Cohérence et volatilité dans un service de partage de données dans les grilles de calcul

Jean-François Deverge, Sébastien Monnet

RENPAR'16
Le Croisic, Presqu'île de Guérande, France, 5 au 8 avril 2005

Cohérence et volatilité dans un service de partage de données dans les grilles de calcul

Jean-François Deverge et Sébastien Monnet

Université de Rennes 1,
IRISA, Campus de Beaulieu, 35042 Rennes, France
Contact : {Jean-Francois.Deverge,Sebastien.Monnet}@irisa.fr

Résumé
Nous nous intéressons au partage de données modifiables à grande échelle dans les grilles de calcul. Une contrainte pour la mise en œuvre d'un tel système est la gestion de la cohérence des données en environnement volatile. Dans ce travail, nous proposons une architecture qui découple les mécanismes de tolérance aux fautes de la gestion de la cohérence. Nous validons cette approche par la conception d'un protocole de cohérence hiérarchique tolérant aux fautes sur la plate-forme JUXMEM, qui est un service de partage de données pour la grille. Nous présentons également une évaluation expérimentale préliminaire de ce protocole.

Mots-clés : grille de calcul, partage de données, tolérance aux fautes, protocoles de cohérence

1. Introduction

La gestion de données à large échelle est un besoin crucial pour les applications sur la grille de calcul. Cependant, il n'existe pas d'approche standard pour le partage de données qui soit largement acceptée par la communauté. Les solutions actuelles sont basées sur *des transferts explicites des données* entre les entités (composants logiciels, nœuds de la grille). Par exemple, la plate-forme Globus [9] permet l'accès et la manipulation de données à travers le protocole GridFTP [1]. GridFTP enrichi le protocole FTP par la possibilité de transferts parallèles, de mécanismes de reprises des téléchargements, etc. Au-dessus de ce protocole, Globus fournit des catalogues des données [1] où l'utilisateur peut enregistrer la localisation de plusieurs copies d'une même donnée. Les problèmes de cohérence entre les copies restent à la charge de l'utilisateur. IBP [5] propose un espace de stockage à grande échelle en utilisant l'ensemble des espaces de stockages disponibles sur les nœuds du réseau. Cependant, la gestion des transferts entre ces espaces de stockages et la gestion de la cohérence des multiples copies des données restent à la charge de l'utilisateur.
Avec le nombre croissant d'applications qui emploient d'importants volumes de données, nous pensons que la gestion explicite des données devient une difficulté majeure pour les programmeurs. La gestion à bas niveau de la localisation des copies et de leur cohérence est un réel problème pour l'exploitation efficace de l'ensemble des ressources de la grille. Par conséquent, un *service de partage de données* pour le calcul sur grille doit fournir un *accès transparent aux données*. Le prototype JUXMEM [3] est une illustration de ce type de service. Le service gère, de manière transparente, la localisation et le transfert des données sans aucune intervention de l'utilisateur. De plus, vu la grande échelle du système (plusieurs milliers de nœuds), un tel

service pour la grille doit être conçu pour un environnement volatile : les arrivées, les départs des ressources ou leur défaillance doivent être supportés de manière transparente. En conséquence, le rôle d'un service de partage réside aussi dans l'emploi de la stratégie de réplication adéquate et dans l'application d'un protocole de cohérence afin de garantir la disponibilité et la cohérence de la donnée.

Objectif : gestion de la cohérence des copies d'une donnée

Un service de partage de données doit permettre aux applications de la grille de calcul d'accéder aux données dans un environnement distribué. Par exemple, une application scientifique typique va fonctionner par le couplage de plusieurs codes distribués sur plusieurs sites. Dans ce type de contexte, les données partagées sont accessibles et *modifiables* par différents codes, potentiellement de manière *concurrente*. Si ces codes sont distribués sur plusieurs sites, le service aura recours à la réplication des données afin d'améliorer la localité des accès. Pour garantir que les opérations de lecture ne délivrent pas des données périmées, le service de partage applique des propriétés de cohérence sur les données. Ces garanties sont alors définies par un *modèle de cohérence* et sont appliquées par un *protocole de cohérence*.

