Routage des Transactions dans les Bases de Données à Large Echelle

Idrissa Sarr

N° d'ordre: 000

THÈSE

présentée devant

l'Université Pierre et Marie Curie (Paris VI)

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ PIERRE ET MARIE CURIE
Mention INFORMATIQUE

par

Idrissa SARR

Équipe d'accueil : Bases de Données
École Doctorale : EDITE
Composante universitaire : LABORATOIRE D'INFORMATIQUE DE PARIS 6

Titre de la thèse :

Routage des Transactions dans les Bases de Données à Large Echelle

date de soutenance prévue : 07 octobre 2010

Rapporteurs :	Esther	PACITTI	Professeure à l'Université de Montpellier 2
	Rachid	GUERRAOUI	Professeur à l'EPFL
Examinateurs :	Pierre	SENS	Professeur à l'UPMC
	Stéphane	GANÇARSKI	Maître de Conférences à l'UPMC (HDR)
	Gabriel	ANTONIU	Chargé de Recherche à l'INRIA de Rennes (HDR)
	Samba	NDIAYE	Maître Assistant à l'UCAD
Directeur de thèse :	Anne	DOUCET	Professeure à l'UPMC
Encadrant :	Hubert	NAACKE	Maître de Conférences à l'UPMC

Table des matières

Chapitre 1

Introduction

1.1 Motivations

L'Internet, réseau mondial d'ordinateurs, fournit une infrastructure pour le stockage des données et facilite le partage de très grands volumes de données. Cette facilité de partage d'information est une des principales clés du succès des applications Web et particulièrement celles dénommées applications Web 2.0. Les applications Web2.0 sont ouvertes car elles peuvent être complétées par des applications tierces qui ajoutent du contenu ou qui accèdent aux données de l'application d'origine. On peut alors considérer qu'une application Web2.0 joue le rôle de pépinière d'applications. De plus, une application Web2.0 contrôle les données manipulées par les utilisateurs directement ou via les applications tierces. Plus précisément, elle gère l'exécution des transactions émises par les utilisateurs ou les applications tierces.

Aujourd'hui, beaucoup d'applications Web2.0 voient leur nombre d'utilisateur croître fortement pour atteindre des centaines de millions de personnes (e.g., début 2010, Facebook aurait 400 millions de membres actifs). Face à cette forte croissance, les applications Web2.0 sont confrontées à un problème de passage à l'échelle. Notamment, le système de gestion de données atteint rapidement les limites de la charge qu'il peut traiter. Cela peut provoquer une dégradation des temps de réponses perçus par l'utilisateur, situation qu'il n'accepte pas. Pour éviter une telle situation, il est nécessaire d'augmenter les ressources (calcul, stockage, communication) utilisées par les applications Web2.0.

L'ajout de ressource se faisant par étapes et progressivement, l'infrastructure obtenue est composée de ressources hétérogènes (diverses capacités de stockage, calcul et communication) réparties à l'échelle mondiale. Du fait de sa taille et de son étendue, l'infrastructure est sujette à de nombreuses pannes dont il faut tenir compte en prévoyant la redondance des données et des fonctions. Dans cette thèse, nous apportons des solutions aux problèmes soulevés par la manipulation de données réparties et répliquées à très large échelle.

1.2 Objectifs et Contexte de la thèse

Cette thèse s'inscrit en partie dans le cadre du projet ANR Respire [Prod] réalisé entre 2006 et 2009, et dont l'objectif principal est d'offrir des fonctionnalités avancées pour le partage de ressources et de services dans les systèmes pair à pair, en présence de données répliquées.

La réplication dans les bases de données a été largement étudiée, au cours des trois dernières décennies. Elle vise à améliorer la disponibilité des données et à augmenter la performance d'accès aux données. Un des défis majeurs de la réplication est de maintenir la cohérence mutuelle des répliques, lorsque plusieurs d'entre elles sont mises à jour, simultanément, par des transactions. Des solutions qui relèvent partiellement ce défi pour un nombre restreint de bases de données reliées par un réseau fiable existent. Toutefois, ces solutions ne sont pas applicables à large échelle car elles nécessitent une communication rapide et fiable entre les nœuds traitant les transactions ; ce qui limite le nombre de nœuds (de l'ordre d'une centaine) et le type d'interconnexion (réseau local).

Dans cette thèse nous nous intéressons à la gestion des transactions dans une base de données répliquées à large échelle et particulièrement au *routage des transactions*. Nos principaux objectifs peuvent être résumés comme suit :

- réduire le temps de réponse des transactions, en équilibrant la charge des répliques et en tenant compte de la disponibilité des ressources (SGBD, gestionnaire de transactions).
- contrôler la cohérence des accès aux données réparties et répliquées afin de rendre aux applications des résultats conformes à leur exigence ;
- garantir l'autonomie des applications et des SGBD *i.e.* pouvoir intégrer les solutions proposées avec des applications et SGBD existants en les modifiant le moins possible.

1.3 Problématiques

Dans les systèmes déployés à grande échelle comme Facebook ou eBay, le nombre de sources de données, répliques incluses, est très élevé (plus de 10.000 pour eBay). Face à cette situation, plusieurs problèmes se posent parmi lesquels le problème de routage que nous tentons de résoudre et spécifions comme suit :

1. Soit un ensemble de bases de données réparties et répliquées, et une transaction envoyée par une application, le problème du routage de transactions se pose ainsi : sur quelle réplique décide-t-on de traiter la transaction et quel traitement doit être effectué sur la base choisie, afin de satisfaire au mieux les objectifs de cohérence et de performance ? Notons que le routage ne consiste pas à faire un ordonnancement des transactions entrantes mais il s'agit plutôt de faire une optimisation du traitement de la transaction.

   Ce problème de routage est très important d'autant plus qu'aujourd'hui les machines hébergeant les données peuvent être fournies par un service tiers tel que les ASP (*Application Service Provider*) ou les clouds. Ainsi, pour un fournisseur de service cloud ou ASP, il devient indispensable d'avoir des mécanismes efficaces pour identifier les données d'une application particulière regroupées avec les données d'autres applications pour des soucis économiques ;

2. Le problème du choix de la réplique pose la question de maintenir l'état du système : quelle information sur les répliques doit-on connaitre pour en choisir une qui permettra de traiter la transaction rapidement ? Dans un système décentralisé et de grande taille, comment disposer de cette information au bon endroit sachant que plusieurs demandes de transactions peuvent arriver simultanément ?

   L'information sur les répliques dénommée souvent métadonnées, requiert une gestion minutieuse puisqu'elle détermine à tout instant l'état global du système. Dans un environnement à grande échelle l'utilisation des métadonnées est indispensable pour exploiter les ressources qui composent le système. De plus, la gestion des métadonnées fait partie des problèmes les plus important pour garantir les performances puisque la largeur d'échelle implique beaucoup de métadonnées. C'est pourquoi nous nous intéressons à ce problème.

1.4 Contributions

Nos contributions dans cette thèse sont résumées en cinq points décrits ci-après.

Architecture globale du système de routage

Nous concevons un intergiciel pour contrôler l'accès à la base de données. L'architecture de l'intergiciel est composée de deux parties : une partie assurant le service de médiation entre les différents composants du système et une autre chargée de la gestion des métadonnées qui sont les données nécessaires au fonctionnement du système en entier. Notre architecture est à mi-chemin entre les systèmes P2P structurés et ceux non structurés afin de tirer profit à la fois des avantages des uns et des autres. De fait, les nœuds chargés d'assurer le routage sont organisés autour d'un anneau logique, ce qui facilite leur collaboration et garantit le traitement cohérent des transactions. Par contre les nœuds chargés de stocker les données sont faiblement structurés, ce qui leur confère une grande autonomie. Notre intergiciel est redondant pour mieux faire face à la volatilité d'un environnement à large échelle puisqu'à chaque fois qu'un nœud chargé de routage ou de stockage des données tombe en panne, nous utilisons un autre nœud disponible pour continuer le traitement ou récupérer les données.

Gestion d'un catalogue réparti

Une exploitation optimale d'une base de données distribuée dans un système à large échelle requiert un service d'indexation (ou catalogue) des ressources qui stockent les données. Le catalogue est conçu pour héberger des informations utiles à l'exécution rapide et cohérente des transactions. Nous concevons uns structure qui permet de garder les informations telles que le schéma d'allocation des données, l'historique de l'exécution des transactions, l'état des répliques et leur disponibilité. Notre choix de garder uniquement ces informations se justifie par le fait que :1) elles sont suffisantes pour pouvoir router une transaction tout en contrôlant la cohérence des répliques ; et 2) elles sont suffisamment compactes pour ne pas trop allonger le temps de réponse.

Pour des soucis de performances, nous fragmentons le catalogue en fonction des classes de conflits. Concrètement un fragment contient les métadonnées décrivant une seule classe de conflit. Cela rend possible l'accès au catalogue en parallèle pour router deux transactions qui ne partagent pas la même classe de conflit. Une classe de conflit contient les données que la transaction peut potentiellement lire (resp. modifier)

Pour garantir la cohérence du catalogue lors de l'accès au métadonnées, nous proposons deux approches : une approche utilisant le verrouillage et une autre sans verrouillage.

La gestion avec verrouillage permet de garantir la cohérence des métadonnées et s'avère très efficace tant que les accès concurrents aux métadonnées restent peu fréquents. Malheureusement, nous vérifions que le verrouillage ne facilite pas le passage à l'échelle. C'est pourquoi nous optons pour une solution sans verrouillage lors de l'accès au catalogue. De fait, nous avons conçu un protocole pour coordonner les accès aux métadonnées sans avoir recours au verrouillage. Ce protocole offre les mêmes garanties de cohérence que le verrouillage et supporte beaucoup mieux les accès concurrents aux métadonnées.

Routage des transactions à travers un intergiciel

L'antinomie entre les besoins de performances et ceux de cohérence étant bien connue, l'approche suivie dans cette thèse consiste à relâcher les besoins de cohérence afin d'améliorer la performance d'accès aux données . Autrement dit, il s'agit de relâcher la fraîcheur pour diminuer le temps de réponse des transactions de lecture seule. Or, dans le contexte du Web2.0, de nombreuses applications tolèrent le relâchement de la fraîcheur *i.e.* , acceptent de lire des données qui ne sont pas nécessairement les plus récentes. Par exemple, il est possible de gérer des transactions de vente aux enchères (sur eBay ou Google Adsense) sans nécessairement accéder à la dernière proposition de prix, puisque l'enchère est sous pli cacheté. Le relâchement de la fraîcheur ouvre la voie vers de nouvelles solutions offrant de meilleures performances en termes de débit transactionnel, latence, disponibilité des données et passage à l'échelle.

Nous proposons des mécanismes de routage des transactions pour garantir une exécution cohérente et rapide des transactions en utilisant un modèle de coût. Le modèle de coût utilise le relâchement de la fraîcheur afin de réduire le temps de réponse et d'équilibrer les charges. Nos algorithmes requièrent des accès au catalogue réparti pour maintenir la cohérence mutuelle à terme et ils définissent l'ordre dans lequel les transactions doivent être exécutées pour ne pas compromettre la cohérence.

Le premier algorithme est dit pessimiste et ordonne toutes les transactions conflictuelles en s'appuyant sur les conflits potentiels. En d'autres mots, le protocole de routage assure une sérialisation globale définie de manière pessimiste et qui est utilisé pour router toutes les transactions. Chaque transaction est associée à une ou plusieurs classes de conflits. En fonction des classes de conflits, les transactions sont ordonnées dans un graphe en s'appuyant sur leur ordre d'arrivée. Bien que cette approche assure une sérialisation globale, il réduit malheureusement la parallélisation du traitement des transactions puisqu'elle s'appuie sur des sur-ensembles potentiels de données réellement accédées.

Pou améliorer le parallélisme du traitement des transactions, nous avons proposé un second algorithme qui combine une approche pessimiste et optimiste. Ce second algorithme s'appuie sur

une tentative d'exécution des transactions afin de minimiser davantage le temps de réponse des transactions. Autrement dit, les transactions accédant aux mêmes classes de conflit sont exécutées de manière optimiste et une phase de validation est utilisée à la fin pour garantir la cohérence. Dans le contexte des applications Web 2.0 où les transactions courantes et potentiellement conflictuelles, sont peu nombreuses, les chances de réussite du routage optimiste s'avèrent très élevées, ce qui fait qu'il soit plus adapté. De plus, nous mettons à jour le graphe de sérialisation pour le rendre plus précis.

L'une des caractéristiques de notre intergiciel est qu'il assure une transparence complète de la distribution des ressources en jouant le rôle d'interface entre les applications et les données. Les transactions envoyées par les applications sont transmises aux sources de données par l'intergiciel, mais les résultats sont directement envoyés aux applications. Ce protocole de communication permet à notre architecture de s'écarter de la structure client/serveur et de faciliter aussi le passage à l'échelle.

Gestion des pannes

Nous proposons également un mécanisme de gestion des pannes. Ce mécanisme est basé sur la détection sélective des fautes et sur un algorithme de reprise des transactions. Contrairement à la plupart des autres approches, notre mécanisme n'implique pas l'utilisation de nœuds qui ne participent pas à l'exécution de la transaction en cours, ce qui permet de passer à l'échelle. Pour cela, nous adaptons des approches existantes de détection des pannes afin de les rendre opérationnelles pour chaque type de nœud (gestionnaire de transaction et nœud de données) de notre système. Nous avons proposé un protocole permettant de gérer toutes les situations lorsqu'un nœud quitte le système pendant le traitement d'une transaction. Ceci est nécessaire et suffisant pour contrôler la cohérence du système, surtout en cas de déconnexions intempestives.

Cependant, pour maintenir le débit transactionnel en cas de fréquentes pannes, il faut être capable d'ajouter de nouvelles ressources en fonction des déconnexions. Pour ce faire, nous avons proposé un modèle permettant de déterminer et contrôler le nombre de répliques requises pour garder le système disponible. Ce modèle permet de déterminer le nombre minimum de répliques nécessaires au bon fonctionnement du système et donc de minimiser les surcoûts liés à la gestion des répliques. Notons que les pannes des noeuds sont détectées et prises en compte de manière complètement transparent aux applications.

Validation et résultats

Pour valider la faisabilité de nos approches, nous implémentons deux prototypes nommés respectivement DTR (*Distributed Transaction Routing*) et TRANSPEER (*TRANSaction on PEER-to-peer*). L'implémentation de deux prototypes est liée au besoin de gérer le catalogue avec verrouillage ou sans verrouillage. De fait, DTR constitue le prototype développé avec verrouillage du catalogue alors que TRANSPEER est conçu pour une gestion du catalogue sans verrouillage et un modèle de communication de type P2P. Puis, nous effectuons quelques simulations pour étudier le passage à l'échelle et la tolérance aux pannes de notre solution. Notre choix d'utiliser à la fois de l'expérimentation et de la simulation se justifie par le fait que : (1) l'expérimentation permet

d'évaluer un système dans des conditions réelles ; et (2) la simulation est une représentation simplifiée du système, facile à réaliser et requiert moins de ressources que l'implémentation, ce qui favorise l'évaluation d'un système à grande échelle. Nous avons mené une série d'expériences sur nos deux prototypes pour étudier les performances de notre système : débit transactionnel, temps de réponse, passage à l'échelle et tolérance aux pannes.

Les résultats obtenus pendant la thèse ont fait l'objet de plusieurs publications : [SNG08a, SNG10b, SNG10c, GSN09, SNG10a, SNG08b, DSS10].

Ces résultats montrent que l'utilisation d'un catalogue pour stocker les métadonnées permet de router les transactions en contrôlant le niveau de fraîcheur sollicité par les applications. De plus, la surcharge induite par la gestion d'un catalogue réparti est faible et donc n'a pas trop d'impact négatif sur le débit du routage. Les expériences ont montré que le relâchement de la fraîcheur des données améliore le temps de réponse des requêtes et l'équilibrage des charges, ce qui est économiquement important vis-à-vis de l'utilisation totale des ressources disponibles. Les résultats montrent également que la redondance du routeur accroît le débit de routage et réduit le temps de réponse tout en introduisant plus de disponibilité. Les résultats obtenus avec notre prototype TRANSPEER démontrent le gain de la gestion des métadonnées sans verrouillage, ce qui favorise la réduction du temps de réponse. Enfin, nous avons montré que la prise en compte de la dynamicité du système permet de maintenir les performances. Les méthodes de détection et de résolution des pannes utilisées sont simples à mettre en œuvre et s'avèrent bien adaptées pour un système à large échelle.

1.5 Organisation du manuscrit

Ce manuscrit est structuré en huit chapitres.

- Le chapitre 1 présente l'introduction de nos travaux.
- Le chapitre 2 présente les systèmes distribués. Nous y décrivons nos applications visées et l'architecture et le fonctionnement des systèmes distribués : caractéristiques, avantages et implémentation via un intergiciel. Puis nous faisons une discussion sur le modèle d'architecture requis pour gérer une base de données destinées à des applications à grande échelle.
- Le chapitre 3 décrit les travaux connexes au nôtre. Il présente quelques méthodes existantes pour gérer les transactions dans les bases de données répliquées. Plus précisément, nous étudions les solutions de gestion de transactions en privilégiant quatre caractéristiques des systèmes distribués que nous détaillons dans le chapitre 2 à savoir le passage à l'échelle, la tolérance aux pannes (ou disponibilité), la transparence et l'autonomie des données. Les solutions étudiées sont placées dans notre contexte afin de bien situer leurs limites mais aussi de bien comprendre les principes à mettre en œuvre pour mieux satisfaire les applications à large échelle.
- Le chapitre 4 présente l'architecture de notre système. Il détaille les différents composants du système de routage, leur rôle et leur interaction pour assurer le traitement des transactions.
- Le chapitre 5 décrit nos mécanismes de routage et de gestion du catalogue. Dans ce chapitre, nous détaillons les algorithmes de routage pour envoyer les transactions vers les répliques optimales et nous présentons les algorithmes proposés pour gérer le catalogue réparti afin

d'assurer sa cohérence .
– Le chapitre 6 présente la gestion des pannes. Nous y étudions la détection et la résolution des pannes afin de maintenir la cohérence et de borner le temps de réponse.
– Le chapitre 7 détaille notre validation. Nous y étudions les performances de notre système : débit transactionnel, temps de réponse, passage à l'échelle et tolérance aux pannes. Pour ce faire, nous étudions d'abord la surcharge liée à la gestion du catalogue, puis les performances globales du routage et enfin les apports de la gestion des pannes.
– Le chapitre 8 présente la conclusion et les perspectives de cette thèse.

Chapitre 2

Systèmes répartis à grande échelle

La gestion des données dans un environnement à grande échelle est indispensable pour prendre en compte les besoins des nouvelles applications telles que les applications Web 2.0. Si la gestion des données dans les systèmes distribués a été largement étudiée dans les dernières années, des solutions efficaces et à bas coût tardent à voir le jour à cause de la complexité des problèmes introduits par la largeur de l'échelle et le caractère hétérogène et dynamique de l'environnement. Plus particulièrement, l'étude des applications Web 2.0 nous a permis de comprendre leurs exigences à satisfaire. Ces exigences qui peuvent se résumer en un grand débit transactionnel, une haute disponibilité et une latence faible, nécessitent de nouvelles approches de concevoir et de gérer les systèmes distribués. Ceci se décline en deux problèmes qui sont, *i*) une bonne conception et implémentation des architectures réparties qui hébergent les services ; et *ii*) des mécanismes efficaces de gestion des données utilisées par ces applications. Dans ce chapitre, nous décrivons les spécifications d'une implémentation des systèmes distribués.

Nous décrivons d'abord nos applications visées avant de décrire l'architecture et le fonctionnement des systèmes distribués : caractéristiques, avantages et implémentation via un intergiciel. Puis, nous faisons une discussion sur le modèle d'architecture requis pour gérer une base de données destinées à des applications à grande échelle.

