La plate-forme JuxMem : support pour un service de partage de données sur la grille

Gabriel Antoniu, Luc Bougé, Mathieu Jan

RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

La plate-forme JuxMem : support pour un service de partage de données sur la grille

Gabriel Antoniu, Luc Bougé, Mathieu Jan

IRISA/INRIA et ENS Cachan/Antenne de Bretagne
Campus de Beaulieu, 35042 Rennes, France
Contact : Gabriel.Antoniu@irisa.fr

Résumé
Ce papier propose une architecture hiérarchique pour un service de partage de données modifiables pour une fédération de grappes. Cette architecture est illustrée par une plate-forme logicielle appelée JuxMem, qui permet un accès transparent aux données tout en assurant leur persistance en environnement dynamique. Nous introduisons les principaux concepts de cette architecture, et tout particulièrement les mécanismes permettant de gérer la volatilité des ressources. Un prototype de la plate-forme JuxMem a été mis en œuvre sur la plate-forme pair-à-pair JXTA. Nous présentons une évaluation expérimentale préliminaire de ce prototype à l'aide d'une application synthétique.

Mots-clés : partage de données, grille, pair-à-pair, architecture hiérarchique, JXTA

1. Introduction

L'un des principaux apports des environnements de calcul sur grille développés jusqu'à présent est d'avoir *découplé les traitements de leur déploiement*. En effet, le déploiement est vu comme un service proposé par 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. Au mieux, des fonctionnalités peu évoluées de type transfert de fichiers sont proposées. 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.
Nous pensons qu'une approche adéquate pour répondre à ce problème consiste à *découpler l'application de calcul de la gestion des données associées*, à travers un *service de partage de données pour la grille*, adapté aux contraintes du calcul scientifique. Nous considérons le cas particulier d'une grille constituée d'une fédération de grappes distribuée. Ce service doit fournir essentiellement deux propriétés.

La persistance a pour objectif de permettre de stocker des données sur la grille indépendamment des applications, afin de permettre leur réutilisation efficace.

La transparence vise à décharger les applications de la gestion explicite de la localisation et du transfert des données.

Trois contraintes principales doivent être prises en compte.

Extensibilité. Alors que l'algorithmique du calcul parallèle a été bien étudiée pour des configurations à nombre relativement réduit de machines, de l'ordre de quelques dizaines, on vise aujourd'hui une architecture à plusieurs milliers ou dizaines de milliers de nœuds.

Volatilité. La disponibilité des grappes de machines constituant la grille de calcul n'est pas toujours garantie. En effet, les nœuds peuvent être arrêtés puis redémarrés suite à des problèmes techniques ou à cause de l'indisponibilité temporaire des ressources. Cette indisponibilité ne doit pas entraîner l'indisponibilité du service de données.

Cohérence. Dans les applications de calcul visées, les données sont généralement partagées et peuvent être modifiées par plusieurs sites. Un service de partage de données pour la grille doit intégrer des protocoles de cohérence adaptés à la grande échelle, ce qui signifie que les protocoles traditionnels doivent être revisités.

Le type de service que nous proposons se rapproche de plusieurs types de systèmes de gestion de données existants mais qui ne prennent en compte que partiellement les objectifs et les trois contraintes mentionnées ci-dessus.

Gestion de données sans transparence. Actuellement, l'approche la plus utilisée pour la gestion des données nécessaires aux calculs répartis sur différentes machines d'une grille de calcul repose sur des *transferts explicites* des données entre les clients et les serveurs de calcul. À titre d'exemple, la plate-forme Globus [**?**] fournit des mécanismes d'accès aux données (GASS) basés sur le protocole GridFTP [**?**], un protocole de type FTP enrichi de quelques fonctionnalités (gestion de l'authentification, transferts parallèles, gestion des reprises en cas d'échec, etc.), qui exige une gestion explicite de la localisation des données.