Problème : gestion de la cohérence en environnement volatile

Le partage de données dans les systèmes distribués a déjà été largement étudié ces vingt dernières années pour les systèmes à mémoire virtuellement partagée (MVP) [14]. Ces systèmes délivrent, via un espace d'adressage unique, l'accès transparent à des données physiquement distribuées sur plusieurs nœuds. Pour chaque mise à jour d'une donnée, un ensemble de traitements sont réalisés afin de garantir la cohérence de la donnée. Dans ce contexte, un large spectre de modèles et de protocoles de cohérence [6, 12, 14, 16] ont été mis en œuvre. Ces modèles et protocoles visent à offrir différents compromis entre les garanties en terme de cohérence et l'efficacité de la mise en œuvre.
Cependant, les MVP traditionnelles n'ont démontré leur efficacité que sur des petites configurations de l'ordre de quelques dizaines de nœuds [14]. En effet, les algorithmes mis en œuvre pour gérer la cohérence fonctionnent, le plus souvent, par des invalidations ou des mises à jour de toutes le copies d'une donnée. De plus, la majorité de ces protocoles considèrent une configuration statique qui ne prend pas en compte les déconnections ou les défaillances des nœuds. De telles hypothèses ne sont pas réalistes pour des infrastructures de grille. Dans un environnement à large échelle, les fautes ne sont plus des exceptions mais font parti des hypothèses de base ; les ressources peuvent disparaître puis finalement se reconnecter ; de nouvelles ressources peuvent dynamiquement s'agréger au système. Ici, on ne peut pas adopter une approche qui considérerait que certaines entités sont stables. En conséquence, une nouvelle approche pour la conception de protocoles de cohérence est nécessaire, afin d'intégrer ces nouvelles hypothèses.
Dans cet article, nous proposons une approche qui permet de construire des protocoles de cohérence en les découplant des aspects liés à la tolérance aux fautes. Les motivations et les principes généraux pour cette approche sont décrits dans la section 2. Dans la section 3, nous proposons une étude de cas et la construction d'un protocole de cohérence tolérant aux fautes. La section 4 décrit l'intégration de ce protocole dans la plate-forme JUXMEM et présente des évaluations préliminaires sur les performances des opérations du protocole de cohérence. Plusieurs remarques sur les travaux futurs sont données en conclusion à la section 5.

2. Notre approche : séparer la gestion de la tolérance aux fautes et la gestion de la cohérence

Les mécanismes de tolérance aux fautes et les protocoles de cohérence sont souvent mis en œuvre par l'utilisation de la *réplication*. Cependant, les objectifs initiaux ne sont pas les mêmes dans ces deux cas.

La réplication employée pour les protocoles de cohérence.

Dans le cadre de la mise en œuvre de protocoles de cohérence, la réplication des données est utilisée à des fins de *performance*. Elle permet à plusieurs nœuds de lire une même donnée en parallèle en n'utilisant que des accès locaux. Cependant, lors de la mise à jour d'une donnée, le protocole de cohérence doit actualiser ou invalider les autres copies de la donnée présentes dans le système, de manière à maintenir la cohérence des différentes copies.

La réplication employée pour la tolérance aux fautes.

De nombreux mécanismes de tolérance aux fautes utilisent la réplication [15] pour supporter la volatilité des nœuds. Ainsi, malgrès la défaillance d'un des nœuds hébergeant une copie d'une donnée, cette donnée reste disponible grâce à l'existence des autres copies. Plusieurs techniques de réplication ont été étudiées [11], elles présentent des compromis efficacité/garanties différents.
Dans les systèmes distribués, lorsqu'il faut prendre en considération à la fois la gestion de la cohérence et la gestion de la tolérance aux fautes, la réplication peut être employée simultanément pour ces deux problèmes. Nous mettons en exergue qu'il existe deux architectures logicielles possibles pour résoudre la double difficulté localité/disponibilité.

Architecture intégrée. Cette approche consiste à gérer au même niveau les problèmes de la disponibilité et de la cohérence des données. Un seul ensemble de réplicats pour une donnée sert alors à la fois à améliorer sa localité et sa disponibilité. Par exemple, une copie créée par le protocole de cohérence à des fins de localité peut servir de sauvegarde en cas de défaillance. De même, des copies créées par les mécanismes de tolérance aux fautes peuvent être utilisées par le protocole de cohérence. Notamment employée pour la conception de systèmes à MVP tolérants aux fautes, cette approche a pour principal avantage de permettre la mise en œuvre de protocoles qui ont montré leur efficacité [13], au prix d'un fonctionnement complexe.

