Routage Décentralisé de Transactions avec Gestion des Pannes dans un Réseau à Large Echelle

Idrissa Sarr — Hubert Naacke — Stéphane Gançarski

UPMC Paris Universitas
Laboratoire d'Informatique de Paris 6
104 Avenue du Président Kennedy
75016 Paris, France

Prénom.Nom@lip6.fr

RÉSUMÉ. *Les applications Web 2.0 émergentes comme les réseaux sociaux sont confrontées à de fortes charges. Pour faire face à cette charge, nous envisageons une nouvelle solution qui assure le passage à l'échelle sans l'utilisation d'un centre de données. Dans cette perspective, les données sont répliquées sur une grille informatique pour assurer la disponibilité et le traitement parallèle des transactions Cependant, assurer à la fois un accès efficace et cohérent aux données est un défi majeur à plusieurs niveaux:* 1) *un contrôle centralisé constitue une source de contention et* 2) *la volatilité des nœuds peut compromettre la cohérence des données. Dans cet article, nous proposons une solution de routage distribué des transactions avec un mécanisme de gestion des pannes adéquat pour contrôler la dynamicité des nœuds. Nous démontrons par la suite, la faisabilité de nos approches à travers une implémentation et une simulation.*

ABSTRACT. *Emerging Web 2.0 applications such as social networking websites deal with a heavy workload. We envision a novel solution that would allow these applications to scale-up without the need to use a data center. To this end, databases are replicated over a grid, thus, availability and fast transaction processing are achieved. However, achieving both fast and consistent data access challenging at many points:* 1) *centralized control leads to bottleneck source, and* 2) *dynamic behaviour of nodes can compromise mutual consistency. In this article, we propose a distributed transaction routing algorithm with a suitable fault-management mechanism to deal with node dynamicity and/or node failures. Moreover, we demonstrate the feasibility of our approaches through experimentation and simulation.*

MOTS-CLÉS : *Bases de données répliquées, intergiciel, tolérance aux pannes.*

KEYWORDS: *Database replication, middleware, fault-management.*

1. Introduction

Les systèmes à large-échelle tels que les Grilles (*Grids*) utilisent une approche répartie pour résoudre les problèmes d'hétérogénéité, d'échelle et d'autonomie des ressources. Ils sont donc intéressants dans de nombreux domaines d'application émergents du Web 2.0, tels que les mondes virtuels (Second Life, n.d.) ou les réseaux sociaux (MySpace, n.d.). On peut observer deux particularités dans les applications du Web 2.0, qui les distinguent des applications standard spécifiées par le *Transaction Processing Council* (TPC-Council, 2007). Premièrement, les applications web 2.0 exécutent les transactions d'une manière très spécifique : la plupart des transactions sont encapsulées dans les appels de procédure d'une interface de programmation. Par exemple, dans SecondLife (Second Life, n.d.), les utilisateurs peuvent invoquer les procédures fournies par le monde virtuel afin de mettre en œuvre le comportement de leurs avatars. Il est donc possible, grâce à cette prédéfinition des accès, de déterminer avec précision les données auxquelles les transactions vont accéder, avant de démarrer leur exécution. Deuxièmement, bien que simultanées, les transactions générées par les applications Web 2.0 sont très peu concurrentielles. Par exemple, toujours dans SecondLife, moins d'un millième (en moyenne) des utilisateurs connectés ont leur avatar dans un endroit occupé par celui d'un autre utilisateur. En ce qui concerne la charge, nous supposons que certaines applications Web 2.0 vont avoir, pour le moins, le même succès croissant que par exemple eBay, l'application la plus populaire du web aujourd'hui. Dans (Shoup, 2007), eBay annonce 100 millions d'items, classés en 50 000 catégories, représentant une masse de données de 2 000 teraoctets. Chaque jour, la base de données sous-jacente exécute 130 millions d'appels de procédures, ce qui correspond à une charge moyenne de 1 500 transactions par seconde. Le défi engendré par de telles charges est d'assurer à la fois la disponibilité et la cohérence des données tout en exécutant les transactions très rapidement. Pour résoudre ce problème, les applications utilisent actuellement des serveurs parallèles très onéreux. De plus, les données sont centralisées sur quelques sites, ce qui limite le passage à l'échelle et la disponibilité. Dans cet article, nous présentons une solution alternative qui doit permettre aux applications de la future génération de supporter des milliers de transactions par seconde sans qu'il soit nécessaire d'investir dans de coûteux serveurs parallèles.

L'utilisation d'une grille pour mettre en œuvre les applications du Web 2.0 apparaît comme économiquement viable. Dans une telle architecture, les données de l'application sont stockées par les institutions participantes (entreprises, universités...) et peuvent être partagées. Les données sont donc réparties et les transactions effectuées en parallèle permettent un meilleur équilibrage de la charge. Pour augmenter la disponibilité des données, celles-ci sont répliquées et les transactions routées vers les répliques les plus intéressantes. Cependant, il est nécessaire de contrôler la cohérence mutuelle des répliques qui peut être mise à mal par les mises à jour concurrentes et par les éventuelles pannes des nœuds de la grille.

Pour illustrer le premier problème, supposons deux répliques R_1^i et R_2^i de la relation R^i, et deux transactions T_1 et T_2 envoyées par deux utilisateurs de SecondLife U_1 et U_2. Chaque transaction cherche à louer une parcelle de 512 m^2 sur la même île.

Supposons qu'une seule parcelle de cette taille soit disponible sur l'île et que T_1 (resp. T_2) soit routée sur R_1^i (resp. R_2^i). Une exécution simultanée de T_1 et T_2 produit une incohérence : l'un des deux utilisateurs aura loué une parcelle indisponible.

Supposons maintenant qu'une application A envoie une transaction T pour acheter une parcelle. T est envoyée à R_1^i qui l'exécute mais tombe en panne avant d'envoyer les résultats à A. Au bout d'un moment, A renvoie T qui est donc routée sur R_2^i. Par conséquent, T est exécutée deux fois et A aura acheté deux parcelles au lieu d'une seule. Il est donc nécessaire de contrôler le comportement du système en cas de panne.

Un autre point important est que le contrôle de la fraîcheur des nœuds permet un meilleur équilibrage de charge. En effet, certaines requêtes en lecture seule peuvent très bien se satisfaire de données non parfaitement à jour. Par exemple, une requête calculant le nombre total de parcelles disponibles (plusieurs milliers par exemple), peut être exécutée sur un nœud légèrement obsolète mais peu chargé, et rendra rapidement un résultat très proche du nombre réel. Une telle approche est possible si les deux conditions suivantes sont satisfaites : *(i)* l'obsolescence du nœud choisi n'excède pas un seuil donné (associé à la requête), et *(ii)* la requête ne fait pas de mise à jour, afin que l'erreur ne se propage pas.

Dans (Sarr *et al.,* 2008), nous proposons une solution appelée *Distributed Transaction Routing* (DTR) qui garantit la sérialisabilité et contrôle la fraîcheur afin d'améliorer les performances. Dans sa version originale, DTR ne tient pas compte de la dynamicité (déconnexions, nœuds en panne) pourtant caractéristique des systèmes à large échelle. Prendre en compte la dynamicité est un véritable défi. Il s'agit de concevoir des algorithmes efficaces pour gérer les pannes afin de garantir un bon niveau de disponibilité, en relançant automatiquement les transactions ayant échoué pour cause de panne.

Un grand nombre de solutions ont été proposées dans le domaine des systèmes répartis pour gérer la réplication à large échelle, telles que (Patino-Martinez *et al.,* 2005, Pacitti *et al.,* 2005, Pacitti *et al.,* 1999). Certaines solutions incluent le contrôle de fraîcheur, comme par exemple (Rohm *et al.,* 2002, Gançarski *et al.,* 2007, Le Pape *et al.,* 2004, Akal *et al.,* 2005, Ramamritham *et al.,* 1995). D'autres solutions incluent des mécanismes de tolérance aux pannes, comme (Antoniu *et al.,* 2006, Guerraoui *et al.,* 1997, Koo *et al.,* 1987). Nous basons notre travail sur l'approche Leg@net (Gançarski *et al.,* 2007), qui propose de mettre à jour n'importe quelle réplique tout en contrôlant la fraîcheur. L'approche Leg@net ne nécessitant aucune modification du SGBD sous-jacent, ni du code de l'application, elle permet une grande portabilité et une adaptation à coût minimum.

Cependant, Leg@net a été conçu pour des grappes (*clusters*) de PC, mettant en œuvre un système parallèle homogène. Transposer cette approche à des systèmes hétérogènes dont les entités sont autonomes, ne se fait pas de manière directe. Or nous prévoyons que de nombreuses applications Web 2.0 utiliseront très prochainement de tels sytèmes. Il est nécessaire de modifier l'architecture Leg@net en répliquant le mo-

dule de routage ainsi que les métadonnées sur différents nœuds, et de mettre en œuvre la gestion des pannes et des connexions/déconnexions fréquentes.

