Modèles et protocoles de cohérence des données en environnement volatil

Loïc Cudennec

Modèles et protocoles de cohérence des données en environnement volatil

Loïc Cudennec
Loic.Cudennec@irisa.fr

Superviseurs : **Gabriel Antoniu, Luc Bougé, Sébastien Monnet**
{Gabriel.Antoniu,Luc.Bouge,Sebastien.Monnet}@irisa.fr

IRISA, Projet Paris
Février — juin 2005

Rapport de stage du Master de Recherche en Informatique de l'IFSIC, université de Rennes I.

Ce rapport correspond au stage de recherche du *Master Recherche Informatique* de l'IFSIC (*Institut de Formation Supérieure en Informatique et Communication*) ainsi qu'au projet de fin d'étude (PFE) en école d'ingénieur à l'*INSA de Rennes* (*Institut National des Sciences Appliquées*). Il s'est déroulé au sein du projet PARIS (*Programming pArallel and distRibuted systems for large scale numerical simulation applicationS*) de l'IRISA. L'IRISA (*Institut de Recherche en Informatique et Systèmes Aléatoires*) est une unité mixte de recherche dont les partenaires sont l'INRIA (*Institut National de Recherche en Informatique et Automatique*), le CNRS (*Centre National de la Recherche Scientifique*), l'Université de Rennes 1 et l'INSA de Rennes. L'antenne Bretagne de l'ENS Cachan (*Ecole Normale Supérieure*) est partenaire de l'IRISA dans le projet PARIS.

Le présent stage s'intitule "Modèles et protocoles de cohérence des données en environnement volatil" et s'est déroulé sous la responsabilité de Gabriel Antoniu, chargé de recherche à l'INRIA, Luc Bougé, Professeur à l'antenne Bretagne de l'ENS Cachan et Sébastien Monnet, doctorant. Je tiens à remercier mes encadreurs ainsi que Mathieu Jan, pour l'accueil qu'ils m'ont réservé.

Table des matières

Chapitre 1

Introduction

De tout temps l'homme a voulu construire l'ordinateur le plus puissant du monde. Une méthode simple vouée à la quête de ce Graal consiste à mettre en commun les ressources de sites géographiquement distants. Les systèmes distribués utilisent donc la puissance de calcul et l'espace de stockage de plusieurs unités de traitement interconnectées par des réseaux. Leur taille peut aller du plus petit système composé de deux entités aux millions de nœuds que forment le réseau internet. Déployer des applications sur les systèmes distribués n'est pas sans poser de nouveaux problèmes autour des méthodes de programmation classiques. L'ordinateur tel qu'il est défini en 1946 par le mathématicien Von Neumann est principalement composé d'une unité de commande ainsi que d'une mémoire locale contenant les instructions et les données. Dans ce modèle, le partage de ressources avec un autre ordinateur ne s'effectue pas de manière transparente et les instructions d'accès aux données distantes ne sont pas les mêmes que pour les données locales. La tâche du programmeur se voit alors complexifiée par la gestion des messages de communication propres aux systèmes répartis.

Pour faciliter le développement d'applications sur les systèmes distribués, de nombreuses recherches portent sur les mécanismes permettant de donner l'illusion au programmeur que plusieurs unités de traitement n'en forment qu'une : c'est le système à image unique. Dans un tel système, le programmeur ne peut plus faire d'hypothèses sur l'emplacement physique des processus et des données qu'il manipule. Ce rapport s'intéresse exclusivement à la gestion des données. Il est donc du ressort du système d'effectuer pour le programmeur la localisation et le transfert des données et ce, de manière transparente. Afin d'améliorer les performances d'accès, les données sont couramment répliquées sur les sites effectuant une lecture ou une écriture. L'affaire se complique lorsque deux processus distants, dont au moins un écrivain, veulent accéder à une même donnée. Des scénarios équivalents entraînent souvent la création, à un instant donné, de versions différentes d'une donnée sur des sites différents. La question de la cohérence des données est alors soulevée.

Ces problématiques ont été étudiées dans le cadre des systèmes à mémoire virtuellement partagée (MVP) qui offrent la possibilité de partager des données sur des grappes de calculateurs de manière transparente. Ces systèmes sont adaptés à des environnements regroupant quelques dizaines de nœuds. A cette échelle, un nombre important de modèles et protocoles de cohérence ont pu être mis en œuvre de telle sorte que les réplicas soient gérés efficacement. Si ce modèle a fait ses preuves à petite échelle, il faut admettre qu'il n'est pas adapté aux environnements distribués de taille plus conséquente.

1.1 Le partage des données sur les grilles de calculateurs

Depuis quelques années déjà, les besoins en puissance de calcul ont poussé les scientifiques à fédérer les ressources de leurs universités respectives au sein du concept de grille. La grille, telle qu'elle est définie dans [20], consiste en une infrastructure composée de ressources informatiques hétérogènes interconnectées par un réseau. L'idée est de proposer un *super-ordinateur* en réunissant la puissance de calcul et l'espace de stockage de sites géographiquement dispersés. Ces sites peuvent appartenir à des institutions différentes et être connectés ou déconnectés de la grille en fonction des centres d'intérêt du moment. La taille d'un tel système est de l'ordre de la dizaine de milliers de nœuds. Les applications liées au calcul scientifique et, d'une manière générale, massivement parallèles, peuvent tirer parti des performances théoriques d'un tel environnement.

La gestion des données sur la grille est l'un des grands sujets de réflexion en informatique qui soit donné à la communauté scientifique. Malgré cet engouement, aucune solution sophistiquée n'a été jusqu'à aujourd'hui proposée. L'approche la plus répandue pour partager des données sur une grille consiste à transférer *explicitement* celles-ci entre le client et les serveurs de calcul. A titre d'exemple, la plate-forme Globus [19] utilise le protocole GridFTP [3] qui requiert des informations explicites sur la localisation et le transfert des données. Le système IBP [13] permet à un utilisateur de réserver des espaces mémoire sur Internet et de les utiliser pour optimiser les transferts à travers le réseau. Là encore la localisation et le transfert des données doivent être effectués de manière *explicite*. Il faut de plus remarquer que dans ces systèmes, les problèmes de *cohérence* des données sont laissés sous l'entière responsabilité de l'utilisateur.

1.2 JuxMem : un service de partage de données pour les grilles

La gestion explicite de la localisation des données par le programmeur s'annonce comme une limitation de l'utilisation efficace des grilles. La mise en place de services de partage des données offrant des accès transparents est une étape ineluctable pour ce type d'environnement. Ces services doivent prendre principalement en charge la localisation et le transfert des données. Des stratégies adaptées de réplication doivent alors être associées à des protocoles de cohérence pour assurer les propriétés de persistance et de cohérence des données. Ces mécanismes fonctionnent sur des grilles dynamiques et supportent donc la volatilité caractérisée par des connexions et déconnexions de nœuds.

JuxMem est un service de partage de données modifiables pour la grille développé par le projet PARIS (*Programming pArallel and distRibuted systems for large scale numerical simulation applicationS*). Les utilisateurs de cette plate-forme accèdent à une donnée partagée grâce à un identifiant global unique. La localisation et le transfert des données sont complètement à la charge du service. Les propriétés offertes par JuxMem sont la persistance des données, la tolérance aux fautes et la cohérence des données. La persistance permet de conserver les données dans le système entre deux utilisations de celles-ci. Les performances de l'application sont améliorées grâce à la diminution des échanges réseaux qui en résulte. Les arrivées et départs imprévus de ressources physiques dans la grilles sont pris en compte dans JuxMem par l'utilisation de techniques de réplication des données et de mécanismes de détection des défaillances. Les données partagées manipulées dans la grille sont modifiables : différents sites peuvent décider de mettre à jour le réplica le plus proche. La cohérence des réplicas

est assurée par des protocoles de cohérence spécialisés dans le contexte de la grille. JuxMem s'affiche comme une plate-forme générique sur laquelle divers protocoles de cohérence tolérants aux fautes sont experimentés.

Le modèle pair-à-pair (PàP) réussi à fédérer des centaines de milliers de nœuds essentiellement pour du partage de données. Les sites participants jouent à la fois les rôles de client et serveur, ce qui se traduit par une décentralisation forte des responsabilités. Malheureusement si le PàP est très bien adapté au passage à l'échelle et à la volatilité des nœuds, la plupart des applications actuellement proposées ne traitent que du partage de données non-modifiables. L'approche de JuxMem est de combiner les paradigmes MVP et PàP pour proposer un service prenant en compte la cohérence des données dans un environnement dynamique.

1.3 Un scénario : observation de données partagées

Une classe d'applications portée sur les grilles de calculateurs est basée sur le couplage de code. C'est-à-dire qu'un calcul global est séparé en plusieurs codes s'executant en parallèle sur des grappes de calculateurs différentes. Même si la distribution des codes est effectuée de telle sorte que les échanges réseaux inter-grappes soient minimes, des données doivent être partagées entre les sites. Elles peuvent constituer des données d'entrée ou de sortie de chacun des sous-calculs. Un cas fréquent est le chaînage des sites : ce qu'une grappe produit est utilisé en paramètre du calcul sur une autre grappe.

Le couplage de code est utilisé dans le calcul scientifique haute performance. Ces calculs peuvent durer des jours, voir des mois et il n'est pas souhaitable d'en attendre la fin pour vérifier que tout s'est bien passé. L'observation des données partagées dans la grille peut renseigner sur l'état du calcul. Elle s'effectue à partir d'un site distant. Son utilisation directe est l'affichage de l'avancement du calcul ou l'analyse de traces d'execution. Observer un calcul doit cependant s'effectuer en causant le moins d'intéractions avec celui-ci, et donc de retardements.

La plate-forme JuxMem s'adapte parfaitement comme service de partage de données pour les applications basées sur le couplage de code. Elle permet, dans le cadre du protocole de cohérence, d'effectuer des lectures et des écritures sur les variables partagées. Ceci est possible non seulement sur les sites supportant un sous-calcul mais aussi sur les sites chargés de l'observation du calcul. L'objectif de ce stage est d'améliorer le comportement du protocole de cohérence lorsque des observateurs sont utilisés. La contribution principale présentée dans ce rapport est de proposer une extension du protocole de cohérence qui introduit la possibilité d'effectuer des lectures en parallèle d'une écriture. Ceci est possible si l'observateur accepte d'exploiter des données dont la version est considérée comme ancienne. Une implémentation de cette stratégie a été effectuée dans le cadre de la plate-forme JuxMem. Des évaluations de performances ont été menées pour montrer que cette solution améliore la vitesse d'accès des observateurs sans dégrader celle des autres sites.

Chapitre 2

Partage de données dans les systèmes distribués : l'existant

Ce chapitre présente l'état de l'art dans le domaine du partage de données en environnement distribué. Les concepts de mémoire virtuellement partagée (MVP), modèle de cohérence, réseau pair-à-pair et tolérance aux fautes y sont développés afin de comprendre les mécanismes pouvant être mis en jeu sur une architecture de type grille. Le service JuxMem de partage de données pour les grilles est introduit comme une synthèse des systèmes à MVP et pair-à-pair. Le scénario d'observation de couplage de code est enfin discuté avec la notion de version d'une donnée.

2.1 Systèmes à MVP : Accès transparent aux données

Les systèmes à MVP[1], historiquement introduits par Kai Li [27] donnent l'illusion au programmeur que plusieurs mémoires physiquement distribuées n'en forment qu'une. Dans ce contexte, les mécanismes d'adressage classiques de la mémoire sont conservés et il est du ressort de la MVP d'aller chercher les données, qu'elles soient situées localement sur le site ou distantes. Il n'est alors pas nécessaire d'utiliser des primitives de lecture et d'écriture spécialisées car la mémoire détecte les accès distants et se charge d'effectuer les transferts de données. Les échanges de messages induits par la gestion de la mémoire distribuée entre les différents sites sont masqués par le système à MVP. La transparence de localisation des objets obtenue permet de conserver le modèle classique de programmation par mémoire partagée.

2.1.1 Modèles et protocoles de cohérence

Le partage des données entre plusieurs processus exhibe le problème d'ordonnancement des opérations de lecture et d'écriture, c'est à dire la vision que chaque processus peut avoir de l'entrelacement des instructions. Ceci est d'autant plus difficile à appréhender que les données peuvent être répliquées sur plusieurs sites. La mise en place d'une horloge globale synchronisée sur tous les sites n'est pas réalisable et la technique de l'horodatage ne permet alors plus d'établir une trace d'exécution unique.

La notion centrale dans un système distribué est le modèle de cohérence utilisé. Un modèle de cohérence s'affiche comme un contrat passé entre le système et le programmeur. Il

[1] Mémoire virtuellement partagée.

définit les critères déterminant la valeur retournée par une lecture en fonction des écritures précédentes. Il existe plusieurs modèles de cohérence appartenant aux classes de cohérence forte et relachée. Les modèles de cohérence forte ont été introduits les premiers et facilitent la vision des données fournie au programmeur. Ils imposent des contraintes importantes entre la dernière lecture et la prochaine écriture d'une donnée. Cette stratégie entraîne l'implémentation de protocoles complexes et extrêmement couteux en communications réseau. Afin de gagner en efficacité, plusieurs modèles de cohérence ont donné lieu à différents protocoles de cohérence.

2.1.2 Modèles de cohérence forte

Les modèles de cohérence forte sont caractérisés pas des contraintes fortes entre la dernière écriture et la prochaine lecture.

Le modèle strict (*atomic consistency*) est un modèle idéal où chaque lecture rend la dernière valeur écrite dans la donnée. Dans les systèmes distribués, le protocole associé nécessite l'utilisation d'une horloge globale, ce qui rend son implémentation impossible.

Le modèle séquentiel (*sequential consistency (SC)*) formalisé par Lamport en 1979 assure que chaque site voit toutes les opérations dans le même ordre. Les premières MVP, comme IVY [26], utilisent ce modèle de cohérence. [33] propose de combiner les modèles strict et séquentiel en offrant deux primitives de lecture en fonction du modèle à utiliser pour la donnée considérée.

Le modèle causal (*causal consistency*) se base sur la relation de causalité introduite dans [25] pour déterminer un ordre entre les écritures. De nombreuses applications tolèrent que deux évènements ne soient pas vus dans le même ordre sur tous les sites. Le modèle causal permet alors de lier certains évènements entre eux par un ordre bien fondé tout en relâchant les contraintes sur les évènements indépendants.

2.1.3 Modèles de cohérence relâchée

Les modèles de cohérence relâchée ont été introduits afin de diminuer le nombre d'échanges réseau induit par les protocoles de cohérence forte. Ils tirent parti du fait que les applications distribuées imposent déjà un ordre sur les accès mémoire par l'utilisation explicite de mécanismes de synchronisation. La technique employée consiste à retarder les mises à jour jusqu'aux prochains points de synchronisation, diminuant ainsi le nombre de communications, ce qui améliore les performances générales.

La cohérence faible (*weak consistency*) fait la distinction entre les accès ordinaires à la mémoire et les accès synchronisés. Seuls les accès synchronisés garantissent la cohérence de la mémoire partagée par l'utilisation d'objets de synchronisation comme les verrous ou les barrières. Ce modèle garantit que la mémoire est cohérente à chacun des points de synchronisation pendant lequels toutes les informations sont mises à jour.