2.1 Applications Web 2.0

Dans cette section nous décrivons les applications Web 2.0 qui sont nos applications cibles. La définition du Web 2.0 est malaisée. Il n'en reste pas moins que l'ensemble des applications Web 2.0 présente des caractéristiques communes, parmi lesquelles nous pouvons citer :

- l'utilisation du réseau internet comme une plateforme puisque qu'elles interagissent avec les autres applications via des navigateurs ;
- l'offre d'un environnement collaboratif en donnant l'opportunité aux utilisateurs d'ajouter et de contrôler leur propre contenu ;
- la proposition de services permettant de mettre en relation des utilisateurs partageant des intérêts commun, par exemple les réseaux sociaux.

Les types d'applications Web 2.0 sont très nombreux, par exemple, weblogging, wikis, réseaux sociaux, podcasts, flux, etc. Pour tous ces exemples d'application, le défi majeur consiste à délivrer le service attendu par l'utilisateur et à assurer sa satisfaction en termes de fonctionnalités et de temps de réponse. De plus, si la mission première d'un site web consiste, par exemple, à "convertir les visiteurs en clients", son principal objectif peut être résumé comme étant "tout d'abord d'attirer suffisamment de visiteurs, puis de convertir suffisamment de visiteurs en clients".

Pour atteindre cet objectif, les applications Web 2.0 doivent satisfaire les exigences suivantes : haute disponibilité, forte réactivité (temps de réponse faible), maintenance ou évolution facile, chacune étant essentielle à son succès. L'environnement Web 2.0 offre de nouveaux moyens pour atteindre ces objectifs, mais croise également de nouveaux obstacles. En effet le nombre de visiteurs convertis en utilisateurs (ou clients) est très important et dépasse facilement une centaine de millions, par exemple plus de 200 millions pour Facebook, ou eBay ou Myspace, etc. Ce nombre d'utilisateurs engendre des téraoctets de données à gérer puisque chaque utilisateur peut éditer et contrôler son propre contenu. Par conséquent, le nombre de requêtes qui en découle correspond à des dizaines de milliards d'instructions par jour. De plus, certaines applications Web 2.0 fonctionnent en mode pair-à-pair, ce qui élargit la taille de l'application mais aussi les difficultés de gérer les données du fait de leur hétérogénéité.

Cependant cette forte charge applicative générée par les utilisateurs des applications Web 2.0 est très particulière comme le montre l'étude de Jean-Francois Ruiz [Rui]. En effet, parmi les utilisateurs d'une application Web 2.0, il y a 1% qui génèrent ou créent un contenu, 10% qui réagissent (commenter, améliorer, noter, voter) sur le contenu et 89% qui ne font que consulter, *i.e.* lire le contenu. En d'autres mots, la charge applicative est fortement dominées par les requêtes de lecture mais cela n'empêche pas que la charge d'écriture restante est supérieur à 30 millions d'instructions d'écritures par jour pour un site comme Facebook. Cette valeur est obtenue en supposant que chaque utilisateur ne fasse au plus qu'une opération d'écriture par jour.

Cette brève étude montre les fortes exigences des applications Web 2.0 qui sont entre autres, un grand débit transactionnel, une haute disponibilité, une latence faible, etc.

De plus, cette étude montre que la charge d'écriture intensive est due à un un volume important de données modifiées et non pas par une faible portion de données très fréquemment mises à jour (*hotspot*). Cette caractéristique des applications Web 2.0 qui est la conséquence directe de l'autorisation accordée aux utilisateurs à ne modifier que leurs propres données ou celle de leurs accointances est très importante car elle révèle que les conflits d'accès aux données sont rares dans ce contexte.

2.2 Notions de système distribués

Selon Tanenbaum [TS99], un système réparti est un ensemble d'ordinateurs (ou processus) indépendants qui apparaît à un utilisateur comme un seul système cohérent. Les ordinateurs peuvent garder leur autonomie et être regroupés dans un même lieu ou dispersés sans que cela ne soit visible de l'extérieur par un utilisateur. Du fait que l'ensemble des ordinateurs forment un système en entier, la défaillance d'un ordinateur peut avoir un impact négatif le fonctionnement du système et introduire des incohérences. En prenant en compte cet aspect, un système distribué peut

être défini comme "un système qui vous empêche de travailler si une machine dont vous n'avez jamais entendu parler tombe en panne, (*Leslie Lamport*). S'il existe moult définitions dont nous ignorons le nombre, on peut dire que les principaux objectifs des systèmes répartis sont de faire coopérer plusieurs ressources dans l'optique de partager des tâches, de faire des traitements parallèles, etc . Ainsi, un système distribué peut être vu comme une application qui coordonne les tâches de plusieurs équipements informatiques (ordinateurs, téléphones mobile, PDA, capteurs...). Cette coordination se fait le plus souvent par envoi de messages via un réseau de communication qui peut être un LAN (*Local Area Network*), WAN (*Wide Area Network*), Internet, etc.

Les systèmes distribués peuvent être classés de différentes manières et dans [TS99], trois classes ont été essentiellement identifiées à savoir les systèmes de calcul distribué (*Distributed Computing Systems*), les systèmes d'information distribués (*Distributed Information Systems*) et les systèmes pervasifs distribués (*Distributed Pervasive Systems*). Cette classification est axée essentiellement sur les domaines applicatifs des systèmes distribués plutôt que sur leur organisation interne (répartition géographique et support de communication) et les spécificités des ressources (hétérogénéité, stabilité des ressources, etc.). Pourtant, l'organisation et les caractéristiques des ressources sont des éléments essentiels qui permettent de bien prendre en compte les besoins spécifiques des applications en termes de performance. Ce faisant, nous avons fait un classement qui s'appuie plus sur l'organisation et les caractéristiques des ressources.

- la grappe d'ordinateur (*Cluster*) : c'est un ensemble d'ordinateurs connectés par un réseau LAN rapide et fiable pour assurer la disponibilité. Les ressources quasi-homogènes (même système d'exploitation, logiciels quasi-similaires) sont sous le contrôle d'un seul noeud appelé noeud maître ;
- la grille informatique (*Grid*) : c'est une collection d'ordinateurs hétérogènes réparties sur différents sites maintenus par plusieurs organisations. Les sites sont reliés par un réseau WAN et chacun d'entre eux contient plusieurs ressources informatiques administrées de manière autonome et uniforme ;
- les systèmes pair-à-pair (*peer-to-peer (P2P)*) : c'est un ensemble d'ordinateurs appelés pairs, qui s'accordent à exécuter une tâche particulière. Les ressources peuvent être des ordinateurs ou des assistants personnels interconnectés via Internet, ce qui rend le système beaucoup plus flexible mais aussi moins fiable et stable ;
- *cloud* : du point de vue système, le cloud est un ensemble d'ordinateurs (ressources) stockés sur de vastes grilles de serveurs ou de data centres. Les ressources du cloud sont mutualisées à travers une virtualisation qui favorise la montée en charge, la haute disponibilité et un plan de reprise à mondre coût.
- les réseaux de capteurs (*Sensor Network*) : c'est une collection de micro-ressources informatiques (micro-capteur ou système embarqué) reliées le plus souvent par un réseau sans fil en vue de collecter, d'échanger et de transmettre des données (en général environnementales) vers un ou plusieurs points de collecte, d'une manière autonome.

2.3 Les propriétés requises des systèmes distribués

Un système réparti doit assurer plusieurs propriétés pour être considéré comme performant. Nous ne citerons dans cette section que celles que nous trouvons les plus connexes à notre contexte d'études : la transparence, le passage à l'échelle, la disponibilité et l'autonomie.

2.3.1 Transparence

La transparence permet de cacher aux utilisateurs les détails techniques et organisationnels d'un système distribué ou complexe. L'objectif est de pouvoir faire bénéficier aux applications une multitude de services sans avoir besoin de connaître exactement la localisation ou les détails techniques des ressources qui les fournissent. Ceci rend plus simple, le développement des applications mais aussi leur maintenance évolutive ou corrective. Selon la norme (ISO, 1995) la transparence a plusieurs niveaux :

- **accès** : cacher l'organisation logique des ressources et les moyens d'accès à une ressource ;
- **localisation** : l'emplacement d'une ressource du système n'a pas à être connu ;
- **migration** : une ressource peut changer d'emplacement sans que cela ne soit aperçu ;
- **re-localisation** : cacher le fait qu'une ressource peut changer d'emplacement au moment où elle est utilisée ;
- **réplication** : les ressources sont dupliquées mais les utilisateurs n'ont aucune connaissance de cela ;
- **panne** : si un noeud est en panne, l'utilisateur ne doit pas s'en rendre compte et encore moins de sa reprise après panne ;
- **concurrence** : rendre invisible le fait qu'une ressource peut être partagée ou sollicitée simultanément par plusieurs utilisateurs.

Le parcours de cette liste montre qu'il n'est pas évident d'assurer une transparence totale. En effet, masquer complètement les pannes des ressources est quasi impossible aussi bien d'un point de vue théorique que pratique. Ceci est d'autant plus vrai qu'il n'est pas trivial de dissocier une machine lente ou surchargée de celle qui est en panne ou dans un sous-réseau inaccessible. En conclusion, le choix d'un niveau de transparence moyen, ne prenant en compte que les spécificités des applications Web 2.0 que nous ciblons dans cette thèse, est nécessaire pour amoindrir les coûts de mise en place du système et de sa complexité.

2.3.2 Passage à l'échelle

Le concept de passage à l'échelle désigne la capacité d'un système à continuer à délivrer avec un temps de réponse constant un service même si le nombre de clients ou de données augmente de manière importante. Le passage à l'échelle peut être mesuré avec au moins trois dimensions : *i*) le nombre d'utilisateurs et/ou de processus (passage à l'échelle en taille) ; *ii*) la distance maximale physique qui sépare les noeuds ou ressources du système (passage à l'échelle géographique) ; *iii*) le nombre de domaines administratifs (passage à l'échelle administrative). Le dernier type de passage à l'échelle est d'une importance capitale dans les grilles informatiques car il influe sur le degré d'hétérogénéité et donc sur la complexité du système (voir section 2.4.2).

Pour assurer le passage à l'échelle, une solution coûteuse consiste à ajouter de nouveaux serveurs puissants (plusieurs dizaines de CPU) pour garder le même niveau de performance en présence de fortes charges. Ce type de passage à l'échelle est plus connu sous le nom de *Scale Up*. Le problème avec cette solution est que si le système est sous-chargé les ressources consomment de l'énergie inutilement et ne servent à rien. D'autres solutions moins chères consistent à utiliser plusieurs machines moins puissantes (d'un à deux CPU par machine) pour faire face aux pics des charges, on parle alors de *Scale Out*. Trois techniques peuvent être utilisées pour favoriser le passage à l'échelle à faible coût. La première technique consiste à ne pas attendre la fin d'une tâche pour commencer une autre si elles sont indépendantes. Cela permet de cacher la latence du réseau ou la lenteur d'une ressource. Ce modèle de fonctionnement est appelé modèle asynchrone. La deuxième technique consiste à partitionner les données et les stocker sur plusieurs serveurs. Ceci permet de distribuer la charge applicative et de réduire le temps de traitement global des tâches. Il est aussi possible d'effectuer certains traitements aux niveaux des clients (Java Applets). La troisième technique est l'utilisation de la réplication et/ou des mécanismes de cache. L'accès à des informations stockées dans un cache réduit les accès distants et donne de bonnes performances aux applications de type web. La réplication, quant à elle, permet une distribution des traitements sur plusieurs sites permettant ainsi une amélioration du débit du traitement.

Cependant, l'utilisation de ces techniques présente quelques inconvénients. En effet, le partitionnement et la distribution d'un traitement nécessitent des mécanismes de contrôle plus complexes pour intégrer les résultats et assurer leur cohérence. En plus, garder plusieurs copies d'une même donnée (caches ou répliques) peut entraîner des incohérences à chaque fois que l'on met une copie à jour. Pour éviter ce problème, il faut penser à faire des synchronisations, ce qui est souvent contradictoire avec la première technique de passage à l'échelle à savoir faire de l'asynchronisme. En conclusion, les besoins de passage à l'échelle et de cohérence sont en opposition, d'où la nécessité de trouver des compromis en fonction des besoins des applications cibles.

2.3.3 Disponibilité

Un système est dit disponible s'il est en mesure de délivrer correctement le ou les services de manière conforme à sa spécification. Pour rendre un système disponible, il faut donc le rendre capable de faire face à tout obstacle qui peut compromettre son bon fonctionnement. En effet, l'indisponibilité d'un système peut être causée par plusieurs sources parmi lesquelles nous pouvons citer :

- les pannes qui sont des conditions ou évènements accidentels empêchant le système, ou un des ses composants, de fonctionner de manière conforme à sa spécification ;
- les surcharges qui sont des sollicitations excessives d'une ressource du système entraînant sa congestion et la dégradation des performances du système ;
- les attaques de sécurité qui sont des tentatives délibérées pour perturber le fonctionnement du système, engendrant des pertes de données et de cohérences ou l'arrêt du système.

Pour faire face aux pannes, deux solutions sont généralement utilisées.

La première consiste à détecter la panne et à la résoudre, et ce dans un délai très court. La détection des pannes nécessite des mécanismes de surveillance qui s'appuient en général sur des *timeouts* ou des envois de messages périodiques entre ressources surveillées et ressources sur-

veillantes. Cette détection, outre la surcharge qu'elle induit sur le système, ne donne pas toujours des résultats fiables. En effet, avec l'utilisation des *timeouts*, à chaque fois qu'un *timeout* est expiré, la ressource sollicitée, ne pouvant être contactée ou n'arrivant pas à envoyer une réponse, est en général suspectée d'être en panne. Par conséquent, une simple variation de la latence du réseau ou de la surcharge d'une ressource peut entraîner l'envoi et/ou la réception d'une réponse en dehors du *timeout* et donc conduire à des fausses suspicions. Par ailleurs, une fois la panne détectée, il faut des mécanismes de résolution efficace pour arriver à la cacher aux clients. Cette tâche est loin d'être triviale car il existe plusieurs types de pannes et chacune d'entre elle requiert un traitement spécifique (voir chapitre suivant).

La deuxième solution consiste à masquer les pannes en introduisant de la réplication. Ainsi, quand une ressource est en panne, le traitement qu'elle effectuait est déplacé sur une autre ressource disponible. La réplication peut être aussi utilisée pour faire face à la seconde cause d'indisponibilité d'un système (surcharge du système). Pour réduire la surcharge d'une ressource, les tâches sont traitées parallèlement sur plusieurs répliques ou sur les différentes répliques disponibles à tour de rôle (tourniquet). Une autre technique qui permet de réduire la surcharge d'une ressource consiste à distribuer les services (ou les données) sur plusieurs sites et donc de les solliciter de manière parallèle. Le partitionnement des services ou des données permet d'isoler les pannes et donc de les contrôler plus simplement.

Enfin, la gestion de la dernière source d'indisponibilité nécessite des politiques de sécurité sur l'accès et l'utilisation des ressources. Ces politiques ont pour objet la mise en œuvre de mécanismes permettant de garantir les deux propriétés suivantes : la confidentialité et l'intégrité des ressources ou informations sensibles. La confidentialité permet de protéger les accès en lecture aux informations, alors que l'intégrité permet de protéger les accès en écriture. S'il existe une seule politique de sécurité pour l'ensemble des ressources, la sécurité est dite centralisée, dans le cas contraire, elle est dite distribuée et permet à chaque partie du système d'avoir sa propre politique. Il est à noter qu'une politique de sécurité distribuée permet plus d'hétérogénéité et s'adapte à des systèmes comme les grilles informatiques mais nécessite des mécanismes plus complexes. Nous mentionnons aussi que bien que la sécurité est capitale pour la disponibilité, nous ne l'aborderons pas dans cette thèse pour ne pas trop s'éloigner de nos objectifs.

Nous venons de voir que quelque soit la manière dont la gestion de la disponibilité est assurée, des modules supplémentaires sont requis (gestion de réplication, détection et résolution des pannes, politique de sécurité, etc.). Ceci entraîne d'une part, une surcharge du système (nombre de messages très important, collection de processus en arrière-plan) et d'autre part, une complexité du système. Une solution idéale serait de minimiser la complexité mais aussi la surcharge introduite pour assurer la disponibilité. Ainsi, les modules supplémentaires intégrés doivent être très légers (requièrent peu de ressources pour leur fonctionnement) et les techniques de détection/résolution des pannes ne doivent pas être réalisées sur la base d'une communication qui génère plusieurs messages (ex : *broadcast*).

2.3.4 Autonomie

Un système ou un composant est dit autonome si son fonctionnement ou son intégration dans un système existant ne nécessite aucune modification des composants du système hôte. L'autono-

mie des composants d'un système favorise l'adaptabilité, l'extensibilité et la réutilisation des ressources de ce système. Par exemple, une ressource autonome peut être remplacée avec une autre ressource plus riche en termes de fonctionnalités, ce qui étend les services du système. Le changement du pilote d'accès à une base de données ODBC par un pilote JDBC illustre bien cette notion d'autonomie et son impact dans le fonctionnement d'un système, puisqu'aucune modification ne sera effectuée au niveau du SGBD.

L'une des motivations de maintenir l'autonomie est que les applications existantes sont difficiles à remplacer car d'une part, elles sont le fruit d'une expertise développée pendant plusieurs années et d'autre part, leur code n'est pas souvent accessible, *i.e.* les SGBD tels que Oracle, SQL Server, Sybase, etc. Pourtant, il est indispensable de s'appuyer sur les applications existantes car les clients ont leurs préférences et ne sont pas nécessairement prêts à confier leur traitement à une nouvelle application ou de stocker leurs données sur un SGBD nouveau qui ne présente pas les mêmes garanties qu'un système connu, fiable et maintenu. Une solution pour garder l'autonomie d'une application ou d'un SGBD est d'intégrer toute nouvelle fonctionnalité supplémentaire et spécifique à une application sous forme d'intergiciel.

Cependant, l'implémentation de l'intergiciel introduit de nouvelles surcharges en ajoutant une couche de plus à l'accès aux données. A cela, s'ajoute que l'intergiciel devient indispensable au fonctionnement du système et peut devenir très rapidement source de congestion s'il est partagé par plusieurs applications. Pour minimiser ces impacts négatifs, l'intergiciel doit être conçu de tel sorte qu'il soit toujours disponible et non surchargé. La complexité de son implémentation doit être contrôlée afin de minimiser le coût de sa traversée. Pour ce faire, il faut éviter de concevoir un intergiciel très générique à destination de plusieurs applications au profit d'un intergiciel ad-hoc.

2.4 Etude de quelques systèmes distribués

Dans cette section, nous étudions trois catégories de systèmes distribués à savoir les systèmes pair-à-pair (P2P), les grilles informatiques et le cloud. Le choix d'étudier ces trois catégories est fortement tributaire de leur caractère très hétérogène et leur besoin de passage à l'échelle, que nous avons privilégié dans cette thèse. L'étude met l'accent sur les architectures et les mécanismes permettant d'assurer la disponibilité, le passage à l'échelle et la transparence et l'autonomie.

2.4.1 Systèmes P2P

Comme nous l'avons mentionné ci-avant, le terme P2P fait référence à une classe de systèmes distribués qui utilisent des ressources distribuées pour réaliser une tâche particulière de manière décentralisée. Les ressources sont composées d'entités de calcul (ordinateur ou PDA), de stockage de données, d'un réseau de communication, etc. La tâche à exécuter peut être du calcul distribué, du partage de données (ou de contenu), de la communication et collaboration, d'une plateforme de services, etc. La décentralisation, quant à elle, peut s'appliquer soit aux algorithmes, soit aux données, soit aux métadonnées, soit à plusieurs d'entre eux.

