Réplication Préventive dans une grappe de bases de données

Cedric Coulon

To cite this version:

Cedric Coulon. Réplication Préventive dans une grappe de bases de données. Réseaux et télécommunications [cs.NI]. Université de Nantes, 2006. Français. NNT : . tel-00481299

HAL Id: tel-00481299
https://theses.hal.science/tel-00481299

Submitted on 6 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Centrale de Nantes   Université de Nantes   École des Mines de Nantes

ÉCOLE DOCTORALE STIM
« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DES MATÉRIAUX »

Année 2007

N° attribué par la bibliothèque

Réplication Préventive dans une grappe de bases de données

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE NANTES

Discipline : INFORMATIQUE

présentée et soutenue publiquement par

Cédric COULON

le 19 Septembre 2006

au Laboratoire d'Informatique de Nantes Atlantique (LINA)

devant le jury ci-dessous

Président	Ricardo JIMÉNEZ PERIS, École polytechnique de Madrid
Rapporteurs	Georges GARDARIN, Professeur, Université de Versailles Saint-Quentin-en-Yvelines
	Ricardo JIMÉNEZ PERIS, Professeur, École polytechnique de Madrid
Examinateurs	Stéphane GANÇARSKI, Maître de conférences, Université Pierre et Marie Curie
	Rui Carlos OLIVEIRA, Professeur, Université de Minho
	Esther PACITTI, Maître de conférences, Université de Nantes
	Patrick VALDURIEZ, Directeur de recherche, INRIA

Directeur de thèse : Pr. Patrick VALDURIEZ
Encadrants de thèse : Mc. Esther PACITTI
Laboratoire : LINA (Laboratoire d'Informatique de Nantes Atlantique)

N° ED 0366-XXX

RÉPLICATION PRÉVENTIVE DANS UNE GRAPPE DE BASES DE DONNÉES

Preventive Replication in a Database Cluster

Cédric COULON

favet neptunus eunti

Université de Nantes

Cédric COULON

Réplication Préventive dans une grappe de bases de données

viii+154 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `these-LINA` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-LINA` est disponible à l'adresse :

`http://login.lina.sciences.univ-nantes.fr/`

Impression : these.tex – 10/10/2007 – 10:55

Révision pour la classe : $Id: these-LINA.cls,v 1.3 2000/11/19 18:30:42 fred Exp

Sommaire

Introduction 1
1 Réplication dans les grappes de bases de données 7
2 Architecture pour une grappe de bases de données 39
3 Algorithme de réplication préventive 47
4 Optimisation des temps de réponse 69
5 Le prototype RepDB* 85
6 Validation 97
Conclusion 115
Liste des publications 121

Bibliographie 123
Table des matières 131
A Code source de RepDB* 137

Introduction

La haute performance et la haute disponibilité des bases de données ont été traditionnellement gérées grâce aux systèmes de bases de données parallèles, implémentés sur des multiprocesseurs fortement couplés. Le traitement parallèle des données est alors obtenu en partitionnant et en répliquant les données à travers les noeuds du multiprocesseur afin de diviser les temps de traitement. Cette solution requiert un Système de Gestion de Base de Données (SGBD) ayant un contrôle total sur les données. Bien qu'efficace, cette solution s'avère très coûteuse en termes de logiciels et de matériels.

De nos jours, les grappes d'ordinateurs personnels (PC) fournissent une alternative aux multiprocesseurs fortement couplés. Le concept de grappe est né en 1995 avec le projet Beowulf sponsorisé par la NASA. Un des buts initiaux du projet était l'utilisation des ordinateurs massivement parallèles pour traiter les larges quantités de données utilisées dans les sciences de la terre et de l'espace. La première grappe a été construite pour adresser des problèmes liés aux gigantesques volumes de données des calculs sismiques. Les raisons d'être des grappes sont multiples :

- La démocratisation des ordinateurs permet désormais l'utilisation de nouveaux types de composants produits en masse ;
- La disponibilité de sous-ensembles entièrement assemblés (microprocesseurs, cartes mères, disques et cartes d'interface de réseau) ;
- La concurrence des marchés grand-public qui a poussé les prix vers le bas et la fiabilité vers le haut pour ces sous-ensembles ;
- L'existence des logiciels libres, des compilateurs et des outils de programmation parallèles ;
- L'expérience accumulée par les chercheurs dans les algorithmes parallèles ;
- Les progrès très significatifs dans le développement des protocoles réseaux à hautes performances à base de composants standard ;
- L'augmentation des besoins de calcul dans tous les domaines.

Ces nouvelles solutions permettent aux utilisateurs de bénéficier de plates formes fiables et très performantes à des prix relativement faibles : le rapport prix / performance d'une grappe de PC est de 3 à 10 fois inférieur à celui des supercalculateurs traditionnels. Les grappes sont composées d'un ensemble de serveurs (PC) interconnectés entre eux par un réseau. Ils permettent de répondre aux problématiques de haute performance et de haute disponibilité. Elles ont été utilisées avec succès [Top06] pour, par exemple, les moteurs de recherches internet utilisant des fermes de serveurs à grands volumes (e.g., Google).

Les applications dans lesquelles sont utilisées les grappes sont typiquement des applications de lectures intensives, ce qui rend plus facile l'exploitation du parallélisme. Cependant,

les grappes peuvent également être utilisées dans un nouveau modèle économique, les Fournisseurs de Services d'Applicatons (*ASP - Application Service Providers*). Dans un contexte ASP, les applications et les bases de données des clients sont stockées chez le fournisseur et sont disponibles, typiquement depuis Internet, aussi efficacement que si elles étaient locales pour les clients. Pour améliorer les performances, les applications et les données peuvent être répliquées sur plusieurs noeuds. Ainsi, les clients peuvent être servis par n'importe quel noeud en fonction de la charge. Cet arrangement fournit également une haute disponibilité: dans le cas de la panne d'un noeud, d'autres noeuds peuvent effectuer le même travail. Un autre avantage du modèle ASP concerne le déploiement. La mise à jour d'une application ou d'un SGBD ne demande pas le déplacement d'un technicien chez tous les clients. La mise à jour des noeuds chez le fournisseur est suffisante. En revanche, on ne peut pas faire d'hypothèses sur les applications disponibles ou sur le type de SGBD. Il se peut que certaines soient des applications qui produisent des écritures intensives et les SGBD doivent être en mesure de garder leur autonomie. La solution consistant à utiliser des SGBD parallèles n'est pas appropriée car elle est coûteuse et requiert une migration lourde vers les SGBD parallèles qui ne respectant pas l'autonomie des bases de données.

La gestion de la cohérence dans les grappes

Dans cette thèse, nous considérons une grappe de PC avec des noeuds similaires, chacun ayant son propre processeur, sa mémoire et son disque. À l'instar des multiprocesseurs, différentes architectures de grappes sont possibles : *disques partagés*, *mémoire partagée* et *sans partage*. Les architectures à *disques partagés* et à *mémoire partagée* demandent une interconnexion matérielle spéciale fournissant un espace commun à tous les noeuds. L'architecture *sans partage (shared-nothing)* est la seule architecture qui fournit une autonomie suffisante des noeuds. De plus, ce type d'architecture permet de passer à l'échelle pour les grandes configurations.

En s'appuyant sur ce type d'architecture, notre principal objectif dans cette thèse est de gérer la réplication des données sur les copies à travers la grappe. Le problème majeur de la réplication est de garantir la cohérence des copies lors de la présence de mises à jour. La solution de base dans les systèmes distribués garantissant une cohérence forte est la réplication synchrone (typiquement en utilisant le protocole *ROWA - Read One Write All*). Lorsque une transaction met à jour une copie, toutes les autres copies sont également mises à jour dans la même transaction distribuée. Ainsi, la cohérence mutuelle des copies est assurée. Cependant, la réplication synchrone n'est pas appropriée dans une grappe de bases de données pour deux raisons principales. Premièrement, tous les noeuds doivent implémenter de manière homogène le protocole ROWA dans leur gestionnaire de transaction local, ce qui implique la violation de l'autonomie des noeuds. Deuxièmement, la validation atomique des transactions distribuées (comme la Validation à 2 Phases - 2PC) est connue pour être coûteuse et bloquante, ne permettant pas le passage à l'échelle du système.

Une meilleure solution passant à l'échelle est la réplication asynchrone, où une transaction

peut être validée après avoir mis à jour une seule copie, la copie primaire. Après la validation de cette transaction, les autres copies, les copies secondaires, sont mises à jour dans des transactions séparées. Quand un seul noeud possède une copie primaire, bien que relâchant légèrement la cohérence mutuelle, la cohérence forte est garantie. Du fait des restrictions sur le placement des copies primaires et secondaires, la réplication synchrone limite le nombre de configurations utilisables. De plus, comme les données doivent être mises à jour sur les copies primaires, le système devient plus sensible aux pannes des noeuds possédant des copies primaires. Lorsque plusieurs noeuds possèdent des copies primaires, la disponibilité du système augmente, mais en contrepartie, il faut gérer les transactions conflictuelles sur différentes copies primaires susceptibles d'introduire des incohérences dans le système.

Ces dernières années, l'utilisation des techniques de communication de groupes a beaucoup influencé les protocoles de réplication, particulièrement pour les techniques asynchrones. Les communications de groupe sont en fait un ensemble de primitives et d'outils conçus pour aider à la construction de services répliqués. La notion de communication de groupe provient du monde des systèmes distribués et est utilisée dans des systèmes comme Amoeba [KT91]. La recherche sur les groupes de communications a donné des résultats sur le plan théorique et pratique (prototypes). Puis peu à peu, les prototypes se sont éloignés des systèmes distribués et sont devenus des boîtes à outils autonomes utilisées pour tout type de réplication. Toutes ces boîtes à outils offrent un niveau similaire d'abstractions de haut-niveau : les propriétés d'ordonnancement et les propriétés de fiabilité. Les propriétés d'ordonnancement assurent que les messages sont envoyés à tous les membres dans un certain ordre et les propriétés de fiabilité assurent que tous les membres d'un groupe sont d'accord sur une même valeur. L'utilité de ces deux propriétés devient évidente pour l'utilisation d'un algorithme de réplication qui garantit la cohérence des données et donc la sérialisabilité des transactions.

Contributions de la thèse

Les algorithmes de réplication existants ne sont pas forcément adaptés à la réplication dans les grappes de PC, soit parce qu'ils ne passent pas à l'échelle (réplication synchrone) soit parce qu'ils sont trop restrictifs sur le placement des données (réplication asynchrone). De plus, tous ne respectent pas l'autonomie des SGBD et donc des noeuds, ce qui est incompatible avec un contexte ASP.

Dans cette thèse, nous proposons un nouvel algorithme de réplication basé sur la réplication préventive de Pacitti et al. [PMS99, PMS01]. Notre algorithme utilise une technique asynchrone qui garantit la cohérence forte en utilisant les services réseau et notamment la communication de groupes. Il supporte les configurations où toutes les données sont totalement répliquées et peuvent être mises à jour sur n'importe quel noeud. Il supporte aussi les configurations où une partie des copies seulement est répliquée.

Cette thèse apporte plusieurs contributions :

- Une architecture sans partage divisée en cinq couches. Chaque couche est indépendante

l'une de l'autre ce qui permet de garder les noeuds autonomes tout comme les couches entre elles. Ce type d'architecture est particulièrement adapté au contexte ASP [PÖC03] ;
- Un algorithme de réplication préventive qui représente la couche de gestion de la réplication dans notre architecture globale. L'algorithme supporte les copies primaires, secondaires et multimaîtres, ce qui autorise un grand nombre de configurations. Le SGBD est ici considéré comme une boîte noire, l'autonomie des noeuds est ainsi respectée et l'algorithme peut être adapté à un grand nombre de SGBD. Nous proposons en plus de l'algorithme de réplication une architecture pour la couche de réplication et également les preuves montrant que la cohérence des données est bien assurée [CPV04, PCVÖ05] ;
- Des optimisations pour améliorer les temps de réponse lorsque le système est soumis à de fortes charges. Nous montrons comment exécuter les transactions de manière optimiste pour supprimer le délai introduit par l'ordonnancement des transactions sans relâcher l'incohérence. Enfin, afin d'augmenter le débit des transactions, nous montrons comment autoriser l'exécution des transactions en parallèle tout en gardant une cohérence forte sur l'ensemble du système [CPV05b, PCVÖ05] ;
- Le prototype RepDB*, un logiciel libre implémentant le gestionnaire de réplication et utilisable sous PostgreSQL. Ce logiciel permet de valider notre algorithme en montrant le passage à l'échelle et les gains de performance de notre solution [CGPV04][VPC05].

Organisation de la thèse

Dans le Chapitre 1, nous présentons les techniques de réplication existantes et les critères pour les classifier. Nous présentons également trois modèles de cohérences: la cohérence faible, la cohérence forte et la cohérence *Snapshot*. Ensuite, nous montrons les spécificités de la réplication dans les grappes de PC en décrivant les différentes architectures disponibles. Puis, nous montrons la spécificité de la réplication dans les grappes en décrivant les architectures disponibles et l'exploitation possible de la communication de groupe. Ensuite, nous décrivons trois algorithmes représentatifs de la réplication dans les grappes.

Dans le Chapitre 2, nous présentons l'architecture que nous avons choisie. Celle-ci est basée sur un modèle sans partage. Elle est composée de cinq couches indépendantes. Nous présentons également un exemple de routage d'une requête d'un client dans notre système. Nous montrons que chaque noeud peut traiter indépendamment n'importe quelle requête. Finalement, nous présentons toutes les caractéristiques des noeuds nécessaires à l'exploitation de notre algorithme de réplication dans une grappe. Nous nous appuyons sur des services de communication qui exploitent les propriétés de rapidité et de fiabilité du réseau dans les grappes.

Dans le Chapitre 3, nous nous intéressons à la mise au point de la couche de gestion de la réplication de notre architecture. Nous nous focalisons plus particulièrement sur l'élaboration d'un algorithme de réplication, appelé *Algorithme de réplication préventive*. Nous reprenons l'algorithme de réplication asynchrone *Lazy-Master* de Pacitti et al. [PMS01, PMS99] en tentant de l'améliorer. Dans un premier temps, nous ajoutons le support des *Copies multimaîtres* ; dans ce mode, plus d'un noeud peut effectuer des mises à jour sur une même copie. Puis nous faisons évoluer l'algorithme pour autoriser la réplication partielle des données. Ainsi, les noeuds ne

sont plus obligés de détenir toutes les copies et celles-ci peuvent être de type copies primaires, multimaîtres ou secondaires sans aucune restriction sur le placement.

Le Chapitre 4 présente des optimisations de notre algorithme de réplication. Nous supprimons ainsi le délai d'attente des transactions introduit par l'algorithme de réplication préventive. Nous introduisons ensuite une modification autorisant l'exécution en parallèle des transactions. Finalement, nous démontrons la validité de nos optimisations en prouvant que les algorithmes n'introduisent pas d'incohérences.

Le Chapitre 5 présente le prototype *RepDB** [ATL05] qui est l'implémentation de l'algorithme de réplication préventive. *RepDB** est un logiciel libre ayant fait l'objet d'un dépôt à l'Agence Pour la Protection des logiciels (APP). Nous décrivons donc la plate forme logicielle utilisée pour implémenter RepDB*. Puis nous montrons son déploiement et sa configuration.

Dans le Chapitre 6, nous utilisons le prototype RepDB* pour montrer les propriétés de l'algorithme de réplication préventive vues au Chapitre 3 et ses optimisations vues Chapitre 4. Nous voulons prouver que l'algorithme peut être utilisé sur une grappe à grande échelle sans pertes de performance. Nous voulons également comparer l'influence de la réplication partielle sur les performances par rapport à la réplication totale. Un point important dans les algorithmes de réplication asynchrone est de déterminer les différences entre les copies. Pour cela, nous présentons un nouveau critère de qualité : le degré de fraîcheur d'un système. Nous décrivons ainsi l'implémentation de notre système et nous définissons deux bancs d'essai : un spécifiquement créé pour notre algorithme et un banc d'essai standard, TPC-C. Pour chacun d'entre eux, nous décrivons leurs configurations (tables, placements des données), les transactions utilisées et les paramètres qui varient au cours des tests.

Finalement, dans le dernier Chapitre, nous concluons et nous ouvrons quelques perspectives de recherche sur cette thèse.

CHAPITRE 1

Réplication dans les grappes de bases de données

1.1 Introduction

Dans le calcul en grappes, la charge est distribuée parmi plusieurs noeuds connectés par un réseau rapide. Les grappes de PC (*cluster*) sont utilisées pour le passage à l'échelle et la tolérance aux pannes [GNPV02, JPPMA02]. Une grappe de PC est définie comme un ensemble limité (de quelques dizaines à plusieurs milliers) de systèmes inter-connectés qui partagent des ressources de façon transparente [Che00]. Chaque système peut être autonome et les ressources peuvent être des processeurs, de la mémoire, des dispositifs d'entrées-sorties, des organes de stockages. Les systèmes qui composent une grappe sont appelés des noeuds. L'objectif d'une grappe est d'offrir la continuité des services en cas de panne d'un noeud et un accroissement de la capacité de calcul par le regroupement de systèmes existants. Contrairement à un système distribué, une grappe : (i) est homogène, les noeuds sont similaires et sont gérés par une même version du système d'exploitation ; (ii) offre une image système unique, les utilisateurs ont l'impression qu'ils sont sur le même système quelque soit le noeud par lequel ils se connectent ; (iii) et sont concentrés géographiquement, les noeuds sont proches les uns des autres ce qui facilite leur maintenance. Les grappes offrent donc un accroissement de la capacité de traitement, une haute disponibilité et une tolérance aux pannes. Lorsque la charge augmente, de nouveaux noeuds sont ajoutés au système pour augmenter les capacités de calcul. De plus, la panne d'un noeud ne pénalise pas les autres noeuds. Dans certains cas, le travail en cours sur le noeud en panne peut même être récupéré par un autre noeud.

Quand les grappes sont utilisées avec des systèmes de gestion de bases de données (SGBD), une solution pour stocker les données est de les partitionner parmi les différents noeuds, chaque noeud n'exécutant que les transactions qui concernent ses données. Le partitionnement est une solution simple mais comporte plusieurs inconvénients. Premièrement, il est difficile de diviser les données afin d'obtenir une charge équitable sur l'ensemble des noeuds. Deuxièmement, il est nécessaire de mettre en place un gestionnaire de transactions distribuées nécessaire lorsqu'une transaction veut accéder à des données qui se trouvent sur différents noeuds. La plupart des transactions issues des SGBD accèdent aux mêmes données. Pour augmenter le parallélisme entre les transactions, il est alors plus efficace de répliquer cette partie de la base que de la

stocker sur un seul noeud. La réplication est alors une solution qui offre ici plusieurs avantages :

- L'*équilibrage de charge* est possible s'il n'y a pas de restriction sur le lieu d'exécution des transactions (et c'est rarement le cas).
- La *gestion des transactions distribuées* peut être évitée si toutes les données sont répliquées sur tous les noeuds. En fait, pour se passer des transactions distribuées, il est suffisant qu'un seul noeud possède toutes les données, ainsi n'importe quelle transaction peut être exécutée au moins sur ce noeud.
- La *tolérance aux pannes* est un des objectifs évidents de la réplication. La réplication est une approche naturelle aux problèmes des défaillances : si une copie ne fonctionne plus, on a recours à une autre copie.

Avant de proposer un algorithme de réplication exploitant au mieux les avantages des grappes, nous faisons un survol de la réplication des données distribuées où nous classons les algorithmes selon trois critères qui influencent la disponibilité des données : où, quand et comment sont mises à jours les données. De plus, chaque algorithme propose un modèle de cohérence des données dans le système. Un modèle de cohérence définit la divergence maximale autorisée par l'algorithme sur les données entre plusieurs noeuds. Nous présentons trois modèles de cohérence différents. Le premier modèle, dit cohérence forte, offre peu de divergence entre les données au détriment des performances. Le deuxième modèle, dit cohérence faible, permet de meilleures performances, mais au prix du relâchement de la cohérence. Le troisième modèle, dit cohérence *Snapshot*, propose une extension du modèle de cohérence forte pour les systèmes qui utilisent une isolation *Snapshot*.

Une autre propriété de notre algorithme est de garantir l'autonomie des noeuds et ceci pour autoriser l'utilisation de noeuds hétérogènes. Ici, le terme "noeuds hétérogènes" s'applique au fait que les noeuds peuvent avoir des capacités de calculs différentes (Processeur, Mémoire, Disque de stockage, ...), mais ils contiennent surtout des applications et des SGBD très variés (des SGBD Oracle et PostgreSQL peuvent être mélangés sur une grappe ou même sur un même noeud). Les noeuds restent cependant homogènes dans le sens où ils utilisent la même architecture matérielle et qu'ils sont gérés par des systèmes d'exploitation compatibles. Par la suite, nous montrons les configurations utilisables dans une grappe et leurs influences sur l'autonomie des noeuds. À l'instar des noeuds, l'autonomie des SGBD doit aussi être respectée, elle se traduit par la visibilité du SGBD pour la couche de réplication sur le noeud. De plus, une nouvelle approche au problème de la réplication dans les grappes a récemment été proposée [SPS+05] : elle s'appuie sur une infrastructure de communication généralement appelée communications de groupes. Cette infrastructure permet, au moyen de primitives de haut niveau, de simplifier l'architecture et l'implémentation de techniques de réplication. Nous verrons comment ces techniques ont été utilisées pour fournir la réplication.

La suite de ce chapitre est organisée comme suit. Dans un premier temps, nous présentons les techniques de réplication existantes. Nous décrivons également des critères pour classifier ces algorithmes. Puis nous présentons les différents modèles de cohérences existants. Dans

la section suivante, nous montrons les spécificités de la réplication dans les grappes de PC en décrivant les différentes architectures disponibles. Puis, nous spécifions les propriétés de la communication de groupe qui peuvent exploiter les capacités réseaux des grappes pour la réplication. Ensuite, nous décrivons trois algorithmes représentatif de la réplication dans les grappes. Finalement, nous concluons.

1.2 Réplication des données distribuées

Dans cette section, nous définissons tous les éléments nécessaires à la mise en oeuvre de la réplication des données dans une grappe. Dans un premier temps, nous présentons les différents types de copies et où elles sont stockées avec quels droits en lectures et écritures. Puis, nous introduisons les différentes configurations. Pour finir, nous présenterons les modes de réplications existants en utilisant une classification selon trois critères.

1.2.1 Les types de copies

Nous supposons qu'une copie est une table relationnelle entière. Il est alors facile de considérer des partitions horizontales ou verticales, il suffit de diviser la table avant son placement sur la grappe, les partitions devenant autant de copies distinctes. Soit une table R, il peut exister trois différents types de copies : primaires, secondaires et multimaîtres. Une copie primaire, que l'on notera R, est stockée sur un noeud appelé noeud maître. Une copie primaire peut être lue et mise à jour. Alors qu'une copie secondaire, que l'on notera r_i, est stockée sur un ou plusieurs noeuds esclaves i en lecture seule. Finalement une copie multimaître, que l'on notera R_i, est stockée sur un ou plusieurs noeuds multimaître i où la copie peut être lue et mise à jour.

Au sein d'une même configuration, plusieurs copies primaires pour une même copie correspondent à des copies multimaîtres. De même, si dans une configuration, on ne retrouve qu'une seule copie multimaître pour une même copie, alors c'est une copie primaire. Le nombre de copies secondaires n'étant pas important.

1.2.2 Les types de configurations

Dans cette section, nous présentons trois types de configurations générales : maître-paresseux, réplication totale et réplication partielle. La réplication maître-paresseux utilise uniquement des copies primaires et secondaires. La réplication totale n'utilise que des copies multimaîtres. La réplication partielle utilise tous les types de copies.

Dans nos exemples (Figure 1.1) nous utilisons la notation présentée dans la section précédente pour des configurations où sont stockées deux tables R et S.

1.2.2.1 Réplication maître-paresseux

La Figure 1.1a montre une configuration maître-paresseux (*Lazy-master replication*). Ces configurations sont composées uniquement de copies primaires et de copies secondaires. Ce

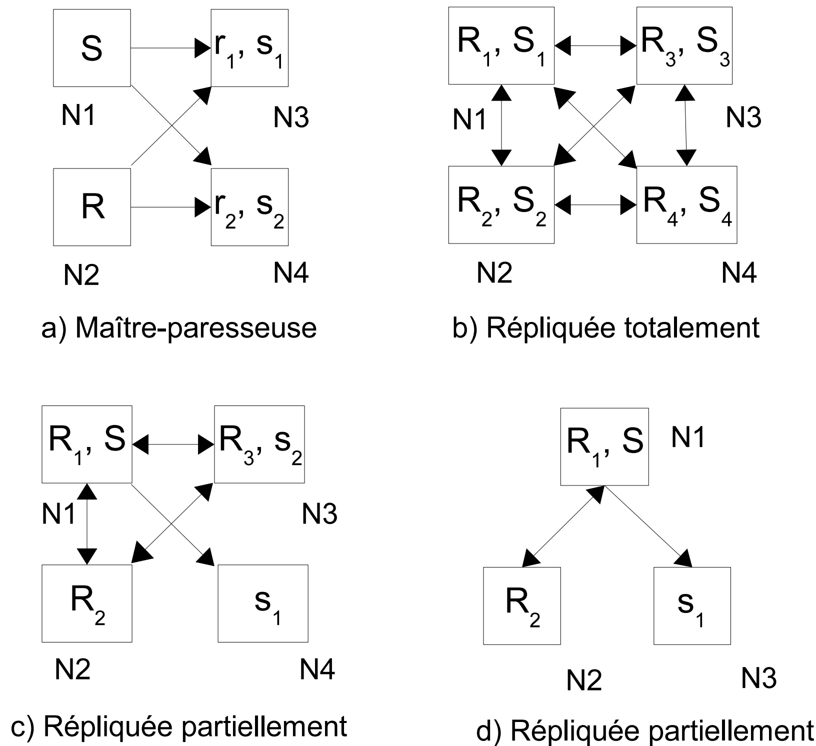


Figure 1.1 – Exemples de configurations avec la réplication de deux tables R et S

type de configuration est utile pour augmenter le débit des requêtes en lecture seule grâce à l'ajout de noeuds esclaves qui ne gèrent pas les transactions de mises à jour, typiquement pour des applications de type entrepôts de données. Cependant, dans toutes ces configurations à copie primaire, la disponibilité des données est limitée car dans le cas de panne du noeud maître la copie ne peut plus être mise à jour. On nomme cette configuration paresseuse, car dans un premier temps seul le maître est mis à jour, puis dans un second temps on rafraîchit les copies secondaires.

1.2.2.2 Réplication totale

La Figure 1.1b montre une configuration avec une réplication totale (*Full replication*). Dans cette configuration, tous les noeuds possèdent toutes les copies en mode multimaître. Ici, tous les noeuds doivent gérer toutes les transactions de mises à jour. En effet, lorsque R ou S est mise à jour sur un noeud, toutes les autres copies ont besoin d'être rafraîchies. Ainsi, seules les requêtes de lecture diffèrent sur chaque noeud. Cependant, comme tous les noeuds exécutent toutes les transactions, tous les noeuds ont la même charge (lorsque les noeuds sont homogènes) et l'équilibrage de charges en est facilité. De plus, la disponibilité est améliorée puisqu'en cas de panne, n'importe quel noeud peut être remplacé par n'importe quel autre.

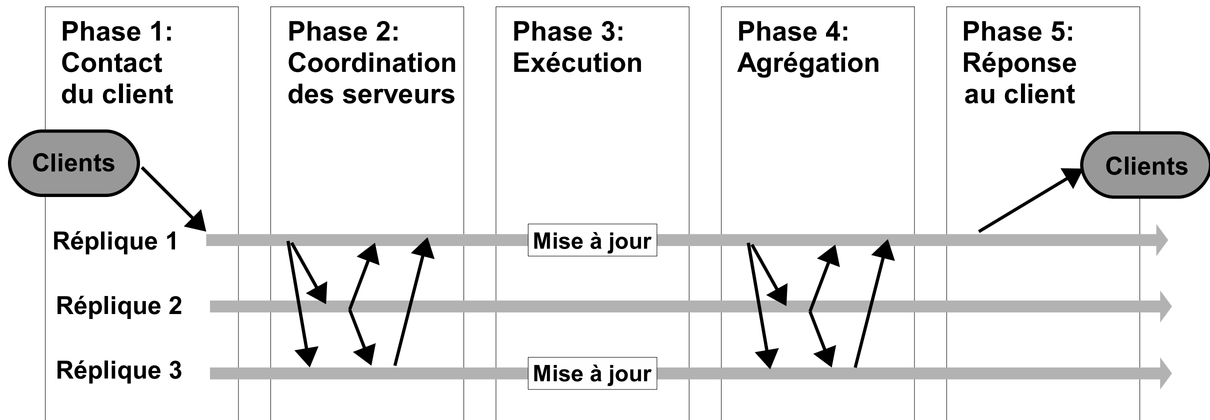


Figure 1.2 – Modèle général à 5 phases

1.2.2.3 Réplication partielle

Les Figures 1.1c et 1.1d montrent des configurations avec une réplication partielle (*Partial replication*) où l'on peut retrouver n'importe quel type de copies sur les noeuds (on les appelle aussi configurations hybrides). Par exemple, sur la Figure 1.1c, le noeud N_1 possède la copie multimaître R_1 et la copie primaire S, le noeud N_2 possède la copie multimaître R_2 et la copie secondaire s_1, et finalement le noeud N_4 possède la copie secondaire s_2. Comparés aux configurations à réplication totale, seulement une partie des noeuds est affectée par les transactions de mises à jour sur une copie multimaître. Ainsi, les transactions n'ont pas à être diffusées à tous les noeuds. Les noeuds sont alors moins chargés et le nombre de messages échangés est diminué. Malgré cela, ce type de configuration assure toujours la disponibilité des noeuds : on peut répliquer totalement les tables sur plusieurs noeuds, les autres noeuds sont spécialisés à certains types de transactions (par exemple sur la Figure 1.1c, le noeud N_4 est spécialisé pour les requêtes de lecture sur la table S).

1.2.3 Classification des modes de réplication

Dans cette section, nous présentons dans un premier temps un modèle fonctionnel général pour les algorithmes de réplications. Puis nous verrons les deux critères introduits par Gray et al. [GHOS96] pour classer les algorithmes : *où* sont mises à jour les données et *quand*. Finalement, nous présentons un troisième critère de classification qui indique *comment* les données sont mises à jour [PMS99, WPS$^+$00a].

1.2.3.1 Modèle fonctionnel général à cinq phases

On peut représenter un algorithme de réplication par un modèle fonctionnel qui comprend cinq phases [WPS$^+$00b] comme le montre la Figure 1.2 :

- Phase 1 (Requête) : le client soumet une transaction à un noeud.

- Phase 2 (Coordination des serveurs) : les serveurs se coordonnent afin de savoir qui doit exécuter la transaction (ordonnancement des transactions et prises de verrou...).
- Phase 3 (Exécution) : La transaction est exécutée sur les copies concernées.
- Phase 4 (Agrégation) : Les noeuds se mettent d'accord sur le résultat de la transaction (pour garantir l'atomicité).
- Phase 5 (Réponse) : La réponse est renvoyée au client.

Selon le mode de réplication, les phases ne s'exécutent pas toutes dans cet ordre, par exemple dans la réplication asynchrone, la phase d'agrégation (phase 4) a lieu avant la phase de réponse au client (phase 5). Et parfois, certaines phases ne se sont même pas nécessaires au bon déroulement de la réplication.

1.2.3.2 Mises à jour Copie Primaire ou Partout

Pour savoir sur quelles copies une transaction peut être exécutée (*"où ?"*), il existe deux possibilités : le mode centralisé et le mode distribué.

Le modèle de *Mises à jour Copie Primaire (Primary Copy - PC)* n'accepte qu'une copie primaire par table. Ainsi, seul le noeud maître peut recevoir des transactions de mises à jour sur la donnée qu'il possède, les noeuds secondaires n'exécutent que des transactions de lecture. Le contrôle de concurrence est ici fortement réduit du fait que le noeud maître sert de référence aux autres noeuds pour l'ordonnancement des requêtes. Les transactions peuvent être exécutées dès leur réception sur le noeud maître sans risque de conflits, car aucune transaction de mise à jour ne peut être exécutée sur un autre noeud. Les noeuds secondaires se contentent d'appliquer l'effet de la transaction sur leur copie. Par contre, le noeud maître devient un goulot d'étranglement potentiel et le système est très sensible aux pannes du noeud. De plus, cette approche limite l'équilibrage de charge pour les transactions de mises à jour au contraire de l'approche distribuée.

Dans le modèle *Mises à jour Partout (Update Everywhere - UE)*, le système supporte des copies multimaîtres (au moins 2 noeuds sont des copies primaires). Dans ce cas, il faut garantir l'ordonnancement des transactions à la fois sur les noeuds secondaires et multimaîtres. Il faut noter que dans la littérature, on ne fait pas la distinction entre les configurations où toutes les copies sont multimaîtres et celles où seulement une partie des noeuds possèdent des copies multimaîtres (les autres noeuds possèdent des copies secondaires).

1.2.3.3 Synchrone / Asynchrone

Le deuxième critère de classification permet de définir le moment où sont rafraîchies les copies (*"quand ?"*). Elles peuvent être faites dans la transaction originale (*synchrone*) ou en dehors (*asynchrone*).

Synchrone. Les mises à jour sont effectuées en une seule transaction. Il y a une coordination forte entre les noeuds (tous valident la transaction ou aucun). La réponse au client est renvoyée après que les serveurs se soient coordonnés.

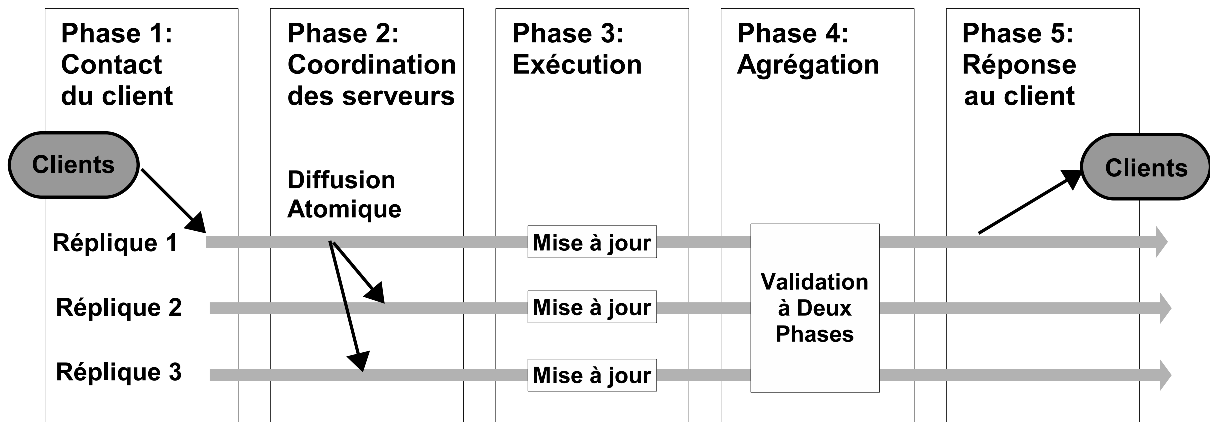


Figure 1.3 – Modèle synchrone à 5 phases

Comme dans un système centralisé, avant d'accéder à une donnée, on acquiert un verrou exclusif dessus. Avec le protocole *ROWA* (*Read One Write All*) [BHG87], on améliore la disponibilité des lectures en ne demandant le verrou que sur une copie pour les lectures et sur toutes les copies pour les écritures. Par la suite, le protocole est adapté aux pannes avec sa variante *ROWA-A* (*Read One Write All Available*) [BG84, GSC+83] où seules les copies disponibles sont mises à jour. Dès qu'une copie redevient disponible, elle doit d'abord être synchronisée avant d'être de nouveau utilisable. D'autres protocoles synchrones comme le protocole *NOn-Disjoint conflict classes and Optimistic multicast (NODO)* [KA00a, PMJPKA00] se basent sur la division des données en classes de conflits. On détermine ainsi que deux transactions accédant à la même classe de conflit ont une grande probabilité de conflit. À l'inverse, deux transactions qui n'accèdent pas aux mêmes classes peuvent s'exécuter en parallèle. Les classes de conflits doivent être connues à l'avance et chaque classe de conflit a un seul noeud maître. Pour exécuter une transaction, le protocole diffuse deux messages. Le premier message permet d'exécuter la transaction sur une copie ombre (une copie de la dernière version cohérente de la base), le deuxième message permet de valider la transaction ou alors de l'annuler pour la réordonnancer. L'algorithme utilisé dans le système Database State Machine [SOMP01, SPMO02] supporte les configurations répliquées partiellement et ne viole pas l'autonomie des noeuds. Cependant, il requiert un protocole de terminaison atomique du même type que la validation à 2 phases, et de tels protocoles sont connus pour ne pas passer à l'échelle.

Une stratégie alternative de réplication est basée sur les quorums. Ces protocoles requièrent d'accéder à un quorum de copies pour les opérations d'écritures et de lectures [Gif79, JM90, PL88, Tho79]. Tant que le quorum accepte d'exécuter une opération, l'opération peut s'exécuter. Par exemple, l'algorithme de consensus par quorum [Gif79] requiert deux quorums d'écritures et un quorum de lecture pour écrire sur une copie. Ainsi, au moins une des copies secondaires aura la dernière valeur. Les autres solutions combinent l'approche ROWA/ROWAA avec des protocoles utilisant des quorums.

La Figure 1.3 montre le modèle de réplication à 5 phases pour la réplication synchrone. Ce modèle utilise un protocole de validation de type Validation à 2 Phases (V2P) [GR93] pour valider la transaction. Le protocole s'assure alors que tous les noeuds font le même choix : vali-

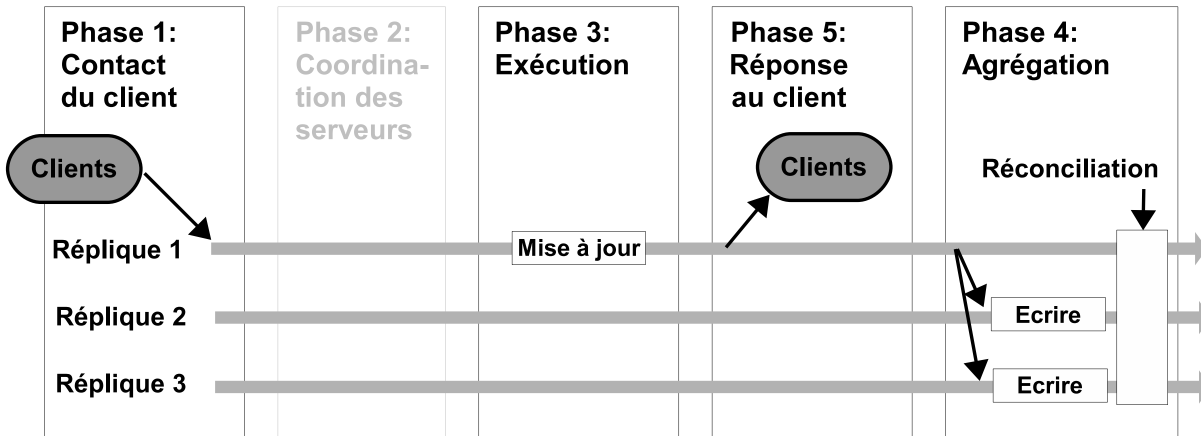


Figure 1.4 – Modèle asynchrone à 5 phases

dation ou abandon de la transaction. Ce modèle assure une cohérence forte sur toutes les copies.

Asynchrone. Les mises à jour se font dans des transactions séparées. Le serveur initiateur renvoie la réponse au client immédiatement après la mise à jour sur sa base de données. Il propage ensuite la mise à jour aux autres noeuds.

La Figure 1.4 montre le modèle de réplication à cinq phases pour la réplication asynchrone. Ce modèle a la particularité d'envoyer la réponse au client (phase 4) avant la coordination des copies et avant même la diffusion de la mise à jour sur les autres copies (phase 5). Cette propriété impose une contrainte forte, une réconciliation doit être effectuée pour assurer la cohérence de l'ensemble des copies. Cependant, les algorithmes de réplication asynchrone sont souvent déterministes : ils appliquent les mises à jour en étant sûrs qu'elles ne donneront pas d'incohérences. On peut classer les algorithmes de réplication synchrone en deux grandes familles : la réplication asynchrone optimiste et la réplication asynchrone pessimiste.

Avec la réplication asynchrone optimiste [SS05], les transactions sont exécutées en parallèle sans contrôle de cohérence. En cas de conflit, il faut réconcilier les copies pour que les données convergent vers la même valeur : soit en ré-exécutant les transactions et en les réordonnant, soit en annulant certaines transactions. Cette technique est efficace dans les cas où les conflits entre les transactions sont rares [AT02, GHOS96, PGV03, PGV04] et lorsque les transactions acceptent une certaine incohérence des données. L'un des avantages de la réplication asynchrone optimiste est que les transactions sont exécutées localement, un noeud du système peut donc exécuter une transaction en étant déconnecté du reste du système, la réconciliation se faisant à la reconnexion du noeud. De plus, en l'absence de contrôle de cohérence, les transactions sont exécutées sans les délais supplémentaires introduits par la réplication pessimiste synchrone ou asynchrone. Un désavantage de cette technique est que la réconciliation s'effectue après avoir renvoyé la réponse au client. Ainsi, le client peut recevoir une réponse incohérente et le noeud peut se trouver dans un état incohérent jusqu'à la réconciliation. En ce qui concerne le mécanisme de réconciliation, il peut s'avérer coûteux et le choix des transactions à réordonner ou à abandonner n'est pas trivial.

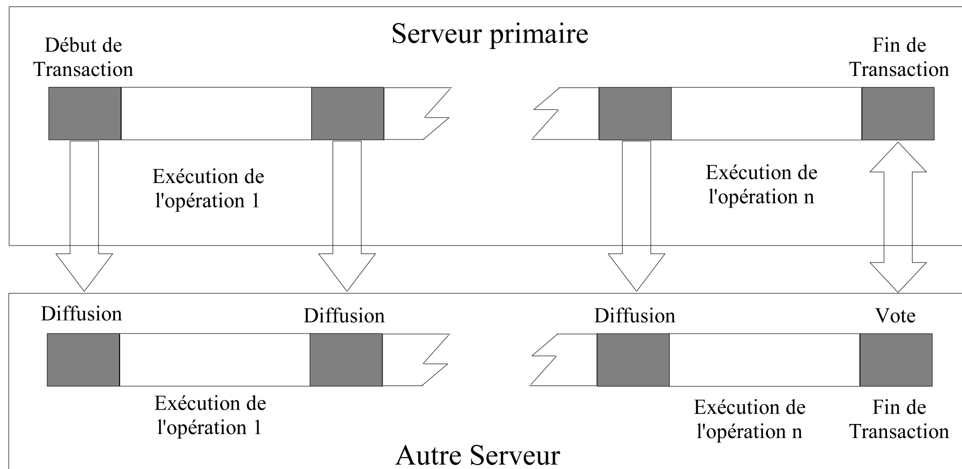


Figure 1.5 – Mode immédiat

La réplication asynchrone pessimiste [PV98], tout comme la réplication synchrone, garantit une cohérence forte. Chundi et al. [CRR96] ont montré, que même avec une configuration maître-paresseux, la sérialisabilité ne peut pas être garantie dans tous les cas. Pour contourner ce problème, la solution consiste à restreindre le placement des copies primaires et secondaires sur les noeuds. L'idée principale est de définir un ensemble de configurations autorisées en utilisant des graphes où chaque noeud du graphe est un site, et où un arc représente un lien entre une copie secondaire et une copie primaire d'une même donnée. Si le graphe est acyclique, la sérialisabilité peut être garantie en propageant les mises à jour peu après la validation d'une transaction [CRR96]. Pacitti et al. [PMS01, PMS99] améliorent ces résultats en autorisant certaines configurations acycliques mais les transactions sur les noeuds secondaires doivent être exécutées dans le même ordre total sur tous les noeuds en estampillant les transactions. Breitbart et al. [BKR+99] proposent une solution alternative en introduisant un graphe dirigé de la configuration (les arcs sont dirigés de la copie primaire vers les copies secondaires) qui ne doit pas avoir de cycles. Cette solution demande aussi une stratégie de propagation des mises à jour plus complexe. Elle transforme le graphe en arbre où une copie primaire n'est pas nécessairement directement connectée avec toutes ses copies secondaires. La propagation est alors effectuée le long du chemin du graphe. Breitbart et al. [BKR+99] proposent également une solution hybride où la réplication synchrone est utilisée lorsqu'il y a des cycles dans le graphe.