Dans cet article, nous décrivons la conception d'un nouveau système, basé sur l'approche Leg@net, capable de contrôler (routage, gestion des pannes) l'exécution de transactions à très large échelle. Les principales contributions sont les suivantes :

– un modèle de routage de transaction complètement décentralisé pour les applications à fort taux de mise à jour. L'intergiciel qui met en œuvre ce modèle garantit la transparence de la répartition/réplication des données. Nous utilisons la réplication asynchrone, ce qui fait que notre système ne nécessite pas de protocole de validation à deux phases ni de communication de groupe pour valider les mises à jour. De plus, nous intégrons le contrôle de fraîcheur, ce qui permet d'améliorer les performances par un meilleur équilibrage de charge ;

– un répertoire réparti à large-échelle, et donc hautement disponible, pour le stockage des métadonnées. L'accès aux métadonnées étant optimisé, notre système maintient la cohérence des données avec peu de messages de communication entre les (instances de) routeurs ;

– un mécanisme de tolérance aux fautes fonctionnant à large échelle. Ce mécanisme est basé sur la détection sélective des fautes et sur un algorithme de reprise. Contrairement à la plupart des autres approches, notre mécanisme n'implique pas l'utilisation de nœuds qui ne participent pas à l'exécution de la transaction en cours ;

– une mise en œuvre et une évaluation expérimentale de notre approche, aussi bien sur une infrastructure réelle de taille moyenne que par des simulations à large échelle. Ceci prouve la faisabilité de notre solution et quantifie les bénéfices qu'elle apporte en termes de performances.

La suite de cet article est organisée de la manière suivante. Dans la section 2, nous présentons l'architecture globale du système, ainsi que le modèle de réplication et de fraîcheur. La section 3 décrit l'algorithme de routage avec contrôle de la fraîcheur. La section 4 traite de la dynamicité des nœuds. La section 5 présente la mise en œuvre de notre solution et les évaluations expérimentales que nous avons menées sur le routage et le contrôle de fraîcheur. La section 6 se focalise sur les performances observées de notre mécanisme de tolérance aux pannes. Les travaux connexes aux nôtres sont résumés dans la section 7. La section 8 conclut.

2. Modèles et architecture du système

Dans cette section, nous présentons la structure et les spécifications de notre système. Nous décrivons dans un premier temps l'architecture globale du système pour faciliter la compréhension de notre solution. Ensuite, nous présentons le modèle de réplication et de traitement des transactions dans la section 2.2. Le modèle de relâchement de la fraîcheur et la gestion de la cohérence globale sont discutés respectivement dans les section 2.3 et section 2.4. Enfin, nous présentons dans la section 2.5 le modèle des pannes prises en compte dans cet article.

2.1. *Architecture globale*

L'architecture de notre système est schématisée sur la figure 1. Tous les nœuds sont identifiés par leur adresse IP et communiquent entre eux par le biais de messages asynchrones. Nous distinguons quatre types de nœuds.

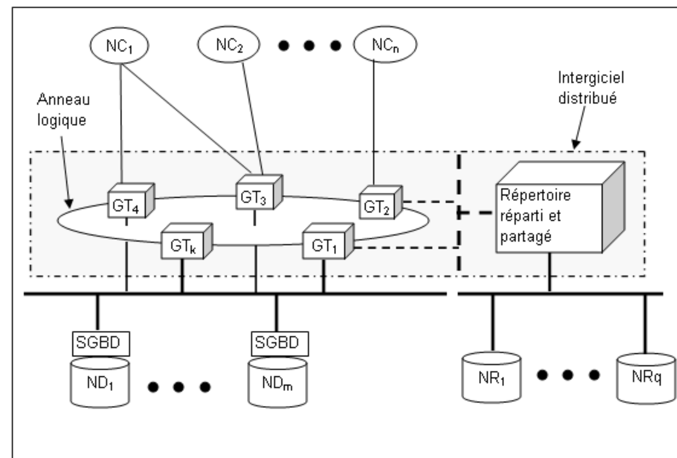

Figure 1. *Architecture globale*

Les nœuds client (*NC*) envoient des transactions aux différents GT. Un NC associe à chaque transaction un identifiant unique global utilisé pour éviter des ambiguïtés durant les différentes phases d'exécution d'une transaction. L'identifiant global est défini par la paire <Id, NS>, où Id est l'identifiant du client et NS le numéro de séquence local de la transaction.

Les nœuds gestionnaire des transactions (*GT*) transmettent les transactions pour exécution sur les nœuds de données tout en préservant la cohérence globale. Les GT utilisent les métadonnées stockées sur le répertoire réparti. Les GT sont organisés sous forme d'un anneau logique (Larrea *et al.,* 1999) pour faciliter la détection collaborative des pannes. Chaque GT est directement relié avec k prédécesseurs et k successeurs. La valeur de k dépend du temps maximal pour propager la panne d'un GT vers ses voisins.

Les nœuds de données (*ND*) utilisent un système de gestion de bases de données local pour stocker les données et exécuter les transactions envoyées par les GT. Ils retournent les résultats directement aux NC.

Les nœuds gestionnaire du répertoire réparti (*NR*) sont les nœuds sur lesquels est stocké le répertoire réparti. Ils sont implémentés avec JuxMem (Antoniu *et al.,* 2005). JuxMem fournit un service transparent et cohérent de partage de données sur une grille informatique. Le répertoire réparti contient les informations relatives aux transactions

envoyées sur les ND, *i.e.* pour chaque relation et chaque nœud, la liste des transactions courantes et/ou déjà exécutées. Il est utilisé pour calculer l'obsolescence d'une réplique comme décrit dans la section 2.3. Le répertoire réparti stocke aussi, pour chaque transaction T, le temps estimé pour exécuter T, qui est une moyenne variable obtenue par l'exécution des précédentes exécutions de T. Il est initialisé par une valeur par défaut en exécutant T sur un nœud non chargé. Cette mesure est indispensable pour effectuer le routage et l'équilibrage de la charge de mise à jour (voir section 3.2).

2.2. *Modèle de réplication et de gestion des transactions*

Nous considérons une base de données unique composée de n relations $R^1, ..., R^n$ et totalement répliquée sur m nœuds de données $N_1, ..., N_m$. Une copie locale de R^i sur un nœud N_j est notée par R^i_j et gérée par le SGBD local. Nous utilisons un modèle de réplication asynchrone multimaître. Chaque ND peut être mis à jour par une transaction entrante et est appelé ensuite le nœud initial de cette transaction. Les autres ND sont mis à jour plus tard par propagation de la transaction. Nous distinguons trois types de transactions :

– une transaction de mise à jour est une séquence d'instructions de SQL dont au moins une d'entre elles modifie la base de données ;

– Une transaction de rafraîchissement est utilisée pour propager les transactions de mise à jour sur les autres ND. En d'autres mots, elle ré-exécute une transaction de mise à jour sur un ND autre que le ND initial. Pour distinguer les transactions de rafraîchissement des transactions de mise à jour, on mémorise dans le répertoire réparti, pour chaque ND, les transactions déjà routées sur ce ND ;

– Une requête effectue une simple lecture de la base de données. Ainsi, il n'est pas nécessaire de la propager.

Puisque nous avons considéré une réplication totale de la base de données, alors les transactions réparties sont exclues de cette étude. Chaque transaction est entièrement exécutée sur un seul ND.

2.3. *Modèle de fraîcheur*

Chaque transaction (mise à jour, rafraîchissement, requête) lit un certain nombre de relations. Cette information est obtenue en analysant le code de la transaction.

Les requêtes peuvent accéder à des données obsolètes dont l'obsolescence est contrôlée par les applications. L'obsolescence peut être définie de différentes manières (Le Pape *et al.,* 2004). Dans cet article, nous considérons une seule mesure à savoir le nombre de transactions de mise à jour manquantes. Précisément, l'obsolescence de R^i_j est égale au nombre de transactions de mise à jour modifiant R^i sur un ND quelconque mais non encore propagée sur le nœud N_j. L'obsolescence tolérée d'une requête est donc, pour chaque relation lue par la requête, le nombre de transactions qui ne sont

pas encore exécutées sur le nœud traitant la requête. L'obsolescence tolérée reflète le niveau de fraîcheur requis par une requête pour s'exécuter sur un ND. Par exemple, si une requête requiert des données parfaitement fraîches, alors l'obsolescence tolérée est zéro. Pour des raisons de cohérence, les transactions de mise à jour ainsi que celles de rafraîchissement doivent lire des données parfaitement fraîches.

Nous calculons l'obsolescence de la copie d'une relation R_j^i en s'appuyant sur l'état global du système stocké dans le répertoire réparti, qui donne des informations détaillées sur les transactions courantes ou déjà exécutées.