Stockage de données à grande échelle pour le calcul. Le projet IBP [**?**] propose un système de stockage à grande échelle permettant de gérer un ensemble de tampons présents sur Internet. L'utilisateur peut *louer* ces espaces de stockage et les utiliser pour des transferts de données. IBP a été utilisé par l'environnement Netsolve [**?**] pour implémenter un service de gestion de données persistantes. La gestion des transferts et de la cohérence restes encore à la charge de l'utilisateur.

Partage transparent à petite échelle. 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). Dans ce cadre, un nombre important de modèles et de protocoles de cohérence ont été définis pour permettre la gestion efficace des données répliquées. Ces systèmes offrent un accès transparent 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 et de la cohérence des copies. Cependant, les MVP ont montré une efficacité satisfaisante uniquement lorsque le nombre de machines est faible, de l'ordre d'une dizaine [**?**].

Partage pair-à-pair de données non-modifiables. Le modèle pair-à-pair (*peer-to-peer*) [**?**] complémente le modèle classique *client-serveur*, en *symétrisant* la relation des machines : chaque machine peut être client dans une transaction et serveur dans une autre. Popularisé par Napster, Gnutella, et aujourd'hui KaZaA [**?**], ce paradigme a été utilisé essentiellement pour le *partage de fichiers répliqués en lecture seule*.

Partage pair-à-pair de données modifiables. Les systèmes comme OceanStore [**?**] et Ivy [**?**] proposent quelques mécanismes de partage de données modifiables en environnement pair-à-pair. Dans OceanStore, seul un petit ensemble des copies d'une donnée décide de l'ordre des modifications et propage les modifications aux autres copies. Toutefois, OceanStore utilise un mécanisme de versions pour la gestion des données qui n'est pas prouvé être efficace à grande échelle, comme le montre les mesures publiées qui supposent un seul écrivain par donnée. Ivy délègue la résolution des éventuels conflits en écriture au niveau applicatif, limitant le nombre d'écrivains par donnée. Enfin, ces systèmes visent à assurer le stockage persistant de *fichiers* et non pas la gestion de données pour le calcul sur la grille (par exemple le transfert de matrices réparties à l'aide de flux parallèles).

2. Convergence entre pair-à-pair et mémoires virtuellement partagées

Notre idée est de proposer un service de partage de données pour le calcul scientifique sur grille, en s'inspirant essentiellement des points forts des deux dernières approches :

les systèmes MVP qui proposent des modèles et protocoles de cohérence permettant la gestion efficace et transparente de données modifiables à petite échelle ;

les systèmes pair-à-pair qui fournissent des mécanismes de gestion de données non-modifiables à très grande échelle, en environnement très volatile.

L'architecture que nous visons pour notre 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. Ceci est illustré dans le tableau ci-dessous.

	MVP	**Service pour la grille**	**P2P**
Échelle	10^1–10^2	10^3	10^5–10^6
Contrôle des ressources	Fort	Moyen	Nul
Dynamicité	Nulle	Moyenne	Forte
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 scientifique	Calcul scientifique et stockage de données	Partage et stockage de fichiers

3. JuxMem : une plate-forme support pour le partage de données sur la grille

L'architecture du service que nous proposons reflète une architecture constituée d'une fédération largement distribuée de grappes de machines. Il s'agit donc d'une architecture *hiérarchique*, que nous illustrons à travers la proposition d'une plate-forme logicielle appelée JuxMem (pour *Juxtaposed Memory*), dont l'objectif est de servir de support à la mise en œuvre d'un service de partage de données.