Bien que toutes ces solutions couvrent un large spectre de configurations, dans les applications réelles (et spécialement dans les grappes), la restriction sur le placement et l'approche centralisée des mises à jour sont trop restrictives et limitent le type de transactions autorisées à être exécutées.

1.2.3.4 Interaction : immédiate / différée

Le dernier critère de classification des algorithmes de réplication indique le degré de communication entre les serveurs (*"comment ?"*). Ceci permet de déterminer le trafic des messages

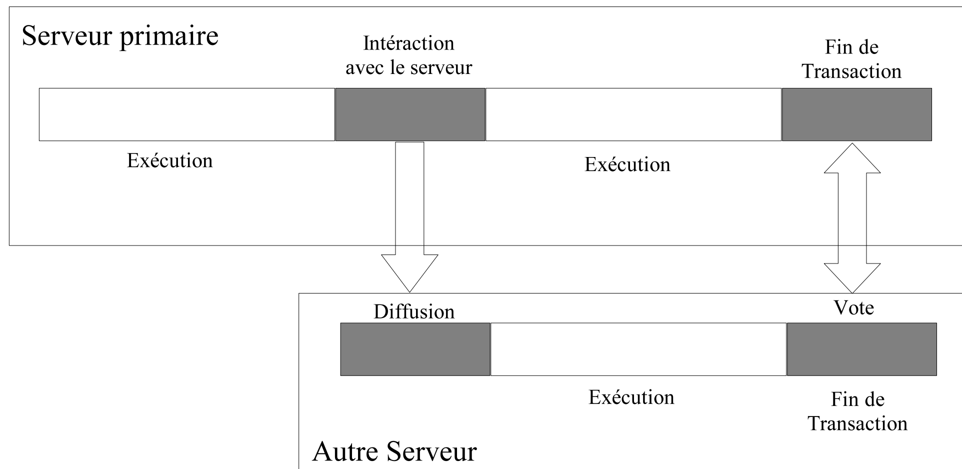


Figure 1.6 – Mode différé

généré par le protocole de réplication.

Une interaction immédiate (*immediate*) correspond à des techniques de réplication où chaque opération est envoyée aux autres noeuds le plus tôt possible. Comme on peut le voir sur la Figure 1.5, dans ce mode la mise à jour est plus rapide, car la transaction de rafraîchissement est démarrée avant la fin de la transaction sur le noeud d'origine. Le rafraîchissement des copies peut ainsi s'effectuer en parallèle avec l'exécution sur le noeud d'origine. L'inconvénient de ce mode est qu'il nécessite autant de messages qu'il y a d'opérations.

Une interaction différée (*deffered*) correspond à une technique de réplication où le nombre de messages ne dépend pas du nombre d'opérations de la transaction. Comme le montre la Figure 1.6, on réduit généralement le nombre de messages à un, cet unique message contient toute la transaction et est envoyé à la fin de l'exécution de la transaction. L'avantage est que le nombre de messages nécessaire pour le rafraîchissement des copies est minimisé (au minimum un message). En revanche, le message de rafraîchissement n'est envoyé qu'à la fin de l'exécution de la transaction , ce qui introduit un délai supplémentaire avant de pouvoir mettre à jour les copies.

Il faut noter que le type d'interaction choisi est rarement dépendant de l'algorithme de réplication. L'adaptation d'un mode d'interaction vers l'autre est souvent possible et les implémentations des algorithmes de réplication existent souvent avec les deux modes.

1.3 Cohérence des données

Dans cette section, nous définissons les critères de cohérence des données que l'on peut trouver dans un système de réplication [ABKW98, GM79, Ora95, Val93]. Afin d'améliorer les performances, il est possible de relâcher la cohérence des données. On classe traditionnellement les algorithmes de réplication en deux catégories : ceux qui garantissent une cohérence forte des données tout au long du processus de réplication et ceux qui autorisent une divergence entre les copies pendant une certaine période de temps, aussi appelé cohérence faible. Finalement, nous

présenterons plus en détail un troisième modèle de cohérence, la cohérence *Snapshot*. Pour ce dernier modèle de cohérence nous rentrons plus dans le détail car il est beaucoup plus récent que les autres modes de cohérences même s'il est dérivé de la cohérence forte et qu'il s'appuie sur un isolement *Snapshot* des données.

1.3.1 Cohérence forte

Le critère d'exactitude le plus fort pour un système répliqué est la sérialisabilité-1-copie (SR1C) [BHG87] : en dépit de l'existence de multiples versions d'une copie, un objet apparaît comme une copie logique (équivalence-1-copie) et l'exécution de transactions concurrentes est coordonnée de façon à ce qu'elle soit équivalente à une exécution sérielle sur la copie logique. Rappelons qu'une exécution est sérielle si toutes les actions des transactions ne sont pas entrelacées, une transaction doit se terminer entièrement avant qu'une autre ne commence. De plus, l'atomicité des transactions garantit qu'une transaction est validée sur tous ou aucun des noeuds participants malgré la possibilité de pannes. Tous les protocoles de réplications ne garantissent pas la sérialisabilité-1-copie ou l'atomicité. La plupart des algorithmes synchrones et asynchrones pessimistes garantissent une cohérence forte.

1.3.2 Cohérence faible

De récentes recherches [PL91, KB94, SAS^{+}96] fournissent aux utilisateurs un moyen de contrôler l'incohérence, bien que les données soient obsolètes ou même incohérentes. Le degré à partir duquel une donnée est considérée comme incorrecte est limité et bien défini. De nombreux modèles à cohérence faible ont été définis pour fournir une cohérence plus faible que la sérialisabilité-1-copie. Des exemples de modèles de réplication à cohérence faible sont la Sérialisabilité-Epsilon [PL91] et N-Ignorance [KB94]. La Sérialisabilité-Epsilon mesure la distance entre les éléments de la base de données. Une distance entre deux éléments peut être représentée par exemple par la différence des valeurs des éléments ou encore la différence de nombres de mises à jour effectuées sur les éléments. L'application peut ensuite spécifier la quantité d'incohérence tolérée par une transaction. Le modèle N-Ignorance est basé sur les quorums. Il relâche les besoins des quorums de telle manière que les incohérences introduites par des transactions concurrentes sont limitées.

Le système de réplication de *Mariposa* [SAS^{+}96] construit une plate forme économique pour la réplication de données. La fréquence des propagations de mises à jour dépend de combien l'administrateur d'une copie est prêt à payer. En outre, le manque de fraîcheur d'une donnée dans une requête est déterminé par le prix qu'un utilisateur veut payer. Cependant, pour toutes ces approches, faire le choix d'une limite d'incohérence d'une donnée n'est pas facile et les utilisateurs doivent avoir une bonne compréhension de la mesure de l'incohérence.

Ce niveau de cohérence est aussi appelé cohérence éventuelle : si le système n'accepte plus de transactions, toutes les données deviennent identiques au bout d'un temps fini.

1.3.3 Cohérence *Snapshot*

De nombreux SGBD [IBM05, Mic05, Ora04, Pos05] ont récemment adopté un mode d'isolation des transactions appelé isolation *Snapshot* (Snapshot Isolation - SI) [BBG+95]. Son principe est basé sur l'idée que chaque nouvelle transaction qui met à jour une donnée en créé une nouvelle version. Les transactions qui ne font que des lectures sur une donnée vont lire la version validée la plus récente. Lorsque deux transactions concurrentes mettent à jour la même valeur, la première qui termine valide, la deuxième abandonne. Grâce à cette fonctionnalité, les utilisateurs peuvent accéder à la dernière valeur validée tout au long de la transaction au moyen d'une vue temporaire cohérente. Cette fonctionnalité permet d'augmenter la concurrence des accès tout en réduisant les cas de blocage, car il n'y a plus de conflit entre les lectures et les écritures sur une même donnée.

Afin de bénéficier des avantages de ce mode d'isolation [LKPMJP05, PMJPKA05, WK05, PA04], un nouveau critère de cohérence basé sur l'isolation *Snapshot* a été défini par Lin et al. [LKPMJP05], il est appelé *Isolation Snapshot-1-copie* (1-copy-SI).

Définition 1.1. (SI-Exécution) Soit T un ensemble de transactions validées, où chaque transaction T_i est définie par son jeu de lecture RS_i et son jeu d'écriture WS_i. Une SI-Exécution S sur T est une séquence d'opérations $o \in b, c$ (où b indique quand RS_i est exécuté et c indique quand WS_i est exécuté). Soit $(o_i < o_j) \in S$ montre que o_i intervient avant o_j dans S. S vérifie les propriétés suivantes :
(i) Pour chaque $T_i \in$ T : $(b_i < c_i) \in S$;
(ii) Si $(b_i < c_j < c_i) \in S$, alors $WS_i \cap WS_j = \emptyset$.

Pour mieux comprendre ce modèle, il faut dans un premier temps définir l'ordonnancement *SI-Exécution* sur un système centralisé fournissant une isolation *Snapshot* (voir Définition 1.1). La propriété *(i)* indique que les lectures ont lieu avant les écritures (et ceci uniquement pour des raisons de simplification). Enfin, la propriété *(ii)* indique que si deux transactions sont exécutées en concurrence et sont validées alors elles n'étaient pas conflictuelles.

Définition 1.2. (SI-Équivalence) Soit S^1 et S^2 deux SI-Exécution sur le même jeu de transactions T. S^1 et S^2 sont équivalents si pour tout $T_i, T_j \in T$ les propriétés suivantes sont vérifiées :
(i) Si $WS_i \cap WS_j \neq \emptyset$ alors $(c_i < c_j) \in S^1 \Leftrightarrow (c_i < c_j) \in S^2$;
(ii) Si $WS_i \cap RS_j \neq \emptyset$ alors $(c_i < b_j) \in S^1 \Leftrightarrow (c_i < b_j) \in S^2$.

Puis, à l'instar du modèle de cohérence forte, un critère d'équivalence *SI-Équivalence* est défini pour comparer deux exécutions *SI-Exécution* (voir Définition 1.2). La propriété *(i)* indique que deux transactions conflictuelles seront exécutées dans le même ordre sur deux SI-Exécutions équivalentes. La propriété *(ii)* requiert que deux exécutions aient la même causalité entre les lectures et les écritures, de sorte qu'une transaction lise la même valeur sur les deux exécutions.

Ensuite, afin d'utiliser ce modèle dans un contexte distribué, chaque base de données produit sa propre *SI-Exécution* sur les transactions exécutées en local. De plus, le système de réplication distribué suit une approche ROWA et chaque transaction de mise à jour a une copie locale qui exécute toutes ces opérations. La transaction est locale sur cette copie, et distante sur les autres

copies. Seules les opérations d'écriture sont exécutées sur les copies distantes. Par conséquent, toutes les copies exécutent le même jeu de transactions de mises à jour, mais une transaction de mise à jour T_i a un jeu de lectures RS_i seulement sur sa copie locale alors qu'elle a le même jeu d'écritures WS_i sur toutes les copies. Par contre, les transactions en lecture-seule existent seulement sur les copies locales. Ces hypothèses sont formalisées à travers une fonction de mappage ROWA **rmap** qui prend en entrée un ensemble de transactions T et un ensemble de copies R et retourne un ensemble de transactions T' qui seront validées sur R.

Définition 1.3. (SI-1-copie) Soit R un ensemble de copies suivant une approche ROWA. Soit T un ensemble de transactions soumises pour exécution pour lesquelles $T_i \in T$ sont validées sur sa copie locale. Soit S^k une *SI-Exécution* sur l'ensemble de transactions T^k sur la copie $R^k \in R$. Nous pouvons dire que R garantit la cohérence SI-1-copie si les propriétés suivantes sont vérifiées :
(i) Il y a une fonction de mappage ROWA, rmap, tel que $\cup_k T^k = rmap(T, R)$;
(ii) Il y a une SI-Exécution S sur T tel que pour chaque S^k et $T_i^k, T_j^k \in T^k$ soient des transformations de $T_i, T_j \in T$:
(a) Si $WS_i^k \cap WS_j^k \neq \emptyset$ alors $(c_i^k < c_j^k) \in S^k \Leftrightarrow (c_i < c_j) \in S$;
(b) Si $WS_i^k \cap RS_j^k \neq \emptyset$ alors $(c_i^k < b_j^k) \in S^k \Leftrightarrow (c_i < b_j) \in S$.

Finalement, il ne reste plus qu'à définir le critère de cohérence SI-1-copie (voir Définition 1.3). La propriété *(i)* garantit que l'ensemble de transactions validées est un sous-ensemble de la fonction de mappage ROWA des transactions soumises. Une transaction de mises à jour doit donc être validée sur tous les noeuds ou sur aucun et une transaction lecture-seule doit être validée seulement sur sa copie locale. La propriété *(ii)* requiert une SI-Exécution S produite sur l'ensemble des transactions validées par un ordonnanceur centralisé. Chaque exécution locale S^k doit être équivalente à cette exécution S.

Nous avons pu voir que la cohérence *Snapshot* permet d'obtenir une cohérence équivalente à la cohérence forte : les exécutions **snapshot** concurrentes sont correctes si et seulement si leur résultat est équivalent à celui d'une exécution *snapshot* sérielle. Cependant, ce modèle utilise l'isolation *snapshot* qui est plus souple et qui entraîne moins de conflits entre les transactions (pas de conflit entre les transactions de lectures et d'écritures), il améliore le débit du système en augmentant la concurrence entre les transactions. Cependant, ce modèle n'est applicable qu'aux SGBD supportant ce mode d'isolation, ce qui élimine tous les anciens SGBD.

1.4 Architecture des grappes

Dans cette section, nous présentons deux propriétés des grappes de PC importantes pour le choix d'un algorithme de réplication. Nous décrivons dans un premier temps les architectures possibles dans une grappe de PC. Le choix d'une architecture permet de définir les liens entres les noeuds du système, et ainsi de définir son autonomie. Le choix de l'architecture repose également sur le type d'applications utilisé sur la grappe en terme de concurrence entre les

transactions et en pourcentage d'écritures. Nous montrons enfin en quoi la visibilité du SGBD doit être prise en compte lors du choix d'un algorithme de réplication.

1.4.1 Les choix d'architectures en grappe

Les architectures des systèmes de données distribuées peuvent être réparties en quatre classes [OV99] : les bases de données centrales, les architectures à mémoire partagée, les architectures à disques partagés (*Shared disks*) et les architectures sans partage (*Shared nothing*). Cependant, les grappes étant des systèmes faiblement couplés, seules les deux dernières architectures sont utilisables dans une grappe. En effet, le partage des composants d'un noeud, comme la mémoire, est plus coûteux que le gain que l'on en retire. Voici donc les deux architectures exploitables dans les grappes de PC :

- Architecture à disques partagés : Les noeuds possèdent chacun leurs processeurs et leurs mémoires mais ils partagent une ou plusieurs ressources de stockage. Tous les noeuds ont alors accès aux disques de stockage. La répartition des clients est faite uniquement en fonction de l'équilibrage de charges du système, ce rôle étant dévolu généralement à un noeud. Cette architecture autorise donc un bon équilibrage de charge du fait que la charge d'un seul noeud est à prendre en compte. L'un des inconvénients de l'architecture à disques partagés est que le réseau est sollicité dès qu'un noeud veut accéder aux données. Les performances dépendent donc fortement des capacités du réseau même si cela est atténué par le fait que les disques sont connectés à travers un réseau dédié. Les noeuds disques sont donc un goulot d'étranglement qui peut être atténué par la duplication des disques. Cette solution s'avère limitée du fait que les disques partagés constituent l'élément le plus coûteux du système. Elle est de surcroît uniquement efficace pour les larges bases de données en lecture seule et pour les bases de données où il n'y a pas de partages. Quand un noeud souhaite accéder à une donnée d'un disque partagé pour écrire dessus, le contrôle de concurrence est un problème.

- Architecture sans partage : Ici chaque noeud est autonome, il possède son propre processeur, sa mémoire et son disque dur. Chaque noeud possède alors ses propres données et la répartition des clients doit être faite en priorité selon la ressource à laquelle le noeud veut accéder. L'équilibrage de charge est donc étroitement lié à la répartition des données sur le système. En revanche, le passage à l'échelle en nombre de processeurs est très bon par rapport à une architecture à disques partagés où les disques deviennent les goulots d'étranglement. Si cette architecture respecte l'autonomie des noeuds, elle implique un certain nombre de contraintes. Les noeuds doivent en effet échanger de nombreux messages pour connaître leurs états, charges ou fraîcheurs. Enfin, la détection des pannes est beaucoup plus difficile à mettre en oeuvre sur ce type d'architecture.

Ces deux architectures sont très proches l'une de l'autre. Une architecture à disques partagés peut simuler une architecture sans partage en partitionnant les disques. Quant à l'architecture sans partage, celle-ci peut simuler de manière logicielle une architecture à disques partagés au

moyen du d'entrées-sorties distantes. Dans les deux cas le coût du maintien de copies répliquées est important, surtout si les mises à jours sont fréquentes. Cependant, les architectures à disques partagés sont mieux adaptées aux applications en lecture seule. Celles-ci permettent notamment un équilibrage plus simple de la charge du système du fait que tous les noeuds ont accès à toutes les données.

1.4.2 Visibilité du SGBD

Un autre point important dans l'architecture d'une grappe est le niveau d'intrusion de la couche de réplication dans le SGBD. On peut traduire ce niveau d'intrusion par la visibilité que possède l'algorithme de réplication sur le SGBD. En effet, la couche de réplication considère le SGBD comme une boîte plus ou moins opaque à laquelle elle se connecte.

- **Boîte blanche :** Tous les mécanismes de réplication sont implémentés directement dans le SGBD. L'avantage est que les développeurs peuvent accéder directement à toutes les fonctions internes du SGBD. L'implémentation est donc beaucoup plus efficace en terme de performances. De plus, le mécanisme de réplication est totalement transparent pour l'utilisateur du SGBD. Cependant, la modification du noyau d'un SGBD n'est pas possible dans tous les cas. Par exemple, les systèmes commerciaux ne fournissent pas de codes sources, et dans les cas des SGBD libres comme PostgreSQL ou MySQL, l'investissement pour comprendre et modifier le SGBD est très important, d'autant plus que chaque modification n'est pas portable d'un SGBD à un autre.

- **Boîte noire :** Avec cette approche, le SGBD est considéré comme une boîte noire, et l'on est incapable de voir quels sont les mécanismes internes utilisés par le SGBD. Il doit être possible de remplacer un SGBD par un autre sans avoir à modifier les mécanismes de mise en oeuvre de la réplication. Bien que cette approche soit simple et élégante, elle a cependant plusieurs désavantages. Premièrement, le contrôle de concurrence devient difficile à réaliser car on ne sait pas quelles sont les données à verrouiller avant de soumettre une transaction au SGBD. Deuxièmement, les interfaces de communication entre le module de réplication et le SGBD doivent être standards. Les SGBD ne possédant pas ces interfaces ne pourront pas être utilisé comme boîte noire. Finalement, en ajoutant une couche de communication entre le SGBD et le module de réplication, on augmente le temps de routage des requêtes et on dégrade donc les performances du système.

- **Boîte grise :** La réplication est implémentée dans un logiciel de médiation au dessus du SGBD. Un SGBD est considéré comme une boîte grise s'il fournit les deux services minimums nécessaires à la réplication : capturer le jeu d'écritures résultant d'une transaction et appliquer ce jeu d'écritures sur le SGBD. Cette solution est aussi simple que le mode boîte noire et elle passe à l'échelle même lorsque le taux d'écritures des transactions est élevé. En revanche, le SGBD doit fournir les deux services de lecture et de mise en place des jeux d'écritures pour être utilisé avec un service de médiation pour la réplication.

Le choix d'une approche dépend surtout du type de réplication que l'on souhaite effectuer. L'approche boîte blanche est plus adaptée pour un module réplication qui recherche les performances et surtout dans une grappe homogène ou tous les noeuds utilisent le même SGBD. Cependant, dans le cas où l'on ne veut pas faire d'hypothèse sur le type de SGBD, on utilise une approche boîte noire. Enfin, l'approche boîte grise semble être un compromis idéal qui supprime les défauts de l'approche boîte noire sans être aussi intrusive que l'approche boîte blanche.

1.5 Exploitation du réseau de communication

La communication de groupe est un moyen de fournir une communication multi-point à multi-point en organisant les processus en groupes. Un groupe est un ensemble de processus qui ont des intérêts communs et se sont donc abonnés à un groupe. Par exemple, un groupe peut consister en un ensemble de participants à une conférence multimédia. Un autre groupe peut regrouper les abonnés à un système de diffusion d'informations sur une équipe sportive. Chaque groupe est associé à un nom logique et les processus communiquent avec les autres membres du groupe en envoyant un message destiné à un groupe nommé. Le *Service de Communication de Groupe* (SCG) délivre alors le message aux membres du groupe. Un SCG fournit trois services principaux [CKV01] :

- Tolérance aux pannes des systèmes distribués,
- Service d'abonnement : maintenance d'une liste des processus actifs et connectés dans un groupe (vue d'un groupe),
- Service de diffusion : envoi de messages à la vue d'un groupe.

Si la communication de groupe est très difficile à mettre en place et à maintenir sur un réseau dynamique comme internet, elle devient très efficace dans un environnement stable comme une grappe de PC où il y a peu d'évolution de la topologie (coupures de liens, nouveaux participants...). Pour offrir ces services, le principal problème d'un SCG est de fournir un accord sur la vue du groupe et sur délivrance des messages dans des environnements enclins aux pannes. Il doit pour cela garantir deux propriétés : la sûreté et la vivacité du système.

1.5.1 Propriété de sûreté

La sûreté du système se divise en quatre parties : les propriétés de sûreté du service d'abonnement, les propriétés de sûreté du service de diffusion fiable, les propriétés d'un message sûr et enfin les propriétés d'ordonnancement et de fiabilité du service de diffusion.

Un **service d'abonnement** est une partie vitale d'un groupe de communication utilisant des vues. Une vue est un état du SCG à un moment donné et elle contient la liste des noeuds, la liste des groupes et les messages échangés. Le but du service d'abonnement est de maintenir une liste des processus actifs et connectés. La liste peut se modifier quand un membre rejoint le groupe ou après un départ volontaire ou encore lors de la panne de l'un des membres. Lorsque

cette liste change, le service d'abonnement rapporte ce changement aux membres en installant une nouvelle vue du groupe. Il tente d'installer la même vue pour tous les membres connectés. Pour s'en assurer, on vérifie les trois propriétés suivantes :

- Un processus est toujours membre des vues qu'il installe (un processus doit pouvoir communiquer avec les autres et donc au moins avec lui-même),
- Un processus n'installe pas deux fois la même vue et si deux processus installent les deux mêmes vues, ils les installent dans le même ordre,
- Chaque événement doit être associé à une vue particulière. Le service d'abonnement doit donc initialiser la vue au démarrage du système et lors d'une reprise après panne.

Les SCG fournissent différents types de **services de diffusion** avec différentes garanties de livraisons dans l'ordre des messages. Pour garantir que les messages soient bien reçus par tous les membres de la vue d'un groupe, il faut s'assurer de plusieurs points. Tout d'abord, aucun message ne doit être généré spontanément par le système (afin de garantir la causalité des messages avec des horloges comme le définit Lamport [Lam78]). Ensuite, aucun message ne doit être dupliqué par le système, chaque processus ne doit recevoir qu'un message au maximum. Finalement, la troisième propriété basique du service de diffusion garantit que deux processus qui reçoivent un même message, le reçoivent dans la même vue. Cette dernière propriété permet d'empêcher les inter-blocages entre les processus.

Les applications distribuées requièrent souvent une sémantique *"tout ou rien"*, ce qui signifie que tous les processus reçoivent le message ou aucun d'entre eux. Cependant, cette sémantique n'est pas possible dans les systèmes distribués où les messages peuvent être perdus. Comme approximation de la sémantique *"tout ou rien"*, le concept des **messages sûrs** est introduit [MAMSA94]. Un message sûr n'est reçu par les processus que si le message est stable, c'est-à-dire s'il a été reçu par tous les membres de la vue courante du groupe. Une fois que le message est stable, il est soumis au processus par le SCG à moins que le processus ne tombe en panne. Il faut noter que dans cette sémantique le processus peut tomber en panne à n'importe quel moment de l'exécution, le système ne peut donc garantir que le message a bien été délivré à un processus. Une optimisation permet de soumettre le message au processus sans délai et d'avertir l'application lorsque le message devient sûr.

Les SCG fournissent différents services de diffusion des messages, nous pouvons diviser ces services en deux types de garanties : les garanties d'**ordonnancements** et les garanties de **fiabilités**. Les garanties d'ordonnancements concernent l'ordre dans lequel les messages vont être reçus par le système. Les garanties de fiabilités étendent quant à elles les propriétés d'ordonnancement précédentes en proposant la diffusion de messages sûrs. Traditionnellement, les services de fiabilité proposés par les SCG sont les suivants [FLS97] :

- Non fiable : Le message peut être perdu et ne sera pas récupéré par le SCG.
- Fiable : Le message sera délivré à tous les processus membres du groupe auxquels il a été envoyé. Le message sera récupéré par le SCG en cas de perte réseau.
- Sûr : Le message sera délivré seulement si tous les processus membres du groupe auquel il a été envoyé ont reçu le message. Le message sera récupéré par le SCG en cas de perte

réseau.

Finalement, les services d'ordonnancement sont [ADMSM94, WS95] :
- Aucun : Tout autre message envoyé avec la même propriété peut arriver avant ou après le message envoyé.
- FIFO par émetteur : Tous les messages envoyés par un même processus sont reçus dans le même ordre que leur ordre d'envoi. Ce qui n'est pas garanti si les deux messages sont envoyés par deux processus différents.
- Causal : Ce service étend l'ordre FIFO par émetteur en garantissant que deux messages ayant un ordre causal, par exemple le message *m* et sa réponse *m'* (donc *m'* dépend de *m*), soient reçus dans le même ordre sur tous les noeuds.
- Ordre Total : Tous les messages d'un même groupe sont reçus dans le même ordre par tous les processus.

Il faut noter que les deux services (d'ordonnancement et de fiabilité) ne sont pas toujours dépendants. Par exemple, l'ordre *FIFO par émetteur* requiert que tous les messages d'un même processus soient reçus dans le même ordre par tous les processus du groupe alors que l'ordre *FIFO par émetteur fiable* interdit en plus les pertes de messages dans l'ordre FIFO.

1.5.2 Propriété de vivacité

Nous allons maintenant présenter les propriétés de vivacité d'un SCG. La vivacité est la propriété de garantir que des messages pourront être diffusés dans le futur. Ceci est un complément important à la propriété de sûreté. Sans les propriétés de vivacité, on peut réaliser des implémentations qui ne font rien même si elles satisfont les propriétés de sûreté. La difficulté étant de spécifier des propriétés de vivacité suffisamment fiables pour être implémentées et suffisamment fortes pour être efficaces.

Pour spécifier les propriétés de vivacité, il faudrait observer le comportement d'un SCG idéal (qui n'existe pas). Idéalement, le service d'abonnement doit être précis (chaque vue envoyée représente parfaitement le système) et le service de diffusion doit être correct (tous les processus d'une vue reçoivent un message diffusé). Cependant, l'instabilité du réseau ne permet pas un tel comportement et la vivacité d'un SCG dépend du comportement du réseau. La solution est de spécifier les propriétés de vivacité comme conditionnelles [KD00, KK00].

Il faut faire trois suppositions pour décrire les propriétés de vivacité. *(i)* Le réseau est vivant : si deux processus sont disponibles et que la connexion entre eux est active, alors tous les messages envoyés entre ces deux processus seront éventuellement reçus. *(ii)* Il existe un ensemble de processus stables qui représentent un groupe de processus disponibles, dont les connections entre eux sont toutes actives et où toutes les connections vers les processus qui ne sont pas dans ce groupe ne sont pas actives. *(iii)* Il existe un détecteur de pannes éventuellement parfait [CT96] : après une certaine période le détecteur ne fait plus d'erreurs et connaît précisément les processus vivants dans le système.

Nous pouvons alors définir trois propriétés de vivacité : *Précision d'abonnement*, *vivacité de la diffusion* et l'*indication de vivacité sûre*. Ces propriétés sont conditionnelles et nécessitent qu'un détecteur de pannes éventuellement parfait détecte un ensemble de processus stables S qui composent la vue V :

- *Précision d'abonnement* : Tous les processus p de S ont V comme leur dernière vue.
- *Vivacité de la diffusion* : Tous les messages envoyés à V sont reçus par les processus de S.
- *Indication de vivacité sûre* : Tous les messages envoyés à V sont déclarés comme sûrs par les processus de S (cette propriété n'est bien sûr valable que si le message est envoyé dans un mode sûr).

Il faut noter que des propriétés supplémentaires de vivacité [CKV01] existent pour les cas où il n'y a pas un ensemble de composants stables, mais nous ne les détaillerons pas ici.

Les SCG offrent de nombreux services utiles pour les algorithmes de réplication. La gestion de groupe permet de regrouper les noeuds par intérêt commun (tous les noeuds qui ont la même copie) et a diffusion sûre garantit qu'un message est délivré à tous les noeuds disponibles.De plus, le choix d'un ordonnancement lors de la diffusion des messages permet d'obtenir un ordre total pour l'exécution des transactions sur tous les noeuds.

1.6 La réplication dans les grappes

Il existe de nombreux algorithmes de réplications des données pour les systèmes distribués utilisant des techniques éprouvées comme le protocole *ROWA* [BHG87]. Cependant, si ces algorithmes fonctionnent parfaitement dans des grappes, les réseaux rapides et fiables peuvent être beaucoup mieux exploités en utilisant un Service de Communication de Groupe (SCG).

Nous présentons donc trois algorithmes de réplications asynchrones pessimistes dont l'objectif est d'empêcher les conflits en ordonnant les transactions afin de les exécuter dans le même ordre sur tous les noeuds. Ceci revient donc à avoir un ordre total dans la diffusion des transactions. Pour se faire, ils exploitent les SCG afin d'obtenir une cohérence forte des données, ils sont représentatifs des algorithmes de réplication pour grappes.

1.6.1 Graphe Dirigé Acyclique

Dans cette section, nous présentons un algorithme qui utilise des graphes dirigés acycliques pour garantir une cohérence forte des données entre les noeuds. L'algorithme *DAG(WT)* [BKR+99], *Directed Acyclic Graph (Without Timestamp)* définit un noeud maître pour chaque copie. Puis, d'après le graphe de dépendance entre les noeuds, il construit un arbre avec la propriété suivante : si le site S_i est un fils du site S_j dans le graphe de dépendance, alors S_i sera un descendant de S_j dans l'arbre. La figure 1.7 montre un graphe de dépendance (DAG) et trois des arbres qui peuvent en être issus (CHAIN, FOREST F1, FOREST F2).

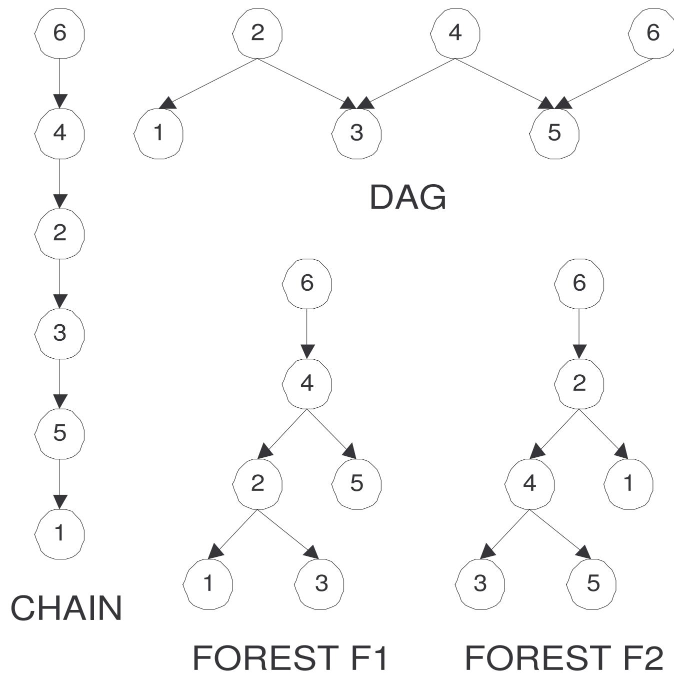


Figure 1.7 – Graphe dirigé acyclique - Graphe de dépendance et arbres associés

La figure 1 décrit l'algorithme DAG(WT). Lors d'une mise à jour (toujours sur le noeud maître), la transaction est exécutée, la réponse est renvoyée au client puis les jeux d'écritures de la transaction sont transmis aux noeuds fils de l'arbre (même s'ils ne sont pas concernés par la transaction, leurs fils peuvent l'être) qui l'exécutent à leurs tours avant de les diffuser à ses fils. Les jeux d'écritures sont validés et renvoyés aux noeuds fils dans l'ordre dans lequel ils ont été reçus.

La figure 1.8a montre un exemple de schéma de dépendance de réplique avec une exécution incorrecte des transactions. T_1 est exécutée sur S_1 et T_2 sur S_2, S_3 reçoit T_1 avant T_2 et S_4 reçoit T_2 avant T_1. Si T_1 et T_2 écrivent sur les mêmes données, on se retrouve alors avec une

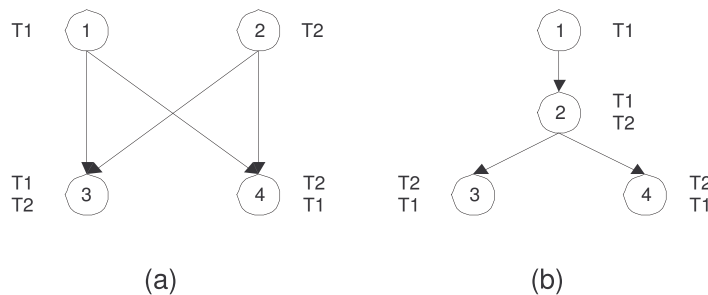


Figure 1.8 – Graphe dirigé acyclique - Exemple d'exécution

Algorithme 1 – Algorithme DAG(WT)

Entrées:
G : l'arbre issu du graphe de dépendance
Variables:
T : une transaction d'un client C
E_T : le jeu d'écriture de la transaction T
$liste_E$: liste des jeux d'écriture reçus

1: **début**
2: **boucler**
3: **sur** la réception de la transaction T depuis un client C **faire**
4: Exécuter T
5: Retourner la réponse au client
6: Extraire le jeu d'écriture E_T issu de T
7: Diffuser E_T à tous ses fils d'après G
8: **fin sur**
9: **sur** la réception du jeu d'écriture E_T depuis un autre noeud **faire**
10: Ajouter E_T à la fin de $liste_E$
11: **fin sur**
12: **tant que** $liste_E \neq$ vide **faire**
13: Extraire E_T de la tête de $liste_E$
14: Appliquer E_T
15: Diffuser E_T à tous ses fils d'après G
16: **fin tant que**
17: **fin boucler**
18: **fin**

incohérence.

La figure 1.8b montre un arbre associé au graphe de la figure 1.8a. T_1 s'exécute sur S_1 et T_2 sur S_2. T_1 est envoyé à S_2 par S_1, et à sa réception, elle attend la fin de T_2 avant d'être traitée. Une fois T_2 terminée, elle est envoyée à S_3 et S_4, T_1 est alors diffusée aux autres noeuds. Ainsi sur S_3 comme sur S_4, les transactions s'exécutent dans le même ordre, la propriété de sérialisabilité-1-copie est donc assurée.

Cet algorithme n'est pas spécifique aux grappes, le choix de s'appuyer sur des graphes dirigés acycliques avec l'emploi de copies primaire le rend mieux adapté aux applications d'entrepôt de données. Il offre une cohérence forte grâce à une exécution déterministe des transactions, éliminant ainsi tout abandon. Cependant, l'algorithme possède quelques inconvénients. Tout d'abord sur le choix de l'arbre, il n'existe pas un arbre optimal pour un graphe de dépendance. La qualité d'un arbre dépendra des transactions qui seront exécutées dessus. Il est donc impossible de choisir un arbre a priori. De plus, pour qu'une transaction soit exécutée complètement, elle doit parcourir l'intégralité de l'arbre, elle doit donc être exécutée et diffusée autant de fois qu'il y a de noeuds dans le chemin de l'arbre. Cette dernière caractéristique rend le passage à l'échelle impossible pour des graphes possédant beaucoup de noeuds.

1.6.2 Diffusion optimiste

Nous présentons dans cette section l'algorithme NODO (NOn-Disjoint conflict classes and Optimistic multicast) [KA00a, KA00b, PMJPKA00, JPPMKA02] qui s'appuie sur le service de diffusion des messages en ordre total sur tous les noeuds pour garantir l'ordonnancement des transactions.

Pour gérer le contrôle de concurrence, l'algorithme se base sur la division des copies en classes de conflits. Une classe de conflit représente une copie ou une partie d'une copie. On détermine ainsi que deux transactions accédant à la même classe de conflit ont une grande probabilité de conflit. À l'inverse, deux transactions qui n'accèdent pas aux mêmes classes peuvent s'exécuter en parallèle. L'algorithme ne permettant d'associer à une transaction qu'une seule classe de conflit, une abstraction supplémentaire regroupe des classes de conflits en groupe de classes de conflit. On peut alors choisir d'associer à des transactions des classes de conflits ou des groupes de classes de conflits. Les classes et les groupes de classes de conflits doivent être connus à l'avance. De plus, chaque classe de conflit a un seul noeud maître. Les transactions ne pouvant être soumises qu'au noeud maître de la classe de conflit qu'ils touchent. Il faut donc connaître la classe de conflit à laquelle accède une transaction avant son exécution.

La Figure 2 montre une version simplifiée de l'algorithme NODO. Au moment où le noeud maître reçoit une transaction, il la diffuse à tous les noeuds (y compris lui-même). Ainsi, chaque noeud possède une file associée à un groupe de classes de conflit. La diffusion des transactions dans le système se fait en deux étapes. La première diffusion est une diffusion optimiste (*opt-deliver*), les messages sont diffusés sans garantie d'ordonnancement. À la deuxième diffusion, ils ont une garantie d'ordre total (*to-deliver*). Le système doit simplement garantir qu'un message *opt-deliver* ne précédera jamais un message *to-deliver*.

Quand une transaction est soumise au noeud maître par un client, elle n'est exécutée que sur

la copie ombre de ce noeud. Une transaction est dite locale quand elle se trouve sur son noeud d'origine (le noeud maître). Ainsi, annuler une transaction requiert simplement l'invalidation de la copie ombre. Elle est ensuite diffusée sur tous les noeuds. Quand une copie reçoit le message *opt-deliver*, la transaction est placée dans la ou les files correspondantes aux classes de conflits que touchent la transaction. Dès que ce message est en tête de l'une de ces files et que la transaction est locale, la transaction est exécutée.

Algorithme 2 – Algorithme NODO

Entrées:
T : une transaction d'un client
$C(T)$: les classes de conflits touchées par la transaction T

Variables:
T : une transaction d'un client
CQ_1, CQ_2,..., CQ_n : files d'attentes des classes de conflit C_1, C_2, ..., C_n

1: **début**
2: **boucler**
3: **sur** la réception de la transaction T d'un client **faire**
4: Diffuser le message opt-deliver(T)
5: Diffuser le message to-deliver(T)
6: **fin sur**
7: **sur** la réception d'un message opt-deliver pour T **faire**
8: **pour** Pour toutes les classes de conflits C_i de C_T **faire**
9: Ajouter T en fin de CQ_i
10: **fin pour**
11: **si** T est locale **et** T est en tête de l'une des files CQ_1, CQ_2,..., CQ_n **alors**
12: Exécuter T sur la copie ombre
13: Marquer T comme *exécutée*
14: **fin si**
15: **fin sur**
16: **sur** la fin de l'exécution de T **faire**
17: **si** T est locale **et** T est *to-deliver* **alors**
18: Valider la copie ombre
19: Diffuser le jeu d'écriture WS_T
20: **fin si**
21: **fin sur**
22: **sur** la réception d'un jeu d'écriture WS_T **faire**
23: **si** T n'est pas locale **alors**
24: Attendre que T soit *to-deliver*
25: **pour** Pour toutes les classes de conflits C_i de $C(T)$ **faire**

26: **si** Premier(CQ_i) = T **alors**
27: Appliquer WS_T correspondant à C_i
28: Enlever T de CQ_i
29: **fin si**
30: **fin pour**
31: **sinon**
32: Enlever T de toutes les CQ_i de C_T
33: **fin si**
34: **fin sur**
35: **sur** la réception d'un message to-deliver pour T **faire**
36: Marquer T comme *to-deliver*
37: **si** T est locale **et** T est *exécutée* **alors**
38: Valider la copie ombre
39: Diffuser le jeu d'écriture WS_T
40: **fin si**
41: **si** T n'est pas locale **alors**
42: **pour** Pour toutes les classes de conflits C_i de $C(T)$ **faire**
43: **si** Premier(CQ_i) = T_j **et** T_j est locale **alors**
44: Abandonner T_j
45: **fin si**
46: Replacer T_j dans CQ_i
47: **fin pour**
48: **fin si**
49: **fin sur**
50: **fin boucler**
51: **fin**

Une fois que la transaction est *to-deliver*, le noeud vérifie que les exécutions se sont effectuées dans le bon ordre et valide la transaction. Si l'ordre total n'est pas respecté, on se retrouve avec plusieurs cas :

- Si aucun conflit n'est entraîné par l'absence d'agrément, l'ordre total peut-être ignoré.
- Si aucune transaction locale n'est impliquée, la transaction peut être tout simplement ré-ordonnancée et replacée dans sa file avant les transactions déjà *opt-deliver* mais pas encore *to-deliver*. Ainsi, les transactions suivront l'ordre total.
- Si une transaction locale est impliquée, la procédure est la même que dans le cas précédent mais les transactions locales (celles qui ne suivent pas l'ordre total) doivent être annulées et ré-ordonnancées.

Illustrons l'algorithme avec un exemple, on considère deux classes de conflits C_x et C_y et deux noeuds N et N'. N est le maître des groupes de classe de conflit $\{C_x\}$ et $\{C_x, C_y\}$. N' est maître de $\{C_y\}$. Puis on considère trois transactions $T_1 = \{C_x, C_y\}, T_2 = \{C_y\}$ et $T_3 = \{C_x\}$.