2.4. *Cohérence globale*

Avec la réplication asynchrone multimaître, la cohérence mutuelle de la base de données peut être compromise par l'exécution simultanée des transactions conflictuelles sur différents sites. Pour éviter ce problème, les transactions de mise à jour sont exécutées sur les nœuds de la base dans un ordre compatible, produisant ainsi des états cohérents de toutes les répliques de la base de données (cohérence à terme des données). Les requêtes sont envoyées sur n'importe quel nœud, suffisamment frais vis-à-vis des conditions requises par la requête. Ceci implique qu'une requête peut lire différents états de la base de données en fonction du nœud sur lequel elle est exécutée. Néanmoins, les requêtes lisent toujours des états cohérents (probablement obsolètes) du moment qu'elles ne sont pas distribuées. Pour assurer la cohérence globale, nous maintenons un graphe dans le répertoire réparti, appelé graphe de sérialisation global (GSG). Il garde la trace des dépendances conflictuelles entre les transactions actives *i.e.* les transactions courantes mais non encore validées. Il est basé sur la notion de conflit potentiel : une transaction entrante est en conflit potentiel avec une transaction courante si elles manipulent simultanément une même relation et que l'une des transactions effectue au moins une écriture sur cette relation. Cette stratégie de pré-ordonnancement utilisé dans Leg@net, est comparable à celle décrite dans (Bernstein *et al.,* 1987). La principale différence est que le graphe de sérialisation est aussi utilisé pour calculer la fraîcheur dans notre cas.

2.5. *Modèle de pannes*

Dans cet article, nous considérons les systèmes constitués uniquement de deux types de composants : les nœuds qui traitent les transactions (NC, ND et GT), et les canaux de communications. Chacun de ces types de composants peut tomber en panne durant le fonctionnement du système, engendrant ainsi une panne de nœud ou de communication. Dans la suite de ce travail, nous axons notre réflexion sur les types de pannes suivants :

– panne d'un nœud. Quand un nœud tombe en panne, ses traitements s'arrêtent anormalement, ce qui peut conduire à des incohérences. Nous supposons qu'un nœud fonctionne correctement ou s'arrête complètement (il est en panne). En d'autres mots,

nous ne prenons en compte que les pannes de type *fail-stop* et non les pannes byzantines ;

– panne de communication. Une panne de communication survient quand un nœud N_i est incapable de contacter le nœud N_j, bien qu'aucun d'entre eux ne soit en panne. Si une telle panne survient, aucun message n'est délivré. Dans notre contexte, la communication est asynchrone et chaque message reçu par un nœud doit être acquitté. Sans cet acquittement, nous supposons que le message est perdu à cause d'une panne de communication ou d'un nœud.

Détection des pannes. En général, les pannes sont détectées soit par des messages périodiques de type *heartbeat* (Aguilera *et al.,* 1999), soit à la demande par des messages *ping-pong* (Larrea *et al.,* 1999). (Chandra *et al.,* 1996) présentent les principes de détection collaborative pour les systèmes à large échelle. Nous nous inspirons de ces travaux en combinant l'utilisation des messages *heartbeat* et *ping-pong*. En effet, nous utilisons une détection de pannes à la demande pour les ND et une détection périodique pour les GT. L'utilisation des messages périodiques pour détecter les pannes des GT se motive par le nombre réduit de GT comparé à celui des ND, générant moins de messages. En outre la détection des pannes de ND se fait par collaboration entre les GT, d'où l'importance de détecter le plus tôt possible une panne de GT. Par contre l'utilisation de cette technique pour détecter les pannes des ND engendrerait beaucoup de bande passante du fait de leur nombre important. Ainsi, pour détecter les pannes des ND sans surcoût significatif, on intègre la détection dans le protocole de routage utilisant la méthode *ping-pong*. De ce fait, un ND en panne n'est détecté que si un GT essaie de lui envoyer une transaction. Pour prendre en compte des pannes survenant lors de l'accès au répertoire réparti, nous nous appuyons sur le gestionnaire du répertoire réparti (à savoir Juxmem) qui empêche qu'un lecteur (ou écrivain) en panne verrouille l'accès aux données indéfiniment.

3. Routage des transactions et contrôle de la fraîcheur

Dans cette section, nous décrivons comment les transactions sont routées de manière efficace. Nous présentons d'abord l'algorithme de routage, inspiré de celui de (Sarr *et al.,* 2008) et (Gançarski *et al.,* 2007). Puis nous discutons des problèmes spécifiques liés à l'utilisation d'un répertoire partagé.

3.1. *Spécification du protocole de routage*

Afin de séparer les problèmes, nous décrivons dans cette section le routage en l'absence de panne (voir la section 4 pour la gestion des pannes). Dans ce contexte, le traitement des transactions peut être décomposé en trois phases principales :

phase d'initialisation : Durant cette phase, un NC envoie une transaction à un GT. Nous supposons que chaque NC connaît certains GT (pas forcément tous) et il choisit par une méthode de tourniquet le GT à contacter parmi ceux qu'il connaît ;

phase de routage : Le GT exécute l'algorithme de routage (voir section 3.2) et envoie ainsi la transaction à un ND ;

phase d'exécution : Pendant cette phase, un ND reçoit une transaction T, l'exécute localement et envoie le résultat au NC qui avait initié la transaction. Il informe aussi le GT qui l'a contacté que T a été correctement exécutée.

La figure 2 représente le déroulement du processus d'exécution de la transaction T.

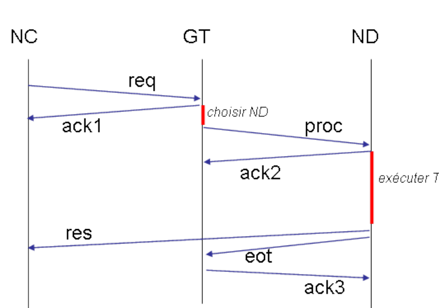

Figure 2. *Les phases d'exécution d'une transaction*

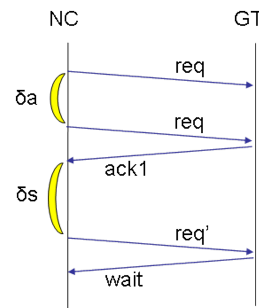

Figure 3. *Comportement du NC en fonction des temps d'attente*

3.2. *Algorithme de routage*

Dans cette section, nous décrivons l'algorithme de routage que chaque GT applique lorsqu'il reçoit une transaction. Notre stratégie est basée sur le coût et utilise la synchronisation à la demande. Il tient compte du coût de rafraîchissement d'un nœud dans le calcul du coût global d'un nœud. Comme mentionné dans (Gançarski *et al.,* 2007), la complexité de notre algorithme est linéaire en fonction du nombre de transactions actives et passe donc à l'échelle.

Dans un premier temps, l'algorithme évalue, pour chaque nœud N_j :

– la charge de N_j. Cette composante du coût du nœud est obtenue en évaluant le temps d'exécution restant de toutes les transactions actives sur N_j ;

– le coût de rafraîchissement de N_j afin que celui-ci soit suffisamment frais par rapport à l'obsolescence tolérée par la transaction T (on rappelle que si T fait des mises à jours, cette tolérance est forcément nulle). A cet effet, l'algorithme calcule un graphe de rafraîchissement GR pour N_j. GR est le plus petit sous-graphe du GSG tel que l'exécution des nœuds de GR dans l'ordre rend N_j suffisament frais pour l'exécution de T. En d'autres termes, après application de GR sur N_j, l'obsolescence de ce dernier par rapport à chaque relation lue par T est inférieure à l'obsolescence tolérée par T pour la relation correspondante. Le coût de rafraîchissement est donc le temps total estimé pour exécuter toutes les transactions dans GR ;

– le coût d'exécution de T elle-même ;

– le coût total de N_j pour T, obtenu en faisant la somme des trois coûts précédents.

Une fois que le coût est estimé pour tous les nœuds, l'algorithme choisit le nœud N de moindre coût et lui envoie le GR correspondant suivi de T. Puis il met à jour le répertoire partagé en conséquence : toutes les transactions envoyées à N sont retirées de la liste des transactions en attente d'envoi à N.

GR étant un sous-graphe du GSG, l'ordre d'exécution des transactions sur chaque nœud est compatible avec l'ordre global, ce qui garantit la cohérence globale puisque toutes les transctions sont exécutées sur tous les nœuds dans des ordres compatibles (voir la section 2.4).

3.3. *Accès concurrents au répertoire partagé*

Contrairement à la version centralisée de (Gançarski *et al.,* 2007), où un seul routeur accède aux métadonnées, nous devons ici prendre en compte les possibles accès concurrents aux métadonnées par plusieurs routeurs. Pour résoudre ce problème, nous avons décidé d'utiliser un verrouillage en deux phases (les verrous sur les métadonnées sont tenus jusqu'à la fin du processus de routage), en nous basant sur deux observations : (1) le routage est un processus extrémement court par rapport à l'exécution des transactions et des graphes de rafraîchissement, ce qui fait que les verrous sont relâchés assez rapidement ; et (2) JuxMem offre un mécanisme de verrouillage, ce qui rend notre implémentation directe. Afin de valider notre choix, nous avons mené des expériences afin de mesurer la surcharge dûe à ces accès concurrents aux métadonnées. Les résultats sont présentés à la section 5.