Architecture découplée. Une autre approche possible est de séparer les protocoles de cohérence des mécanismes de tolérance aux fautes. Cette approche présente trois principaux avantages. 1) Cela simplifie la conception du protocole de cohérence car il n'a pas besoin de se préoccuper des problèmes liés à la volatilité. Il est donc possible d'adapter des protocoles de cohérence existants qui ne supportent pas la volatilité. 2) Les mécanismes de tolérance aux fautes et les protocoles de cohérence peuvent être conçus indépendamment, en se concentrant sur leurs rôles spécifiques. 3) Enfin, cette approche permet d'expérimenter facilement différents protocoles de cohérence avec de multiples stratégies de tolérance aux fautes.

Dans cet article, nous nous intéressons à la conception de protocoles selon la deuxième approche. Par une étude de cas, nous montrons comment gérer à la fois la cohérence des données et la volatilité des nœuds en utilisant une architecture découplée.

Des modules de tolérance aux fautes comme briques de base des protocoles de cohérence.

Les protocoles de cohérence reposent souvent sur des entités supposées stables. Par exemple, dans les systèmes à MVP, les protocoles à *copie de référence* [12,16] reposent sur un nœud responsable du maintient de la cohérence d'une copie de la donnée. Cette hypothèse de stabilité n'est pas envisageable dans une grille où le grand nombre de nœuds implique une volatilité inévitable. Afin de pouvoir continuer à assurer un accès à la donnée en présence de *pannes franches*, nous utilisons des mécanismes courants dans les systèmes distribués tolérants aux fautes. Nous utilisons des mécanismes de réplication basés sur des protocoles de *diffusion atomique* décrits par [11]. La diffusion atomique garantie que toutes les répliques reçoivent les mêmes messages dans le même ordre. Ainsi, les entités critiques des protocoles de cohérence sont répliquées afin de garantir leur disponibilité.

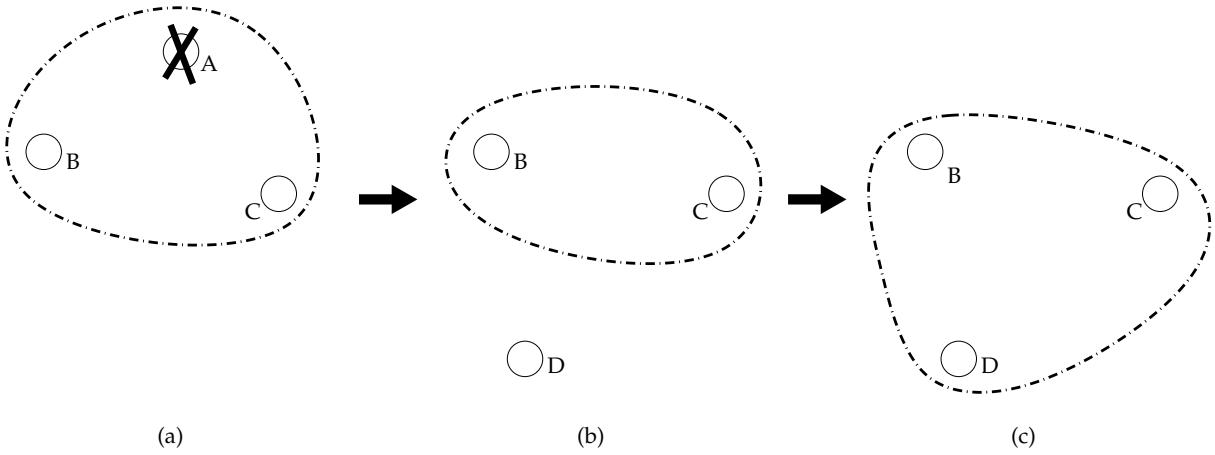

FIG. 1 – Remplacement d'un nœud défaillant dans un groupe de réplicats.