Sur N :	Sur N' :
$CQ_x = T_1, T_3$	$CQ_x = T_3, T_1$
$CQ_y = T_1, T_2$	$CQ_y = T_2, T_1$

Figure 1.9 – État des files d'exécution sur N et N'

T_1 et T_3 sont donc locales à N et T_2 est locale à N'. L'ordre de la tentative d'exécution sur N est T_1, T_2, T_3 et sur N', T_2, T_3, T_1. L'ordre définitif d'exécution est sur les deux noeuds T_1, T_2, T_3. Quand toutes les transactions sont *opt-delivered*, le contenu des files sur chaque noeud est représenté sur la Figure 1.6.2.

Sur N, l'exécution de T_1 peut commencer ses opérations sur C_x, et C_y car elle se trouve en tête des deux files. Quand T_1 est *to-delivered*, on compare les ordres d'exécution. Dans ce cas T_1 a été exécuté dans le bon ordre. T_1 peut-être validé. N envoie un message de validation aux autres noeuds et retire T_1 de ses files. Même procédé pour T_3, y compris si T_3 doit être exécutée avant T_2, il n'y a pas de conflits entre les transactions, on peut donc avoir un ordre différent de l'ordre définitif. Sur N', T_2 commence son exécution car elle est local et en tête de sa file. Puis quand T_1 est *to-delivered* (par N), N' réalise que T_2 a été exécutée dans le mauvais ordre et l'annule donc avant de la remettre à la fin de la file. T_1 est alors placée en tête des deux files, mais comme elle est distante à N', aucune transaction n'est exécutée. Quand le message de validation de T_1 arrive, T_1 est exécutée, validée et retirée des files. T_2 peut alors s'exécuter, puis T_3.

Bien que ce protocole garantisse la sérialisabilité-1-copie, la construction d'une classe de conflit n'est pas claire (pas de règle) et les classes de conflit touchées par une transaction doivent être connues à l'avance. De plus, avec un seul noeud maître par groupe de classe de conflit, l'algorithme se rapproche plus d'un mode de mise à jour Copie Primaire. Finalement, les transactions de lecture sont favorisées par rapport aux écritures (selon les auteurs, le pourcentage de transaction d'écriture optimal est 20%, au-delà les performances se dégradent).

1.6.3 Utilisation du délai réseau

Un autre algorithme de réplication qui garantit la sérialisabilité-1-copie est présenté dans [PMS01, PMS99]. Seules les configurations maîtres-paresseux (*Lazy-Master*) sont supportées et doivent pouvoir être représentées par des graphes dirigés acycliques. Afin de garantir la cohérence sur les noeuds secondaires, l'algorithme utilise une communication de groupe avec une diffusion sûre FIFO par émetteur. De plus, tous les noeuds doivent pouvoir fournir une estampille globale. Finalement, l'hypothèse la plus forte implique que le temps de transmission maximal d'un message soit connu (Max) [TC94].

Comme illustré sur la figure 3, les transactions de mises à jour sont exécutées sur le noeud maître puis sont propagées à toutes les copies sous forme de transaction de rafraîchissement. À la réception de l'une de ces transactions, le noeud copie place la transaction dans une file d'attente, il y a une file d'attente par maître que possède la copie. La transaction de rafraîchissement attend alors un temps Max avant d'être élue et placée dans une file d'exécution. En attendant un temps Max après son départ, on s'assure alors qu'aucun message n'a été émis auparavant et

transite encore sur le réseau (le réseau est fiable et un message met au maximum un temps Max pour arriver à sa destination). Le moment exact où le message sera élu pour exécution est donc : $C + Max + \epsilon$. Où C est l'estampille globale (heure d'exécution de la transaction sur la copie primaire), Max le temps maximal d'arrivée d'un message d'un noeud à l'autre et ϵ le temps de traitement d'un message.

Algorithme 3 – Algorithme Lazy-Master

Entrées:
 T : une transaction d'un client C
Variables:
 WS_T : le jeu d'écriture de la transaction T
 $q_i, \ldots, q_j$: files d'attente pour les noeud maîtres des copies $i, \ldots, j$
 $timer$: compte à rebours

1: **début**
2: **boucler**
3: **sur** la réception de la transaction T depuis un client C **faire**
4: Exécuter T
5: Retourner la réponse au client
6: Extraire le jeu d'écriture WS_T issu de T
7: Diffuser WS_T
8: **fin sur**
9: **sur** la réception du jeu d'écriture E_T depuis un noeud i **faire**
10: Ajouter WS_T à q_i
11: $WS_T \leftarrow$ jeu d'écriture avec l'estampille minimum parmi ceux en tête de $q_i, \ldots, q_j$
12: timer $\leftarrow$ (estampille(WS_T) + Max + ϵ) - temps-local
13: **fin sur**
14: **sur** $timer$ expire **faire**
15: Appliquer WS_T
16: $WS_T \leftarrow$ jeu d'écriture avec l'estampille minimum parmi ceux en tête de $q_i, \ldots, q_j$
17: timer $\leftarrow$ (estampille(WS_T) + Max + ϵ) - temps-local
18: **fin sur**
19: **fin boucler**
20: **fin**

En revanche, les lectures n'attendent pas et sont exécutées directement. On peut donc lire sur une copie secondaire une donnée obsolète puisqu'elle a pu déjà être mise à jour ou est en train de l'être sur le noeud primaire. Cependant, l'algorithme assure que l'on ne lira pas sur un noeud un état qui n'a pas existé ou qui n'existera pas sur les autres noeuds.

Voici un exemple d'exécution de l'algorithme, il comprend un noeud esclave possédant deux maîtres i et j. Il a donc deux files $q(i)$ et $q(j)$. On prend Max = 10 et ϵ = 1 :

- 15:10 : Arrivée sur $q(i)$ d'un message avec comme valeur C=15:05
  $q(i)$ = {15:05}, $q(j)$ = { }
  On élit $q(i)$ avec une alarme à 15:16 (15:05 + 10 + 1)
- 15:12 : Arrivée sur $q(j)$ d'un message avec comme valeur C=15:03
  $q(i)$ = {15:05}, $q(j)$ = {15:03}
  On élit $q(j)$ avec une alarme à 15:14 (15:03 + 10 + 1)
- 15:14 : Expiration de l'alarme, le message est retiré de $q(j)$ et est placé dans la file d'exécution.
  $q(i)$ = {15:05}, $q(j)$ = { }
  On élit alors $q(i)$ avec une alarme à 15:16
- 15:16 : Expiration de l'alarme, le message est retiré de $q(i)$ et est placé dans la file d'exécution.
  $q(i)$ = { }, $q(j)$ = { }

Bien que les transactions n'aient pas été reçues dans l'ordre chronologique sur le noeud esclave, les transactions sont ré-ordonnancées dans un ordre chronologique (en fonction de leur estampille), ce qui revient à avoir un ordre total sur tous les noeuds.

Ce protocole garantit également la cohérence sérialisabilité-1-copie. Cependant, contrairement au protocole précédent, il n'y a pas d'abandons des transactions puisque l'algorithme est préventif et ne permet pas l'exécution d'une transaction avant que celle-ci ait été ordonnée. En revanche, l'algorithme attend un délai fixe de Max avant l'exécution de chaque transaction ce qui oblige cette valeur Max a être parfaitement calibrée par rapport au système si on ne veut pas que ce délai soit trop important. L'autre point faible de l'algorithme est qu'il n'autorise que les copies primaires et secondaires, de plus le placement est restreint puisque que seules les configurations représentées par des graphes acycliques dirigés sont autorisées afin de garantir la cohérence.

1.6.4 Synthèse

Dans cette section, nous comparons les trois algorithmes étudiés dans les trois sections précédentes. La table 1.1 montre les cinq critères de comparaisons.

Le premier critère est le niveau de cohérence offert par l'algorithme. Pour tous les algorithmes de réplication en grappe, la norme est la cohérence forte, ou Sérialisabilité-1-Copie (S1C).

Le deuxième critère indique les restrictions sur le placement des copies. Dans ce cas, les algorithmes Lazy-Master et DAG(WT) sont similaires, ils n'autorisent que les configurations dont les dépendances entre les copies forment des Graphes dirigés acycliques (DAG). De plus, pour chaque donnée, il ne peut y avoir qu'une seule copie primaire dans le système. L'algorithme NODO est plus particulier, car s'il autorise la mise à jour d'une même donnée sur plusieurs noeuds, un type de transaction ne pourra être envoyé que sur un noeud bien particulier. On peut donc dire que le mode de mise à jour est Copie Primaire par transaction.

Le troisième critère prend en compte la capacité de l'algorithme à passer à l'échelle. L'al-

gorithme DAG(WT) est très rapidement limité par le fait que pour mettre à jour une donnée, la transaction doit être diffusée à travers l'arbre de dépendance. Nous constatons donc que plus l'arbre est profond, plus le nombre de messages à envoyer est important. L'algorithme NODO est quant à lui limité par les deux messages qu'il doit diffuser pour rafraîchir les données, dont un message est en ordre total qui est le mode de diffusion le plus coûteux. En conséquence, plus le nombre de noeuds augmente, plus les temps de réponse du système se dégradent car le temps d'émission d'un tel message augmente. L'algorithme Lazy-Master ne souffre pas de ces défauts et offre un bon passage à l'échelle grâce en particulier à la diffusion d'un seul message pour rafraîchir les données.

Le quatrième critère indique les types d'applications pour lesquels sont adaptés les algorithmes. Parce qu'ils utilisent des configurations qui forment des graphes dirigés acycliques, les algorithmes DAG(WT) et Lazy-Master sont plus adaptés à des applications de types entrepôt de données où les dépendances entre copies primaires et secondaires sont bien exploitées. L'algorithme NODO permet un support de beaucoup plus d'applications, mais les taux d'écritures doivent rester faible (20%), faute de quoi, les performances s'effrondent.

Le dernier critère prend en compte l'intrusion de l'algorithme dans le SGBD. Les algorithmes DAG(WT) et Lazy-Master doivent être implémentés à l'intérieur du SGBD. Bien que cette solution viole l'autonomie des noeuds, elle permet de meilleures performances mais oblige aussi l'écriture d'une version de l'algorithme pour chaque base de données. Les dernières versions du protocole NODO [JPPMKA02] considèrent le SGBD comme une boîte grise. Malheureusement, aucun SGBD ne supporte encore les deux fonctions que réclame cette solution (voir Section 1.4.2).

Les objectifs de la thèse prennent en compte les faiblesses de ces algorithmes. Notre but est de mettre au point un algorithme de réplication pour les grappes qui n'offre aucune restriction sur le placement des copies dans le système et qui supporte tous les types d'applications, y compris celles qui font des écritures intensives. Notre algorithme doit donc permettre la mise à jour d'une copie sur n'importe quel noeud, ainsi que le support de la réplication partielle. Nous devons également optimiser notre algorithme afin que les performances restent correctes même lorsque les applications ont un grand taux d'écritures. Finalement, notre algorithme doit considérer le SGBD comme une boîte noire, ce qui est le meilleur moyen de ne pas violer l'autonomie des noeuds et de s'assurer que notre implémentation est compatible avec les SGBD les plus standards.

1.7 Conclusion

Nous avons présenté cinq critères de classification d'un algorithme de réplication. Les deux premiers critères sont les critères sur le mode de réplication (où, quand?). Le troisième critère indique le niveau de cohérence que peut offrir l'algorithme. Enfin, les deux derniers critères concernent plus spécifiquement le choix d'une architecture dans une grappe.

Copie Primaire / Partout (Section 1.2.3.2). Ce critère indique où peuvent être mises à jour les données : sur un seul noeud (Copie Primaire) ou sur plusieurs (Partout). Si le mode mises

	DAG(WT)	*NODO*	*Lazy-Master*
Cohérence	S1C	S1C	S1C
Placement	Copie Primaire, DAG	Copie Primaire par transaction	Copie Primaire, DAG
Scalabilité	Limitée par le nombre de messages	Limitée par le type de messages	Bonne
Applications	Entrepôt de données	Faible taux d'écritures (20%)	Entrepôt de données
Autonomie	Boîte blanche	Boîte grise	Boîte blanche

Table 1.1 – Comparaison des algorithmes existants

à jour Copie Primaire est plus simple à mettre en oeuvre, c'est aussi un goulot d'étranglement beaucoup plus sensible aux pannes. Le mode mises à jour partout n'a pas ces problèmes mais il faut garantir l'ordonnancement des transactions à la fois sur les noeuds secondaires et multi-maîtres.

Synchrone / Asynchrone (Section 1.2.3.3). Ce critère indique quand sont mises à jour les données : dans une même transaction (Synchrone) ou dans plusieurs transactions (Asynchrone). Les algorithmes synchrones ont longtemps été utilisés pour gérer la réplication. Ils sont simples à implémenter et offrent une cohérence forte. Cependant, ils ne passent pas à l'échelle du fait des protocoles de validation utilisés qui deviennent bloquant à la terminaison des transactions. Actuellement, les algorithmes asynchrones offrent le même niveau de cohérence sans bloquer les transactions.

Cohérence faible / forte (Section 1.3). Ce critère indique le niveau de cohérence des données dans le système : la divergence des données entre les noeuds peut être grande et durable (faible) ou au contraire la divergence des données est très faible et uniquement dûe au délai de rafraîchissement des données d'un noeud à l'autre. Il faut noter que la cohérence Snapshot offre les mêmes propriétés de cohérence que la cohérence forte.

Architecture (Section 1.4.1). Le choix d'une architecture permet de définir les liens entre les noeuds du système, et ainsi de définir son autonomie. Le choix de l'architecture doit être également effectué en fonction du taux de concurrence entre les transactions et du pourcentage d'écritures des applications utilisées.

Boîte Noire / Grise / Blanche (Section 1.4.2). Ce critère définit la visibilité du SGBD pour le gestionnaire de réplication. Un SGBD est considéré comme une boîte noire si le gestionnaire de réplication ne connaît aucun de ses mécanismes internes, il n'utilise que des interfaces standard pour communiquer. À l'inverse, si le SGBD est une boîte blanche alors le gestionnaire de réplication peut même être implémenté à l'intérieur du SGBD.

Finalement, l'utilisation d'un système de communication de groupes est réservée aux al-

gorithmes garantissant une cohérence forte ou *snapshot*. En effet, les services de diffusion des messages de tels systèmes servent à garantir l'exécution des transactions dans un ordre total et donc un ordre sériel comme le requiert la cohérence forte et *snapshot*.

La réplication est une bonne solution pour améliorer les performances et la disponibilité des données dans une grappe de PC.

Dans ce chapitre nous avons présenté et classifié les différentes techniques de réplication. Nous avons pris trois paramètres qui regroupent les cinq critères précédents afin de choisir les techniques à utiliser pour notre algorithme de réplication. Le premier paramètre de notre algorithme est qu'il doit garantir l'autonomie des noeuds. Le second paramètre concerne la stratégie de réplication : dans notre système une donnée doit pouvoir être mise à jour sur n'importe quel noeud et de manière asynchrone afin de pouvoir passer à l'échelle. Finalement, le dernier paramètre concerne le modèle de cohérence, nous avons décidé de nous placer sur une cohérence forte qui est l'un des objectifs de la thèse.

Autonomie des noeuds. Pour le critère de l'architecture, nous avons vu que plusieurs architectures s'offrent à nous dans une grappe de PC. Nous avons également remarqué que les deux seules qui ne violaient pas l'autonomie étaient les architectures sans partage et les architectures à disques partagés. Quant au critère sur la visibilité du SGBD, nous avons noté que la boîte blanche violait l'autonomie des noeuds car trop intrusif. La visibilité boîte blanche ou boîte grise sont deux solutions qui respectent l'autonomie même si pour utiliser une boîte grise il est nécessaire que le SGBD possède certaines fonctions spécifiques.

Stratégie de réplication. La *réplication synchrone* est traditionnellement la plus utilisée pour la réplication des données distribuées. Elle offre une cohérence forte et autorise les configurations avec une réplication totale. Cependant, l'utilisation d'un protocole de validation atomique pour garantir la cohérence rend ces algorithmes inutilisables lorsqu'on augmente le nombre de noeuds. De plus, la réplication synchrone viole l'autonomie des noeuds en modifiant le gestionnaire de transactions local du SGBD. La *réplication asynchrone* est une solution plus efficace et qui permet le passage à l'échelle mais au prix de restrictions dans le placement des données (les configurations doivent pouvoir être représentées par des graphes dirigés acycliques). De plus, l'introduction de la communication de groupe comme outil pour la réplication asynchrone (et plus particulièrement les propriétés d'ordonnancements lors de la diffusion des messages) permet de remplacer la validation atomique en exécutant les transactions en ordre total. On peut ainsi garantir la cohérence forte.
Pour le placement des copies sur les noeuds (critère Mises à jours Copies Primaires / Partout), nous avons vu que les noeuds maîtres dans le mode Copie Primaire peuvent devenir des goulots d'étranglement, surtout pour les applications à fort taux de mises à jour. Quant au mode mise à jour Partout, il permet d'augmenter le parallélisme et d'améliorer ainsi le débit.

Modèle de cohérence. Pour le critère de cohérence, le modèle de *cohérence faible* ne pouvant être utilisé sans modifications des applications, on ne peut donc pas utiliser ce niveau de cohérence dans notre cas. La *cohérence forte* et la *cohérence snapshot* offrent le même degré

de cohérence, mais en utilisant un mode d'isolation différent. Cependant, la cohérence forte est supportée par tous les SGBD alors que la cohérence *snapshot* n'est utilisable que lorsque le SGBD supporte le niveau d'isolation *snapshot*.

CHAPITRE 2

Architecture pour une grappe de bases de données

2.1 Introduction

Dans ce chapitre, nous présentons tous les éléments nécessaires à l'exploitation d'un algorithme de réplication dans une grappe. L'objectif de notre travail est d'utiliser les capacités de calculs des grappes pour l'exécution d'applications qui accèdent à des bases de données (les bases de données étant considérées comme des applications particulières). Un exemple typique d'utilisation est le contexte *Fournisseur d'Applications Hébergées* (ASP - Application Service Providers) où les applications et les bases de données de différents fournisseurs sont stockées sur une grappe.

Dans un premier temps, nous présentons l'architecture que nous avons choisie. Cette architecture a pour but de favoriser le passage d'un modèle centralisé vers un modèle pour grappe.Nous minimisons ainsi les modifications à apporter pour utiliser les applications et les SGBD avec notre architecture. Celle-ci est basée sur un modèle sans partage. Elle est composée de cinq couches indépendantes dont nous définissons brièvement le rôle. Nous nous efforçons aussi de démontrer à la fois l'indépendance des noeuds et des couches.

Nous présentons ensuite un exemple de routage d'une requête d'un client dans notre système. Nous montrons que chaque noeud peut traiter indépendamment n'importe quelle requête. Ainsi, sur certaines couches, une requête peut être transmise à un autre noeud dans deux cas : le noeud est incapable de traiter la requête ou un autre peut la traiter plus rapidement.

Finalement, nous présentons toutes les caractéristiques des noeuds nécessaires à l'exploitation de notre algorithme de réplication dans une grappe. Nous nous appuyons sur des services de communication qui exploitent les propriétés de rapidité et de fiabilité du réseau dans les grappes. Ces suppositions font que notre architecture ne peut être utilisée que sur des grappes ou sur des systèmes aux propriétés réseau équivalentes. Des systèmes pair-à-pair ou basés sur Internet ne supportent donc pas notre architecture.

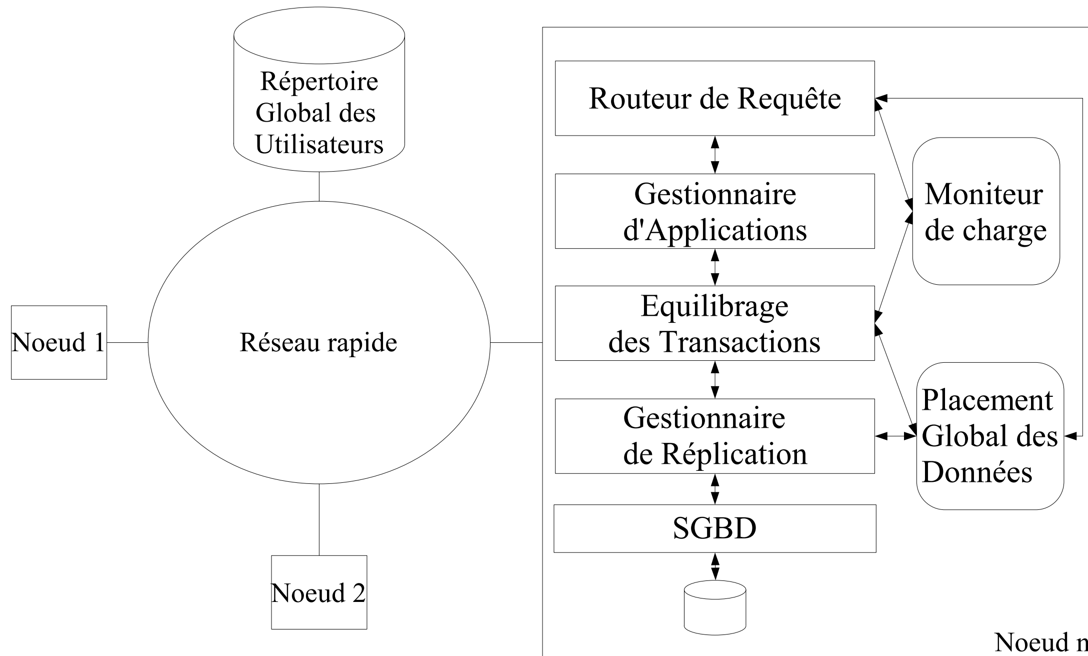


Figure 2.1 – Architecture globale

2.2 Architecture en cinq couches

Dans notre solution, nous exploitons une architecture *sans partage* où les composants ne partagent pas ni disque ni mémoire. C'est l'unique architecture qui autorise l'autonomie des noeuds sans coûts supplémentaires pour les inter-connecter (Mémoire ou disques partagés). Dans notre architecture, chaque noeud de la grappe est composé de cinq couches (voir Figure 2.1) :

- **Routeur de Requêtes : (Request Router)** Reçoit les requêtes du client et les diffuse au noeud le mieux adapté.
- **Gestionnaire d'Application (Application Manager) :** Contient les applications présentes sur le noeud.
- **Équilibrage des transactions (Transaction Load Balancer) :** Reçoit les transactions depuis les applications et les envoie sur le noeud le mieux adapté.
- **Gestionnaire de Réplication (Replication Manager) :** Gère la réplication des données entre les noeuds. C'est la couche qui nous intéresse plus particulièrement et que nous développerons dans la suite de la thèse.
- **SGBD :** Traite les transactions.

Comme les noeuds ne partagent ni mémoire ni disques, ils ont besoin de services en communs [OV99]. Un premier service est le *Répertoire Global des Utilisateurs* (Global User Directory), il permet aux utilisateurs de s'identifier à la grappe la première fois qu'ils se connectent sur la couche *Routeur de requêtes*. Sans ce service, un utilisateur devrait s'authentifier à la fois

sur la couche Gestionnaire d'Application et sur la couche SGBD, et ceci sur chaque noeud différent sur lequel il fait une requête. Ce service gère donc les problèmes de confidentialité et de sécurité.

Dans la suite de cette section, nous décrivons chaque composant de notre architecture et plus particulièrement la couche *Gestionnaire de réplication*. Puis nous présentons les deux services utilisés par les différentes couches : le *Moniteur de charge* (Load Monitor) et le *Placement Global des Données* (Global Data Placement).

2.2.1 Routeur de requêtes

La couche *Routeur de requête* est importante car elle est le point d'accès du noeud pour les clients. Tous les noeuds peuvent être un point d'accès, il n'y a pas de noeuds dédiés à cette tâche. Lors de sa première connexion le client doit s'authentifier, pour cela, le Routeur de requêtes va utiliser le service d'authentification contenu dans le Répertoire Global des Utilisateurs. Une fois authentifié, le Routeur de requêtes soumet la requête à la couche *Gestionnaire d'Application* du noeud qui remplit deux conditions : (i) posséder l'application requise par la requête (en utilisant le service de Placement Global des Données) ; et (ii) être le noeud le moins chargé (en utilisant le service de Moniteur de Charge). Grâce à la première condition, tous les noeuds sont capables de traiter toutes les requêtes : soit en les soumettant à la couche Gestionnaire d'applications de son propre noeud, soit en les déléguant à un autre noeud. Une requête utilisateur peut donc changer de noeud dans cette couche. La seconde condition permet un premier équilibrage de charge au niveau des applications, le Routeur de requêtes est donc également une couche où est gérée une partie de l'équilibrage de charge.

2.2.2 Gestionnaire d'applications

La couche *Gestionnaire d'applications* soumet les requêtes qu'elle reçoit aux applications. Dans le cas où l'application a besoin d'effectuer une transaction sur la base de données, elle ne la soumet pas directement à un SGBD, mais à la couche d'*Équilibrage des transactions* du noeud sur laquelle elle se trouve. Cette couche sert simplement d'interface entre la grappe et les applications. Étant donné que le code des applications ne doit être pas modifié, nous avons besoin d'un programme enveloppant pour les applications. Ainsi, chaque entrée-sortie d'une application est encapsulée afin d'autoriser la réception de requêtes depuis la couche Routeur de requête et non plus depuis un client. la couche Gestionnaire d'applications intercepte également les transactions émises par les applications pour les SGBD.

Pour des raisons de simplification, nous considérons les applications comme des programmes dont l'état ne change jamais. C'est-à-dire que l'exécution d'une requête ne modifie pas un programme. Plus simplement, on peut considérer que les applications sont composées uniquement de fichiers exécutables (aucun fichier de données). Ainsi, la réplication des applications n'entraîne pas de gestion de cohérence entre les différentes versions d'une même application comme peut en introduire la réplication des données.

2.2.3 Équilibrage des transactions

La couche d'*Équilibrage des transactions* effectue le même travail que la couche *Routeur de requête*, mais pour les transactions et non pas pour les requêtes d'applications. La couche reçoit une transaction d'une application et la re-route sur le noeud capable de la traiter et qui a la plus faible charge. Tout comme avec la couche *Routeur de requêtes*, n'importe quel noeud peut gérer toutes les transactions. Il faut donc noter qu'une application et son SGBD n'ont pas besoin de se trouver sur le même noeud. Il est même possible de créer une grappe avec une partie des noeuds dédiée aux applications et l'autre partie dédiée aux SGBD. Par exemple, un noeud sans application n'a pas besoin de la couche *Gestionnaire d'applications* et un noeud sans SGBD n'a pas besoin de la couche *Gestionnaire de réplication* et bien sûr de la couche *Base de données*. Cependant, tous les noeuds doivent posséder les couches Routeur de Requêtes car n'importe quel noeud doit pouvoir traiter les requêtes d'un client. La couche d'Équilibrage des Transactions est seulement nécessaire si le noeud possède des applications.

L'indépendance des noeuds est donc garantie par le fait que chaque noeud peut recevoir n'importe quelle requête de n'importe quel client. Les clients n'ont pas besoin de savoir où se trouvent les applications et les données avant de soumettre leur requête. De même, on a une indépendance entre le placement des applications et des données, ce qui autorise un fonctionnement indépendant des couches entre elles. La panne de la couche Gestionnaire d'applications ne bloquera pas le reste du noeud qui peut toujours envoyer ses requêtes des clients vers un autre noeud et peut recevoir les requêtes venant des applications d'un autre noeud.

2.2.4 Gestionnaire de réplication

La couche *Gestionnaire de réplication* est la couche qui nous intéresse plus particulièrement dans cette thèse. Elle gère à la fois l'exécution de la transaction et le rafraîchissement des autres copies. En revanche, une transaction est toujours soumise à la couche SGBD du noeud local. C'est la couche Équilibrage des Transactions qui s'assure qu'une transaction a été soumise à un noeud capable de l'exécuter. Il faut cependant noter que l'exécution d'une transaction avec écritures peut entraîner un rafraîchissement d'autres copies des données, le Gestionnaire de réplication utilise alors le service Global Data Placement pour connaître les noeuds qui possèdent des copies touchées par la transaction.

2.2.5 Base de données

Cette couche ne contient en fait que les SGBD standards (non modifiées). Il est à noter que la couche *Gestionnaire de réplication* n'est pas intrusive. Le SGBD ne devra pas être modifié pour pouvoir être utilisé. Il devra simplement fournir des services standards que nous détaillerons dans un prochain chapitre.

2.2.6 Services partagés

Pour utiliser notre architecture, nous avons besoin en plus des cinq couches de deux services supplémentaires : le *Moniteur de charge* et le *Placement Global des Données*.

Le premier service, le *Moniteur de charge*, permet de connaître la charge des autres noeuds. Il est utilisé par les couches *Routeur de requêtes* et *Équilibrage des transactions* qui effectuent toutes les deux de l'équilibrage de charge. Il existe différentes techniques d'équilibrage basé sur la charge des noeuds. Une première est le **Weighted Round Robin** (WRR) [HGKM98] où chaque noeud possède un poids qui est proportionnel à sa charge, réévaluée périodiquement. Les serveurs sont examinés successivement et le moins chargé d'entre eux est retenu. Une deuxième technique est le **Shortest Queue First** (SQF) où la charge d'un noeud est estimée comme le nombre de transactions en attente sur ce site. Une dernière technique est **Shortest Execution Length First** (SELF) [ACZ03]. Elle mesure hors-ligne le temps d'exécution de référence de chaque classe de requêtes. La charge d'un site est estimée comme la somme des temps d'exécution de toutes les requêtes en attente en cours d'exécution sur ce site pour tenir compte de l'hétérogénéité des durées des requêtes. Bien sûr, l'implémentation du *Moniteur de charge* dépend de la stratégie utilisée. Il faut cependant noter que le système n'a pas besoin d'une connaissance précise de l'état des autres noeuds pour fonctionner, cette connaissance du système peut être différente sur chaque noeud. Ce service ne trahit donc pas l'esprit de modèle sans partage sur lequel est basée notre architecture puisque les noeuds ne partagent pas la même information.

Le second service, le *Placement Global des Données*, donne une connaissance du placement des données et des applications sur la grappe. Il est utilisé par les couches de *Routeur de requêtes*, *Équilibrage des transactions* et *Gestionnaire de réplication*. D'un point de vue implémentation ce service est trivial. Pour une donnée (dans notre cas une table relationnelle) ou une application identifiée par son nom, le service renvoie la liste des noeuds où se trouvent cet élément. Cependant, contrairement au service *Moniteur de charge*, le service doit renvoyer sur tous les noeuds la même vision du placement des données. En effet, si pour une même donnée le service renvoyait une liste différente sur deux noeuds différents, alors sur la couche *Gestionnaire de réplication*, les noeuds à rafraîchir ne seraient pas les mêmes, ce qui entraînerait une incohérence des données. Pour ce service, il faut donc utiliser un système de mémoire partagée [MP97]. Ce service est cependant très rarement mis à jour, lors des pannes d'un noeud ou lors de l'ajout ou la suppression d'une copie.

2.3 Exemple de routage de requête

Pour nos exemples de routage de requêtes, nous supposons une grappe avec trois noeuds N_1, N_2 et N_1 ; deux applications A_1 et A_2 ; et deux données D_1 et D_2. Le noeud N_1 possède l'application A_2 et la donnée D_2. Le noeud N_2 possède l'application A_1. Le noeud N_3 possède les applications A_1 et A_2 et les données D_1 et D_2. Cette configuration est représentée dans les figures 2.2 et 2.3 où l'on peut voir les trois noeuds avec leurs cinq couches.

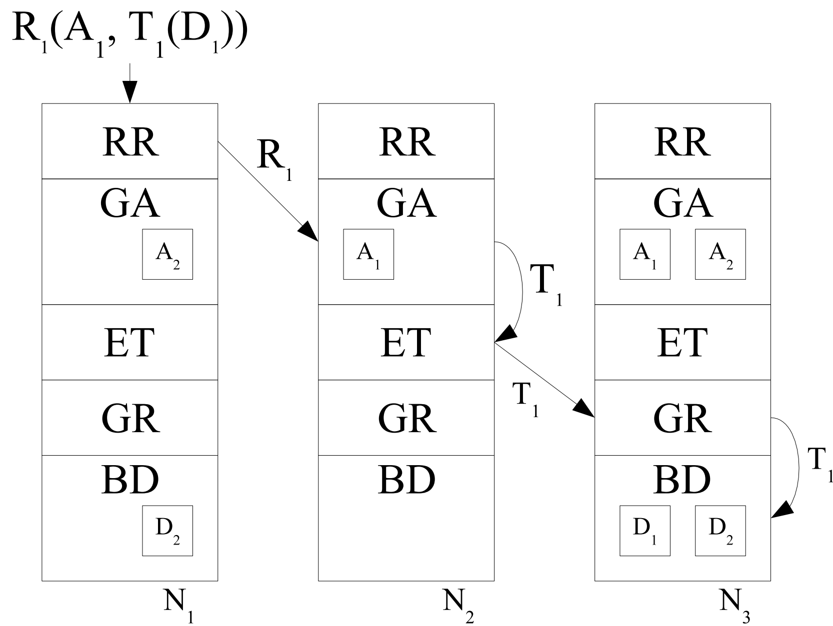


Figure 2.2 – Exemple A de routage de requête

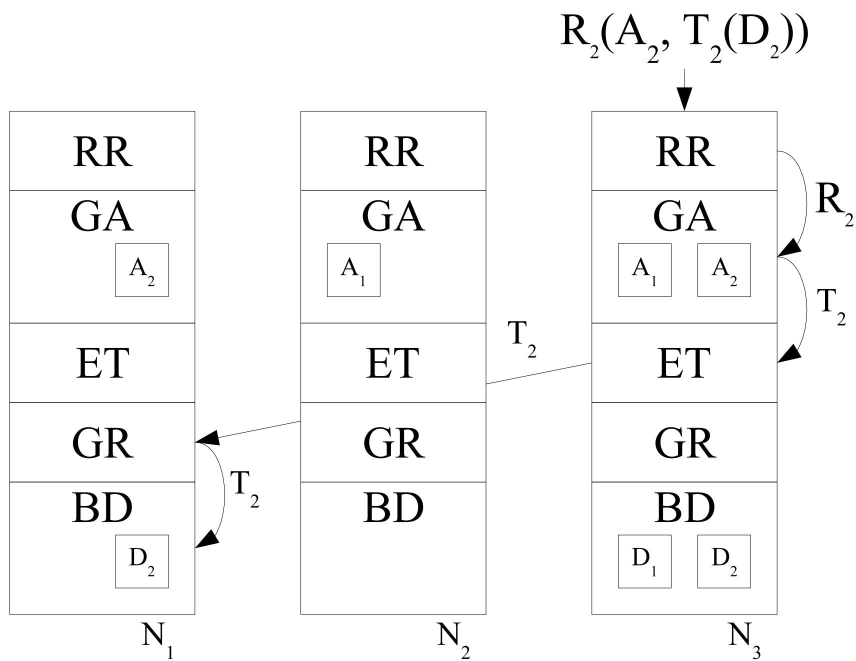


Figure 2.3 – Exemple B de routage de requête

Dans l'exemple A représenté sur la figure 2.2, un client soumet une requête R_1. Cette requête appelle l'application A_1 qui va elle-même mettre à jour la donnée D_1 en créant une transaction T_1. On note une telle requête $R_1(A_1, T_1(D_1))$. Dans un premier temps, la requête R_1 est soumise au noeud N_1. La couche Routeur de requête (RR) va analyser la requête et comme le noeud ne possède pas l'application A_1, la couche va router la requête vers le noeud le moins chargé qui possède A_1. Dans notre cas, la couche RR a le choix entre le noeud N_2 et N_3, mais comme le noeud N_3 est plus chargé, la requête est envoyée à la couche Gestionnaire d'applications (GA) du noeud N_2. La requête va alors être soumise à l'application qui va générer une transaction T_1 qui met à jour la donnée D_1. La transaction est récupérée par la couche GA pour être transmise à la couche d'Équilibrage des transactions (ET) du noeud local. Ici, comme la transaction T_1 met à jour la donnée D_1 et que seul le noeud N_3 possède cette donnée, alors la transaction est routée vers la couche de Gestionnaire de réplication (GR) du noeud N_3. Finalement sur le noeud N_3, la couche GR soumet la transaction à la couche Base de données (BD) et va aussi gérer la cohérence entre les différentes copies de D_1.

Dans l'exemple B représenté sur la figure 2.3, un client soumet une requête R_2. Cette requête appelle l'application A_2 qui va elle-même mettre à jour la donnée D_2 en créant une transaction T_2. On note une telle requête $R_2(A_2, T_2(D_2))$. Dans un premier temps, la requête R_2 est soumise au noeud N_3. La couche RR va analyser la requête et décider de la soumettre à la couche GA du même noeud car il possède l'application A_2 et qu'il n'est pas chargé. L'application A_2 génère une transaction T_2 qui est soumise à la couche ET du noeud N_3. Par contre, alors que le noeud peut exécuter la transaction car il possède la donnée D_1, la couche ET choisit de la re-router sur le noeud N_1 qui est le moins chargé. Pour finir, la couche GR va soumettre la transaction à la couche Base de données (BD) et va aussi gérer la cohérence entre les différentes copies de D_2

2.4 Les services réseaux

Pour la mise au point de notre algorithme de réplication, nous nous basons sur les travaux sur la réplication préventive [PMS01, PMS99]. Pour fonctionner correctement, l'algorithme utilise le modèle réseau reprit dans [GM97]. Nous admettons plusieurs hypothèses sur le réseau.

D'abord, l'interface réseau doit fournir un service de diffusion FIFO par émetteur des messages. C'est-à-dire que les messages diffusés par un même noeud sont reçus par tous les noeuds dans le même ordre que l'ordre de diffusion. À la différence d'un message envoyé en ordre causal ou en ordre total, il n'y a aucun ordonnancement entre les messages envoyés par différents noeuds. C'est donc un service qui augmente peu le coût d'un message.

Une autre hypothèse exploitée par l'algorithme de réplication est que le réseau garantit un délai maximal nécessaire à la diffusion d'un message d'un noeud *i* à n'importe quel autre noeud *j*. Nous notons ce délai par Max. L'estimation de ce délai n'est pas évidente et la valeur Max est en fait sur-évaluée afin qu'aucun message ne soit jamais au-delà de ce délai. La valeur Max est donc très largement au-dessus du temps moyen que met un message pour être émis d'un noeud à un autre. Cependant, dans une grappe avec un réseau rapide, cette estimation est acceptable et

le délai Max n'est pas trop important.

Enfin, les horloges doivent être ϵ-synchronisées. La différence entre deux horloges de n'importe quel noeud ne doit pas être supérieure à ϵ. De nombreux protocoles comme *NTP* [Mil03] existent depuis de longues années et sont parfaitement éprouvés. Leurs précisions sont de l'ordre de quelques dizaines de nanosecondes [Mil03]. De plus, chaque transaction est associée à une estampille chronologique C qui correspond à sa date d'arrivée dans la grappe (dans la couche *Routeur de requêtes*). Cependant, nous n'admettons pas de points d'entrées unique. L'estampille dépend donc de l'horloge du noeud sur lequel elle a été définie. Cependant comme les horloges sont ϵ-synchronisées, la précision d'une estampille est aussi d'ϵ. Les requêtes d'un client sont ϵ-estampillées. Et d'un point de vue pratique, les estampilles sont prises avec une précision de l'ordre de la milliseconde, et comme ϵ est de l'ordre de quelques nanosecondes, il ne modifie pas la valeur de l'estampille.

2.5 Conclusion

Dans ce chapitre, nous présentons une architecture sans partage adaptée au contexte ASP où une grappe contient les bases de données mais aussi les applications d'un client.

Notre architecture, par sa modélisation en cinq couches et l'utilisation de trois services, permet de garantir l'autonomie des noeuds. Nos exemples montrent que l'architecture en couches permet un équilibrage de charge à deux niveaux : les applications et les bases de données. De plus, par un routage des requêtes sur deux des couches, nous autorisons la séparation des applications et des données sur plusieurs noeuds. Ce routage permet également aux noeuds de pouvoir traiter n'importe quelle requête, et ceci même si le noeud ne possède pas les applications ou les données nécessaires à la requête, en la déléguant à un autre noeud. Finalement, dans notre solution, les noeuds sont autonomes les uns par rapport aux autres et la panne d'un noeud n'entraîne pas le blocage des autres sites. De même chaque couche est indépendante l'une de l'autre et la panne de l'une d'entre elles n'entraîne pas la panne des autres.

Nous avons également présenté les hypothèses réseau nécessaires au fonctionnement de notre algorithme de réplication : une diffusion sûre en ordre FIFO par émetteur des messages, une synchronisation des horloges sur tous les noeuds et un service garantissant qu'un message ne met pas plus qu'un délai maximum pour être diffusé sur tous les noeuds.

CHAPITRE 3

Algorithme de réplication préventive

3.1 Introduction

Dans ce chapitre, nous nous intéressons à la mise au point de la couche *Gestionnaire de réplication* présentée dans la section 2.2 du chapitre 2. Nous nous focalisons plus particulièrement sur l'élaboration d'un algorithme de réplication. Pour ce faire, nous nous basons sur l'algorithme de réplication asynchrone *Lazy-Master* de Pacitti et al. [PMS01, PMS99]. Dans cet algorithme (voir Section 1.6.3), l'exécution des transactions de rafraîchissement est retardée pour permettre leur ordonnancement. Ainsi, l'ordre d'exécution des transactions est le même sur tous les noeuds. L'algorithme a pour avantages d'utiliser un faible nombre de messages peu coûteux, de prévenir les abandons et de maintenir une cohérence forte. Cependant, il introduit des restrictions sur le placement des données et supporte uniquement les *Copies primaires*.

Nous reprenons cet algorithme en tentant de l'améliorer. Dans un premier temps, nous ajoutons le support des *Copies multimaîtres* ; dans ce mode, plus d'un noeud peut effectuer des mises à jour sur une même copie. De plus, les copies sont disponibles sur tous les noeuds, d'où le nom de réplication totale. Puis nous faisons évoluer l'algorithme pour autoriser la réplication partielle des données. Ainsi, les noeuds ne sont plus obligés de détenir toutes les copies et les copies peuvent être de types copies primaires, multimaîtres ou secondaires sans aucune restriction sur le placement.

Ce chapitre est organisé comme suit. Nous décrivons dans un premier temps un modèle de cohérence pour les différentes configurations supportées par notre algorithme. Puis nous présentons l'algorithme pour la réplication totale avec le détail de l'algorithme et une architecture pour la couche *Gestionnaire de réplication*. Ensuite, nous présentons son extension pour les configurations partielles avec une nouvelle architecture. Finalement, nous démontrons la validité de nos algorithmes grâce à des preuves qui montrent que les configurations totales et partielles sont bien supportées, qu'un ordre total d'exécution des transactions sur tous les noeuds est bien respecté et également qu'aucun verrou mortel n'apparaît lors de l'exécution de l'algorithme. Nous prouvons ainsi que les algorithmes garantissent la cohérence forte pour toutes les configurations.

3.2 Modèle de cohérence

Un algorithme de rafraîchissement correct garantit que 2 noeuds qui possèdent un ensemble commun de copies, R_1, R_2, ..., R_n, doivent toujours produire la même séquence de mises à jour sur R_1, R_2, ..., R_n. Pour chaque configuration, Maître-paresseux, Multimaîtres et Partielles (définies en Section 1.2.2), nous fournissons un critère qui doit être satisfait par l'algorithme de rafraîchissement pour que ce dernier soit correct. Notre modèle de cohérence s'appuie sur les *Systèmes de Communications de Groupes* (SCG) qui fournissent des services de diffusion qui changent l'ordre final de réception des messages. Nous utilisons ces ordres connus [PV98] comme un guide pour exprimer notre critère de cohérence. Un exemple pour chaque configuration est présenté en Section 1.2.2.