La cohérence à la libération (*eager release consistency (ERC)*) améliore la cohérence faible en ne mettant à jour que les données modifiées entre deux synchronisations. Ce modèle utilise le principe de section critique gérée par un verrou et délimitée par les primitives *acquire* et *release*. La cohérence de la mémoire est assurée par la propagation

vers les autres sites des modifications effectuées sur la donnée dans la section critique. La particularité de ce modèle est que la mise à jour intervient lors de l'appel à la primitive *release*. Ce protocole génère des communications inutiles vers des sites n'effectuant par la suite aucun accès sur les données mises à jour. Le protocole associé est mis en œuvre dans le système à MVP Munin [15].

La cohérence à la libération paresseuse (*lazy release consistency (LRC)*) est une version plus relachée d'*ERC* qui tente de réduire les communications inutiles de ce dernier. Une liste des données modifiées (*write notice (wn)*) est envoyée au site effectuant le prochain appel à la primitive *acquire*. Les modifications ne s'appliquent alors que sur ce site et uniquement lors de l'accès en lecture ou écriture à une donnée déclarée modifiée dans *wn*. Le protocole a été proposé avec le système à MVP Treadmarks [4]. Une variante de ce modèle peut-être obtenue en utilisant un nœud de référence chargé de la gestion d'un sous-ensemble des données. Cette technique est appellée *home-based lazy release consistency (HLRC)* [35].

La cohérence à l'entrée (*entry consistency (EC)*) proposée par le système à MVP Midway [14] associe à chaque variable partagée un objet de synchronisation comme le verrou. Cette stratégie permet de ne transférer que des mises à jour en rapport avec la donnée, et non plus un morceau de mémoire. Un autre avantage est qu'une écriture interdit uniquement l'accès à la donnée et non pas l'accès à une zone mémoire où d'autres données peuvent être stockée. L'établissement de la relation entre donnée et verrou est cependant laissée à la charge du programmeur. La cohérence à l'entrée fait la distinction entre les accès en lecture et les accès en écriture grâce à l'utilisation de deux verrous. Si l'écrivain reste unique, les lecteurs sont multiples et peuvent donc lire en parallèle. Une lecture ne peut cependant pas avoir lieu en même temps qu'une écriture sur la même donnée.

La cohérence de portée (*scope consistency (ScC)*) [22] reprend le principe de l'*EC* et tente d'éviter au programmeur d'effectuer lui-même l'association entre verrous et données. Cette technique se base sur les instructions de synchronisation déjà présentes dans le programme. Lors de l'acquisition d'un verrou par un site, seules les modifications effectuées dans les portées correspondant à ce verrou sont visibles.

2.1.4 Modèles pour grappes de grappes

L'interconnexion de grappes entraîne l'utilisation de supports réseaux hétérogènes. Les communications inter-grappes sont par nature plus coûteuses que celles disponibles sur les liens intra-grappes. L'objectif principal des protocoles dédiés à ce type d'environnement est de diminuer autant que possible les échanges inter-grappes. Le protocole *clustered lazy release consistency (CLRC)* [12] met en œuvre un système de cache au sein de chaque grappe, limitant ainsi les requêtes sur les liaisons entre grappes.

Une autre stratégie est proposée par le protocole *hierarchy-aware home-based release consistency (H-HBRC)* [9]. Ce protocole minimise le nombre de messages sur les liens lents (inter-grappes) et privilégie les requêtes d'accès à un verrou en provenance des sites les plus proches en terme de performances réseau (intra-grappes).

2.2 Systèmes PàP : dynamicité et passage à l'échelle

Les systèmes à MVP ont prouvé leur efficacité à partager des données sur des environnements composés de quelques dizaines de nœuds. Avec la multiplication du matériel informatique interconnecté par des réseaux comme internet, l'envie de porter des applications distribuées de partage de données sur plusieurs centaines de milliers de nœuds ouvre la porte à de nouveaux problèmes. Dans ces conditions, l'algorithmique mise en œuvre par les MVP montre ses limites d'extensibilité. A titre d'exemple, l'invalidation des toutes les copies d'une donnée modifiée n'est pas une stratégie raisonnable lorsque le nombre de participants devient trop important. C'est en tenant compte de ces nouvelles propriétés que le modèle PàP[2] réussi le pari de passer à l'échelle.

2.2.1 Généralités

L'augmentation du nombre de participants dans une application distribuée calquée sur le modèle client-serveur influe directement sur les risques de congestion du ou des sites chargés de rendre un service. Les défaillances des serveurs entraînent bien souvent l'interruption complète du ou des services supportés.

Assurer le passage à l'échelle, tel pourrait alors être le leitmotiv des systèmes PàP où le principe de décentralisation est poussé à l'extrême. Chaque participant est client et serveur à la fois et n'intéragit qu'avec un petit groupe de pairs, définissant la notion de connaissance locale du système. Cette politique permet de décentraliser la charge et d'utiliser les ressources disponibles sur les sites jusqu'alors considérés comme de simples clients. Le même service peut ainsi être rendu par plusieurs nœuds, offrant ainsi des propriétés de tolérance aux fautes.

Les contacts entre les sites s'établissent au gré des connexions et déconnexions ainsi que des centres d'intérêt définis par les applications. Les systèmes PàP sont caractérisés par :

- la propriété de passage à l'échelle qui garantit que les performances du système ne sont pas dramatiquement affectées par l'augmentation du nombre de ses participants,
- l'extensibilité qui assure que l'ajout de nouvelles ressources améliore l'espace de stockage ou la puissance de calcul,
- l'auto-organisation des nœuds pour faire face à la volatilité de l'environnement et ainsi proposer des mécanismes de tolérance aux fautes,
- la transparence de la localisation, masquant aux applications le cheminement physique de la recherche d'une donnée sur le réseau.

Si il faut reconnaître que les avantages énumérés précédemment sont séduisants, le PàP souffre tout de même de plusieurs défauts inhérents à la dimension du système.

- A cette échelle il n'est désormais plus concevable d'utiliser des outils de synchronisation comme les barrières globales, au risque de perdre la propriété de vivacité.
- La mise en œuvre des systèmes PàP s'effectue au dessus de réseaux hétérogènes présentant des performances inégales. Le matériel supportant les nœuds est bien souvent du même acabit que le simple ordinateur personnel et présente un faible MTBF[3]. Les procédures de reconfiguration du réseau, et notamment des tables de routage, peuvent dégrader les performances en cas de volatilité trop élevée.

[2] Pair-à-pair.

[3] Mean time between failure.

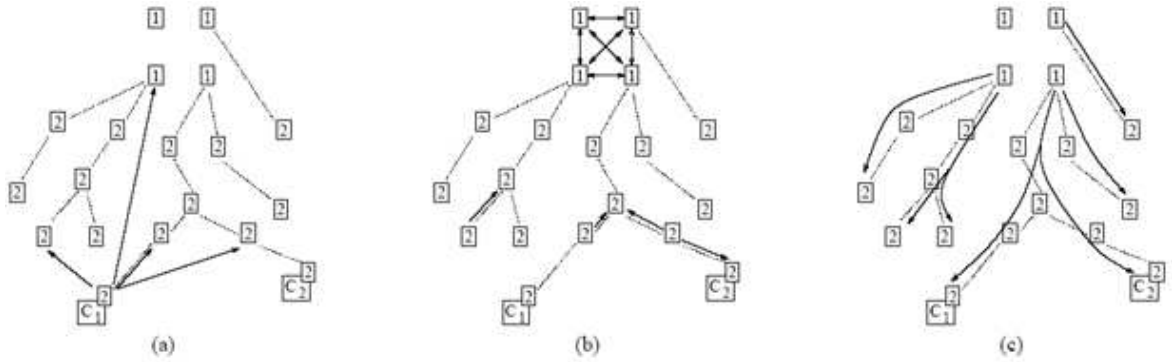


FIG. 2.1 – Les mises à jour dans OceanStore. (a) Le fichier de mises à jour est envoyé par le client C_1 vers une copie primaire ainsi qu'à un ensemble de réplicas connus et choisis aléatoirement. (b) Pendant qu'un consensus sur l'ordre des mises à jour est réalisé par les copies primaires, les copies secondaires poursuivent la dissémination des mises à jour et décident ou non de les appliquer localement. (c) Lorsque le consensus est établi, le résultat de la mise à jour est propagé le long de l'arbre de diffusion.

– Enfin, si la localisation des données s'effectue de manière transparente, il ne faut pas oublier que celles-ci transitent bien souvent par des chemins non-sécurisés. La mise en place de protocoles cryptographiques, du chiffrement des données ainsi que de serveurs d'authentification va à l'encontre des raisons pour lesquelles le PàP s'avère performant à grande échelle.

2.2.2 Gestion de la cohérence

La plupart des systèmes PàP sont utilisés pour du stockage de données non-modifiables, éludant ainsi les concurrences d'écriture et la mise en place des protocoles de cohérence qui en découlent. Ivy [31], OceanStore [23] et Pastis [32] sont des systèmes pair-à-pair de gestion de fichiers qui donnent la possibilité de modifier les données. Tous partent du principe que la mise à jour d'un bloc revient à en insérer un nouveau dans le système.

Dans Ivy, chaque écrivain ajoute la nouvelle version d'une donnée en déclarant un nouveau bloc aussitôt chainé dans un journal recensant toutes les versions produites par cet utilisateur. Chaque bloc est caractérisé par son numéro de version (localement incrémenté sur le site propriétaire) et le vecteur d'horloge[4] du site. Les journaux ne sont accessibles en écriture que pour le propriétaire mais restent lisiblent par tout le monde. On est donc bien en présence d'un système à écrivains multiples, mais chacun dans son journal.

OceanStore utilise un ensemble de machines dédiées à la gestion des mises à jour des fichiers dénommé *Inner Ring*. Celui-ci satisfait les requêtes des clients et décide de l'ordre des mises à jour en utilisant un protocole de type consensus byzantin[5]. Le système de réplicas met en jeu deux types de copies : 1) des copies primaires du fichier sont placées sur l'*Inner*

[4]Le vecteur d'horloge défini par Mattern [28] et Fidge [17] est local à chaque site et se compose de la connaissance des numéros de version de autres sites.

[5]Le consensus byzantin permet à plusieurs sites de décider d'une même valeur et ce, en dépit de quelques sites malicieux désireux de semer la confusion dans le protocole.

Ring et représentent la version la plus récente disponible sur le système et 2) des copies secondaires, caractérisées par un numéro de version et disséminées sur les sites participant à *Oceanstore*. Le protocole de mise à jour est expliqué dans la *figure 2.1*. Dans le cas de la concurrence des écritures et si le consensus échoue, les modifications peuvent être annulées. Le protocole de cohérence autorise par ailleurs la mise à jour des données sur les copies secondaires sans aucune concertation.

Pastis manipule des données dont la structure est fortement inspirée par le système de fichiers Unix (UFS). Deux modèles de cohérence sont envisagés : 1) le *close-to-open* s'apparente à l'*ERC* (*section 2.1.3*) en ne propageant les modifications sur le réseau que lors de la fermeture du fichier et 2) le très relâché *read-your-writes* qui ne garantit à un site que de voir ses propres modifications. Lors de l'apparition d'un conflit en écriture, c'est le dernier écrivain qui voit ses modifications prises en compte. L'estampillage des écritures s'effectue grâce aux horloges locales des sites faiblement synchronisées par un serveur de temps *NTP*.

2.3 JuxMem : un système distribué tolérant aux fautes

Le problème de partager des données modifiables dans les environnements distribués a été largement étudié ces quinzes dernières années à travers le concept des systèmes à MVP (*section 2.1*). Les MVP apportent la transparence au niveau de la gestion des données, proposent l'implémentation de nombreux modèles de cohérence ainsi que des mécanismes de tolérance aux fautes. Cependant, pour des raisons de performances, leur déploiement n'est réalisable qu'à petite échelle, sur des configurations de quelques dizaines de nœuds. Le modèle PàP (*section 2.2*) quant à lui a prouvé son efficacité à passer à l'échelle en réussissant à rassembler sur le réseau internet, essentiellement pour du partage de données, plusieurs centaines de milliers de nœuds. Le PàP est caractérisé par un très bon comportement face à la volatilité, offrant les propriétés de persistance des données et de tolérance aux fautes. Cependant, la plupart des systèmes PàP portent sur la gestion de données non-modifiables.

2.3.1 Approche : partage transparent des données

JuxMem [6, 7, 10, 11] est un service de partage de données pour la grille dont la particularité est d'être défini comme un système hybride inspiré des modèles MVP et PàP. A la MVP, JuxMem emprunte la possibilité de modifier les données et d'intégrer des modèles de cohérence, au PàP, la gestion de réseaux dynamiques. Mais JuxMem est avant tout une plate-forme générique sur laquelle divers protocoles de cohérence et stratégies de tolérance aux fautes peuvent être implémentés. L'aspect dynamique de JuxMem est assuré par l'utilisation de la plate-forme générique JXTA [1] qui fournit les blocs de base pour l'intégration de services pait-à-pair (arrivée d'un nœud, découverte, formation de groupe...etc).

2.3.2 Architecture de JuxMem

L'architecture de JuxMem reflète l'architecture matérielle de la grille, à savoir un modèle hiérarchique composé de grappes de calculateurs interconnectées par des réseaux. La figure 2.2 met en valeur les différents rôles attribués dans JuxMem. Les ensembles A et B peuvent correspondre à des sites placés sur deux grappes physiques distantes, ce sont des groupes *cluster*. Chaque groupe *cluster* est composé de nœuds spécialisés dans le stockage

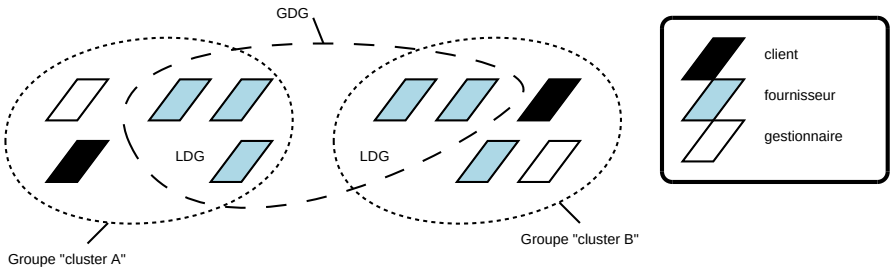


FIG. 2.2 – JuxMem : architecture. LDG : *Local Data Group*, GDG : *Global Data Group*.

de données : les *fournisseurs*. Dans chaque groupe *cluster* se trouve un nœud chargé de gérer la mémoire mise à la disposition de la grille : le *gestionnaire*. Enfin, tout nœud utilisant le service de localisation et d'accès aux données joue le rôle de *client*. Ce sont les clients qui supportent les différentes tâches de l'application distribuée déployée par l'utilisateur. Il est biensûr possible pour un site de cumuler plusieurs rôles JuxMem.

Le système associe à chaque bloc de données stocké un groupe de données : c'est le *groupe de données global* (*global data group (GDG)*). Ce groupe est formé d'un ensemble de fournisseurs hébergeant une copie du bloc. Les fournisseurs peuvent être choisis dans plusieurs *cluster group*, ce qui facilite la dissémination des blocs sur plusieurs grappes physiques. L'ensemble des fournisseurs d'un bloc situés dans un même *cluster group* forment un *groupe de données local* (*local data group (LDG)*).

2.3.3 Tolérance aux fautes

La réplication est introduite dans JuxMem à des fins de disponibilité et de persistance des données. Ces données étant modifiables, des protocoles sont mis en œuvre pour assurer la cohérence des copies. La difficulté provient de la volatilité et des défaillances pouvant toucher les nœuds.