L'une des particularités des systèmes P2P est que tous les nœuds (pairs) sont en général symétriques, c'est à dire qu'ils jouent à la fois le rôle de client et de serveur. En particulier, les

systèmes de partage de fichiers permettent de rendre les objets d'autant plus disponibles qu'ils sont populaires, en les répliquant sur un grand nombre de nœuds. Cela permet alors de diminuer la charge (en nombre de requêtes) imposée aux nœuds partageant les fichiers populaires, ce qui facilite l'augmentation du nombre de clients et donc le passage à l'échelle en taille des données.

Les pairs sont organisés autour d'une architecture qui peut être centralisée ou non. Dans l'architecture centralisée, un noeud joue le rôle de coordinateur central (serveur, ou index central) et gère soit les partages, soit la recherche, soit l'insertion d'informations et de nouveaux nœuds. Cependant l'échange d'informations entre noeuds se passe directement d'un noeud à l'autre. D'aucun considèrent que de telles architectures ne sont pas pair-à-pair, car un serveur central intervient dans le processus. Par contre d'autres arguent que ce sont bien des systèmes pair-à-pair car les fichiers transférés ne passent pas par le serveur central. Néanmoins, c'est une solution fragile puisque le serveur central est indispensable au réseau. Ainsi, s'il est en panne ou non accessible, tout le réseau s'effondre. En plus, le système n'est pas transparent puisque les nœuds ont besoin de savoir à tout moment l'emplacement de l'index et sont sensibles à tout changement de celui-ci. Un exemple de solution P2P centralisée est Napster [Nap], qui utilise un serveur pour stocker un index ou pour initialiser le réseau.

Pour faire face à ces insuffisances, une architecture distribuée s'impose, puisqu'un nœud pourrait solliciter plusieurs serveurs en même temps. Le système est ainsi plus robuste mais la recherche d'informations est plus difficile. Elle peut s'effectuer dans des systèmes décentralisés non-structurés, comme Gnutella [Gnu]. Dans ce système, la recherche se fait par propagation de la requête aux voisins jusqu'à trouver les résultats, ce qui nécessite dès lors un nombre de messages élevé, proportionnel au nombre d'utilisateurs du réseau (et exponentiel suivant la profondeur de recherche). Dans les systèmes décentralisés structurés, une organisation de connexion et de répartition des objets partagés est maintenue entre les nœuds. Souvent cette organisation est basée sur les tables de hachage distribuées, permettant de réaliser des recherches en un nombre de messages croissant de façon logarithmique avec le nombre d'utilisateurs du réseau, comme CAN [RFH+01], Chord [SMK+01], Kademlia [MM02] Pastry [RD01a], etc.

Une autre solution possible est l'utilisation de « super-pairs », noeuds du réseau choisis en fonction de leur puissance de calcul et de leur bande passante, réalisant des fonctions utiles au système comme l'indexation des informations et le rôle d'intermédiaire dans les requêtes. Cette solution que l'on retrouve dans Kazaa [KaZ], rend le système un peu moins tolérant aux pannes que les systèmes complètement décentralisés et englobe un peu l'idée de client-serveur entre pairs et super-pairs.

Comme tous les noeuds sont au même niveau avec les architectures complètement distribuées, celle-ci offrent plus de transparence dans l'organisation et la localisation des ressources que les architectures centralisées ou semi-centralisées (super-pairs).

Les systèmes P2P soulèvent plusieurs problèmes bien qu'ils facilitent le passage à l'échelle et la disponibilité des ressources avec un faible coût. Il faut noter en premier lieu que les noeuds du système sont totalement autonomes et donc qu'ils peuvent choisir de partager ou non leur CPU et leur capacité de stockage. Cette autonomie leur confère aussi le choix de rejoindre ou quitter le système à tout moment. Cela a pour effet de compromettre la capacité de calcul totale et réelle du système mais aussi la disponibilité des ressources (informations, capacité de stockage, etc.). En

second lieu, le système de communication utilisé est de faible bande passante (en général Internet), ce qui peut créer une surcharge du système et une latence plus élevée que dans les clusters. Enfin, l'absence d'infrastructures de contrôle sur les systèmes P2P rend ces derniers moins pratiques pour prendre en compte certains types d'applications qui exigent une grande qualité ou des services transactionnels. Néanmoins, les systèmes P2P sont devenus incontournables aujourd'hui dans le domaine du partage de fichiers, de la recherche d'information et de la collaboration.

2.4.2 Les grilles informatiques ou *grid*

Le terme anglais *grid* désigne un système distribué d'électricité. Initialement, le concept de grille partait du principe d'un tel système : les ressources d'un ordinateur (processeur, mémoire, espace disque) étaient mises à la disposition d'un utilisateur aussi facilement que l'on branche un appareil électrique à une prise électrique. Une grille informatique est une infrastructure virtuelle constituée d'un ensemble de ressources informatiques potentiellement partagées, distribuées, hétérogènes, délocalisées et autonomes. Une grille est en effet une infrastructure, c'est-à-dire des équipements techniques d'ordre matériel et logiciel. Cette infrastructure est qualifiée de virtuelle car les relations entre les entités qui la composent n'existent pas sur le plan matériel mais d'un point de vue logique. D'un point de vue architectural, la grille peut être définie comme un système distribué constitué de l'agrégation de ressources réparties sur plusieurs sites et mises à disposition par plusieurs organisations différentes [Jan06]. Un site est un lieu géographique regroupant plusieurs ressources informatiques administrées de manière autonome et uniforme. Il peut être composé d'un super-calculateur ou d'une grappe de machines (cluster).

Contrairement aux systèmes P2P, une grille garantit des qualités de service non triviales, c'est-à-dire qu'elle répond adéquatement à des exigences (accessibilité, disponibilité, fiabilité, ...) compte tenu de la puissance de calcul ou de stockage qu'elle peut fournir.

Il existe plusieurs projets de grilles qui on été mis en place aussi bien à des échelles nationales qu'internationales : la grille expérimentale française Grid'5000 [Proc], la grille de calcul scientifique nord américain TeraGrid [proe], la grille chinoise CNGrid [prob], la grille Asie-Pacifique ApGrid [proa], etc.

Les grilles informatiques sont caractérisées par une forte hétérogénéité et une grande dynamicité. En effet, les machines d'une grappe sont reliés par un réseau gigabits alors que les sites sont liés par un réseau WAN, dont la latence peut aller jusqu'à 100 millisecondes. De là, nous pouvons noter une différence importante de la latence entre deux machines d'un même site et celle entre deux machines de deux sites. En outre, chaque site est administré de manière autonome et par conséquent les politiques de sécurité varient d'un site à l'autre. Un autre exemple d'hétérogénéité relève de la composition interne d'un site. Chaque site est libre de choisir le type de processeur de ses machines (Intel, AMD, IBM, ...), la capacité de stockage et le réseau d'interconnexion entre machines (Gigabit Ethernet, Infiniband, ...). Enfin l'infrastructure d'une grille est composée d'un nombre important de sites et de machines qui sont susceptibles de tomber en panne à tout moment. A cela s'ajoute le fait que de nouveaux sites peuvent être ajoutés ou retirés de la grille sans trop impacter le fonctionnement de la grille. De par leur hétérogénéité, leur gestion décentralisée et leur taille, les grilles sont des infrastructures très complexes à mettre en oeuvre. La transparence n'est assurée qu'à moitié parce qu' il faut avoir une information précise des ressources dans un site

pour pouvoir distribuer les tâches convenablement. Cependant à l'intérieur d'un site, la machine qui effectue concrètement la tâche n'est pas connue en général.

2.4.3 Le cloud

Le *cloud* est un concept plus récent dont une définition unanime tarde à voir le jour [VRMCL09, Gee09, BYV08, WLG+08]. Cette divergence découle des principes considérés par les chercheurs pour définir le *cloud*. En effet , certains auteurs mettent l'accent sur le passage à l'échelle et la mutualisation de l'usage des ressources [Gee09] alors que d'autres privilégient le concept de virtualisation [BYV08] ou le *business model* (collaboration et *pay-as-you-go*) [WLG+08]. Cependant, il est unanimement reconnu que le *cloud* permet l'utilisation de la mémoire et des capacités de calcul des ordinateurs et des serveurs répartis dans le monde entier et liés par un réseau, tel Internet. Les ressources sont en général logées dans des data centres qui sont géographiquement distribués dans l'optique de garantir le passage à l'échelle et la disponibilité. Avec le *cloud*, les utilisateurs ne sont plus propriétaires de leurs serveurs informatiques mais peuvent ainsi accéder de manière évolutive à de nombreux services en ligne sans avoir à gérer l'infrastructure sous-jacente, souvent complexe. C'est pourquoi, on peut considérer le cloud comme une extension des ASP. Les applications et les données ne se trouvent plus sur l'ordinateur local, mais dans un nuage (*cloud*) composé d'un certain nombre de serveurs distants interconnectés au moyen d'une excellente bande passante indispensable à la fluidité du système. L'accès au service se fait par une application standard facilement disponible, la plupart du temps un navigateur Web. Les service offerts par le *cloud* sont nombreux parmi lesquels nous avons :

- *Infrastructure as a service (IaaS)* : c'est un service qui donne à l'utilisateur un ensemble de ressources de traitement et de stockage dont il a besoin à un instant précis pour effectuer ses tâches ;
- *Platform as a Service (PaaS)* : ce service, en dehors de fournir une infrastructure virtuelle, assure aussi la plateforme logicielle pour que les applications de l'utilisateur puissent tourner ;
- *Software as a Service (SaaS)* : c'est une alternative de toute application locale. Un exemple de ce cas est l'utilisation en ligne de la suite bureautique de Microsoft Office.

Pour assurer ces services qui varient d'un utilisateur à un autre, l'architecture des *clouds* est composée à son niveau le plus basique d'une couche logique composée d'un ensemble de machines virtuelles et d'une couche physique composée de data centres[BYV08]. Ainsi, plusieurs machines virtuelles peuvent être démarrées dynamiquement sur une seule machine physique pour satisfaire les besoins de plusieurs services. Les data centres qui regroupent les machines physiques sont en général répartis sur des sites géographiquement distants afin d'assurer une haute disponibilité. En plus cette répartition permet aussi d'assurer un niveau de performances élevé en branchant un utilisateur sur le site le plus proche de son emplacement afin de réduire les délais de communication.

La mutualisation du matériel permet d'optimiser les coûts par rapport aux systèmes conventionnels et de développer des applications partagées sans avoir besoin de posséder ses propres machines dédiées au calcul. Comme pour la virtualisation, l'informatique dans le nuage est plus économique grâce à son évolutivité. En effet, le coût est fonction de la durée de l'utilisation du service rendu et ne nécessite aucun investissement préalable (homme ou machine). Notons également

que l'élasticité du nuage permet de fournir des services évolutifs et donc de supporter les montées de charges. Par exemple, Salesforce.com, pionnier dans le domaine de l'informatique dans le nuage gère les données de 54 000 entreprises, et leurs 1,5 millions d'employés, avec seulement 1 000 serveurs (mars 2009). De plus, les services sont extrêmement fiables car basés sur des infrastructures performantes possédant des politiques efficaces de tolérance aux pannes (notamment des répliques). Grâce à ses avantages, on assiste aujourd'hui à une multiplication rapide d'entreprises qui proposent des solutions *cloud* parmi lesquelles figurent : Amazon, IBM, Microsoft, Google, etc.

Cependant, le problème fondamental reste d'une part la sécurisation de l'accès à l'application entre le client et le serveur distant. On peut aussi ajouter le problème de sécurité générale du réseau de l'entreprise : sans le *cloud computing*, une entreprise peut mettre une partie de son réseau en local et sans aucune connexion (directe ou indirecte) à internet, pour des raisons de haute confidentialité par exemple ; dans le cas du *cloud computing*, elle devra connecter ces postes à internet (directement ou pas) et ainsi les exposer à un risque d'attaque ou a des violations de confidentialité.

2.4.4 Exemple d'utilisation des systèmes distribués à large échelle

Les systèmes à large échelle tels que les P2P ou les grilles peuvent être utilisés pour concevoir et réaliser différentes classes d'applications.

- Communication et collaboration. Cette catégorie inclut les systèmes qui fournissent des infrastructures pour faciliter la communication et la collaboration en temps réel (ex. Skype [Sky], MSN [MSN], Yahoo [Yah], Groove [Gro]) ;
- Calcul distribué. Ce type d'application permet le partage de ressources CPU en divisant et répartissant le travail sur plusieurs machines. Seti@home [Pau02], le projet de recherche d'une intelligence extraterrestre de l'université de Californie et de Berkley d'une part et Folding@Home [prof], le projet de calcul réparti étudiant les protéines de l'université de Standford d'autre part sont des illustrations de ce cas d'application.
- Support de service internet. Un nombre considérable d'applications basées sur les systèmes P2P ont été développées pour supporter les services sur internet. Parmi les exemples de ce type d'applications, nous pouvons citer les systèmes de multicast P2P publish/subscribe [VBB$^+$03, CDKR02] ou les applications de sécurité et d'antivirus [VMR02, JWZ03, VATS04].
- Système de Base de données. Les sytèmes distribués sont également utilisés pour concevoir des bases de données distribuées. Par exemple, Local Relational Model [SGMB01], PIER [HCH$^+$05] et Piazza [HIM$^+$04] utilisent des architectures P2P pour stocker et exploiter des données.
- Distribution et gestion de contenu. Elle constitue la charnière centrale des utilisations que l'on peut faire avec les systèmes distribués. Elle couvre une variété d'applications allant du simple partage de fichiers direct aux systèmes sophistiqués qui fournissent des supports de stockage distribué, une technique d'indexation et de localisation efficace et enfin des procédures de mises à jour. Google File System (GFS) [GGL03], Hadoop Distributed File System (HDFS) [Prog] sont des exemples de support de stockage distribués avec un mécanisme d'indexation efficace. Un autre avantage majeur de cette classe d'applications est qu'elle donne

aux utilisateurs la possibilité de publier, de stocker et de gérer leur contenu de manière assez efficace. Les applications Web2.0 tels que eBay [Sho07] ou Facebook [Fac] reposent sur ce concept et en constituent des exemples très populaires.

Dans cette thèse, nos objectifs sont orientés vers les applications Web 2.0. En effet, nous voulons gérer une base de données distribuée à grande échelle dans l'optique de donner la possibilité aux utilisateurs d'un accès rapide et cohérent. La conception d'un tel système nécessite une étude d'un des types d'applications ciblées à savoir eBay pour mieux comprendre les exigences à satisfaire.

Exemple d'applications Web 2.0 : eBay

eBay [Sho07] est devenu le leader mondial du commerce en ligne avec plus de 276 millions de membres inscrits. La société fondée en 1995, est devenue une place de marché mondiale où une communauté de passionnés, composée aussi bien de particuliers que de professionnels, peut acheter et vendre en ligne des biens et des services aux enchères ou à prix fixe. Chaque jour, plus de 113 millions d'articles répartis dans plus de 55.000 catégories sont à vendre sur eBay et correspondent à 2 Pétaoctects de données. D'un point de vue technique, la plate-forme d'eBay est très dynamique avec l'ajout de plus de 100.000 lignes de code tous les deux semaines pour satisfaire les besoins de 39 pays, dans 7 langues et ce 24h/24 et 7j/7. Cette charge applicative correspond à plus de 48 milliards d'instructions SQL par jour. Les utilisateurs sont soit des vendeurs qui offrent des biens soit des acheteurs qui achètent les biens. Un bien est dans une catégorie donnée et appartient à un seul vendeur. Pour faire face à cette intense charge applicative, l'architecture de eBay est conçue pour prendre en compte cinq critères de performances à savoir la disponibilité, le passage à l'échelle, le coût, la maintenance et la latence des réponses. La prise en compte de la disponibilité et du passage à l'échelle est faite en adoptant quatre stratégies :

- Partitionner tout en divisant tout problème (données, traitement, ...) en taille maîtrisable : permet de passer à l'échelle, assure le disponibilité par isolation des pannes, moins coûteux à maintenir et requiert peu de hardware.
- Introduire de l'asynchronisme en mettant en œuvre des processus (ou composants) asynchrones et en regroupant certains traitements (*batch*) : favorise le passage à l'échelle indépendante de chaque composant, une meilleure disponibilité par la détection et la reprise ciblées des composants en panne et diminue la latence des réponses par parallélisation des traitements.
- Automatiser tout en intégrant des systèmes auto-adaptatifs : le passage à l'échelle sans une intervention manuelle, la latence comme la disponibilité sont assurées par l'auto-configuration quand l'environnement change et le coût est faible du fait de l'absence de l'intervention humaine.
- Garder à l'esprit que tout tombe en panne et donc être prêt à détecter toute panne et la résoudre dans un délai court : assure la disponibilité du système

Pour stocker les données, eBay utilise un partitionnement horizontal et des milliers de SGBDs MySQL comme Facebook. Pour assurer la disponibilité, eBay a développé des techniques de cache autour du SGBD MySQL, ce qui lui permet de pouvoir exécuter un nombre très significatif d'opérations de lecture/écriture à un coût relativement faible. En plus, il utilise un mécanisme de réplication asynchrone qui est très adapté au passage à l'échelle.

L'étude de l'exemple d'eBay montre les exigences des applications Web 2.0 en termes de débit transactionnel, de haute disponibilité, de latence faible, etc. Pour prendre en compte tous ces critères, il faut de nouvelles approches basées sur une bonne conception et implémentation du système distribué qui héberge les services mais aussi des mécanismes efficaces de la gestion des données utilisées par ces applications. Dans la prochaine section, nous décrivons les spécifications d'une bonne implémentation des systèmes distribués et détaillerons les spécifications adéquates de la gestion des données dans le prochain chapitre.

2.5 Implémentation d'un système distribué avec un middleware

La plupart des systèmes distribués sont implémentés avec un middleware comme le montre la figure 2.1.

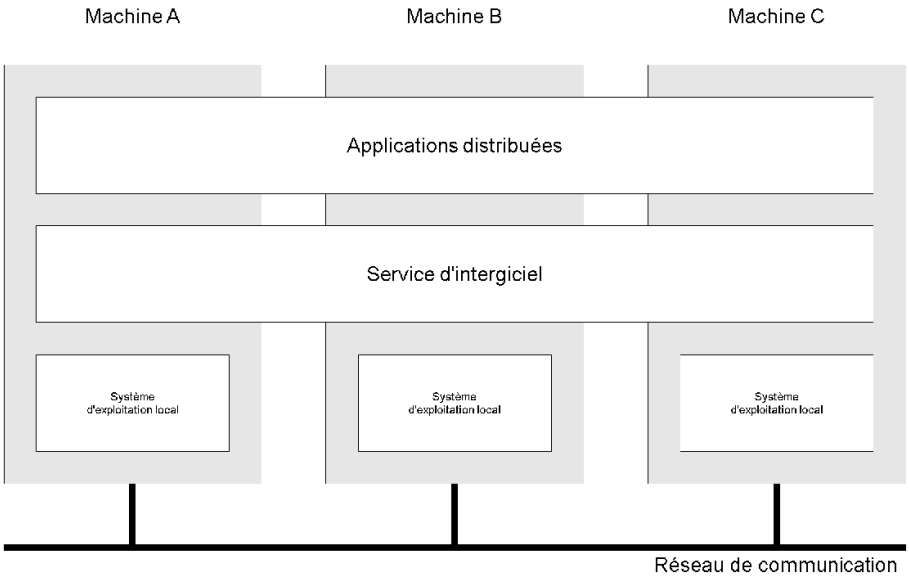

FIGURE 2.1 – Architecture d'un système distribué structuré avec un integiciel

Le mot middleware (intergiciel ou logiciel médiateur en français) désigne un ensemble de logiciels ou de technologies informatiques qui servent d'intermédiaire entre les applications. Il peut être défini aussi comme un outil de communication entre des clients et des serveurs. Ainsi, il fournit un moyen aux clients de trouver leurs serveurs et aux serveurs de trouver leurs clients et en général de trouver n'importe quel objet atteignable. L'intérêt que présente un middleware est multiple et peut être résumé en quatre points [Kra09] :

- cacher la répartition, c'est-à-dire le fait qu'une application est constituée de parties interconnectées s'exécutant à des emplacements géographiquement répartis ;
- cacher l'hétérogénéité des composants matériels, des systèmes d'exploitation et des protocoles de communication utilisés par les différentes parties d'une application ;

– fournir des interfaces uniformes, normalisées, et de haut niveau aux équipes de développement et d'intégration, pour faciliter la construction, la réutilisation, le portage et l'interopérabilité des applications ;
– fournir un ensemble de services communs réalisant des fonctions d'intérêt général, pour éviter la duplication des efforts et faciliter la coopération entre applications.