3.2.1 Maître-paresseux

Dans les configurations maîtres-paresseuses, des incohérences peuvent apparaître lorsque les noeuds esclaves valident les transactions de rafraîchissement dans un ordre différent des noeuds maîtres correspondants. La Définition 3.1 montre les conditions qui permettent d'éviter cette situation.

Définition 3.1. (Ordre Total). Deux transactions de rafraîchissement RT_1 et RT_2 sont dites en ordre total si n'importe quel noeud esclave qui valide RT_1 et RT_2, les valident dans le même ordre.

Proposition 3.1. *Pour n'importe quel configuration C qui rencontre les contraintes d'une configuration maître-paresseux, l'algorithme de rafraîchissement utilisé par C est correct si l'algorithme garantit l'ordre total.*

3.2.2 Multimaîtres

Dans les configurations multimaîtres, des incohérences peuvent apparaître lorsque l'ordre d'exécution de deux transactions sur deux noeuds ayant les mêmes copies est différent. Ainsi, l'ordonnancement FIFO n'est pas suffisant pour garantir l'exactitude de l'algorithme de rafraîchissement. Par conséquent, le critère d'exactitude suivant est nécessaire :

Définition 3.2. (Ordre Total). Deux transactions T_1 et T_2 sont dites exécutées en ordre total si tous les noeuds multimaîtres qui valident à la fois T_1 et T_2 les valident dans le même ordre.

Proposition 3.2. *Pour n'importe quel configuration C qui rencontre les contraintes d'une configuration multimaître, l'algorithme de rafraîchissement utilisé par C est correct si l'algorithme garantit l'ordre total.*

3.2.3 Partielles

Dans une configuration partielle, les problèmes d'incohérences sont similaires à ceux trouvés dans chaque sous-configuration, maître-paresseuses et multimaîtres. Ainsi, deux transactions T_1 et T_2 doivent être exécutées dans le même ordre sur les noeuds multimaîtres et leurs transactions de rafraîchissement correspondantes RT_1 et RT_2 doivent être exécutées dans l'ordre dans lequel les noeuds multimaîtres ont validé T_1 et T_2. Par conséquent, le critère d'exactitude suivant prévient l'incohérence.

Définition 3.3. Pour une configuration C qui rencontre les contraintes d'une configuration partielle, l'algorithme de rafraîchissement utilisé par C est correct si pour chaque sous-configuration SC l'exactitude est assurée (voir les propositions 3.1 et 3.2).

Proposition 3.3. *Pour n'importe quelle configuration C qui rencontre les contraintes d'une configuration partielle, l'algorithme de rafraîchissement est correct si et seulement si l'algorithme garantit l'ordre total.*

3.3 Réplication totale

Comme base de notre algorithme, nous reprenons l'algorithme *Lazy-Master*. L'un des désavantages de cet algorithme est d'utiliser un mode Copie Primaire où une copie ne peut être mise à jour que sur un seul noeud. Pour permettre l'utilisation de copies multimaîtres nous n'allons plus seulement ordonner les transactions de rafraîchissement, mais aussi les transactions qui sont envoyées par les clients. Ainsi, même si deux transactions mettent à jour une copie sur deux noeuds différents, elles sont totalement ordonnées grâce aux propriétés réseau et s'exécutent dans un ordre total.

Dans la suite de cette section, nous décrivons dans un premier temps le principe de l'algorithme de réplication préventive pour la réplication totale. Puis, nous donnons un exemple d'exécution de l'algorithme qui montre que la cohérence forte est garantie. Ensuite, nous présentons l'architecture de la couche Gestionnaire de Réplication. Et pour finir, nous détaillons les algorithmes de chaque module de notre architecture.

3.3.1 Principe de l'algorithme

Le principe de l'algorithme de rafraîchissement préventif est de soumettre une séquence de transactions dans le même ordre chronologique sur tous les noeuds. Avant de soumettre une transaction pour exécution à un noeud i, nous devons vérifier qu'aucune transaction plus ancienne n'est en route pour le noeud i. Pour accomplir cela, la soumission d'une transaction est retardée de $Max + \epsilon$. Ainsi, la date de soumission d'une transaction ayant pour estampille C est nommée le *delivery-time* et correspond à la valeur $C + Max + \epsilon$.

Lorsqu'une transaction T_i est reçue sur un noeud i, le noeud i diffuse T_i à tous les noeuds y compris lui-même. Dès que T_i est reçue sur un noeud j (i peut être égal à j). Elle est placée dans la file d'attente correspondante à i. Donc, pour chaque noeud multimaître i il y a une file

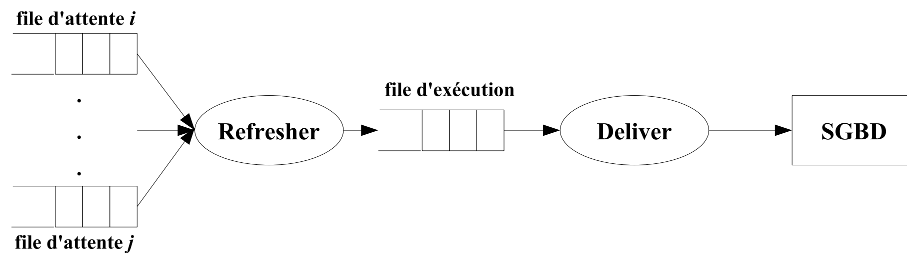


Figure 3.1 – Architecture de rafraîchissement

d'attente q_i correspondante sur tous les noeuds. Chacune de ces files correspond à un noeud multimaître et est utilisée pour exécuter les transactions dans l'ordre chronologique. La Figure 3.1 montre une partie des composants nécessaires pour exécuter l'algorithme. Le module de rafraîchissement (*Refresher*) lit les transactions en tête des files d'attente et les ordonnances chronologiquement en fonction de leur *delivery-time*. Quand une transaction est ordonnée, elle est alors soumise à la file d'exécution dans un ordre FIFO. Finalement, le module d'exécution (*Deliver*) vérifie continuellement la file d'exécution et exécute les transactions sur le SGBD local dans leur ordre d'arrivée, l'une après l'autre.

3.3.2 Exemple d'exécution

Voici un exemple qui illustre notre algorithme. Nous supposons deux noeuds i et j, maîtres de la copie R. Donc, sur le noeud i, il y a deux files d'attente : $q(i)$ et $q(j)$ correspondantes aux noeuds maîtres i et j. T_1 et T_2 sont deux transactions qui mettent à jour R, respectivement sur le noeud j et sur le noeud i. Nous supposons également que la valeur de Max est égale à 10 et la valeur de ϵ est égale à 1. Sur le noeud i, on pourra retrouver l'exécution suivante :

- Au temps 10 : T_2 arrive sur le noeud i avec une estampille C_2 = 5
  - $q(i)$ = [T_2 (5)], $q(j)$ = [ ]
  - T_2 est choisie par le Refresher pour être soumise au Deliver au temps *delivery-time* 16 (5 + 10 + 1) et le minuteur est réglé pour expirer à 16.

- Au temps 12 : T_1 arrive sur le noeud j avec une estampille C_1 = 3
  - $q(i)$ = [T_2 (5)], $q(j)$ = [T_1 (3)]
  - T_1 est choisie par le Refresher pour être soumise au Deliver au temps *delivery-time* 14 (3 + 10 + 1) et le minuteur est réglé pour expirer à 14.

- Au temps 14 : le minuteur expire, le Refresher soumet T_1 à la file d'exécution
  - $q(i)$ = [T_2 (5)], $q(j)$ = [ ]
  - T_2 est choisie par le Refresher pour être soumise au Deliver au temps *delivery-time* 16 (5 + 10 + 1) et le minuteur est réglé pour expirer à 16.

- Au temps 16 : le minuteur expire, le Refresher soumet T_2 à la file d'exécution

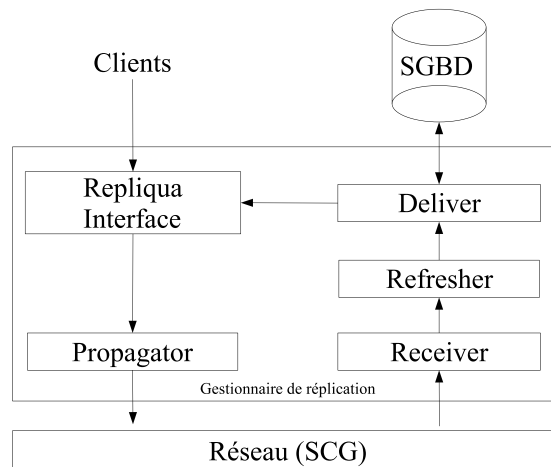


Figure 3.2 – Architecture du gestionnaire de réplication pour les configurations multimaîtres

- $q(i)$ = [ ], $q(j)$ = [ ]

Sur cet exemple, on peut voir que bien que les transactions aient été reçues dans le mauvais ordre (l'ordre inverse de leur estampille, T_2 puis T_1), elles ont été soumises à la file d'exécution dans le bon ordre (l'ordre de leur estampille, T_1 puis T_2). L'ordre total est garanti alors même que les messages ne sont pas diffusés en ordre total.

Ainsi, tous les noeuds qui reçoivent T_1 et T_2, les exécutent en fonction de leur estampille et non de leur ordre d'arrivée. L'algorithme garantit ainsi que tous les noeuds exécutent les transactions dans le même ordre.

3.3.3 Architecture

Dans cette section, nous présentons l'architecture du gestionnaire de réplication (*Replica Manager*) nécessaire à l'implémentation de l'algorithme de réplication préventive totale. Le gestionnaire de réplication se situe entre le gestionnaire de transaction et le SGBD dans l'architecture générale présentée au chapitre 2. Pour rendre notre algorithme de réplication le plus indépendant possible de l'architecture générale nous considérons la couche d'équilibrage de charge comme un client soumettant des transactions. Ainsi, le gestionnaire de réplication n'a aucun objectif d'équilibrage de charge du système. Pour construire cette architecture, nous avons ajouté plusieurs composants autour du SGBD, mais toujours dans l'optique de préserver son autonomie (sans utiliser les mécanismes internes du SGBD).

Le gestionnaire de réplication est composé de cinq modules. L'interface de réplication (*Replica Interface*) reçoit les transactions des clients. Le propagateur (*Propagator*) et le receveur (*Receiver*) gèrent, respectivement, l'envoi et la réception des transactions dans des messages diffusés entre les noeuds sur le réseau. Comme tous les noeuds possèdent toutes les copies, le Propagator diffuse la transaction à tous les noeuds (y compris lui-même). Dès que le Receiver reçoit une transaction, il la place dans la file d'attente correspondante à l'émetteur du message.

C'est le module de rafraîchissement (*Refresher*) lit cette file continuellement afin de déterminer la prochaine transaction prête à être exécutée en fonction de son estampille comme décrit dans la section 3.3.1. La transaction élue est alors retirée de la file d'attente afin d'être placée dans la file d'exécution. Finalement, le module d'exécution (*Deliver*) se charge de lire la plus ancienne transaction placée dans la file d'exécution afin de la soumettre au SGBD.

3.3.4 Algorithmes

Dans cette section, nous présentons les algorithmes de chaque module présentés dans l'architecture (section 3.3.3). S'ils communiquent entre eux, chaque module est indépendant et totalement autonome.

3.3.4.1 Replica Interface

L'algorithme 4 décrit en détail le fonctionnement du module Replica Interface. Le rôle principal de ce module est de recevoir les requêtes des clients (ligne 3). Nous rappelons que notre algorithme de réplication utilise une technique *ROWA* où les transactions de lecture sont exécutées sur un seul noeud. Ainsi, si la transaction ne contient pas de mises à jour (ligne 7), elle est soumise directement au SGBD sans être diffusée sur tous les noeuds, et sans être retardée. Si la transaction contient des mises à jour (ligne 4), elle est envoyée au module Progator par une file appelée file de propagation qui est lue par le Propagator. La connexion avec le client est alors maintenue et mise en attente jusqu'à ce que le module Deliver exécute la transaction et retourne la réponse (ligne 13). Une fois cette réponse renvoyée au client, la connexion peut être retirée de la liste des transactions en cours ($liste_c$). Un autre rôle du Replica Interface est d'estampiller les transactions, il n'est pas nécessaire que le même noeud estampille toutes les transactions, car la différence entre deux horloges n'est pas supérieure à ϵ.

3.3.4.2 Propagator

L'algorithme 5 décrit en détail le module Propagator. Le rôle du Propagator est d'attendre l'arrivée d'une nouvelle transaction dans la file de propagation (ligne 4). Il retire alors la transaction (ligne 5) et la diffuse à tous les noeuds (ligne 6) en s'appuyant sur un SCG. Le mode de diffusion du message est FIFO par émetteur afin d'éviter une désynchronisation entre deux messages envoyés par un même noeud. Rappelons que dans les configurations multimaîtres, tous les noeuds possèdent toutes les copies. Ainsi, dans ce type configuration, il est indispensable de diffuser la transaction à tous les noeuds.

3.3.4.3 Receiver

L'algorithme 6 décrit le module Receiver. Le rôle de ce module est de recevoir les transactions du réseau et de les soumettre au module Refresher. À la réception d'une transaction (ligne 3), il devra simplement la placer dans la file d'attente correspondante au noeud émetteur de la transaction (ligne 4).

Algorithme 4 – Module Replica Interface pour la réplication multimaître

Entrées:
C : une connexion d'un client
r_T : la réponse du Deliver pour la transaction T

Sorties:
file de propagation

Variables:
T : une transaction d'un client
$liste_C$: liste des clients actifs

1: **début**
2: **boucler**
3: **sur** la réception de la transaction T depuis un client C **faire**
4: **si** T contient des opérations de mise à jour **alors**
5: estampiller T
6: ajouter T à la file de propagation
7: ajouter C à $liste_C$ en fonction de T
8: **sinon**
9: soumettre T au SGBD
10: attendre la réponse r_T du SGBD
11: envoyer r_T à C
12: **fin si**
13: **fin sur**
14: **sur** la réception sur le résultat d'un transaction r_T du Deliver **faire**
15: récupérer C de $liste_C$ en fonction de T
16: envoyer r_T à C
17: enlever C de $liste_C$
18: **fin sur**
19: **fin boucler**
20: **fin**

Algorithme 5 – Module Propagator pour la réplication multimaître

Entrées:
file de propagation
Variables:
T : une transaction

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T dans la file de propagation **faire**
4: retirer T de la file de propagation
5: diffuser T à tous les noeuds
6: **fin sur**
7: **fin boucler**
8: **fin**

Algorithme 6 – Module Receiver pour la réplication multimaître

Sorties:
files d'attente q_1,... q_n
Variables:
T : une transaction

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T du noeud n **faire**
4: ajouter T à la file d'attente correspondante q_n
5: **fin sur**
6: **fin boucler**
7: **fin**

3.3.4.4 Refresher

L'algorithme 7 montre l'algorithme du module de rafraîchissement (*Refresher*). Il se déroule en trois étapes. La première étape permet de choisir la transaction la plus ancienne (avec l'estampille la plus petite) parmi les transactions en tête des files d'attente (ligne 7), cette transaction est nommée $premier_T$. Comme les transactions sont reçues en ordre FIFO par émetteur, la transaction en tête d'une file est toujours la plus ancienne pour la file. Dans l'étape 2, on compare la transaction nouvellement élue $premier_T$ avec la transaction en cours de traitement $elue_T$ (lignes 9 à 12). Si elles sont différentes, alors $premier_T$ devient la prochaine transaction a exécuter. On calcule également un compte à rebours (*timer*) qui se termine lorsque la transaction est prête a être exécutée. L'étape 3 est parcourue lorsqu'une transaction est prête à être exécutée, son *timer* a expiré (lignes 15). La transaction est alors envoyée vers la file d'exécution (lignes 16 et 17). Puis une nouvelle transaction $elue_T$ est élue grâce à un saut vers l'étape 1 (ligne 19).

Algorithme 7 – Module Refresher pour la réplication multimaître

---

Entrées:
  files d'attente q_1,... q_n
Sorties:
  file d'exécution
Variables:
  $elue_T$: transaction actuellement élue pour être la prochaine à être exécutée
  $premier_T$: transaction avec la plus petite estampille dans les files d'attente
  $timer$: compte à rebours local

1: **début**
2:   $elue_T \Leftarrow 0$
3:   $premier_T \Leftarrow 0$
4:   **boucler**
5:     **sur** l'arrivée d'une nouvelle transaction **ou lorsque** (timer.state = active **et** timer.value = 0) **faire**
6:       **Etape** 1**:**
7:         $premier_T \Leftarrow$ transaction avec l'estampille C la plus petite parmi les messages en tête des files d'attente
8:       **Etape** 2**:**
9:         **si** $premier_T \neq elue_T$ **alors**
10:           $elue_T \Leftarrow premier_T$
11:           calculer *delivery-time(*$elue_T$*)*
12:           timer.value $\Leftarrow$ *delivery-time(*$elue_T$*)* - temps local
13:         **fin si**
14:       **Etape** 3**:**
15:         **si** $elue_T \neq 0$ **et** timer.value = 0 **alors**
16:           ajouter $elue_T$ à la file d'exécution

17: enlever $elue_T$ de sa file d'attente
18: $elue_T \Leftarrow 0$
19: saut à l'étape 1
20: **fin si**
21: **fin sur**
22: **fin boucler**
23: **fin**

3.3.4.5 Deliver

L'algorithme 8 décrit en détail le module Deliver. Le Deliver lit en permanence la file d'exécution où sont déposées les transactions prêtes à être exécutées par le module Refresher (ligne 3). À l'arrivée d'une nouvelle transaction T, il soumet directement T au SGBD (ligne 4 et 5) et attend la réponse r_T afin de la renvoyer au client par l'intermédiaire du module Replica Interface (ligne 7). Bien sûr, la réponse n'est retournée au client que si la transaction est locale au noeud (ligne 6), c'est-à-dire si le client est bien connecté sur le noeud.

Algorithme 8 – Module Deliver pour la réplication multimaître

Entrées:
file d'exécution
Sorties:
SGBD
Replica Interface
Variables:
T : une transaction
r_T : la réponse du SGBD à T

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T dans la file d'exécution **faire**
4: enlever T de la file d'exécution
5: soumettre T au SGBD
6: **si** T est locale au noeud **alors**
7: envoyer r_T au Replica Interface
8: **fin si**
9: **fin sur**
10: **fin boucler**
11: **fin**

```
UPDATE R1 SET att1=value
    WHERE att2 IN
    (SELECT att3 FROM S)
COMMIT;
```

Figure 3.3 – Exemple de transaction

3.4 Réplication partielle

Dans cette section, nous faisons évoluer l'algorithme pour autoriser la réplication partielle des données. En effet, l'utilisation d'une configuration totale est dans la pratique assez peu exploitable car tous les noeuds doivent posséder toutes les copies. Avec une réplication partielle des données [CMZ04, CMZ03, HAA02] les noeuds ne sont plus obligés de détenir toutes les copies et les copies peuvent être de types copies primaires, multimaîtres ou secondaires sans aucune restriction sur le placement.

Une transaction T peut être composée de séquences de lectures et d'écritures suivies d'un *commit* (comme dans la Figure 3.3). Pour rafraîchir une copie multimaître dans le cas d'une réplication totale, la diffusion des transactions à tous les noeuds est suffisante. Mais dans le cas d'une réplication partielle, même si une transaction T est diffusée à tous les noeuds, il peut arriver que certains noeuds ne puissent pas exécuter T parce qu'ils ne possèdent pas localement toutes les copies requises par la transaction. Par exemple, la Figure 3.4 permet la réception d'une transaction sur le noeud N_1 pour lire la copie S et écrire sur la copie R. Cette transaction peut être entièrement exécutée sur le noeud N_1 et sur le noeud N_2. Cependant, elle ne peut pas être exécutée sur le noeud N_3 puisqu'il ne possède pas la copie S. Par conséquent, l'algorithme de rafraîchissement multimaître ne supporte pas ce genre de configuration.

Dans la suite de cette section, nous décrivons les modifications apportées à l'algorithme de réplication préventive pour le support des configurations partielles. Puis, à l'aide d'un exemple d'exécution de l'algorithme, nous montrons que la cohérence forte est maintenue. Ensuite, nous présentons les différences d'architecture par rapport à la réplication totale. Pour finir, nous détaillons les algorithmes des modules modifiés dans l'architecture.

3.4.1 Principe de l'algorithme

Nous définissons une transaction de rafraîchissement, notée RT, comme l'ensemble des écritures de la transaction T. Une transaction de rafraîchissement peut donc être considérée comme le jeu d'écritures de la transaction T. Nous pouvons alors utiliser cette transaction de rafraîchissement pour être exécutée sur les noeuds qui ne possèdent pas toutes les copies. Cependant, RT doit également être ordonnancée, et pour garantir la cohérence, il faut exécuter RT en fonction de l'estampille de T. T est donc normalement diffusée aux autres noeuds, mais, au lieu d'être soumise pour exécution, le noeud attend la transaction de rafraîchissement associée RT en provenance du noeud d'origine. Ainsi, sur le noeud d'origine, quand la validation de T est détectée, la transaction RT est créée et diffusée aux noeuds qui en ont besoin. Ainsi,

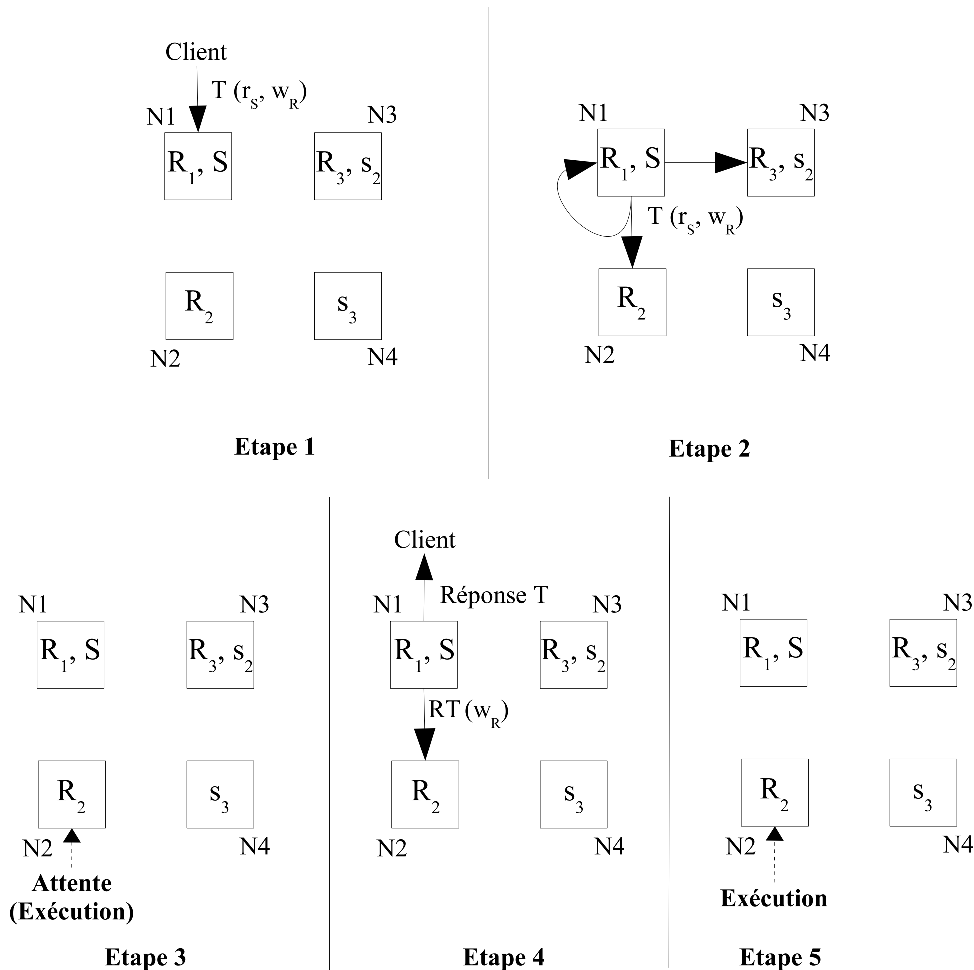


Figure 3.4 – Exemple de rafraîchissement préventif avec une configuration partielle

les noeuds en attente sur T pourront remplacer le contenu de T par celui de RT et exécuter la transaction.

3.4.2 Exemple d'exécution

Nous illustrons le principe de l'algorithme de réplication préventive pour les configurations partielles avec un exemple d'exécution d'une transaction. Dans la Figure 3.4, nous supposons une configuration avec 4 noeuds (N_1, N_2, N_3 et N_4) et 2 copies (R et S). N_1 possède une copie multimaître de R et une copie primaire de S, N_2 possède une copie multimaître de R, N_3 possède une copie secondaire de S et N_4 possède une copie multimaître de R et une copie secondaire de S. Le rafraîchissement de la copie R sur N_2 s'effectue en cinq étapes. À la première étape, le noeud d'origine (N_1) reçoit la transaction T depuis un client, T lit la copie S et met à jour la copie R. Par exemple, T peut être la transaction présentée dans l'Exemple 3.3. Puis, à l'étape 2, N_1 diffuse T aux noeuds concernés par la transaction : N_1, N_2 et N_4. N_3 n'est

pas concerné par T parce qu'il possède seulement une copie secondaire de S et S n'est pas mise à jour par T. À l'étape 3, T peut être exécutée sur N_1 et N_4. Sur N_2, T est également prise en compte par le Refresher et est ensuite envoyée dans la file d'exécution. Cependant, T ne peut pas être exécutée sur ce noeud car N_2 ne possède pas la copie S. Par conséquent, le Deliver doit attendre la transaction de rafraîchissement RT correspondante afin d'appliquer les mises à jour sur R. À l'étape 4, après la validation de T sur le noeud d'origine, la réponse est renvoyée au client et dans le même temps, RT est produite et diffusée à tous les noeuds impliqués. À la dernière étape, N_2 reçoit RT et le contenu de T est remplacé par le contenu de RT. Le Deliver peut alors soumettre RT au SGBD et la copie R de N_2 est rafraîchie.

3.4.3 Gestion des blocages

Un des problèmes de cette solution est que la réplication dans les configurations partielles peut être bloquante en cas de panne. En effet, entre l'envoi d'un transaction T et de sa transactions de rafraîchissement correspondantes RT, si le noeud tombe en panne, alors certains noeuds cibles peuvent être bloqués dans l'attente de la transaction RT. Cependant, ce problème peut être facilement résolu en remplaçant le noeud d'origine par un noeud équivalent. Dans notre cas, un noeud équivalent est un noeud qui possède les copies nécessaires à l'exécution de T. Dès que le noeud cible détecte la panne du noeud d'origine, il peut demander à un noeud équivalent j de diffuser la RT correspondante à l'identifiant de T. Sur le noeud j, RT a déjà été produite de la même manière que sur que le noeud d'origine : la transaction T est exécutée et à la détection de la validation de T, une RT est produite et stockée dans un journal des RT. Dans le pire cas, aucun autre noeud ne possède les copies nécessaires à l'exécution de T, alors T est abandonnée sur tous les noeuds cibles. Donc T n'est validée sur aucun noeud disponible.

Reprenons l'exemple vu dans la section précédente (Figure 3.4). Si N_1 tombe en panne à l'étape 3, N_2 ne peut recevoir la RT_n correspondante à la transaction en attente T_n. Une fois que N_2 détecte que N_1 n'est plus disponible, il identifie N_4 comme équivalent de N_1 pour la transaction T (grâce au service de placement global des données décrit au Chapitre 2.2) et lui demande l'envoi de RT_n. Dans le cas où N_4 est également indisponible, alors aucun noeud ne peut exécuter T_n. Par conséquent, N_2 peut abandonner T_n. La cohérence des données entre les noeuds est assurée car aucun des noeuds actifs n'a exécuté la transaction. Dans ce cas, lors de la phase de reprises, les noeuds tombés en pannes devront défaire T_n (si elle a été exécutée).

3.4.4 Architecture

L'architecture pour la réplication partielle est présentée sur la Figure 3.5. Elle est très similaire à l'architecture pour la réplication totale (section 3.3.3). Cependant, on peut voir l'ajout d'un nouveau module appelé *Log Monitor*. Le rôle de ce module est de lire constamment les fichiers de journalisation du SGBD afin de détecter si une copie a été mise à jour. Pour chaque transaction T qui met à jour une ou plusieurs copies, il produit une transaction de rafraîchissement RT correspondante ne contenant que les jeux d'écritures. Sur le noeud d'origine, lorsque d'autres noeuds ont besoin de la transaction de rafraîchissement RT pour exécuter la transac-

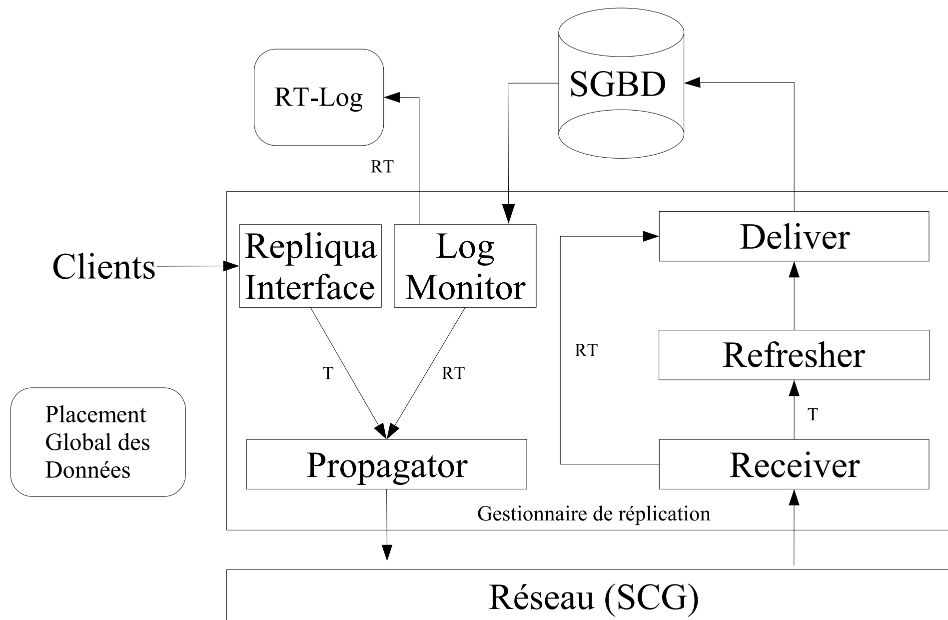


Figure 3.5 – Architecture du module de réplication

tion T correspondante, RT est soumise au module Propagator pour diffusion. Afin de fournir une tolérance en cas de panne du noeud d'origine (voir Section 3.4.3), le module Log Monitor stocke toutes les RT sur tous les noeuds capables d'exécuter la transaction T (tous les noeuds possédant toutes les copies nécessaires à la transaction). Ainsi, en cas de panne du noeud d'origine, l'un de ces noeuds peut remplacer le noeud d'origine et diffuser la RT aux noeuds qui ne peuvent exécuter la transaction T correspondante.

Dans une configuration partielle, tous les noeuds ne possèdent pas toutes les copies. Le nombre de messages émis est alors être réduit. En effet, la diffusion d'une transaction n'est faites qu'en direction des noeuds concernés par la transaction. Le rôle du Propagator est donc de filtrer les noeuds sur lesquels une transaction est diffusée. Le module a donc besoin de deux informations pour fonctionner correctement : la liste des copies touchées par la transaction et le placement des copies dans le système. Le SGBD étant considérée comme une boîte noire, la liste des copies lues ou écrites ne peut être déterminée en fonction du contenu de la transaction. Elle est déterminée à l'avance. Cette hypothèse est acceptable dans un contexte de grappes, car la grande majorité des transactions sont déjà connues. Finalement, une requête au module de Placement global des données (*Global Data Placement*) permet de déterminer la liste des noeuds détenteur d'une copie.

Dès que le Receiver reçoit une transaction, il la place dans la file d'attente correspondante à l'émetteur du message. Le module de rafraîchissement (*Refresher*) lit cette file continuellement afin de déterminer la prochaine transaction prête à être exécutée en fonction de son estampille comme décrit dans la section 3.3.4. Cette transaction est alors retirée de la file d'attente afin d'être placée dans la file d'exécution. Ensuite, le module d'exécution (*Deliver*) lit la transaction en tête de la file d'exécution afin de la soumettre au SGBD. Cependant, certaines transactions

ne pourront pas être exécutées, car le noeud ne possède pas toutes les copies nécessaires. La transaction est alors mise en attente au lieu d'être soumise au SGBD. Le Deliver détecte ces cas en comparant la liste des copies du noeud avec la liste des copies utilisées par la transaction (en lecture et en écriture). À la réception d'une transaction de rafraîchissement, le module Receiver transmettra directement la transaction au module Deliver sans passer par le Refresher. En effet, la transaction de rafraîchissement n'a pas besoin d'être ordonnée, la transaction correspondante l'est déjà. Le Deliver remplace alors le contenu de T par son jeu d'écritures contenu dans RT.

Le module de surveillance des journaux (*Log Monitor*) est un nouveau module par rapport à l'architecture de réplication préventive pour la réplication totale (voir Section 3.3.3). Il vérifie constamment le contenu des journaux du SGBD. Et pour chaque transaction qui met à jour une copie, il produit la transaction de rafraîchissement qui contient les jeux d'écritures de la transaction T correspondante. Alors, sur le noeud d'origine de T, si certains noeuds ne détiennent pas toutes les copies nécessaires à l'exécution de T, le Log Moniter va soumettre la RT au Propagator pour diffusion. Finalement pour fournir une tolérance aux fautes, les RT sont non seulement soumises au SGBD, mais elles sont également stockées dans le journal des transactions de rafraîchissement (*RT-Log*). Tous les noeuds capables d'exécuter T génèrent ainsi la RT correspondante et la stocke dans le RT-Log. En cas de panne du noeud d'origine, l'un de ces noeuds peut le remplacer et diffuser la RT correspondante même s'il n'est pas le noeud d'origine.

3.4.5 Algorithmes

Dans cette section, nous présentons les algorithmes des modules présentés dans l'architecture (voir la Section précédente). Tout comme pour la réplication totale, les modules sont indépendants les uns des autres. Le Replica Interface et le Refresher étant identique à ceux de la réplication totale, nus ne les décrirons pas dans cette section (voir Section 3.3.4).

3.4.5.1 Log Monitor

L'algorithme 9 décrit en détail le Log Monitor. Pour ce module de détection des jeux d'écritures, nous supposons que chaque transaction est identifiée localement sur chaque noeud. Typiquement, les SGBD possèdent un numéro de transaction pour chaque transaction. Le Log Monitor reçoit individuellement les opérations d'une transaction. Ces opérations sont de 3 types : les opérations d'initialisation (BEGIN), de mises à jour (UPDATE, INSERT et DELETE) et de terminaison (COMMIT et ABORT). À la réception d'une opération d'initialisation ($o_T(id)$), le module crée une structure en mémoire identifiée par le numéro de transaction ($liste_{op}(id)$) pour stocker les opérations (ligne 4 et 5). Puis chaque opération reçue est stockée dans cette structure (ligne 17). Et finalement, à la réception d'une opération de terminaison (ligne 6), la structure est retirée de la mémoire et soumise au module de propagation afin d'être diffusée à tous les noeuds qui ont besoin du jeu d'écritures (ligne 7 à 15). Les jeux d'écritures ainsi obtenus sont identifiés localement ($local_T$). Nous utilisons alors cette identifiant local pour obtenir l'identifiant global ($global_T$). Rappelons qu'une transaction est identifiée par son noeud d'origine et son estampille. Donc au moment de l'exécution de la transaction (dans le Deliver), on récupère le numéro

de transaction local que l'on stocke dans une liste ($liste_T$) en l'associant à l'identifiant global qui est alors connu. Lors de la récupération des opérations, on peut donc retrouver l'identifiant global grâce à l'identifiant local (ligne 8).

3.4.5.2 Propagator et Receiver

Les algorithmes du Propagator et du Receiver sont très peu modifiés par rapport à la réplication totale. C'est pourquoi nous décrivons les changements sans détailler les algorithmes.

Le rôle du Propagator est toujours de diffuser les transactions. Mais, comme dans une configuration partielle tous les noeuds ne possèdent pas toutes les copies, il se limite aux noeuds touchés par la transaction au lieu de les diffuser sur tous les noeuds. Il a également un nouveau rôle qui est de diffuser les transactions de rafraîchissement. Là encore, il doit se limiter aux noeuds qui n'ont pas toutes les copies pour exécuter la transaction originelle.

Le Receiver récupère maintenant deux types de messages : les transactions qu'il ajoute toujours en queue de sa file d'attente correspondante, et les transactions de rafraîchissement qu'il envoie directement au Deliver.

3.4.5.3 Deliver

L'algorithme 10 décrit en détail le module Deliver pour la réplication partielle. Le Deliver lit en permanence la file d'exécution contenant les transactions prêtes à être exécutées par le module Refresher (ligne 3). Dans le cas où le noeud possède toutes les copies nécessaires à une transaction T (ligne 4), la transaction est démarrée sur le SGBD (ligne 6). Puis, on récupère l'identifiant local ($local_T$) de la transaction (ligne 7) afin de pouvoir associer les jeux d'écritures à la transaction (ligne 8). À ce moment, le Deliver connaît l'identifiant global ($global_T$) de la transaction, il correspond au nom du noeud d'origine et à l'estampille de la transaction. On associe alors l'identifiant global d'une transaction à son identifiant local (dans $liste_T$) afin que le Log Monitor puisse à partir du jeu d'écritures (qui contient l'identifiant local) récupérer l'identifiant global (voir Section 3.4.5.1). Et finalement, le Deliver exécute les opérations de T (ligne 9).

Si le noeud n'est pas en mesure d'exécuter la transaction (ligne 14), alors il la démarrée mais sans soumettre aucune opération (ligne 15). Puis, la transaction est mise en attente (ligne 16). À la réception d'un jeu d'écritures ws_T (ligne 19), le Deliver est réveillé et le contenu de la transaction en attente est remplacé par le contenu des jeux d'écritures (ligne 20 à 24).

3.5 Preuves

Dans cette section, nous fournissons des preuves pour les deux nouvelles configurations supportées par nos algorithmes de réplication totale et partielle. Nous montrons également que les algorithmes de rafraîchissement sont corrects en garantissant qu'il existe un ordre total d'exécution des transactions sur tous les noeuds. Finalement, nous prouvons que malgré le fait que les

Algorithme 9 – Module Log Monitor pour la réplication partielle

Entrées:

$o_T(id)$: opération de la transaction T identifiée par son id local
$liste_T(local_T, global_T)$: couple (id local, id global) des transactions en cours d'exécution sur le noeud

Sorties:

file de propagation

Variables:

$liste_{op}(id)$: une liste des opérations des transactions identifié par leur identifiant local
$global_T$: identifiant global de la transaction T

1: **début**
2: **boucler**
3: **sur** la réception d'une opération $o_T(id)$ de la transaction T identifiée par son id local **faire**
4: **si** $o_T(id)$ = *BEGIN* **alors**
5: ajouter une entrée à $liste_{op}$ avec id comme clé
6: **sinon si** $o_T(id)$ = (*COMMIT* | *ABORT*) **alors**
7: ajouter l'opération o_T à $liste_{op}(id)$
8: enlever et récupérer $global_T$ dans $liste_T$ en fonction de id
9: **si** $global_T$ existe **alors**
10: **si** T est locale **alors**
11: soumettre $liste_{op}(id)$ à la file de propagation
12: **fin si**
13: journaliser $liste_{op}(id)$
14: **fin si**
15: retirer $liste_{op}(id)$
16: **sinon**
17: ajouter l'opération o_T à $liste_{op}(id)$
18: **fin si**
19: **fin sur**
20: **fin boucler**
21: **fin**

Algorithme 10 – Module Deliver pour la réplication partielle

Entrées:
file des jeux d'écritures
file d'exécution
$liste_T$: couple (id local, id global) des transactions en cours d'exécution sur le noeud

Sorties:
SGBD
Replica Interface

Variables:
$local_T$: identifiant local de la transaction T
$global_T$: identifiant global de la transaction T
ws_T : les jeux d'écritures de la transaction T
T : une transaction
r_T : la réponse du SGBD à T

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T dans la file d'exécution **faire**
4: enlever T de la file d'exécution
5: **si** le noeud possède toutes les copies pour exécuter T **alors**
6: démarrer T sur le SGBD
7: récupérer $local_T$, l'identifiant local de T
8: créer une entrée ($local_T$, $global_T$) dans $liste_T$
9: soumettre T au SGBD
10: valider T
11: **si** T est locale au noeud **alors**
12: envoyer r_T au Replica Interface
13: **fin si**
14: **sinon**
15: démarrer T sur le SGBD
16: mettre T en attente
17: **fin si**
18: **fin sur**
19: **sur** l'arrivée d'un nouveau jeu d'écritures ws_T dans la file des jeux d'écritures **faire**
20: enlever ws_T de la file des jeux d'écritures
21: réveiller la transaction T en attente
22: filtrer ws_T pour ne garder que les opérations dont le noeud possède les copies
23: soumettre ws_T au SGBD
24: valider T
25: **fin sur**
26: **fin boucler**
27: **fin**

noeuds s'attendent entre eux (des noeuds peuvent attendre les transactions de rafraîchissement), il ne peut pas exister d'inter-blocage distribué.

3.5.1 Réplication totale

La Preuve 3.1 montre que l'algorithme de rafraîchissement est correct pour les configurations multimaîtres. Le problème de la réplication totale multimaître est que deux transactions mettant à jour la même donnée peuvent être soumises au même moment sur deux noeuds différents.

Lemme 3.1. *L'algorithme de rafraîchissement est correct pour les configurations multimaîtres.*

Démonstration. Considérons n'importe quel noeud N d'une configuration multimaître qui possède des copies multimaîtres. Soit T une transaction validée par le noeud N. Le module Propagator du noeud N propagera les opérations de T en utilisant une diffusion sûre. Par conséquent, tous les noeuds reçoivent la transaction. Comme (i) le message contenant l'estampille de n'importe quel transaction T est le dernier relié à cette transaction, et (ii) le service de diffusion sûre préserve l'ordre FIFO des messages, quand un noeud N' reçoit le message contenant l'estampille de T (au pire au temps $C + Max + \epsilon$), il a précédemment reçu toutes les opérations liées à T. Par conséquent, la transaction peut être validée quand toutes ses opérations sont finies et au plus tôt au temps $C + Max + \epsilon$. □

3.5.2 Réplication partielle

La Preuve 3.2 montre que l'algorithme de rafraîchissement est correct pour les configurations partielles. Un problème de la réplication partielle est que les noeuds qui ne possèdent pas toutes les copies sont obligés d'attendre une transaction de rafraîchissement du noeud d'origine.

