-à-vis du niveau supérieur d'un ensemble de pairs organisés eux aussi grâce à une DHT. Grâce à cette approche il est possible de diminuer le nombre de messages pour rechercher une donnée.

6.1.2 Stockage et accès transparent aux blocs de données

JuxMem permet l'allocation de zones mémoires au sein d'une fédération de grappes. Au niveau applicatif, la création de telles zones se traduit par la récupération d'un identifiant. Ainsi, *la localisation et le transfert de données sont transparents au niveau applicatif* puisqu'il suffit de spécifier un identifiant de zone mémoire pour accéder et manipuler le bloc de données contenu dans cette zone mémoire. Ces identifiants étant dans un même espace global, des lectures et des écritures par différents clients sont possibles. Ceci est un progrès significatif dans les systèmes de gestion de données pour la grille, car il est possible de découpler les applications de calculs de la gestion des données associées. La modification des blocs de données est possible par l'utilisation de verrous sur les blocs. Il est à noter que contrairement à des systèmes comme CFS ou PAST qui déterminent l'identifiant de la donnée, et donc le pair responsable de celle-ci, à partir de la valeur de la donnée, l'identifiant utilisé dans JuxMem correspond à une zone mémoire dans le système. Ainsi, lors de la modification d'un bloc de données aucun nouveau calcul pour déterminer le pair responsable de la zone mémoire modifiée n'est effectué. Enfin, les blocs de données n'étant pas liés aux clients, ils sont stockés de manière persistante dans le systèmes.

6.1.3 Support de la volatilité des pairs

L'architecture définie supporte la volatilité de tout type de pairs. Bien que cette volatilité soit évidemment supportée dans les systèmes pair-à-pair comme Gnutella ou KaZaA, celle-ci n'est pas dynamiquement prise en compte pour la gestion des données introduites dans de tels systèmes. En effet, la réplication des données est assurée par les différents utilisateurs de tels systèmes, mais n'offre aucune garantie vis-à-vis de la persistance des données, notamment celles qui sont peu populaires. Le système défini ici prend *activement en compte cette volatilité pour assurer la persistance* et donc la disponibilité des données. De plus, la volatilité des pairs responsables d'une grappe est transparente pour les différents clients. La disponibilité d'une grappe est donc maximisée afin de minimiser les perturbations au niveau des clients.

6.2 Problèmes ouverts

6.2.1 Gestion de la sécurité

La gestion de la sécurité est un domaine actuellement en voie d'exploration dans les systèmes pair-à-pair [13, 46]. C'est un problème important dans les systèmes de gestion de données. En effet, les simulations numériques utilisatrices de grilles de calculs sont notamment réalisées par des industriels. Or, les grilles de calculs étant la mise en commun de ressources par différentes entités, une garantie de confidentialité des données est nécessaire. Cependant, la notion de groupe introduite dans JuxMem permet d'introduire de manière plus aisée des notions de droits d'accès aux données. Le créateur d'une donnée pourrait ainsi spécifier les critères d'accès aux données et par exemple interdire l'accès à certains clients. Toutefois, un utilisateur autorisé peut également avoir un comportement malveillant, comme par exemple demander l'allocation de zones mémoire dont il n'a pas besoin, privant ainsi d'autres utilisateurs des services rendus par le système. Pour résoudre ce problème,

l'utilisation de quotas est envisageable pour l'accès aux ressources proposées au sein du service. Le problème de la gestion de la sécurité n'a ni été abordé ni traité dans ce rapport en raison de son étendue et de sa complexité, les exemples mentionnés précédemment n'étant évidemment pas une liste exhaustive des problèmes à traiter dans ce domaine.