Ces défaillances sont de deux types. 1) Les nœuds peuvent être sujets à des pannes franches, c'est à dire qu'ils cessent toute activité à partir d'un temps donné (modèle *fail-silent*). 2) Les liens réseaux peuvent perdre certains messages mais pas tous (modèle *fair-lossy*). Le modèle de temps considéré permet d'utiliser l'existence de bornes sur la vitesse d'execution des processus et le temps de transmission des messages. Ce modèle, introduit dans [16] est dit partiellement synchrone et semble raisonnable dans le contexte de la grille.

Les groupes de données (LDG et GDG) sont des ensembles de réplicas qu'un certain nombre d'abstractions issues du monde de l'algorithmique distribuée permet de gérer. Le concept de *communication de groupe* donne la possibilité de représenter un ensemble de nœuds en une seule entité. Il est notamment possible d'envoyer un message à cette entité virtuelle avec la garantie que l'un des deux cas suivants soit vrai : 1) le message est délivré à tous les nœuds non-fautifs ou 2) le message est délivré à aucun d'entre-eux.

L'une des difficultés de la réalisation de la communication de groupe est de réussir à assurer que les messages soient délivrés dans le même ordre sur tous les sites. C'est le rôle de la *diffusion atomique*. Cette propriété implique que les participants s'accordent sur un ordre commun d'arrivée. La solution utilisée dans JuxMem est basée sur le *consensus*.

Le protocole du consensus permet à un ensemble de nœuds de décider d'une valeur

commune. Pour ce faire, chaque site propose une valeur et le protocole garantit que 1) tous les nœuds non-fautifs décident une valeur, 2) la valeur décidée est une valeur proposée par un nœud participant au consensus et 3) tous les nœuds non-fautifs décident la même valeur. Dans le cadre de la diffusion atomique, le consensus est alors utilisé pour décider d'un ordre d'arrivée des messages.

Le problème du consensus dans un système partiellement asynchrone peut être résolu par des détecteurs de défaillances [16]. Chaque nœud tient à jour une liste des nœuds qu'il suspecte d'avoir crashé. Cette liste peut contenir des sites par erreur. En revanche, un site fautif sera obligatoirement suspecté au bout d'un temps fini. JuxMem utilise une version hiérarchique de ces détecteurs de défaillances.

2.4 Observation des données

La plate-forme JuxMem offre les propriétés de transparence de localisation et de persistance des données. Ces propriétés intéressent tout particulièrement les applications basées sur le couplage de code dans les grilles, permettant de prendre en compte l'aspect dynamique de l'architecture. Ces applications concernent principalement le calcul scientifique hautes performances. A titre d'exemple, la conception d'un satellite peut s'effectuer dans un environnement complètement distribué : les moteurs de rendu graphique et physique peuvent être supportés par des grappes de calculateurs géographiquement distantes. Certaines données restent cependant partagées et soumises à des lectures-écritures des deux sites.

Observer les données partagées permet de surveiller le calcul, d'en connaître l'état d'avancement ou tout simplement de vérifier son bon déroulement. Cette fonctionnalité n'apporte cependant rien au calcul et doit rester la moins intrusive possible. Elle doit aussi s'effectuer dans des temps d'accès très courts, afin de détecter à temps une anomalie ou d'obtenir un affichage de contrôle fluide. Ces deux contraintes semblent antagonistes : lire la donnée rapidement peut provoquer la perturbation des accès en écriture et ainsi ralentir le calcul. A l'inverse, limiter les intéractions avec le système de gestion de la donnée partagée peut conduire à une famine pour l'observateur.

Une approche visant à conserver les propriétés de vitesse de lecture et d'observation silencieuse pour le système est de relâcher les contraintes sur le modèle de cohérence de la donnée. Ainsi la version de la donnée retournée par une lecture de l'observateur peut ne plus correspondre à la donnée la plus récente introduite par le dernier écrivain. De nombreuses applications tolèrent l'utilisation de données considérées comme périmées, à titre d'exemples les pages internet lues par les navigateurs peuvent correspondre à des pages périmées récupérées dans un des nombreux serveurs-cache du réseau. Une requête dans un moteur de recherche peut rendre des résultats collectés il y a plus d'une semaine sans inquiéter l'utilisateur. Un système de version est donc mis en place pour attribuer à chaque donnée un numéro de version.

La technique du *versioning* est utilisée dans l'univers des bases de données depuis quelques années déjà. Le modèle de gestion *multi-versions* des données impose au système de conserver une partie ou la totalité des versions produites pour une donnée [34]. Le nombre de versions sauvegardées peut aussi être borné : les écrivains doivent attendre que la plus vieille version détenue par le système soit utilisée et supprimée pour pouvoir en créer une nouvelle. Une autre approche proposée dans [30] consiste à différencier les *transactions* de

mise à jour, des transactions effectuant des lecture-seules. La particularité de ces dernières est qu'elles ne se synchronisent pas pour lire la donnée. L'accès est donc rapide et non perturbateur pour les autres sites. Enfin, le modèle de cohérence temporel [24] assure que les copies d'une donnée soient mises à jour dans une fenêtre de temps bornée.

Chapitre 3

Analyse d'un protocole de cohérence hiérarchique tolérant aux fautes

Dans le cadre du scénario d'observation de couplage de code, les performances d'accès à la donnée sont largement influencées par le protocole de cohérence. Il est en effet de son ressort d'indiquer lorsqu'une donnée doit être transférée, ce qui se répercute sur la charge réseau et l'efficacité de l'application. Le premier modèle de cohérence des données implémenté dans JuxMem [1] est la cohérence à l'entrée (*entry consistency (EC)*). Le protocole associé est adapté pour tirer parti de l'architecture hiérarchique de JuxMem. Augmenter les performances de ce protocole dans le contexte de l'observateur requiert l'analyse des mécanismes mis en œuvre. La première contribution de ce stage est donc la formalisation du protocole à l'aide d'automates d'états et de diagrammes séquentiels.

3.1 Cohérence à l'entrée

Le modèle de cohérence à l'entrée est un modèle de cohérence relâchée. Il a été proposé par le système à mémoire virtuellement partagée Midway [14]. Le principe est d'associer aux variables partagées un objet de synchronisation comme un verrou. Cette déclaration explicite est laissée à la charge du programmeur. La cohérence d'une donnée partagée n'est garantie que lors de l'acquisition de son verrou associé.

L'une des particularités de la cohérence à l'entrée est de différencier les accès en lecture des accès en écriture en offrant deux primitives spécialisées.

- La primitive *acquire* permet d'obtenir le verrou en mode exclusif. Elle est principalement utilisée pour garantir qu'un écrivain soit le seul accesseur de la donnée.
- La primitive *acquireR* permet à plusieurs lecteurs d'accéder à la donnée de façon concurrentielle.

L'appel à ces deux fonctions entraîne la mise à jour de la donnée sur le site concerné et garanti d'en obtenir la version la plus à jour. La primitive *release* permet de relâcher le verrou, qu'il soit pris en lecture ou en écriture.

La figure 3.1 illustre un scénario de comportement avec ce modèle de cohérence. Les processus $P0$, $P1$ et $P2$ se partagent les variables X et Y. Après avoir obtenu le verrou en mode exclusif, $P0$ écrit dans la donnée Y. L'obtention de ce verrou implique que les modifications

[1] La conception a été effectuée par Jean-Francois Deverge lors de son stage de DEA en 2004.

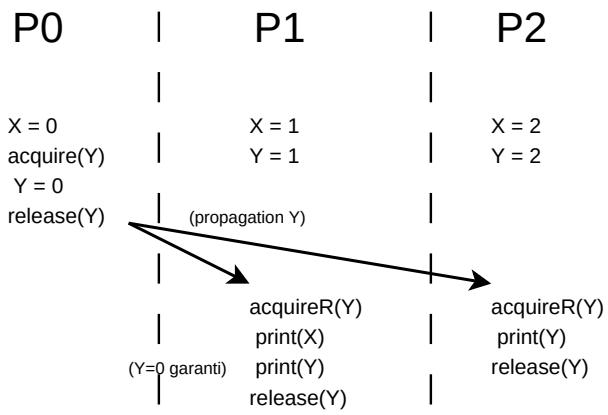


FIG. 3.1 – Partage de données avec le modèle de cohérence à l'entrée. X et Y sont des variables partagées. *acquire(Y)* signifie que l'on veut acquerir le verrou de Y. Les seules garanties de lectures portent sur Y.

soient propagées sur les sites effectuant les demandes suivantes. Cette propagation intervient lorsque $P0$ relâche le verrou. Les processus $P1$ et $P2$ peuvent alors acquérir le verrou en lecture et lire la donnée en parallèle.

3.2 Un protocole hiérarchique

Le protocole de cohérence permet de mettre en œuvre les propriétés du modèle de cohérence. Comme expliqué dans la section 2.3, le protocole utilisé dans *JuxMem* respecte l'organisation hiérarchique des sites. Un site client est attaché à un groupe de données local (*Local Data Group (LDG)*) dans lequel la donnée est stockée et qui est lui-même inclus dans le groupe de données global (*Global Data Group (GDG)*) regroupant tous les LDG. La donnée n'est plus seulement présente sur les sites clients mais se retrouve répliquée au niveau des LDG. Ainsi le rôle du protocole de cohérence est de faire circuler les données entre clients et LDG en respectant les contraintes imposées par le modèle de cohérence à l'entrée. Lorsque le client désire accèder à une donnée partagée, sa requête traverse les différents niveaux de hiérarchie pour être satisfaite. Le principe de ce protocole hiérarchique peut se résumer de la sorte : si un client détient un verrou alors le LDG auquel il appartient détient le verrou du même type. Il y a donc 2 jeux de verrous : l'un pour les LDG et l'autre pour les clients.

3.2.1 Fonctionnement général

La figure 3.2 illustre un scénario où 4 clients répartis dans 2 LDG vont demander le verrou en écriture, puis en lecture. Le schéma 3.2(a) représente l'architecture hiérarchique de JuxMem. Le client 3 demande le verrou en écriture auprès de son LDG qui, ne le possédant pas, en fait la demande au GDG (flèche 1). Aucun verrou n'étant jusqu'alors pris, le GDG donne le verrou en écriture au LDG 2 qui le transmet au client 3 avec un exemplaire de la donnée à jour (flèche 2, schéma 3.2(b)). Le LDG 2 et le client 3 sont alors dans un état indiquant qu'ils sont les uniques possesseurs du verrou en écriture. Les clients 1 et 4 demandent

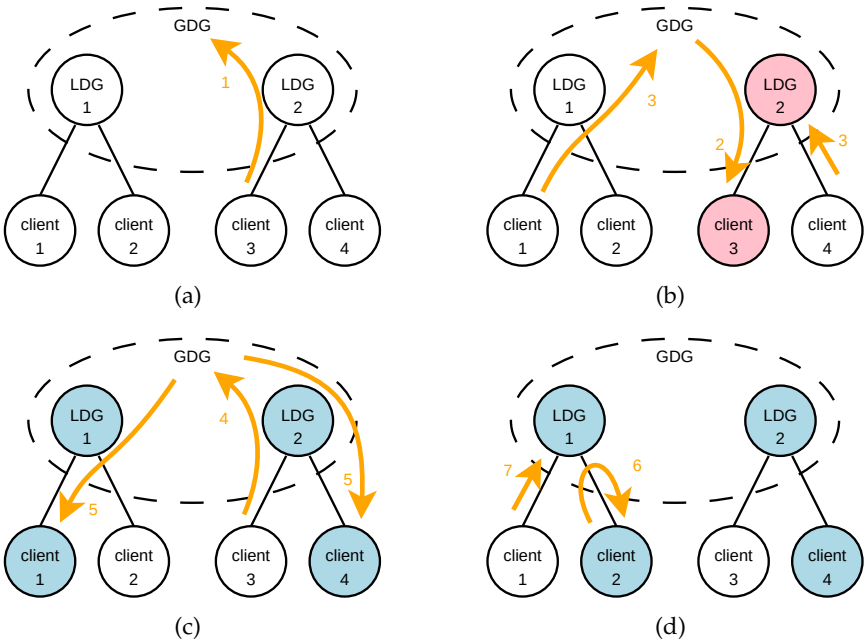


FIG. 3.2 – Un protocole hiérarchique de cohérence des données.

alors le verrou en lecture à leurs LDG respectifs (flèche 3). Le LDG 1 transmet immédiatement la demande au GDG qui le place en file d'attente. Le LDG 2 est déjà propriétaire du verrou exclusif et place donc le client 4 en file d'attente des lecteurs.

Lorsque le client 3 termine son écriture, il envoie ses mises à jour au LDG 2 et relâche le verrou en écriture (flèche 4, schéma 3.2(c)). Le LDG 2 vérifie qu'aucun autre client n'attend ce verrou puis le rend au GDG. A ce moment, tous les LDG ont recu les modifications effectuées par le client 3. Le LDG 2 envoie alors au GDG la demande de verrou en lecture provoquée par le client 4. Le GDG vérifie alors qu'aucun autre LDG n'attend le verrou en écriture avant de distribuer le verrou en lecture aux LDG qui en ont fait la demande (flèche 5). Les 2 LDG distibuent à leur tour le verrou en lecture aux clients 1 et 4 en y joignant un exemplaire de la donnée.

Le client 2 demande à son LDG le verrou en lecture. Le LDG 1, qui en est déjà l'un des possesseurs, vérifie qu'aucun de ses clients n'a entre temps fait de demande de verrou en écriture et le distribue au client 2 (flèche 6, schéma 3.2(d)). Le client peut rendre le verrou au LDG 1 (flèche 7) qui le conserve tant que le client 2 ne l'aura pas relâché.

Ce scénario montre comment des écritures peuvent être alternées avec des lectures. Certains choix stratégiques ont été faits quant à la gestion des sites en attente du verrou. Ainsi les lecteurs doivent attendre que tous les écrivains aient été satisfaits pour obtenir le verrou en lecture. Des demandes discontinues du verrou en écriture peuvent ici causer une famine pour les lecteurs. De plus, si des clients accèdent de façon concurrente à la donnée, dès lors qu'un site en demande l'accès exclusif, les requêtes en lecture sont mises en attente et satisfaites après les écritures. D'autres politiques sont envisageables : leur discussion sort du cadre de ce travail.

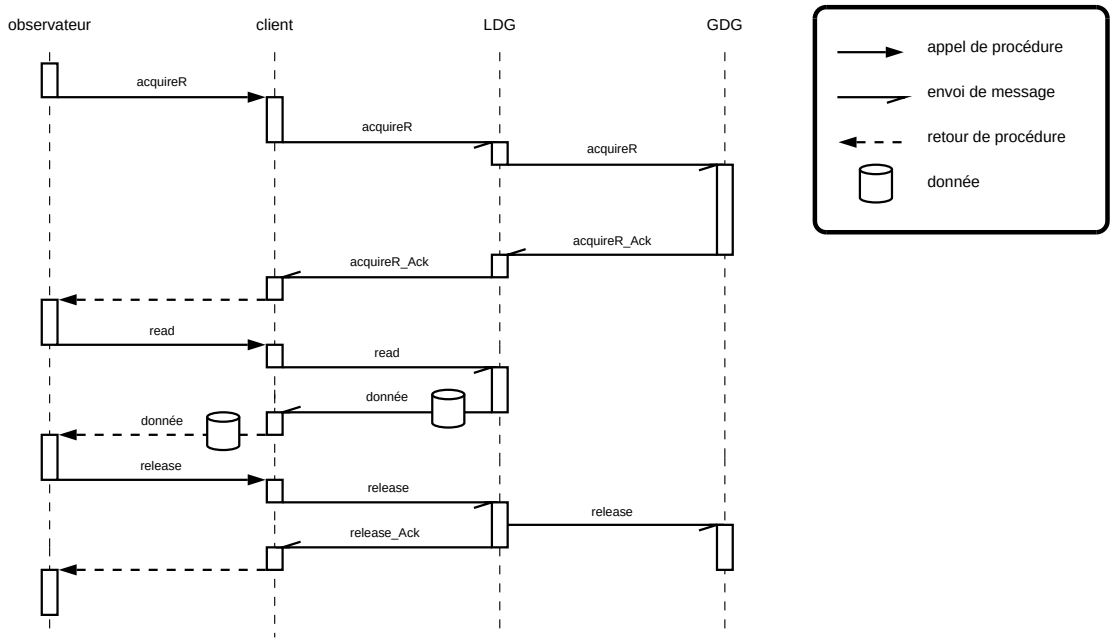


FIG. 3.3 – Représentation *UML* des intéractions entre une application cliente (observateur), son LDG et le GDG lors d'une séquence classique de lecture d'une donnée partagée.