Lemme 3.2. *L'algorithme de rafraîchissement est correct pour les configurations partielles.*

Démonstration. Considérons n'importe quel noeud N d'une configuration multimaître qui possède au moins une copie multimaître. Soit T une transaction validée par le noeud N, alors N est le noeud d'origine de T. Lorsque le message de mise à jour est reçu par n'importe quel noeud impliqué par l'exécution de la transaction, à partir du Lemme 3.1, on peut déterminer que la transaction T peut être validée au plus tôt au temps $C + Max + \epsilon$. Mais dans le cas où le noeud ne possède pas toutes les copies nécessaires à la transaction, T attend. Puis le noeud N produit et diffuse RT qui contient les jeux d'écritures associés à T à tous les noeuds cibles en attente. Les noeuds cibles en attente exécutent ainsi T en remplaçant le contenu de la transaction par son jeu d'écritures. Par conséquent, la transaction est toujours validée au plus tôt au temps $C + Max + \epsilon$. □

3.5.3 Ordre total

Les Preuves 3.3 et 3.4 montrent que tous les noeuds reçoivent les transactions en ordre total. La première preuve montre que l'ordre chronologique est respecté et la seconde que l'ordre total est garanti sur tous les noeuds.

Lemme 3.3. ***(Ordre Chronologique des Transaction).*** *L'algorithme de rafraîchissement garantit que, si T_1 et T_2 sont deux transactions démarrant leur exécution aux temps globales, respectivement t_1 et t_2, alors si t_2 - t_1 > ϵ, les estampilles C_2 pour T_2 et C_1 pour T_1 satisfont le critère suivant $C_2 > C_1$; n'importe quel noeud qui valide à la fois T_1 et T_2, les valide dans l'ordre donné par C_1 et C_2.*

Démonstration. Supposons que t_2 - $t_1 > \epsilon$. Même si l'horloge du noeud validant T_1 est en avance de ϵ par rapport à l'horloge du noeud qui valide T_2, nous avons $C_2 > C_1$. Nous supposons maintenant que $C_2 > C_1$ et nous considérons un noeud N qui valide d'abord T_1 puis T_2. D'après l'algorithme de rafraîchissement, T_2 n'est pas validée avant le temps local C_2 + Max + ϵ. À ce moment, si N valide T_2 avant T_1, cela signifie que N n'a pas reçu le message contenant T_1. Comme les horloges sont ϵ-synchronisées, ce message devrait avoir été envoyé avec un délai supérieur à Max. □

Lemme 3.4. ***(Ordre Total).*** *L'algorithme de rafraîchissement satisfait le critère d'ordre total pour n'importe quelle configuration.*

Démonstration. Si l'algorithme de rafraîchissement est correct (Lemme 3.1 et Lemme 3.2) et si les transactions sont exécutées dans l'ordre chronologique sur chaque noeud (Lemme 3.3), alors l'ordre total est assuré. □

3.5.4 Verrous mortels

La Preuve 3.5 montre que le Gestionnaire de Réplication n'est jamais bloqué par des verrous mortels. Avec la réplication partielle, les noeuds peuvent être mis en attente de transaction de rafraîchissement. Un verrou mortel peut apparaître lorsque deux noeuds attendent mutuellement la transaction de rafraîchissement. Nous prouvons ici qu'une telle situation n'est pas possible.

Lemme 3.5. ***(Verrous mortels).*** *L'algorithme de rafraîchissement garantit qu'aucun verrou mortel n'apparaît.*

Démonstration. Supposons une transaction T_1 qui a pour noeud d'origine N_1 et attend pour son jeu d'écritures sur le noeud N_2. Supposons également une transaction T_2 qui a pour noeud d'origine N_2 et attend pour son jeu d'écritures sur le noeud N_1. Un verrou mortel apparaît si et seulement si T_1 est exécutée avant T_2 sur N_2 et si T_2 est exécutée avant T_1 sur N_1. Par conséquent, l'ordre total n'est pas garanti. Ceci contredit le Lemme 3.4 où toutes les transactions sont toujours exécutées dans l'ordre chronologique sur tous les noeuds. □

3.6 Conclusion

Dans ce chapitre, nous avons présenté un nouvel algorithme de réplication que nous avons appelé Réplication Préventive. Nous nous sommes basés sur l'algorithme de réplication *Lazy-Master* proposé par Pacciti et al. [PMS01, PMS99]. Le principe de l'algorithme *Lazy-Master* est de soumettre les transactions dans un ordre total sur tous les noeuds en fonction de leur estampille d'arrivée. Pour accomplir ceci, il retarde l'exécution des transactions pour s'assurer qu'aucune transaction plus ancienne n'est en route pour le noeud. Cependant, l'algorithme *Lazy-Master* n'autorise que la mise à jour par copies primaires (où une copie ne peut être mise à jour que sur un seul site) et il impose une restriction sur le placement des données (le graphe de dépendances des noeuds ne doit pas former de cycles).

Pour résoudre ces problèmes, nous avons ajouté dans un premier temps le support de la réplication totale où tous les noeuds possèdent toutes les copies et sont des noeuds maîtres. Les copies multimaîtres sont alors rafraîchies en diffusant et en retardant les transactions sur tous les noeuds, y compris sur le noeud d'origine. En autorisant ainsi davantage de noeuds multimaîtres, nous supprimons le goulet d'étranglement que représente un seul noeud maître. Ce type de configuration n'est pas exploitable du fait des coûts trop importants qu'il introduit : tous les noeuds doivent exécuter toutes les transactions.

Dans un second temps nous avons donc introduit le support de la réplication partielle. Dans une configuration partielle, nous ne faisons de restrictions ni sur le mode d'accès aux copies (mode Copie Primaire ou Mise à jour Partout) ; ni sur le placement des données (un noeud peut posséder une partie seulement des copies et un ou plusieurs types de copie : Primaires, Multimaîtres et Secondaires). Dans le cas de la réplication, un noeud n'est pas forcément en mesure d'exécuté toutes les transactions car il ne possède pas toutes les copies nécessaire. L'algorithme place alors la transaction en attente des jeux d'écritures qui sont diffusé par le noeud d'origine. L'algorithme de réplication partielle autorise donc un plus grand nombre de configurations mais introduit la diffusion d'un message de rafraîchissement.

Pour chacune des versions de l'algorithme réplication préventive (totale et partielle), nous avons proposé : une architecture pour le gestionnaire de réplication, une description détaillée des algorithmes et les preuves que ces algorithmes garantissent la cohérence forte sans introduire d'inter-blocages.

Dans le chapitre suivant, nous nous attaquons aux performances de notre algorithme. En effet, si le support des configurations partielles améliore les performances en diminuant la charge de travail et réduit le nombre de messages échangés, l'algorithme de réplication préventive possède toujours deux goulots d'étranglement : le délai d'attente avant l'exécution d'une transaction et l'obligation d'exécuter séquentiellement les transactions sur le SGBD.

CHAPITRE 4

Optimisation des temps de réponse

4.1 Introduction

Les utilisateurs de grappes de PC recherchent avant tout la performance. Dans une grappe, les applications et les bases de données sont soumises à de fortes charges (*Bursty Workload*) où les transactions sont pour la plupart des mises à jour. Dans ce chapitre nous nous appliquons donc à optimiser le débit des transactions de l'algorithme de réplication préventive. Pour améliorer les performances de l'algorithme, nous nous attaquons à deux problèmes.

Le premier problème concerne le délai introduit par la réplication. Pour garantir un ordre total des transactions, la transaction doit attendre un délai Max + ϵ avant d'être envoyée au SGBD. Notre première optimisation a donc pour but de supprimer ce délai en exécutant les transactions de manière optimiste. Des exécutions optimistes impliquent cependant des abandons. Nous montrons alors que notre algorithme garantit toujours la cohérence forte malgré l'introduction d'abandons.

Le second problème est introduit par l'exécution séquentielle des transactions. Après ordonnancement, les transactions sont soumises une par une au SGBD. Une seule transaction peut être exécutée à la fois. Cette restriction est très pénalisante pour les performances, surtout sous des fortes charges où les débits des transactions sont élevés. Pour contourner ce problème, nous autorisons l'exécution en parallèle des transactions non conflictuelles. Nous devons cependant contrôler l'ordre de déclenchement et de validation des transactions pour ne pas introduire d'incohérences ou de lectures incohérentes.

Dans ce chapitre, nous présentons dans un premier temps une optimisation qui supprime le délai d'attente des transactions. Puis nous introduisons une modification de l'algorithme autorisant l'exécution en parallèle des transactions. Finalement, nous démontrons la validité de nos optimisations en prouvant que les algorithmes n'introduisent pas d'incohérences.

4.2 Élimination du délai d'attente

Pour ordonnancer les transactions dans un ordre total, l'algorithme de réplication préventive utilise les propriétés réseau. Il garantit qu'une transaction est en ordre total après un délai Max + ϵ où Max est le temps maximum mit par une transaction pour être diffusée et ϵ est la différence maximale entre deux horloges. La valeur Max correspond au temps le plus pessimiste de

diffusion d'un message, l'écart entre le temps moyen de diffusion (quelques millisecondes) et la valeur Max (quelques centaines de millisecondes) peut donc être important. Dans cette section nous présentons une optimisation qui élimine ce temps d'attente en exécutant les transactions optimistiquement.

4.2.1 Principe

Dans un réseau de grappe qui est typiquement rapide et fiable, la plupart des messages sont naturellement ordonnés. Seule une faible partie des messages est reçue dans un ordre différent de celui de l'ordre d'émission. Nous appelons message non-ordonné un message reçu en retard, c'est-à-dire qu'un message plus récent a déjà été reçu. En nous basant sur cette propriété, nous pouvons améliorer notre algorithme en soumettant une transaction pour exécution dès sa réception. Ainsi nous supprimons le délai qui précède la soumission des transactions. Nous avons cependant toujours besoin de garantir la cohérence forte. Pour atteindre cette cohérence, nous n'ordonnançons plus les transactions complètes mais uniquement leur validation. Une transaction ne peut donc valider qu'après le délai $Max + \epsilon$. Les transactions sont alors exécutées pendant leur ordonnancement et non plus après. Ainsi, dans la plupart des cas le délai ($Max + \epsilon$) est supprimé. Dans l'algorithme de réplication préventive présenté au Chapitre 3, le délai pour rafraîchir une copie après réception d'une transaction T, est décrit par la formule 4.1 où t est la durée d'exécution de T :

$$Max + \epsilon + t \tag{4.1}$$

Dorénavant une transaction T est ordonnée pendant son exécution. Ainsi le délai de rafraîchir une copie est décrit par la formule 4.2 :

$$maximun[(Max + \epsilon), t] \tag{4.2}$$

Dans la plupart des cas, t est plus grand que le délai $Max + \epsilon$. Par conséquent, cette simple optimisation améliore le débit des transactions comme nous le montrons au Chapitre .

L'exécution optimiste des transactions peut cependant entraîner des abandons. En effet, si une transaction est reçue non-ordonnée, alors toutes les transactions plus récentes déjà en exécution doivent être annulées. Ensuite on re-soumet les transactions en fonction de leur estampille. Notons qu'une transaction plus récente ne pourra jamais être validée car elle n'a pas été ordonnancée par le Refresher.

La Figure 4.1 présente une partie des composants nécessaires pour le fonctionnement de notre algorithme. Le Refresher lit les transactions en tête des files d'attente et garantit l'ordre chronologique en fonction de l'estampille de la transaction. Une fois qu'une transaction T est ordonnée, le Refresher notifie au Deliver que T est ordonnée et prête à être exécutée. Pendant ce temps, le Deliver lit les messages en-tête de la file d'exécution pour démarrer les transactions optimistiquement, l'une après l'autre, dans le SGBD local. Le Deliver valide uniquement une transaction lorsque le Refresher lui a notifié que la transaction est ordonnée.

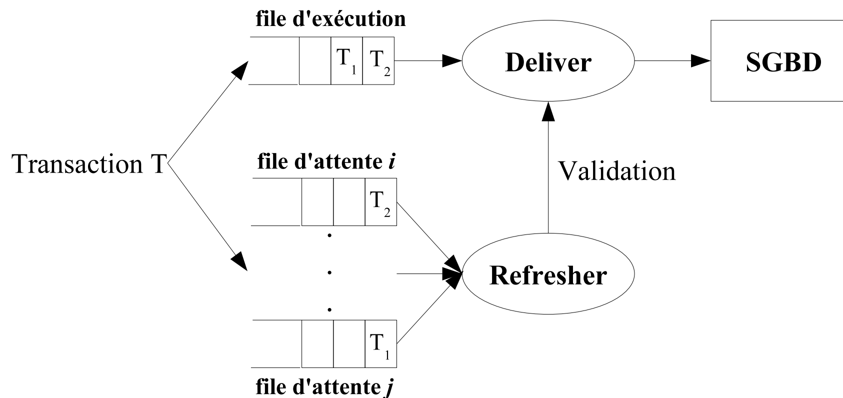


Figure 4.1 – Architecture de rafraîchissement avec suppression du délai d'attente

4.2.2 Exemple

Nous illustrons le fonctionnement de l'algorithme grâce à l'exemple de la Figure 4.1. Soit un noeud i possédant une copie maître de R. Le noeud i reçoit T_1 et T_2, deux transactions mettant à jour R, respectivement du noeud j avec une estampille $C_1 = 10$ et du noeud i avec une estampille $C_2 = 15$. T_1 et T_2 doivent être exécutées dans l'ordre chronologique, soit d'abord T_1 puis T_2. Voyons ce qui se passe lorsque les messages sont reçus non-ordonnés sur le noeud i. Dans notre exemple, T_2 est reçue avant T_1 sur le noeud i et immédiatement écrite dans la file d'exécution et dans la file d'attente correspondante.T_2 est alors soumise pour exécution par le Deliver mais doit attendre la décision du Refresher pour valider. Pendant ce temps, T_1 est reçue sur le noeud i, elle est également écrite dans la file d'exécution et dans la file d'attente correspondante au noeud j. Le Refresher détecte cependant qu'une transaction plus jeune, T_2, a déjà été soumise pour exécution avant T_1. T_2 est alors annulée puis redémarrée, entraînant sa ré-insertion dans la file d'exécution (après T_1). T_1 est ensuite soumise pour exécution par le Deliver et également élue pour être validée par le Refresher. À la fin de son exécution T_1 peut valider. Ensuite, T_2 est démarrée et élue pour être la prochaine transaction à valider. Ainsi, les transactions sont validées dans l'ordre de leur estampille, même si elles sont reçues non-ordonnées.

Avec cet exemple, nous pouvons voir une petite optimisation très simple qui limite le nombre d'abandon de notre nouvel algorithme. Au lieu d'ajouter les transactions en fin de la file d'exécution, elles sont insérées en fonction de leur estampille, la plus jeune étant placée en queue de la file. Ainsi, les transactions sont pré-ordonnancées. Cette optimisation n'est possible que pour les transactions qui n'ont pas encore été soumises pour exécution.

4.2.3 Algorithmes

Pour décrire l'algorithme de rafraîchissement avec élimination du délai d'attente, nous pouvons définir trois états différents pour une transaction T comme présenté dans la Figure 4.2 : *En attente*, *Validée* et *Abandonnée*. Quand une transaction T arrive sur le gestionnaire de réplication, son état est initialisé à *En attente*. Ensuite, lorsque le Refresher a ordonné la transaction et lorsque T peut être exécuté, l'état de la transaction T de *Validée*. Rappelons que dans une

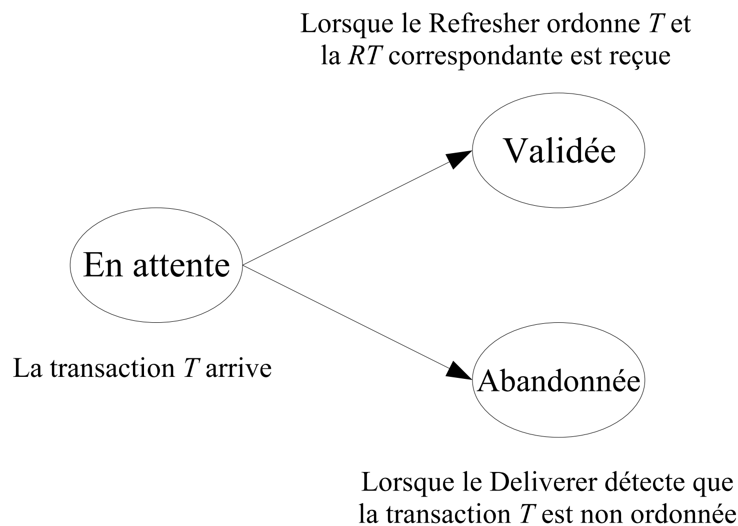


Figure 4.2 – Graphe de transition d'une transaction T

configuration partielle, une transaction ne peut être exécutée que lorsque le noeud détient toutes les copies nécessaires ou lorsqu'il reçoit la transaction de rafraîchissement correspondante. Finalement, lorsque le Deliver reçoit une autre transaction non-ordonnée (son estampille est plus petite que l'estampille de la transaction en cours d'exécution), l'état de la transaction est positionné sur *Abandonnée*.

Pour pouvoir implémenter cette optimisation, nous avons besoin de modifier trois des modules du gestionnaire de réplication : le Receiver, le Refresher et le Deliver.

4.2.3.1 Receiver

Le détail du Receiver est présenté dans l'Algorithme 11. L'algorithme du Receiver change très peu par rapport à l'algorithme présenté au Chapitre précédent. En plus d'ajouter à la file d'attente une nouvelle transaction reçue, le Receiver doit également ajouter la transaction à la file d'exécution (ligne 4 et 5).

4.2.3.2 Refresher

Le détail du Refresher est présenté dans l'Algorithme 12. Le Refresher sélectionne les prochaines transactions totalement ordonnées au temps d'exécution ($C + Max + \epsilon$). À l'étape 1, à l'arrivée d'une nouvelle transaction, le Refresher choisit la plus vieille transaction T en tête des files d'attente ($premier_T$) et calcule l'estampille d'exécution *delivery-time* de T (ligne 11). Ensuite, le Refresher initialise une horloge ($timer$) qui expire à *delivery-time*. À l'étape 2, lorsque $timer$ expire, le Refresher regarde toutes les transactions non abandonnées correspondantes à $elue_T$ (ligne 17 et 18) et positionne leur état à *Validée*. Nous verrons dans l'algorithme du Deliver (voir Section 4.2.3.3) qu'il peut y avoir plusieurs versions d'une transaction dans la file

Algorithme 11 – module Receiver avec élimination du délai d'attente

Sorties:
files d'attente q_1,... q_n file d'exécution

Variables:
T : une transaction
ws_T : les jeux d'écritures de la transaction T

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T du noeud n **faire**
4: ajouter T à la file d'exécution
5: ajouter T à la file d'attente correspondante q_n
6: **fin sur**
7: **sur** l'arrivée des jeux d'écritures ws_T de la transaction T **faire**
8: envoyer ws_T au Deliver
9: **fin sur**
10: **fin boucler**
11: **fin**

d'exécution mais qu'une seule a un état sur *En attente*. Les autres sont sur *Abandonnée*.

Algorithme 12 – Module Refresher avec élimination du délai d'attente

Entrées:
files d'attente q_1,... q_n
file d'exécution

Variables:
$curr_T$: transaction actuellement élue pour la prochaine exécution
$premier_T$: transaction avec la plus petite estampille dans les files d'attente
$running_T$: liste des transactions dans la file d'exécution
$timer$: compte à rebours local dont l'état est soit actif soit inactif

1: **début**
2: $curr_T \Leftarrow 0$
3: $timer.state \Leftarrow$ inactif
4: **boucler**
5: **Etape** 1**:**
6: **sur** l'arrivée d'une nouvelle transaction **ou lorsque** ($timer.state$ = actif **et** $timer.valeur$ = 0) **faire**
7: $premier_T \Leftarrow$ transaction avec l'estampille C la plus petite parmi les messages en tête des files d'attente

```
8:          si premier_T ≠ curr_T alors
9:            curr_T ⇐ premier_T
10:           calculer delivery-time(curr_T)
11:           timer.valeur ⇐ delivery-time(curr_T) - temps local
12:           timer.state ⇐ actif
13:         fin si
14:       fin sur
15:     Etape 2:
16:       si timer.state = actif et timer.valeur = 0 alors
17:         pour toutes les transactions running_T dans la file d'exécution tel que running_T =
            curr_T faire
18:           running_T.state ⇐ validée
19:         fin pour
20:         enlever curr_T de sa file d'attente
21:         timer.state ⇐ inactif
22:       fin si
23:   fin boucler
24: fin
```

4.2.3.3 Deliver

La Figure 13 décrit l'algorithme du Deliver avec l'optimisation de la suppression du délai d'attente. Le Deliver lit les transactions de la file d'exécution et les exécute. Si une transaction T est reçue non-ordonnée, le Deliver abandonne la transaction en cours d'exécution, $curr - T$, et exécute T suivi de la transaction abandonnée. Le Refresher valide une transaction lorsque le Refresher met son état à *Validée*. À l'étape 1, à la fin de l'exécution courante de $elue_T$, le Deliver valide ou annule la transaction en fonction de son état (*Validée* ou *Annulée*). Comme le Deliver n'a pas accès au gestionnaire de transaction du SGBD, il ne peut pas annuler directement les transactions et il doit attendre la fin de l'exécution de la transaction. À l'étape 2, le Deliver met l'état de la nouvelle transaction reçue ($nouvelle_T$) à *En attente* et vérifie si $nouvelle_T$ est reçue naturellement ordonnée. Dans le cas contraire, l'état de la transaction $elue_T$ est mis à *Abandonnée*. Comme le Deliver doit attendre la fin de l'exécution de la transaction pour l'annuler, une copie de $elue_T$ est réinsérée dans la file d'exécution avec un état à *En attente*. Ainsi, une transaction abandonnée sera ré exécutée depuis une copie de $elue_T$. À l'étape 3, le Deliver choisit la transaction en tête de la file d'exécution et l'exécute si le noeud possède toutes les copies nécessaires à la transaction. Autrement, le Deliver met la transaction en attente. Finalement, à l'étape 4, à l'arrivée d'une transaction de rafraîchissement $new_R T$, le Deliver remplace le contenu de la transaction en attente par le contenu de sa transaction de rafraîchissement correspondante. Ainsi, la transaction en attente peut être exécutée.

Algorithme 13 – Module Deliver avec élimination du délai d'attente

Entrées:
file des jeux d'écritures
file d'exécution
$liste_T$: couple (id local, id global) des transactions en cours d'exécution sur le noeud

Sorties:
SGBD
Replica Interface

Variables:
$local_T$: identifiant local de la transaction T
$global_T$: identifiant global de la transaction T
ws_T : les jeux d'écritures de la transaction T
T : une transaction
r_T : la réponse du SGBD à T

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T dans la file d'exécution **faire**
4: enlever T de la file d'exécution
5: **si** le noeud possède toutes les copies pour exécuter T **alors**
6: démarrer T sur le SGBD
7: récupérer $local_T$, l'identifiant local de T
8: créer une entrée ($local_T$, $global_T$) dans $liste_T$
9: soumettre T au SGBD
10: valider T
11: **si** T est locale au noeud **alors**
12: envoyer r_T au Replica Interface
13: **fin si**
14: **sinon**
15: démarrer T sur le SGBD
16: mettre T en attente
17: **fin si**
18: **fin sur**
19: **sur** l'arrivée d'un nouveau jeu d'écritures ws_T dans la file des jeux d'écritures **faire**
20: enlever ws_T de la file des jeux d'écritures
21: réveiller la transaction T en attente
22: filtrer ws_T pour ne garder que les opérations dont le noeud possède les copies
23: soumettre ws_T au SGBD
24: valider T
25: **fin sur**
26: **fin boucler**
27: **fin**

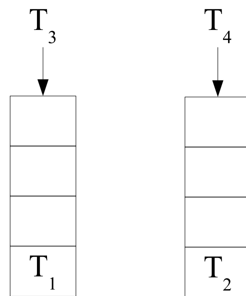


Figure 4.3 – Élection anticipée

4.3 Élection anticipée

En étudiant l'algorithme, nous nous sommes aperçus qu'une transaction peut être élue pour exécution avant l'expiration du délai d'attente.

Imaginons deux noeuds maîtres d'une même copie R, les noeuds possèdent donc deux files d'attente. Les deux noeuds maîtres envoient chacun une transaction au même instant. Cet exemple est représenté sur la Figure 4.3.

Étant donné que les messages sont diffusés avec une propriété FIFO par émetteur, si une transaction se présente, par exemple T_3 ou T_4, elle sera placée à la fin de sa file. La requête à exécuter restera T_1 ou T_2 (celle qui a la plus petite estampille). Ainsi, lorsque toutes les files possèdent au moins une transaction on est sûr que la prochaine transaction à exécuter est déjà présente.

Une nouvelle règle d'élection peut donc être ajoutée :

Une transaction est transférée de la file d'attente à la file d'exécution si son delivery-time *a expiré* ***ou si toutes les files d'attente ont au moins une transaction en attente et que l'estampille de cette transaction est la plus petite****.*

Cette règle permet d'éliminer un temps d'attente inutile d'une transaction. Et dans le cas particulier du mode Copie Primaire (où il y a un seul maître et donc une seule file d'attente), le temps d'attente est systématiquement supprimé. Avec une seule file, dès qu'une transaction est reçue par le Refresher, elle répond aux critères d'élections : toutes les files sont pleines et la transaction a l'estampille la plus petite. Elle est donc immédiatement transférée dans la file d'exécution.

4.4 Exécution en parallèle

Toujours pour améliorer le débit, nous introduisons la concurrence dans l'algorithme de rafraîchissement. Actuellement, le Receiver écrit les transactions directement dans la file d'exécution (optimistiquement) puis le Deliver lit le contenu de cette file dans le but d'exécuter les transactions. Afin de garantir la cohérence, les mêmes transactions sont également écrites dans

les files d'attente. Par conséquent, le Deliver extrait les transactions de la file d'exécution et les exécute une par une en ordre sériel. Ainsi, si le Receiver remplit la file d'exécution plus vite que le Deliver ne la vide, soit si le taux moyen d'arrivé des transactions est plus grand que le taux moyen d'exécution d'une transaction (c'est typiquement le cas avec des fortes charges - *bursty workload*), les temps de réponse augmentent exponentiellement et les performances se dégradent.

4.4.1 Principe

Afin d'améliorer les temps de réponse dans un environnement à fortes charges, nous proposons de d'exécuter les transactions en concurrence. En utilisant les propriétés d'isolation des systèmes de bases de données [Gar78], nous pouvons garantir à tout moment sur tous les noeuds les transactions voient un état cohérent de la base . Pour maintenir la cohérence forte sur tous les noeuds, nous garantissons que les transactions sont validées dans l'ordre de soumission. Ainsi, l'ordre total est toujours garanti.

Notons que sans accès au gestionnaire de concurrence (pour préserver l'autonomie), nous ne pouvons garantir que deux transactions conflictuelles concurrentes obtiennent un verrou dans le même ordre sur deux noeuds différents. Ainsi, nous ne déclenchons pas en concurrence des transactions conflictuelles.

Dans notre solution, la gestion de concurrence est faite en dehors de la base de données afin de préserver l'autonomie. Pour détecter que deux transactions sont conflictuelles, nous déterminons l'ensemble des éléments de la base de données accédés par la transaction. Si les ensembles de deux transactions n'ont pas d'intersections, alors les transactions ne sont pas conflictuelles et peuvent être déclenchées en concurrence. Par exemple, dans le Benchmark TPC-C, les paramètres des transactions nous permettent de définir les tuples utilisés par la transaction. Dans un environnement de grappe, cette solution est efficace car la plupart des transactions sont connues à l'avance. Notons que si un ensemble ne peut être déterminé, alors la transaction est considérée comme conflictuelle avec toutes les autres transactions.

Nous pouvons maintenant définir deux nouvelles conditions que doit vérifier le Deliver : (i) avant de démarrer une transaction et (ii) avant sa validation :

(i) Démarrer une transaction si et seulement si la transaction n'est pas en conflit avec des transactions déjà démarrées et non encore validées.

(ii) Valider une transaction si et seulement si aucune transaction plus ancienne n'est en cours d'exécution.

La condition *(i)* permet de ne pas exécuter deux transactions conflictuelles en même temps. La condition *(ii)* permet de valider les transactions dans leur ordre de soumission.

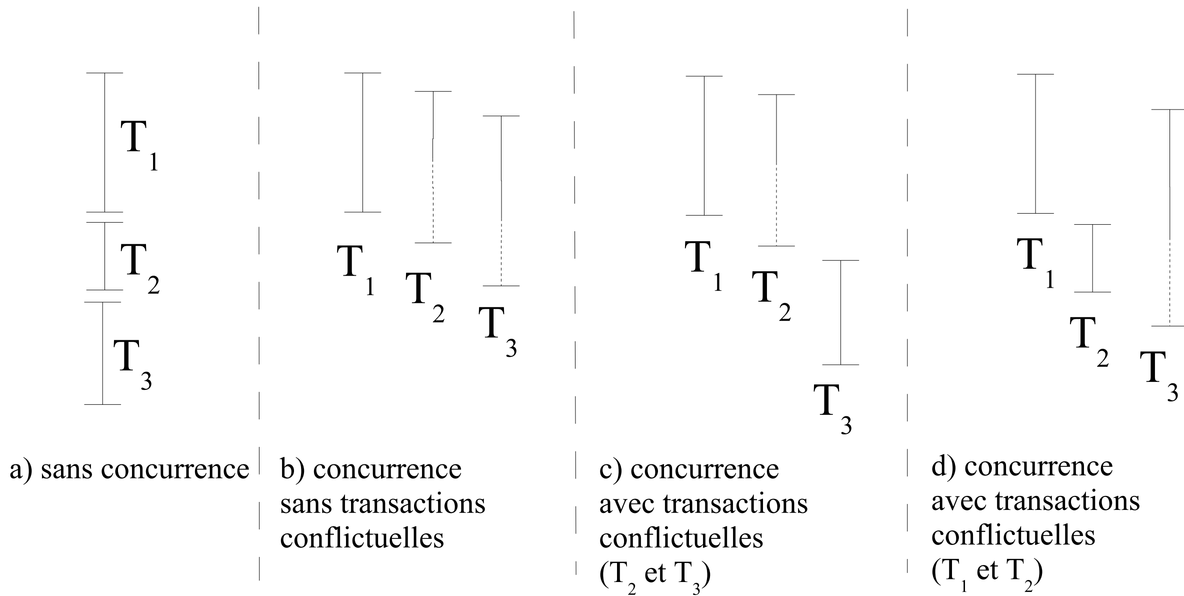


Figure 4.4 – Exemple d'exécutions concurrentes

4.4.2 Lectures incohérentes

Les transactions non conflictuelles sont également ordonnées pour éviter les lectures incohérentes. Si on autorise une exécution en parallèle des transactions non conflictuelles, le résultat d'une requête de lecture dépendra du noeud sur lequel elle a été soumise. Rappelons d'abord que la cohérence forte implique que pour 2 noeuds N_1 et N_2 possédant les mêmes copies, une vue E de la base sur noeud N_1 a été ou sera vue sur le noeud N_2.

Illustrons le problème des lectures incohérentes par un exemple. Soit les deux noeuds identiques N_1 et N_2 recevant deux transactions T_1 et T_2 : d'abord T_1 puis T_2 sur N_1 et T_2 puis T_1 sur N_2. T_1 est donc exécutée et validée sur N_1 et même chose pour T_2 sur N_2. Le résultat est différent selon le noeud où une requête de lecture est soumise. La lecture est donc incohérente.

En ordonnant l'ordre de validation de toutes transactions, la cohérence forte est garantie et les lectures incohérentes sont impossibles. Au pire, on peut obtenir des requêtes de lecture obsolètes lorsque la requête est soumise à un noeud ayant exécuté moins de transactions (parce qu'il est plus chargé).

4.4.3 Exemple

La Figure 4.4 montre des exemples d'exécutions concurrentes des transactions T_1, T_2 et T_3. La Figure 4.4a illustre le cas où les transactions sont exécutées séquentiellement, ce qui est équivalent au cas où toutes les transactions sont en conflits. Les Figures 4.4b, 4.4c et 4.4d montre des exécutions parallèles selon les conflits des transactions T_1, T_2 et T_3. Dans la Figure 4.4b et 4.4c, la transaction T_2 finit avant la transaction T_1 mais attend la validation de T_1 avant elle-même de valider (l'attente est représentée par la partie du trait en pointillée). Dans la Figure

4.4b, aucune transaction n'est conflictuelle, elles peuvent alors être exécutées en concurrence. D'un autre coté, dans la Figure 4.4c, T_2 est en conflit avec T_3 qui doit alors attendre la fin de T_2 avant de démarrer. Finalement, dans la Figure 4.4d, T_1 et T_2 sont en conflit. T_2 ne peut pas démarrer avant la validation de T_1. Par contre T_3 n'étant ni en conflit avec T_1 et T_2, elle peut démarrer avant T_2 car les propriétés d'isolation nous garantissent que les effets de la transaction ne seront pas vues avant sa validation. Mais pour ne pas entraîner d'incohérence, T_3 doit attendre la fin de T_1 et T_2 pour valider.

4.4.4 Algorithmes

Algorithme 14 – Module Deliver avec gestion de la concurrence

Entrées:
file des jeux d'écritures
file d'exécution
$liste_T$: couple (id local, id global) des transactions en cours d'exécution sur le noeud
Sorties:
SGBD
Replica Interface
Variables:
$local_T$: identifiant local de la transaction T
$global_T$: identifiant global de la transaction T
ws_T : les jeux d'écritures de la transaction T
T : une transaction
$statut_T$: statut de la transaction T, soit *En attente* soit *Validée*
r_T : la réponse du SGBD à T
en-cours$_T$: liste des transactions en cours d'exécution
$E(T)$: Estimation de l'ensemble des données touchées par la transaction T

1: **début**
2: **boucler**
3: **sur** l'arrivée d'une nouvelle transaction T dans la file d'exécution **ou** sur la fin d'exécution d'une transaction **faire**
4: **si** une transaction T est finie **alors**
5: $statut_T \Leftarrow$ *Validée*
6: **fin si**
7: $T \Leftarrow Premier($*en-cours*$_T)$
8: **tant que** $T \neq 0$ **faire**
9: **si** $statut_T$ = *Validée* **alors**
10: valider T
11: **si** T est locale au noeud **alors**

12: envoyer r_T au Replica Interface
13: **fin si**
14: enlever T de *en-cours*$_T$
15: $T \Leftarrow$ transaction en tête de *en-cours*$_T$)
16: **sinon**
17: $T \Leftarrow 0$
18: **fin si**
19: **fin tant que**
20: **pour** toutes les transactions T dans la file d'exécution en commençant par la tête **faire**
21: **si** $E(T) \cap E($*en-cours*$_T) = 0$ **alors**
22: enlever T de la file d'exécution
23: ajouter T à la liste $en - cours_T$
24: $statut_T \Leftarrow$ *En attente*
25: **si** le noeud possède toutes les copies pour exécuter T **alors**
26: démarrer T sur le SGBD
27: récupérer $local_T$, l'identifiant local de T
28: créer une entrée ($local_T$, $global_T$) dans $liste_T$
29: soumettre T au SGBD
30: **sinon**
31: mettre T en attente
32: **fin si**
33: **fin si**
34: **fin pour**
35: **fin sur**
36: **sur** l'arrivée d'un nouveau jeu d'écritures ws_T dans la file des jeux d'écritures **faire**
37: enlever ws_T de la file des jeux d'écritures
38: démarrer T sur le SGBD
39: filtrer ws_T pour ne garder que les opérations dont le noeud possède les copies
40: soumettre ws_T au SGBD
41: **fin sur**
42: **fin boucler**
43: **fin**

L'algorithme 14 montre le détail du module Deliver pour la gestion de la concurrence. C'est le seul module qui doit être modifié pour utiliser cette optimisation. Pour des raisons de lisibilité, nous avons repris l'algorithme du module Deliver pour la réplication partiel (voir Chapitre 3) sans autres optimisations. Cependant, la gestion de la concurrence est totalement compatible avec les autres optimisations présentées dans ce chapitre. Dans cet algorithme, à la fin de l'exécution d'une transaction ou sur l'arrivée d'une nouvelle transaction (ligne 3), on valide les transactions dans l'ordre de la file d'exécution (*en-cours*$_T$) si ces transactions ont fini de s'exécuter (ligne 7 à 19). Puis on démarre toutes les transactions non conflictuelles (ligne 20 à 34), une transaction peut démarrer si l'ensemble des données qu'elle touche est distinct de l'ensemble des transactions déjà démarrées (ligne 21). Finalement à la réception d'un jeu

d'écritures, on peut terminer la transaction (ligne 36 à 41).

4.5 Preuves

Dans cette section, nous prouvons que les optimisations que nous avons proposées n'introduisent pas d'incohérence dans le système et que la cohérence forte est garantie.

4.5.1 Élimination du délai d'attente

La preuve 4.1 montre que l'exécution optimiste des transactions pour éliminer le délai $Max + \epsilon$ n'introduit pas d'incohérence malgré l'exécution optimiste des transactions et les abondons que cela entraîne.

Lemme 4.1. *L'élimination du délai d'attente $Max + \epsilon$ n'introduit pas d'incohérences.*

Démonstration. Soit T_1 et T_2 deux transactions quelconques avec des estampilles C_1 et C_2. Si T_1 est plus ancienne que T_2 ($C_1 < C_2$) et que T_2 est reçue sur le noeud i avant T_1, alors T_2 est exécutée optimistiquement. Cependant T_2 ne peut pas être validée avant $C_2 + Max + \epsilon$. Comme T_1 est reçue sur le noeud i au plus tard à $C_1 + Max + \epsilon$, alors T_1 est reçue avant la validation de T_2 ($C_1 + Max + \epsilon < C_2 + Max + \epsilon$). Par conséquent, T_2 est abandonnée et les deux transactions sont insérées dans la file d'exécution, exécutées et validées en fonction de leur estampille. Au final, T_1 est exécutée avant T_2, et la cohérence forte est garantie même dans le cas de messages non ordonnés. □

4.5.2 Élection anticipée des transactions

La Preuve 4.3 montre que l'ordre d'élection des transactions est équivalent avec ou sans l'optimisation d'élection anticipée.

Lemme 4.2. *L'élection anticipée des transactions n'introduit pas d'incohérences.*

Démonstration. Soit une copie R possédant n maîtres [R_0, R_1, ..., R_{n-1}]. Le noeud i possédant la copie R a donc n file d'attente [q_0, q_1, ..., q_{n-1}]. Si chaque noeud maître envoie une transaction de mise à jour pour la copie R alors le noeud i reçoit n messages et chacun est placé en tête de sa file d'attente, avec des estampilles [C_0, C_1, ..., C_{n-1}]. Le noeud i élit alors le message avec la plus petite estampille C_x. Donc, si un nouveau message M est reçu du noeud j par le noeud i avec une estampille C_M, il sera placé en fin de la file d'attente q_j. Étant donné que les messages sont FIFO par émetteur, $C_M > C_j$ et $C_j > C_x$, alors $C_M > C_x$. Par conséquent, lorsque toutes les files contiennent au moins un message, l'arrivée d'un nouveau message ne change pas la transaction élue. On peut donc ordonner la transaction sans attendre la fin du délai $Max + \epsilon$. □

4.5.3 Exécution concurrente

La Preuve 4.3 montre que malgré l'exécution en parallèle des transactions, l'ordre d'exécution des transactions est la même sur tous les noeuds, garantissant ainsi la cohérence forte.

Lemme 4.3. *L'exécution parallèle des transactions ne modifie pas la cohérence forte entre les noeuds.*

Démonstration. Soit T_1 et T_2, deux transactions quelconques avec, respectivement, des estampilles C_1 et C_2 démarrant leur exécution aux temps t_1 et t_2 et valident aux temps c_1 et c_2. Dans le cas où T_1 et T_2 sont reçues non-ordonnées, les transactions sont abandonnées et ré exécutées dans l'ordre correct comme décrit dans le Lemme 4.1. Dans le cas où les transactions sont reçues correctement ordonnées. Si T_1 et T_2 sont conflictuelles, elles démarrent et valident l'une après l'autre en fonction de leur estampille. Par conséquent, si $C_1 < C_2$, alors $t_1 < c_1 < t_2 < c_2$. Si elles ne sont pas conflictuelles, T_2 peut démarrer avant la validation de T_1. Cependant, une transaction n'est jamais validée avant que toutes les transactions plus anciennes aient été elles aussivalidées. Si $C_1 < C_2$, alors on en déduit que $t_1 < t_2$ et $c_1 < c_2$. L'état de la base de données vue par la transaction avant son exécution et sa validation est donc le même sur tous les noeuds. Par conséquent,la cohérence forte est garantie. □

4.6 Conclusion

Afin de mieux supporter les applications à fortes charges où les transactions de mise à jour sont majoritaires, nous avons amélioré les points faibles de l'algorithme de réplication préventive.

Nous avons dans un premier temps éliminé le délai introduit par l'ordonnancement des transactions en les exécutant de manière optimiste dès leur réception dans le noeud et non plus après le délai $Max + \epsilon$. Si les transactions n'ont pas été exécutées dans l'ordre correct (celui de leurs estampilles), alors elles sont annulées et ré-exécutées après ordonnancement. Le nombre d'abandons reste faible car dans un réseau rapide et fiable les messages sont naturellement ordonnés. Malgré cette optimisation la cohérence forte est garantie car nous retardons la validation des transactions (et non plus la totalité de la transaction) exécutées optimistiquement. Les transactions sont ordonnancées pendant leur exécution et non plus avant, supprimant ainsi les délais d'ordonnancement.

La deuxième optimisation concerne la soumission des transactions. Dans les algorithmes présentés au Chapitre 3, les transactions sont soumises au SGBD une par une pour garantir la cohérence. Le module de soumission représente donc un goulet d'étranglement dans le cas où le temps moyen d'arrivée des transactions est supérieur au temps moyen d'exécution d'une transaction. Pour supprimer ce problème, nous avons autorisé l'exécution parallèle des transactions non conflictuelles. Cependant, pour garantir la cohérence des données, nous ordonnançons toujours le démarrage et la validation des transactions, ceci afin de garantir que toutes les transactions soient exécutées dans le même ordre sur tous les noeuds malgré le parallélisme. Nous

prouvons que l'exécution en parallèle des transactions est équivalente à une exécution séquentielle.

Dans le chapitre suivant nous présentons RepDB* qui est le prototype issue de l'implémentation de l'algorithme de réplication préventive.

CHAPITRE 5

Le prototype RepDB*

5.1 Introduction

Pour valider l'algorithme de réplication préventive, nous avons implémenté plusieurs versions du gestionnaire de réplication. Le prototype *RepDB** [ATL05] représente la plus abouti de ces implémentations. Aujourd'hui *RepDB** est un logiciel libre ayant fait l'objet d'un dépôt à l'Agence Pour la Protection des logiciels (APP). Bien que RepDB* soit prévu pour supporter plusieurs SGBD, pour des raisons de temps, seul la version pour PostgreSQL existe. En effet, notre algorithme a besoin de lire les journaux du SGBD et les techniques de lectures de journaux ne sont pas portables d'un système à l'autre. Nous avons donc décider de nous focaliser sur un système ouvert et largement utilisé: PotgreSQL.

Dans ce chapitre, nous présentons dans un premier temps la plate forme logicielle utilisée pour implémenter RepDB*. Puis nous montrons sa mise en oeuvre.