4. Gestion de la dynamicité des nœuds

Nous décrivons l'approche utilisée pour gérer l'arrivée ou le départ d'un nœud durant l'exécution du système. Nous supposons que chaque nœud (NC, GT ou ND) rejoignant le système est capable de localiser un GT disponible. Le GT contacté, est alors responsable d'inclure le nouveau nœud en mettant à jour l'anneau (arrivée d'un GT) ou le répertoire partagé (connexion d'un NC ou ND). Comme notre principal objectif est de préserver la cohérence quand un nœud quitte le système, nous restreignons notre étude sur les déconnexions des GT et ND (si un NC quitte le système, la cohérence du système n'est nullement menacée puisque le NC délègue l'exécution des transactions aux GT). Par ailleurs, nous distinguons deux situations : les déconnexions prévues et celles imprévues.

4.1. *Déconnexion prévue*

Une déconnexion prévue survient si un nœud décide volontairement de quitter le système.

– Déconnexion prévue d'un GT. Quand un GT décide de quitter le système, il informe ses prédécesseurs (resp. successeurs) qui vont mettre à jour l'anneau logique en l'enlevant sur leur liste de successeurs ou de prédécesseurs. Durant la déconnexion, le GT ignore simplement les messages entrants.

– Déconnexion prévue d'un ND. Si un ND veut se déconnecter, il envoie un message appelé *Disconnection request* au dernier GT qui lui avait envoyé une transaction et qui est encore disponible. Ce GT, s'il a reçu ce message, enlève le ND de la liste des ND disponibles pour éviter qu'un autre GT route une transaction vers ce nœud en cours de déconnexion. Par la suite, le GT envoie au ND un message appelé *Disconnection accepted*, ce qui permet de considérer ce ND comme étant déconnecté jusqu'à ce qu'il notifie son retour dans le système.

4.2. *Déconnexion imprévue*

Dans l'optique de gérer les déconnexions imprévues, nous détaillons chacune des trois phases définies dans la section 3. Nous supposons qu'il y a toujours au moins un nœud disponible (GT ou ND) sur lequel on peut s'appuyer à chaque fois que l'on détecte une panne de nœud. Dans la suite, nous utilisons les noms de messages définis dans la figure 2.

4.2.1. *Gestion des pannes faite par le NC*

Un NC envoie un message *req* à un GT et initialise ensuite un temps d'attente δ_a (cf. figure 3). Quand δ_a expire, le NC conclut que le message *req* ou l'acquittement *ack1* est perdu à cause d'une panne de communication ou du GT contacté. Alors, NC retransmet le message *req* avec le même identifiant global. Pour accroître les chances de réussite de la retransmission, le NC incrémente le nombre de GT cibles. Plus précisément, le NC ajoute un GT de plus sur la liste des destinataires à chaque fois qu'il retransmet *req*. Les GT candidats sont choisis parmi les GT connus par le NC en utilisant l'algorithme du tourniquet. Remarquons que la cohérence ne peut être compromise puisque la transaction n'est transmise à aucun ND pour exécution. Même si plusieurs GT reçoivent la même transaction, cette dernière n'est transmise qu'à un seul ND grâce à l'utilisation de l'identifiant global et de l'accès exclusif au répertoire partagé.

Quand un NC reçoit *ack1* de la part d'un GT, il arrête toute tentative de retransmission de *req* et initialise un autre temps d'attente δ_s. Si le NC n'a pas de résultats jusqu'à l'expiration de δ_s, il conclut que la transaction T est toujours en exécution ou ses résultats sont perdus ou T a échoué. Ce faisant NC envoie *req'* aux GT précédemment contactés (*req'* ressemble à *req*, mais signifie aussi que le NC avait déjà

reçu *ack1*). Afin de réduire les retransmissions inutiles, les valeurs des temps d'attente sont basées sur la latence du réseau et le temps moyen d'exécution des transactions. D'où, $\delta_a \geq\ 2 * \lambda_N$ et $\delta_s \geq 2 * \delta_a + \lambda_D + Avg(T)$ avec λ_N la latence du réseau, λ_D est le temps pour lire/écrire sur le répertoire partagé et *Avg (T)* est le temps moyen d'exécution de T.

4.2.2. *Gestion des pannes faite par le GT*

A la réception d'un message *req*, le GT envoie l'acquittement *ack1* au NC. Ensuite, le GT vérifie si la transaction T est terminée ou est encore en exécution. Si T n'est pas mentionnée dans le répertoire partagé, alors le GT route T vers un ND (ceci évite d'exécuter deux fois la même transaction).

A la réception d'un message *req'* de la part d'un NC, trois cas peuvent être identifiés en fonction de l'état de la transaction T :

1. si T est déjà exécutée sur le nœud *ND*$_i$, alors le GT retransmet T sur *ND*$_i$. Ce cas survient, si le résultat de l'exécution de la transaction n'a pu être envoyé à cause d'une panne de communication ;

2. si T est en progression (déjà routée mais non encore exécutée), alors le GT répond par un message *wait* au NC. Ce cas se présente quand T dure plus longtemps que prévu à cause d'une panne d'un nœud ND ;

3. si T n'est pas mentionnée dans le répertoire partagé, alors le GT achemine T vers un ND. Ce cas est identifié si le GT tombe en panne avant de choisir un ND (donc avant d'écrire sur le répertoire partagé).

A chaque évaluation de l'algorithme de routage, le GT garde la liste des ND candidats triés suivant l'ordre croissant du coût. Par la suite, le GT envoie le message *proc* au premier candidat *ND*$_i$, et initialise un temps d'attente δ_a. Une fois que le message *ack2* est reçu, le GT inscrit dans le répertoire partagé que T est en cours d'exécution. A partir de ce moment, il initialise un second temps d'attente δ_r.

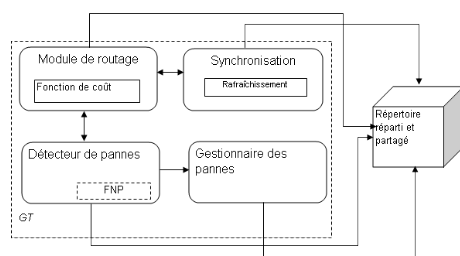

Figure 4. *Architecture du GT*

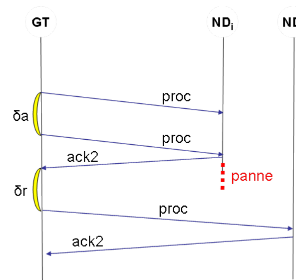

Figure 5. *Comportement du GT en fonction des temps d'attente*

Quand δ_a et δ_r expirent, le GT conclut qu'une panne de communication ou du ND est survenue. Il envoie alors le message *proc* au prochain candidat ND_j sur la liste

(cf. figure 5). En parallèle, il invoque le module de détection de pannes (cf. figure 4) qui vérifie si ND_i est disponible ou non. Pour cela, il contacte ses prédécesseurs et successeurs et chacun d'entre eux essaie d'entrer en contact avec ND_i en lui envoyant un message. Les résultats de ces échanges sont envoyés au nœud initial. Si tous les résultats sont négatifs, il conclut que ND_i est en panne et l'ajoute dans la file des nœuds en panne, appelée *FNP* et stockée dans le répertoire partagé. Par contre, si au moins un des résultats est positif, le GT suppose qu'il y a une panne de communication temporaire entre lui et ND_i. Il peut donc considérer ND_i comme un candidat potentiel lors des prochains routages. Le gestionnaire des pannes, appelé module de reprise sur panne (cf. figure 4), est chargé de vérifier la reprise de chaque nœud en panne et de l'enlever de la file *FNP* comme décrit dans (Chandra *et al.,* 1996).

Finalement, quand un GT reçoit un message *eot* de la part d'un ND_i, il met à jour le répertoire partagé en mentionnant que T est exécutée sur ND_i, et il envoie un acquittement *ack3* à ND_i.

La valeur de δ_r est proportionnelle à la latence du réseau et au temps moyen d'exécution de T. Nous définissons $\delta_r \geq \delta_a + Avg(T)$. Pour éviter les messages inutiles, nous posons $\delta_r < \delta_s$ de tel sorte qu'un NC ne peut retransmettre une requête avant qu'un GT n'ait la possibilité de détecter une potentielle panne du ND.