Un middleware peut être générique ou spécifique à un type d'applications. Il faut remarquer que les gains introduits par un middleware ne vont pas sans inconvénients. En effet, un inconvénient potentiel est la perte de performances liée à la traversée de couches supplémentaires de logiciel. L'utilisation de techniques intergicielle implique par ailleurs de prévoir la formation des équipes de développement.

2.5.1 Catégories de Middleware

Les middlewares peuvent être classés suivant plusieurs critères, incluant les propriétés de l'infrastructure de communication, leur propre architecture et la structure globale des applications.

Propriétés de la communication

L'infrastructure de communication sur laquelle se repose un middleware peut être caractérisé par plusieurs propriétés qui permettent une première classification.

Topologie statique ou dynamique. Avec un système de communication statique, les entités communicant sont logées dans des endroits fixes et la configuration du système ne change pas. Si toutefois, la configuration doit changer, elle est programmée en avance, peu fréquente et bien intégrée dans le fonctionnement du middleware. Par contre, un système de communication dynamique donne la possibilité aux entités communicantes de changer de localisation, de se connecter et/ou se déconnecter pendant le fonctionnement de l'application (téléphones mobiles, PDA, P2P, ...).

Comportement prévisible ou imprévisible. Dans certains systèmes de communication, des bornes peuvent être établies dans l'optique de maintenir les facteurs de performances des applications (par exemple la latence). Hélas, dans la plupart des cas pratiques, ces bornes ne sont pas connues car les facteurs de performances dépendent de la charge des composants du système mais aussi du débit du réseau de communication. Le système de communication est dit synchrone si le temps de transmission d'un message est borné. Si par contre, cette borne ne peut être établie, le système est dit asynchrone.

Il est possible de faire une combinaison de ces différentes propriétés pour obtenir :
– topologie statique, comportement prévisible ;
– topologie dynamique, comportement prévisible ;
– topologie statique, comportement imprévisible ;
– topologie dynamique, comportement imprévisible.

La dernière combinaison inclut les applications déployées sur les systèmes mobiles ou P2P avec lesquelles un noeud peut à tout moment joindre ou quitter le système. Cependant, le caractère imprévisible de cette classe de système, impose une surcharge supplémentaire au middleware afin qu'il puisse maintenir à un niveau acceptable les performances du système.

Structuration des composants des applications.

Les middlewares peuvent être classés aussi en fonction des types d'entités (composants) gérées ou en fonction de la structure des différents rôles que jouent les composants.

Type de composants. Un middleware peut avoir à sa charge plusieurs types d'entités qui diffèrent par leur définition, leur propriété et leur mode de communication. Suivant le type de l'entité, le rôle du middleware peut varier. Le premier type d'entité que l'on peut avoir est le message. Dans ce cas, le rôle du middleware est de fournir aux applications la capacité d'envoyer et de recevoir des messages. Ce type de middleware est plus connu sous le nom *Messaging-Oriented Middleware* (MOM) ou plus récemment de *Loosely Coupled Message Passing* (LCMP). Un autre type d'entité que l'on peut avoir est l'objet (de programmation orientée objet). Le middleware aura comme tâche de faire collaborer (communiquer) des objets qui se trouvent sur différentes plates-formes. Comme exemple de middleware de ce type, nous pouvons citer entre autres CORBA, J2EE, DCOM/DCOM+, ORB, etc. Un autre type d'entité manipulée peut être une base de données, ce qui a donné naissance au middleware de base de données (Database Middleware). Ce type de middleware donne le possibilité aux clients d'accéder à des données hétérogènes et ce, quel que soit le modèle de donnée ou le SGBD utilisé. Ces middlewares sont souvent conçus sous forme d'API comme ODBC, JDBC, etc.

Structure des composants. Les composants gérés par un middleware peuvent jouer différents rôles comme celui de client (demandeur de service), ou celui de serveur (fournisseur de service), celui d'annonceur (éditeur de service à diffuser), ou celui d'abonné (souscripteur à une service). Parfois, tous les composants peuvent être au même niveau et donc il est possible de trouver tous les rôles au sein d'une seule entité, comme par exemple dans les systèmes P2P.

Architecture des middlewares.

Les middlewares peuvent être aussi catégorisés en s'appuyant sur leur architecture. Il existe deux types d'architectures : l'une centralisée et l'autre distribuée.

Architecture centralisée. Avec l'architecture centralisée, l'ensemble des modules du middleware sont stockés sur un seul site (machine). Cette approche est beaucoup plus facile à mettre en oeuvre mais elle facilite aussi la maintenance et l'exploitation cohérente du système. Cependant, si le nombre de composants pris en compte devient important, cela peut induire à une source de contention et donc réduire les performances du système.

Architecture distribuée. L'approche distribuée répartit les tâches du middleware sur plusieurs sites, ce qui donne la possibilité d'accéder au middleware de manière parallèle. Ce type d'architecture est beaucoup plus tolérant aux pics de charges grâce à la répartition des demandes sur les différents composants répartis du middleware. Cependant, la maintenance et la gestion cohérente des entités du systèmes deviennent beaucoup plus fastidieuse. En fait, les composants du middleware sont obligés de travailler de manière collaborative pour éviter des incohérences. Ce qui nécessite l'envoi de messages ou le partage de structures de données et par conséquent, des surcharges supplémentaires. Une autre approche de la distribution d'un middleware est d'avoir plusieurs instances du middleware qui s'exécute sur plusieurs sites. Cette technique, outre la tolérance aux pics de charges qu'elle offre, assure aussi une tolérance aux pannes. En fait, la duplication du middle-

ware sur plusieurs sites permet de masquer la panne de l'une des instances du middleware ou du site qui l'héberge.

2.6 Modèle d'architecture pour la gestion des données à large échelle

Notre objectif dans cette thèse, est de proposer un modèle de traitement des transactions dans une base de données distribuée sur un système à large échelle (grille ou P2P). Pour atteindre cet objectif, il est nécessaire de concevoir une infrastructure qui tire profit autant des systèmes P2P que des grilles. Ce faisant, l'architecture proposée doit être auto-configurable et tolérante aux pannes pour prendre en compte l'instabilité, le caractère transitoire des composants (ou ressources) et les pannes des systèmes à grande échelle à l'image des systèmes P2P. Une telle infrastructure doit permettre l'interopérabilité entre les ressources du système de manière complètement décentralisée. Par ailleurs, l'architecture du système doit être basée sur une approche middleware qui ne doit pas être centralisée pour assurer un bon niveau de transparence et d'autonomie (surtout des données), une haute disponibilité et une utilisation efficace des ressources du système (minimiser les surcharges et contrôler l'accès aux données). Dans la quête de l'idéal, des modules de contrôle de la sécurité sont aussi à envisager mais nous les ignorons dans cette thèse qui tente d'apporter plutôt des réponses sur les problèmes liés aux pannes et aux surcharges.

Chapitre 3

Gestion des transactions dans les bases de données répliquées

La réplication dans les bases de données a été largement étudiée dans les trois dernières décennies. Elle consiste en la gestion des copies stockées sur différents sites qui utilisent leur propre SGBD autonome. Le défi majeur des bases de données répliquées est le maintien de la conformité des copies (cohérence mutuelle) lorsque les données sont mises à jour. Dans le contexte des bases de données les mises à jour sont souvent encapsulées dans des transactions. En d'autres termes, la base de données supporte les opérations regroupées sous forme de transactions plutôt que l'exécution indépendante des opérations les unes à la suite des autres. Par ailleurs, les applications Web 2.0 utilisent souvent les transactions pour manipuler les données (lire, modifier, insérer). C'est pourquoi la manière (quand et où) dont les transactions sont exécutées sur les différentes copies d'une base de données répliquées peut avoir un impact sensible sur la cohérence des données et les performances des applications. Par conséquent, le modèle de traitement des transactions doit être judicieusement choisi pour satisfaire les exigences des applications aussi bien en termes de cohérence que de performances. Les applications exigent en général comme performances un grand débit transactionnel, une latence faible, une disponibilité des données, un passage à l'échelle, une utilisation efficiente des ressources, etc.

Cependant, il est connu de tous que les besoins de performances et de cohérence sont en opposition car le coût de maintien de la cohérence est un frein à l'exécution des traitements de l'application. Par exemple, pour assurer une conformité des répliques à tout moment, il est nécessaire de mettre à jour toutes les répliques au sein de la même transaction. Ceci entraîne le ralentissement des traitements puisqu'il faut attendre que tous les sites, quelque soit leur endroit et leur état, valident la transaction localement pour que les données redeviennent disponibles. Cet exemple montre qu'il est indispensable de trouver un compromis entre la cohérence et les performances. Malheureusement, ce compromis n'est pas facile à trouver et dépend surtout des besoins de l'application considérée. De fait, une application de type Web 2.0 qui requiert une forte disponibilité et un passage à l'échelle exige nécessairement un relâchement de la cohérence. Par contre, une application de gestion de stock destinée à un petit magasin a besoin d'une cohérence plutôt forte. Dans cette thèse, nos travaux se situent dans le contexte des applications Web 2.0 et donc le compromis est penché vers le relâchement de la cohérence pour un meilleur passage à l'échelle.

L'objectif de ce chapitre est de donner quelques généralités sur les transactions et leur gestion dans une base de données répliquée. Pour commencer, nous donnons quelques définitions et notions sur les transactions et sur les bases de données répliquées. Ensuite, nous étudions quelques méthodes existantes pour gérer les transactions dans les bases de données répliquées. Plus précisément, nous étudions les solutions de gestion de transactions en privilégiant les quatre caractéristiques des systèmes distribués que nous avons mentionnées dans le premier chapitre à savoir le passage à l'échelle, la tolérance aux pannes (ou disponibilité), la transparence et l'autonomie des données. En d'autres mots, nous ne nous intéressons qu'aux approches proposées par certaines solutions pour garantir ces quatre propriétés. Ainsi, il faut noter qu'en ce qui concerne le passage à l'échelle, il existe plusieurs principes qui permettent de l'obtenir parmi lesquels nous avons choisi de détailler ceux utilisant la cohérence à terme, ou la réplication partielle, ou le maintien de plusieurs versions des copies (*Snapshot Isolation (SI)*). Quant à la gestion de la tolérance aux pannes, deux approches seront étudiées : les techniques de masquage des pannes avec la réplication active et les techniques basées sur la détection et la résolution des pannes. Enfin, pour aborder la transparence et l'autonomie, nous étudions les solutions de réplication basées sur des intergiciels, qui permettent de cacher la distribution des données (répartition et localisation des répliques) mais aussi l'indisponibilité de certaines ressources. Les solutions étudiées seront placées dans notre contexte afin de bien situer leurs limites mais aussi de bien comprendre les principes à mettre en œuvre pour mieux satisfaire les applications à large échelle.

3.1 Notions de transactions

Une transaction peut être considérée comme une unité de traitement cohérente et fiable. Une transaction prend un état d'une base de données, effectue une ou des actions sur elle et génère un autre état de celle-ci. Les actions effectuées sont des opérations de lecture ou d'écriture sur les données de la base. Par conséquent, une transaction peut être définie comme étant une séquence d'opérations de lecture et d'écriture sur une base de données, qui termine en étant soit validée soit abandonnée. Si la base de donnée est cohérente au début de la transaction, alors elle doit rester cohérente à la fin de l'exécution de la transaction bien que cette dernière peut s'exécuter de manière concurrente avec d'autres ou qu'une panne survienne lors de son exécution. Une base de données est dite cohérente si elle est correcte du point de vue de l'utilisateur, c'est à dire qu'elle maintient les invariants de la base ou les contraintes d'intégrité. La notion de cohérence recouvre plusieurs dimensions comme décrit dans [RC96]. Du point de vue des demandes d'accès, il s'agit de gérer l'exécution concurrente de plusieurs transactions sans que les mises à jour d'une transaction ne soient visibles avant sa validation, on parle de cohérence transactionnelle ou isolation. Du point de vue des données répliquées, il consiste à garantir que toutes les copies d'une même donnée soient identiques, on parle de cohérence mutuelle. La cohérence transactionnelle est assurée à travers quatre propriétés, résumées sous le vocable ACID :

- **Atomicité** : toutes les opérations de la transaction sont exécutées ou aucune ne l'est. C'est la loi du tout ou rien. L'atomicité peut être compromise par une panne de programme, du système ou du matériel et plus généralement par tout évènement susceptible d'interrompre la transaction.

- **Cohérence** : La cohérence signifie que la transaction doit être correcte du point de vue de l'utilisateur, c'est-à-dire maintenir les invariants de la base ou contraintes d'intégrité. Une transaction cohérente transforme une base de données cohérente en un base de données cohérente. En cas de non succès, l'état cohérent initial des données doit être restauré.
- **Isolation** : elle assure qu'une transaction voit toujours un état cohérent de la base de données. Pour ce faire, les modifications effectuées par une transaction ne peuvent être visibles aux transactions concurrentes qu'après leur validation. En outre, une transaction a une opération marquant son début (*begin transaction*) et une autre indiquant sa fin (*end transaction*). Si la transaction s'est bien déroulée, la transaction est terminée par une validation (*commit*). Dans le cas contraire, la transaction est annulée (*rollback*, *abort*).
- **Durabilité** : une fois que la transaction est validée, ses modifications sont persistantes et ne peuvent être défaites. En cas de panne de disque, la durabilité peut être compromise.

Les propriétés ACID sont très difficiles à maintenir car elles représentent un frein aux performances du système. Par exemple, l'atomicité cause un sérieux problème quand l'environnement est réparti sur un système à large échelle puisque toutes les sites participant à une transaction doivent valider localement avant que la transaction ne soit validée globalement. Autrement dit, le maintien de la cohérence exige que toutes les sites participants soient mises à jour au sein de la même transaction, ce qui ralentit la validation. Pour des besoins de performances, certaines propriétés ne sont pas parfois garanties dans l'optique d'améliorer les performances du système. En effet, les propriétés C et I peuvent être relâchées au profit d'un degré de concurrence plus élevé et donc d'un débit transactionnel plus important. Les transactions peuvent être classées suivant plusieurs critères [OV99]. Un des critères utilisé est la nature des différentes opérations qui composent la transaction. Ainsi, si une transaction contient au moins une opération qui effectue des modifications sur les données de la base, la transaction est dite transaction d'écriture ou de mise à jour. Si toutes les opérations ne font que des lectures sur les données de la base, la transaction est dite transaction de lecture.

Un autre classement peut être fait à partir de la durée de la transaction. Avec ce critère, une transaction peut être classée *on-line* ou *batch* [GR92, OV99]. Les transactions *on-line*, communément appelées transactions courtes, sont caractérisées par un temps de réponse relativement court (quelques secondes) et accèdent à une faible portion des données. Les applications qui utilisent ce modèle de transaction sont dénommées applications OLTP (*On-line Transactional Processing*) parmi lesquelles, nous avons les applications bancaires, de réservation de billets, de gestion de stocks, etc. Les transactions *batch*, appelées transactions longues, peuvent prendre plus de temps pour s'exécuter (minutes, heures, jours) et manipulent une très grande quantité des données. Les applications utilisant ce type de transactions sont les applications décisionnelles ou OLAP (*On-line Analytical Processing*), de conception, de workflow, de traitement d'image, etc.

Dans cette thèse nous nous focalisons sur les transactions de type OLTP, et nous allons étudier leur traitement dans le contexte des bases de données distribuées et répliquées dans la section 3.3. En plus, nous précisons que nous prenons en compte aussi bien les transactions de lecture que d'écriture mais en mettant plus l'accent sur les transactions d'écritures qui sont des sources d'incohérences.

3.2 Bases de données réparties et répliquées

Dans cette partie, nous donnons quelques généralités afférentes aux bases de données réparties (distribuées) et aux mécanismes de base de la réplication.

3.2.1 Objectifs et principes des bases de données réparties

Une base de données répartie est une collection de sites connectés par un réseau de communication.Chaque site est une base de donnée centralisée qui stocke une portion de la base de données. Chaque donnée est stockée exactement sur un seul site [BHG87]. La gestion d'une base de données répartie est gérée de manière transparente par un SGBD réparti. Les transactions peuvent être envoyées sur chaque site puis traduites en transactions locales avant d'être routées sur les sites appropriés (stockant une portion des données manipulées). Les résultats sont intégrés puis renvoyés aux applications clientes. Pour améliorer les performances, les données peuvent être répliquées sur plusieurs sites. La principale motivation de la réplication des données est l'augmentation de la disponibilité. En stockant les données critiques sur plusieurs sites, la base de données distribuée peut fonctionner même si certains sites tombent en panne. Un second avantage de la réplication consiste à l'amélioration des temps de réponses des requêtes grâce à la parallélisation des traitements et un accès plus facile et rapide des données. Cependant, l'introduction de la réplication introduit un nouveau problème car si une réplique est mise à jour à travers une transaction, toutes les autres répliques devraient l'être pour garantir la cohérence mutuelle, ce qui signifie que toutes les copies sont identiques. Ce faisant, il est évident que plus le nombre de répliques est grand, plus le coût du maintien de la cohérence est important. Par conséquent, il doit exister un compromis entre gestion de la cohérence et les performances (passage à l'échelle, latence, ...). Ce compromis dépend surtout des types d'applications conduisant à plusieurs mécanismes de réplication que nous allons décrire très brièvement dans la prochaine section.

3.2.2 Mécanismes de réplication

La réplication a fait l'objet de plusieurs études dans le contexte des systèmes distribués et aussi dans les bases de données répliquées. La motivation principale est la disponibilité pour les systèmes distribués et la performance (parallélisme) pour les bases de données. Dans [Gan06], l'auteur présente la gestion de la réplication suivant plusieurs dimensions. Pour des soucis de présentation, nous regroupons les dimensions décrites dans [Gan06] en quatre concepts à savoir la distribution ou placement des données, la configuration (rôle) des répliques, la stratégie de la propagation des mises à jour et enfin la stratégie de maintien de la cohérence.

– **Placement des données.** Les données peuvent être répliquées partiellement ou totalement. La réplication totale stocke entièrement la base de données sur chaque site. La réplication partielle nécessite une partition des données en fragments. Chaque fragment est stocké par la suite sur plusieurs noeuds. L'avantage de la réplication partielle est qu'elle permet de réduire de manière significative les accès concurrents aux données. En outre, le coût de la mise à jour des copies est moins important que dans le cas de la réplication totale compte tenu de la

faible portion des données sollicitée par les opérations de mises à jour.