5.2 Plateforme

Dans cette section nous décrivons tous les produits utilisés pour implémenter RepDB*:

- Le langage Java de Sun Microsystems et notamment le JDK 1.4 [Sun03] (le prototype est compatible avec le JDK 1.5) ;
- La boîte à outils Spread [Spr06, AS98] qui offre un bus de communication pour les applications distribuées ;
- La librairie *Log4j* issue du projet jakarta d'Apache [Apa05] qui gère tout ce qui concerne l'archivage et qui s'avère très important pour les tests de performance ;
- Le paquetage pour la gestion de la concurrence dans java de Doug Lea [Lea06] ;
- Un serveur JDBC pour standardiser les connexions au gestionnaire de réplication, ce serveur est une version modifiée de RmiJdbc [Obj04].

5.2.1 Langage

Pour implémenter le prototype RepDB*, nous avons choisi d'utiliser Java qui est à la fois un langage de programmation et une plateforme d'exécution. Le langage Java a la particularité principale d'être portable sur plusieurs systèmes d'exploitation. C'est sa plateforme qui garantit alors la portabilité des applications développées en Java. Dans notre implémentation, nous avons choisi de modéliser chaque module de l'architecture du Gestionnaire de réplication par

un processus léger. Ainsi chaque module continue à traiter les données qui lui sont transmisses même si l'un des modules est en attente.

Pour la communication entre processus, nous avons décidé d'utiliser des objets partagés. En reprenant la Figure 4.1 du Chapitre 4, nous pouvons voir que l'architecture contient de nombreuses files FIFO (les files d'attente et les files d'exécution). Nous avons généralisé ce mode de communication pour tous les modules. Ainsi quand le *ReplicaInterface* reçoit une transaction, il la stocke dans une file, et c'est le *Propagator* qui retire les messages de cette file. Pour garantir une bonne gestion de la concurrence, nous utilisons le package de Doug Lea, *util.concurrent*, qui offre tous les outils nécessaires pour créer les variables partagées dont nous avons besoin. Il offre notamment une classe qui définit une file FIFO avec une fonction *take()* qui endort le thread appelant tant que la file est vide. Une autre fonction *take(long t)* endort le processus pendant une durée de *t* millisecondes au maximum si aucun message n'est présent dans la file.

5.2.2 La couche réseau

Nous avons vu que la couche réseau est un élément important de notre algorithme de réplication. Spread [PS98, Spr06] est une boîte à outils en logiciel libre qui fournissant un service de messagerie à haute performance résistant aux fautes pour les réseaux locaux et à large échelle. Spread a été créé et développé par le CNDS (Center for Networkingand Distributed Systems) à l'université de Johns Hopkins et par le Spread Concepts LLC. Les fonctions de Spread possèdent un bus de message unifié pour les applications distribuées. Elles sont tout à fait adaptées à la diffusion au niveau applicatif, à la communication de groupe et la communication point à point. Les services de Spread s'étendent notamment des messages fiables aux messages totalement ordonnés avec garanties de délivrance.

Spread est utilisé dans beaucoup d'applications ayant besoin d'une haute fiabilité, une haute performance et une communication robuste entre les membres. La boîte à outils est conçue pour répondre aux aspects difficiles des réseaux asynchrones et permet la construction d'applications distribuées fiables passant à l'échelle.

Spread consiste en une librairie utilisée par les applications utilisatrices, un démon démarré sur chaque noeud et un ensemble d'utilitaire et de programmes de tests. Voici quelques-uns des services et des avantages fournis par Spread:

- Des services de messagerie et de communications fiables et qui passent à l'échelle ;
- Une API très simple mais très puissante qui simplifie la construction d'architectures distribuées ;
- Une facilité d'utilisation, de déploiement et de maintenance ;
- Le support de plusieurs milliers de groupes avec différents groupes de membres.
- Le support de la fiabilité des messages malgré les pannes des noeuds, des processus, du partitionnement et de la fusion du réseau ;
- Différentes garanties de fiabilités, d'ordonnancements et stabilités pour les messages ;
- Des algorithmes entièrement distribués sans point de pannes central.

Type de message	**Ordonnancement**	**Sûreté**
UNRELIABLE_MESS	Non	Non
RELIABLE_MESS	Non	Oui
FIFO_MESS	FIFO par émetteur	Oui
CAUSAL_MESS	Ordre causal	Oui
AGREED_MESS	Ordre total (cohérent sans ordre causal)	Oui
SAFE_MESS	Ordre total	Oui

Table 5.1 – Les différentes méthodes de diffusions des messages dans *Spread*

En plus d'être la boîte à outils standard pour les applications distribuées, Spread est particulièrement adapté à notre algorithme. Comme le montre la liste des méthodes de diffusion des messages de Spread dans le Tableau 5.1, il contient un mode de diffusion FIFO par émetteur fiable dont notre algorithme a besoin. L'utilisation des groupes permet de modéliser les copies. Un groupe étant une copie, et un noeud possédant cette copie (primaire ou secondaire) est un membre du groupe. Un noeud appartient à autant de groupes qu'il détient de copies. De plus, la communication de groupes permet le contrôle des pannes. En effet, après la détection du départ d'un membre d'un groupe (que ce soit volontaire ou du à une panne), le démon Spread en avertit le gestionnaire de réplication.

L'un des rares problèmes de Spread est qu'il nécessite un démon sur chaque noeud du système pour gérer la communication de groupes entres les applications. Ceci introduit une difficulté dans le déploiement et l'installation du prototype, il faut installer deux applications: RepDB* et Spread. De plus, pour des raisons de performances, le démon a été écrit en C et non en Java (même si l'API pour le client existe en C et en Java) et est disponible pour un nombre restreint de systèmes: Linux et Windows XP/NT.

5.2.3 Gestion des journaux

Log4j est un projet *Jakarta* d'*Apache* (http://jakarta.apache.org/log4j/), il permet de gérer l'archivage de projet java en minimisant le code. Le comportement de l'archivage se contrôle grâce à un fichier de configuration en XML. Il est donc inutile de recompiler le fichier binaire pour modifier l'archivage.

Le format de sortie des journaux peut être multiple: console, fichiers, base de données, serveurs distants Log4j, courrier électronique. . .Il est même possible de rajouter ses propres modules.

Chaque classe d'un projet peut avoir son ou ses propres modules de sortie (*appender*) si log4j est supporté par la classe. De plus chaque message de journalisation possède un niveau. L'importance d'un message est définie par l'application (il existe quatre niveaux par défaut dans log4j: INFO, DEBUG, WARN, ERROR et FATAL). Ainsi, les messages sont filtrés grâce aux niveaux. La configuration d'une application est alors possible pour que les messages d'archivage d'une classe (INFO) soit envoyés vers un fichier et à sur la console, alors que les messages d'erreur (ERROR) sont envoyés par courrier électronique à l'administrateur.

Log4j a toujours été développé dans l'optique de ne pas détériorer les performances de l'application, apache annonce donc un temps moyen de 5 nanosecondes pour déterminer si une instruction log4j doit être archivé ou non et 21 microsecondes en moyenne pour exécuter cette instruction.

La Figure 5.1 présente un exemple de fichier de configuration de Log4J pour RepDB*. Dans cet exemple, nous déclarons trois modules d'archivage (*appender*): `Console`, qui écrit dans la console ; `Deliver`, qui écrit dans un fichier ; `Log`, qui écrit dans une base de données.

Puis nous déclarons que la classe `org.atlas.replication.Deliver` utilise les modules `Log` et `Deliver` quand le message a un niveau supérieur ou égale à DEBUG. Et enfin, la classe root (qui correspond en fait à toutes les classes de l'application) utilise le module `Console` pour tous les niveaux.

5.2.4 Interface de communications avec les SGBD

Les interfaces de communications dans RepDB* sont au nombre de trois: entre le client et le gestionnaire de réplication (module Replica Interface), entre le gestionnaire de réplication et le SGBD (module Deliver et DeliverRead) et entre le journal du SGBD et le gestionnaire de réplication (module Log Monitor). Toutes les interfaces de communications sont représentées sur la Figure 5.2.

Comme nous considérons les SGBD comme des boîtes noires, pour communiquer avec les SGBD, nous avons choisi d'utiliser JBDC (*Java DataBase Connectivity*). La technologie JDBC est une API fournie avec Java (depuis sa version 1.1) permettant de se connecter à des bases de données, c'est-à-dire que JDBC constitue un ensemble de classes permettant de développer des applications capables de se connecter à des serveurs de bases de données (SGBD).

L'API JDBC a été développée de telle façon à permettre à un programme de se connecter à n'importe quelle base de données en utilisant la même syntaxe, ceci signifie que l'API JDBC est indépendante du SGBD.

De plus, JDBC bénéficie des avantages de Java, dont la portabilité du code, ce qui lui vaut en plus d'être indépendant de la base de données d'être indépendant de la plate-forme sur laquelle elle s'exécute.

5.2.4.1 Client / Gestionnaire de réplication

Pour que le client se connecte au gestionnaire de réplication (module Replica Interface), nous avons donc choisi JDBC. Pour l'écriture du pilote JDBC, nous nous sommes basés sur le projet *RmiJDBC*. Ce projet consiste en gros à programmer un *serveur proxy JDBC*. Ainsi chaque fonction JDBC appelle une fonction serveur.

Pour lancer une transaction, il est nécessaire de passer par trois étapes :
- établissement d'une connexion
- création d'un transaction

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE log4j:configuration SYSTEM 'log4j.dtd'>

<log4j:configuration xmlns:log4j='http://jakarta.apache.org/log4j/'>
 <appender name='Console' class='org.apache.log4j.ConsoleAppender'>
  <layout class='org.apache.log4j.PatternLayout'>
   <param name='ConversionPattern'
          value='%p [%t] %c{2} (%M:%L) - %m%n'/>
  </layout>
 </appender>

 <appender name='Deliver' class='org.apache.log4j.FileAppender'>
  <param name='File' value='deliver.log'/>
  <param name='Append' value='false'/>
  <layout class='org.apache.log4j.PatternLayout'>
    <param name='ConversionPattern'
          value='%p [%t] %c{2} (%M:%L) - %m%n'/>
  </layout>
 </appender>

 <appender name='Log' class='org.apache.log4j.jdbc.JDBCAppender'>
  <param name='user' value='rep_ln'/>
  <param name='password' value='rep_ln'/>
  <param name='URL'
         value='jdbc:oracle:thin:@node0:1521:ln1a'/>
   <layout class='org.atlas.repdb.log4j.ReplicationPatternLayout'>
    <param name='ConversionPattern'
           value="INSERT INTO logReplication VALUES (
                  '%m{JEU}', '%m{NODE_NAME}', '%m{FROM_NODE}',
                  '%m{DATA_NODE}', '%m{DATA_TABLE}', '%m{ID}',
                  '%m{TS}', '%m{TS_PROPAGATOR}',
                  '%m{TS_RECEIVER}', '%m{TS_REFRESHER}',
                  '%m{TS_DELIVER}', '%m{TS_DELIVERED}',
                  '%m{SQL}', '%m{TYPE}')"/>
   </layout>
 </appender>

 <category name='org.atlas.repdb.replication.Deliver'
  additivity='false'>
  <priority value='debug'/>
  <appender-ref ref='Log'/>
  <appender-ref ref='Deliver'/>
 </category>

 <root>
  <priority value='info'/>
  <appender-ref ref='Console'/>
 </root>
</log4j:configuration>
```

Figure 5.1 – Fichier de configuration des journaux (Log4J)

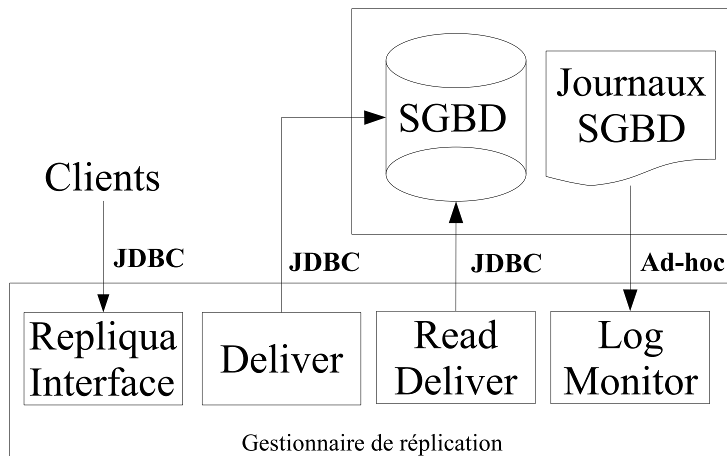


Figure 5.2 – Interfaces de communication entre le client, le SGBD et le Gestionnaire de réplication

- exécution de la requête

L'établissement d'une connexion est assez longue, aussi est-il intéressant de gérer une file de connexions. Ainsi au démarrage, le prototype ouvre un nombre prédéfini de connexions puis si ce nombre se révèle être insuffisant, d'autres sont ouvertes. Et lorsqu'une connexion n'est plus utilisée, elle est mise en mode réutilisable au lieu d'être fermée.

Afin de ne pas faire d'appels RMI intempestifs, on peut agréger les trois étapes au niveau du client en une seule ce qui permettra de ne faire qu'un appel RMI.

Les résultats retournés par l'exécution des requêtes sont également agrégés pour retourner en une seule fois la réponse au client..

5.2.4.2 Gestionnaire de réplication / SGBD

Pour que le gestionnaire de réplication exécute les transactions sur le SGBD, nous utilisons également JDBC. La plupart des SGBD possède une interface JDBC, il donc inutile d'en implémenter une pour chaque SGBD.

Cependant, nous avons vu dans les chapitres précédents que les transactions de mises jours sont exécutées en utilisant le Deliver. Pour les transactions en lecture seule, nous avons ajouté un nouveau module dans l'architecture du gestionnaire de réplication. Il permet d'exécuter les requêtes des clients : le DeliverRead. Il récupère simplement les requêtes de lecture envoyées par les clients via le Replica Interface et les soumet au SGBD.

Comme le coût d'une connexion au SGBD est important, nous ouvrons un pool de connexions restant ouvert même après la validation des transactions. Ainsi pour la prochaine transaction, une connexion existe déjà et le coût d'ouverture d'une connexion est supprimé.

```
// Transaction originale
Begin;
Insert into PERSONNE(PRENOM, NOM) values ('Pierre', 'Dupont');
Update PERSONNE set AGE = 35 where NOM = 'Dupont';
Commit;

// Jeux d'écritures
I 570 PERSONNE 3 NOM PRENOM AGE NULL NULL NULL 'Dupont' 'Pierre' NULL
U 570 PERSONNE 3 NOM PRENOM AGE 'Dupont' 'Pierre' NULL 'Dupont' 'Pierre' '35'
C 570

// Transaction reconstruite
Begin;
Insert into PERSONNE(NOM, PRENOM, AGE)
            values ('Dupont', 'Pierre', '35');
Update PERSONNE set AGE = 35 where NOM = 'Dupont' and
                                   PRENOM = 'Pierre' and AGE = NULL;
Commit;
```

Figure 5.3 – Exemple de transactions reconstruite à partir du jeu d'écritures

5.2.4.3 Journaux du SGBD / Gestionnaire de réplication

La dernière interface permet de lire le journal du SGBD (module Log Monitor). Ici, aucune interface standard n'existe même si certaines techniques ont été mises au point [KR87, SKS86], le module Log Monitor est donc spécifique au SGBD. Le but est d'à partir du journal de recréer une transaction contenant les jeux d'écritures sous forme d'opérations SQL simples : INSERT, UPDATE et DELETE. Une opération ne doit affecter qu'un seul tuple, ce tuple étant spécifié grâce à une clause where qui reprend tous les attributs de la table. Par conséquent, toutes les tables doivent avoir des clés primaires. Faute de quoi, si dans une table sans clés primaire, un tuple doit être supprimé et qu'il existe plusieurs tuples avec les mêmes valeurs, alors plusieurs tuples seront supprimés au lieu d'un seul.

Pour notre prototype, nous avons modifié PostgreSQL 8 afin qu'il envoie un message (via le protocole UDP/IP) à chaque modification des données. Voici la structure d'un message :

$$operation, transaction_{ID}, table, nb_{attribut}, attribut_1, attribut_2, \ldots, attribut_n,$$
$$ancien_1, ancien_2, \ldots, ancien_n, nouveau_1, nouveau_2, \ldots, nouveau_n$$

Un message est composé d'un type d'opération, de l'identifiant local de la transaction, de la table mise à jour, du nombre d'attribut de la table, de la liste des noms des attributs, de la liste des anciennes valeur du tuple et de la liste des nouvelles valeurs du tuple. Les différentes opérations possibles sont : U pour une mise à jour (UPDATE), D pour une suppression (DELETE), I pour un ajout (INSERT), C pour une validation (COMMIT) et R pour un abandon (ROLLBACK).

La Figure 5.3 un exemple qui montre la transaction originale, les jeux d'écritures reçus par le Log Monitor et la transaction SQL reconstruite. On peut noter que la connaissance des clés primaires d'une table n'est pas nécessaire pour reconstruire les transactions.

5.3 Mise en oeuvre du prototype

Dans cette section, nous voyons comment installer et configurer RepDB*.

5.3.1 PostgreSQL

Voici la procédure pour modifier PostgreSQL 8 afin de lire les journaux. Pour le moment, la modification est valable pour la version de PostgreSQL 8.0.1.

- Télécharger PostgreSQL sur http://www.postgresql.org/
- Décompresser l'archive
- Remplacer les fichiers suivants par ceux de RepDB* :
  - posmaster.c > postgresql-8.0.1/src/backend/posmaster/
  - xact.c > postgresql-8.0.1/src/backend/access/transam/
  - execMain.c > postgresql-8.0.1/src/backend/executor/
- Compiler PostgreSQL
- Por démarrer PostgreSQL avec le support des jeux d'écritures, utiliser l'option -L suivie du port UDP/IP sur lequel sont envoyés les jeux d'écritures :
  - `postmaster -L 5555 -D pgdata &`
- Ne pas oublier de modifier la configuration pour que PostgreSQL accepte les connexions TCP/IP et donc les connexions JDBC.

5.3.2 Service réseau (Spread)

Voici maintenant la procédure pour installer et configurer Spread.

- Télécharger et installer Spread sur http://www.spread.org
- Créer un fichier de configuration (spread.conf) avec une entrée par noeud. Dans notre exemple le port utilisé par Spread pour communiquer est le 4683 :

```
Spread_Segment 192.168.1.255:4683 {
  node0          192.168.1.100
  node1          192.168.1.101
  node2          192.168.1.102
}
DebugFlags = { PRINT EXIT }
```

- Démarrer le démon Spread sur chaque noeud :
  - `spread -n node0 -c spread.conf &`

5.3.3 Synchronisation des horloges

Les horloges des noeuds ont besoin d'être synchronisées. L'utilisation d'un protocole comme NTP [Mil03] est recommandé.

5.3.4 RepDB*

Comme configuration, RepDB* ne demande que la création d'un fichier de configuration XML. Ce fichier est unique pour tous les noeuds et il contient :

- Les informations communes sur le service réseau : la valeur $Max + \epsilon$ et le préfix du groupe de communication (COMM)
- Le nombre d'écrivains et de lecteurs maximum (COMMOM)
- Et pour chaque noeud (NODE) :
  - le nom du noeud (attribut name de NODE)
  - le port utilisé par les clients JDBC pour se connecter à RepDB* (RMI)
  - le fichier de configuration Log4J (LOG)
  - la classe Java utilisée pour lire les journaux du SGBD (LOGMINER)
  - les infos sur Spread (COMM), le démon Spread doit être démarré en local (localhost)
  - les paramètres JDBC pour se connecter au SGBD local au noeud (CONNECT)
  - la liste des tables que possède le noeud en copie maître (MASTER)
  - la liste des tables que possède le noeud en copie secondaire (SLAVE)

La Figure 5.4 montre un exemple de fichier de configuration avec trois noeuds.

Avant de démarrer RepDB*, vous devez vous assurez que : le SGBD est démarré, la base est cohérente (RepDB* ne vérifie pas la cohérence de la base entre les autres noeuds) et que le démon Spread est démarré. Voici la commande pour démarrer RepDB* :

```
java org.atlas.repdb.replication.Replication NODE_NAME CONFIG_FILE
```

5.3.5 Les clients

Pour se connecter au gestionnaire de réplication, les clients doivent utiliser le pilote JDBC de RepDB* :

- Nom du driver : `"org.atlas.repdb.jdbc.Driver"`
- Chaîne de connexion : `"jdbc:repdb@node0://node0:4681/"`

De plus, les clients doivent aussi baliser leur transaction avec la liste des tables lues et la liste des tables mises à jour. Si la transaction n'est pas marquée, elle est malgré tout exécutée mais le gestionnaire de réplication considère qu'elle met à jour toute la base de données (même si c'est une transaction en lecture seule). Voici un exemple de transaction :

```
<WRITE>R,S</WRITE><READ>T<READ>
UPDATE R SET att2 = 1 WHERE att1 IN (SELECT att3 FROM T);
```

```
UPDATE S SET att2 = 1 WHERE att1 NOT IN (SELECT att3 FROM T);
```

5.4 Conclusion

Dans ce chapitre, nous avons décrit le prototype RepDB* qui implémente l'algorithme de réplication préventive avec toutes les optimisations. Nous nous sommes focalisés ici sur la partie technique du prototype. Nous avons décrit tous les composants logiciels qui forment RepDB* :

- Le gestionnaire de réplication écrit en Java avec un serveur JDBC (RmiJDBC) pour être utilisable par un grand nombre de clients ;
- Le service de bus réseau fourni par Spread pour une haute fiabilité, une haute performance et une communication robuste ;
- La librairie Log4J qui permet l'écriture de journaux sans modification du code du gestionnaire de réplication. Même si le prototype n'a pas encore de support pour la tolérance aux pannes, il contient tous les éléments nécessaires à son élaboration ;
- Une modification de PostgreSQL qui permet de lire efficacement les journaux et de transmettre les données au gestionnaire de réplication.

Il faut noter que la première version de RepDB* était implémentée entièrement à l'intérieur de PostgreSQL. La différence avec la version actuelle se trouve surtout au niveau des performances. La connexion au travers d'un serveur JDBC avant de soumettre la requête au SGBD augmente les temps de réponse. Nous avons abandonné le projet de développer RepDB* dans PostgreSQL car ce dernier est un projet conséquent et le coût de maintenance devenait important. Nous avons préféré considérer le SGBD comme une boîte noire et ainsi pouvoir fournir RepDB* pour plusieurs SGBD.

RepDB* est un prototype encore embryonnaire qui peut encore évoluer (notamment avec le support d'un plus grand nombre de SGBD). Mais depuis sa mise en domaine public en 2005, il a été téléchargé plus de 800 fois. Nous avons malheureusement peu de retour sur l'utilisation réelle qui en est faite.

Dans le chapitre suivant, nous présentons les performances de RepDB*.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<REPLICATION>
  <COMM max='100' group='rp_' />
  <COMMON nbDeliver='8' nbReader='8' />

  <NODE name='node0'>
    <RMI port='4681' />
    <LOG configFile='log4j.xml' />
    <LOGMINER class='org.atlas.repdb.replication.logMiner.PosgresLogMiner'>
           <ATTRIBUTE name='port' value='4685' />
    </LOGMINER>
    <COMM server='localhost' port='4683' />
    <CONNECT driver='org.postgresql.Driver'
             connect='jdbc:postgresql://node0:4684/base_name'
             login='user'
             password='pwd' />
    <MASTER>R</MASTER>
    <MASTER>S</MASTER>
  </NODE>

  <NODE name='node1'>
    <RMI port='4681' />
    <LOG configFile='log4j.xml' />
    <LOGMINER class='org.atlas.repdb.replication.logMiner.PosgresLogMiner'>
           <ATTRIBUTE name='port' value='4685' />
    </LOGMINER>
    <COMM server='localhost' port='4683' />
    <CONNECT driver='org.postgresql.Driver'
             connect='jdbc:postgresql://node1:4684/base_name'
             login='user'
             password='pwd' />
    <MASTER>R</MASTER>
  </NODE>

  <NODE name='node2'>
    <RMI port='4681' />
    <LOG configFile='log4j.xml' />
    <LOGMINER class='org.atlas.repdb.replication.logMiner.PosgresLogMiner'>
           <ATTRIBUTE name='port' value='4685' />
    </LOGMINER>
    <COMM server='localhost' port='4683' />
    <CONNECT driver='org.postgresql.Driver'
             connect='jdbc:postgresql://node2:4684/base_name'
             login='user'
             password='pwd' />
    <SLAVE>R</SLAVE>
    <SLAVE>S</SLAVE>
  </NODE>

</REPLICATION>
```

Figure 5.4 – Exemple de fichier de configuration RepDB*

CHAPITRE 6

Validation

6.1 Introduction

Dans ce chapitre, nous utilisons le prototype RepDB* décrit au chapitre précédent pour montrer les propriétés de l'algorithme de réplication préventive (voir Chapitre 3) avec ses optimisations (voir Chapitre 4). Nous voulons prouver que l'algorithme peut être utilisé sur une grappe à grande échelle sans pertes de performance. Nous voulons également comparer l'influence de la réplication partielle sur les performances par rapport à la réplication totale. Un point important dans les algorithmes de réplication asynchrone est de déterminer les différences entre les copies. Pour cela, nous présentons un nouveau critère de qualité : le degré de fraîcheur d'un système. Finalement, nous montrons que les deux optimisations (voir Chapitre 4) proposées ont une réelle utilité et améliorent les performances en cas de fortes charges.

Nous décrivons dans un premier temps l'implémentation de notre système, puis nous définissons deux bancs d'essais : un spécifiquement créé pour notre algorithme et un banc d'essai standard, TPC-C. Pour chacun d'entre eux, nous définissons leurs configurations (tables, placements des données), les transactions utilisées et les paramètres variant au cours des tests. Puis nous montrons les résultats de nos expérimentations . Finalement, nous concluons.

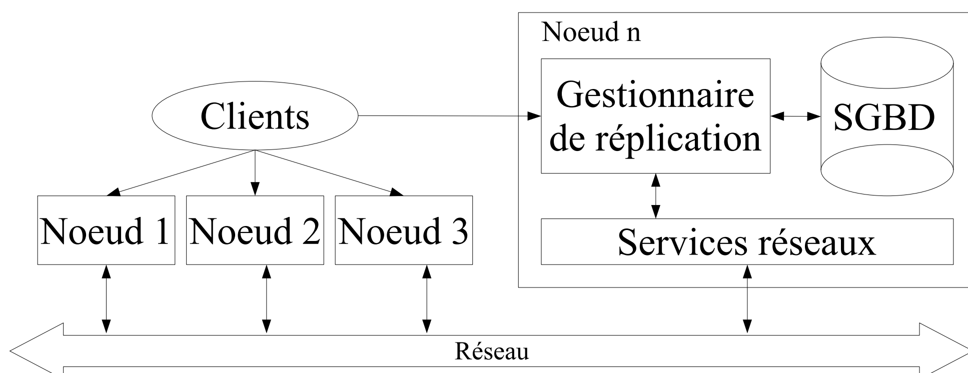


Figure 6.1 – Architecture de validation

6.2 Implémentation

Nous avons implémenté notre algorithme de réplication préventive dans le prototype RepDB* et nous l'avons utilisé sur une grappe de 64 noeuds (128 processeurs). Chaque noeud a deux processeurs Intel Xeon 2.4Ghz, une mémoire de 1 Go et un disque dur 40 Go. Les noeuds sont reliés par un réseau de 1 Gb/s. Nous utilisons Linux Mandrake 8.0 comme système d'exploitation, Java comme plateforme d'exécution et la boîte à outils Spread de CNDS pour fournir un bus FIFO fiable d'émission des messages parmi les noeuds de la grappe. Nous utilisons le logiciel PostgreSQL comme SGBD sur chaque noeud. Nous avons choisi PostgreSQL parce qu'il est très complet en terme de support des transactions et qu'il est livré en logiciel libre, ce qui permet de le modifier.

Comme décrite sur la Figure 6.1, notre implémentation est divisée en quatre modules : les clients, le gestionnaire de réplication, les services réseau et le serveur de bases de données. Le module client simule l'ensemble des clients, Il soumet des transactions aléatoirement à n'importe quel noeud de la grappe, en utilisant RMI-JDBC. Chaque noeud accueille un Système de Gestion de Bases de Données, une instance du gestionnaire de réplication et un démon qui fournit des services réseaux. Comme décrit dans le Chapitre 5, les composants du gestionnaire de réplication ont été écrit en Java en dehors du SGBD à l'exception du module de lecture des journaux (*Log Monitor*). Ce dernier a été implémenté directement dans PostgreSQL pour des raisons de facilités et de performances. De même bien que nous ne gérons pas les pannes dans notre prototype, nous avons implémenté tous les journaux nécessaires à la récupération du système. Finalement les services réseau sont fournis au travers de la boîte à outils Spread.

6.3 Modèle de performance

Pour valider l'algorithme de réplication préventive, nous avons utilisé deux bancs d'essais. Le premier a été développé spécifiquement pour tester les propriétés de l'algorithme. Pour le deuxième, nous avons choisi un banc d'essai standard, *TPC-C* [Raa93, Tra04], particulièrement adapté aux applications *OLTP* (On-Line Transactional Processing). Pour chaque banc d'essai, nous définissons les configurations supportées ainsi que les paramètres utilisés lors des tests.

6.3.1 Banc d'essai spécifique

Ce banc d'essai est relativement simple, il permet surtout de tester l'impact de la réplication partielle sur les performances du système. Nous avons donc volontairement défini peu de tables et peu de types de transactions différents pour bien comprendre le comportement de l'algorithme.

	Répliquée Totalement (FR)	**Répliquée Partiellement 1 (PR1)**				**Répliquée Partiellement 2 (PR2)**			
noeuds	*RST*	*RST*	*RS*	*RT*	*ST*	*RST*	*R*	*S*	*T*
4	4	1	1	1	1	1	1	1	1
8	8	2	2	2	2	2	2	2	2
16	16	4	4	4	4	4	4	4	4
24	24	6	6	6	6	6	6	6	6
32	32	8	8	8	8	8	8	8	8

Table 6.1 – Placement des données sur les noeuds pour le banc d'essai spécifique

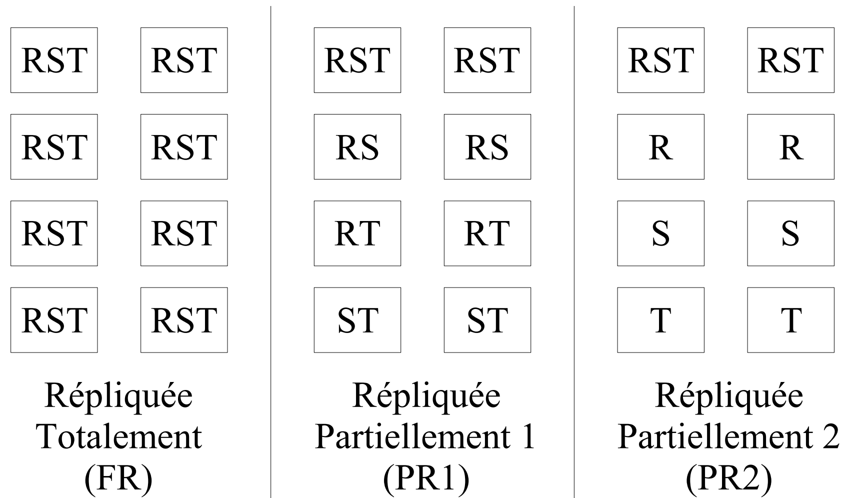


Figure 6.2 – Exemple de configuration pour 8 noeuds

Paramètre	**Définition**	**Valeurs**
/T/	Taille des transactions	5 ; 50
$Type_T$	Implique une ou plusieurs copies	SC (seule), MC (multiple)
ltr	Pourcentage de transactions longues	66%
λ	Temps moyen d'arrivée des transactions (charges)	forte : 100ms, faible : 3s
Nb-Noeuds	Nombre de noeuds	4, 8, 16, 24, 32
M	Nombre de transactions soumises lors du test	50
$Max + \epsilon$	Délai introduit avant de soumettre une transaction	200ms
Configuration	Réplication de R, S et T	*FR*, *PR1*, *PR2*

Table 6.2 – Paramètres du banc d'essai spécifique

6.3.1.1 Configurations et transactions

Notre banc d'essai définit trois configurations impliquant trois copies R, S et T dans cinq nombres différents de noeuds (4, 8, 16, 24 et 32). Ce choix permet de faire varier facilement le placement des copies tout en gardant une distribution équilibrée. Dans la configuration *Répliquée Totalement* (Fully Replicated - *FR*), tous les noeuds possèdent les trois copies $\{R, S, T\}$. Dans la configuration *Répliquée Partiellement 1* (Partially Replicated 1 - *PR1*), un quart des noeuds possède les trois copies $\{R, S, T\}$, un quart les deux copies $\{R, S\}$, un quart les deux copies $\{R, T\}$ et le dernier quart les deux copies $\{S, T\}$. Dans la configuration *Répliquée Partiellement 2* (Partially Replicated 2 - *PR2*), un quart des noeuds possèdent les trois copies $\{R, S, T\}$, un quart la copie $\{R\}$, un quart la copie $\{S\}$ et le dernier quart la copie $\{T\}$. Le Tableau 6.1 montre le nombre de noeuds possédant une combinaison de copies pour chaque configuration et pour les différents nombres de noeuds. Finalement, la Figure 6.2 montre les trois configurations obtenues avec huit noeuds (2ème ligne du Tableau 6.1).

Finalement, les copies R, S et T sont composées de deux attributs (le premier est utilisé comme clé primaire et le second comme valeur) et possèdent chacune 10000 tuples sans index.

6.3.1.2 Modèle de performance

Notre modèle de performance prend en compte le type des transactions, leurs tailles, leurs taux d'arrivée et le nombre de noeuds. Les transactions peuvent être de deux types ($Type_T$) : soit Copie Seule (Single Copy - SC) où la transaction ne met à jour qu'une seule copie ; soit Copie Multiple (Multiple Copy - MC) où la transaction peut mettre à jour plus d'une copie. Par conséquent avec trois copies R, S et T, nous avons trois transactions SC (sur $\{R\}$, $\{S\}$ ou $\{T\}$) et sept transactions MC (sur $\{R\}$, $\{S\}$, $\{T\}$, $\{R, S\}$, $\{R, T\}$, $\{S, T\}$ ou $\{R, S, T\}$). Nous considérons également le taux d'arrivée des transactions (noté λ) qui correspond au délai moyen entre deux transactions. Nous définissons ainsi deux charges de travail différentes, une

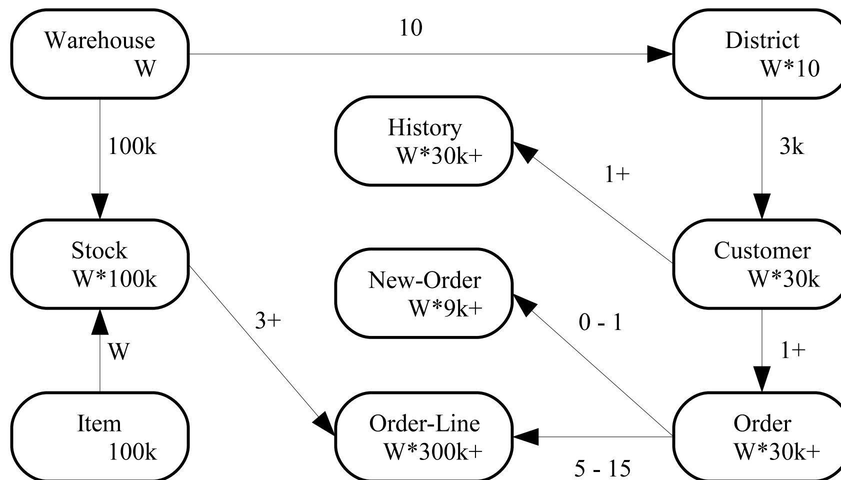


Figure 6.3 – Les tables du banc d'essai TPC-C

charge faible (Low) où λ est égal à 3s et une charge forte (Bursty) où λ est égal à 100ms. Nous définissons deux tailles de transactions ($|T|$), les transactions courtes mettent à jour 5 tuples alors que les transactions longues mettent à jour 50 tuples. Ainsi, le nombre de transactions longues par rapport aux transactions courtes est défini par le paramètre ltr que nous fixons à 66 (66% des transactions de mises à jour sont longues). Les paramètres du modèle de performances sont résumés dans la Table 6.2.

6.3.2 Banc d'essai TPC-C

Le deuxième banc d'essai que nous utilisons est TPC-C du Transaction Processing Performance Council (TPC). TPC-C est un jeu d'essai OLTP (On-Line Transaction Processing) qui définit un programme ayant une architecture permettant de gérer des transactions en temps réel. Des applications de réservation de billets de train ou de bourse nécessitent ce genre d'architecture.

6.3.2.1 Configurations et transactions

TPC-C implique cinq types différents de transactions exécutées en ligne ou en traitement par lot. La base de données est composée de neuf tables avec une large gamme d'enregistrement et de taille de population. TPC-C simule un environnement complet où des utilisateurs exécutent des transactions sur la base de données. Le banc d'essai est centré sur l'activité principale (les transactions de mises à jour) d'un environnement de prise de commande. Ces transactions incluent la prise et la validation de commandes, l'enregistrement des paiements, la vérification du statut des commandes et la surveillance du niveau du stock.

Dans le modèle TPC-C, un fournisseur possède un certain nombre de succursales (*Warehouse*) et leurs départements de vente associés (*District*). TPC-C peut gérer l'expansion du fournisseur en ajoutant des succursales. La taille de la base est donc déterminée par le nombre

Transactions	Écritures	Lectures	Fréq..	Poid
New-order	(M) District, Stock (I) New-Order, Order, Order-Line	Warehouse, District, Customer, Item, Stock	45%	Moyen
Payment	(M) Warehouse, District, Customer (I) History	Warehouse, District, Customer	43%	Léger
Order-status		Customer, Order, Order-Line	6%	Léger
Stock-level		District, Order-Line, Stock	6%	Lourd
I : Insertion, M : Mise à jour				

Table 6.3 – Paramètres du banc d'essai TPC-C

de succursales (W). Le nombre de départements est fixé à 10 par succursales (soit W*10 départements au total) et chaque département gère 3000 clients (*Customers*). Nous ne détaillerons pas les autres tables mais elles sont représentées sur la Figure 6.3. La taille des tables dépend du nombre de succursales excepté la taille de table des produits (Items) qui est fixe (100000 éléments). Sur cette figure, les nombres dans les cadres représentent la cardinalité de la table, les nombres sur les flèches représentent le nombre de fils par rapport au parent et le (+) indique qu'une variation de la population d'une table ou de la relation est possible dans l'état initial.

Un vendeur, qui représente un client du banc d'essai, peut à tout moment utiliser l'une des cinq transactions disponibles pour traiter des commandes.

La transaction la plus courante consiste en l'ajout d'une nouvelle commande (*New-order*) comprenant en moyenne 10 produits différents. Une autre transaction fréquente est le paiement d'une commande (*Payment*) d'un client. Les trois autres transactions sont moins fréquentes. La première effectue une requête sur le statut d'une commande (*Order-Status*). La deuxième calcule le niveau du stock sur une succursale (*Stock-Level*), enfin la troisième transaction exécute un traitement par lot pour valider une commande (*Delivery*). Cette dernière transaction n'est pas utilisée dans notre version de TPC-C car seule l'exécution des transactions en temps réel nous intéresse. De plus, cette transaction ne représente que 5% du total des transactions. Toutes les autres transactions sont résumées dans la Table 6.3. On y retrouve les quatre transactions utilisées avec pour chacune d'elle les tables mises à jour, les tables lues, leur fréquence et leur poids (temps d'exécution moyen). On peut noter que le banc d'essai TPC-C privilégie les transactions de mises à jour puisqu'elles représentent 88% du total des transactions.

Pour nos expérimentations, nous utilisons deux configurations. Dans la configuration *Répliquée Totalement* (Fully Replicated - Fr), tous les noeuds sont maîtres de toutes les copies. Dans la configuration *Répliquée Partiellement* (Partially Replicated - PR), un quart des noeuds possède les copies nécessaires à l'exécution de la transaction New-order, un autre quart les copies nécessaires à Payement, le troisième quart les copies nécessaires à Order-status et le dernier

Paramètre	Définition	Valeurs
W	Nombre de succursales	1 ; 5 ; 10
Clients	Nombre de clients par succursale	10
$\lambda_{clients}$	Taux d'arrivée moyen pour chaque client	10s
λ	Taux d'arrivée moyen	1s, 200ms, 100ms
Configuration	Réplication des tables	*FR*, *PR*
Nb-Noeuds	Nombre de noeuds	4, 8, 16, 24, 32, 48, 64
M	Nombre de transactions soumises pendant le test pour chaque client	100
$Max + \epsilon$	Délai introduit avant de soumettre une transaction	200ms

Table 6.4 – Paramètres du banc d'essai TPC-C

quart les copies nécessaires à Stock-level. Toutes les copies sont en mode multimaître. Voici un exemple avec une configuration partielle à quatre noeuds qui détiennent les copies :

- $N_{New-order}$: District, Stock, New-Order, Order, Order-Line, Warehouse, Customer et Item ;
- $N_{Payement}$: District, Warehouse, Customer et History ;
- $N_{Order-status}$: Customer, Order et Order-Line ;
- $N_{Stock-level}$: District, Order et Stock.

Au final, un quart des noeuds peut exécuter New-order ($N_{New-order}$), un quart peut exécuter Payement ($N_{Payement}$), la moitié peut exécuter Order-status ($N_{New-order}$ et $N_{Order-status}$) et la moitié peut exécuter Stock-level ($N_{New-order}$ et $N_{Stock-level}$).

6.3.2.2 Modèle de performance

Les paramètres du modèle de performance pour le banc d'essai TPC-C sont résumés dans la Table 6.4. Le nombre de succursales (W) détermine la taille de la base mais aussi le nombre de client car il y a dix clients par succursale (paramètre $clients$) comme spécifié dans TPC-C. Pour un client, nous fixons le taux d'arrivé moyen des transactions ($\lambda_{clients}$) à 10s. Ainsi, avec 100 clients (10 succursales et 10 clients par succursale), le taux d'arrivée moyen des transactions λ est de 100ms. Les valeurs choisies pour le nombre de succursales sont : 1, 5 et 10. Les différents taux d'arrivés des transactions sont donc 1s, 200ms et 100ms.

Au cours des expérimentations, chaque client soumet une transaction parmi les quatre disponibles à un noeud choisi au hasard capable de la traiter. À la fin de l'exécution, après avoir soumis M transactions, chaque client doit avoir respecté la fréquence des transactions définies dans TPC-C (voir Table 6.3) : 6% de Order-status, 6% de Stock-level, 45% de New-order et 43% de Payment.