4.2.3. *Gestion des pannes faite par le ND*

A la réception d'un message *proc*, le ND répond au GT par l'envoi de l'acquittement *ack2*. Le ND vérifie d'abord si T n'est pas déjà exécutée (en consultant son journal). Dans la négative, le ND exécute T. Si T a fini de s'exécuter, le ND répond par un message *res* (contenant le résultat de l'exécution de T) au client qui a initialisé T. Si toutefois, T a déjà été exécutée, le ND envoie un nouveau message *res* au NC. Le ND répond également par un message *eot* au GT, et initialise un temps d'attente δ_a. Quand δ_a expire, le ND conclut qu'il y a une panne de communication ou du GT. Alors, il ajoute un message *eot* dans un buffer pour l'envoyer lors de la prochaine notification en utilisant la technique de *piggybacking*. Ceci a pour objectif de réduire le nombre de messages envoyés au GT par rapport à une stratégie de tentatives périodiques.

En outre, nous remarquons que si le nœud suspecté a déjà traité la transaction avant de tomber en panne, alors l'exécution de T sur un autre ND ne compromet pas la cohérence, puisque l'exécution de toutes les transactions est faite de manière similaire sur tous les nœuds (*i.e.* avec un ordre de précédence global).

4.3. *Analyse de la surcharge des déconnexions*

Comme décrit précédemment, notre protocole de routage se termine malgré la présence de pannes. Néanmoins, le délai pour exécuter totalement une transaction augmente proportionnellement avec le nombre de pannes. Plus le nombre de retransmissions nécessaires pour exécuter une transaction est grand, plus le temps d'exé-

cution s'élève. Pour démontrer l'efficacité de notre approche, nous montrons que le nombre d'essais requis pour exécuter une transaction est souvent faible. Dans cette perspective, nous notons $avg(T)$, le temps moyen d'exécution d'une transaction, λ_N, la latence moyenne du réseau, λ_D, le temps d'accès moyen au répertoire partagé et $\bar{S}$, la taille de la séquence de rafraîchissement (cf. section 3.2). En absence de toute panne, le temps nécessaire pour exécuter T est :

$$time(T) = 3 * \lambda_N + (\bar{S} + 1) * avg(T) + \lambda_D$$

Si un GT et/ou un ND participant à l'exécution de T tombe en panne, alors dans le pire des cas, la transaction va être exécutée après k tentatives initiées par le NC. Soit k le nombre de tentatives requises lors de l'exécution de T, alors le temps total d'exécution de T est :

$$time_k(T) = k * time(T) + (k - 1) * \delta_s$$

Supposons que p est la probabilité qu'une transaction tombe en panne (*i.e.* NC n'a reçu aucun résultat) et X une variable aléatoire représentant le nombre de tentatives. $P(X = i) = (1 - p) * (p)^{i-1}$ est la probabilité que les *(i-1)* premières tentatives ont échoué et que le *i*ème a réussie. Le nombre de tentatives exécutées est obtenu avec la formule suivante :

$$E(X) = \sum_{i=0}^{n} i * P(X = i) = (1 - p) * (\sum_{i=0}^{n} i * p^{i-1})$$

Une majoration possible de $E(X)$ est $E(X) < (1 - p) * (\sum_{i=0}^{\infty} i * p^{i-1})$. En remplaçant $\sum_{i=0}^{\infty} i * p^{i-1}$ par sa limite $\frac{1}{(1-p)^2}$, on obtient :

$$E(X) < \frac{1}{(1 - p)}$$

Par conséquent, nous pouvons approximer le nombre de tentatives : $k \approx \lceil \frac{1}{1-p} \rceil$. Par exemple, k=2 pour une probabilité de panne inférieure à 50 %. A partir de ces résultats, nous concluons que la surcharge du protocole de routage est raisonnable dans notre contexte.

5. Evaluation

Dans cette section, nous évaluons les performances de notre solution. Il a été démontré que l'algorithme proposé dans Leg@net (Gançarski *et al.,* 2007) donne de meilleures performances que l'algorithme du tourniquet ou que la stratégie choisissant le nœud le moins chargé. Puisque notre approche est basée sur le coût des transactions, nous comparons la version distribuée avec la version centralisée de Leg@net.

D'abord, nous vérifions que le routeur distribué n'est pas une source de congestion, *i.e.* il route rapidement toute transaction. Par la suite, nous évaluons si le routeur présente des opportunités aux applications, *i.e.* s'il améliore le temps de réponse.

Nous avons effectué toutes les expériences sur un cluster de 20 nœuds sur lesquels s'exécutent soit le routeur distribué, soit un SGBD pour stocker des données. Nous avons utilisé les ordinateurs personnels des membres du laboratoire pour stocker les applications. Le routeur est implémenté avec le langage C et s'appuie sur les services de Juxmem, qui est conçu au-dessus de la plate-forme JXTA de Sun. Nous remarquons que notre système est faiblement dépendant de Juxmem, puisque Juxmem est utilisé comme une API (une librairie). Alors, nous pouvons utiliser tout logiciel qui fournit une API d'accès à une mémoire virtuelle partagée. Notre routeur fonctionne comme un intergiciel ; il fournit une interface de traitement de transactions pour les applications.

5.1. *Surcharge de l'accès au répertoire partagé*

Les premières expérimentations s'intéressent au routage proprement dit. Elles mesurent la surcharge engendrée par l'utilisation d'un répertoire partagé pour stocker les métadonnées. La charge applicative est générée par un nombre croissant d'applications, chacune d'entre elles envoie une transaction par seconde à un routeur. Nous mesurons le débit (en transaction/seconde) qu'un routeur peut assurer. La figure 6 montre que chacun des routeurs peut traiter plus de 40 transactions/seconde.

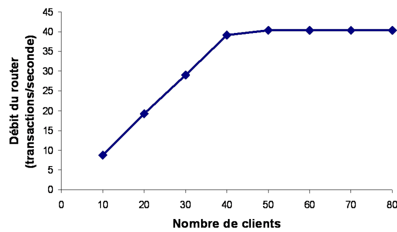

Figure 6. *Débit de l'intergiciel*

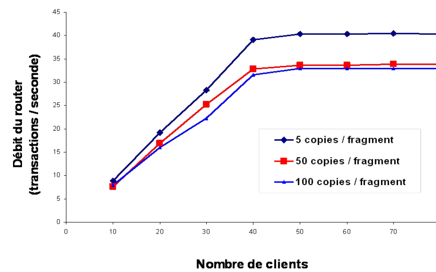

Figure 7. *Surcharge du répertoire*

Une part importante du processus du routage est l'accès au répertoire partagé. Ainsi, pour évaluer le coût de cet accès et s'assurer que notre solution passe à l'échelle, nous augmentons la taille du répertoire en ajoutant de nouvelles répliques, puisque plus de répliques impliquent plus de métadonnées. Nous reportons dans la figure 7, le débit traité dans trois situations : petite, moyenne et grande taille du répertoire (respectivement 5, 50, 100 répliques). Nous mesurons une baisse des performances inférieure à 20 % pour une grande quantité de métadonnées (100 répliques). Pour un degré de réplication moyen (par exemple 50 répliques), la baisse est uniquement de 5 %. Ceci montre que le répertoire pénalise peu les performances.

Par la suite, nous étudions l'impact de l'accès concurrent de plusieurs routeurs au répertoire partagé. La charge applicative est générée de la même manière que les premières expérimentations, cependant les transactions sont envoyées à 2 routeurs de telle sorte que la moitié de la charge va sur chacun des nœuds. Les résultats de la figure 8 sont obtenus dans le pire des cas (*i.e.* toutes les transactions accèdent aux mêmes données, conduisant les routeurs à faire la même chose). Ces résultats montrent un débit maximal de 20 transactions/seconde, c'est-à-dire la moitié d'un seul routeur. En effet, l'attente d'un verrou dégrade les performances. Néanmoins, dans le pire des cas où chaque accès au répertoire est retardé par un autre accès concurrent sur la même donnée, le routeur est encore capable de fournir de bons résultats. Dans notre contexte, les situations de concurrence sont peu fréquentes puisque les métadonnées sont fragmentées. C'est pourquoi, nos prochaines expériences visent à évaluer la baisse de performances induite par l'accès concurrent au répertoire en fonction du degré de concurrence. En d'autres termes, nous faisons varier le degré de concurrence entre 0 % et 100 % et mesurons la variation de performances. Dans le meilleur des cas (degré de concurrence égal à 0), le débit global théorique est égal au débit maximal d'un routeur (40 transactions/seconde) multiplié par le nombre de routeurs comme indiqué dans la section suivante.