3.1. Architecture hiérarchique

La figure **??** illustre la hiérarchie des entités présentes dans l'architecture de JuxMem. Elle se compose d'un réseau de groupes de pairs (groupes `cluster` A, B et C), qui correspondent généralement au niveau physique à des grappes de machines. L'ensemble des groupes sont englobés dans un groupe plus large qui inclut l'ensemble des pairs (groupe `juxmem`). Chaque groupe `cluster` regroupe un ensemble de nœuds qui offrent de l'espace mémoire pour le stockage de données. Nous appellerons ces nœuds des *fournisseurs*. Dans chaque groupe `cluster` il existe un nœud responsable de la gestion des zones mémoires disponibles au sein du groupe. Ce nœud est appelé *gestionnaire de grappe*. Enfin, un nœud qui utilise le service est appelé un *client*. Il est à noter qu'un nœud peut être à la fois gestionnaire de grappe, client et fournisseur, toutefois sur la figure et pour plus de clarté, chaque nœud ne joue qu'un seul rôle.
À chaque bloc de données stocké dans le système est associé un groupe de pairs appelé groupe `data`. Ce groupe peut être composé de fournisseurs appartenant à des groupes différents. En effet, un bloc de données n'est pas spécifique à une grappe de machines et peut être réparti sur deux grappes (ici A et C). Les groupes `data` et `cluster` sont donc situés sur le même niveau de la hiérarchie des groupes. Du point de vue des clients seuls les groupes `cluster` et les groupes `data` sont visibles. Ainsi, un client membre du groupe `juxmem` ne voit pas l'ensemble des fournisseurs des différentes grappes de la grille. Remarquons aussi qu'on peut également envisager de faire correspondre plusieurs groupes `cluster` à des sous-ensembles d'une grappe physique. Enfin, il faut noter que l'architecture de JuxMem est dynamique puisque les groupes de type `cluster` et `data` peuvent être créés à l'exécution. Ainsi, pour chaque bloc de données introduit dans le système, un groupe de type `data` est automatiquement instancié.

3.2. Interface de programmation du service de partage

La plate-forme JuxMem offre au niveau applicatif l'interface d'un service de partage de bloc de données, dont une partie des primitives sont décrites ci-dessous.

`alloc(size,options)` permet d'allouer une zone mémoire d'une taille de `size` sur une grappe. Le paramètre `options` permet de spécifier le degré de réplication et le protocole de cohérence utilisé par défaut pour gérer le bloc de données qui va être stocké dans la zone mémoire allouée. La primitive retourne un identifiant qui peut être assimilé au niveau applicatif à un identifiant de bloc de données.

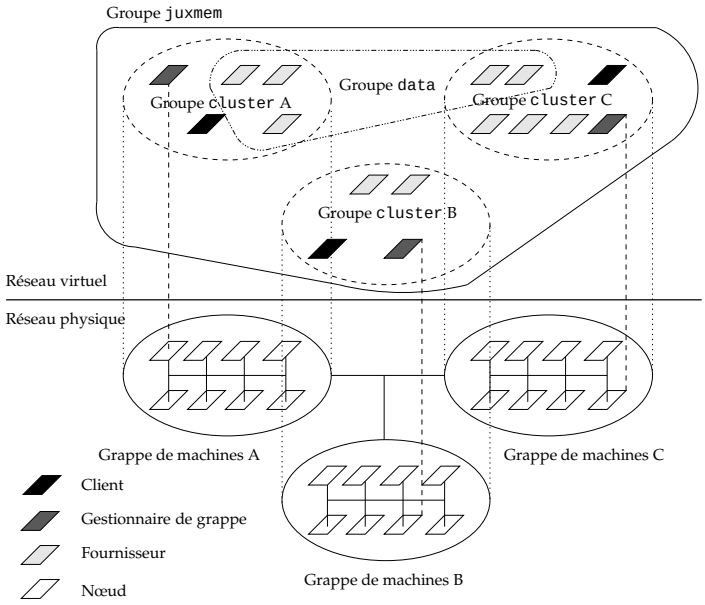

FIG. 1 – Hiérarchie des entités dans le réseau virtuel défini par JuxMem.

`map(id,options)` permet de récupérer l'annonce du canal de communication permettant 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 client, comme par exemple ce que nous appelons le degré de cohérence : par exemple une cohérence plus faible, par rapport au protocole par défaut, peut suffire pour certains clients.

`put(id,value)` permet de modifier la valeur du bloc de données spécifié par l'identifiant `id`. La nouvelle valeur du bloc de données sera alors `value`.

`get(id)` permet d'obtenir la valeur courante du bloc de données spécifié par l'identifiant `id`.