De plus, ces groupes de nœuds doivent toujours conserver le degré de réplication demandé. Ce type de propriété est possible par le remplacement dynamique des réplicats défaillants. La figure 1 montre un exemple de scénario du fonctionnement du module de tolérance aux fautes. Le nœud A subit une défaillance (Figure 1(a)) et ne participe plus à la réplication de la donnée. Le module de tolérance aux fautes détecte la panne (Figure 1(b)) grâce à un détecteur de défaillances [8] et retire le nœud A du groupe. Il faut alors sélectionner un nouveau nœud à rajouter. La figure 1(c) montre l'intégration du nœud D dans le groupe.
Les interactions entre la couche du protocole de cohérence et la couche de tolérance aux fautes sont limitées. Pour la sélection des réplicats d'une donnée, la couche de réplication peut rechercher des nœuds qui ont des caractéristiques particulières (e.g. leur localisation) requises pour le déploiement correct du protocole de cohérence. De même, lorsqu'un nouveau nœud rejoint un groupe de réplicats, il est nécessaire de lui communiquer l'état courant du protocole de cohérence. Dans la suite de cet article, nous nous focalisons sur la conception d'un protocole de cohérence qui illustre cette architecture découplée.

3. Etude de cas : la conception d'un protocole de cohérence

Nous définissons un service de partage de données pour des applications de couplage de code qui fonctionnent sur une grille de calcul. Nous nous plaçons dans un contexte où plusieurs codes s'exécutent en parallèle sur plusieurs grappes de machines et veulent s'échanger des données. L'objectif du service de partage est d'assurer un accès cohérent aux données et de gérer de manière transparente les défaillances des nœuds qui détiennent une copie de la donnée. C'est pourquoi, dans cette section, un protocole de cohérence tolérant aux fautes est conçu via l'approche présentée dans la section 2. Pour illustrer cette idée, nous allons partir d'un protocole de cohérence *non tolérant aux fautes* qui met en œuvre le modèle de *cohérence d'entrée*.

3.1. Le modèle de cohérence d'entrée

L'expérience des protocoles de cohérence dans les systèmes à MVP montre que des modèles affaiblis de cohérence permettent la mise en œuvre de protocoles plus efficaces [14]. Cette efficacité est obtenue au prix de l'utilisation, par le programmeur, de directives de synchronisation. Par exemple, l'assertion `acquire` dans le code permet de s'assurer que toutes les mises à jours ont été appliquées sur les données et l'assertion `release` permet de diffuser (au plus tôt ou de manière paresseuse) les modifications locales de la donnée vers les autres copies. Ce type de fonctionnement s'est généralisé pour plusieurs modèles de cohérence comme la cohérence à la libération [10], la cohérence d'entrée [6] ou la cohérence de portée [12]. Nous donnons quelques détails supplémentaires sur le modèle de *cohérence d'entrée* que nous mettons en œuvre dans la suite de cet article.

Contrairement aux autres modèles de cohérence faible, le modèle de cohérence d'entrée nécessite une association explicite entre les données et les verrous de synchronisation. Pour le protocole de cohérence, cela permet d'identifier les données à mettre à jour lors de l'accès à un verrou. La vue d'un client pour des données n'est alors cohérente que lorsqu'il entre en section critique sur le verrou associé à ces données. Par conséquent, seuls les nœuds qui déclarent l'intention d'accéder aux données *avec ces assertions* auront leurs données mises à jour. Ici, l'avantage majeur de ce modèle réside dans la diminution de communications réseaux inutiles. Cela permet de penser que ce modèle est un bon candidat pour la programmation des applications scientifiques sur la grille.

Avec ce modèle de cohérence, le programmeur doit appliquer deux règles principales. Toutes les données doivent être associées à au moins un verrou de synchronisation. De plus, les accès exclusifs (accès cohérents en lecture/écriture) aux données partagées doivent être distingués des accès non exclusifs (accès cohérents en lecture). La primitive `acquire` assure l'accès exclusif à la donnée et la primitive `acquireRead` permet l'accès concurrent par plusieurs clients à la donnée en lecture. Une description détaillée du modèle est donnée dans [6].

3.2. Un protocole *non tolérant aux fautes*

Notre point de départ est la conception d'un protocole de cohérence qui met en œuvre le modèle de cohérence d'entrée. Chaque donnée est hébergée par un nœud qui est la copie de référence de la donnée (ou *copie globale*). Ce nœud gère aussi le verrou associé à la donnée. Ainsi, lorsqu'un client rentre dans une section critique par l'obtention d'un verrou, la copie de la donnée du client est mise à jour si nécessaire. A la sortie de la section critique, le verrou est libéré et les modifications locales du client sont transmises à la copie globale. On peut remarquer que les accès à des données partagées par un client induisent des communications avec la copie globale. Cette configuration de protocole est illustrée par la figure 2 (a).