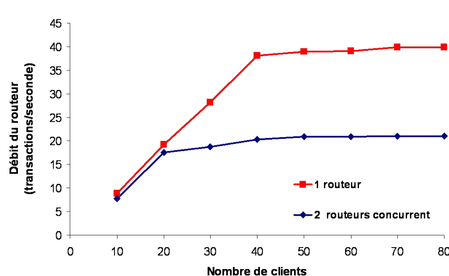

Figure 8. *Accès concurrent*

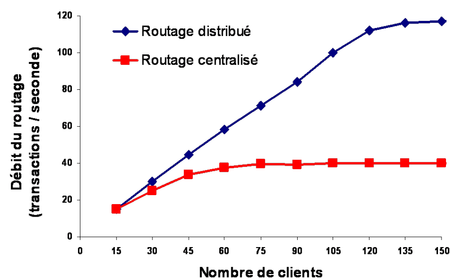

Figure 9. *Algorithme distribué vs centralisé*

5.2. *Performance globale du routage*

Cette expérience cible la performance globale du routage distribué. Nous mesurons les améliorations du routage distribué en ce qui concerne le débit, comparées à la version centralisée de (Gançarski *et al.,* 2007). La charge est générée par N applications classées dans 3 catégories en proportion égale. Chaque type d'application accède à une partie distincte de la base de données et donc est connecté à un routeur spécifique. En d'autres termes, il n y a aucune concurrence entre routeurs au niveau de l'accès au répertoire (le degré de concurrence vaut 0). Nous mesurons le débit du traitement quand N varie de 15 à 150 applications. Sur la figure 9, nous comparons ces résultats avec le cas où un seul routeur reçoit cette même charge. Plus N s'accroît,

plus la différence entre le routage centralisé et distribué devient importante. Pour une forte charge de 150 applications, le gain en distribué atteint un facteur de 3. La raison principale est que le routage centralisé atteint ses limites très rapidement. Nous observons un gain égal au nombre de routeurs, ce qui démontre une montée en charge linéaire. Par extrapolation, moins de 50 routeurs seraient capables de supporter une charge transactionnelle de 1 500 transactions/seconde (comme celle de eBay) pour un taux de conflit inférieur à 10 %.

5.3. *Impact du relâchement de la fraîcheur*

Dans cette section, nous étudions et mesurons l'influence du relâchement de la fraîcheur sur les performances en termes de temps de réponse et d'équilibrage de charge. Nous avons choisi une taille intermédiaire : 40 applications (20 pour des mises à jour et 20 pour des lectures) et 20 ND. Nous faisons varier l'obsolescence tolérée des transactions de lecture. La figure 10 montre le temps de réponse des transactions en fonction de l'obsolescence tolérée (exprimée en nombre de mises à jour manquantes). Les résultats révèlent qu'augmenter l'obsolescence tolérée diminue considérablement le temps de réponse. Cela est principalement dû au fait que relâcher la fraîcheur donne plus de souplesse pour retarder la synchronisation, et donc l'exécution des transactions se fait de manière plus rapide. Nous notons que si le degré d'obsolescence dépasse 15, le temps de réponse ne s'accroît plus. La raison est que le temps de réponse minimal d'exécution d'une transaction est atteint. La valeur précise 15 découle du paramétrage de la taille de notre système, *i.e.* un nombre d'applications et de ND différents de celui que nous avons considéré engendrerait une valeur différente de 15.

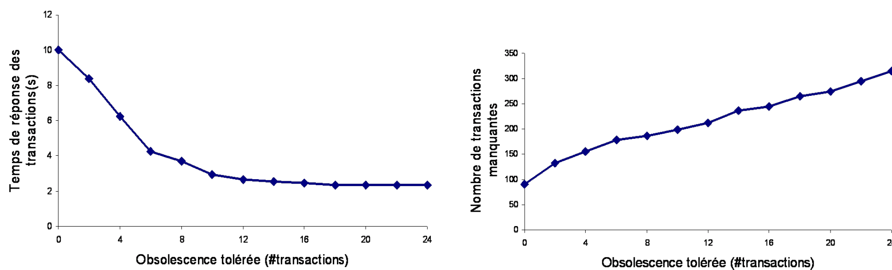

Figure 10. *Temps de réponse vs. obsolescence tolérée*

Figure 11. *Nombre de mises à jour manquantes vs. obsolescence tolérée*

Bien entendu, le relâchement de la fraîcheur s'accompagne d'une perte de cohérence mutuelle des répliques. La figure 11 montre l'obsolescence des données à la fin de l'expérience, en fonction de l'obsolescence tolérée des transactions de lecture. Pour maintenir les nœuds à un niveau de fraîcheur raisonnable, nous envisageons d'utiliser les stratégies de rafraîchissement présentées dans

5.4. *Passage à l'échelle de notre solution*

Dans l'optique de prendre en compte une échelle plus large, nous avons considéré une réplication de la base de données et du répertoire partagé sur une grille informatique fédérant plusieurs grappes de machines. Dans cet article, notre principal objectif est de montrer les bénéfices obtenus en distribuant le protocole du routage. Ce faisant, une simple réplication de l'intergiciel sur quelques nœuds (au moins un routeur par grappe de la grille) est suffisante. En outre, les expériences menées sur Grid5000 (Project, n.d.) pour valider Juxmem, montrent qu'une écriture nécessite plus de 90 ms. Ainsi, dérouler notre approche dans un environnement similaire, engendrerait un temps de routage d'environ 100 ms, d'où un débit transactionnel inférieur à 10 transactions/seconde. Cependant, si chaque routeur accède uniquement aux portions des métadonnées stockées dans la grappe où il se trouve, le débit de traitement reste acceptable (plus de 40 transactions/seconde), même si les données sont répliquées sur des sites distants. Dans ce dernier cas, le temps de réponse s'accroît légèrement en fonction de la latence entre les grappes.

6. Validation par simulation

Pour évaluer notre protocole de gestion des pannes, sur un système à large échelle avec des centaines de nœuds, nous l'avons implémenté en utilisant le simulateur PeerSim (PeerSim, n.d.). Puisque nous cherchons à mesurer l'impact des pannes de nœuds, nous avons utilisé la simulation par cycle de PeerSim, qui fournit le passage à l'échelle et cache les détails du protocole de communication. Le comportement de chaque nœud (NC, GT et ND) est implémenté avec Java et est inclus dans la plate-forme de PeerSim sous forme de protocoles. Nous avons également implémenté le répertoire partagé utilisé par les GT pour router les requêtes.

6.1. *Détection d'une panne de ND*

Pour simuler la dynamicité des nœuds de données (ND), nous avons fait varier le nombre de ND en panne de 0 à 10 %, ce qui est assez représentatif pour évaluer les gains de notre approche. Dans notre solution, la détection des pannes de ND est intégrée dans l'algorithme de routage (appelé DTR2) du GT. Notre objectif est de comparer DTR2 avec l'algorithme de routage (DTR) présenté dans (Sarr *et al.,* 2008). DTR ne détecte pas les pannes et ne les gère pas : chaque transaction envoyée à un ND en panne ne se termine pas. Nous comparons les performances de DTR2 par rapport à celles de DTR, en mesurant le nombre de transactions exécutées en un seul passage. La charge de travail est soumise par 40 applications. Chaque application envoie une transaction par cycle. Une simulation dure 60 cycles. Nous utilisons 10 GT qui sont disponibles durant toute la simulation. Nous faisons varier le nombre de ND de 50 à 200 et nous reportons sur la figure 12 le nombre de transactions exécutées. Nous observons que les performances de DTR2 dépassent par un facteur de 140 %

celles de DTR, quand le nombre de ND est supérieur à 100. Deux raisons peuvent expliquer cette amélioration. Premièrement, avec DTR2, un nœud qui a été détecté comme indisponible ne recevra plus de transaction à exécuter. Deuxièmement, si un ND tombe en panne au moment où il exécutait une transaction, le GT va choisir un autre ND pour finir son exécution. Nous remarquons que les performances de DTR et DTR2 ne s'améliorent pas si le nombre de répliques dépasse 180. En effet, ajouter plus de ND ne fait pas évoluer les performances car il y a suffisamment de ND pour traiter la charge entrante. (Le Pape *et al.,* 2006).

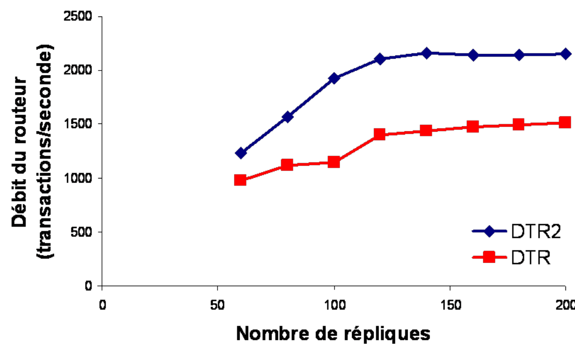

Figure 12. *Débit vs. nombre de répliques*

6.2. *Détection d'une panne de GT*

Dans cette section, nous comparons DTR2 avec l'approche proposée dans SWIM (Das *et al.,* 2002), en mesurant le coût de communication (*i.e.* le nombre total de messages nécessaires pour détecter les pannes). Dans SWIM, la détection est faite en envoyant périodiquement des messages de "ping-pong" à un ensemble aléatoire de nœuds (nous avons choisi 4 dans notre cas). Dans DTR2, chaque GT collabore avec les autres GT (4 dans cette expérience) pour détecter la panne. Nous commençons la simulation avec 18 GT disponibles et 2 GT en panne. Ensuite on fait varier le nombre de GT en panne de 2 à 18. Nous reportons sur la figure 13 le nombre de pannes détectées.