`lock(id)` permet de verrouiller le bloc de données spécifié par l'identifiant `id`. À chaque bloc de données est associé un verrou, qui doit être utilisé lorsqu'on souhaite manipuler le bloc de données.

`unlock(id)` permet de déverrouiller le bloc de données spécifié par l'identifiant `id`.

D'autres opérations sont disponibles dans l'API de JuxMem, comme par exemple une primitive permettant de reconfigurer un nœud afin de modifier dynamiquement la quantité de mémoire mise à disposition en tant que fournisseur.

3.3. Gestion des ressources mémoires

3.3.1. Publication et placement des annonces de ressources

Les ressources mémoires sont gérées par l'utilisation d'*annonces*. Chaque fournisseur publie la taille mémoire dont il dispose au sein du groupe `cluster` dont il fait partie par l'intermédiaire d'une *annonce de type fournisseur*. Le gestionnaire de grappe stocke les différentes annonces de ce type présentes au sein de son groupe. Il est également responsable de la publication de la liste des tailles mémoires disponiblent sur la grappe par une *annonce de type grappe* au sein du groupe `juxmem`. Ces annonces peuvent alors être utilisées par tous les clients pour demander l'allocation d'une zone mémoire.

L'une des contraintes fixées est la volatilité des nœuds formant une grappe de machines. Par conséquent, les annonces publiées à l'instant t ne sont plus forcément valides à l'instant $t + 1$. En effet, un fournisseur peut disparaître de l'architecture à tout moment. Le mécanisme utilisé pour gérer cette volatilité des pairs consiste à republier l'annonce de type grappe lorsqu'une variation de l'espace global mis à disposition est détectée. Toutefois, afin d'éviter que trop d'annonces invalides soient disponibles, celles-ci ont une durée de vie paramétrable. Il est alors nécessaire de les republier périodiquement.

Les gestionnaires de grappes sont chargés de faire le lien entre le groupe `cluster` et le groupe `juxmem`. Ils sont utilisés pour constituer un réseau de pairs organisés en utilisant une *table de hachage distribuée* au

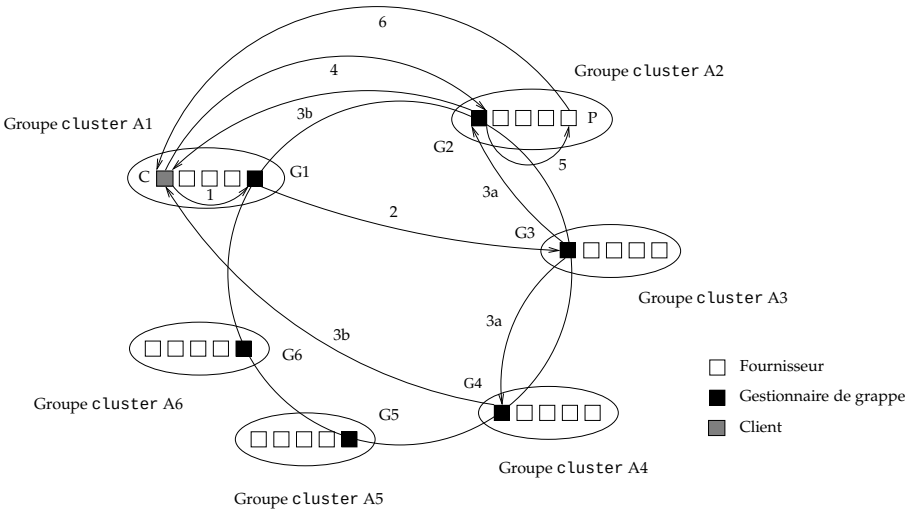

FIG. 2 – Étapes d'un requête d'allocation par un client.

niveau du groupe `juxmem`, afin de former l'ossature du service de partage de blocs de données. Cette ossature est représentée par l'anneau de la figure **??**. Chaque gestionnaire de grappe G1 à G6 est responsable d'une grappe de machines respectivement A1 à A6 constituée de cinq nœuds. Chaque élément de la liste des tailles mémoires contenu dans les annonces de type grappe est utilisé séparemment pour générer un identifiant, par application d'une fonction de hachage. Cet identifiant est utilisé pour déterminer le gestionnaire de grappe responsable de l'ensemble des annonces qui contiennent cette taille mémoire. Le gestionnaire de grappe 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.