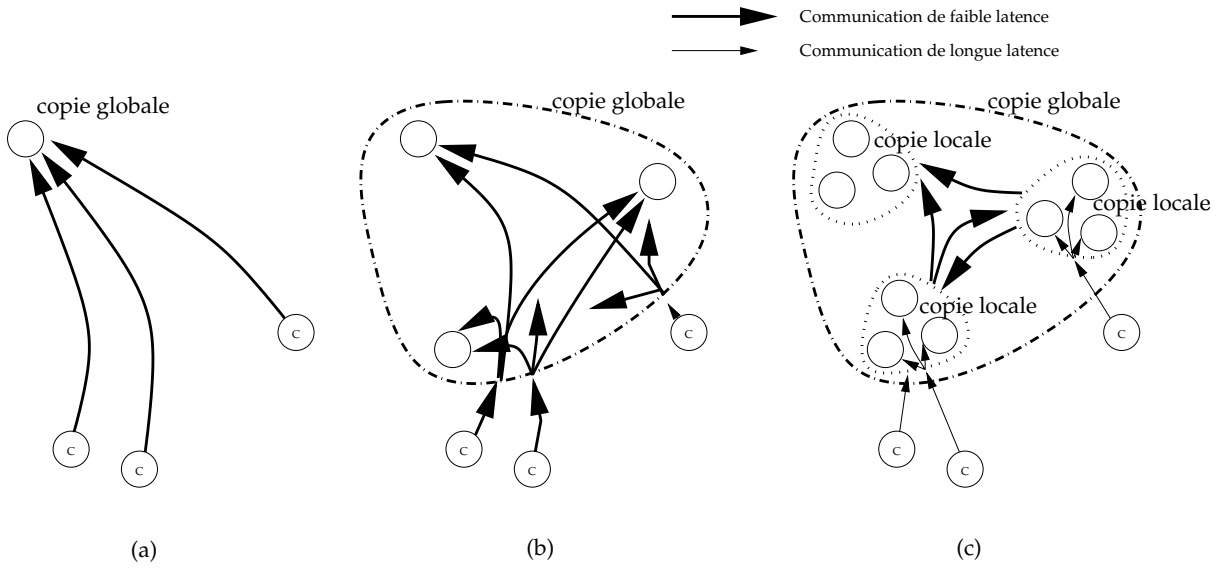

FIG. 2 – Plusieurs clients demandent l'accès à une donnée dans trois configurations possibles du protocole de cohérence.

3.3. Un protocole *tolérant aux fautes*

Dans le protocole qui vient d'être décrit, la copie globale est clairement une entité critique du système : sa disponibilité assure le fonctionnement du protocole. Dans un environnement à large échelle, nous ne pouvons pas construire notre protocole en considérant des nœuds stables, mais nous pouvons employer les méthodes de *réplication* pour améliorer la disponibilité, comme mentionné dans la section 2. La figure 2 (b) montre une configuration où la copie globale est répliquée sur trois nœuds. Notre proposition considère l'utilisation d'un protocole de gestion de groupe de haut niveau muni de deux propriétés : 1) Tous les messages sont reçus dans le même ordre par toutes les copies du groupe (diffusion atomique) ; 2) Les groupes sont dynamiques et doivent respecter le niveau de réplication spécifié par l'utilisateur en ajoutant de nouveaux réplicats pour chaque départ (panne franche ou déconnection).

Le nombre de fautes simultanées qui sont supportées par notre protocole de cohérence dépend des caractéristiques des protocoles tolérants aux fautes sous-jacents (diffusion atomique, consensus). Notre implémentation actuelle supporte jusqu'à $\lfloor \frac{n-1}{2} \rfloor$ fautes simultanées (voir section 4) pour chaque copie globale répliquée.

Nous pouvons remarquer que ce protocole de cohérence fonctionne *exactement de la même manière* avec ou sans l'entité répliquée du protocole. Ici, nous pouvons considérer que la copie globale est toujours disponible, car cette propriété est effectivement réalisée par les mécanismes de réplication. Grâce à cette approche, le protocole de cohérence et les mécanismes de tolérance aux fautes basés sur la réplication sont clairement découplés.