La figure 13 montre qu'avec un nombre de pannes élevé, DTR2 génère de meilleures performances que SWIM en ce qui concerne le nombre de pannes détectées. La principale raison est que chaque GT en panne est détecté par au moins un de ses successeurs, donc il est exclu de l'anneau logique. Au contraire, avec SWIM, un GT peut tomber en panne sans pour autant être détecté puisqu'il peut rester longtemps sans être choisi aléatoirement par les autres GT.

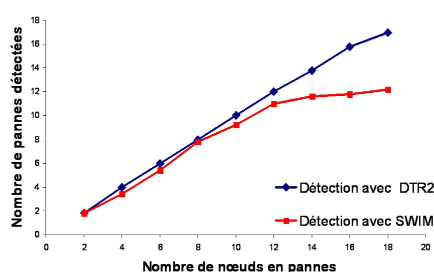

Figure 13. *Pannes détectées vs. pannes survenues*

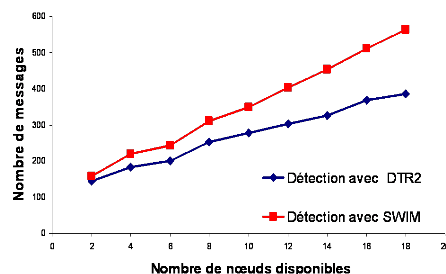

Figure 14. *Nombre de messages vs. pannes survenues*

En outre, nous comparons le coût de communication engendré par la détection des pannes entre DTR2 et SWIM. Pour cela, nous calculons le nombre total de messages envoyés par les nœuds pour détecter l'occurrence des pannes. Les résultats présentés sur la figure 14 montrent que le nombre total de messages pour détecter les pannes augmente si le nombre de nœuds disponibles est important. En fait, avec 20 GT le nombre de messages est supérieur à 400 durant une simulation, puisqu'un GT contacte ses successeurs périodiquement. En plus, nous soulignons que DTR2 requiert toujours moins de messages que SWIM lors de la détection. En effet l'anneau logique est restructuré dès qu'une panne est détectée, évitant ainsi de contacter un nœud en panne plus d'une fois. Avec SWIM, un nœud en panne peut être contacté par les autres nœuds qui ne sont pas encore au courant de la panne, puisque la notification de panne n'est pas immédiate. Nous planifions d'autre simulations pour étudier les cas de pannes simultanées, *i.e.* quand le GT et le ND impliqués dans le traitement d'une transaction tombent en panne simultanément. Bien que ce dernier cas ne compromette pas la cohérence des données, il augmenterait néanmoins le temps de réponse.

7. Travaux connexes

Notre travail s'inscrit dans le contexte du traitement de transactions sur des bases de données réparties et répliquées. Nous classons les solutions existantes en fonction du type de réplication qu'elles emploient. Le premier paramètre de la réplication est le nombre de répliques dites primaires (*i.e.* , pouvant être connectées directement avec l'application) : on distingue les solutions monomaître et celles qui sont multimaître. Le deuxième paramètre concerne le couplage entre le traitement initial d'une transaction et sa propagation sur les répliques. Le couplage fort est appelé propagation synchrone car la propagation fait partie intégrante de la transaction initiale. Au contraire, le couplage faible est appelé propagation asynchrone car la transaction initiale s'exécute localement indépendamment de la propagation. Cependant, les solutions asynchrones étudiées empêchent les conflits afin de garantir la cohérence des données. Les fonc-

tionnalités des travaux connexes sont proches de celles de notre solution. Nous les recensons afin de mieux positionner notre approche.

Autonomie : préserver les bases de données existantes, sans intrusion, afin d'intégrer des bases opérationnelles.

Fraicheur des données : Lorsque la lecture de données non fraîches est possible, le résultat d'une requête de lecture a toujours une fraîcheur au moins égale à celle exigée par l'application. Ceci permet d'accéder à des données obsolètes, dans une limite fixée, et présente l'avantage de pouvoir décaler ultérieurement la propagation sur les répliques afin d'économiser les ressources des bases de données.

Equilibrage de charge : répartir les traitements sur plusieurs répliques afin d'améliorer le temps de réponse moyen des transactions ou le débit.

Disponibilité : Gérer des cas spécifiques de pannes provoquées par la volatilité des sources de données qui rejoignent ou quittent le système.

Echelle : Prendre en compte de nombreux SGBD répartis sur un réseau dont nous distinguons deux tailles, moyenne (M) et large (L) : (i) à l'échelle d'un réseau local, les machines forment une grappe, cela assure une latence homogène pour toute communication entre machines ; (ii) à l'échelle mondiale, les machines communiquent par Internet. La latence peut varier fortement dans le temps et selon la localisation des machines.

Notre travail apporte des éléments de solutions pour assurer toutes les fonctionnalités énumérées ci-avant. Par la suite, nous résumons les solutions connexes avant de les comparer.

Middle-R (Patino-Martinez *et al.,* 2005) est un intergiciel de réplication synchrone garantissant une cohérence forte des données. Il est tolérant aux pannes et optimisé pour limiter les abandons en cas de forte charge transactionnelle. Il améliore des travaux précédents sur la réplication active tels que (Guerraoui *et al.,* 1997) et (Schneider, 1993). Cependant, Middle-R synchronise les répliques au moyen de primitives de communication de groupe qui s'avèrent coûteuses, notamment pour les réseaux large-échelle.

C-JDBC (Cecchet *et al.,* 2005) est un intergiciel de réplication gérant un cluster de SGBD. Etant conçu comme un pilote JDBC, il permet à l'utilisateur de traiter des transactions de manière transparente. La stratégie de routage est simple et efficace : chaque requête de lecture seule est envoyée à un SGBD différent à tour de rôle, chaque transaction est diffusée à tous les SGBD. La cohérence des répliques n'est pas garantie car la première réplique ayant fini de traiter une transaction est désignée pour servir de référence sans tenir compte des autres répliques. Ainsi, cette solution est restreinte à un environnement stable.

FAS (Rohm *et al.,* 2002) est un intergiciel de réplication monomaître et asynchrone. Il prend en compte la fraîcheur des SGBD afin de garantir que les exigences de fraîcheur d'une requête soient satisfaites. Il transmet les transactions sur le SGBD

maître, et les requêtes de lecture sur le nœud le moins chargé. La synchronisation des répliques est différée périodiquement. FAS étant monomaître, cela ne permet pas de supporter une charge transactionnelle croissante. De plus si aucun SGBD n'est suffisamment frais pour traiter une requête, celle-ci est mise en attente, ce qui peut provoquer la surcharge d'un SGBD au moment où il devient disponible pour traiter les requêtes en attente. Dans ce cas précis, la synchronisation anticipée des répliques aurait été bénéfique.

UMS/KTS (Akbarinia *et al.,* 2007) aborde la gestion de versions des données dans des systèmes pair-à-pair structurés reposant sur une table de hachage distribuée. La cohérence mutuelle est garantie à l'aide d'un service d'estampillage, tolérant aux pannes, qui permet de retrouver efficacement la version courante d'une réplique. Cependant la disponibilité des nœuds stockant les données n'est pas étudiée. Une incohérence peut se produire si un nœud stockant la dernière version d'une donnée quitte le système avant d'avoir propagé sa donnée.

RepDB (Pacitti *et al.,* 2005) est une solution entièrement décentralisée pour gérer des SGBD répliqués. Elle suppose des communications fiables pour préserver l'ordre des messages. La cohérence forte des données est garantie par un ordonnancement des transactions entrantes. Une transaction peut valider dès qu'elle est planifiée dans le même ordre sur toutes les répliques, c'est-à-dire après un délai maximum égal à la durée d'une communication entre deux répliques. Par conséquent, cette solution est restreinte à l'échelle d'un cluster muni d'un réseau fiable où les temps de communication sont bornés.

Sprint (Camargos *et al.,* 2007) est un intergiciel offrant de hautes performances et une haute disponibilité pour un SGBD en mémoire et répliqué. La cohérence est maintenue en ordonnant les transactions ce qui évite les interblocages. Toutefois, Sprint utilise un protocole de vote et nécessite la mise en œuvre d'un protocole de terminaison sur chaque participant. Par conséquent, l'autonomie des SGBD est compromise. De plus, la panne d'un nœud peut provoquer l'abandon d'une transaction qui devra être renouvelée.

Leg@net (Gançarski *et al.,* 2007) est une solution de réplication muti-maîtres pour le routage de transactions dans un cluster de bases de données. Leg@net relâche autant que possible la fraîcheur des données, dans les limites acceptées par les requêtes. Cela réduit le surcoût de synchronisation des répliques et permet ainsi d'allouer davantage de ressources au traitement des transactions. Leg@net cible les applications transactionnelles dont l'autonomie doit être préservée. Toutefois, cette solution ne passe pas à une échelle plus grande car l'intergiciel est centralisé.