Le fonctionnement du protocole de cohérence n'est en soi pas très difficile à appréhender. Ce qui en fait un système complexe est la multiplication des acteurs : le LDG est une entité centrale qui a la charge de communiquer avec le GDG et de nombreux clients. Chacun des messages qu'il recoit est susceptible de modifier son état interne, ce qui n'est pas sans poser des problèmes de concurrence d'accès à ses variables d'état.

3.2.2 Analyse d'une lecture

La lecture d'une donnée par une application entraîne des appels de procédures coté client et l'envoi de plusieurs messages vers les membres des groupes de données. La figure 3.3 illustre les intéractions entre un site client, son LDG et le GDG lors d'une séquence classique de lecture d'une donnée partagée. L'*observateur* correspond à une application utilisant la bibliothèque JuxMem. Celui-ci demande le verrou en lecture (*acquireR*), lit la donnée(*read*) puis relâche le verrou (*release*). L'invocation de ces trois primitives synchrones s'effectue via l'interface proposée par le client JuxMem qui est déployé sur le même site que l'application.

Un appel à *acquireR* provoque l'envoi d'un message réseau vers le LDG. Le client est alors endormi jusqu'à ce que le verrou en lecture lui soit concédé. Ce temps dépend directement de l'état du protocole de cohérence et plus précisemment de la fréquence des accès exclusifs. Enfin, la mise à jour de la donnée est retardée jusqu'à la demande de lecture de la part de l'application.

Dans cet exemple, la lecture de la variable partagée nécessite l'envoi de 9 messages dont 1 contient la donnée. Le nombre total de messages augmente dès lors que la donnée est répliquée sur plusieurs LDG ainsi qu'au sein des LDG.

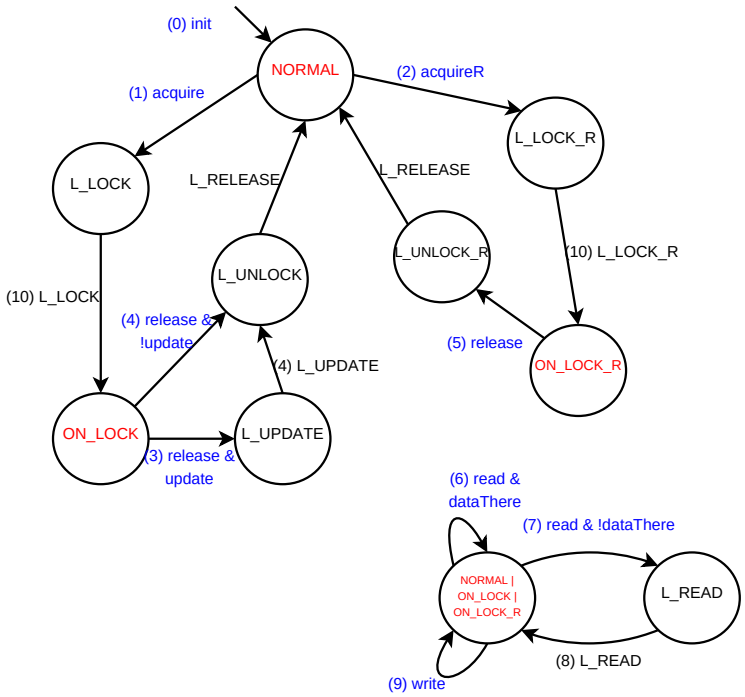


FIG. 3.4 – Automate-action du client. Les états rouges ne bloquent pas l'application qui utilise le client alors que les états noirs correspondent aux primitives synchrones. Les transitions bleues représentent des appels à des procédures offertes par le client, les autres représentent le *tag* du message réseau reçu.

3.2.3 Modélisation du comportement des sites

La modélisation du protocole de cohérence est une étape nécessaire pour déterminer les améliorations pouvant y être apportées. L'automate-action est une représentation permettant de modéliser des applications distribuées. Chaque rôle du système est caractérisé par un automate. Les transitions s'effectuent lors d'évènements comme la réception d'un message particulier ou l'appel à une procédure par un processus local. Les transitions sont décorées de la condition devant être satisfaite pour les emprunter. Cette condition peut être une expression booléenne ou la réception d'un message dont le *tag* est indiqué en lettres majuscules. A coté de la condition se trouve un numéro indiquant la liste d'actions à effectuer.

La figure 3.4 représente l'automate du client. Les actions associées à cet automate sont présentées dans l'annexe A.1. A titre d'exemple, la lecture d'une donnée se traduit par l'appel de la primitive *acquireR* ce qui modifie l'état du client de *normal* à celui de *L_LOCK_R* et provoque l'envoi d'un message demandant le verrou au LDG (action 2). Le client reste alors dans cet état jusqu'à la réception d'un message *L_LOCK_R* lui donnant le verrou en lecture. L'ensemble des automates réalisés pour les différents rôles sont disponibles en annexe A.

3.3 Discussion

Le protocole de cohérence mis en place dans JuxMem implémente le modèle de cohérence à l'entrée tout en s'adaptant à une architecture hiérarchique. Il permet de distribuer les verrous en lecture et en écriture entre les clients et les LDG. Le scénario de couplage de code étudié ici met en jeu un site ayant la charge d'observer la donnée partagée. Ces lectures doivent être rapides et n'occasionner que peu de retard pour les autres sites. Même si le protocole actuel permet une solution valide, certaines modifications sont à considérer pour améliorer ses performances. La réduction du nombre de messages provoqués par la lecture d'une donnée est possible en acceptant d'accèder à des données dont la version n'est pas la plus récente. Ceci est envisageable dans des applications de visualisation où il n'est pas nécessaire de connaître la denière version. Par ailleurs la politique de distribution des verrous peut entrainer la famine des lecteurs si plusieurs écrivains s'accordent à monopoliser le verrou. Le protocole doit prendre en compte ce cas pour éviter à l'observateur d'être bloqué en évitant par exemple d'attendre le verrou.

Chapitre 4

Extension : autoriser les accès concurrents entre lectures et écritures

L'analyse du premier protocole de cohérence utilisé par JuxMem a montré que dans le cadre du scénario d'observation de couplage de code, l'observateur pouvait être bloqué par l'accès fréquent de la donnée en écriture. Dans ce cas, l'autorisation de certaines lectures en parallèle des écritures est une solution pouvant accélérer les performances de l'observateur. Ce chapitre propose donc une extension au protocole de cohérence autorisant des accès concurrents entre les lectures et les écritures.

4.1 Proposition d'amélioration : la lecture relâchée

Le scénario considéré met en scène un site observateur dont le rôle est d'accèder à la donnée de façon rapide. Cette amélioration de la vitesse de lecture ne doit causer que peu ou pas de surcoût pour les autres sites. Le principal facteur limitant la vitesse de lecture est l'attente de la prise du verrou. Celle-ci peut être rendue longue si plusieurs écrivains monopolisent de manière exclusive l'accès à la donnée.

L'idée principale de l'extension du protocole est donc d'outrepasser la demande de verrou en lecture et de retourner la donnée dans les plus brefs délais. Une conséquence directe de cette stratégie est d'autoriser ces lectures d'un genre particulier en même temps que les écritures. Ces lectures sont nommées *lectures relâchées* et la primitive associée est *rlxRead* (pour *relaxed read*). Le problème de la concurrence d'accès à la donnée, jusqu'alors résolu par le protocole de cohérence, se pose de nouveau : si il n'est pas possible de lire une donnée en cours de modification par un écrivain, différentes copies de celle-ci restent disponibles et inutilisées sur les LDG et les sites clients.

Ces réplicas sont issus de mises à jour ne refletant pas forcément l'état actuel de la donnée. En effet, celle-ci se trouve peut-être en cours de modification, auquel cas, et même si le verrou en écriture n'est pas encore relâché, toutes les copies sont considérées comme anciennes. Une autre possibilité est que plusieurs écrivains modifient successivement la donnée au sein d'un même LDG qui ne relâche le verrou ni ne propage les modifications au GDG. Le modèle de cohérence à l'entrée ne garantit qu'une donnée soit à jour que lors de l'acquisition de son verrou. Dans ce modèle, pour un observateur qui n'acquiert pas le verrou, l'utilisation de la donnée contenue dans son propre cache ou celle disponible sur son LDG ne peut garantir que la donnée soit à jour.

L'approche que nous considérons propose d'autoriser ce type de lectures non-synchronisées tout en contrôlant la "fraîcheur" de la donnée. Ceci est possible en bornant l'écart entre la version retournée et la version la plus récente. Ainsi pour chaque lecture relâchée, l'application spécifie le nombre de versions de retard autorisées avant qu'une donnée ne soit considérée comme périmée.

4.2 Formalisation des lectures relachées

L'expression du retard entre la version la plus récente et celle retournée par une lecture relâchée n'est pas chose aisée. L'aspect hiérarchique du protocole de cohérence ne permet pas de connaître immédiatement la version la plus récente. Ainsi la donnée peut être représentée en deux versions différentes sur deux LDG dont l'un d'entre eux détient le verrou en écriture. A un niveau plus bas de la hiérarchie, les versions détenues par les clients n'ayant pas acquis le verrou récemment peuvent être plus anciennes que celle de leur LDG.

4.2.1 Fenêtre de lecture

Pour exprimer le retard entre la version la plus récente et celle retournée par la lecture relâchée, nous introduisons deux paramètres qui prennent en compte ces deux niveaux de hiérarchie :

- La constante D, propre à chaque donnée, exprime le nombre de fois que le LDG peut transmettre successivement le verrou en écriture, sans effectuer de mise à jour du GDG. Si $D = 0$ alors le LDG doit propager les modifications après chaque relâchement du verrou en écriture par l'un de ses clients. Dans ce cas tous les LDG ont la même version de la donnée.
- Le paramètre w est spécifié par le client lors de chaque appel à la primitive de lecture relâchée *rlxRead*. C'est la *fenêtre de lecture*. Elle exprime l'écart total maximum entre la version la plus récente et celle retournée par la lecture relâchée. C'est en fait la somme de la valeur D avec la distance maximale entre la version détenue par le LDG du client et celle retournée par la lecture relâchée. En quelque sorte, w absorbe D.

Les distances D et w sont positives ou nulles et respectent le fait que w soit supérieur ou égal à D. La différence $w-D$ correspond donc à la distance maximale entre la version détenue par le LDG du client et celle retournée par la lecture relâchée. A titre d'exemple, si $D = 3$ alors un LDG ne pourra distribuer successivement le verrou en écriture que 3 fois sans propager les modifications. Si $w = 4$, la version de la donnée lue par un client effectuant une lecture relâchée retourne soit la "dernière" version de la donnée, soit la version précédente.

4.2.2 Discussion sur la sémantique des paramètres

La notion de "dernière" version fait référence à la dernière version propagée par le LDG ayant le verrou en écriture. C'est à dire qu'elle peut déjà être ancienne de D versions. Si $w = D$ alors le client lit la même version que celle détenue par son LDG. Considérer $D = 0$ et $w = 0$ ne revient pas pour autant à effectuer une lecture avec prise du verrou et ce, pour deux raisons :

- Lors de la lecture relâchée, le verrou en écriture peut de nouveau être distribué et la donnée en cours de modification. Ceci n'est pas autorisé dans le modèle de cohérence à l'entrée.

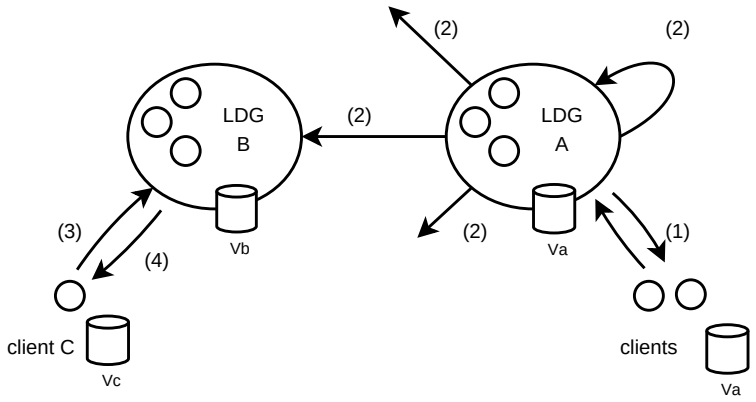


FIG. 4.1 – Lecture relâchée. Des clients attachés au LDG A incrémentent la version V_a de la donnée (1). Ces modifications sont propagées sur les autres LDG toutes les D redistributions du verrou (2). Un client effectue une lecture relâchée (3) et récupère V_b ou lit V_c.

– Entre le moment où le LDG répond à une requête de lecture relâchée et celui où le client utilise effectivement la donnée, un nombre non controlé de versions ont pu être produites.

Lorsqu'un LDG (dont la version de la donnée est V_{LDG}) reçoit une demande de lecture relâchée (de fenêtre w) de la part d'un client C (dont la version de la donnée est V_c), la formule (F) permettant de décider si le client peut utiliser sa version de la donnée (V_c) est :

$$V_c \geq V_{LDG} - (w - D)$$

L'utilisation de la version du client réduit le coût réseau de la lecture. Dans le cas contraire, la donnée du LDG est transférée au client.

La lecture relâchée donne lieu à une extension du modèle de cohérence : le modèle de cohérence à l'entrée y est toujours respecté et garantit d'accéder à la version la plus récente de la donnée lorsque le verrou est acquis. Des garanties supplémentaires sont offertes là où l'EC n'en donne pas. Ainsi la lecture d'une valeur sans la prise du verrou n'est-elle plus complètement incontrôlée.

4.2.3 Une vue d'ensemble

La figure 4.1 illustre l'aspect hiérarchique du protocole et les rôles de D et w. La donnée d est présente en trois versions différentes (V_a, V_b et V_c) sur les LDG et les clients. Plusieurs clients effectuent des écritures à tour de rôle et incrémentent la version V_a de la donnée d (1). Toutes les D_d redistribution successives du verrou en écriture au sein du même LDG, les mises à jour sont envoyées aux autres LDG (2). Le client C effectue une lecture relachée en précisant le nombre de versions de retard w autorisées. Une requête contenant la version V_c qu'il détient est envoyée à son LDG (3). En fonction de F la version V_b de la donnée détenue par le LDG_B ou une autorisation d'utiliser la version V_c est transmise au client (4).