6.2.2 Tolérance aux fautes

JuxMem supporte la volatilité des pairs qui peut être considérée comme une faute de type crash. Toutefois, le service ne supporte pas les fautes par omission et les fautes byzantines. En effet, le service bâti suppose que les canaux de communications sont fiables mais JXTA n'offre aucune garantie sur la livraison du message à son destinataire. Ainsi, si un pair hébergeant un bloc de données ne reçoit pas un message de modification du bloc de données, la copie de ce bloc de données peut être incohérente avec les autres copies. Or, lorsqu'un client cherche à récupérer un bloc de données, tous les pairs qui hébergent une copie de ce bloc de données répondent en parallèle. Le client prenant la première réponse afin de favoriser la localité physique, il est donc possible qu'il ne reçoive pas la dernière valeur pour le bloc de données. Malgré tout, sur l'environnement de test aucune perte de message n'a été constatée en raison de l'isolement de la grappe sur laquelle les mesures ont été effectuées. La mise à disposition prochaine de canaux de communications fiables dans JXTA permettra de résoudre simplement ce problème. Concernant les fautes byzantines, différents travaux sur ce domaine sont en cours de réalisation [46]. Ainsi, un pair fournisseur peut prétendre avoir alloué une zone mémoire et rendre l'identifiant de cette zone au client, sans toutefois l'avoir réellement fait. Le client peut ainsi stocker des blocs de données dans une zone mémoire non réellement disponible et donc perdre ces mêmes blocs de données. L'hypothèse d'un pair byzantin n'a pas été prise en compte pour la conception de JuxMem et n'a donc fait l'objet d'aucune étude dans ce rapport.

6.2.3 Dépendance envers JXTA et Java

L'objectif principal de la conception et de la réalisation d'un prototype de service de partage de données modifiables sur une infrastructure pair-à-pair est de montrer la faisabilité d'un tel système. Pour l'instant, l'efficacité ne constitue donc pas l'un des objectifs principaux visés. Toutefois, ni l'utilisation de JXTA ni celle de Java pour implémenter JuxMem ne sont un obstacle pour l'amélioration des performances. En effet, l'utilisation JXTA concerne principalement les phases de découverte des différentes ressources disponibles dans le service. Il est de ce fait tout à fait possible d'utiliser d'autres systèmes pour communiquer plus efficacement une fois cette découverte effectuée. De plus, la conception de JuxMem n'est pas dépendante de JXTA, il est donc possible d'utiliser d'autres librairies comme JavaGroups [55]. Enfin, l'architecture modulaire de JXTA permet d'ajouter et d'enlever des services notamment au niveau des protocoles réseaux utilisés. Concernant le choix de Java comme langage d'implémentation, celui-ci n'est aucunement dicté par l'utilisation de JXTA. En effet, il existe des implémentations de JXTA dans différents langages. Ces implémentations sont certes moins avancées que la version Java, mais un effort important de mise à jour sur l'implémentation C/C++ de JXTA est en cours.

6.2.4 Gestion de la cohérence et de la synchronisation

Les blocs de données n'étant pas cachées par les clients, la gestion de la cohérence se situe au niveau des pairs qui hébergent des copies d'un même bloc de donnée. Cette gestion est assurée par la mise à jour simultanée de toutes les copies d'un bloc de données. Ce mécanisme a l'avantage d'être simple, mais en contrepartie il est coûteux en nombre de messages émis et en bande passante consommée. Un relâchement de ce modèle de cohérence stricte et des mécanismes de synchronisation associés est envisageable afin d'améliorer l'efficacité du système. Les différents modèles et protocoles de cohérence développés dans le cadre de la MVP sont à étudier, notamment ceux qui prennent en compte l'architecture physique sous-jacente [5]. De même, les clients ne sont pas informés de modifications sur la totalité ou sur des parties d'un bloc de données. Un mécanisme de notification des clients selon un niveau de mise à jour auquel ils se seraient au préalable enregistrés est envisageable. La gestion de la synchronisation entre différents clients s'effectue par l'utilisation d'un verrou sur la totalité de la zone mémoire allouée. Le grain du partage d'une zone mémoire pourrait donc être affiné et surtout être paramétrable par les clients afin d'augmenter le nombre d'écritures simultanées sur un même bloc de données.