– **Configuration des répliques.** Les mises à jour peuvent être effectuées sur une seule réplique (appelée maître) avant d'être propagées vers les autres (esclaves). Une telle configuration est appelée mono-maître (*primary copy*) car les autres répliques ne sont utilisées que pour les requêtes de lecture seule, ce qui améliore les performances des opérations de lecture seule. L'avantage d'une telle approche est qu'elle facilite la gestion de la cohérence globale du système (cohérence mutuelle) puisque toutes les mises à jour sont effectuées sur une seule copie. Cependant, cet avantage est contradictoire avec le passage à l'échelle et avec la disponibilité qui nécessitent que plusieurs copies soient utilisées en même temps pour faire face à une charge applicative d'écritures très importante et variable. Une approche multi-maître (*update anywhere*), dans laquelle les mises à jour peuvent être exécutées sur n'importe quelle réplique, permet d'améliorer aussi bien les performances des transactions de lecture seule que d'écriture. L'inconvénient de cette approche est qu'elle requiert des mécanismes de contrôle de concurrence distribués plus complexes pour garantir la cohérence mutuelle.
La configuration mono-maître profite plus aux applications de type OLAP avec lesquelles les données de production (sujettes à des modifications) doivent être séparées des données d'analyse (milliers d'opérations de lecture seule). Par contre les applications OLTP tirent plus de bénéfices de l'approche multi-maître et particulièrement quand le nombre de mises à jour est très important et provient de diverses sources (utilisateurs). Il existe beaucoup de produits commerciaux qui offrent des solutions de réplication mono-maître ou multi-maître. Pour les architectures mono-maître, nous pouvons citer de manière non-exhaustive Microsoft SQL Server replication, Oracle Streams, Sybase replication Server, MySQL replication, IBM DB2 DataPropagator, GoldenGate TDM platform, et Veritas Volume Replicator. Alors que les exemples de solutions multi-maître incluent Continuent uni/Cluster, Xkoto Gridscale, MySql Cluster, DB2 Integrated Cluster.

– **Stratégies de rafraîchissement (propagation).** Elles définissent comment la propagation va être effectuée en précisant le contenu à propager, le modèle de communication utilisé, l'initiateur et le moment du déclenchement. Le rafraîchissement est fait grâce à une transaction appelée transaction de rafraîchissement dont le contenu peut être les données modifiées (*writesets* en anglais) ou le code de la transaction initiale [EGA08, Gan06]. La caractéristique principale d'une transaction de rafraîchissement est qu'elle est déjà exécutée sur au moins une réplique. La propagation de la transaction par la ré-exécution de celle-ci sur toutes les répliques, ne garantit pas toujours le même résultat sur chaque site si des instructions SQL contextuelles comme RANDOM ou LIMIT ou plus simplement SYSDATE sont utilisées dans la requête [EGA08]. La propagation des données modifiées requiert la récupération de celles-ci, ce qui est souvent très coûteux (*triggers*, *log sniffing*, comparaison de *snapshot*, etc.) et donc peut impacter les performances du système. Néanmoins, la propagation des données modifiées évite de rejouer une transaction qui a nécessité beaucoup de calcul avant de produire un résultat et ne dépend pas du contexte. La propagation peut être initiée par le noeud qui a exécuté une première fois les mises à jour, on parle de PUSH (pousser). Elle

peut être aussi initiée par les noeuds recevant les mises à jour, c'est l'approche PULL (tirer). Quant à la communication, elle peut se faire de point à point, ou par communication de groupe (*multicast*, *broadcast*), de manière épidémique, etc. Cependant, le modèle de communication est fortement tributaire du type de système. Par exemple l'utilisation du multicast dans un réseau P2P réduit le nombre de messages mais n'est pas toujours faisable à cause du caractère dynamique du système.
Enfin, le déclenchement peut se faire dès la réception ou l'exécution d'une nouvelle transaction (immédiat), périodiquement, quand une réplique est obsolète, quand un noeud est peu chargé, à la demande, etc. Comme nous allons le montrer dans le prochain chapitre, le moment du déclenchement joue un rôle capital dans les performances du système (cohérence mutuelle, surcharge du réseau, ...). Le déclenchement à la demande ou quand un noeud est moins chargé permet de réduire la surcharge du réseau en regroupant plusieurs mises à jour mais a l'inconvénient de laisser diverger les copies.

– **Maintien de la cohérence.** La cohérence peut être gérée de manière stricte (réplication synchrone) ou relâchée (réplication asynchrone) [GHOS96]. Avec la réplication synchrone, toutes les répliques sont mises à jour à l'intérieur de la transaction. Cette approche a l'avantage de garder toutes les copies cohérentes à chaque instant, mais nécessite que toutes les répliques soient disponibles et synchronisées au moment de l'exécution de la transaction (ROWA pour *Read-One/ Write-All*). Une amélioration de cette approche est de synchroniser uniquement les répliques disponibles au moment de l'exécution d'une transaction (ROWAA pour *Read-One/ Write-All-Available*). Pour des systèmes à large échelle, de telles approches entraîneraient des retards dans la validation des transactions puisque la communication n'est pas toujours stable. Par conséquent et comme il a été démontré dans [GHOS96], cette solution ne passe pas généralement à l'échelle.
La réplication asynchrone est plus souple car une transaction est d'abord validée sur une seule réplique avant d'être propagée sur les autres répliques dans une autre transaction. L'inconvénient de cette approche est que les copies peuvent diverger. Cette divergence a pour impact de retourner aux utilisateurs des résultats faux (par exemple l'achat d'une place de cinéma qui n'existe pas) ou de faire varier les invariants (les règles de gestion) du système (par exemple créditer un compte qui n'est pas encore créé).
Cependant, il est possible de laisser volontairement les copies diverger dans l'optique d'améliorer les performances du système. En effet, il est possible d'effectuer des transactions de lectures sur des copies pas nécessairement fraîches pour accélérer l'exécution des transactions de lecture [GNPV07, LG06, LGV04, RBSS02]. Néanmoins, l'obsolescence des copies doit être contrôlée en fonction des exigences des applications [GNPV07]. L'obsolescence peut être définie de différentes manières [LGV04]. Dans cette thèse, nous considérons une seule mesure à savoir le nombre de transactions de mise à jour manquantes. Précisément, l'obsolescence d'une réplique R_i sur un site s_j est égale au nombre de transactions de mise à jour modifiant R^i sur un site quelconque mais non encore propagée sur s_j. L'obsolescence tolérée d'une transaction de lecture est donc, pour chaque objet de la base lue par la transaction, le nombre de transactions qui ne sont pas encore exécutées sur le site traitant la

transaction. L'obsolescence tolérée reflète le niveau de fraîcheur requis par une transaction de lecture pour s'exécuter sur un site. Par exemple, si une transaction de lecture requiert des données parfaitement fraîches, alors l'obsolescence tolérée est zéro. Pour des raisons de cohérence, les transactions de mise à jour ainsi que celles de rafraîchissement doivent lire des données parfaitement fraîches.
Par ailleurs, on peut classer la réplication asynchrone en deux familles : réplication optimiste et réplication pessimiste. Avec la réplication pessimiste, les transactions sont ordonnées *a priori* avant d'être envoyées sur les répliques en respectant leurs contraintes conflictuelles. L'ordonnancement *a priori* ne permet pas un contrôle d'accès concurrent très fin, car deux transactions peuvent apparaître conflictuelles sans pour autant l'être réellement. La réplication asynchrone optimiste autorise l'exécution de plusieurs transactions simultanément sur plusieurs sites. Au moment de la validation globale (maintien de la cohérence mutuelle), on vérifie les conflits et les résoud. La résolution peut se faire par abandon de certaines transactions ou par ré-exécution des transactions. Si les données sont bien partitionnées, les conflits sont moins fréquents et donc l'utilisation de la réplication optimiste devient bénéfique car le débit transactionnel est fortement augmenté.

3.3 Gestion des transactions dans les bases de données répliquées

Les propriétés ACID d'une transaction doivent être garanties aussi bien dans le cadre d'un système centralisé que celui d'un système distribué. L'objectif des transactions est de pouvoir assurer la cohérence de la base, même en présence de mises à jour. Cependant dans le contexte des bases de données, nous pouvons identifier deux types de cohérences à prendre en compte. Tout d'abord une cohérence transactionnelle qui consiste à garantir la cohérence sémantique (validité des contraintes d'intégrité) d'une copie de la base après l'exécution d'une transaction. Puis, une cohérence mutuelle qui assure la conformité de toutes les copies de la base après l'exécution de la transaction. La gestion de la cohérence mutuelle définie quand et comment les modifications d'une transaction sont écrites sur l'ensemble des copies. Comme nous l'avons déjà souligné dans le chapitre précédent, si les modifications sont écrites sur l'ensemble des copies avant que la transaction ne soit validée, la réplication est dite synchrone, autrement, elle est dite asynchrone. Il est à noter que parfois les écritures d'une transaction ne sont pas appliquées sur toutes les répliques mais elles y sont envoyées dans le seul but d'assurer la cohérence globale (vérifier s'il n'y a pas de conflit compromettant la cohérence) avant de valider la transaction. Une telle approche bien qu'elle soit considérée comme synchrone est moins stricte et favorise des temps de réponses beaucoup plus bas. L'avantage et l'inconvénient d'une approche dépendent de la stabilité de l'environnement, de sa composition et surtout des attentes des applications. Plusieurs dizaines de travaux ont abordé le problème de la gestion des transactions dans les bases de données répliquées en privilégiant l'une ou l'autre approche en fonction des applications cibles.

Dans la suite de ce chapitre, nous allons décrire les solutions de gestion de transactions les plus connexes à nos travaux et ce, suivant les quatre caractéristiques des systèmes distribués que nous

avons mentionnées dans le premier chapitre : passage à l'échelle, tolérance aux pannes, transparence des données et autonomie des applications et des données. Nous mentionnons également que dans cette thèse, nous ne nous occupons pas de la gestion des contraintes d'intégrité.

3.3.1 Gestion des transactions et passage à l'échelle en taille

Pour faire face aux besoins des nouvelles applications qui gèrent plusieurs millions d'utilisateurs, il faut des techniques de gestion de transactions efficaces pour garder au bon niveau les performances du système en cas de fortes charges. Cela est d'autant plus vrai que dans [GHOS96], les auteurs ont montré que les techniques usuelles de gestion de transactions ne passent pas à l'échelle si l'environnement est dynamique.

La réplication a été introduite dans les bases de données pour résoudre le problème de passage à l'échelle. Malheureusement plusieurs solutions n'arrivent pas à atteindre une large échelle à cause de deux limitations. Premièrement, la plupart des approches adopte une réplication totale où chaque site stocke intégralement la base de données. Par conséquent, la synchronisation des répliques pour garantir la cohérence mutuelle entraîne une surcharge supplémentaire très importante qui n'améliore guère les performances du système à partir d'un certain degré de réplication [GSN09, BGRS00]. Deuxièmement, les protocoles de réplication utilisées sont synchrones et s'appuient souvent sur le critère de cohérence *1-copy-serializability*. Dans la théorie de la sérialisabilité, un système répliqué est dit *1-copy-serializable* si l'exécution des transactions dans le système répliqué est équivalent à une exécution sérielle sur une seule copie de la base. Ce critère de cohérence exige que toutes les copies soient synchronisées avant de valider une mise à jour. Ceci limite le degré de concurrence et par conséquent le passage à l'échelle. Pour repousser ces limites, de nouvelles solutions de réplication ont été proposées [BGRS00, BKR+99, CMZ05, HSAA03, PMS99, ATS+05, LKMPJP05, RBSS02, PA04, JEAIMGdMFDM08, SPMJPK07, MN09, PA04, GNPV07, BFG+08, LM09, ATS+05, PST+97, PCVO05, LFVM09, FDMBGJM+09, APV07, SSP10, ACZ03]. L'un des principaux objectifs de ces nouvelles approches est d'éliminer le contraignant critère de *1-copy-serializability*. Pour ce faire, la solution la plus utilisée est de retarder l'application des écritures d'une transaction vers toutes les répliques mais aussi de diminuer le volume des données à propager. En effet, le résultat est envoyé au client dès que la transaction est validée sur une des répliques mais avant que toutes les répliques restantes n'appliquent les écritures (données modifiées).

Pour garantir la cohérence, une première solution est la réplication synchrone et consiste à coordonner toutes les répliques au moment de la validation d'une transaction. Ce type de coordination permet de s'assurer qu'il n'y a pas d'incohérences mais aussi d'obtenir des temps de réponses faibles puisque la transaction n'est écrite que sur une seule réplique. Une possible clé de voûte de cette solution est, d'une part, le maintien de plusieurs versions d'une même copie (*Snapshot Isolation*) [BBG+95, PGS97, DS06, CRF09, LKJP+09, LKMPJP05, MN09, PA04, FDMBGJM+09] et d'autre part, l'utilisation de modèles de communication par groupe [PGS03, HAA99, KA00b, SR96, WK05] pour synchroniser les répliques.

Une autre solution consiste à exécuter une transaction et la valider sans faire de coordination entre toutes les répliques au moment de la validation ni au moment de l'exécution (réplication asynchrone). Cette solution utilise des techniques d'ordonnancement afin de définir l'ordre d'exécution

des transactions. Une transaction est exécutée puis validée sur une seule réplique puis les modifications sont propagées plus tard. En cas d'absence de nouvelles transactions de mises à jour, le système converge vers un même état, on parle de cohérence à terme. Parmi les solutions s'appuyant sur cette technique, figurent [GNPV07, BGL+06, BFG+08, LM09, ATS+05, PST+97, Vog09].

Une autre solution est de minimiser le nombre de répliques à synchroniser en fragmentant les données (réplication partielle) [JEAIMGdMFDM08, SPMJPK07, PCVO05, SOMP01, HAA02, PCVO05, LFVM09]. Avec la réplication partielle, la surcharge due aux propagations des mises à jour diminue car les sites ne stockent qu'une partie de la base de données. En fait, si un site ne stocke pas la partie de la base de données modifiée par une transaction, il n'est pas concerné ni par le protocole de validation ni par la propagation des modifications.

Dans la suite de cette section, nous allons décrire quelques travaux s'appuyant sur le *Snapshot Isolation (SI)* ou la cohérence à terme pour gérer la cohérence mutuelle entre les copies. Puis, nous décrivons quelques travaux sur la réplication partielle.

Passage à l'échelle avec Snapshot Isolation

L'une des propriétés les plus importantes du *SI* est que les opérations de lecture seule ne sont pas en conflit avec les opérations d'écriture. Par conséquent les opérations de lecture ne sont jamais bloquées, ce qui augmente le degré de concurrence et améliore les performances du système et particulièrement pour les transactions de lecture seule. Dans la spécification classique du *SI* décrite dans [BBG+95], une transaction obtient une estampille de démarrage (ED) quand elle commence son exécution. Cette estampille indique la dernière version de la base de données (*snapshot*) vue par la transaction. Toutes les opérations de lecture sont faites sur le *snapshot* associé à la transaction. En fait, le *snapshot* associé à une transaction reflète toutes les mises à jour validées avant le début de celle-ci. Quand une transaction met à jour les données, elle produit une nouvelle version qui ne sera correcte (ou cohérente) qu'au moment de la validation. Cependant, il faut remarquer qu'une transaction peut lire à tout moment ses propres modifications avant même leur validation. Au moment de valider une transaction, une seconde estampille dite estampille finale (EF) lui est associée. Les estampilles accordées au début et à la fin d'une transaction permettent de décider si elle peut valider ou annuler ses modifications. En effet, une transaction *T* valide ses écritures (ou modifications) s'il n'existe aucune autre transaction *T'* modifiant la même donnée et dont l'estampille finale est comprise entre les deux estampilles (ED et EF) de *T*. Cet algorithme de validation appelée règle du "*First-Committer-Wins* (FCW)" signifie que deux transactions concurrentes qui modifient les mêmes données ne peuvent pas valider toutes les deux à la fois. En pratique, la plupart des implémentations du *SI* utilisent le verrouillage lors des opérations de modifications pour empêcher qu'une transaction écrive sur une donnée qui est déjà modifiée par une transaction concurrente. La première transaction qui a le verrou sur une donnée est autorisée à la modifier : si la transaction valide et relâche le verrou, toute autre transaction qui était en attente du verrou est annulée. Cette approche connue sous le nom de "*First-Updater-Wins*" produit les mêmes effets que la règle FCW du point de vue des histoires d'exécutions permises. L'algorithme de *SI* est implémenté par les SGBDR Oracle, PostgreSQL, SQL Server 2005, Interbase 4 et Oracle Berkley DB. Il évite les problèmes connus de pertes d'écritures et de lectures sales et introduit des améliorations considérables par rapport au protocole de 2PL et notamment le Strict-2PL en augmentant le débit

transactionnel. Cependant, comme mentionné dans [BBG+95], *SI* ne garantit pas que toutes les exécutions soient sérialisables (au sens des conflits). De plus, il peut entraîner la violation de certaines contraintes d'intégrité par l'entrelacement des transactions concurrentes. Ce problème plus connu sous le nom de *write skew* [BBG+95, CRF09] est illustré dans l'exemple suivant.

Exemple. Supposons l'exécution concurrentielle de deux transactions *T1* et *T2* retirant de l'argent à partir de deux comptes bancaires C_1 et C_2. Les deux comptes sont liées par la contrainte $C_1 + C_2 > 0$. Voici un entrelacement qui peut être obtenu avec *SI* :

$$r_1(C_1 = 50)r_1(C_2 = 50)r_2(C_1 = 50)r_2(C_2 = 50)w_1(C_1 = -20)w_2(C_2 = -30)c1c2$$

Au démarrage des deux transactions, tous les deux comptes ont chacun comme solde 50 euros et à tout moment, chaque transaction isolée, maintient la contrainte $C_1 + C_2 > 0$: quand *T1* valide il calcule la contrainte $C_1 + C_2 = -20 + 50 = 30$ et quand *T2* valide il calcule $C_1 + C_2 = -30 + 50 = 20$. Pourtant les résultats de l'entrelacement produisent $C_1 + C_2 = -50$, ce qui viole la contrainte. Ce problème découle du fait que seuls les conflits "écriture-écriture" sont considérés et par conséquent si deux transactions accèdent en même temps à deux données et que chacune des transactions ne modifie qu'une donnée de manière disjointe, alors toutes les deux transactions seront validées.

Regardons en détail les conséquences d'un tel problème sur une base de données répliquée asynchrone. Comme *T1* et *T2* ne sont pas en conflit "écriture-écriture" sous le protocole *SI*, leur modification peut donc être validée dans l'ordre *T1* suit *T2* sur un site s_1 mais avec l'ordre inverse sur un autre site s_2. Ainsi, si une troisième transaction *T3* lit le contenu des comptes C_1 et C_2 sur le site s_1 obtient une version de la base de données non équivalente à celle qu'elle aurait eu si elle avait consulté le site s_2. Ce problème ne survient jamais avec un système centralisé utilisant le *SI* car soit *T1* suit *T2* soit l'inverse mais pas les deux à la fois. Pour assurer la cohérence mutuelle avec l'utilisation de *SI* et éviter ce problème dans les systèmes asynchrone distribués et répliqués, les transactions doivent être exécutées dans le même ordre sur tous les sites [DS06, LKMPJP05, PA04, EZP05, SPMJPK07, WK05]. Ces approches tentent d'améliorer le degré de concurrence, conformément à l'idée de base du *SI* [BBG+95], dans un environnement répliqué ou chaque réplique utilise localement le *SI* comme protocole de contrôle de concurrence.