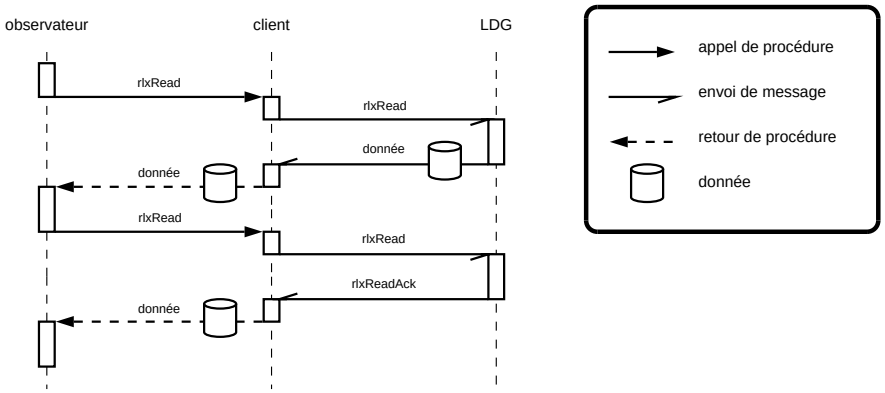


FIG. 4.2 – Représentation *UML* des intéractions entre une application cliente (observateur) et son LDG lors de deux appels successifs à la primitive de lecture issue de l'extension du protocole. Le deuxième appel bénéficie du cache du client et ne transfert pas la donnée.

4.3 Analyse d'une lecture relachée

Tout comme la lecture d'une donnée dans le cadre strict du protocole de cohérence à l'entrée, la lecture relâchée provoque l'appel à des primitives coté client ainsi que l'envoi de messages réseau. La figure 4.2 illustre les intéractions entre une application cliente (un observateur) et son LDG lors de deux appels consécutifs à *rlxRead*. Les lectures relâchées déclenchent chez le client l'envoi vers son LDG d'un message contenant le numéro de version de sa donnée ainsi que la fenêtre de lecture w. La première lecture nécessite la transmission de la donnée depuis le LDG vers le client, alors que la seconde ne transmet qu'un acquittement. Il existe un troisième scénario non représenté sur cette figure : un client peut ne pas avoir la donnée en cache lors de sa demande de lecture relâchée. La stratégie est ici de transformer son appel à *rlxRead* en la séquence classique *acquireR*, *read* et *release* qui initialisera ainsi son cache.

La primitive *rlxRead* est asynchrone, au même titre que les primitives *acquireR*, *read* et *release*. Cependant, le nombre de messages qu'elle génère et l'indépendance de son traitement vis-à-vis de l'état des autres sites, en fait une procédure dont l'execution est rapide. L'observateur n'attend pas qu'un site relâche un verrou pour que ses requêtes soient satisfaites. La lecture relâchée ne modifie pas le comportement du protocole de cohérence, à l'exception de la contrainte imposée par la constante D sur la gestion du verrou en écriture par les LDG. Le surcoût de cette extension de protocole n'en est que plus faible.

4.4 Modélisation du comportement des sites

L'extension du protocole de cohérence n'apporte que quelques modifications au comportement décrit dans la section 3.2.3.

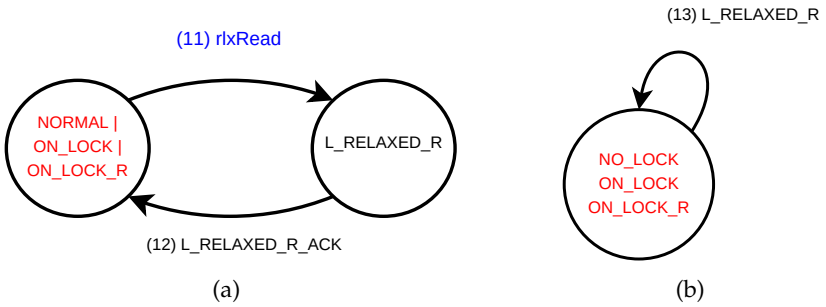


FIG. 4.3 – Ajout des transitions liées à l'extension du protocole. (a) Automate du client. (b) Automate du LDG.

4.4.1 Le client

Le client doit désormais prendre en compte les appels à la procédure *rlxRead*. La figure 4.3(a) présente les ajouts à effectuer sur l'automate du client (figure 3.4). Un nouvel état (*L_RELAXED_R*) est nécessaire pour signifier que le client est en attente d'une réponse pour sa requête de lecture relâchée. Les actions associées s'ajoutent aux 10 actions déjà présentées en annexe A.1 :

11. rlxRead - Le client envoie à son LDG une demande de lecture relâchée avec la taille de la fenêtre ainsi que la version de la donnée qu'il détient.
```
send(LDG, L_RELAXED_R, window, localData.version)
```
12. L_RELAXED_R_ACK - Le client récupère la donnée si elle est présente dans le message.
```
IF (message contains data) THEN
 localData = message.data;
ENDIF
return localData;
```

4.4.2 Le groupe de données local

Quelque soit l'état du LDG, celui-ci est disponible pour satisfaire les requêtes de lecture relâchée. La figure 4.3(b) ajoute une transition à l'automate complet du LDG présenté en figure A.1 de l'annexe. Cette transition permet d'évaluer la formule et de décider de l'envoi de la donnée.

13. L_RELAXED_R
```
IF (message.dataVersion < localData.version - (message.window - D)
THEN
 send(clientID, L_RELAXED_R_ACK, localData);
ELSE
 send(clientID, L_RELAXED_R_ACK, localData.version);
ENDIF
```

Chapitre 5

Implémentation

La mise en œuvre du protocole de cohérence s'inscrit dans le cadre de la plate-forme JuxMem. Elle a été largement facilitée par l'architecture en couches du projet qui offre une interface dédiée à la réalisation des protocoles de cohérence. Ainsi l'installation d'un nouveau protocole consiste à instancier les actions de cohérence déclenchées par une liste restreinte d'évènements. Ce niveau d'abstraction est inspiré par des travaux précédents dans le domaine des systèmes à mémoire virtuellement partagée tels que DSM-PM^2 [5].

5.1 JuxMem : une architecture en couches

La conception de JuxMem a donné lieu à un empilement de couches remplissant chacune des fonctionnalités bien particulières. Cette stratégie permet d'interchanger les briques pour tester de nouveaux algorithmes. Ces couches, représentées par la figure 5.1, permettent d'établir des communications et stocker des objets dans un environnement pair-à-pair, de proposer de la tolérance aux fautes et de gérer les données suivant différents modèles de cohérence.

5.1.1 La couche JXTA

Le projet JXTA [1] propose une plate-forme générique offrant les fonctionnalités que l'on retrouve dans la plupart des systèmes pair-à-pair. Les fonctionnalités les plus communes sont la découverte des pairs et des ressources, la gestion des communications entre pairs et la gestion des groupes. Le routage des messages s'effectue à ce niveau grâce à une table de hachage distribuée (*distributed hash table (DHT)*). JXTA est un projet de recherche de Sun Microsystems ouvert à la communauté du pair-à-pair.

L'entité de base de JXTA est le pair. Il peut être enrichi de différentes fonctionnalités selon sa nature : envoyer et recevoir des messages, découvrir et stocker des annonces de ressources disponibles dans le réseau, propager et centraliser les requêtes des autres pairs ou même établir un relai pour contourner un pare-feu. C'est sur la notion de pair JXTA que la version actuelle de JuxMem évolue. Les rôles de client, fournisseur et gestionnaire sont donc déployés sur des pairs. Plusieurs rôles peuvent cohabiter sur un même pair.

5.1.2 Le noyau JuxMem

Le noyau de JuxMem (*core*), développé par Mathieu Jan dans le cadre de son doctorat, offre une interface de gestion des ressources de plus haut niveau. Les principaux services rendus par cette couche sont la correspondance entre les groupes JuxMem et les groupes JXTA ainsi que la gestion de la mémoire sur les pairs. Les fonctions disponibles permettent de lire et écrire la donnée présente sur le pair. Il est aussi possible de l'extraire ou de l'insérer dans un message JXTA. L'aspect hiérarchique de JuxMem (figure 2.2) est déjà présent à ce niveau et la procédure d'allocation se charge d'effectuer des recherches sur des *cluster groups* distants. L'allocation retourne alors la liste des fournisseurs sélectionnés pour héberger la donnée.

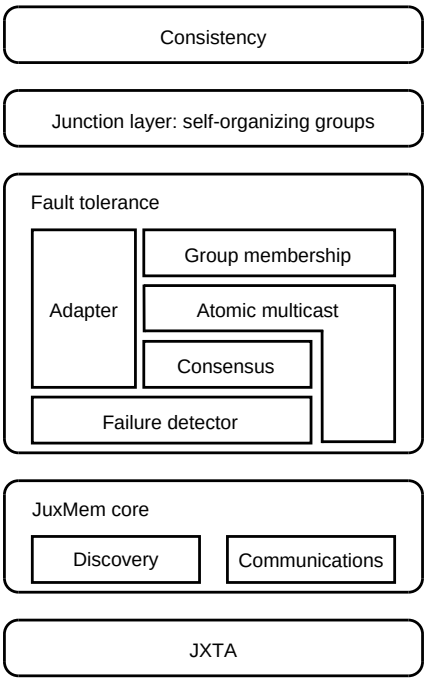


FIG. 5.1 – JuxMem : Une architecture en couches.

5.1.3 Support de la tolérance aux fautes

La couche de tolérance aux fautes, développée par Sébastien Monnet dans le cadre de son doctorat, abstrait l'ensemble des pairs détenant une copie de la donnée en une entité logique. Ainsi le GDG identifie tous les pairs jouant les rôles de fournisseurs JuxMem et les LDG une partition du GDG. Ceci est réalisé par l'implémentation d'un protocole de communication de groupe basé sur la diffusion atomique et le consensus. Les sites sont surveillés et suspectés en cas de panne franche par les détecteurs de fautes hiérarchiques de Marin Bertier.

5.2 Interfaces utilisées par le protocole de cohérence

L'implémentation du protocole de cohérence est résolument évènementielle. C'est à dire que c'est l'appel à une procédure ou la réception d'un message qui déclence l'execution du code associé à la gestion de la cohérence. Les pairs ayant la charge d'une copie de la donnée sont directement concernés par la cohérence. Ces pairs sont les clients et les fournisseurs.

5.2.1 Le client

Le *client* est directement lié avec l'application utilisateur. Il est donc sollicité par celle-ci suite à l'appel de l'une des primitives d'accès à la donnée (*acquire*, *write*, *release*...etc). Ces requêtes sont satisfaites en communiquant avec le LDG auquel il appartient. L'interface de programmation du protocole de cohérence offerte au client lui permet de lire et écrire sa propre copie de la donnée, d'en changer le numéro de version, de l'extraire ou de l'insérer dans un message. Un objet de communication propose des méthodes spécialisées dans l'envoi de messages[1] comme la demande du verrou ou la propagation d'une mise à jour. Pour que les primitives d'accès à la donnée soient synchrones, après l'envoi d'une requête par

[1]La liste des messages peut-être retrouvée à partir des automates présentés en annexe.

le réseau, le processus léger se bloque sur une condition jusqu'à ce que la réception de la réponse lui envoie un signal.

5.2.2 Le fournisseur

Le *fournisseur* supporte deux rôles du point de vue du protocole de cohérence : il est à la fois membre du LDG et du GDG. Ces deux rôles ne déclenchent des actions que lors de la réception d'un message. Pour le LDG, les messages proviennent de ses clients et du GDG. Pour le GDG, les messages proviennent des LDG. Tout comme le client, l'interface de programmation du protocole de cohérence permet de lire et d'écrire la donnée hébergée sur le pair. Cette possibilité n'est pas sans poser des questions de concurrence d'accès. C'est pourquoi toutes les fonctions appelées par l'arrivée d'un message sont executées de manière atomique à l'aide d'une section critique.

La version de la donnée ne peut pas être non plus gérée comme des variables locales au LDG et au GDG : la mise à jour de la donnée au niveau du LDG entraînerait alors une incohérence entre la donnée du pair et la version de la donnée affichée par le GDG. La version de la donnée est donc elle aussi propre à chaque pair et s'accède de la même manière que la donnée.

Chapitre 6

Evaluation préliminaire et discussion

L'objectif de ce chapitre est de montrer que l'extension du protocole de cohérence autorisant la concurrence entre les lectures et les écritures permet d'améliorer les temps d'accès en lecture. Ce gain de temps est possible si l'utilisateur accepte de lire des données considérées comme anciennes au moment de l'accès. Le retard maximum entre la donnée lue et sa version la plus à jour est cependant borné par l'utilisateur. Cette contrainte reste cependant dans le cadre d'un scénario applicatif réaliste.

6.1 Description de la plate-forme experimentale

Le développement et l'évaluation du protocole de cohérence mis en place pour JuxMem se sont déroulés au sein du projet *Grid'5000* [2]. Ce projet français vise à construire une plate-forme experimentale réunissant jusqu'à cinq mille processeurs répartis sur huit sites géographiquement distribués que sont Bordeaux, Grenoble, Lille, Lyon, Orsay, Rennes, Sophia et Toulouse. Ces sites sont interconnectés par le réseau gigabit d'enseignement et de recherche *Renater*. *Grid'5000* propose un ensemble de ressources matérielles hétérogènes ainsi qu'un environnement logiciel hautement configurable. Il est complètement dédié à la recherche sur les grilles de calculateurs.

6.1.1 L'environnement matériel

La mise en place de la plate-forme *Grid'5000* n'est pas une chose aisée et nécessite la coordination des différentes équipes impliquées dans le projet. Cette environnement n'était pas complètement fonctionnel lors de la réalisation de l'évaluation du protocole de cohérence. C'est pourquoi les tests se sont déroulés sur les ressources rennaises.

Le projet Paris est le gestionnaire des ressources mises à la disposition de *Grid'5000* par le site rennais. Elles sont composées de trois grappes dont les 164 nœuds bi-processeurs sont de type Intel Xeon, AMD Opteron ou IBM PowerPC. La grappe utilisée pour les expérimentations présentées dans la suite de ce document est composée de 64 nœuds *Sun Fire V20z* à base de bi-processeurs AMD Opteron cadencés à 2.2GHz, pourvus de 2Go de mémoire vive (RAM) et interconnectés par un réseau ethernet gigabit. Le système d'exploitation présent sur les nœuds est Linux Debian.

6.1.2 L'environnement logiciel

Le développement du protocole de cohérence a été réalisé avec la version de JuxMem basée sur JXTA 2.3.2 pour Java. La machine virtuelle Java utilisée est la version 1.4.2 de la Sun JVM. Les tailles des projets JXTA et JuxMem ont nécessité l'utilisation d'un outil spécialisé dans la génération de fichiers de traces d'execution. La librairie *log4j* proposée par la fondation Apache permet de journaliser l'execution des tests dans plusieurs fichiers en spécifiant des niveaux de priorité ou en marquant les évenements avec un *tag*.

6.1.3 Un outil de déploiement des tests : le JXTA Distributed Framework

Dans les systèmes distribués, dès lors que l'on veut utiliser plusieurs nœuds physiques pour supporter les différents acteurs d'une application parallèle, le protocole experimental implique souvent que l'utilisateur se connecte sur chacun des nœuds pour y démarrer ses processus. Le JXTA Distributed Framework (JDF) [8] permet d'automatiser les déploiements à grande échelle pour les applications JXTA. L'utilisateur spécifie des fichiers de configuration décrivant les processus de l'application, les bibliothèques et fichiers nécessaires à son execution ainsi que les sites physiques disponibles. Le JDF se charge de déployer les fichiers sur les sites, de démarrer les processus et de collecter les résultats.