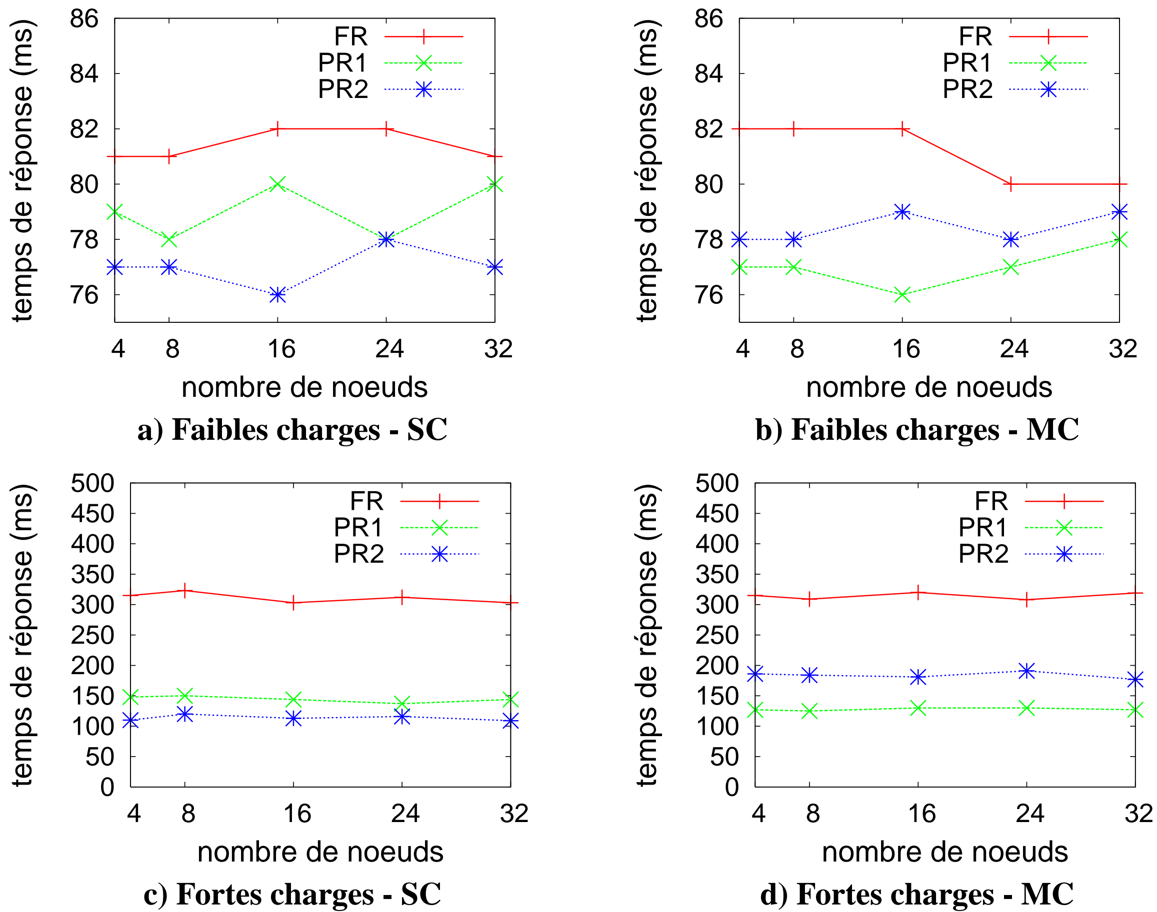


Figure 6.4 – Passage à l'échelle avec le banc d'essai spécifique

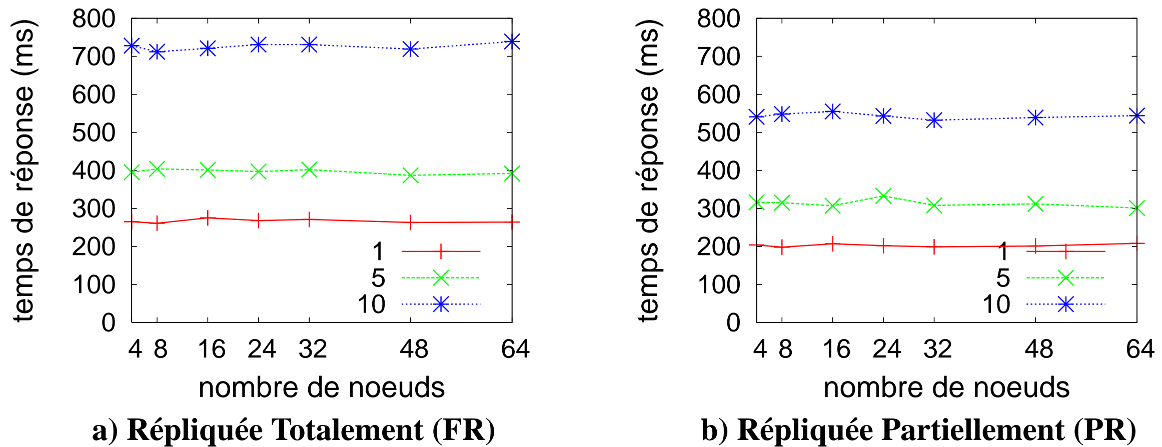
6.4 Performances

Dans cette section, nous utilisons les deux bancs d'essai définis dans la section précédente pour montrer certaines propriétés de notre algorithme.

6.4.1 Passage à l'échelle

Notre première expérimentation étudie le passage à l'échelle (Scale-up) de l'algorithme de réplication préventive. Le passage à l'échelle est atteint lorsque pour des transactions de mises à jour, l'augmentation du nombre de noeuds n'entraîne pas de dégradation des performances. Pour chaque banc d'essai, nous faisons varier la configuration, le nombre de noeuds et la charge de travail.

Dans le cas du **banc d'essai spécifique**, nous considérons les transactions *SC* et *MC* avec une forte charge et une faible charge (λ = 200ms et λ = 3s) en variant le nombre de noeuds pour chaque configuration (*FR*, *PR1*, *PR2*). Les transactions SC et MC mettent à jour au hasard R,



a) Répliquée Totalement (FR) **b) Répliquée Partiellement (PR)**

Figure 6.5 – Passage à l'échelle avec le banc d'essai TPC-C

S et/ou T. Pour chaque test, nous mesurons le temps moyen des réponses par transaction. La durée d'une expérience est le temps de soumettre 100 transactions.

Les résultats (voir Figure 6.4) montrent clairement que l'algorithme passe à l'échelle pour tous les tests. Contrairement aux algorithmes de réplications synchrones, l'algorithme de réplication préventive a des temps de réponse linéaires car pour chaque transaction il requiert uniquement la diffusion d'un message pour les noeuds possédant toutes les copies nécessaires à la transaction et un autre message pour les autres noeuds. Les résultats montrent aussi l'impact de la configuration sur les performances. Tout d'abord, en faible charge (voir Figure 6.4a et 6.4b), nous observons un très faible impact, les temps de réponse sont similaires pour toutes les configurations, la transaction s'exécute seule sur chaque noeud. En revanche, en forte charge (voir Figure 6.4c et 6.4d), les temps de réponse des configurations partielles sont meilleurs que ceux de la configuration totale. Ces différences sont dues au parallélisme introduit par la réplication partielle. En effet, ne possédant pas les mêmes copies, les noeuds traitent des transactions différentes. L'amélioration des performances de *PR* sur *FR* est meilleure pour les transactions SC (voir Figure 6.4c) et assez significative (un facteur de 2 pour *PR1* et de 3 pour *PR2*). Pour les transactions MC (voir Figure 6.4d), l'amélioration des performances a toujours un facteur de 2 pour *PR1* mais seulement de 1.5 pour *PR2*. Cette différence est due au fait que les configurations *FR* sont relativement plus adaptées aux transactions MC qu'aux transactions SC. *PR2* stocke des copies individuelles de R, S et T qui peuvent exécuter les transactions SC sans être surchargés par des mises à jours sur d'autres copies. Ainsi, *PR2* est plus performante pour les transactions SC, alors que *PR1* est plus performante pour les transactions MC.

Dans le cas du **banc d'essai TPC-C**, nous faisons varier pour chaque test le nombre de noeuds pour chaque configuration (*FR* et *PR*), le tout pour un nombre différent de succursales (1, 5 ou 10). Pour chaque test nous mesurons le temps de réponse moyen des transactions de mises à jour (New-order et Payement). Une expérience dure le temps de soumettre 100 transactions par client.

Les résultats montrent également que le passage à l'échelle est atteint (voir Figure 6.5). Tout

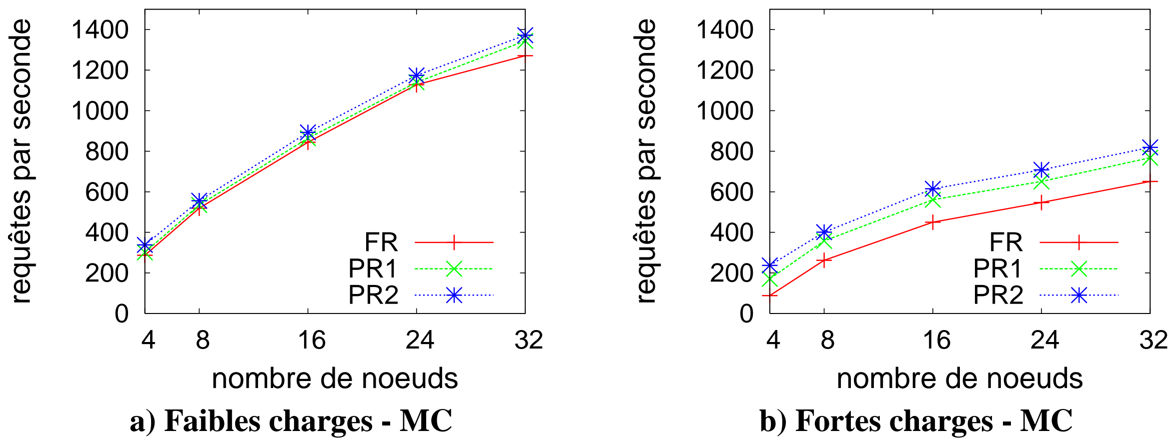


Figure 6.6 – Gains de performances pour le banc d'essai spécifique

comme pour le banc d'essai spécifique, les temps de réponse sont linéaires et ne dépendent pas du nombre de noeuds. Les résultats nous montrent aussi que les performances diminuent lorsque le nombre de succursales augmente (ce qui augmente la charge). Sur la Figure 6.5a, la charge est deux fois plus grandes pour 10 succursales que pour 5 (100ms contre 200ms), et comme attendu les temps de réponse sont deux fois moins bons (400ms contre 800ms). Nous pouvons donc en déduire que l'algorithme de réplication préventive a de bons temps de réponse lorsque la charge augmente et nous pouvons nous attendre à un comportement similaire avec une charge plus importante. Les résultats montrent aussi l'impact de la configuration sur les performances. Dans les configurations *PR* (voir Figure 6.5b), les noeuds sont moins chargés car ils exécutent moins de transactions que dans une configuration *FR* (voir Figure 6.5a). Les performances sont donc meilleures pour les configurations *PR* que pour les configurations *FR* (d'environ 30%).

Finalement, nous pouvons conclure que notre algorithme passe à l'échelle et que les performances dépendent de la charge et non du nombre de noeud. Nous avons aussi remarqué que la configuration et le placement des copies doivent être adaptés aux types de transactions choisies. Réduire le nombre de copies sur un noeud n'entraîne pas forcément une amélioration des temps de réponse.

6.4.2 Gains de performance

Ces expérimentations étudient les gains de performances (Speed-up) pour les transactions de lectures lorsque le nombre de noeuds augmente. Pour tester les gains, nous reproduisons l'expérience réalisée à la section précédente mais nous ajoutons en plus des clients qui vont soumettre des requêtes au SGBD. Pour chaque banc d'essai, nous faisons varier la configuration, le nombre de noeuds et la charge de travail. Le nombre de clients soumettant des transactions de lectures est de 128. Chaque client est associé à un noeud et les clients sont distribués équitablement sur les noeuds. Le nombre de clients par noeud est donc de 128/nombre de noeuds. Les clients soumettent séquentiellement des transactions de lectures pendant le déroulement de

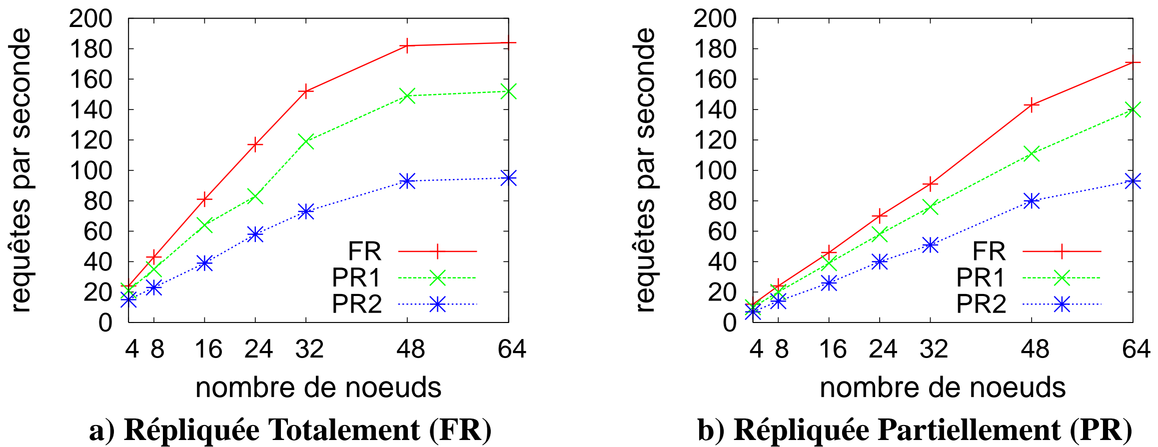


Figure 6.7 – Gains de performances pour le banc d'essai TPC-C

l'expérience. Nous mesurons le débit de la grappe, c'est-à-dire le nombre de transactions de lectures par secondes.

Pour le **banc d'essai spécifique**, nous considérons des transactions de lecture MC en charge forte et faible (λ = 200ms et λ = 3s) en variant le nombre de noeuds pour chaque configuration (*FR*, *PR1*, *PR2*). Comme chaque requête de lecture implique un seul noeud et que les clients sont affectés à un seul noeud, les différences entres les transactions SC et MC sont négligeables, nous utilisons donc uniquement les transactions MC.

Les résultats nous montrent (voir Figure 6.6) que lorsque le nombre de noeuds augmente, le débit du système augmente aussi. Pour une faible charge (voir Figure 6.6a), la différence de gain obtenu pour les trois configurations n'est pas significative. En revanche, pour les fortes charges (voir Figure 6.6b) les gains sont moins important en augmentant le nombre de noeuds car ces derniers sont déjà surchargés. Cependant la différence entre les configurations est significative car sous les fortes charges les configurations ont plus d'influence sur les performances et les noeuds sont moins chargés dans les configurations *PR*.

Pour le **banc d'essai TPC-C**, nous considérons des transactions de lecture Order-status pour un nombre différent de succursales (1, 5 ou 10) en variant le nombre de noeuds pour chaque configuration (*FR*, *PR*). Nous avons choisi la transaction Order-status plutôt que Stock-level car c'est une requête beaucoup plus légère.

Les résultats montrent (voir Figure 6.7) que l'augmentation du nombre de noeuds améliore le débit du système. Par exemple dans la Figure 6.7a, quelque soit le nombre de succursales, le nombre de requêtes par seconde pour 32 noeuds (150 requêtes par seconde) est presque deux fois meilleurs que pour 16 noeuds (80 requêtes par seconde). Cependant, si nous comparons *FR* avec *PR*, nous remarquons que le débit est meilleur avec *FR*. Bien que les noeuds soient moins chargés que pour *FR*, les performances sont deux fois moins bonnes puisque seule la moitié des noeuds supporte la transaction. Ceci est dû au fait qu'avec une configuration *PR*, seule la moitié des noeuds peuvent exécuter la transaction Order-status. Avec *FR*, au delà de 48 noeuds,

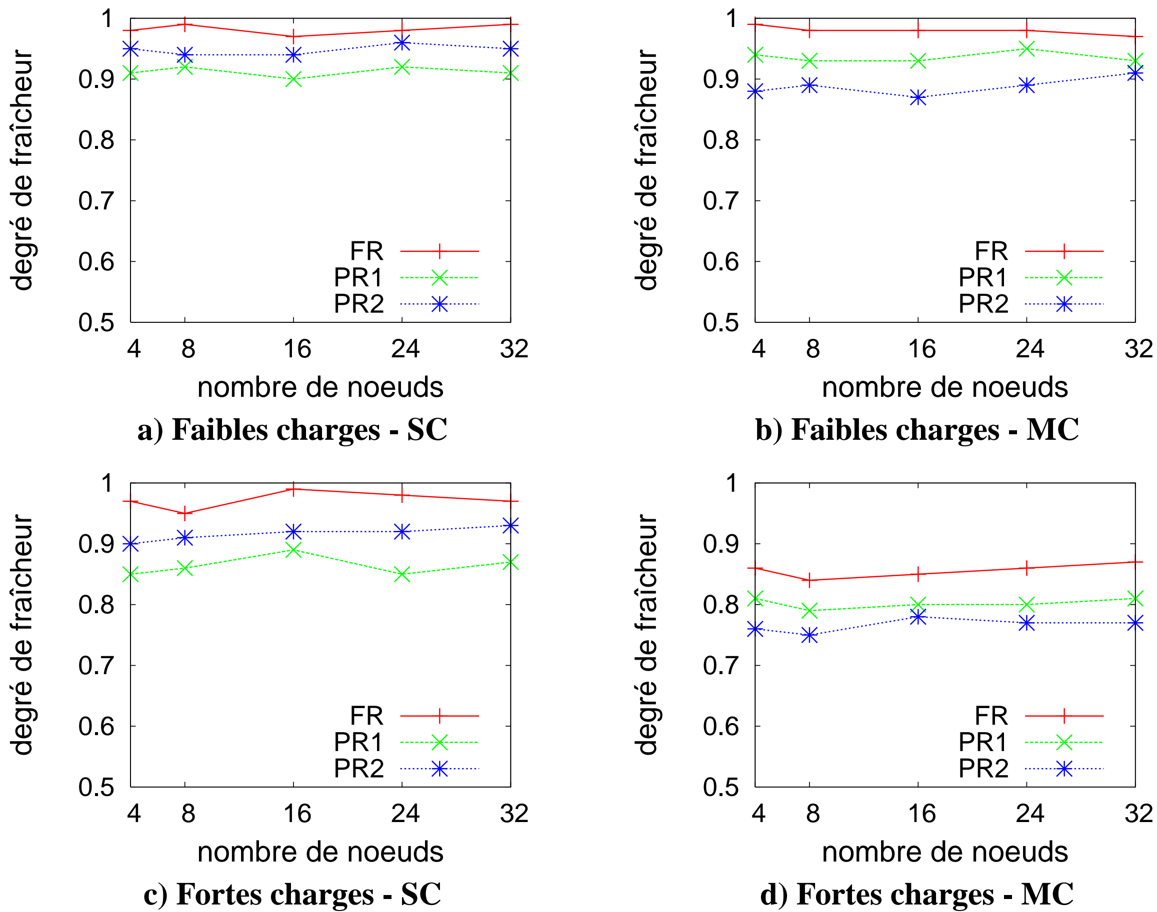


Figure 6.8 – Degrée de fraîcheur pour le banc d'essai spécifique

le débit n'augmente plus parce que le nombre optimal de noeuds a été atteint, les requêtes sont alors exécutées aussi vite que possible.

6.4.3 Degré de fraîcheur

Notre troisième mesure prend en compte le degré de fraîcheur des transactions. Intuitivement, le degré de fraîcheur d'une copie R sur le noeud i est défini comme le taux de transactions validées sur la copie R sur le noeud i par rapport à la moyenne des transactions validées sur les autres noeuds. Ainsi le meilleur degré de fraîcheur est 1, où les copies sont mutuellement consistantes. En revanche un degré de fraîcheur proche de zéro exprime le fait qu'il existe un nombre important de mises à jour validées n'ayant pas été propagées aux autres copies. Pour ce test, nous reprenons les paramètres de la section 6.4.1.

Pour le **banc d'essai spécifique** (voir Figure 6.8) nous remarquons clairement que les degrés de fraîcheur sont tous bons, ils ne descendent jamais en dessous de 0.7 et sont plus proches de 0.9. Bien que la configuration *FR* soit la plus chargée, elle possède toujours le meilleur degré de fraîcheur, ceci est dû au fait que les noeuds possèdent tous la même charge et exécutent les

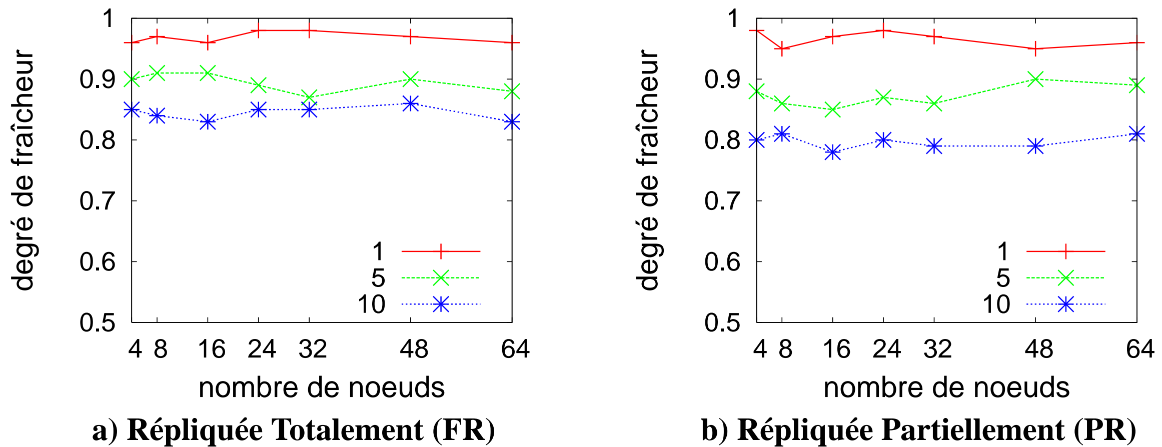


Figure 6.9 – Degré de fraîcheur pour le banc d'essai TPC-C

transactions au même moment. Pour les configurations *PR*, les résultats sont moins bons car les transactions sont validées en une seule phase sur les noeuds possédant toutes les copies et en deux phases sur les autres. Le temps d'effectuer la seconde phase explique la différence de fraîcheur. Nous constatons finalement que lorsque les types de transactions ne sont pas adaptés aux configurations (MC pour *PR2*) alors le degré de fraîcheur est plus faible.

Pour le **banc d'essai TPC-C** (voir Figure 6.9) le degré de fraîcheur est toujours bon (autour de 0.96) pour une faible charge (une seule succursale), en revanche il baisse lorsque la charge de travail augmente (entre 0.8 et 0.9). De même le degré de fraîcheur est moins bon pour les configurations partielles (voir Figure 6.9b) que pour les configurations totales (voir Figure 6.9a). Et ceci pour des raisons identiques à celles du banc d'essai spécifique, les transactions s'effectuent plus souvent en deux phases ce qui entraîne une différence d'état plus longue entre les copies qui sont mises à jour en une seule phase (sur les noeuds qui possèdent toutes les copies nécessaires à la transaction) et celles qui le sont en deux phases (sur les noeuds qui ne possèdent pas toutes les copies).

6.4.4 Impact sur les délais

Nous étudions désormais l'effet d'une exécution optimiste des transactions dès leur réception. Notre premier test montre l'impact des messages non-ordonnés sur le nombre de transactions abandonnées à la suite des exécutions optimistes (voir Section 4.2 au Chapitre 4). Enfin, notre second test montre les gains de l'approche optimiste sur le délai d'ordonnancement des transactions.

Dans notre première expérimentation, la Figure 6.10 montre que le pourcentage de messages non-ordonnés et le pourcentage d'abandons. Ces mesures sont effectuées sur le **banc d'essai TPC-C** sur l'expérience de passage à l'échelle (voir Section 6.4.1). Moins de 5% des messages sont non-ordonnés et seulement 1% des transactions sont annulées. Ces chiffres prouvent que notre solution d'exécuter les transactions optimistiquement est viable car elle entraîne très

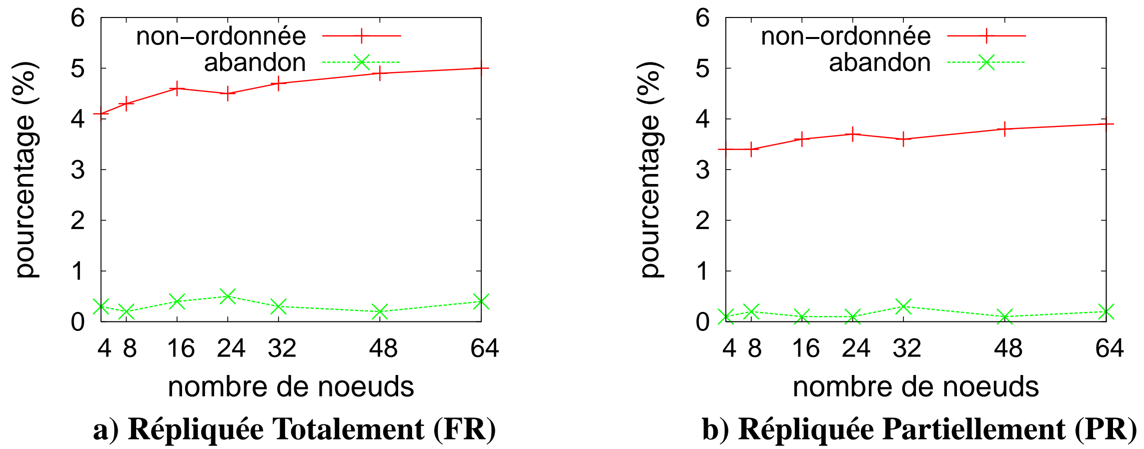


Figure 6.10 – Impact des messages non ordonnées

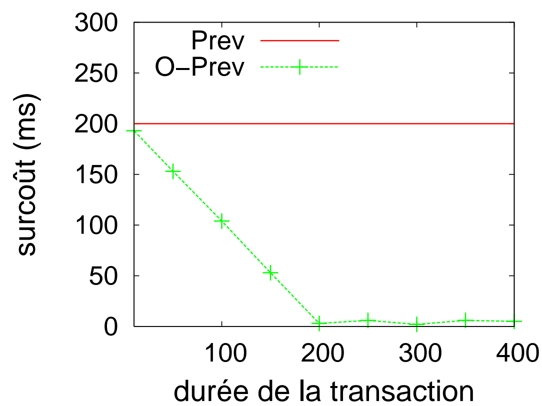


Figure 6.11 – Gain de l'élimination de délai d'ordonnancement

peu d'annulations. Seulement 20% des transactions non-ordonnées introduisent des abandons puisque deux messages non-ordonnés sont reçus pendant une très courte période (environ 2ms). Ainsi, le second message est reçu avant que le premier message ne soit traité. Par conséquent, ils sont ré-ordonnés avant l'exécution du premier message.

Pour les configurations *PR* (voir Figure 6.10b), le pourcentage de messages non-ordonnés est plus bas que pour les configurations *FR* (voir Figure 6.10a) puisque moins de messages sont émis dans une configuration partielle.

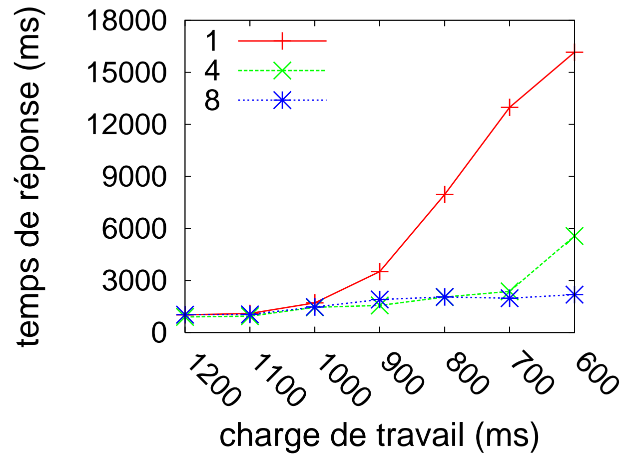
Dans notre seconde expérimentation, nous étudions l'impact de la taille des transactions sur l'élimination du délai d'ordonnancement. Dans notre approche optimiste, les transactions ont toujours besoin d'être retardées (de $Max + \epsilon$) avant validation, mais ce délai est partiellement supprimé. La Figure 6.11 montre l'importance du délai Max par rapport à la taille de la transaction. Notre test a été effectué sur le **banc d'essai TPC-C** avec seulement 8 noeuds et une configuration *FR*. Nous ne faisons pas varier le nombre de noeuds car cela n'influence pas le temps d'attente. Nous soumettons 100 transactions au système avec une faible charge et nous faisons varier la taille des transactions. Nous mesurons enfin le délai introduit par l'algorithme de rafraîchissement. Rappelons que le délai normal est de 200ms sans optimisation. Le délai sans optimisation est représenté par la courbe $Prev$ et le délai avec optimisation est représenté par la courbe $O - Prev$.

Une observation importante : nous constatons une diminution rapide du délai lorsque la taille de la transaction augmente. Depuis que les transactions sont exécutées dès que possible, l'ordonnancement a en effet lieu pendant l'exécution des transactions. Le temps d'ordonnancement est donc égal au délai $Max + \epsilon$ moins la taille de la transaction. Par exemple, avec une transaction de 50ms, le délai est de 150ms. Par conséquent, avec des transactions plus longues que $Max + \epsilon$ (200ms), le délai est proche de 0 puisque le temps d'ordonnancement est inclus dans le temps d'exécution. Ainsi, le gain est proche de $Max + \epsilon$ qui est le gain optimal pour son élimination.

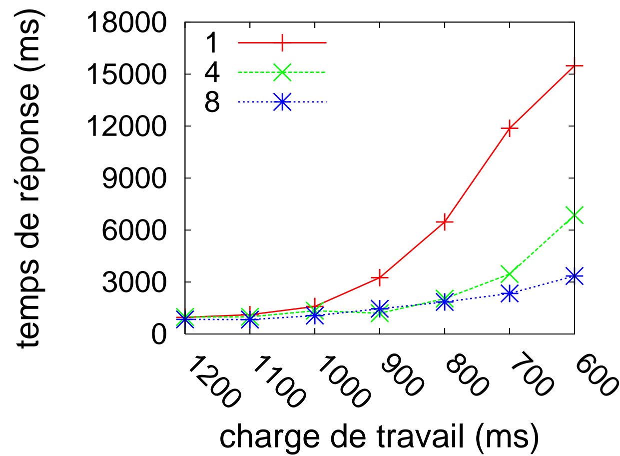
6.4.5 Exécution en parallèle

Pour valider l'optimisation d'exécution en parallèle des transactions, nous mesurons les temps de réponse des transactions en fonction de la charge. Nous ajoutons deux nouveaux paramètres. C qui est le pourcentage de transactions en conflit avec la transaction précédente (C=0%, C=25% ou C=50%). Et P qui est le nombre maximum de transactions qui peuvent s'exécuter en concurrence (P=1, P=4 ou P=8). Notons qu'une seule une transaction en concurrence est équivalent à supprimer l'exécution en concurrence. Nous effectuons ce test sur le **banc d'essai spécifique** car il est plus facile de gérer la concurrence sur ce modèle. Nous utilisons des transactions SC avec un ltr à 0 (les transactions sont toutes courtes et mettent à jours 5 tuples). Finalement, l'expérience dure le temps d'envoyer 100 transactions.

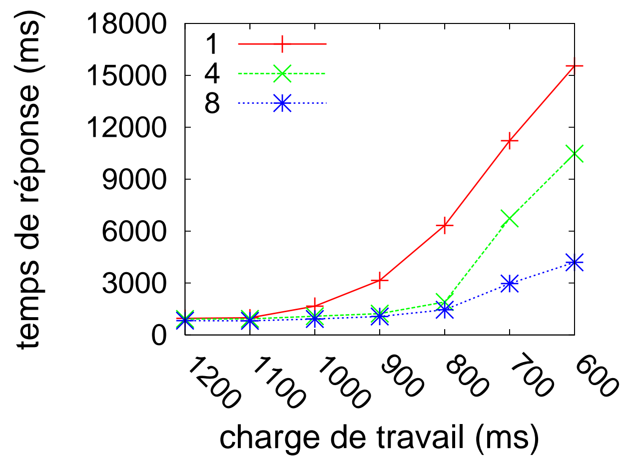
La Figure 6.12 montre que les résultats de cette expérience. Sans concurrence (P=1), nous constatons clairement que le point de saturation est atteint dès que la charge de la transaction est supérieur au temps moyen d'exécution d'une transaction (autour de 1000ms). Par contre, sur la Figure 6.12a, lorsque les transactions sont déclenchées en concurrence (P=4 ou P=8), le



a) Fortes charges - 0% de concurrence



b) Fortes charges - 25% de concurrence



c) Fortes charges - 50% de concurrence

Figure 6.12 – Exemple de configuration pour 8 noeuds

goulot d'étranglement disparaît quasiment et les performances sont bien meilleures. Cependant sur les Figures 6.12b et 6.12c, avec un très fort pourcentage de conflits entre les transactions (C=25% ou C=50%), les transactions finissent par s'exécuter séquentiellement. Mais ce point faible peut toujours être maîtrisé en augmentant le nombre de transactions en concurrence.

6.5 Conclusion

Dans cette section, nous avons éprouvé l'algorithme de réplication préventive en testant trois propriétés recherchées pour un algorithme de réplication asynchrone en grappe :

- Le passage à l'échelle : nous avons montré que l'augmentation du nombre de noeuds n'influence pas les performances quelques soient la charge et la configuration. Nous avons également montré l'influence de la configuration sur les temps de réponse. Quand la configuration et le placement des données sont adaptés au type de transactions soumises au système alors les performances deviennent optimales.
- Le gain de performance : si les performances ne se dégradent pas pour les transactions de mises à jour lorsqu'on augmente le nombre de noeuds, en revanche le débit du système pour les lectures augmente.
- Le degré de fraîcheur : le retard en nombre de validation de transactions reste toujours faible quelque soit le banc d'essai et la configuration. C'est lorsque les transactions ne sont pas adaptées aux types de configurations que le degré de fraîcheur diminue. En effet, dans ce cas certains noeuds (ceux qui possèdent le plus de copies) sont plus chargés que d'autres et mettent plus de temps à exécuter toutes les transactions qui leurs sont soumis.

De plus, nous avons montré que nos optimisations permettent un meilleur support face aux fortes charges. En effet, en exécutant les transactions en parallèle et en éliminant le délai d'ordonnancement, notre système supporte mieux l'émission massive de transactions. Les expérimentations ont prouvé que l'exécution optimiste des transactions n'entraînait que 1% d'abandon, ce qui rend notre optimisation viable.

Dans le chapitre suivant, nous concluons.

Conclusion

Synthèse

Dans cette thèse, nous avons dans un premier temps proposé une architecture adaptée aux grappes qui conserve l'autonomie des noeuds. Nous avons ensuite décrit un algorithme de réplication asynchrone garantissant la cohérence forte et supportant la réplication totale et partielle des données. Nous avons ensuite optimisé cet algorithme pour le support des lourdes charges. L'algorithme et ses optimisations ont alors fait l'objet d'une implémentation dans un prototype: RepDB*. Et finalement, nous avons validé le prototype grâce à deux bancs d'essai : le premier est spécifique à l'algorithme et le second est TPC-C.

Une architecture de grappes

Nous avons présenté une architecture sans partage adaptée au contexte ASP où une grappe contient les bases de données mais aussi les applications d'un client.

Cette architecture, par sa modélisation en cinq couches et l'utilisation de trois services, permet de garantir l'autonomie des noeuds. Nos exemples montrent que l'architecture en couches permet un équilibrage de charge à deux niveaux : les applications et les bases de données. De plus, par un routage des requêtes sur deux des couches, nous autorisons la séparation des applications et des données sur plusieurs noeuds. Ce routage permet également aux noeuds de pouvoir traiter n'importe quelle requête, et ceci même si le noeud n'e possède pas les applications ou les données nécessaires à la requête, en la déléguant à un autre noeud. Finalement, dans notre architecture, les noeuds sont autonomes les uns par rapport aux autres et la panne d'un noeud n'entraîne pas le blocage des autres sites. De même chaque couche est indépendante l'une de l'autre et la panne de l'une d'entre elles n'entraîne pas la panne des autres.

Un algorithme de réplication asynchrone

Nous avons présenté un nouvel algorithme de réplication que nous avons appelé Réplication Préventive. Nous nous sommes basés sur l'algorithme de réplication *Lazy-Master* proposé par Pacciti et al. [PMS01, PMS99]. Le principe de l'algorithme *Lazy-Master* est de soumettre les transactions dans un ordre total sur tous les noeuds en fonction de leur estampille d'arrivée. Pour accomplir ceci, il retarde l'exécution des transactions pour s'assurer qu'aucune transaction plus ancienne n'est en route pour le noeud. Cependant, l'algorithme *Lazy-Master* n'autorise que la mise à jour par copies primaires (où une copie ne peut être mise à jour que sur un seul site) et il

impose une restriction sur le placement des données (le graphe de dépendances des noeuds ne doit pas former de cycles).

Pour résoudre ces problèmes, nous avons ajouté dans un premier temps le support de la réplication totale où tous les noeuds possèdent toutes les copies et sont des noeuds maîtres. Les copies multimaîtres sont alors rafraîchies en diffusant et en retardant les transactions sur tous les noeuds, y compris sur le noeud d'origine. En autorisant ainsi davantage de noeuds multimaîtres, nous supprimons le goulet d'étranglement que représente un seul noeud maître. Ce type de configuration n'est pas exploitable du fait des coûts trop importants qu'il introduit : tous les noeuds doivent exécuter toutes les transactions.

Dans un second temps, nous avons donc introduit le support de la réplication partielle. Dans une configuration partielle, nous ne faisons de restrictions ni sur le mode d'accès aux copies (mode Copie Primaire ou Mise à jour Partout) ; ni sur le placement des données (un noeud peut posséder une partie seulement des copies et un ou plusieurs types de copie : Primaires, Multimaîtres et Secondaires). Dans le cas de la réplication, un noeud n'est pas forcément en mesure d'exécuter toutes les transactions car il ne possède pas toutes les copies nécessaires. L'algorithme place alors la transaction en attente des jeux d'écritures qui sont diffusés par le noeud d'origine. L'algorithme de réplication partielle autorise donc un plus grand nombre de configurations mais introduit la diffusion d'un message de rafraîchissement.

Pour chacune des versions de l'algorithme réplication préventive (totale et partielle), nous avons proposé : une architecture pour le gestionnaire de réplication, une description détaillée des algorithmes et les preuves que ces algorithmes garantissent la cohérence forte sans introduire d'inter-blocages.

Un meilleur support aux fortes charges

Afin de mieux supporter les applications à fortes charges où les transactions de mise à jour sont majoritaires, nous avons amélioré les points faibles de l'algorithme de réplication préventive.

Nous avons dans un premier temps éliminé le délai introduit par l'ordonnancement des transactions en les exécutant de manière optimiste dès leur réception dans le noeud et non plus après le délai $Max + \epsilon$. Si les transactions n'ont pas été exécutées dans l'ordre correct (celui de leurs estampilles), alors elles sont annulées et ré-exécutées après ordonnancement. Le nombre d'abandons reste faible car dans un réseau rapide et fiable les messages sont naturellement ordonnés. Malgré cette optimisation la cohérence forte est garantie car nous retardons la validation des transactions (et non plus la totalité de la transaction) exécutées optimistiquement. Les transactions sont ordonnancées pendant leur exécution et non plus avant, supprimant ainsi les délais d'ordonnancement.

La deuxième optimisation concerne la soumission des transactions. Dans les algorithmes présentés au Chapitre 3, les transactions sont soumises au SGBD une par une pour garantir la cohérence. Le module de soumission représente donc un goulet d'étranglement dans le cas où le temps moyen d'arrivée des transactions est supérieur au temps moyen d'exécution d'une transaction. Pour supprimer ce problème, nous avons autorisé l'exécution parallèle des transac-

tions non conflictuelles. Cependant, pour garantir la cohérence des données, nous ordonnançons toujours le démarrage et la validation des transactions, ceci afin de garantir que toutes les transactions soient exécutées dans le même ordre sur tous les noeuds malgré le parallélisme. Nous prouvons que l'exécution en parallèle des transactions est équivalente à une exécution séquentielle.

Le prototype RepDB*

Dans ce chapitre, nous avons décrit le prototype RepDB* qui implémente l'algorithme de réplication préventive avec toutes les optimisations. Nous nous sommes focalisés ici sur la partie technique du prototype. Nous avons décrit tous les composants logiciels qui forment RepDB* :

- Le gestionnaire de réplication écrit en Java avec un serveur JDBC (RmiJDBC) pour être utilisable par un grand nombre de clients ;
- Le service de bus réseau fourni par Spread pour une haute fiabilité, une haute performance et une communication robuste ;
- La librairie Log4J qui permet l'écriture de journaux sans modification du code du gestionnaire de réplication. Même si le prototype n'a pas encore de support pour la tolérance aux pannes, il contient tous les éléments nécessaires à son élaboration ;
- Une modification de PostgreSQL qui permet de lire efficacement les journaux et de transmettre les données au gestionnaire de réplication.

Les avantages de la réplication partielle aux bancs d'essai

Nous avons présenté le prototype RepDB* qui implémente notre algorithme de réplication préventive avec toutes les optimisations. Nous nous sommes focalisés ici sur la partie technique du prototype.

Puis nous avons éprouvé RepDB* en testant trois propriétés recherchées pour un algorithme de réplication asynchrone en grappe :

- Le passage à l'échelle : nous avons montré que l'augmentation du nombre de noeuds n'influence pas les performances quelque soit la charge et la configuration. Nous avons également montré l'influence de la configuration sur les temps de réponse. Quand la configuration et le placement des données sont adaptés au type de transactions soumises au système alors les performances deviennent optimales.
- Le gain de performance : si les performances ne se dégradent pas pour les transactions de mises à jour lorsqu'on augmente le nombre de noeuds, en revanche le débit du système pour les lectures augmente.
- Le degré de fraîcheur : le retard en nombre de validation de transactions reste toujours faible quelque soit le banc d'essai et la configuration. C'est lorsque les transactions ne

sont pas adaptées aux types de configurations que le degré de fraîcheur diminue. En effet, dans ce cas certains noeuds (ceux qui possèdent le plus de copies) sont plus chargés que d'autres et mettent plus de temps à exécuter toutes les transactions qui leur sont soumis.

De plus, nous avons montré que nos optimisations permettent un meilleur support face aux fortes charges. En effet, en exécutant les transactions en parallèle et en éliminant le délai d'ordonnancement, notre système supporte mieux l'émission massive de transactions. Les expérimentations ont prouvé que l'exécution optimiste des transactions n'entraînait que 1% d'abandon, ce qui rend notre optimisation viable.

Perspectives

Les perspectives de cette thèse concernent à court terme les points qui n'ont pas pu être approfondis comme le support de la tolérance aux pannes. Puis, à moyen terme, nous pourrions proposer un système complet qui comporte notamment un équilibrage de charge. Et finalement, à long terme, nous pourrions adapter notre algorithme à des contextes autres que ceux des grappes avec des réseaux à plus large échelle comme le pair-à-pair.

Intégration de la tolérance aux fautes et de la dynamicité

Un point important d'un algorithme de réplication que nous n'avons pas eu le temps de traiter dans cette thèse est la tolérance aux pannes. Bien que notre algorithme possède tous les journaux nécessaires à la mise en oeuvre d'un mécanisme de reprise en cas de panne, nous n'avons pas présenté d'algorithme pour un tel mécanisme. En plus de remettre à jour le noeud défaillant, les autres noeuds doivent pouvoir pallier son absence en recopiant sur d'autres noeuds les copies qu'il possède (à partir d'un ou plusieurs noeuds similaires). Dans le même esprit, un système adaptatif pourrait être mis en place. Il permettrait de changer dynamiquement la configuration des noeuds en déplaçant, en copiant, en ajoutant ou en supprimant des copies pour s'adapter aux mieux à la charge actuelle.