Dans [LKMPJP05], les auteurs présentent une solution de réplication basée sur le *SI*, appelé *1-copy-snapshot-isolation*, pour garantir la cohérence mutuelle dans les bases de données répliquées. La solution conçue s'appuie sur un intergiciel et, à l'image de Postgres-R(SI) [WK05] et de Pangea [MN09], est implémentée avec le SGBD relationnel PosgresSQL pour prouver sa faisabilité. L'objectif principal de cette solution est de permettre l'exécution des transactions (lecture seule et écriture) sur n'importe quelle réplique sans savoir au préalable les données sollicitées par la transaction. En effet, cette approche assure le contrôle de concurrence à deux niveaux : chaque SGBD sous-jacent assure le *SI*, et un protocole appelé *SI-Rep* détecte les conflits entre les transactions s'exécutant sur différentes répliques. Au dessus de chaque SGBD, une réplique de l'intergiciel est installée et permet la coopération avec les autres répliques. La communication ou coopération des intergiciels se fait via une communication par groupe. L'exécution d'une transaction T requiert plusieurs étapes. Tout d'abord, T est exécutée sur une réplique locale. A la fin de l'exécution, les

tuples modifiés par T sont extraits sous forme de *writesets*. L'extraction des *writesets* est un mécanisme standard utilisé par plusieurs produits commerciaux et implémentée via des triggers ou du log-sniffing [SJPPMK06]. Bien que les solutions commerciales extraient les *writesets* uniquement après la validation d'une transaction, une extraction avant validation est utilisée de manière similaire aux travaux de [PA04]. Après récupération des *writesets*, *SI-Rep* vérifie si il n'y as pas de conflit "écriture-écriture" entre T et les autres transactions exécutées sur d'autres répliques et qui sont déjà validées. Si aucun conflit n'est détecté alors T est validée au niveau de la réplique locale et les *writesets* sont appliquées de manière asynchrone sur les répliques distantes. Si par contre il y a un conflit, T est annulée localement. Le critère de cohérence sur lequel s'appuie ces travaux est le *1-copy-SI*. Ce critère signifie que l'exécution de plusieurs transactions sur différentes répliques produit le même résultat qu'une exécution sur un système centralisé utilisant *SI* comme protocole de contrôle concurrence. Un système de bases de données répliquées assure le critère de *1-copy-SI* si deux conditions sont garanties :

- l'exécution des transactions suit l'approche ROWA : soit une transaction est validée sur toutes les répliques soit aucune ne l'est ; par contre les transactions de lecture seule sont toujours validées sur une seule réplique.
- Soit WS_l^k et RS_l^k représentant respectivement les *writesets* et *readsets* (données lues) de T_l sur une réplique S^k , alors pour toute paire de transactions T_i et T_j :
  i) si $WS_i^k \cap WS_j^k \neq \emptyset$: l'ordre dans lequel T_i et T_j sont validées est identique à l'ordre produit par un système centralisé utilisant *SI* ;
  ii) si $WS_i^k \cap RS_j^k \neq \emptyset$: si T_i valide avant le début de T_j alors, (1) T_j doit forcément lire les écritures de T_i, (2) cette relation entre T_i et T_j est équivalente à celle produit par un système centralisé utilisant *SI*.

Pour assurer la deuxième condition du critère de *1-copy-SI*, les transactions sont envoyées vers toutes les répliques par multicast en utilisant des primitives de communication par groupe. Avec ces primitives, les transactions sont envoyées dans un ordre total et traitées dans cet ordre sur tous les sites. Il est à remarquer que cette approche et comme celle présentée dans [WK05] peuvent être considérées comme synchrone car les *writesets* sont envoyés à toutes les autres répliques avant que la transaction ne soit validée.

Les solutions basées sur le *SI* offrent de bonnes performances en réduisant le temps de synchronisation entre les répliques tout en maintenant un niveau de cohérence élevé. Ces solutions ne prennent pas en compte les conflits "lecture-écriture", ce qui diminue le nombre de transactions annulées mais aussi la taille des données (données lues non incluses) à envoyer aux répliques pour valider une transaction. Cependant, l'utilisation des communications par groupe pour assurer un ordonnancement total des transactions sur les différentes répliques ne fonctionne que pour des réseaux à latence faible et stable (Cluster, LAN). Dans des systèmes où la latence est non négligeable et l'environnement est dynamique (grille, P2P, ...), la coordination entre les répliques nécessite plus de temps et par conséquent augmente le temps de réponse. Pour des applications de type Web2.0 qui sont conçues sur des architectures à large échelle avec des réseaux WAN, utiliser les mécanismes de réplication *SI* ne semble pas être une solution adaptée pour assurer le passage à l'échelle à cause de la dynamicité et de la faible latence de l'environnement.

Passage à l'échelle avec la cohérence à terme

La cohérence à terme [Vog09] est un niveau de cohérence faible qui stipule que les différentes répliques peuvent diverger pendant une période mais convergent à terme vers un même état en l'absence de nouvelles transactions entrantes. Ainsi, une séquence d'accès sur les différentes répliques ne retourne pas nécessairement la version la plus à jour. La cohérence à terme est de plus en plus utilisée dans les nouvelles solutions pour faire face aux besoins des applications web notamment dans les *cloud*. Parmi les solutions dédiées aux *clouds* et qui utilisent une cohérence à terme, nous pouvons citer Cassandra [LM09] et Amazon S3 [BFG+08]. Cependant, ces solutions du *cloud* qui commencent à émerger ne s'appuient pas sur les traditionnels SGBD et nécessitent de nouvelles manières de concevoir les SGBD et les applications, ce qui les éloignent un peu de notre contexte. Pour rappel, notre objectif dans cette thèse est de s'appuyer les SGBDs existants pour concevoir une solution de traitement de transactions à large échelle.

Des approches qui utilisent les SGBDs classiques pour assurer le traitement des transactions sont décrites dans [DS06, PA04, RBSS02]. Dans ces approches, une réplication asynchrone avec une configuration maître-esclave est utilisée. En plus, il y a une séparation des transactions de lecture seule des transactions d'écriture. Précisément, les transactions de mises à jour sont exécutées sur le site primaire alors que les transactions de lecture seule sont envoyées sur les sites secondaires. Les modifications faites par les transactions de mises à jour sont propagées vers les autres répliques à travers des transactions de rafraîchissement. Une transaction de rafraîchissement est utilisée pour propager les transactions de mise à jour sur les autres répliques.

En plus, pour garantir la cohérence globale, les transactions de rafraîchissement doivent être appliquées dans un ordre compatible à l'exécution des transactions d'écriture correspondantes sur le site primaire. De manière plus précise, si deux transactions $T1$ et $T2$ produisent respectivement les transactions de rafraîchissement $R1$ et $R2$, alors si $T1$ valide avant $T2$ sur le site primaire, donc sur n'importe quel site secondaire $R1$ précède $R2$. Pour atteindre ce but, une liste FIFO des transactions validées est utilisée dans [DS06] afin d'envoyer les transactions de rafraîchissement conformément à l'ordre obtenu sur le site primaire. Cependant pour accélérer l'application des transactions de rafraîchissement sur les sites secondaires, la base de données locale les exécute simultanément en associant un *thread* à chaque transaction de rafraîchissement mais en veillant à ce que les écritures faites par une transactions soient visibles à toutes celles qui la suivent dans la file FIFO. Les travaux menés dans [DS06] assurent aussi la cohérence globale par session. En d'autres mots, si un client envoie une première transaction de modification, T_m, puis une autre de lecture seule, T_l, alors cette dernière accédera à la copie des données incluant la modification faite par Tm. Cette propriété n'est pas garantie par toutes les solutions de réplication basées sur le *SI* notamment [FLO+05].

Cependant, l'utilisation d'une architecture maître-esclave ne facilite pas le passage à l'échelle car : *i*) le site maître devient très rapidement une source de congestion si le nombre de mises à jour augmente, *ii*) si le nombre de noeuds esclave est important, la synchronisation entre maître et esclaves handicape la disponibilité du noeud maître à traiter de nouvelle requêtes entrantes. Les solutions proposées pour les *clouds* requièrent souvent des data centres qui nécessitent des mécanismes de maintenance très coûteux. Pour remédier à ces limites, nous envisageons de concevoir des solutions multi-maîtres tout en assurant la cohérence à terme. Notre solution s'intègre aux tra-

ditionnels SGBDs existants et donc ne requiert aucune modification de leur conception ni de celles des applications qui les utilisent.

Par ailleurs la divergence (obsolescence) introduite par l'utilisation de la cohérence à terme, permet de minimiser les synchronisations entre répliques et donc améliore significativement les performances si la divergence est contrôlée [GN95]. De plus, dans le contexte du web2.0, de nombreuses applications tolèrent une cohérence relâchée et acceptent de lire des données qui ne sont pas nécessairement les plus récentes ; cela ouvre la voie vers de nouvelles solutions offrant de meilleures performances en termes de débit transactionnel, latence, disponibilité des données et passage à l'échelle. Par exemple, il est possible de gérer des transactions de vente aux enchères (sur eBay ou Google Adsense) sans nécessairement accéder à la dernière proposition de prix, puisque l'enchère est sous pli cacheté, autrement dit, on lit les quelques informations relatives au produit sans lire la dernière proposition de prix fait sur ce produit. L'application doit pouvoir spécifier la limite de divergence tolérée ainsi que la nature de cette divergence en fonction de sa sémantique. Le système doit quant à lui garantir que cette limite est respectée. Différents modèles de divergence ont été proposés dans la littérature [GN95, LGV04]. Pour définir a divergence, on peut utiliser des mesures temporelles, numériques, par version, mixte, etc. [Pap05]. De plus, la nature de la divergence tolérée doit être fonction de l'application. Les approches de contrôle best-effort [PA04, LKMPJP05, PMS99] permettent de minimiser la divergence temporaire des données mais ne tirent pas profit de la divergence autorisée, et ne garantissent pas non plus qu'elle reste bornée. Le projet MTCache [GLRG04] borne la divergence mais a l'inconvénient de nécessiter la modification du gestionnaire de transactions. De plus, il ne prend en compte que la mesure temporelle. Néanmoins, les algorithmes proposés nécessitent de modifier le gestionnaire de transaction, par exemple en étendant les verrous avec des compteurs [Pu91, WYP97, YV00]. Le contrôle de la divergence de certaines mesures (mesure numérique) nécessite un mécanisme de détection des conflits à la granularité fine. Néanmoins, la seule méthode exacte et non intrusive (indépendante) pour le gestionnaire de transaction est l'analyse de journal, réputée lourde. Dans cette thèse, nous utilisons des techniques non intrusives pour contrôler la divergence tout en se passant des techniques d'analyse de journal. En définissant la divergence d'une réplique comme étant le nombre de transactions manquantes sur cette réplique, nous utilisons le catalogue réparti qui stocke des informations pour calculer à tout moment la divergence.

Passage à l'échelle avec réplication partielle

La réplication totale d'une base de données passe mal à l'échelle puisque toutes les mises à jour doivent être appliquées sur tous les sites [JPMPAK03], et qu'une augmentation du nombre de sites ne fait qu'augmenter la surcharge du système. Dans ce contexte, la réplication partielle a plus de sens et consiste à diviser la base de données en plusieurs portions et à répliquer chaque portion dans un sous-ensemble des sites du système [SOMP01, JEAIMGdMFDM08, HAA02, SSP10]. L'un des gains visés par la réplication partielle est la réduction de la taille des données à propager vers les répliques pour vérifier les conflits. Ce gain est d'une utilité capitale si le nombre de répliques est très élevé puisque la surcharge du système est tributaire de la quantité des informations à envoyer et du nombre de sites destinataires. Par ailleurs, avec les applications Web 2.0, les utilisateurs ne modifient qu'une faible portion de leur données personnelles. Ce faisant, même si le nombre d'utilisa-

teurs est de l'ordre de centaines de millions, les accès aux données sont disjoints et par conséquent une répartition des données augmente le degré de concurrence et diminue le temps de réponse. Par conséquent, le système passe plus facilement à l'échelle. Dans [HAA02], les auteurs proposent un algorithme de réplication partielle dans un environnement WAN. Chaque donnée a un ou plusieurs sites permanents qui en stockent une copie. Le protocole de réplication utilise une communication multicast épidémique pour propager les logs des bases données. Les logs (*readsets* et *writesets*) utilisés pour ordonner les transactions sont envoyés sur l'ensemble des sites indépendamment du fait qu'ils stockent ou non les données modifiées. La différence principale de cette approche par rapport à leur protocole de réplication totale présenté dans [HAA00] est qu'un site n'applique les modifications que pour les données qu'ils stockent. Ceci constitue un problème car envoyer des logs à des sites qui ne stockent pas les données modifiées n'est qu'une source de surcharge de trop, surtout si le nombre de site est élevé. Par contre dans [SOMP01], les logs ne sont envoyés qu'aux sites stockant une portion des données sollicitées par la transaction. Bien que cela réduit la surcharge notée dans les travaux de [HAA02], il demeure que l'utilisation du critère de cohérence de *1-copy-serializability* sur laquelle se base l'approche rend la solution impraticable dans les systèmes à environnement dynamique ou à latence faible. Dans [SPMJPK07, JEAIMGdMFDM08] le protocole de réplication partielle utilisé s'appuie sur le *SI*. Par conséquent, seules les données modifiées (*writsets*) sont envoyées pour détecter les conflits au moment de la validation, ce qui réduit le nombre de transactions annulées. Si les données sollicitées par une requête se trouvent sur un seul site alors la transaction peut être exécutée sur ce site. Par contre, si les données sont réparties sur plusieurs sites, alors la transaction doit être distribuée tout en préservant la cohérence mutuelle. Bien que la plupart des travaux sur la réplication partielle suppose qu'une transaction ne s'exécute que sur un seul site, la solution présentée dans [SPMJPK07] étudie par contre le cas des transactions distribuées.

L'objectif des auteurs de [SPMJPK07] est d'assurer la cohérence globale pour les transactions distribuées bien qu'aucun site n'ait une connaissance globale du système. Pour ce faire, un coordinateur est choisi pour chaque transaction et correspond au site qui stocke au moins quelques unes des données requises par les premières opérations de la transaction. Si le coordinateur stocke toutes les données sollicitées, il exécute la totalité des opérations et valide la transaction. Autrement, il envoie les opérations restantes aux sites contenant les données non disponibles sur le coordinateur. Si un site reçoit des opérations à exécuter, il envoie après exécution les résultats et les *writesets* au coordinateur qui peut alors initialiser la phase de validation. Lors de la phase de validation, le coordinateur envoie par multicast les *writesets* et l'estampillage de la transaction qui lui est attribuée à son démarrage. Ainsi toutes les répliques peuvent valider la transaction en s'appuyant sur l'estampille de la transaction et les conflits "écriture-écriture". L'inconvénient de cette approche est qu'il est bloquant car si un site ne renvoie pas sa réponse (résultats et les *writesets*) pour une quelconque raison (panne, latence faible, site chargé, ...), la transaction ne peut pas être validée. En outre, l'envoie des writesets à toutes les répliques est quasi-identique à une réplication totale avec laquelle les transactions de mises à jour ne contiennent pas beaucoup d'opérations de lecture.

Dans [JEAIMGdMFDM08], l'approche aborde la gestion des transactions distribuées dans le même sens que dans [SPMJPK07]. L'une de leurs différences est que dans [JEAIMGdMFDM08], le coordinateur (site maître de la transaction) transmet les opérations qui touchent des données

stockées dans d'autres sites en envoyant, souvent, les *writesets* des opérations déjà faites sur le coordinateur. Ceci s'explique par le fait qu'une opération O_j envoyée sur un site distant peut avoir besoin des écritures de O_i effectuée sur le coordinateur. En outre, quand une opération O_j avec ses *writesets* sont reçus par un site secondaire S_k, toutes les transactions locales sur S_k sont annulées pour appliquer les *writesets* de O_j puis l'exécuter. Ensuite, les *writesets* de O_j sont envoyés au coordinateur qui peut valider la transaction si toutes les opérations distantes ont réussi. Tant que la transaction n'est pas validée par le coordinateur, aucune autre opération d'écriture n'est permise sur S_k. Cependant, l'opération O_j peut être aussi bloquée par des transactions globales en phase de certification sur S_k et dans ce cas elle sera mise en attente. Outre le fait que ce protocole est bloquant, l'annulation des transactions peut avoir un impact très négatif sur le système : *i*) une transaction locale ayant déjà fait toutes ses opérations est reprise même s'il ne lui reste que la validation, *ii*) l'annulation des transactions locales augmente la charge d'un site, le rendant du coup moins disponible pour traiter et participer à la validation des transactions globales.

P-Store[SSP10] est une solution de réplication partielle pour des données de type "clé-valeur" stockées sur un WAN. Lors de l'exécution d'une transaction seuls les sites contenant une copies des données lues et/ou modifiées sont synchronisés. Ceci réduit considérablement la charge de certains nœuds qui peuvent dès lors exécuter en parallèle d'autres transactions, ce qui augmente le passage à l'échelle. Une transaction globale (qui sollicite des données stockées sur plusieurs sites) T, est pilotée par un coordinateur appelée *Proxy(T)*. Les opérations de lecture d'une transaction sont exécutées de manière optimiste et à la validation le *Proxy* initie une phase de certification pour assurer le critère de *1-copy-serializability*. L'un des inconvénients de cette solution est qu'il se base sur le critère de *1-copy-serializability* et par conséquent les transactions de lecture seule sont bloquées par les transactions d'écriture dès qu'elles ne sont pas locales. En outre, la notion de transaction décrite n'est pas identique à la notion de transaction dans les bases de données puisque les opérations sont très simplistes et consistent à accéder à une donnée via sa clé. En d'autres termes, les transactions autorisées sont des transactions basées sur la clé : il n'est pas possible de faire des transactions qui utilisent la valeur (ou un attribut de la valeur) comme prédicat ni de faire des requêtes par intervalle.

En conclusion, l'objectif de la réplication partielle est de réduire les situations de conflits (pour diminuer les reprises de transactions) et donc de traiter plus de requêtes de manière parallèle. Une bonne solution est d'éviter des mécanismes d'exécution de transaction bloquants qui génèrent souvent des annulations et donc plusieurs reprises. Certes, faire l'hypothèse que le partitionnement des données peut être parfait à tel point qu'une transaction puisse se tenir sur une seule partition est très irréaliste. Cependant, les solutions proposées pour faire face à ce problème sont bloquantes et ne s'éloignent pas du principe du *2-PC* avec des aller-retours entre *master* (coordinateur) et sites distants (participants). En outre, le partitionnement des données doit permettre de minimiser le temps de synchronisation par réduction de la taille des données mais aussi du nombre de sites à synchroniser. De ce fait, les solutions qui coordonnent toutes les répliques lors des phases de validation même si ces dernières ne stockent pas les données manipulées s'éloignent de cet objectif. Par conséquent, pour un meilleur passage à l'échelle avec l'utilisation de la réplication partielle, il faut envisager des solutions non bloquantes et qui nécessitent une faible synchronisation des répliques lors de l'exécution des transactions.

3.3.2 Gestion des transactions et disponibilité

La gestion des transactions dans les bases de données répliquées nécessite la prise en compte de la disponibilité des répliques pour assurer une cohérence mutuelle. En effet, il y a plusieurs motivations qui encouragent la gestion de la disponibilité dans les bases de données répliquées. La première raison est de pouvoir borner le temps de réponse de la transaction : si elle est envoyée sur un site qui tombe en panne avant sa validation, il faut pouvoir continuer le traitement de la transaction sur une autre réplique afin de pouvoir répondre aux clients dans des délais acceptables. Une deuxième raison est qu'avec des systèmes très volatiles (connexion et déconnexion fréquentes des répliques), il est important de gérer l'indisponibilité de certaines ressources afin de minimiser la dégradation des performances en cas de présence de pannes. Une autre raison est de maintenir les copies identiques sur toutes les répliques : si une copie est indisponible lors d'une synchronisation, il faut à son retour lui envoyer toutes les modifications qu'elle n'a pas pu recevoir durant son absence. Cependant, il est à remarquer que l'introduction de modules pour gérer la disponibilité entraîne des surcharges dans le système et réduit donc le débit transactionnel. Par conséquent, des compromis doivent être trouvés pour éviter de trop surcharger le système et de gérer efficacement l'indisponibilité des répliques pouvant compromettre la cohérence du système.