6.2 Le programme de test

Le programme développé pour l'évaluation des performances est un programme appartenant à la couche utilisateur et reposant sur la librairie JuxMem. Il met en jeu les fonctionnalités du protocole de cohérence dans le cadre d'un scénario particulier (*section 1.3*). Une donnée partagée est modifiée par un site (le producteur), lue par un second (le consommateur) et observée par un troisième (l'observateur). Ce scénario ne colle pas tout à fait au schéma du producteur-consommateur classique avec un tampon à une place puisque les lectures-écritures ne sont pas synchronisées du point de vue applicatif : le producteur n'attend pas que la donnée soit lue pour en produire une nouvelle et inversement, le consommateur n'attend pas que le producteur aie modifié la donnée pour la lire. Cette orientation vers un scénario de type lecteur-rédacteur permet l'utilisation importante des mécanismes du protocole de cohérence provoquée par des lectureset des écritures concurrentes.

6.2.1 Description des rôles

Les sites intervenant dans l'application de test sont répartis au sein de deux groupes de données locaux (LDG pour *Local Data Group*) comme présentés sur la figure 6.1. Chaque LDG est composé d'un fournisseur (*provider*) auquel se rattache un ou plusieurs sites clients. Le producteur et le consommateur sont associés au premier LDG alors que l'observateur est attaché au second. Tous les rôles ne sont pas déployés dans un même LDG, ce qui permet de prendre en compte l'impact des messages émis au niveau du groupe de données global (GDG). D'un point de vue hiérarchique, l'observateur est plus distant du producteur que ne l'est le consommateur . Ceci met en valeur le cout de la prise d'un verrou en lecture d'un client isolé sur un autre LDG.

Les instructions executées par les clients sont réduites à leur plus simple expression. Dans la version originale du protocole de cohérence, les rôles de consommateur et d'observateur

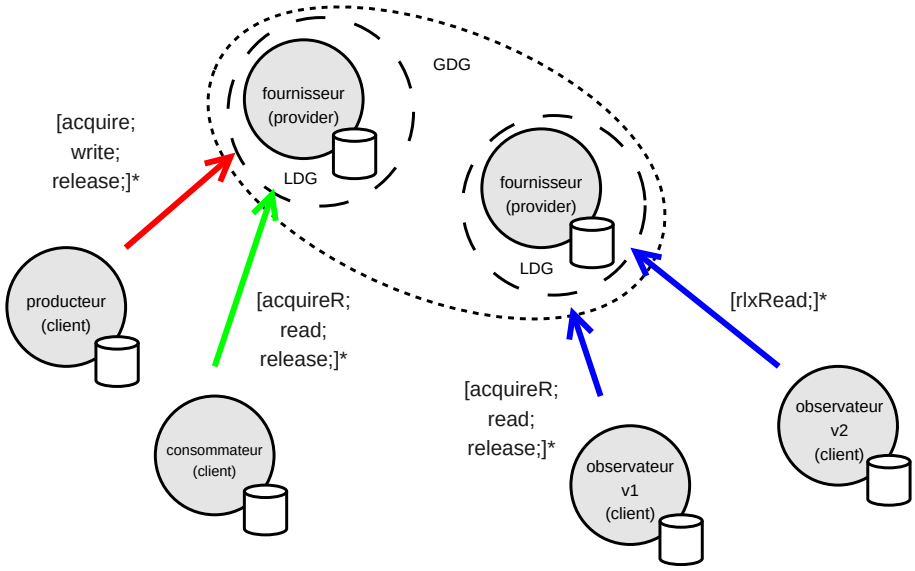


FIG. 6.1 – Les rôles proposés par l'application de test.

ne diffèrent pas.

- Le *producteur* effectue l'allocation de la donnée, récupère son identifiant dans JuxMem et le transmet au consommateur ainsi qu'à l'observateur. Cette étape a aussi comme intérêt de synchroniser les trois clients avant le démarrage du test. A partir de ce moment, le producteur effectue 50 itérations correspondant à la prise du verrou en écriture, l'écriture et le relachement du verrou.
- Le *consommateur* attend l'identifiant de la donnée puis effectue 50 itérations correspondant à la prise du verrou en lecture, la lecture et le relachement du verrou.
- L'*observateur* (v_1) attend l'identifiant de la donnée puis se comporte comme le consommateur. D'un point de vue experimental, l'observateur (v_1) diffère du consommateur en se déployant dans le second LDG.
- L'*observateur* (v_2) est utilisé pour effectuer les tests portant sur l'extension du protocole. L'observateur (v_2) attend l'identifiant de la donnée puis effectue 50 itérations correspondant à une lecture relâchée (*rlxRead*).

La mesure des performances s'effectue en prenant l'écart entre les temps sytème de début et de fin d'itération dans le code des clients. A l'exception de la journalisation du traitement des mesures, aucune autre instruction n'est executée dans les boucles et l'on obtient un stress important au niveau de l'accès à la donnée et donc une utilisation intensive du protocole de cohérence.

6.2.2 Description des messages

Le protocole de cohérence et, d'une manière générale, les couches inférieures de la pile JuxMem sont peu gourmands en temps de calcul. Les performances sont directement dépendantes 1) du nombre de messages à transmettre sur le réseau pour effectuer un accès à une donnée partagée et 2) de la taille de ces messages. Les figures 3.3 et 4.2 donnent un aperçu

type observateur	sans	v_1			v_2			v_2'		
taille donnée	1Ko	1Ko	1Mo	4Mo	1Ko	1Mo	4Mo	1Ko	1Mo	4Mo
producteur	18	18	79	505	18	69	479	18	60	519
consommateur	20	18	77	496	18	63	474	19	61	522
observateur		20	79	505	9	44	89	7	23	43

FIG. 6.2 – Temps d'accès moyens en millisecondes pour les 3 sites, en fonction de la version de l'observateur et de la taille de la donnée.

des messages provoqués par les instructions executées dans une itération d'observateur avec et sans l'utilisation de l'extension du protocole.

- Dans le cas d'un observateur utilisant l'ancien protocole de cohérence (figure 3.3), la réalisation d'une itération correspond à la séquence *acquireR*, *read* et *release* provoque la transmission de 9 messages dont 8 sont bloquant pour le client. Un seul de ces 8 messages contient la donnée et est de taille plus importante.
- Dans le cas d'un observateur utilisant l'extension du protocole de cohérence (figure 4.2), la réalisation de 2 appels successifs à la primitive *rlxRead* peut donner lieu à 2 comportements différents. Le premier appel provoque la transmission de 2 messages dont l'un contient la donnée. Le deuxième met en jeu le cache du client et ne provoque que 2 messages sans donnée. La fréquence de chacun de ces 2 types d'accès est étudiée dès la section 6.5.

Les rôles de producteur et de consommateur ont un comportement similaire en terme d'échanges réseaux à celui de l'observateur utilisant l'ancien protocole de cohérence.

6.3 Un exemple simple : le producteur-consommateur

La première mesure effectuée concerne une configuration simple sans observateur. La donnée a une taille de 1 Ko et est répliquée sur le deuxième LDG avec une copie par LDG. Le temps d'accès moyen du producteur est de 18 ms et celui du consommateur est de 20 ms. Les performances de ces deux sites sont sensiblement les mêmes. Ceci est expliqué par le fait que l'accès exclusif à la donnée est alterné entre le producteur et le consommateur. Pendant que l'un accède la donnée, l'autre en fait la demande.

La figure 6.3(a) illustre les temps d'accès du producteur et du consommateur en fonction du temps système pour chacune des 50 itérations. Elle permet de visualiser la distribution des coûts d'accès ainsi que des évènements particuliers liés à l'interaction entre les sites. Le temps système correspond à l'horloge du nœud sur lequel le client est déployé. Dans la théorie des systèmes distribués, ces horloges ne peuvent être comparées ou assimilées à une horloge globale. Cependant on peut considérer que les nœuds soient suffisamment synchronisés en comparaison des durées mesurées pour présenter les temps d'accès de plusieurs sites sur un même graphe.

Ces courbes montrent qu'aucun des sites ne rencontre une situation de famine. Le producteur et le consommateur débutent et terminent leurs accès en même temps. Les premiers accès sont plus lents, ce qui s'explique par le démarrage de la machine virtuelle Java et la résolution des communications JXTA au début du protocole.

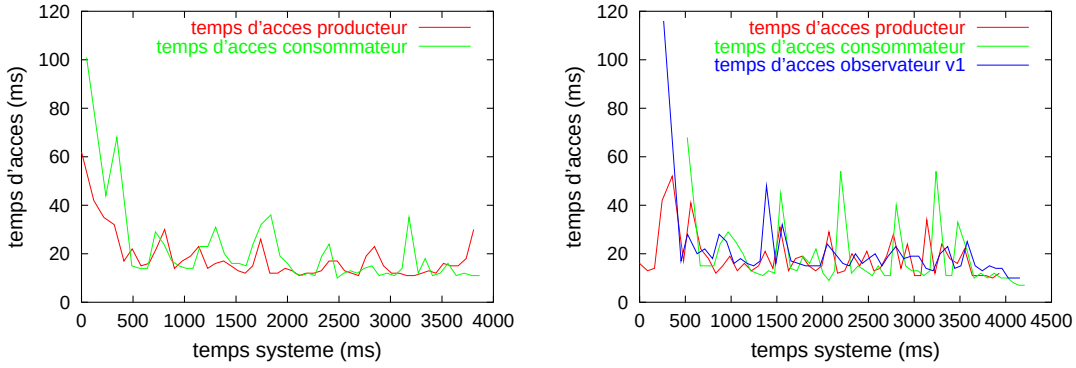


FIG. 6.3 – Temps d'accès à une donnée partagée de 1Ko pour (a) un producteur et un consommateur. (b) montre que l'ajout d'un observateur se comportant comme un consommateur ne dégrade pas les performances.

6.4 Ajout d'un observateur sans lectures-écritures concurrentes

La deuxième mesure permet de constater l'effet de l'ajout dans le second LDG d'un observateur se comportant comme un consommateur (observateur v_1). La taille de la donnée partagée est toujours de 1 Ko et celle-ci est répliquée en 1 exemplaire dans le deuxième LDG. Les temps d'accès moyens des producteur, consommateur et observateur sont respectivement 18 ms, 18 ms et 20 ms.

Ces données montrent que le surcout en terme de latence est faible, voire nul. L'observateur profite ici de deux particularités du protocole de cohérence. Comme dans le premier scénario de test, l'accès à la donnée est équitablement partagé d'un point de vue temporel entre le verrou en écriture et celui en lecture. Il faut de plus souligner l'importance des lectures parallèles pour justifier que l'observateur ne dégrade pas les performances. Ainsi lorsque le producteur relâche le verrou en écriture, le verrou en lecture est distribué aux deux lecteurs qui effectuent leur accès en même temps. Même s'il ne se produit pas de manière systématique, ce dernier point explique que le temps d'accès moyen ne soit pas augmenté.

La figure 6.3(b) donne les temps d'accès des trois sites en fonction du temps système pour chacune des 50 itérations. Les courbes apparaissent plus chaotiques que celles de la figure 6.3(a) et les pics sont dus a une absence de synchronisation des lecteurs : le producteur peut redemander le verrou avant que l'un des lecteurs en aie fait la demande. Dans ce cas, et même si le verrou en lecture est pris, le lecteur retardataire devra attendre que l'écrivain effectue son accès.

Le consommateur et l'observateur commencent respectivement leurs accès 250 et 500 ms après le producteur. En conséquence, ils finissent 250 ms après la fin de l'execution du producteur et l'on constate que les temps d'accès deviennent plus faibles : 10 ms environ, soit 2 fois moins que la moyenne. Lorsque le producteur stoppe son activité, les demandes de verrou en lecture sont immédiatement satisfaites. Ce temps d'accès rapide préfigure des résultats obtenus avec un observateur utilisant l'extension du protocole.

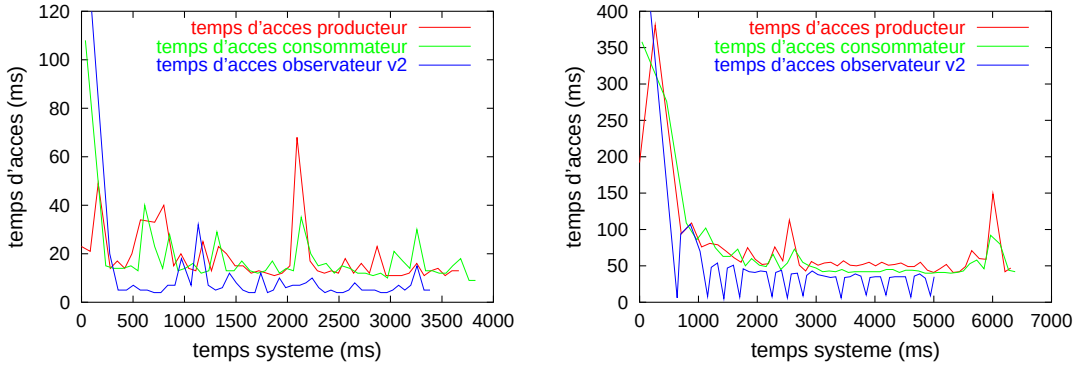


FIG. 6.4 – Temps d'accès à une donnée partagée de (a) 1 Ko et(b) 1024 Ko pour un producteur, un consommateur et un observateur v_2.

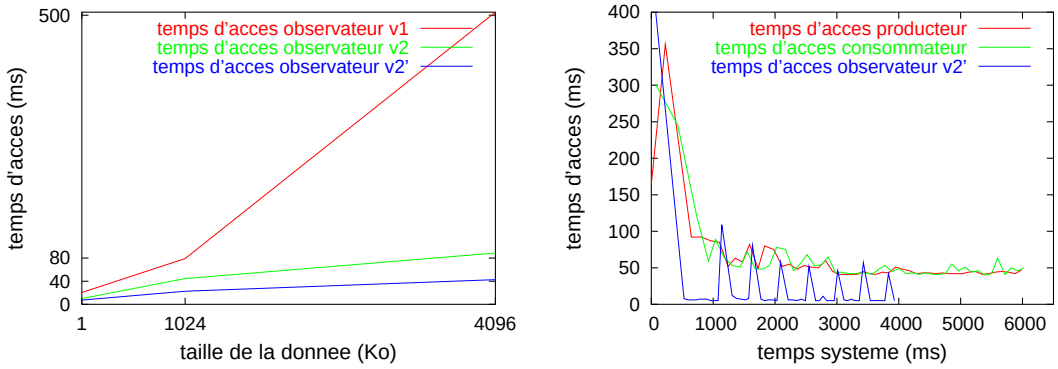
6.5 Ajout d'un observateur avec lectures-écritures concurrentes

Les expérimentations précédentes permettent de constater le fonctionnement du protocole de cohérence dans sa version de base. Elles servent de référence pour évaluer le gain obtenu lors de l'utilisation de l'extension du protocole.

Dans cette troisième mesure, les sites supportant les rôles de producteur et consommateur restent inchangés. L'observateur effectue désormais des appels à la procédure de lecture relâchée *rlxRead*. Le retard maximum autorisé entre les versions de la donnée détenues par les deux LDG est de 0 ($D = 0$). Cela signifie qu'après chaque écriture, la donnée est mise à jour au niveau du GDG, c'est à dire sur tous les LDG. La fenêtre de lecture spécifiée pour les lectures relâchées est de 0 ($w = 0$), ce qui implique que le LDG n'autorise pas l'observateur à utiliser une version de la donnée plus ancienne que la sienne. L'observateur v_2 ne se comporte pas pour autant comme le v_1 : la prise du verrou n'est pas nécessaire pour lire la donnée et les accès peuvent s'effectuer en concurrence avec les écritures. Le fonctionnement du protocole en augmentant la taille de la fenêtre de lecture à 3 ($w = 3$) est analysé dans la section 6.7 avec l'observateur v_2'.