3.3.2. Cheminement d'une requête d'allocation

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

1. Le 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 gestionnaire G1 auquel il est connecté.
2. Le gestionnaire G1 va alors déterminer, en utilisant le mécanisme de hachage décrit précédemment, que le pair responsable des annonces ayant une taille mémoire de 10 dans leur liste est le gestionnaire G3, et donc lui transmettre la requête.
3. Le gestionnaire G3 va alors déterminer que les gestionnaires G2 et G4 repondent aux critères du client, et donc leur demander de transmettre leur annonce de type grappe au client C.
4. Le client C choisit alors le gestionnaire G2 comme étant la meilleure annonce, la grappe sous-jacente offrant actuellement un degré de réplication supérieur à la grappe dont est responsable le gestionnaire G4, et lui soumet sa requête d'allocation.
5. Le gestionnaire G2 reçoit la requête d'allocation qu'il peut satisfaire et va alors demander à l'un de ses fournisseurs, par exemple P, d'allouer une zone mémoire de taille 10. Si une telle requête ne peut être satisfaite un message d'erreur est retourné au client.
6. Le fournisseur P peut également satisfaire cette requête et va donc créer une zone mémoire de taille 10 puis renvoyer l'annonce de cette zone mémoire au client C. Il va devenir responsable pour le groupe `data` associé, ainsi que chercher à répliquer cette zone. De la même manière, si une telle requête ne peut être satisfaite un message d'erreur est retourné au gestionnaire G2.

3.4. Gestion des données partagées

L'allocation d'une zone mémoire par un client se traduit par la création, sur le fournisseur, d'un groupe `data` et par l'envoi d'une annonce au client, permettant de communiquer avec ce groupe. Cette annonce

est publiée au niveau du groupe `juxmem`, mais seul l'identifiant de cette annonce est retourné au niveau applicatif. L'accès aux bloc de données par d'autres clients est ainsi transparente et possible par la seule connaissance de cet identifiant, la plate-forme se chargeant de localiser le bloc de données. Le stockage des bloc de données n'est pas lié à un client et de ce fait est persistant.
Chaque bloc de données est répliqué sur un certain nombre de fournisseurs pour une meilleure disponibilité. Il faut donc assurer la cohérence entre ces copies. L'emploi d'une solution de type *multicast* permet de résoudre ce problème : les différentes copies d'un bloc de données sont ainsi simultanément mises à jour lors de toute opération de modification. Les clients ne stockent pas de copie locale du bloc de données, et ne sont donc pas informés des modifications. Ainsi, le résultat d'une lecture valide à un instant $t1$, peut ne pas être valide à un temps $t2 > t1$. Cette distinction permet de gérer un grand nombre de clients sans avoir un grand nombre de copies d'un bloc de données. La gestion de la synchronisation entre les clients repose sur un mécanisme de type verrou, par utilisation des opérations *lock* et *unlock*.

3.5. Volatilité des fournisseurs

Afin de supporter la volatilité des pairs, une simple réplication statique des blocs de données sur un certain nombre de fournisseurs n'est pas suffisante. En effet, les fournisseurs 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. Chaque groupe `data` a un fournisseur (appelé *gestionnaire de bloc de données*) qui est donc chargé de contrôler le niveau de réplication du bloc de données, et si ce nombre est inférieur à celui souhaité par les clients, il doit chercher puis demander à un fournisseur d'héberger une copie du bloc de données. Lorsque le gestionnaire du bloc de données décide de le répliquer, il doit au préalable le verrouiller afin d'en assurer la cohérence. Le fournisseur qui va accueillir cette copie est alors en charge de le déverrouiller. L'utilisation d'un mécanisme de *timeout* suivi d'un test de type *ping* sur le fournisseur qui réplique le bloc de données, par le gestionnaire du bloc de données, permet de détecter si le fournisseur s'est déconnecté juste avant de déverrouiller le bloc de données.