Dans le cadre des bases de données répliquées deux cas peuvent affecter la disponibilité du système à savoir les déconnexions prévues et celles intempestives appelées souvent pannes. Les déconnexions prévues causent moins de problèmes car elles sont connues à l'avance et prises en compte dans le processus de traitement en cours. Par contre, les déconnexions intempestives survenant lors d'un processus de traitement, peuvent occasionner de sérieux problèmes de cohérence. C'est pour cela qu'elles ont attiré une attention toute particulière dans les récents travaux sur la réplication [PA04, ADM06, Sch90, VBLM07, AT89, JPMPAK03, PRS07, BHG87, PMJPKA05, APV07]. Une des premières approches utilisées pour gérer les pannes est de les masquer ou de les rendre transparentes vis-à-vis du client. Pour ce faire, le noeud effectuant les mises à jour envoie une transaction à toutes les répliques. Les répliques exécutent la transaction simultanément et envoient les résultats (ou acquittements des copies mises à jour) au noeud. Ce dernier attend soit la première réponse d'une réplique soit une majorité de réponses identiques (*quorum*) avant de décider de terminer l'exécution de la transaction. Cette approche de traitement des mises à jour plus connue sous le nom de réplication active ou *state-machine approach* [GS97, Sch90] et parfois sous le nom d'algorithme à base de quorum [Gif79, JPMPAK03, VS05] cache au client l'occurrence d'une panne d'une réplique durant l'exécution d'une transaction. Le principal problème de cette approche est qu'elle réduit les performances du système car à un instant donné, toutes les répliques exécutent la même transaction, ce qui réduit sensiblement le degré de concurrence. En outre pour envoyer une transaction à toutes les répliques, il est nécessaire de les connaître toutes et de pouvoir les localiser.

Cette technique de masquage est également utilisée dans [PRS07] où tous les noeuds stockant une même portion des données sont regroupés dans une même cellule. Un réseau logique structuré est construit aux dessus des cellules formées. Chaque cellule est un groupe de machines physiques dynamiquement paramétré et utilise le *state-machine approach* décrite dans [Sch90, Sch93]. Ce faisant, si un noeud au sein d'une cellule tombe en panne, celle-ci est masquée. Par contre, cette solution fait l'hypothèse qu'une cellule entière ne peut tomber en panne et si plusieurs noeuds

d'une même cellule tombent en panne simultanément, cette dernière s'auto-détruit et les données sont redistribuées sur les cellules voisines. Ainsi, la panne de certains noeuds dans une cellule entraîne la déconnexion des membres du groupe sur lesquels on pouvait faire recours pour une meilleure disponibilité. Outre son coût, la redistribution des données sur les cellules voisines peut les rendre plus chargées et donc réduire leur performance.

Une deuxième solution pour gérer les pannes consiste à les détecter d'abord et à les résoudre après. Avec cette technique, il est beaucoup plus difficile de rendre la panne transparente car sa détection avant sa résolution introduit une latence. Plusieurs mécanismes de détection de pannes ont été proposés [CT96, LAF99, ACT99]. Ces mécanismes sont soit basés sur des échanges périodiques de messages de vie [ACT99], soit sur des allers retours "ping/pong" [DGM02]. Avec la première technique, chaque noeud envoie périodiquement un message de vie à tous les noeuds et attend, à son tour, un message de vie de chacun d'eux à chaque période. L'inconvénient majeur de cette technique est le modèle de communication "tous-vers-tous" qui engendre beaucoup de messages quand le nombre de sites est important. La deuxième méthode permet une détection plus ciblée car elle permet de ne surveiller qu'un sous-ensemble de noeuds. Cette deuxième approche est beaucoup plus adaptée dans les systèmes à large échelle qui contiennent des milliers de noeuds qui ne se connaissent pas tous.

Une fois la panne détectée, il faut l'identifier pour savoir quels mécanismes utiliser afin de la gérer. Il existe plusieurs types de pannes classées en général en trois catégories :

- **Panne franche ou crash (*fail-stop*) :** cette défaillance entraîne l'arrêt total du composant. Avant cette panne le processus a un comportement normal et à partir de celle-ci, le processus cesse définitivement toute activité.
- **Panne transitoire ou ommission (*omission failure*) :** avec cette panne, le composant cesse momentanément son activité puis la reprend normalement.
- **Panne byzantine (*byzantine failure*) :** ce type de panne entraîne le système dans un comportement imprévisible. Ce type de panne représente l'intégralité des comportements possibles. Tout système qui tolère les pannes byzantines peut tolérer tout autre type de pannes.

Dans le contexte des systèmes répliquées, beaucoup de travaux ont été proposés pour faire face aux pannes de type *fail-stop* [PA04, BHG87, MN09, LKMPJP05, LKJP$^+$09, PRS07, PMJPKA05, APV07, CPW07, SSP10] mais aussi de type byzantine [VBLM07, CL02, CVL10]. En général les solutions proposées pour faire face aux pannes byzantines nécessitent une synchronisation de plusieurs répliques avant la validation de toute transaction. Malheureusement cette synchronisation est quasi-impossible à réaliser dans le cas d'un système à grande échelle ou génère une surcharge en termes de messages très important. C'est la raison pour laquelle la plus part des travaux effectués dans le domaine des bases de données répliquées se concentrent plus sur les pannes de type *fail-stop*. Par exemple dans Ganymed [PA04], Middle-R [PMJPKA05] et Pangea [MN09], les auteurs décrivent des solutions de gestion de pannes très simples basées sur une architecture maître-esclave. En fait si le noeud maître (coordonnateur des mises à jour) tombe en panne, un noeud secondaire est choisi pour le remplacer. Il faut remarquer qu'avec Middle-R il n'existe pas un seul noeud maître qui coordonne toutes les transactions mais plutôt un noeud maître pour chaque classe de conflit. Dans Pangea, les clients sont invités à renvoyer au nouveau maître toutes les transactions qui n'ont pas été validées avant l'arrivée du crash. Il faut noter que l'occurrence

d'une panne ne peut pas compromettre la durabilité des transactions car le protocole de réplication est totalement synchrone. Cependant avec Ganymed, le protocole utilisé est asynchrone et donc outre le fait d'élire un nouveau site maître, il faut garantir que toutes les transactions qui ont validé avant le crash soient pérennes. Pour ce faire, un client ne peut recevoir la notification de la validation d'une transaction que si les *writesets* ont été déjà appliqués sur un certain nombre de répliques. Ce retard de notification de la fin d'une transaction augmente le temps de réponse. Quant à l'approche de Middle-R, si le noeud maître d'une classe de conflit tombe en panne, le premier noeud sur la vue (liste des noeuds actifs et connectés) est élu maître et est chargé de continuer toutes les transactions non encore validées. La liste des transactions non validées est connue par le nouveau maître car à chaque fois qu'une transaction est envoyée à un noeud, celui ci l'envoie par *multicast* à tous les autres membres du groupe de la classe de conflits. Cependant, dans tous les deux systèmes (Pangea et Ganymed), si un noeud secondaire tombe en panne celui-ci est ignoré jusqu'à l'intervention manuelle de l'administrateur tandis que Middle-R le supprime simplement de la liste des noeuds actifs et qui peuvent recevoir des messages. Par conséquent, la panne de plusieurs noeuds secondaires dans une courte période pousse le noeud maître à devenir une source de congestion mais aussi de panne totale du système.

UMS/KTS [APV07] aborde la gestion de versions des données dans des systèmes pair-à-pair structurés reposant sur une table de hachage distribuée. La cohérence mutuelle est garantie à l'aide d'un service d'estampillage, tolérant aux pannes, qui permet de retrouver efficacement la version courante d'une réplique. A chaque clé est associé un noeud chargé de gérer l'estampillage de la donnée associée à cette clé. A chaque fois que le noeud responsable de l'estampillage tombe en panne, un autre noeud est choisi pour le remplacer. Cependant la disponibilité des noeuds stockant les données n'est pas étudiée, ainsi une incohérence peut se produire si un noeud stockant la dernière version d'une donnée quitte le système avant d'avoir propagé sa donnée.

En conclusion, plusieurs solutions sont proposées pour faire face aux pannes dans les systèmes distribués. Certaines ne permettent pas de passer à l'échelle, en l'occurrence la réplication active ou *state-machine approach*, car introduisant des surcharges qui ralentissent le fonctionnement du système. D'autres proposent des solutions avec un modèle maître-esclave (réplication passive) tout en s'intéressant essentiellement à la panne du noeud maître. L'inconvénient de cette seconde solution est qu'elle nécessite une détection au préalable de la panne du noeud maître puis suivie d'une élection d'un nouveau maître. Cette phase de transition peut nécessiter un temps considérable, surtout dans les systèmes à large échelle et hélas elle est souvent ignorée. En plus, comme la panne des noeuds secondaires n'est pas prise en compte, le noeud maître devient très rapidement une source de congestion, ce qui limite les performances du système en cas de forte dynamicité. Pour une meilleure prise en compte des pannes dans les systèmes distribués à large échelle, il faut s'assurer de deux choses : *i*) la détection des pannes doit être peu coûteuse en termes de messages : il faut pour cela utiliser des techniques de détection ciblée et en fonction du type de noeud surveillé ; *ii*) la détection doit se faire dans les meilleurs délais, et pour tout type de noeud défaillant, afin de pouvoir donner une suite positive à toute transaction qui se trouvait sur ce noeud en panne.

3.3.3 Gestion transparente des transactions avec transparence et autonomie

L'approche la plus classique pour implémenter la réplication est de l'intégrer au coeur du SGBD [BKR+99, KA00a, BHG87, HSAA03, SSP10]. Cependant, cette approche présente quelques limites et compromet l'autonomie des bases de données. Premièrement, elle nécessite un accès aux codes sources du SGBD, ce qui signifie que seuls les propriétaires des produits commerciaux peuvent l'utiliser. Deuxièmement, elle est fortement couplée avec les autres modules du SGBD, créant du coup une interdépendance avérée des composants du même produit, ce qui ne facilite pas les maintenances et les évolutions du protocole de réplication. Enfin, cette absence de transparence entraîne le client à interroger plus fréquemment certaines répliques au détriment d'autres, ce qui ne facilite pas une meilleure exploitation des ressources notamment l'équilibrage des charges.

En outre, nous avons mentionné dans le chapitre précédent que la transparence permet de cacher aux utilisateurs les détails techniques et organisationnels d'un système distribué ou complexe. L'intérêt visé est de faire bénéficier aux applications d'une multitude de services sans avoir besoin de connaître exactement la localisation ou les détails techniques des ressources qui les fournissent. Dans le cas d'une base de données répliquée, il s'agit essentiellement de cacher la distribution des données (répartition et localisation des répliques) mais aussi l'indisponibilité de certaines ressources. Cette transparence a pour gain : *i*) de mieux exploiter les répliques disponibles en faisant une bonne répartition des charges, *ii*) dans le cas où les transactions de lecture seule acceptent des données obsolètes, il devient plus simple de les router vers les noeuds pouvant satisfaire leur exigence afin de réserver les répliques totalement à jour pour les transactions demandant une forte cohérence.

Pour assurer cette transparence de la réplication, des solutions basées sur des intergiciels ont été largement étudiées dans les dernières années [GNPV07, CPW07, PMJPKA05, CMZ05, LKMPJP05, ACZ03, PA04, PCVO05, RBSS02, MN09]. Avec cette approche, l'interface utilisateur (ou client) du SGBD est utilisée pour faire la médiation entre applications et bases de données. Ce faisant, les protocoles de réplication peuvent être modifiés sans impacter le SGBD, ce qui garantit l'autonomie des SGBD. De plus, le client n'a plus besoin de connaître la localisation et/ou répartition des données dont la connaissance est confiée à l'intergiciel. L'intergiciel garde le niveau de cohérence de chaque réplique (la fraîcheur de chaque réplique) afin de pouvoir router toute transaction sur la réplique satisfaisant ses exigences de fraîcheur. Par ailleurs si le SGBD est réparti, cette approche permet une meilleure exploitation des ressources disponibles et donc un équilibrage de charge de bonne qualité.

Sprint [CPW07] est un intergiciel offrant de hautes performances et une haute disponibilité pour un SGBD en mémoire et répliqué. Sprint dissocie trois types de noeuds logique : *i*) *Edge Server (ES)* qui joue le rôle d'interface entre le client et le reste du système ; *ii*) *Data Server (DS)* qui stocke une base de données en mémoire et exécute les transactions sans accès au disque dur ; *iii*) *Durability Server (XS)* qui assure la durabilité des transactions et la reprise après panne. Une transaction est envoyée par un client à un serveur *ES* qui se charge de l'envoyer sur le ou les *DSs* qui stockent les portions de données requises, ce qui garantit une transparence totale vis-à-vis du client. Si la transaction est en lecture seule, elle est validée sans aucun problème par le serveur *ES* qui se charge de son exécution. Par contre, quand il s'agit d'une transaction de modification, le serveur *ES* coordonne la validation en contactant tous les serveurs *DS* qui ont participé à l'exécution de

la transaction. En effet, tout serveur *DS* qui est prêt à valider envoie par multicast son vote au serveur *ES*, à tous les serveur *DS* participant à la transaction et à tous les serveurs *XS* pour assurer la durabilité. Si tous les serveurs *DS* votent "*commit*", la transaction est validée autrement elle est annulée. Pour garantir que les toutes transactions s'exécutent dans le même ordre sur tous les serveurs, une communication par groupe d'ordre total (*total order multicast*) est utilisée. Si les pannes ne sont pas fréquentes ou si les noeuds logiques se trouvent dans un même réseau, cette approche garantit de bonne performances en termes de temps de réponse. Par contre en cas de panne ou d'un environnement en grande échelle ces performances ne sont plus garanties à cause des pannes fréquentes ou des latences du réseau faible (occasionne des suspicions de pannes) qui entraîne l'annulation et la reprise de plusieurs transactions. De plus, l'autonomie des SGBD est compromise car le protocole de validation nécessite la mise en oœuvre de protocole de terminaison sur chaque participant afin de garantir l'ordre total.

FAS [RBSS02] est un intergiciel de réplication mono-maître et asynchrone. Il prend en compte la fraîcheur des SGBD afin de garantir que les exigences de fraîcheur d'une requête soient satisfaites. Il transmet les transactions sur le SGBD maître, et les requêtes de lecture sur le nœud le moins chargé. La synchronisation des répliques est différée périodiquement. FAS étant mono-maître, cela ne permet pas de supporter une charge transactionnelle croissante. De plus si aucun SGBD n'est suffisamment frais pour traiter une requête, celle-ci est mise en attente, ce qui peut provoquer la surcharge d'un SGBD au moment où il devient disponible pour traiter les requêtes en attente. Dans ce cas précis, la synchronisation anticipée des répliques aurait été bénéfique. Il faut remarquer aussi que l'approche de FAS est quasi-similaire à celle de Ganymed [PA04] en dehors du fait que cette dernière utilise un modèle de communication par groupe pour garantir le *1-copy-SI* décrit précédemment.

C-JDBC [CMZ05] est un intergiciel de réplication gérant un cluster de SGBD. Etant conçu comme un pilote JDBC, il permet à l'utilisateur de traiter des transactions de manière transparente. La stratégie de routage est simple et efficace : chaque requête de lecture seule est envoyée à un SGBD différent à tour de rôle, chaque transaction est diffusée à tous les SGBD. La cohérence des répliques n'est pas garantie car la première réplique ayant fini de traiter une transaction est désignée pour servir de référence sans tenir compte des autres répliques. Ainsi, cette solution est restreinte à un environnement stable.

Leg@net [GNPV07] est une solution de réplication multi-maîtres pour le routage de transactions dans un cluster de bases de données. Leg@net relâche autant que possible la fraîcheur des données, dans les limites acceptées par les requêtes. Cela réduit le surcoût de synchronisation des répliques et permet ainsi d'allouer davantage de ressources au traitement des transactions. Leg@net cible les applications transactionnelles dont l'autonomie doit être préservée. Toutefois, cette solution manque de passage à l'échelle car l'intergiciel est centralisé.

Certes, le liste des travaux cités dans cette section n'est pas exhaustive mais elle reflète la quasi-totalité des approches de gestion des transactions dans une base de données répliquée à travers un intergiciel. La plupart des approches ne passent pas souvent à l'échelle pour plusieurs raisons parmi lesquelles, nous pouvons citer :

– l'utilisation de communication par groupe pour synchroniser les répliques [PMJPKA05, MN09] : cette technique nécessite un environnement stable comme les clusters ou les LAN ;

– une configuration mono-maître ou une architecture centralisée qui ne supporte pas une charge transactionnelle croissante et en même temps constitue une source de pannes.
– une cohérence très forte qui exige que toutes les répliques soient synchronisées (ou bloquées) pour le traitement d'une transaction de mise à jour. En plus, chaque transaction requiert toutes les modifications des transactions conflictuelles qui la précèdent, ce qui empêche l'exécution simultanée ou en parallèle de plusieurs transactions ;

Les applications comme celle du Web 2.0 requièrent une forte disponibilité et des performances très importantes pour des raisons économiques. En effet ces applications sont entretenues grâce à l'argent obtenu à partir des sponsors et des publicités qui se font rares dès que le système est trop souvent indisponible. Ainsi pour satisfaire les besoins de telles applications il faut une gestion des transactions efficaces et par conséquent des intergiciels décentralisés pour mieux absorber la charge applicatives. Cette décentralisation permet à l'intergiciel d'être très disponible et d'assurer un accès rapide et pas nécessairement cohérent aux données, tout en exploitant au mieux l'ensemble des ressources.

3.4 Discussion

Nous avons étudié dans les trois sections précédentes la gestion des transactions dans une base de données répliquées en privilégiant trois dimensions à savoir le passage à l'échelle, la disponibilité et enfin la transparence. Nous avons présenté quelques solutions existantes et leurs limites qui empêchent leur réutilisation dans un environnement à très grande échelle. Nous récapitulons à présent dans cette section les choix que nous avons jugés judicieux pour repousser les limites des solutions existantes afin de mieux prendre en compte les besoins des applications Web 2.0.

3.4.1 Modèle de réplication pour les bases de données à large échelle

Le choix d'un modèle de réplication exige la prise en compte de plusieurs paramètres ou dimensions qui dépendent essentiellement des applications visées. Comme décrit par le théorème CAP (Consistency-Availability, Performance) [Bre00], il est impossible d'assurer à la fois la cohérence, la disponibilité et la performance dans un système distribué. Par conséquent, pour satisfaire les besoins des applications de type Web2.0 qui génèrent un workload avec lectures intensives, nous avons préféré la disponibilité et la performance. Pour la simple raison que le relâchement de la cohérence est souvent toléré par les applications que nous ciblons et permet d'avoir un bon passage à l'échelle [FJB09, GL02], le modèle de réplication que nous voulons mettre en oeuvre s'appuie sur les principes suivants :

– une configuration multi-maître qui donne la possibilité de répartir aussi bien les opérations de lecture que les opérations d'écriture sur l'ensemble des répliques, ce qui permet une parallélisation des traitements. En plus, cela permet d'éviter la surcharge de certaines ressources au détriment d'autres, ce qui aboutit à un meilleur équilibrage des charges ;
– une réplication asynchrone pour éviter de synchroniser toutes les répliques lors des mises à jour, mais aussi pour réduire le temps de réponse des transactions. Entre la validation d'une transaction sur une réplique et la propagation de ses résultats sur les autres répliques, il peut

y avoir une divergence entre les copies. Cette divergence copies est tolérée (ou introduite délibérément) pour améliorer les performances des opérations de lecture. Cependant, elle doit être toujours contrôlée ou bornée en fonction du niveau de cohérence exigé par les applications ;
– les utilisateurs des applications Web2.0 modifient en général une faible portion de leurs propres données. Ainsi, une réplication partielle permettra d'avoir un grain d'accès aux données beaucoup plus fin et par conséquent, améliore le degré de concurrence.
– la propagation des mises à jour se fait exclusivement entre deux sites, celui qui envoie et celui que reçoit. La plupart du temps, elle est initialisée pour rendre cohérente une réplique qui doit traiter une nouvelle requête ou valider une transaction. Si la propagation est initialisée pour valider ou non une transaction, les *datasets* (ensemble des données lues par la ou les transactions à propager) sont envoyés. Autrement, la propagation par envoi du code des mises à jour est utilisée pour éviter de surcharger le réseau. Nous supposons aussi qu'une transaction ne contient pas de clauses SQL (e.g LIMIT, RANDOM, ...) qui ne reproduisent pas le même résultat quelque soit la réplique sur laquelle elle est exécutée [EGA08]. En d'autres termes, cette hypothèse permet de garantir que l'exécution d'une transaction sur deux répliques identiques donne le même résultat. Par ailleurs quand il s'agit de valider une transaction, l'approche PULL sera utilisée par la réplique qui veut valider une transaction. Dans le cas contraire c'est l'approche PUSH qui est utilisé et en général sous le contrôle d'un TM.
– la gestion de la réplication via un intergiciel permet de faire moins de modifications sur les bases de données mais aussi de mieux contrôler les ressources disponibles.