3.4. Un protocole *hiérarchique* tolérant aux fautes

Dans une grille constituée d'une fédération de grappes de nœuds, les latences inter grappes sont généralement plus élevées que les latences intra grappe. La figure 2 (b) montre que si la copie globale est répliquée sur des nœuds physiquement distribués dans des grappes différentes, chaque transaction d'un client vers la copie globale va impliquer des communications

à forte latence. En conséquence, pour améliorer l'efficacité de notre protocole, nous devons tenter de minimiser ces communications inter grappes. Cette idée a déjà été exploitée dans la conception de systèmes à MVP qui mettent en œuvre des protocoles hiérarchiques [4].
Pour améliorer le protocole décrit dans cette section, nous nous inspirons du protocole hiérarchique H2BRC décrit dans [4]. L'approche met en œuvre deux niveaux hiérarchiques de copies de référence. Ainsi, une *copie locale* à chaque grappe va servir l'accès à la donnée à tous les nœuds de la grappe, alors que la copie globale va desservir l'accès à la donnée aux copies locales des grappes. Lorsqu'un client veut l'accès à la donnée, le verrou est demandé à la copie locale. Si cette copie locale détient le verrou, l'accès peut être immédiatement délivré. Sinon, la copie locale va demander l'accès à la donnée depuis la copie globale, avec éventuellement une nouvelle version de la donnée. La figure 2 (c) montre que des copies locales (répliquées) permettent de rapprocher les accès pour des clients d'une même grappe.
Il faut remarquer que la copie globale redistribue les accès aux copies locales et qu'elle n'a aucun contrôle sur les redistributions du verrou effectuées au sein d'une grappe. De plus, la copie locale va maximiser l'accès à la donnée aux clients de sa grappe, ce qui a pour conséquence de minimiser les communications inter grappes. Pour éviter la famine sur l'accès du verrou par les clients des autres grappes, il est nécessaire de limiter le nombre de redistributions consécutives du verrou aux clients d'une même grappe.

4. Implémentation et évaluation préliminaire

Nous avons mis en œuvre notre architecture au sein de JUXMEM [3], une plate-forme expérimentale pour le partage de données sur la grille. Nous avons intégré le protocole de cohérence présenté dans la section 3. Les mécanismes de réplication sont basés sur le protocole de communication de groupe décrit dans [7]. Notre mise en œuvre du protocole est conçue pour supporter les pertes de messages et jusqu'à $\lfloor \frac{n-1}{2} \rfloor$ pannes franches simultanées, n est la cardinalité du groupe. Notre implémentation est basée sur JXTA [17] qui est une plate-forme générique de systèmes pair-à-pair.
Lors d'une allocation, le client spécifie le degré de réplication désiré. Ce degré de réplication est utilisé lors de l'instanciation des groupes de réplicats associés aux copies locales et à la copie globale. Le client obtient alors l'identifiant global de la donnée. Cet identifiant peut être utilisé par n'importe quel client de la grille de calcul pour accéder la donnée. Il est suffisant pour permettre au service de localiser la donnée. A noter que lorsqu'un client accède à une donnée qui ne se trouve pas dans sa grappe, une copie locale à la grappe est instanciée de manière transparente pour le client.

Évaluation préliminaire.

Pour nos expérimentations, nous avons utilisé *JDF* [2], un outil de déploiement et de tests pour les applications pair-à-pair basées sur JXTA. Nos évaluations ont été réalisées sur une grappe de 64 nœuds avec des bi-Pentium IV de 2,4GHz et 1GB de RAM, interconnectés par un réseau Fast-Ethernet. Cette grappe à été découpée de manière logique en 8 grappes de 8 nœuds pour émuler une fédération de grappes. Pour ces évaluations, nous utilisons JXTA 2.2.1 et Java 1.4.2.
Nos premières expériences concernent l'impact du degré de réplication sur le coût de l'allocation d'une donnée sur la grille. Ce mécanisme comporte trois phases : 1) la découverte de nœuds pour satisfaire le degré de réplication et leur sélection en fonction du degré de réplication ainsi que des contraintes de localité ; 2) l'envoi en parallèle des requêtes d'allocations aux nœuds sélectionnés ; 3) les nœuds qui reçoivent une requête d'allocation instancient le protocole de cohérence ainsi que le module de tolérance aux fautes.