3.6. Volatilité des gestionnaires

La perte d'un gestionnaire de grappe peut entraîner l'indisponibilité des ressources d'une grappe, puisqu'il est responsable de traiter les demandes d'allocation des clients. Le rôle du gestionnaire de grappe (noté *gestionnaire de grappe principal*) est donc automatiquement dupliqué sur l'un des fournisseurs de la grappe (noté *gestionnaire de grappe secondaire*). Les gestionnaires de grappes utilisent un mécanisme d'échange des annonces de type fournisseurs pour s'informer des nouvelles annonces de type fournisseur publiées, afin de connaître de manière quasi-exacte la taille mémoire disponible sur la grappe. Un mécanisme basé sur l'échange périodique de messages de vie permet d'assurer de manière dynamique cette duplication des gestionnaire de grappes. Un tel mécanisme est également utilisé pour la gestion des gestionnaires de bloc de données (voir section **??**). Les éventuels changements de gestionnaires au sein des groupes `cluster` et `data`, suite à la disparition d'un gestionnaire, ne sont pas visibles depuis l'extérieur de ces groupes. La disponibilité d'une grappe et des bloc de données est donc maximisée.

4. Implémentation dans l'environnement JXTA et évaluation préliminaire

Afin de bâtir rapidement un prototype de l'architecture logicielle définie à la section précédente, nous avons utilisé la plate-forme JXTA [**?**]. JXTA offre les mécanismes de base pour la gestion de systèmes pair-à-pair (découverte, gestion des groupes de pairs, communications génériques inter-pairs, etc.) et permet la définition de services et d'applications pair-à-pair. Elle se présente sous la forme d'une spécification en XML d'un ensemble de protocoles, indépendants des plates-formes et des langages.
Pour la réalisation de notre prototype JuxMem, nous avons utilisé l'implémentation de référence en Java, qui est la seule à ce jour à respecter la spécification JXTA 2.0. JuxMem est écrit en Java et représente plus de 5000 lignes de code réparties en 48 classes. Pour nos expérimentations préliminaires, nous avons utilisé une grappe de machines PII à 450 MHz, munies de 256 Mo de mémoire vive (RAM), inter-connectées par un réseau FastEthernet à 100 Mbits/s.
Nous avons mesuré l'influence du degré de volatilité des fournisseurs sur la durée d'une séquence `lock-put-unlock` effectuée sur un bloc de données par un client. L'objectif de cette mesure est d'évaluer le surcoût, en pourcentage, engendré par les réplications nécessaires au maintien d'un degré de

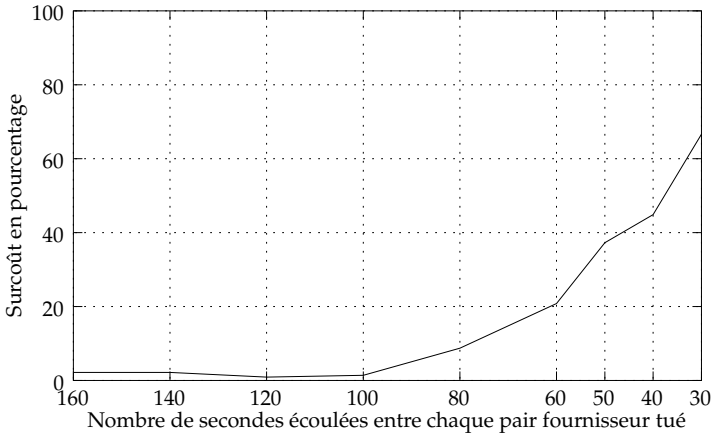

FIG. 3 – Perturbation en pourcentage introduite par la volatilité des fournisseurs pour une séquence `lock-put-unlock` par rapport à un système stable.

réplication pour le bloc de données en présence de perte de fournisseurs qui hébergent une copie du bloc de données.