6.3 Perspectives

6.3.1 Mobilité transparente des blocs de données

Une voie de recherche possible concerne la mobilité transparente des blocs de données. Actuellement, les blocs de données sont dynamiquement répliqués au sein d'une grappe. Le choix de la grappe pour l'allocation d'une zone mémoire est transparent au niveau applicatif. Toutefois, ce choix est capital puisque les blocs de données stockés dans cette zone mémoire ne peuvent pas migrer d'une grappe à une autre. Cette migration est intéressante sur de nombreux points.

- Le plus évident concerne l'incapacité d'une grappe à satisfaire le niveau de réplication désiré par les clients en raison soit de la perte d'un grand nombre de nœuds sur la grappe, soit de l'arrêt prochain de cette même grappe.
- Un autre intérêt est de rapprocher les blocs de données des clients les utilisant afin de minimiser les temps d'accès lors de requêtes ultérieures. Ainsi, les blocs de données seraient présents sur les grappes les plus proches des clients les utilisant.
- Enfin, il serait également intéressant d'étudier les affinités entre les blocs de données eux-mêmes afin de placer par exemple celles-ci, si ils sont liés, sur une même grappe toujours dans le but de minimiser les temps d'accès. Une interface au niveau client pourrait être utilisée afin d'indiquer les relations entre ces blocs de données.

La mobilité transparente des blocs de données au niveau applicatif ouvre donc de nombreux problèmes intéressants à étudier.

6.3.2 Cohérence paramétrable

Les clients ne stockant pas une copie du bloc de données, le problème de la cohérence se situe entre les différents pairs hébergeant un bloc de données. Cependant, un mécanisme de notification du client pourrait être intéressant afin d'informer des clients de mises à jour d'un

bloc de données selon un certain niveau de cohérence auquel il se serait préalablement enregistré. De plus, la granularité de partage des blocs de données est actuellement un obstacle à un partage efficace. Toutefois, la diminution de ce niveau de granularité risque de poser des problèmes de cohérence entre les différentes copies, notamment si les blocs de données peuvent migrer et donc être répartis sur différentes grappes. Sur ce type de configuration le relâchement de la cohérence des copies présentes *entre* chaque grappe, et des mécanismes adéquats pour un contrôle de ce relâchement est intéressant à étudier.

6.3.3 Système de synchronisation hiérarchique

L'utilisation de verrous, quelle que soit la granularité du partage, pour l'accès aux blocs de données pose un problème de contention surtout si on prend en compte les nouvelles voies de recherches mentionnées précédemment. Ainsi, si des copies d'un même bloc de données sont localisées sur différentes grappes, le chef du bloc de données reste unique et donc sur une seule grappe. Cela pose des problèmes de performances lorsque plusieurs clients distants souhaitent modifier la même zone mémoire. L'intégration de mécanismes privilégiant les accès proches, ou mieux les accès au sein de la grappe hébergeant le chef du bloc de données sont à étudier. Dans ce cadre, un système de synchronisation hiérarchique [19] afin de tirer parti de l'architecture physique est intéressant à étudier.

Bibliographie

[1] *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, number 2429 in Lecture Notes in Computer Science, Cambridge, MA, USA, March 2002. Springer.

[2] Bill Allcock, Joseph Bester, John Bresnahan, Ann Chervenak, Lee Liming, Samuel Meder, and Steven Tuecke. *GridFTP Protocol Specification*. GGF GridFTP Working Group Document, September 2002.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] Kelsey Anderson. Analysis of the traffic on the Gnutella network. `http://www-cse.ucsd.edu/classes/wi01/cse222/projects/reports/p2p-2.pdf`, March 2001.