La donnée a une taille de 1 Ko et est répliquée sur les deux LDG avec une seule copie par LDG. Les temps d'accès moyens pour le producteur, le consommateur et l'observateur v_2 sont respectivement 18 ms, 18 ms et 9 ms. La première constatation est que, comme pour l'observateur v_1, les temps d'accès moyens du producteur et du consommateur ne sont pas dégradés. Le deuxième point est que l'observateur v_2 accède en moyenne deux fois plus vite à la donnée que la version v_1. Ce gain de temps est une conséquence directe du faible nombre de messages induits par l'appel à la procédure *rlxRead*.

La figure 6.4(a) donne les temps d'accès des trois sites en fonction du temps système pour chacune des 50 itérations. Le premier accès de l'observateur à la donnée n'est pas plus rapide que celui du consommateur. Lors de ce premier accès, aucune version de la donnée n'est encore présente sur ce site. L'observateur utilise donc les mécanismes de lecture classiques avec la prise du verrou en lecture pour obtenir une première version de la donnée. Ces courbes montrent aussi que même si l'observateur a un temps d'accès deux fois plus petit que les autres sites, il ne termine pas ses 50 lectures en deux fois moins de temps. On en déduit que le temps de journalisation et de traitement des mesures intervenant entre chaque



(a) (b)

FIG. 6.5 – (a) Temps moyens d'accès des différentes versions de l'observateur en fonction de la taille de la donnée partagée. (b) Temps d'accès à une donnée partagée de 1024 Ko pour un producteur, un consommateur et un observateur v_2'.

accès n'est pas négligeable pour cette taille de donnée.

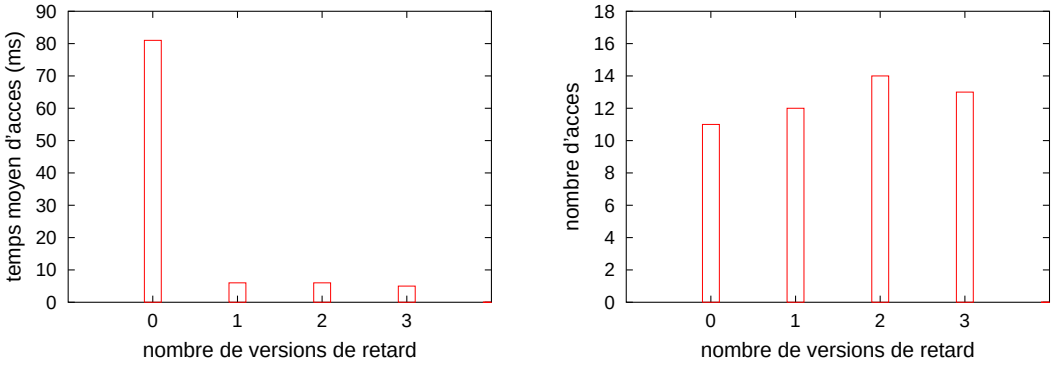
6.6 Variations autour de la taille de la donnée

La taille de la donnée est un paramètre déterminant pour l'évaluation du protocole de cohérence. Des données de petite taille permettent l'envoi fréquent de mises à jour, qu'elles soient nécessaires ou non. A l'inverse, des données de taille plus importante vont considérablement augmenter les temps de transfert réseau des messages qui les transportent. La fréquence des mises à jour devient alors non négligeable pour les performances du protocole.

6.6.1 Nombre de messages et taille de la donnée

Si l'on considère l'experimentation présentée dans la section précédente (*section 6.5*), le temps d'accès moyen de l'observateur est amélioré grâce à la diminution du nombre de messages requis pour réaliser une lecture relâchée et non pas grâce à la diminution du nombre de messages transportant la donnée. Si l'on réitère le test avec une donnée de 1024 Ko, l'envoi de plusieurs petits messages du protocole a beaucoup moins d'influence que l'envoi d'un seul message contenant une donnée. Dans ce cas, les temps moyens des sites producteur, consommateur et observateur sont respectivement 69 ms, 63 ms et 44 ms.

La figure 6.4(b) donne les temps d'accès des trois sites en fonction du temps système pour chacune des 50 itérations. Les paramètres D et w étant définis à 0, l'observateur ne profite que rarement de la version de la donnée détenue dans son cache. Cette optimisation influence la courbe en y formant des creux. La donnée est régulièrement transmise entre le LDG et l'observateur, ce qui explique que l'allure générale de la courbe colle à celle du consommateur. Il faut de plus noter que dans cette configuration, l'observateur met environ un quart de temps en moins que les autres sites pour effectuer ses 50 accès, ce qui est une conséquence directe de son temps moyen d'accès plus faible.



(a) (b)

FIG. 6.6 – Observateur v_2' avec donnée partagée de 1024 Ko. (a) Temps moyen des accès en fonction du nombre de versions de retard retournées. (b) Nombre d'accès pour chaque nombre de versions de retard.

6.6.2 Différents observateurs, différentes tailles de données

Pour chacune des versions de l'observateur, des experimentations ont été effectuées en prenant en compte trois tailles différentes de donnée : 1 Ko, 1024 Ko et 4096 Ko. La figure 6.5(a) propose les temps moyens d'accès des observateurs en fonction de la taille de la donnée. Là où l'observateur v_1 adopte un comportement exponentiel, la version v_2 reste dans une allure linéaire. L'utilisation du système de lecture relâchée permet donc d'améliorer les temps d'accès en lecture sur les données partagées et ceci de facon plus significative que la donnée a une taille importante. Un tableau récapitulatif des mesures effectuées est proposé en figure 6.2.

L'observateur v_2 accélère considérablement ses accès en acceptant de ne pas être l'utilisateur excusif de la donnée. En pratique, il ne permet cependant pas beaucoup de retard entre la version la plus à jour et celle retournée par la procédure *rlxRead*. C'est ce qu'experimente l'observateur v_2' présenté dans la section suivante.

6.7 Autoriser la lecture de données plus anciennes

Cette section traite d'une fonctionnalité intéressante proposée par l'extension du protocole de cohérence : la possibilité de spécifier un nombre de versions de retard avant la mise à jour des copies sur les autres sites. La mesure présentée ici utilise une donnée partagée de 1024 Ko ainsi que le scénario classique déployant un producteur, un consommateur et un observateur. Le nombre de versions produites dans la premier LDG avant qu'une mise à jour ne soit effectuée dans le second est de 3 ($D = 3$). Ce changement n'affecte pas la fréquence des mises à jour puisque lorsque le producteur rend le verrou en écriture, aucun autre site appartenant à ce LDG n'attend ce verrou. Il n'y a donc pas de situation de famine des lecteurs et les modifications sont propagées après chaque écriture.

Le retard entre les versions détenues par l'observateur et son LDG est maintenant de 3. La fenêtre de lecture indiquée pour chaque lecture relâchée est donc de 6 ($w = 6$).

6.7.1 Temps d'accès améliorés

Les temps d'accès moyens des producteur, consommateur et observateur v_2' sont respectivement de 60 ms, 61 ms et 23 ms. En compararant ces temps avec ceux de la section 6.6.1, on remarque que si les performances des sites producteur et consommateur n'ont pas changé, celles de l'observateur ont été considérablement améliorées : le temps d'accès de v_2' est 2 fois plus court que celui de v_2 (44 ms) et presque 4 fois plus court que celui de v_1 (79 ms).

La figure 6.5(b) donne les temps d'accès des trois sites en fonction du temps système pour chacune des 50 itérations. Elle montre que l'observateur v_2' effectue plusieurs lectures rapides d'environ 10 ms entrecoupées d'accès plus lents signifiant que la donnée est transférée. Ces pics apparaissent toutes les 3 écritures effectuées par le producteur. Après chaque modification du producteur, la donnée est mise à jour sur le deuxième LDG. Lorsque la fenêtre de lecture est dépassée, la donnée détenue par l'observateur est rendue obsolète.

Le lien entre la version de la donnée lue par l'observateur et le temps moyen d'accès est illustré par la figure 6.6(a). La récupération d'une donnée à jour entraine des performances d'accès proches d'un observateur v_1 : 81 ms en moyenne pour un retard de 0. Les données retournées avec un retard plus important sont en moyenne lues en 6 ms.

6.7.2 Répartition des versions de la donnée

La figure 6.6(a) permet de voir qu'il n'est pas plus coûteux de récupérer une donnée périmée de 1 version qu'une donnée périmée de 3 versions. Mais qu'en est-il de la distribution des accès ? La figure 6.6(b) montre que sur les 50 lectures, l'observateur v_2' récupère, avec des fréquences sensiblement identiques, toutes les versions possibles. De plus, ces retards sont distribués dans le temps : la figure 6.7 met en relation, pour chaque itération, le temps d'accès de l'observateur v_2' et le retard de version de la donnée retournée. Les pics de la courbe des temps d'accès donnent bien lieu à la lecture d'une donnée récente et les plateaux à des données anciennes. Les lectures retournent des valeurs de plus en plus périmées jusqu'à ce que le transfert d'une donnée récente soit nécessaire.

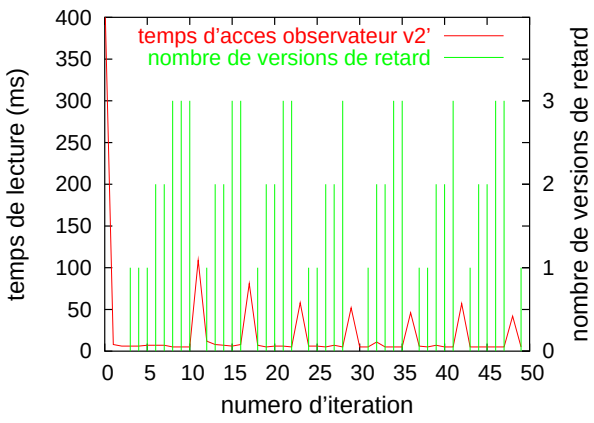


FIG. 6.7 – Donnée partagée de 1024 Ko. Temps d'accès et retard de la version lue d'un observateur v_2' pour chaque itération.

6.8 Passage à l'échelle

Tout au long de son développement et de son évaluation, le protocole de cohérence a été soumis à des tests impliquant plusieurs dizaines de nœuds. L'utilisation de *JDF* a permis le déploiement de JuxMem sur des grappes appartenant à des sites différents.

6.9 Discussion

L'évaluation de ce protocole de cohérence a montré dans un premier temps que l'implémentation de la version de base était tout à fait fonctionnelle. Les données peuvent être lues et écrites dans le respect des règles imposées par le modèle de cohérence à l'entrée. Des accès fréquents et concurrentiels sont pris en charge par les mécanismes de synchronisation du protocole qui ne mènent plus à des situations d'inter-bloquage.

L'extension du protocole proposant une implémentation du concept des lectures relâchées est opérationnelle. Un site client peut lire une donnée pendant une écriture et profiter de son cache ou des réplicas disponibles sur son LDG. Le gain de temps est d'autant plus important que la taille de la donnée partagée est grande. Pour profiter de ces améliorations, l'utilisateur doit accepter de lire des données considérées comme anciennes. Les évaluations ont montré qu'il n'y avait pas de déséquilibre entre les versions des données retournées : l'observateur v'_2 récupère des versions récentes et anciennes. Cette propriété risque cependant de changer si la taille de la donnée est modifiée et elle peut faire l'objet de travaux futurs.

Les résultats présentés ici sont le fruit de choix expérimentaux ne refletant pas toutes les possibilités d'utilisation du protocole. Ainsi le nombre d'itérations a-t-il été fixé à 50 en prenant en compte les premiers accès. Il en résulte que les temps mesurés subissent l'inertie de la mise en route de la machine virtuelle *Java* et ne correspondent pas à un fonctionnement du système en régime permanent.

Plusieurs points font l'objet de travaux futurs. Tout d'abord l'évaluation du protocole de cohérence sur un environnement multi-sites très prochainement disponible dans Grid'5000. Ces experiences doivent mettre en valeur l'aspect hiérarchique du protocole en calquant le modèle logique composé de LDG sur les grappes de calculateurs. Par ailleurs, le retard des versions entre les LDG n'a pas été provoqué et reste donc sans évaluation. L'aspect dynamique de la fenêtre de lecture qui peut être passée en paramètre de la procédure *rlxRead* n'a pas non plus été exploité. Enfin, un ensemble de tailles de données plus grand permettrait d'affiner les courbes proposées par la figure 6.5(a).

Chapitre 7

Conclusion

Les grilles de calculateurs sont désormais considérées comme une alternative sérieuse aux super-calculateurs. Leur exploitation reste limitée par le fait qu'aucun environnement de programmation ne leur est pour l'instant dédié. L'une des difficultés principales est que les données sont gérées par l'utilisateur sans notion de transparence de localisation. Le service de partage de données JuxMem offre aux applications la possibilité de gérer des données modifiables, de manière totalement transparente pour l'utilisateur, dans un environnement de type grille.

Les applications susceptibles d'utiliser JuxMem sont basées sur le couplage de code dans le domaine du calcul haute performance. La surveillance du bon déroulement d'un calcul nécessite d'utiliser un site en lui faisant jouer le rôle d'observateur. Ce rôle ne doit pas provoquer de perturbations dans le calcul, notamment en ralentissant les écritures. Il ne doit pas non plus être affublé de temps d'accès importants, sous peine de ne plus remplir ses tâches de surveillance.

7.1 Contributions

Le premier protocole de cohérence utilisé dans JuxMem implémente le modèle de cohérence à l'entrée. Il a la charge de distribuer les accès en lecture ou en écriture aux sites qui en font la demande. Dans une première étape, nous avons analysé son fonctionnement à l'aide d'automates et de diagrammes *UML*. Ceci a permis de proposer une amélioration face au scénario de l'observateur : autoriser les lectures en parallèle d'une écriture. Ces lectures, nommées *lectures relâchées (rlxRead)*, ne sont possibles qu'en diminuant les contraintes sur la version de la donnée retournée. L'utilisateur accepte alors de lire une version de la donnée "légèrement" ancienne. Il doit spécifier, en paramètre de chacun de ses accès relâchés, le retard maximum autorisé entre la version retournée et la version la plus récente de la donnée.

Pour traduire cette idée, nous proposons une extension du modèle de cohérence. Cette extension ne remet nullement en cause le modèle de cohérence à l'entrée. Les accès aux données effectués en prenant le verrou garantissent toujours de récupérer la version la plus récente. Les lectures relâchées permettent alors de proposer des garanties sur une lecture effectuée sans la prise du verrou.

Une implémentation de cette stratégie a été effectuée au sein de la plate-forme JuxMem. Elle a donné lieu à une extension du protocole initial permettant de réaliser l'appel à la pri-

mitive de lecture relâchée *rlxRead* ainsi que sa prise en charge au niveau du LDG. Plusieurs améliorations ont été apportées au protocole de base, notamment la synchronisation des procédures pour le rendre plus fiable et la gestion partagée par le LDG et le GDG de la version de la donnée sur un même pair. Nous avons écrit une application synthétique utilisant JuxMem pour réaliser les tests et les mesures de performances. Cette application met en jeu les différents comportements du protocole et peu servir d'outil facilement paramétrable pour de futurs développements de JuxMem.