Le programme de test consiste à allouer une zone mémoire de la taille d'un octet, avec un niveau de réplication de 3 sur une grappe constituée initialement de 16 fournisseurs et d'un gestionnaire de bloc de données. Chaque nœud de la grappe héberge un pair. Un client, également hébergé par un nœud de la grappe, réalise une boucle de 100 itérations `lock-put-unlock`. Pendant l'exécution de cette boucle, on tue aléatoirement un fournisseur toutes les δ secondes, δ étant le paramètre de la mesure. Afin de ne mesurer que le surcoût dû à la volatilité des fournisseurs, le gestionnaire de la donnée n'est jamais tué.

La figure **??** présente le surcoût en pourcentage par rapport à un système stable ($\delta = \infty$), c'est-à-dire dans lequel tous les fournisseurs restent connectés durant l'exécution de la boucle. En effet, lorsque le responsable du bloc de données détecte des disparation de copies du bloc de données, il cherche à le répliquer sur d'autres fournisseurs. Or, pendant la réplication, le système doit verrouiller le bloc de données de manière interne : un client ne doit pas pouvoir modifier le bloc de données pendant qu'un fournisseur en instancie une nouvelle copie. Ce verrouillage interne entraîne une augmentation de la durée de la séquence `lock-put-unlock` puisque le client est alors bloqué en attente de libération du verrou.

L'évolution de la courbe s'explique par le nombre de fois que le système réplique le bloc de données sur des fournisseurs afin de maintenir le nombre de copies souhaité par le client, en l'occurrence 3 pour ce test. Pour la durée totale de notre test, ce nombre 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 typiquement plus faible ($\delta \gg 80$ s). Pour de telles valeurs, le surcoût de la reconfiguration du système est très faible, inférieur à 5 %. La plate-forme JuxMem proposée dispose donc d'un mécanisme qui permet de maintenir *dynamiquement* le nombre de copies d'un bloc de données, afin de maximiser sa disponibilité et autorisant la perte de pairs, *sans un surcoût important* pour ce maintien.

5. Conclusion

Ce papier 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.

JuxMem permet l'allocation de zones mémoires au sein de machines constituant une grappe. Au niveau applicatif, la création de telles zones se traduit par la récupération d'un identifiant. Ainsi, *la localisation et le transfert des données sont transparents vis-à-vis des applications* puisqu'il suffit de spécifier ces identifiants pour y accéder et les manipuler.

L'architecture définie supporte la volatilité de tout type de pairs. Cette volatilité est évidemment supportée dans les systèmes pair-à-pair comme Gnutella ou KaZaA, qui assurent la disponibilité d'une donnée grâce à la redondance, mais ceci est un résultat indirect des actions que réalisent les utilisateurs. Notre système prend *activement en compte cette volatilité* afin de maintenir un certain degré de redondance des données (comme dans Ivy ou CFS [**?**]), mais aussi pour supporter la volatilité des pairs ayant des responsabilités de gestion des données ou des grappes.

La réalisation d'un prototype sur JXTA a pour but de montrer la faisabilité d'un tel système. Mais, la conception de JuxMem n'est pas dépendante de JXTA, qui pourrait être remplacé par d'autres librairies (par exemple JavaGroups [**?**]). Concernant le choix de Java comme langage d'implémentation, celui-ci n'est aucunement dicté par l'utilisation de JXTA : il s'agit simplement de l'implémentation la plus aboutie à ce jour (compatible avec la spécification JXTA 2.0). L'architecture modulaire de JXTA permet d'ajouter et d'enlever des services notamment au niveau des protocoles réseaux utilisés, ce qui facilitera, à terme, l'exploitation de réseaux comme Myrinet ou SCI.

À l'avenir, nous nous proposons d'utiliser JuxMem comme plate-forme d'expérimentation de différents protocoles de cohérence des données supportant la volatilité, afin de construire un service de partage de données utilisable par des environnements de calcul numérique sur grille tels que DIET [**?**]. Cela permettra une évaluation plus poussée du service, avec des codes réalistes, utilisant différents schémas d'accès pour les données.