[5] Gabriel Antoniu, Luc Bougé, and Sébastien Lacour. Making a DSM consistency protocol hierarchy-aware: An efficient synchronization scheme. In *3rd IEEE/ACM International Conference on Cluster Computing an the Grid (CCGrid2003)*, pages 516–523, Tokyo, Japan, May 2003. IEEE.

[6] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James Plank, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: A study in resource sharing. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 194–201, Berlin, Germany, May 2002. IEEE.

[7] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In ACM Press, editor, *6th Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 77–88, Atlanta, GA, May 1999.

[8] Sherif Botros and Steve Waterhouse. Search in JXTA and other distributed networks. In *1st International Conference on Peer-to-Peer Computing (P2P '01)*, pages 30–35, Lingköping, Sweden, August 2001.

[9] Daniel Brookshier, Darren Govoni, and Navaneeth Krishnan. *JXTA: Java P2P Programming*. Sams Publishing, March 2002.

[10] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. In B. Monien and R. Feldmann, editors, *Proceedings of the 8th International Euro-Par Conference*, volume

2400 of *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, August 2002. Springer-Verlag.

[11] Nicholas Carriero, David Gelernter, Timothy Mattson, and Andrew Sherman. The linda alternative to message-passing systems. *Parallel Computing*, 10(4):633–655, March 1994.

[12] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 152–164, Pacific Grove, CA, October 1991.

[13] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, December 2002.

[14] Steve Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw. The legion resource management system. In *5th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP '99)*, volume 1659 of *Lecture Notes in Computer Science*, pages 162–178, San Juan, Puerto Rico, April 1999. Springer.

[15] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in Lecture Notes in Computer Science, pages 46–66, Berkeley, CA, USA, July 2000. Springer.

[16] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 477–487. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[17] Frank Dabek, Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[18] Frank Dabek, Ben Zhao, Peter Druschel, and Ion Stoica. Towards a common API for structured peer-to-peer overlays. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003. Springer.

[19] Nirmit Desai and Frank Mueller. A log(n) multi-mode locking protocol for distributed systems. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 4, Nice, France, April 2003. IEEE.

[20] Ian Foster and Adriana Iamnitchi. On death, taxes, and the convergence on peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003. Springer.

[21] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.

[22] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Supercomputer Applications*, 15(3):200–222, March 2001.

[23] Pierre Fraigniaud and Philippe Gauron. An overview of the content-addressable network D2B. Research report 1349, LRI, University Paris-Sud, France, January 2003. `http://www.lri.fr/~pierre/POSTSCRIPTS/PODC2003.ps`.

[24] L. Garces-Erce, E. Biersack, P. Felber, Keith W. Ross, and G. Urvoy-Keller. Hierarchical peer-to-peer systems. To appear in the proceedings of Euro-Par, 2003.

[25] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium Computer Architecture (ISCA 1990)*, pages 15–26, Seattle, USA, June 1990.

[26] Liviu Iftode. *Home-based Shared Virtual Memory*. Thèse de Ph.D., Princeton University, NJ, June 1998.

[27] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, USA, February 2003. Springer.

[28] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS 2000)*, number 2218 in Lecture Notes in Computer Science, pages 190–201, Cambridge, MA, USA, November 2000. Springer.

[29] H. T. Kung and Chun-Hsin Wu. Hierarchical peer-to-peer networks. Technical Report TR-IIS-02-015, Institute of Information Science, Academia Sinica, Taiwan, April 2001. `http://www.iis.sinica.edu.tw/LIB/TechReport/tr2002/threebone0215.html`.

[30] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[31] Sing Li. *Early Adopter JXTA: Peer-to-Peer Computing with Java*. Wrox Press, April 2002.

[32] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP Labs, March 2002. `http://www.hpl.hp.com/techreports/2002/HPL-2002-57.pdf`.

[33] David Molnar, Roger Dingledine, and Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Designing Privacy Enhancing Technologies, International Workshop on Design Issues in Anonymity and Unobservability*, number 2009 in Lecture Notes in Computer Science, pages 67–95, Berkeley, CA, USA, July 2000. Springer.