Nous avons évalué les performances sur le site rennais de la plate-forme expérimentale Grid'5000. Celles-ci ont montré que les lectures relâchées permettaient un gain de temps considérable par rapport à une lecture avec prise du verrou. Ceci est d'autant plus vrai que la taille de la donnée est importante ou que la fenêtre de lecture est grande. Par ailleurs ce gain de performance ne se réalise pas au détriment des temps d'accès des sites utilisant les primitives d'accès classiques. La lecture relâchée répond donc aux critères de lecture d'un observateur de couplage de code.

7.2 Ouverture

Bien qu'implémenté et opérationnel, l'aspect dynamique de la fenêtre de lecture indiquée en paramêtre de chaque lecture relâchée n'a pas été experimenté. Cette fonctionnalité peut cependant être à la base d'un mécanisme d'auto-adaptabilité du protocole de cohérence au regard de la charge du réseau et de l'exigence du client. Un calcul peut ainsi présenter des phases pendant lesquelles une observation fréquente des résultats est nécessaire et dans ce cas l'application réduira la taille de sa fenêtre de lecture. Inversement, une phase de calcul nécessitant une grande utilisation du réseau sera observée avec plus de distance pour ne pas ralentir son fonctionnement.

La technique des versions d'une donnée proposée dans ce rapport est inspirée du *versioning* partiel du monde des bases de données. L'expression du retard s'effectue ici en relatif par rapport à la donnée la plus récente. Le numéro de la version retournée échappe cependant à l'utilisateur. Un modèle de programmation distribué intéressant peut proposer au développeur des primitives de lecture admettant en paramètre le numéro de la version à retourner. Dans un tel modèle, l'implémentation du schéma producteur-consommateur ne demande plus aucun effort de synchronisation au programmeur qui indique tout simplement que son consommateur lit un ensemble de versions de la donnée. Cette stratégie implique au service de partage des données de conserver les versions susceptibles d'être demandées. JuxMem ne gère pas actuellement un système de versions complet : le fait que plusieures versions cohabitent en un instant n'est qu'un artéfact de l'aspect paresseux du protocole de cohérence.

Ce projet de fin d'étude ouvre d'autres perspectives de recherche sur la gestion du partage de données dans les grilles de calculateurs. Les méthodes de travail, la rigueur de la recherche scientifique, la transmission du savoir et l'esprit d'équipe sont autant de points positifs qui me poussent à continuer dans cette voie au sein de ce projet.

Annexe A

Analyse du protocole de cohérence

A.1 Le Client

L'automate du client est présenté en figure 3.4. Les actions suivantes sont déclenchées par les transitions de l'automate.

0. init
```
localData = null; // donnée locale
dataThere = false; // indique si la donnée est présente sur le site
modifs = false; // indique si la donnée a été modifiée
```
1. acquire
```
send(LDG, L_LOCK); // envoie au LDG un message de tag L_LOCK
```
2. acquireR
```
send(LDG, L_LOCK_R);
```
3. release $\wedge$ update // update est équivalent à modifs
```
localData.version++;
send(LDG, L_UPDATE, localData); // envoie au LDG les données modifiées
modifs = false;
```
4. release $\wedge$ $\neg$update, L_UPDATE
```
send(LDG, L_UNLOCK);
```
5. release
```
send(LDG, L_UNLOCK_R);
```
6. read $\wedge$ dataThere
```
return localData;
```
7. read $\wedge$ $\neg$dataThere
```
send(LDG, L_READ);
```
8. L_READ
```
modifs = false;
dataThere = true;
localData = message.data; // récupère les données contenues dans le message recu
return localData;
```
9. write
```
dataThere = true;
modifs = true;
```
10. L_LOCK, L_LOCK_R
```
IF (localData.version < message.dataVersion) THEN
 dataThere = false;
 localData = null;
 modifs = false;
 localData.version = message.dataVersion;
ENDIF
```

A.2 Groupe de données local (LDG)

L'automate du LDG est présenté en figure A.1.

0. ECLocalDataGroup

```
localData = null;
dataThere = false;
modifs = false;
L = {};   // liste des clients en attente du verrou exclusif
LR = {};  // liste des clients en attente du verrou en lecture
OLR = {}; // liste des clients en cours de lecture
```

1. L_READ

```
send(clientID, L_READ, localData);
```

2. G_LOCK

```
localData.version = message.dataVersion;
send(L.getFirst(), L_LOCK, localData.version);
```

3. L_UNLOCK $\wedge\ |L| > 1$

```
// clientID == L.getFirst()
L.removeFirst();
send(L.getFirst(), L_LOCK, localData.version);
send(clientID, L_RELEASE);
```

4. L_UNLOCK $\wedge\ |L| = 1$

```
// clientID == L.getFirst()
L.removeFirst();
IF (modifs) THEN
 send(GDG, G_UPDATE, localData);
 modifs = false;
ENDIF
send(clientID, L_RELEASE);
```

5. G_LOCK_READ

```
localData.version = message.dataVersion;
OLR = LR;
LR = {};
sendAll(OLR, L_LOCK_R, localData.version);
```

6. L_UNLOCK_R $\wedge\ |OLR| > 1$

```
OLR.remove(clientID);
send(clientID, L_RELEASE);
```

7. L_UNLOCK_R $\wedge\ |OLR| = 1$

```
OLR.remove(clientID);
send(GDG, G_UNLOCK_READ);
send(clientID, L_RELEASE);
```

8. G_UPDATE

```
localData = message.data;
modifs = false;
```

9. L_LOCK

```
IF (L.isEmpty()) THEN
 send(GDG, G_LOCK);
ENDIF
L.addLast(clientID);
```

10. L_LOCK_R

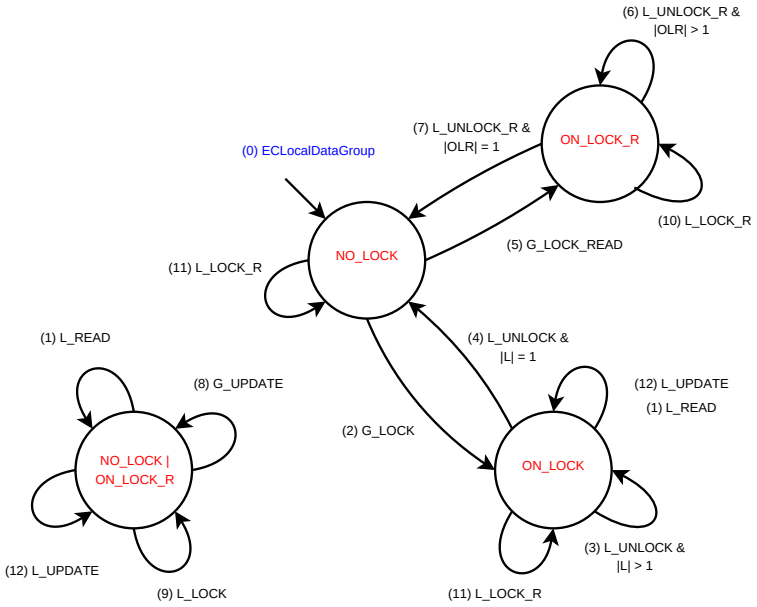


FIG. A.1 – Automate du groupe de données local (LDG).

```
IF (L.isEmpty()) THEN
 OLR.addLast(clientID);
 send(clientID, L_LOCK_R, localData.version);
ELSE
 LR.addLast(clientID);
ENDIF
```

11. L_LOCK_R

```
IF (LR.isEmpty()) THEN
 send(GDG, G_LOCK_READ);
ENDIF
LR.addLast(clientID);
```

12. L_UPDATE

```
localData = message.data;
modifs = true;
send(clientID, L_UPDATE);
```

A.3 Groupe de données global (GDG)

L'automate du LDG est présenté en figure A.2.

0. ECGlobalDataGroup

```
L = {};   // liste des LDG en attente du verrou exclusif
LR = {};  // liste des LDG en attente du verrou en lecture
OLR = {}; // liste des LDG ayant le verrou en lecture
```

1. G_LOCK

```
L.addLast(ldgID);
send(L.getFirst(), G_LOCK, dataVersion);
```

2. G_LOCK_READ, G_LOCK_READ $\wedge$ $L = \emptyset$

```
OLR.addLast(ldgID);
send(ldgID, G_LOCK_READ, dataVersion);
```

3. G_UNLOCK $\wedge\ |L| > 1$

```
L.removeFirst();
send(L.getFirst(), G_LOCK, dataVersion);
```

4. G_UNLOCK $\wedge\ |L| = 1 \wedge\ LR \neq \emptyset$

```
L.removeFirst();
OLR = LR;
LR = {};
sendAll(OLR, G_LOCK_READ, dataVersion);
```

5. G_UNLOCK $\wedge\ |L| = 1 \wedge\ LR = \emptyset$

```
L.removeFirst();
```

6. G_UNLOCK_READ $\wedge\ |OLR| > 1$

```
OLR.remove(ldgID);
```

7. G_UNLOCK_READ $\wedge\ |OLR| = 1 \wedge\ L \neq \emptyset$

```
OLR.remove(ldgID);
send(L.getFirst(), G_LOCK, dataVersion);
```

8. G_UNLOCK_READ $\wedge\ |OLR| = 1 \wedge\ L = \emptyset$

```
OLR.remove(ldgID);
```

9. G_LOCK

```
L.addLast(ldgID);
```

10. G_LOCK_READ $\wedge\ L \neq \emptyset$, G_LOCK_READ

```
LR.addLast(ldgID);
```

11. G_UPDATE

```
dataVersion = message.dataVersion;
send( ? , G_UPDATE, ?);
```

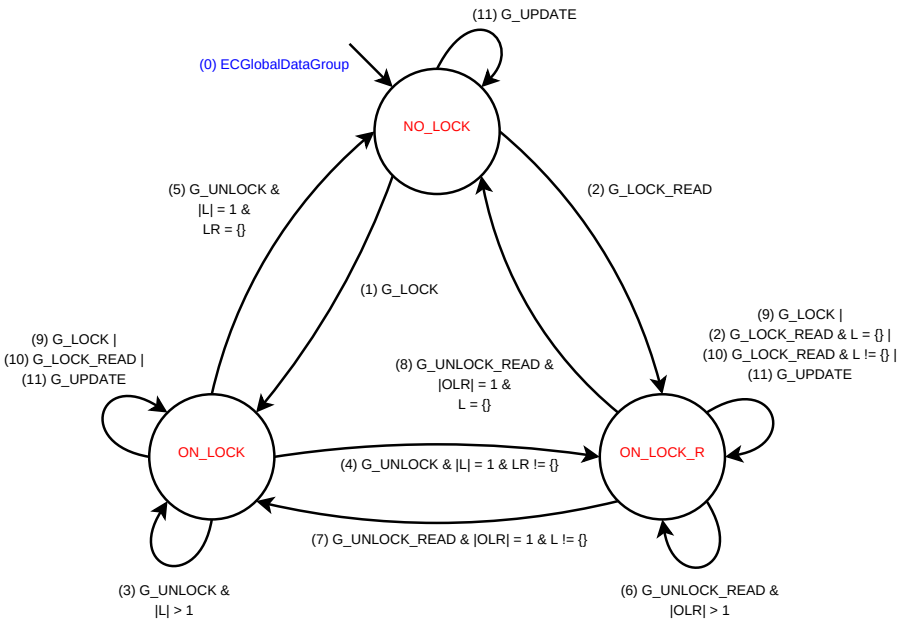


FIG. A.2 – Automate du groupe de données global (GDG).

Bibliographie

[1] The JXTA (juxtapose) project. `http://www.jxta.org`.

[2] Projet Grid'5000. `http://www.grid5000.org`.

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

[4] 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.

[5] Gabriel Antoniu and Luc Bougé. Dsm-pm2: A portable implementation platform for multithreaded dsm consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, volume 2026 of *Lect. Notes in Comp. Science*, pages 55–70, San Francisco, April 2001. Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag.

[6] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. In *Proceedings Workshop on Adaptive Grid Middleware (AGRIDM 2003)*, pages 49–59, New Orleans, Louisiana, September 2003. Held in conjunction with PACT 2003. Extended version to appear in Kluwer Journal of Supercomputing.

[7] Gabriel Antoniu, Luc Bougé, and Mathieu Jan. JuxMem: Weaving together the P2P and DSM paradigms to enable a Grid Data-sharing Service. *Kluwer Journal of Supercomputing*, 2005. To appear. Preliminary electronic version available at URL `http://www.inria.fr/rrrt/rr-5082.html`.

[8] Gabriel Antoniu, Luc Bougé, Mathieu Jan, and Sébastien Monnet. Large-scale deployment in p2p experiments using the jxta distributed framework. In *Euro-Par 2004: Parallel Processing*, number 3149 in Lect. Notes in Comp. Science, pages 1038–1047, Pisa, Italy, August 2004. Springer-Verlag.

[9] 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 (CCGrid 2003)*, pages 516–523, Tokyo, Japan, May 2003. IEEE.

[10] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. Building fault-tolerant consistency protocols for an adaptive grid data-sharing service. In *Proc. ACM Workshop on Adaptive Grid Middleware (AGridM 2004)*, Antibes Juan-les-Pins, France, September 2004. To appear.

[11] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. Technical Report PI 1685, IRISA/INRIA, University of Rennes 1, jan. 2005.

[12] L. Arantes and P. Sens. CLRC : Une mémoire partagée répartie pour des grappes de stations inter-connectées. In *Actes de la 2ème Conférence Française sur les Systèmes d'Exploitation*, pages 8–15, Apr. 2001.

[13] Alessandro Bassi, Micah Beck, Graham Fagg, Terry Moore, James S. Plank, Martin Swany, and Rich Wolski. The internet backplane protocol: A study in resource sharing. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 194, Washington, DC, USA, 2002. IEEE Computer Society.

[14] Brian N. Bershad, Mattew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93)*, pages 528–537, Los Alamitos, CA, February 1993.

[15] 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.

[16] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[17] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[19] 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.

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

[21] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[22] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 277–287, Padova, Italy, June 1996.

[23] 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 *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, November 2000. Springer-Verlag.

[24] Kam-Yiu Lam, Guo Hui Li, and Tei-Wei Kuo. A multi-version data model for executing real-time transactions in a mobile environment. In *MobiDe '01: Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access*, pages 90–97, New York, NY, USA, 2001. ACM Press.

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[26] Kai Li. Ivy: A shared virtual memory system for parallel computing. In *ICPP (2)*, pages 94–101, 1988.

[27] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[28] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[29] Sergio Mena, André Schiper, and Pawel Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of International Middleware Conference*, volume 2672 of *LNCS*, pages 414–432, Rio de Janeiro, Brazil, June 2003. Springer.

[30] C. Mohan, Hamid Pirahesh, and Raymond Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 124–133, New York, NY, USA, 1992. ACM Press.

[31] 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)*, pages 31–44, Boston, MA, December 2002.

[32] Fabio Picconi, Jean-Michel Busca, and Pierre Sens. Exploiting network locality in a decentralized read-write peer-to-peer file system. In *ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference on (ICPADS'04)*, page 289. IEEE Computer Society, 2004.

[33] Michel Raynal, Matthieu Roy, and Ciprian Tutu. Merging atomic consistency and sequential consistency. Technical Report 1629, IRISA, Jun. 2004.

[34] Alexander Yakovlev. A multi-version concurrency control model for distributed mobile databases. In *SYRCoDIS'2004: Proceedings of the Spring Young Researcher's Colloquium on Database and Information Systems*, Saint-Petersburg, Russia, May 2004.

[35] Yuanyuan Zhou, Liviu Iftode, and Kai Li. 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)*, pages 75–88, Seattle, WA, October 1996.