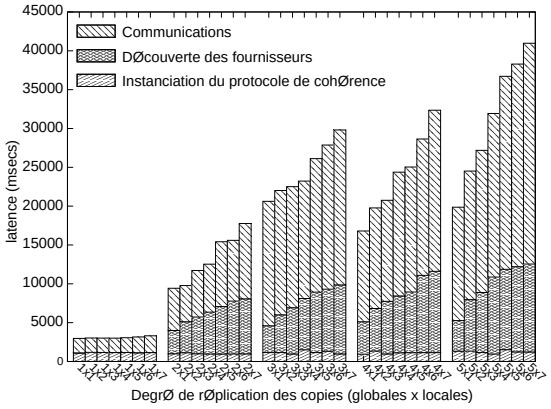

FIG. 3 – Coût de l'allocation en fonction du degré de réplication

Nous mesurons les coûts d'allocation pour différents dégrés de réplication pour la copie globale et les copies locales (Figure 3). Nous remarquons que : 1) le coût de l'allocation est essentiellement dû aux deux premières phases (découverte des nœuds et envoi des requêtes d'allocation) ; 2) le coût de la découverte croît linéairement avec le degré de réplication demandé ; 3) enfin, le coût de la troisième phase (l'instanciation du protocole) est quasi-constant. En effet, les requêtes sont émises et traitées en parallèle.

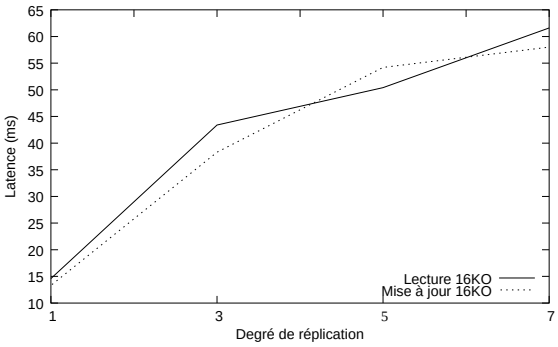

FIG. 4 – Performance des accès sur une donnée de 16 KO selon le degré de réplication.

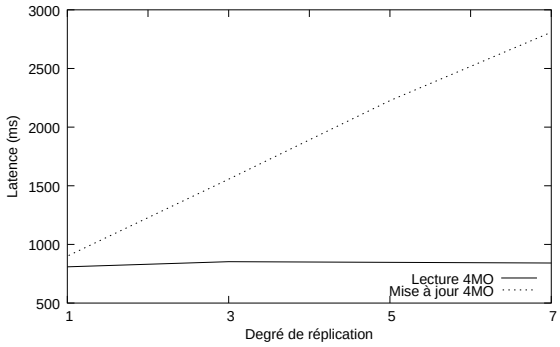

FIG. 5 – Performance des accès sur une donnée de 4 MO selon le degré de réplication.

Notre deuxième série d'expériences est portée sur le coût des opérations de base qu'un client est susceptible d'effectuer, c'est-à-dire les lectures et les mises à jour d'une donnée. Remarquons que ces opérations s'effectuent au sein d'une même grappe : il s'agit de communications entre un client et la copie locale à la grappe. Nous faisons varier le nombre de réplicats de la copie locale et nous en mesurons l'impact sur les coûts des opérations de lecture/mise à jour. Les résultats de ces expériences sont illustrés sur les figures 4 et 5. Nous remarquons que le surcoût dû à la réplication est significatif pour les données de petite taille (par exemple pour 16 KO). Les latences des opérations sont 3 fois plus importantes dès lors que l'on utilise la réplication. Ce surcoût est dû au protocole de communication de groupe : la diffusion atomique implique de nombreux échanges de petits messages, ce qui induit un coût en latence. Cependant, pour

des données de plus grande taille (ici 4 MO) ce surcoût est négligeable par rapport au temps de transfert de la donnée. Nous remarquons également que la latence des mises à jour croît linéairement avec le nombre de réplicats à actualiser. Ceci découle de notre implémentation de la diffusion où un nœud primaire retransmet la donnée à tous les autres réplicats du groupe.