Développement de l'architecture sans partage pour grappe

Nous avons présenté une architecture pour grappe en cinq couches mais nous n'avons approfondi dans cette thèse que la couche de gestion de la réplication. Il serait intéressant de développer également les autres couches afin d'offrir une plate forme complète et opérationnelle. La couche la plus liée à notre travail est la couche d'équilibrage des transactions. En effet, les performances de l'algorithme de réplication dépendent fortement de la distribution des transactions sur la grappe. Tout comme l'équilibrage des requêtes dépend du placement des copies sur la grappe et donc de l'algorithme de réplication employé. Toujours dans le but d'améliorer les performances du système, nous pourrions mettre au point des techniques d'équilibrage qui suppriment les délais introduits par la réplication en remplissant les files d'attente (voir la Section

4.2 du Chapitre 4) ou en réduisent les abandons en envoyant les transactions conflictuelles sur un même noeud.

Extension au contexte pair à pair

Il serait intéressant d'adapter l'algorithme de réplication préventive à un contexte autre que les grappes. En effet, dans un contexte à plus large échelle comme le pair à pair, les propriétés que nous utilisons ne sont plus valables (notamment *Max*, le délai garanti d'échange d'un message). Il faut trouver une autre solution pour garantir l'ordre total des messages. Une solution consisterait à grouper les pairs géographiquement pour que chaque groupe ait une cohérence locale avant de réconcilier les noeuds entre eux. Une politique doit alors être appliquée par tous pour déterminer si un noeud peut ou non joindre un groupe. Cette solution s'applique parfaitement aux architectures super pair où chaque super pair pourrait représenter un groupe géographique.

Liste des publications

Journaux

[PCVÖ05] Esther Pacitti, Cédric Coulon, Patrick Valduriez, M. Tamer Özsu :
Preventive Replication in a Database Cluster.
Distributed and Parallel Databases 18(3) : 223-251 (2005)

[CPV04] Cédric Coulon, Esther Pacitti, Patrick Valduriez :
Scaling Up the Preventive Replication of Autonomous Databases in Cluster Systems.
VECPAR 2004 : 170-183.

Conférences Internationales

[CPV05a] Cédric Coulon, Esther Pacitti, Patrick Valduriez :
Consistency Management for Partial Replication in a High Performance Database Cluster.
IEEE ICPADS (1) 2005 : 809-815.

[PÖC03] Esther Pacitti, M. Tamer Özsu, Cédric Coulon :
Preventive Multi-master Replication in a Cluster of Autonomous Databases.
Euro-Par 2003 : 318-327.

Workshops Internationaux

[VPC05] Patrick Valduriez, Esther Pacitti, Cédric Coulon :
Large-scale Experimentation with Preventive Replication in a Database Cluster.
VLDB DIDDR Workshop 2005.

Conférences nationales

[CPV05b] Cédric Coulon, Esther Pacitti, Patrick Valduriez :
Optimistic-Preventive Replication in a Database Cluster.
BDA 2005 : 393-398.

Démonstrations

[CGPV04] Cédric Coulon, Gaëtan Gaumer, Esther Pacitti, Patrick Valduriez :
The RepDB* prototype : Preventive Replication in a Database Cluster.
BDA 2004 : 333-337.

Bibliographie

[ABKW98] T. Anderson, Y.i Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? *SIGMOD Rec.*, 27(2):484–495, 1998.

[ACZ03] C. Amza, A. Cox, and W. Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. *USENIX Symposium on Internet Technologies and Systems*, 2003.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, The Hebrew University of Jerusalem, Institute of Computer Science, 1994.

[Apa05] Apache Software Foundation. Log4j, http://logging.apache.org/log4j/docs/, 2005.

[AS98] Y. Amir and J. Stanton. The spread wide area group communication system. Technical Report CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University., 1998.

[AT02] Y. Amir and C. Tutu. From Total Order to Database Replication. In *International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, 2002. IEEE.

[ATL05] ATLAS Group. RepDB*: Data Management Component for Replicating Autonomous Databases in a Cluster System. http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB/, 2005.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Rec.*, 24(2):1–10, 1995.

[BG84] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems (TODS)*, 9(4):596–615, 1984.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BKR+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. In *SIGMOD*, pages 97–108, New York, NY, USA, 1999. ACM Press.

[CGPV04] C. Coulon, G. Gaumer, E. Pacitti, and P. Valduriez. The repdb* prototype: Preventive replication in a database cluster. In *Bases de données avancées (BDA)*, pages 333–337, 2004.

[Che00] R. Chevance. *Serveurs multiprocesseurs, clusters et architectures parallèles*. Eyrolles, 2000.

[CKV01] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[CMZ03] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial Replication: Achieving Scalability in Redundant Arrays of Inexpensive Databases. In *International Conference On Principles Of Distributed Systems (OPODIS)*, pages 58–70, 2003.

[CMZ04] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

[CPV04] C. Coulon, E. Pacitti, and P. Valduriez. Scaling Up the Preventive Replication of Autonomous Databases in Cluster Systems. In *VECPAR*, pages 170–183, 2004.

[CPV05a] C. Coulon, E. Pacitti, and P. Valduriez. Consistency Management for Partial Replication in a High Performance Database Cluster. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 809–815, 2005.

[CPV05b] C. Coulon, E. Pacitti, and P. Valduriez. Optimistic Preventive Replication in a Database Cluster. In Véronique Benzaken, editor, *Bases de données avancées (BDA)*, pages 393–398. Université de Rennes 1, 2005.

[CRR96] P. Chundi, Daniel J. Rosenkrantz, and S. S. Ravi. Deferred Updates and Data Placement in Distributed Databases. In *International Conference on Data Engineering (ICDE)*, pages 469–476, Washington, DC, USA, 1996. IEEE Computer Society.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[FLS97] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *ACM symposium on Principles of distributed computing (PODC)*, pages 53–62, New York, NY, USA, 1997. ACM Press.

[Gar78] G. Gardarin. Contribution to the theory of concurrency in databases. In *MFCS*, pages 201–212, 1978.

[Gar03] G. Gardarin. *Bases de données*. Eyrolles, 5ème édition edition, 2003.

[GHOS96] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The Dangers of Replication and a Solution. In *SIGMOD*, pages 173–182, Montreal, Canada, 1996.

[Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Symposium on Operating System Principles (SOSP)*, pages 150–162, Asilomar Conference Grounds, Pacific Grove CA, 1979. ACM, New York.

[GM79] G. Gardarin and M. A. Melkanoff. Proving consistency of database transactions. In *International Conference on Very Large Data Bases (VLDB)*, pages 291–298, 1979.

[GM97] L. George and P. Minet. A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints. In *International Conference on Distributed Computing Systems (ICDCS)*, page 441, Washington, DC, USA, 1997. IEEE Computer Society.

[GNPV02] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. Parallel processing with autonomous databases in a cluster system. In *CoopIS/DOA/ODBASE*, pages 410–428, 2002.

[GR93] J. N. Gray and A. Reuter. *Transaction Processing: concepts and techniques*. Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo (CA), USA, 1993.

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *ACM symposium on Principles of Database Systems (PODS)*, pages 8–15, New York, NY, USA, 1983. ACM Press.

[HAA02] J. Holliday, D. Agrawal, and A. Abbadi. Partial database replication using epidemic communication. In *International Conference on Distributed Computing Systems (ICDCS*, pages 485–493, 2002.

[HGKM98] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Transaction Processing: concepts and techniques. *Proceedings of the seventh international conference on World Wide Web 7 (WWW7)*, pages 347–357, 1998.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. pages 97–145, New York, NY, USA, 1993. ACM Press/Addison-Wesley Publishing Co.

[IBM05] IBM Corporation. DB2, http://www-306.ibm.com/software/data/db2/, 2005.

[JM90] S. Jajodia and D. Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems (TODS)*, 15(2):230–280, 1990.

[JPPMA02] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 150–159, 2002.

[JPPMKA02] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 477–484, 2002.

[KA00a] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *International Conference on Very Large Data Bases (VLDB)*, pages 134–143. Morgan Kaufmann, 2000.

[KA00b] B. Kemme and G. Alonso. A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, 2000.

[KB94] N. Krishnakumar and A. J. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Transactions on Database Systems (TODS)*, 19(4):586–625, 1994.

[KD00] I. Keidar and D. Dolev. Totally Ordered Broadcast in the Face of Network Partitions. In D. Avresky, editor, *Dependable Network Computing*, chapter 3. Kluwer Academic Publications, 2000.

[KK00] I. Keidar and R. I. Khazan. A Client-Server Approach to Virtually Synchronous Group Multicast: Specifications and Algorithms. In *International Conference on Distributed Computing Systems (ICDCS)*, page 344, Washington, DC, USA, 2000. IEEE Computer Society.

[KR87] B. Kähler and O. Risnes. Extending Logging for Database Snapshot Refresh. In *International Conference on Very Large Data Bases (VLDB)*, pages 389–398. Morgan Kaufmann, 1987.

[KT91] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 222–230, Washington, D.C., USA, 1991. IEEE Computer Society Press.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LD93] Scott T. Leutenegger and Daniel M. Dias. A modeling study of the tpc-c benchmark. In *SIGMOD*, pages 22–31. ACM Press, 1993.

[Lea06] D. Lea. *Concurrent Programming in Java(TM): Design Principles and Patterns (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2006.

[LKPMJP05] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based Data Replication providing Snapshot Isolation. In *SIGMOD*, 2005.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

[Mic05] Microsoft. SQL Server 2005, http://www.microsoft.com/sql, 2005.

[Mil03] D. L. Mills. A brief history of NTP time: memoirs of an Internet timekeeper. *SIGCOMM Comput. Commun. Rev.*, 33(2):9–21, 2003.

[MP97] C. Morin and I. Puaut. A Survey of Recoverable Distributed Shared Virtual Memory Systems. *IEEE Trans. Parallel Distrib. Syst.*, 8(9):959–969, 1997.

[Obj04] ObjectWeb Consortium. Rmijdbc : Client/server jdbc driver based on java rmi, http://rmijdbc.objectweb.org/, 2004.

[Ora95] Oracle Corporation. Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. White paper, Oracle Corporation, 500 Oracle Parkway, Redwoord City, CA 94065, 1995.

[Ora04] Oracle Corporation. Oracle 8i, http://www.oracle.com, 2004.

[OV99] T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, January 1999.

[PA04] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *ACM/IFIP/USENIX international conference on Middleware (MIDDLEWARE*, pages 155–174, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[PCVÖ05] E. Pacitti, C. Coulon, P. Valduriez, and T. Özsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(3):223–251, 2005.

[PGV03] C. Le Pape, S. Gançarski, and P. Valduriez. Trading freshness for performance in a cluster of replicated databases. *OTM Workshops*, pages 14–15, 2003.

[PGV04] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco: Improving query performance through freshness control in a database cluster. In *CoopIS/DOA/ODBASE (1)*, pages 174–193, 2004.

[PL88] J-F. Pâris and D. D. E. Long. Efficient Dynamic Voting Algorithms. In *International Conference on Data Engineering (ICDE)*, pages 268–275. IEEE Computer Society, 1988.

[PL91] C. Pu and A. Leff. Replica control in distributed systems: as asynchronous approach. In *SIGMOD*, pages 377–386. ACM Press, 1991.

[PMJPKA00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *International Symposium on Distributed Computing (DISC)*, pages 315–329, 2000.

[PMJPKA05] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *International Conference on Very Large Databases (VLDB)*, Edinburgh - Scotland - UK, 7–10 1999.

[PMS01] E. Pacitti, P. Minet, and E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed and Parallel Databases*, 9(3):237–267, 2001.

[PÖC03] E. Pacitti, T. Özsu, and C. Coulon. Preventive Multi-master Replication in a Cluster of Autonomous Databases. In *Euro-Par*, pages 318–327, 2003.

[Pos05] PostgreSQL Global Development Group. PostgreSQL 8.1, http://www.postgresql.org, 2005.

[PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In *International Symposium on Distributed Computing (DISC)*, pages 318–332, 1998.

[PV98] E. Pacitti and P. Valduriez. Replicated Databases: Concepts, Architectures and Techniques. *Networking and Information Systems (NIS)*, 1(4-5):519–546, 1998.

[Raa93] F. Raab. Tpc-c - the standard benchmark for online transaction processing (OLTP). In *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[SAS+96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. Data Replication in Mariposa. In *International Conference on Data Engineering (ICDE)*, pages 485–494, Washington, DC, USA, 1996. IEEE Computer Society.

[SKS86] S. K. Sarin, C. W. Kaufman, and J. E. Somers. Using History Information to Process Delayed Database Updates. In *International Conference on Very Large Data Bases (VLDB)*, pages 71–78. Morgan Kaufmann, 1986.

[SOMP01] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial Replication in the Database State Machine. In *IEEE International Symposium on Network Computing and Applications (NCA)*, pages 298–309, 2001.

[SPMO02] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic Total Order in Wide Area Networks. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, page 190, Washington, DC, USA, 2002. IEEE Computer Society.

[Spr06] Spread Concepts LLC. The spread toolkit, http://www.spread.org/, 2006.

[SPS+05] A. Sousa, J. Pereira, L. Soares, A. Correia Jr., L. Rocha, R. Oliveira, and F. Moura. Testing the dependability and performance of group communication based database replication protocols. In *DSN*, pages 792–801, 2005.

[SS05] Y. Saito and M. Shapiro. Optimistic Replication. *Computing Surveys*, 37(1):42–81, March 2005.

[Sun03] Sun Microsystems, Inc. Javatm 2 platform, standard edition, v 1.4.2, http://java.sun.com, 2003.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40(2-3):117–134, 1994.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[Top06] Top500 authors. Top500 supercomputer sites. http://www.top500.org., 2006.

[Tra04] Transaction Performance Processing Council. Tpc-c, http://www.tpc.org/tpcc/, 2004.

[Val93] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases*, 1(2):137–165, 1993.

[VPC05] P. Valduriez, E. Pacitti, and C. Coulon. Large-scale Experimentation with Preventive Replication in a Database Cluster. In *VLDB Workshop on Design, Implementation, and Deployment of Database Replication (DIDDR)*, 2005.

[WK05] S. Wu and B. Kemme. Postgres-R(SI): Combining Replica Control with Concurrency Control Based on Snapshot Isolation. In *International Conference on Data Engineering (ICDE)*, pages 422–433, Washington, DC, USA, 2005. IEEE Computer Society.

[WPS+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database Replication Techniques: a three parameter classification. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 206–215, Nürnberg, Germany, 2000. IEEE Computer Society.

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *International Conference on Distributed Computing Systems (ICDCS)*, Taipei, Taiwan, R.O.C., 2000. IEEE Computer Society.

[WS95] U. G. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, Bad Neuenahr, Germany, 1995.

Table des matières

Introduction **1**

1 Réplication dans les grappes de bases de données **7**
1.1 Introduction . . . 7
1.2 Réplication des données distribuées . . . 9
1.2.1 Les types de copies . . . 9
1.2.2 Les types de configurations . . . 9
1.2.3 Classification des modes de réplication . . . 11
1.3 Cohérence des données . . . 16
1.3.1 Cohérence forte . . . 17
1.3.2 Cohérence faible . . . 17
1.3.3 Cohérence *Snapshot* . . . 18
1.4 Architecture des grappes . . . 19
1.4.1 Les choix d'architectures en grappe . . . 20
1.4.2 Visibilité du SGBD . . . 21
1.5 Exploitation du réseau de communication . . . 22
1.5.1 Propriété de sûreté . . . 22
1.5.2 Propriété de vivacité . . . 24
1.6 La réplication dans les grappes . . . 25
1.6.1 Graphe Dirigé Acyclique . . . 25
1.6.2 Diffusion optimiste . . . 28
1.6.3 Utilisation du délai réseau . . . 31
1.6.4 Synthèse . . . 33
1.7 Conclusion . . . 34

2 Architecture pour une grappe de bases de données **39**
2.1 Introduction . . . 39
2.2 Architecture en cinq couches . . . 40
2.2.1 Routeur de requêtes . . . 41
2.2.2 Gestionnaire d'applications . . . 41
2.2.3 Équilibrage des transactions . . . 42
2.2.4 Gestionnaire de réplication . . . 42
2.2.5 Base de données . . . 42
2.2.6 Services partagés . . . 43
2.3 Exemple de routage de requête . . . 43
2.4 Les services réseaux . . . 45
2.5 Conclusion . . . 46

3 Algorithme de réplication préventive 47
3.1 Introduction 47
3.2 Modèle de cohérence 48
3.2.1 Maître-paresseux 48
3.2.2 Multimaîtres 48
3.2.3 Partielles 49
3.3 Réplication totale 49
3.3.1 Principe de l'algorithme 49
3.3.2 Exemple d'exécution 50
3.3.3 Architecture 51
3.3.4 Algorithmes 52
3.4 Réplication partielle 57
3.4.1 Principe de l'algorithme 57
3.4.2 Exemple d'exécution 58
3.4.3 Gestion des blocages 59
3.4.4 Architecture 59
3.4.5 Algorithmes 61
3.5 Preuves 62
3.5.1 Réplication totale 65
3.5.2 Réplication partielle 65
3.5.3 Ordre total 66
3.5.4 Verrous mortels 66
3.6 Conclusion 67

4 Optimisation des temps de réponse 69
4.1 Introduction 69
4.2 Élimination du délai d'attente 69
4.2.1 Principe 70
4.2.2 Exemple 71
4.2.3 Algorithmes 71
4.3 Élection anticipée 76
4.4 Exécution en parallèle 76
4.4.1 Principe 77
4.4.2 Lectures incohérentes 78
4.4.3 Exemple 78
4.4.4 Algorithmes 79
4.5 Preuves 81
4.5.1 Élimination du délai d'attente 81
4.5.2 Élection anticipée des transactions 81
4.5.3 Exécution concurrente 82
4.6 Conclusion 82

5 Le prototype RepDB* 85
5.1 Introduction 85
5.2 Plateforme 85
5.2.1 Langage 85
5.2.2 La couche réseau 86
5.2.3 Gestion des journaux 87
5.2.4 Interface de communications avec les SGBD 88
5.3 Mise en oeuvre du prototype 92
5.3.1 PostgreSQL 92
5.3.2 Service réseau (Spread) 92
5.3.3 Synchronisation des horloges 93
5.3.4 RepDB* 93
5.3.5 Les clients 93
5.4 Conclusion 94

6 Validation 97
6.1 Introduction 97
6.2 Implémentation 98
6.3 Modèle de performance 98
6.3.1 Banc d'essai spécifique 98
6.3.2 Banc d'essai TPC-C 101
6.4 Performances 104
6.4.1 Passage à l'échelle 104
6.4.2 Gains de performance 106
6.4.3 Degré de fraîcheur 108
6.4.4 Impact sur les délais 109
6.4.5 Exécution en parallèle 111
6.5 Conclusion 113

Conclusion 115

Liste des publications 121

Bibliographie 123

Table des matières 131

A Code source de RepDB* 137
A.1 Refresher 137
A.2 Deliver 141

Annexes

ANNEXE A

Code source de RepDB*

A.1 Refresher

```
package org.atlas.repdb.replication;

import java.util.*;

import org.apache.log4j.*;

/**
 *
 * <p>Refresher is a thread which orders messages by timestamp.
 * The oredered value of a message is set to true when the message is
 * ordered.</p>
 *
 * <p>Copyright : Copyright (c) 2005</p>
 *
 * <p>Compagny : INRIA/LINA University of Nantes</p>
 *
 * @author Cédric Coulon
 * @version 1.0
 */
public class Refresher
    extends Thread {
  private Replication replication;

  // List of messages that wait for process
  // Each value of the table represent a pending queue (Vector)
  // that contain messages from a node
  private Hashtable messages;

  // true if a message is elected to be ordered
  private boolean elu;

  // true if all the pending contain at least a message
  private boolean allFull;

  // Value of the elected message for ordering
  private Message mesElu;

  // Time to wait before mesElu been ordered
  private long sleepTime = 0;
```

```
40
41   private static Logger log = Logger.getLogger(Refresher.class.getName());
42
43   private static long INFINITE = -1;
44
45   public Refresher(Replication replication) {
46     this.replication = replication;
47     elu = false;
48     allFull = false;
49     messages = new Hashtable();
50     NodeInfo me = replication.getMyInfo();
51     for (Enumeration e = replication.getNodeInfo().elements();
52          e.hasMoreElements(); ) {
53       NodeInfo node = (NodeInfo) e.nextElement();
54       // Create the pending queues: one vector by origin node
55       if (replication.haveOneCopy(me.getMasters(), node.getMasters()) ||
56           replication.haveOneCopy(me.getSlaves(), node.getMasters())) {
57         messages.put(node.getName(), new Vector());
58       }
59     }
60   }
61
62   public void run() {
63     Message mes;
64     synchronized (this) {
65       while (true) {
66         try {
67           if (sleepTime == INFINITE) {
68             // If no message is elected sleep forever (until a message
                   arrives)
69             mes = replication.getWaitingQueue();
70           }
71           else {
72             // If a message is elected sleep until timestamp+max elapsed
73             mes = replication.getWaitingQueue(sleepTime);
74           }
75           if (mes != null) {
76             // If a new message is arrived, append it in its corresponding
77             // pnding queue
78             Vector v = (Vector) messages.get(mes.getNodeFrom());
79             v.add(mes);
80           }
81
82           // Elect a message
83           elect();
84
85           // If the message is ready to be ordered
86           while (allFull ||
87                  (elu &&
88                   (replication.currentTimeMillis() >=
89                    mesElu.getTimeStamp() + replication.getCommMax()))) {
90             // Set the message ordered
```

```
91              execute();
92              // Elect a new message
93              elect();
94            }
95          }
96          catch (InterruptedException ie) {
97            return;
98          }
99          catch (Exception e) {
100           log.error(e.toString());
101         }
102       }
103     }
104   }
105
106   /**
107    * Set the mesElu message as ordered.
108    *
109    * @throws Exception
110    */
111   private void execute() throws Exception {
112     mesElu.setTimeStampOrdered(replication.currentTimeMillis());
113     log.debug(mesElu);
114     replication.putRunningQueue(mesElu.copy(Message.ORDERED));
115     Vector v = (Vector) messages.get(mesElu.getNodeFrom());
116     v.remove(0);
117     elu = false;
118     mesElu = null;
119   }
120
121   /**
122    * Elect the message with the oldest timestamp as next message to be
          ordered.
123    *
124    * @throws Exception
125    */
126   private void elect() throws Exception {
127     allFull = true;
128     for (Enumeration e = messages.elements(); e.hasMoreElements(); ) {
129       Vector v = (Vector) e.nextElement();
130       if (!v.isEmpty()) {
131         Message mes = (Message) v.firstElement();
132         if (!elu || mes.isLower(mesElu)) {
133           // Set the mes with the lowest timestamp as mesElu
134           elu = true;
135           mesElu = mes;
136         }
137       }
138       else {
139         allFull = false;
140       }
141     }
```

```
142     // If no message is elected sleep forever (until a message arrives)
143     sleepTime = INFINITE;
144     if (elu) {
145       // If a message is elected sleep until timestamp+max elapsed
146       sleepTime = mesElu.getTimeStamp() + replication.getCommMax() -
147           replication.currentTimeMillis();
148     }
149     if (sleepTime <= 0) {
150       sleepTime = INFINITE;
151     }
152   }
153 }
```

A.2 Deliver

```
1  package org.atlas.repdb.replication;
2
3  import java.sql.*;
4  import java.util.*;
5
6  import org.apache.log4j.*;
7  import org.atlas.repdb.utils.*;
8
9  /**
10  *
11  * <p>Deliver is a thread which receives write transactions,
12  * and then dispatch them in a available DeliverThread thread which
       performs
13  * the query.</p>
14  *
15  * <p>Copyright : Copyright (c) 2005</p>
16  *
17  * <p>Compagny : INRIA/LINA University of Nantes</p>
18  *
19  * @author Cédric Coulon
20  * @version 1.0
21  */
22 public class Deliver
23     extends Thread {
24   private Replication replication;
25
26   // Number of currently used Deliver Thread
27   private int nbDeliver;
28
29   // List of the Deliver Thread
30   // Deliver Threads from 0 to nbDeliver-1 are in use
31   private DeliverThread deliverList[];
32
33   // List of transactions to perform
34   private Vector mesWaitToPerform;
35
36   // List of messages from others modules
37   Vector listMes;
38
39   // Each entry correspond to an message in listMes. It contains a pointer
        on
40   // the deliver thread which sent the message. If the message does not
        come
41   // from a deliver thread, the pointer is null.
42   Vector listMesDt;
43
44   private static Logger log = Logger.getLogger(Deliver.class.getName());
45
46   public Deliver(Replication replication) throws Exception {
```

```
47      this.replication = replication;
48      mesWaitToPerform = new Vector();
49      listMes = new Vector();
50      listMesDt = new Vector();
51      nbDeliver = 0;
52
53      // The number of simalute reads authorized is define by nbMaxDeliver.
54      deliverList = new DeliverThread[replication.getNbMaxDeliver()];
55      for (int i = 0; i < deliverList.length; i++) {
56        // For each DeliverThread, connect to the DBMS
57        deliverList[i] = new DeliverThread(i,
58                                           replication.getJdbcConnect(),
59                                           replication.getJdbcLogin(),
60                                           replication.getJdbcPassword());
61        deliverList[i].start();
62      }
63    }
64
65    /**
66     * Return true if the transaction contained in mes is conflicting with
67     * at least one of the transactions in deliverList before the position
          pos.
68     *
69     * @param mes Message
70     * @param pos int position of the message in the deliverList
71     * @return boolean
72     */
73    private boolean isThreadConflict(Message mes, int pos) {
74      boolean found = false;
75      for (int i = 0; !found && i < pos; i++) {
76        found = mes.isConflict(deliverList[i].mes);
77      }
78      return found;
79    }
80
81    /**
82     * Insert a message in a Vector of message according to its timestamp.
83     *
84     * @param message Vector
85     * @param mes Message
86     */
87    private static void insert(Vector message, Message mes) {
88      for (int i = 0; i < message.size(); i++) {
89        Message m = (Message) message.get(i);
90        if (mes.isLower(m)) {
91          message.add(i, mes);
92          return;
93        }
94      }
95      message.add(mes);
96    }
97
```

```
98    /**
99     * Convert a writeset in SQL command and put it the SQL content of a
         message.
100    * The function only keeps the updates on the tables hold by the node.
101    *
102    * @param ws Vector Vector of LogMessage (the writeset)
103    * @param mes Message Message where to set the SQL content
104    */
105   public void applyWriteSet(Vector ws, Message mes) {
106     mes.setWriteSet(ws);
107     mes.setSql("");
108     for (Enumeration e = ws.elements(); e.hasMoreElements(); ) {
109       LogMessage lm = (LogMessage) e.nextElement();
110       if (replication.haveCopy(lm.copy)) {
111         mes.setSql(mes.getSql() + lm.toString());
112       }
113     }
114   }
115
116   /**
117    * Read message from different module: Refresher, Receiver, DeliverThread
         .
118    * Return true if a new message has been received.
119    *
120    * @return boolean
121    * @throws Exception
122    */
123   public boolean readMes() throws Exception {
124     boolean foundMes = false;
125     Message mes;
126
127     for (int i = 0; i < nbDeliver; i++) {
128       // Read message from the DeliverThreads
129       mes = (Message) deliverList[i].queueOut.poll(0);
130       if (mes != null) {
131         foundMes = true;
132         listMes.add(mes);
133         listMesDt.add(deliverList[i]);
134       }
135     }
136     // Read message from the Refresher
137     mes = (Message) replication.getOrderedQueue();
138     if (mes != null) {
139       foundMes = true;
140       listMes.add(mes);
141       listMesDt.add(null);
142     }
143     // Read message from the Receiver
144     mes = (Message) replication.getRunningQueue();
145     if (mes != null) {
146       foundMes = true;
147       listMes.add(mes);
```

```
148        listMesDt.add(null);
149      }
150      return foundMes;
151    }
152
153    public void run() {
154      Message m;
155      Message mes;
156      DeliverThread dt;
157
158      // List of ordered message waiting to process
159      Vector ordered = new Vector();
160
161      // List of writeset corresponding to a message waiting to process
162      Vector writeset = new Vector();
163
164      try {
165        while (true) {
166          // Read waiting messages and put them in ListMes
167          synchronized (this) {
168            if (!readMes()) {
169              wait();
170            }
171          }
172          readMes();
173
174          // Proccess messages
175          while (!listMes.isEmpty()) {
176            dt = (DeliverThread) listMesDt.remove(0);
177            mes = (Message) listMes.remove(0);
178
179            // Message from the receiver
180            if (mes.getType() == Message.WRITE) {
181              // Check if the message is not already ordered
182              for (int i = 0; i < ordered.size(); i++) {
183                m = (Message) ordered.get(i);
184                if (m.getKey().equals(mes.getKey())) {
185                  mes.setOrdered();
186                  ordered.remove(i);
187                }
188              }
189
190              // Check if a writeset is not already received for this message
191              for (int i = 0; i < writeset.size(); i++) {
192                m = (Message) writeset.get(i);
193                if (m.getKey().equals(mes.getKey())) {
194                  applyWriteSet(m.getWriteSet(), mes);
195                  writeset.remove(i);
196                }
197              }
198
199              // Add the message to the list of the messages to perform
```

```
200                 insert(mesWaitToPerform, mes);
201
202                 // Check if most recent messages have been already started
203                 for (int i = 0; i < nbDeliver; i++) {
204                   dt = deliverList[i];
205                   if (mes.isLower(dt.mes) && !dt.abort) {
206                     // Abort recent messages already started and insert a copy
                            in
207                     // the list of transaction according to its timestamp
208                     m = dt.mes.copy();
209                     m.setType(Message.WRITE);
210                     insert(mesWaitToPerform, m);
211                     dt.abort = true;
212                     dt.mes.setType(Message.ABORT);
213                     dt.put(dt.mes.copy(Message.ABORT));
214                   }
215                 }
216               }
217
218               // Message from the receiver
219               else if (mes.getType() == Message.WRITESET) {
220                 // If the message is already arrived, check
221                 boolean found = false;
222                 for (int i = 0; i < mesWaitToPerform.size(); i++) {
223                   m = (Message) mesWaitToPerform.get(i);
224                   if (mes.getKey().equals(m.getKey())) {
225                     // Copy the writeset in its corresponding message
226                     found = true;
227                     applyWriteSet(mes.getWriteSet(), m);
228                   }
229                 }
230
231                 // notify the Deliver Thread
232                 for (int i = 0; i < nbDeliver; i++) {
233                   dt = deliverList[i];
234                   if (!dt.abort && mes.getKey().equals(dt.mes.getKey())) {
235                     found = true;
236                     dt.put(mes);
237                   }
238                 }
239
240                 // If no message correspond to the writeset, then store the
241                 // writeset for a future process
242                 if (!found) {
243                   writeset.add(mes);
244                 }
245               }
246
247
248               // Message from the deliver thread
249               else if (mes.getType() == Message.ABORT) {
250                 for (int i = 0; i < nbDeliver - 1; i++) {
```

```
251       if (deliverList[i].id == dt.id) {
252         System.arraycopy(deliverList, i + 1, deliverList, i,
253                          deliverList.length - i - 1);
254         deliverList[deliverList.length - 1] = dt;
255         i = nbDeliver;
256       }
257     }
258     nbDeliver--;
259     dt.abort = false;
260   }
261 
262   // Message from the deliver thread
263   else if (mes.getType() == Message.COMMITTED) {
264     System.arraycopy(deliverList, 1, deliverList, 0,
265                      deliverList.length - 1);
266     deliverList[deliverList.length - 1] = dt;
267     nbDeliver--;
268     dt.abort = false;
269   }
270 
271   // Message from the refresher
272   else if (mes.getType() == Message.ORDERED) {
273     boolean found = false;
274 
275     // Set the corresponding message ordered
276     for (int i = 0; i < nbDeliver; i++) {
277       dt = deliverList[i];
278       if (mes.getKey().equals(dt.mes.getKey()) &&
279           !dt.abort && !dt.mes.getOrdered()) {
280         found = true;
281         dt.mes.setOrdered();
282 
283         // notify the Deliver Thread
284         dt.put(dt.mes.copy(mes.getType()));
285       }
286     }
287     for (int i = 0; i < mesWaitToPerform.size(); i++) {
288       m = (Message) mesWaitToPerform.get(i);
289       if (mes.getKey().equals(m.getKey())) {
290         found = true;
291         m.setOrdered();
292       }
293     }
294     // If the original message is not yep ordered
295     // store the ordered message
296     if (!found) {
297       ordered.add(mes);
298     }
299   }
300 }
301 
302 // Allow the first deliver thread to perform
```

```
303         if (nbDeliver > 0) {
304           dt = deliverList[0];
305           if (!dt.canPerform) {
306             dt.canPerform = true;
307             dt.put(dt.mes.copy(Message.CAN_PERFORM));
308           }
309           if (!dt.canCommit) {
310             dt.canCommit = true;
311             dt.put(dt.mes.copy(Message.CAN_COMMIT));
312           }
313         }
314 
315         // Allow others thread to perform
316         for (int i = 1; i < nbDeliver; i++) {
317           dt = deliverList[i];
318           // Allow the deliver to perform a transaction: if the thread is
                not
319           // conflicting with already started thread and if the node has
                all
320           // the necessary copies
321           if (!dt.canPerform &&
322               !isThreadConflict(dt.mes, i) &&
323               !Replication.haveOneCopy(replication.getMyInfo().getSlaves(),
324                                        dt.mes.getWrites())) {
325             dt.canPerform = true;
326             dt.put(dt.mes.copy(Message.CAN_PERFORM));
327           }
328         }
329 
330         // Start a new waiting message if a Deliver Thread is available
331         while (!mesWaitToPerform.isEmpty() && nbDeliver < replication.
             getNbMaxDeliver()) {
332           mes = (Message) mesWaitToPerform.remove(0);
333           boolean canPerform = !Replication.haveOneCopy(replication.
                getMyInfo().
334                getSlaves(), mes.getWrites()) &&
335               !isThreadConflict(mes, nbDeliver);
336           boolean canCommit = (nbDeliver == 0);
337           boolean needWriteSet = !mes.haveWriteSet() &&
338               !Replication.haveAllCopies(replication.getMyInfo(), mes);
339 
340           mes.setTimeStampDeliver(replication.currentTimeMillis());
341           deliverList[nbDeliver].init(mes, canPerform, canCommit,
                needWriteSet);
342           deliverList[nbDeliver].put(mes.copy(Message.WRITE));
343           nbDeliver++;
344         }
345       }
346     }
347     catch (Exception e) {
348       System.out.println(Replication.stack2string(e));
349     }
```

```
350   }
351
352   /**
353    *
354    * <p>Thread used to connect to DBMS and perform update transactions.</p>
355    *
356    * <p>Copyright : Copyright (c) 2005</p>
357    *
358    * <p>Compagny : INRIA/LINA University of Nantes</p>
359    *
360    * @author Cédric Coulon
361    * @version 1.0
362    */
363   public class DeliverThread
364       extends Thread {
365     // JDBC variables
366     Connection connection;
367     Statement statement;
368
369     // Message containing the transaction to perform
370     Message mes;
371
372     // true if the transaction can commit
373     boolean canCommit;
374
375     // true if the transaction can perform
376     boolean canPerform;
377
378     // true if the transaction need a writeset to perform
379     boolean needWriteSet;
380
381     // true if the transaction should abort
382     boolean abort;
383
384     // id of the deliver thread
385     int id;
386
387     // Queue used to receive messages from the Deliver
388     Queue queueIn;
389
390     // Queue used to send messages from the Deliver
391     Queue queueOut;
392
393     /**
394      * Initialise a Deliver Thread.
395      *
396      * @param id int id of the Deliver Thread
397      * @param connect String JDBC connect string
398      * @param login String JDBC login string
399      * @param passwd String JDBC passwd string
400      * @throws SQLException
401      */
```

```
402     public DeliverThread(int id, String connect, String login, String
            passwd) throws
403          SQLException {
404       connection = DriverManager.getConnection(connect, login, passwd);
405       connection.setAutoCommit(false);
406       statement = connection.createStatement();
407       queueIn = new Queue();
408       queueOut = new Queue();
409       this.id = id;
410     }
411
412     /**
413      * Allocate a new transaction (contained in a message) to a deliver
            thread
414      *
415      * @param mes Message contains teh transaction
416      * @param canPerform boolean true if the thread is allowed to perform
417      * @param canCommit boolean true if the thread is allowed to commit
418      * @param needWriteSet boolean true if the transaction need writeset
            before
419      * to perform
420      */
421     public void init(Message mes, boolean canPerform, boolean canCommit,
422                      boolean needWriteSet) {
423       this.canCommit = canCommit;
424       this.canPerform = canPerform;
425       this.needWriteSet = needWriteSet;
426       this.mes = mes;
427       abort = false;
428     }
429
430     /**
431      * Receive a message from the Deliver
432      *
433      * @param m Message
434      * @throws Exception
435      */
436     public void put(Message m) throws Exception {
437       queueIn.put(m);
438     }
439
440     /**
441      * Start the thread
442      */
443     public void run() {
444       // JDBC result variables, the type of variable used depends of the
            JDBC
445       // command type
446       int intResult;
447       boolean booleanResult;
448       ResultSet rsResult;
449
```

```
450         // true if the transaction has already been performed
451         boolean hasPerformed;
452
453         Message m;
454
455         try {
456           while (true) {
457             hasPerformed = false;
458
459             // Receive a new transaction to perform
460             // Other type of messages than WRITE are discarded
461             m = (Message) queueIn.take();
462             if (m.getType() == Message.WRITE) {
463               while (!abort && (!canPerform || needWriteSet)) {
464                 // If the tranasction can not be performed, wait a message
                        from
465                 // Deliver
466                 m = (Message) queueIn.take();
467                 if (m.getType() == Message.WRITESET) {
468                   // Replace the content of the transaction by the content of
                          the
469                   // writeset
470                   applyWriteSet(m.getWriteSet(), mes);
471                   needWriteSet = false;
472                 }
473                 else if (m.getType() == Message.CAN_PERFORM ||
474                          m.getType() == Message.CAN_COMMIT) {
475                   // Nothing to do, Deliver has already changed the state of
476                   // canCommit and canPerform
477                 }
478               }
479
480               // Perform the transaction according to the type of JDBC
                    command
481               if (!abort) {
482                 hasPerformed = true;
483                 try {
484                   // If the current node is the origin node, store the
                        writeset
485                   if (mes.getNodeFrom().equals(mes.getNode())) {
486                     int tid = replication.getTID(statement);
487                     if (tid != -1) {
488                       replication.newLogEntry(tid, mes.getKey());
489                     }
490                   }
491
492                   if (mes.getCmd().equals("executeQuery")) {
493                     rsResult = statement.executeQuery(mes.getSql());
494                     // If the current node is the origin node,
495                     // set the answer for the client (Replica Interface
                          thread)
496                     if (mes.getThread() != null) {
```

```
497                     mes.getThread().setResultSet(rsResult);
498                   }
499                 }
500                 else if (mes.getCmd().equals("execute")) {
501                   booleanResult = statement.execute(mes.getSql());
502                   // If the current node is the origin node,
503                   // set the answer for the client (Replica Interface
                          thread)
504                   if (mes.getThread() != null) {
505                     mes.getThread().setBResult(booleanResult);
506                   }
507                 }
508                 else if (mes.getCmd().equals("executeUpdate")) {
509                   intResult = statement.executeUpdate(mes.getSql());
510                   // If the current node is the origin node,
511                   // set the answer for the client (Replica Interface
                          thread)
512                   if (mes.getThread() != null) {
513                       // If the current node is the origin node,
514                       // set the answer for the client (Replica Interface
                              thread)
515                     mes.getThread().setResult(intResult);
516                   }
517                 }
518               }
519               catch (SQLException sqle) {
520                 // If the current node is the origin node,
521                 // set the exception for the client (Replica Interface
                        thread)
522                 if (mes.getThread() != null) {
523                   mes.getThread().setSQLException(sqle);
524                 }
525                 sqle.printStackTrace();
526               }
527
528               // Wait until the transaction is aborted or
529               // ordered and commitable
530               while (!abort && (!canCommit || !mes.getOrdered())) {
531                 // Wait a notification from the Deliver
532                 m = (Message) queueIn.take();
533                 if (m.getType() == Message.ORDERED ||
534                     m.getType() == Message.CAN_COMMIT) {
535                     // Nothing to do, Deliver has already changed the state
                            of
536                     // canCommit and ordered (ordered is stored in the
                          message)
537                 }
538               }
539             }
540
541             // Abort the message
542             if (abort) {
```

```
543                 if (hasPerformed) {
544                   // If the message has already been performed, rollback
545                   connection.rollback();
546                   mes.setTimeStampDelivered(replication.currentTimeMillis());
547                   log.debug(mes);
548                 }
549                 // Notify the Deliver
550                 queueOut.put(mes.copy(Message.ABORT));
551                 synchronized (replication.getDeliver()) {
552                   replication.getDeliver().notify();
553                 }
554               }
555               else {
556                 // Commit the transaction
557                 connection.commit();
558                 mes.setTimeStampDelivered(replication.currentTimeMillis());
559                 log.debug(mes);
560                 abort = false;
561 
562                 // Notify the Deliver
563                 queueOut.put(mes.copy(Message.COMMITTED));
564                 synchronized (replication.getDeliver()) {
565                   replication.getDeliver().notify();
566                 }
567 
568                 //Send the answer to client (Replica Interface thread)
569                 if (mes.getNodeFrom().equals(replication.getName())) {
570                   if (mes.getThread() != null) {
571                     synchronized (mes.getThread()) {
572                       mes.getThread().notify();
573                     }
574                   }
575                 }
576               }
577             }
578           }
579         }
580         catch (Exception e) {
581           System.out.println(Replication.stack2string(e));
582         }
583       }
584     }
585 }
```

Réplication Préventive dans une grappe de bases de données

Cédric COULON

Résumé

Dans une grappe de bases de données, la réplication préventive peut fournir une cohérence forte sans les limitations d'une réplication synchrone. Dans cette thèse, nous présentons une solution complète pour la réplication préventive qui supporte les configurations multimaîtres et partielles, où les bases de données sont partiellement répliquées sur différents noeuds. Pour augmenter le débit des transactions, nous proposons une optimisation qui élimine le délai d'attente pour l'ordonnancement en contrepartie d'un petit nombre d'abandon des transactions et nous introduisons le rafraîchissement parallèle des copies. Nous décrivons des expérimentations à grande échelle de notre algorithme basées sur notre prototype (RepDB*) sur une grappe de 64 noeuds utilisant le SGBD PostgreSQL. Nos résultats utilisant le banc d'essai TPC-C montrent que notre approche dispose d'un excellent passage à l'échelle et d'une excellente amélioration du débit.

Mots-clés : Bases de données, réplication, cohérence forte, grappe de PC, TPC-C Benchmark

Abstract

In a database cluster, preventive replication can provide strong consistency without the limitations of synchronous replication. In this thesis, we present a full solution for preventive replication that supports multi-master and partial configurations, where databases are partially replicated at different nodes. To increase transaction throughput, we propose an optimization that eliminates delay at the expense of a few transaction aborts and we introduce concurrent replica refreshment. We describe large-scale experimentation of our algorithm based on our RepDB* prototype over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that the proposed approach yields excellent scale-up and speed-up.

Keywords: Databases, replication, strong consistency, cluster of PC, TPC-C Benchmark

Classification ACM

Catégories et descripteurs de sujets : Num CR [**catégorie**]: sous-catégorie—*descripteur*; Num CR [**catégorie**]: sous-catégorie—*descripteur*

Termes généraux : terme1, terme2