[34] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A read/write peer-to-peer file system. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, December 2002.

[35] Andy Oram. *Gnutella*, chapter 8 in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, pages 94–122. O'Reilly, May 2001.

[36] Andy Oram. *Performance*, chapter 14 in Peer-to-Peer: Harnessing the Power of Disruptive Technologies, pages 203–241. O'Reilly, May 2001.

[37] James Plank, Micah Beck, Wael Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: Storage in the network. In *Network Storage Symposium (NetStore '99)*, Seattle, WA, October 1999.

[38] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, Newport, Rhode Island, June 1997.

[39] Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. Routing algorithms for DHTs: Some open questions. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, number 2429 in Lecture Notes in Computer Science, pages 45–52, Cambridge, MA, USA, March 2002. Springer.

[40] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2001)*, pages 329–350, Heidelberg, Germany, November 2001.

[41] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[42] Ian Foster S. Vazhkudai, Steven Tuecke. Replica selection in the globus data grid. In *1st IEEE/ACM International Conference on Cluster Computing and the Grid (CCGrid2001)*, pages 106–113, Brisbane, Australia, May 2001. IEEE.

[43] Jean-Marc Seigneur. JXTA pipes performance. `http://bench.jxta.org/papers.html`, 2002.

[44] Kwangwook Shin, Seunghak Lee, Geunhwi Lim, H. Yoon, and Joong Soo Ma. Grapes: Topology-based hierarchical virtual network for peer-to-peer lookup services. In *2002 International Conference on Parallel Processing Workshops (ICPPW'02)*, pages 159–166, Vancouver, Canada, August 2002.

[45] Clay Shirky. What is P2P... and what isn't it? `http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html`, November 2000.

[46] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, number 2429 in

Lecture Notes in Computer Science, pages 261–269, Cambridge, MA, USA, March 2002. Springer.

[47] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM 2001 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM 2001)*, pages 149–160, San Diego, CA, August 2001.

[48] Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul. Project JXTA: A loosely-consistent DHT rendezvous walker, March 2003. Submitted.

[49] Brendon J. Wilson. *JXTA*. New Riders Publishing, June 2002.

[50] Ben Yanbin Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001. `http://citeseer.nj.nec.com/article/zhao01tapestry.html`.

[51] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, Seattle, WA, October 1996.

[52] The Condor project. `http://www.cs.wisc.edu/condor/`.

[53] DIET. `http://graal.ens-lyon.fr/~diet/`.

[54] The Globus project. `http://www.globus.org/`.

[55] JavaGroups. `http://www.javagroups.com/javagroupsnew/docs/index.html`.

[56] JXTA distributed framework. `http://jdf.jxta.org/servlets/ProjectHome`.

[57] JavaSpaces service specification, v1.2.1. `http://wwws.sun.com/software/jini/specs/js1_2_1.pdf`.

[58] Jini architecture specification. `http://wwws.sun.com/software/jini/specs/jini1_2.pdf`.

[59] The JXTA project. `http://www.jxta.org/`.

[60] JXTA v2.0 protocol specification. `http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.pdf`, March 2003.

[61] JxtaView. `http://jxtaview.jxta.org/servlets/ProjectHome`.

[62] KaZaA. `http://www.kazaa.com/`.

[63] Legion - the worldwide virtual computer. `http://www.cs.virginia.edu/~legion/`.

[64] The MetaNEOS project. http://www-unix.mcs.anl.gov/metaneos/.

[65] Napster protocol specification. http://opennap.sourceforge.net/napster.txt, March 2001.

[66] The NetSolve project. http://icl.cs.utk.edu/netsolve/.

[67] Ninf. http://ninf.apgrid.org/.

[68] The SETI@home project. http://setiathome.ssl.berkeley.edu/.