Finalement, nous synthétisons dans la figure 1 les propriétés qui caractérisent les travaux connexes à notre approche. Bien que cette liste ne soit pas exhaustive, nous constatons que pour chaque solution connexe, il y a (à notre connaissance) au moins deux propriétés pour lesquelles la solution présente des limitations.

Caractéristiques	Middle-R	C-JDBC	FAS	UMS KTS	RepDB	Sprint	Leg@net	DTR2
Modèle de réplication	Multi+ Sync	Multi+ Async	Mono+ ASync	Multi+ Async	Multi+ Async	Multi+ Sync	Multi+ Async	Multi+ Async
Autonomie	O	O	N	O	N	N	O	O
Fraîcheur	N	N	O	N	N	N	O	O
Equilibrage des charges	O	O	Lecture seule	N	N	N	O	O
Gestion des pannes	O	O	N	Partielle	N	O	N	O
Échelle	M	M	M	L	M	M	M	L

Tableau 1. *Caractéristiques des travaux connexes vs notre approche*

8. Conclusion

Cet article présente la conception et l'implantation d'un modèle de routage réparti de transactions. Notre solution est conçue pour les systèmes à large échelle et à fort taux de mise à jour tels que les applications Web 2.0 et inclut contrôle de fraîcheur et gestion des pannes.

Nous utilisons JuxMem, un système de gestion de mémoire partagée conçu pour les grilles, afin de mettre en œuvre un répertoire partagé pour les métadonnées nécessaires au routage et au contrôle de la fraîcheur mais aussi pour la gestion des pannes. Nous proposons un protocole permettant de gérer toutes les situations lorsqu'un nœud quitte le système pendant le traitement d'une transaction. Pour cela, nous adaptons des approches existantes de détection des pannes afin de les rendre opérationnelles pour chaque type de nœud (gestionnaire de transaction et nœud de données) de notre système.

Les évaluations expérimentales de notre système montrent plusieurs avantages de notre approche : *(i)* la surcharge due aux accès au répertoire partagé est faible, *(ii)* l'utilisation de ce répertoire partagé permet l'implantation de plusieurs instances du routeur. Lorsque la charge est élevée, ceci augmente significativement le débit transactionnel global par rapport à la version centralisée du routeur, *(iii)* les performances obtenues en termes de temps de réponse et d'équilibrage par contrôle de fraîcheur sont tout à fait satisfaisantes.

De plus, les évaluations par simulation montrent que notre protocole de gestion des pannes est efficace, puisqu'il augmente significativement le taux de transactions exécutées et diminue les coût de communication par rapport aux approches existantes.

Les travaux en cours visent à trouver le nombre optimal d'instances du routeur à mettre en œuvre en fonction des caractéristiques (taux de mise à jour, taux de conflit, nombre de clients...) de la charge transactionnelle. Nous voulons aussi prendre mieux en compte l'hétérogénéité du système sous-jacent (différence de latence entre les liens intra-grappe et liens inter-grappe), afin d'optimiser non seulement le choix du meilleur

nœud mais aussi la détection des pannes. Ensuite, nous envisageons d'améliorer le comportement des nœuds gestionnaires de transaction en les rendant auto-adaptatifs, et de mener des expérimentations en cas de dynamicité très forte des nœuds.

9. Bibliographie

Aguilera M., Chen W., Toueg S., « Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks », *Theoretical Computer Science*, vol. 220, n° 1, p. 3-30, 1999.

Akal F., Türker C., Schek H., Breitbart Y., Grabs T., Veen L., « Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees », *Int. Conf. on Very Large Data-Base (VLDB)*, p. 565-576, 2005.

Akbarinia R., Pacitti E., Valduriez P., « Data Currency in Replicated DHTs », *Int. Conf. on Management of Data (SIGMOD)*, p. 211-222, 2007.

Antoniu G., Bougé L., Jan M., « JuxMem : An Adaptive Supportive Platform for Data Sharing on the Grid », *Scalable Computing : Practice and Experience*, vol. 6, n° 3, p. 45-55, 2005.

Antoniu G., Deverge J., Monnet S., « How to Bring Together Fault Tolerance and Data Consistency to Enable Grid Data Sharing », *Concurrency and Computation : Practice and Experience*, vol. 18, n° 13, p. 1705-1723, 2006.

Bernstein P. A., Hadzilacos V., Goodman N., *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

Camargos L., Pedone F., Wieloch M., « Sprint : A Middleware for High-Performance Transaction Processing », *ACM European Conf. on Computer Systems (EuroSys)*, p. 385-398, 2007.

Cecchet E., Marguerite J., Zwaenepoel W., C-JDBC : Flexible Database Clustering Middleware, Technical report, ObjectWeb, Open Source Middleware, 2005.

Chandra T. D., Toueg S., « Unreliable Failure Detectors for Reliable Distributed Systems », *Journal of the ACM (JACM)*, vol. 43, n° 2, p. 225-267, 1996.

Das A., Gupta I., Motivala A., « SWIM : Scalable Weakly Consistent Infection-style Process Group Membership Protocol », *Int. Conf. on Dependable Systems and Networks (DSN)*, 2002.

Gançarski S., Naacke H., Pacitti E., Valduriez P., « The Leganet System : Freshness-aware Transaction Routing in a Database Cluster », *Journal of Information Systems*, vol. 32, n° 2, p. 320-343, 2007.

Guerraoui R., Schiper A., « Software-Based Replication for Fault Tolerance », *IEEE Computer*, vol. 30, n° 40, p. 68-74, 1997.

Koo R., Toueg S., « Checkpointing and Rollback-Recovery for Distributed Systems », *IEEE Transactions on Software Engineering*, vol. 13, n° 1, p. 23-31, 1987.

Larrea M., Arévalo S., Fernandez A., « Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems », *Int. Symp. on Distributed Computing (DISC)*, Springer, 1999.

Le Pape C., Gançarski S., « Replica Refresh Strategies in a Database Cluster », *High-Performance Data Management in Grid Environments (HPDGrid VECPAR Workshop), selected papers*, 2006.

Le Pape C., Gançarski S., Valduriez P., « Refresco : Improving Query Performance Through Freshness Control in a Database Cluster », *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2004.

MySpace, « http ://www.myspace.com », n.d.

Pacitti E., Coulon C., Valduriez P., Ozsu T., « Preventive Replication in a Database Cluster », *Distributed and Parallel Databases*, vol. 18, n° 3, p. 223-251, 2005.

Pacitti E., Minet P., Simon E., « Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases », *Int. Conf. on Very Large DataBases (VLDB)*, 1999.

Patino-Martinez M., Jimenez-Peres R., Kemme B., Alonso G., « MIDDLE-R, Consistent Database Replication at the Middleware Level », *ACM Transactions on Computer Systems*, vol. 28, n° 4, p. 375-423, 2005.

PeerSim, « http ://peersim.sourceforge.net/ », n.d.

Project G., « http ://www.grid5000.org », n.d.

Ramamritham K., Pu C., « A Formal Characterization of Epsilon Serializability », *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 7, n° 6, p. 997-1007, 1995.

Rohm U., Bohm K., Sheck H., Schuldt H., « FAS - a Freshness-Sensitive Coordination Middleware for OLAP Components », *Int. Conf. on Very Large DataBases (VLDB)*, 2002.

Sarr I., Naacke H., Gançarski S., « DTR : Distributed Transaction Routing in a Large Scale Network », *Int. Workshop on High-Performance Data Management in Grid Environments (HPDGrid)*, 2008.

Schneider F., *Replication Management Using the State-Machine Approach*, Distributed Systems (2nd Ed.), ACM Press, p. 169-197, 1993.

Second Life, « http ://secondlife.com », n.d.

Shoup R., « eBay Marketplace Architecture : Architectural Strategies, Patterns and Forces », *InfoQueue Conf. on Enterprise Software Development*, 2007.

TPC-Council, TPC Benchmark C, Rev 5.9, Technical report, Transaction Processing Performance Council, 2007.

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LA REVUE :
   RSTI - ISI – 15/2010. Bases de données avancées

2. AUTEURS :
   Idrissa Sarr — Hubert Naacke — Stéphane Gançarski

3. TITRE DE L'ARTICLE :
   Routage Décentralisé de Transactions avec Gestion des Pannes dans un Réseau à Large Echelle

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Transaction et gestion des pannes

5. DATE DE CETTE VERSION :
   27 janvier 2010

6. COORDONNÉES DES AUTEURS :

   – adresse postale :
   UPMC Paris Universitas
   Laboratoire d'Informatique de Paris 6
   104 Avenue du Président Kennedy
   75016 Paris, France
   Prénom.Nom@lip6.fr
   – téléphone : 01 44 27 87 61
   – télécopie : 01 44 27 70 00
   – e-mail : Hubert.Naacke@lip6.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com