5. Conclusion

Dans cet article, nous avons montré comment gérer la cohérence des données dans un environnement volatile. Nous avons décrit une architecture générique permettant de découpler la gestion de la cohérence des mécanismes de tolérance aux fautes. Comme première validation, nous avons instancié cette architecture en implémentant un protocole de cohérence tolérant aux fautes au sein de JUXMEM, une plate-forme expérimentale de service de partage de données pour la grille.
Le découplage *gestion de la cohérence / gestion de la tolérance aux fautes* permet d'adapter facilement les nombreux protocoles de cohérence existants. Nous prévoyons d'étudier de multiples protocoles de cohérences et mécanismes de tolérance aux fautes afin d'évaluer différents compromis entre l'efficacité et les garanties offertes par le service de partage.
L'approche présentée dans cet article permet de supporter des défaillances des entités critiques du service. Nous travaillons sur une extension de cette approche qui permet de prendre en compte les pannes des utilisateurs du service.

Bibliographie

1. Allcock (William), Bester (Joseph), Bresnahan (John), Chervenak (Ann), Foster (Ian), Kesselman (Carl), Meder (Sam), Nefedova (Veronika), Quesnel (Darcy) et Tuecke (Steven). – Data management and transfer in high-performance computational grid environments. *Parallel Computing*, vol. 28, N° 5, 2002, pp. 749–771.
2. Antoniu (Gabriel), Bougé (Luc), Jan (Mathieu) et Monnet (Sébastien). – Large-scale deployment in P2P experiments using the JXTA distributed framework. *In : Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par '04)*. pp. 1038–1047. – Pisa, Italy, 2004.
3. Antoniu (Gabriel), Bougé (Luc) et Jan (Mathieu). – La plate-forme JuxMem : support pour un service de partage de données sur la grille. *In : Actes des Rencontres francophones du parallélisme (RenPar 15)*, pp. 145–152. – La Colle-sur-Loup, 2003.
4. Antoniu (Gabriel), Bougé (Luc) et Lacour (Sébastien). – Making a DSM consistency protocol hierarchy-aware : An efficient synchronization scheme. *In : Proceedings of the 3rd IEEE/ACM International Conference on Cluster Computing and the Grid (CCGrid '03)*. pp. 516–523. – Tokyo, Japan, 2003.
5. Bassi (Alessandro), Beck (Micah), Fagg (Graham), Moore (Terry), Plank (James), Swany (Martin) et Wolski (Rich). – The Internet Backplane Protocol : A study in resource sharing. *In : Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '02)*. pp. 194–201. – Berlin, Germany, 2002.
6. Bershad (Brian N.), Zekauskas (Mattew J.) et Sawdon (Wayne A.). – The Midway distributed shared memory system. *In : Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93)*, pp. 528–537. – Los Alamitos, CA, 1993.
7. Castro (Miguel) et Liskov (Barbara). – Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, vol. 20, N° 4, 2002, pp. 398–461.

8. Chandra (Tushar Deepak) et Toueg (Sam). – Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, vol. 43, N° 2, 1996, pp. 225–267.
9. Foster (Ian) et Kesselman (Carl). – Globus : A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, N° 2, 1997, pp. 115–128.
10. Gharachorloo (Kourosh), Lenoski (Daniel), Laudon (James), Gibbons (Phillip), Gupta (Anoop) et Hennessy (John). – Memory consistency and event ordering in scalable shared-memory multiprocessors. *In : Proceedings of the 17th International Symposium Computer Architecture (ISCA '90)*, pp. 15–26. – Seattle, WA, 1990.
11. Guerraoui (Rachid) et Schiper (André). – Software-based replication for fault tolerance. *IEEE Computer*, vol. 30, N° 4, 1997, pp. 68–74.
12. Iftode (Liviu), Singh (Jaswinder Pal) et Li (Kai). – Scope consistency : A bridge between release consistency and entry consistency. *In : Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pp. 277–287. – Padova, Italy, 1996.
13. Kermarrec (Anne-Marie), Cabillic (Gilbert), Gefflaut (Alain), Morin (Christine) et Puaut (Isabelle). – A recoverable distributed shared memory integrating coherence and recoverability. *In : Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems (FTCS 25)*, pp. 289–298. – Pasadena, California, 1995.
14. Protić (Jelica), Tomasević (Milo) et Milutinović (Veljko). – *Distributed Shared Memory : Concepts and Systems*. – IEEE, 1997.
15. Schneider (Fred B.). – Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys*, vol. 22, N° 4, 1990, pp. 299–319.
16. Zhou (Yuanyuan), Iftode (Liviu) et Li (Kai). – Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. *In : Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pp. 75–88. – Seattle, WA, 1996.
17. The JXTA project. – `http://www.jxta.org/`.