3.4.2 Modèle de middleware pour les bases de données distribuées et répliquées

L'étude des middleware a permis de bien comprendre leurs caractéristiques mais aussi leurs avantages. La fonction essentielle du middleware est d'assurer la médiation entre les parties d'une application, ou entre applications elles même. Par conséquent, les considérations architecturales tiennent une place centrale dans la conception du middleware [Kra09]. L'architecture couvre l'organisation, la structure d'ensemble, et les schémas de communication, aussi bien pour les applications que pour les composants du middleware. Dans le contexte d'un système de bases données distribuées, le problème soulevé est soit la persistance (conservation à long terme et procédures d'accès) soit la gestion des transactions (maintien de la cohérence pour l'accès concurrent aux données en présence de défaillances éventuelles). Si la base de données est répliquée, le maintien de la cohérence se traduit le plus souvent par la gestion de la convergence des répliques, dénommée cohérence mutuelle. Ce faisant, une répartition de la base de données dans un environnement à large échelle, caractérisé par un nombre important de noeuds (clients et serveurs de données) et une volatilité avérée du système, requiert un middleware redondant par duplication des instances. La première raison de ce choix est que la charge applicative interceptée peut être répartie sur les différentes instances du middleware. La seconde raison découle du fait que les sources de congestion sont moins fréquentes et le passage à l'échelle peut être donc obtenu plus facilement. Une

dernière raison et non la moindre est que la duplication confère une disponibilité du middleware ou une tolérance aux pannes beaucoup plus importante.

Il est à noter aussi que le modèle de communication utilisé pour l'interaction entre les composants et le middleware doit être le moins contraignant possible. De ce fait, un mode de communication asynchrone est beaucoup plus adapté car dans les systèmes distribués tels que les réseaux P2P, le caractère hétérogène des ressources et les connexions/déconnexions fréquentes des noeuds rendent impossible toute tentative de borner la transmission des messages.

Le chapitre prochain décrit en profondeur l'architecture que nous avons proposée pour le traitement des transactions à large échelle.

Chapitre 4

Architecture d'un Système de Routage des Transactions

Le traitement des transactions dans une base de données répartie est fortement lié au modèle de réplication utilisé. Dans le chapitre précédent, nous avons argumenté notre choix d'utiliser un modèle de réplication multi-maître asynchrone. Avec ce modèle, la base de donnée est répliquée sur plusieurs nœuds et une transaction peut être exécutée sur chaque nœud, appelé nœud initial pour cette transaction. Les mises à jour de la transaction sont envoyées vers les autres répliques après validation. Le principal problème avec la réplication multi-maître asynchrone est d'assurer la cohérence mutuelle des répliques même en présence de transactions concurrentes.

Nous proposons dans ce chapitre une architecture pour gérer les transactions exécutées dans une base de données répliquée destinée aux applications Web 2.0. L'architecture doit être structurée de telle sorte que la disponibilité, la transparence et le passage à l'échelle soient garantis. Notre architecture peut être divisée en deux parties : une partie assurant le service de médiation entre les différents composants du système (intergiciel) et une autre chargée de la gestion des métadonnées qui sont les données nécessaires au fonctionnement du système en entier. Avant de décrire l'architecture de notre système nous présentons d'abord quelques définitions et concepts indispensable à la compréhension de notre approche.

4.1 Modèle et concepts

Dans cette section, nous définissons les généralités et concepts sur les quelles sont définis notre approche. Nous décrivons d'abord les concepts relatifs au modèle de réplication et de transactions. Puis, nous présentons les principes généraux sur l'ordonnancement des transactions.

4.1.1 Modèle de transactions et de données

Nous considérons une base de données unique partiellement répliquée sur m nœuds de données ND_1, ..., ND_m. Les données sont partitionnées dans n relations $R^1, ..., R^n$ et une copie locale de R^i sur un nœud ND_j est notée par R^i_j et gérée par le SGBD local. Nous supposons que le

partitionnement des données est fait de tel sorte qu'une transaction peut être exécutée entièrement sur un seul nœud, les transactions réparties sont exclues de cette étude. Nous utilisons un modèle de réplication asynchrone multi-maître. Chaque ND peut être mis à jour par une transaction entrante et est appelé ensuite le nœud initial de cette transaction. Les autres ND sont mis à jour plus tard par propagation de la transaction. Nous distinguons trois types de transactions :

Définition 1. *Transaction de mise à jour*
Une transaction de mise à jour est une séquence d'instructions lecture/écriture dont au moins une d'entre elles modifie la base de données ;

Définition 2. *Transaction de rafraîchissement*
Une transaction de rafraîchissement est utilisée pour propager les transactions de mise à jour sur les autres ND. En d'autres mots, elle ré-exécute une transaction de mise à jour ou applique les modifications faites par cette dernière sur un ND autre que le ND initial.

Pour distinguer les transactions de rafraîchissement des transactions de mise à jour, on mémorise dans le catalogue réparti, pour chaque ND, les transactions déjà routées sur ce ND ;

Définition 3. *Requête*
Une requête effectue une lecture sans mettre à jour de la base de données. Ainsi, il n'est pas nécessaire de la propager.

Par ailleurs, chaque transaction (mise à jour, rafraîchissement, requête) lit un certain nombre de relations. Nous dissocions les données supposées être modifiées par une transaction de celles réellement modifiées.

Définition 4. *Relations potentiellement accédées par une transaction*
Nous définissons par $Rel(T)$, les relations qu'une transaction T planifie d'accéder durant son exécution. Nous avons $Rel(T) = \{Rel_R(T), Rel_W(T)\}$, avec $Rel_R(T)$ (resp. $Rel_W(T)$) les relations que T a l'intention de lire (resp. modifier).

Notons que $Rel(T)$ peut être obtenu en parsant le code.

Définition 5. *DataSet*
L'ensemble des tuples qu'une transaction T a réellement accédé durant son exécution est appelé $DataSet(T)$. Ainsi, nous avons $DataSet(T) = \{ReadSet(T), WriteSet(T)\}$,
avec $ReadSet(T)$ (resp. $WriteSet(T)$) l'ensemble des tuples que la transaction T a réellement lu (resp. modifié).

$Rel(T)$ est connu au moment où T est soumise pour routage et est obtenu par analyse de code de la transaction.

Les requêtes peuvent accéder à des données obsolètes dont l'obsolescence est contrôlée par les applications. Cette obsolescence introduite permet d'accroître le débit et le temps de réponse des requêtes et elle peut être mesurée de différentes manières (*i.e.* mesure booléenne, mesure numérique, mesure de version, mesure temporelle, etc) [LGV04, Pap05]. Dans cette thèse, nous mesurons l'obsolescence en utilisant le nombre de transactions de mise à jour manquantes.

Définition 6. *Obsolescence*
L'obsolescence de R_j^i est égale au nombre de transactions de mise à jour modifiant R^i sur un ND quelconque mais non encore propagée sur le nœud ND_j.

Le concept d'obsolescence est associé souvent au concept de fraîcheur. La fraîcheur d'une réplique *ND*$_j$ correspond à l'opposé de son obsolescence. La fraîcheur est donc maximale (ou parfaite) si l'obsolescence vaut zéro, autrement dit, *ND*$_j$ est totalement frais par rapport à R_i s'il a reçu toutes les mises à jour faites sur R_i.

L'obsolescence tolérée d'une requête est donc, pour chaque relation lue par la requête, le nombre de transactions qui ne sont pas encore exécutées sur le nœud traitant la requête. L'obsolescence tolérée reflète le niveau de fraîcheur requis par une requête pour s'exécuter sur un ND. Pour des raisons de cohérence, les transactions de mise à jour ainsi que celles de rafraîchissement doivent lire des données parfaitement fraîches : elles sont exécutées toujours sur des nœuds totalement frais. Notons alors que nous garantissons la cohérence à terme qui peut être définie comme suit :

Définition 7. *Cohérence à terme*
La cohérence à terme permet que les différentes répliques d'une base de données peuvent diverger pendant une période mais convergent à terme vers un même état en l'absence de nouvelles transactions entrantes.

Ainsi, une séquence de transactions de lectures les différentes répliques ne retourne pas nécessairement la version la plus à jour.

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

Bien que nous utilisons le modèle de données relationnelle pour décrire notre approche, nous mentionnons que notre solution est aussi applicable pour les autres modèles de données sur lesquelles les opérations d'écriture ou de lecture sont faites à travers un programme.

4.1.2 Ordre de précédence des transactions

Nous nous plaçons dans un contexte de transactions plates, sans sous-transactions imbriquées, et non distribuées. Une transaction peut être traitée en totalité sur un seul nœud. Soit T, une transaction de mise à jour ou de lecture seule (requête), nous distinguons les états dans lesquels T peut se trouver :

- ENTRANTE : T vient d'arriver dans le système mais n'a commencé son exécution sur aucun nœud ND. Une date de début, $debut(T)$ est affectée à T par le GT qui l'a reçu et on suppose que chaque transaction arrive à une date différente. Chaque transaction a son identifiant composé du numéro du client (NA) qui l'a envoyé et d'un numéro de séquence local maintenu au niveau du client. Une transaction dans cet état est dite transaction entrante.
- COURANTE : T a commencé son exécution sur un ND. Ses effets éventuels ne sont pas encore persistants. Toute transaction dans cet état est appelée transaction courante.

- VALIDÉE : T est exécutée et validée au moins sur le nœud ND_i. Ses effets sont visibles sur ND_i et peuvent l'être aussi sur les nœuds ND restants en fonction des exigences des transactions entrantes. Autrement, si T' exige des données totalement fraîches, alors quelque soit le ND, les effets de T seront visibles. La date à laquelle une T est validée est appelée $fin(T)$.
- TERMINÉE : T passe à l'état global si elle est propagée sur tous les ND du système. Ses effets sont visibles sur n'importe quel nœud.

Une transaction validée ou terminée ne peut être défaite et ses effets persistent durablement.

Le processus de routage définit l'ordre dans lequel les transactions entrantes doivent être exécutées sur les différentes répliques pour garder le système cohérent. Avec la réplication asynchrone multi-maître, la cohérence mutuelle de la base de données peut être compromise par l'exécution simultanée des transactions conflictuelles sur différents sites. Pour éviter ce problème, les transactions de mise à jour sont exécutées sur les nœuds de la base dans un ordre compatible, produisant ainsi des états cohérents de toutes les répliques de la base de données (cohérence à terme des données). Les requêtes sont routées sur n'importe quel nœud, suffisamment frais vis-à-vis des conditions requises par la requête. Ceci implique qu'une requête peut lire différents états de la base de données en fonction du nœud sur lequel elle est exécutée. Néanmoins, les requêtes lisent toujours des états cohérents (probablement obsolètes) car elles ne sont pas distribuées. Pour assurer la cohérence globale, nous maintenons un graphe dans le catalogue, appelé graphe de sérialisation global (GSG).

Définition 8. *Graphe de sérialisation globale*
Un GSG <$T, \rightarrow$ > est un graphe au sens mathématique où un sommet est une transaction (T) et un arc ($\rightarrow$), une contrainte de précédence entre deux transactions. Il est orienté et sans circuit et garde la trace des dépendances conflictuelles entre les transactions actives i.e. *les transactions courantes mais non encore validées et les transactions validées mais pas globales.*

Le GSG est construit au départ en se basant sur la notion de conflit potentiel puis peut être raffiné grâce aux conflits réels.

Définition 9. *Conflit potentiel*
Une transaction entrante T_e est en conflit potentiel avec une transaction courante ou validée T_c si elles manipulent une même relation et que l'une des transactions effectue au moins une écriture sur cette relation, autrement dit, $Rel_R(T_e) \cap Rel_W(T_c) \neq \emptyset \vee Rel_W(T_e) \cap Rel_R(T_c) \neq \emptyset \vee Rel_W(T_e) \cap Rel_W(T_c) \neq \emptyset$

Une contrainte de précédence est un ordre pré-établi sur les transactions conflictuelles et est défini suivant l'arrivée des transactions.

Définition 10. *Ordre de précédence*
Soient deux transactions conflictuelles T et T', on dit que T précède T' si $debut(T) < debut(T')$ et nous notons $T \rightarrow T'$.

A partir des deux dernières définition, nous notons que le GSG est un graphe qui contient l'ensemble des transactions conflictuelles ordonnées suivant leur date d'arrivée. En raison des relations

de transitivité entre les contraintes de précédence, nous ne maintenons que la réduction transitive du GSG : un arc de $T1$ à $T2$ n'est pas ajouté si $T1$ précède déjà $T2$ indirectement. La réduction transitive étant unique du fait que le graphe est orienté et acyclique, alors garder la réduction transitive suffit pour garder un ordre unique de l'exécution des transactions. En outre, à cause de la réplication, le GSG contient aussi les transactions déjà validées mais non encore propagées sur l'ensemble des répliques. Ceci est nécessaire pour ordonner les transactions de rafraîchissement. L'ordonnancement des transactions de rafraîchissement peut être basée sur la notion de conflit réel.

Définition 11. *Conflit réel*
Deux transactions T et T' sont en conflit réel si T a lu ou modifié ce que T' a modifié, autrement dit, $WriteSet(T) \cap ReadSet(T') \neq \emptyset \vee WriteSet(T) \cap WriteSet(T') \neq \emptyset$.
Corollairement, deux transactions T et T' en conflit potentiel mais non en conflit réel sont dites transactions sans précédence et nous notons par $T \not\rightarrow T'$. Autrement dit, T et T' peuvent être exécutées dans n'importe quel ordre sur n'importe quelle réplique et le GSG est raffiné.

Notons que nous vérifions les conflits réels entre deux transactions que lorsqu'elles sont en conflit potentiel. Les détails de la gestion du répertoire réparti et en particulier du GSG sont décrits dans la section 5.2.

4.1.3 Structuration des métadonnées

Puisque les métadonnées contiennent plusieurs types d'informations, une structuration logique des informations permet d' accélérer les recherches au sein du catalogue.

Comme nous l'avons mentionné précédemment, notre base de données est fragmentée en plusieurs relations. Toutes les métadonnées relatives à une relation R^i sont regroupées dans une structure appelée *Meta(R^i)*. Cette structure contient les informations nécessaires à un GT pour récupérer toutes les contraintes de précédence relatives à la relation R^i et l'état courant d'une réplique R^i_j. Les informations que l'on retrouve dans cette structure sont :

- $GSG(R^i)$, la partie du GSG relative à la relation R^i ;
- L'historique de toutes les transactions qui ont déjà modifié la relation R^i mais qui ne sont pas encore propagées vers toutes les répliques. Celle-ci correspond à un sous ensemble du $GSG(R^i)$ réduit aux transactions déjà validées. L'historique, dénommée *History(R^i)* mentionne également sur quel site une transaction a été exécutée une première fois et est représentée sous forme de graphe où un sommet est un couple (T_i, DN_j) et un arc représente une contrainte de précédence. *History(R^i)* permet de restaurer l'état cohérent du système même en cas de panne du nœud qui a reçu les dernières mises à jour.
- *State(R^i_j)*, l'état local de chaque réplique N_j : c'est la liste des transactions qui ont accédé à la relation R^i et qui ont déjà validé leur modification. Cette information est indispensable pour mesurer l'obsolescence de chaque réplique et la séquence de transactions à lui envoyer pour qu'elle soit totalement fraîche.

4.2 Définition générale des composants de l'architecture

Cette section présente l'architecture du système de routage (appelé routeur par la suite). Le système est constitué d'un ensemble de composants qui jouent des rôles spécifiques.

- Application : nous appelons application le composant générant la charge applicative. Les applications émettent des demandes de transactions qui sont envoyées au gestionnaire de transactions.
- Gestionnaire de transactions : C'est l'intergiciel transactionnel au cœur du système de routage. Il reçoit les demandes de transactions et les transmet à une base de données, de manière optimale. Il ordonne les transactions pour conserver la cohérence des bases de données sous-jacentes.
- Catalogue. Le catalogue contient toutes les informations nécessaires au routage des transactions. Il informe le gestionnaire de transaction sur l'état des différents composants du système.
- Base de données : nous appelons base de données le composant représentant la base de données. Le composant base de données reçoit et gère l'exécution des transactions que le gestionnaire de transactions lui envoie, puis transmet le résultat à l'application ayant émis la transaction.

La figure 4.1 décrit l'assemblage des 4 composants. Les deux composants gestionnaires de transaction et catalogue forment l'intergiciel de routage. Les deux autres composants, application et base de données, situés aux extrémités de l'architecture, agissent comme des relais vis-à-vis de l'environnement extérieur.

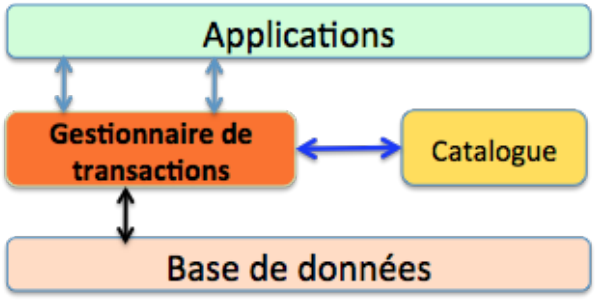

FIGURE 4.1 – Architecture globale en couche

De plus, cette architecture modulaire présente l'avantage de faciliter l'évolution et la maintenance du système. En effet, il est possible d'améliorer l'implémentation de chaque module individuellement, tout en conservant intacts les autres modules.

4.2.1 Impact des besoins applicatifs sur l'architecture

La conception d'une architecture modulaire se justifie pour mieux répondre aux principaux besoins du routeur : préserver l'autonomie des applications, passer à l'échelle et être disponible.

En effet, pour chaque composant, nous définissons son interface avec ses composants connexes. Puis, l'étude détaillée de chaque composant permet de définir sa mise en œuvre dans un contexte réparti.

Besoin de préserver l'autonomie des applications et des SGBD

Le routeur est conçu pour améliorer des applications existantes. Pour cela, le routeur doit pouvoir s'insérer dans une application existante. Cela est possible car les applications visées sont déjà conçues de façon modulaire avec deux couches bien distinctes : la couche au niveau du serveur d'application gère le dialogue avec l'utilisateur, la couche sous-jacente contient les services de gestion de données. Les deux couches communiquent à travers des interfaces standards (e.g. JDBC ou REST), ce qui permet l'insertion du routeur entre ces deux couches. Par exemple, il est possible d'insérer le routeur dans une application qui utilise JDBC pour se connecter aux serveurs de données. Dans ce cas, le routeur se comportera comme un pilote JDBC vis-à-vis de l'application et comme un client JDBC vis-à-vis du serveur de données. Donc, du point de vue de l'application, le routeur se substitue au pilote d'accès à la base, de manière transparente, sans impliquer la modification de l'application existante ni celle du SGBD.

Besoin de passage à l'échelle et de disponibilité

Le routeur est conçu pour fonctionner à grande échelle en s'appuyant sur une infrastructure distribuée constituée d'un grand nombre de ressources (ou nœuds) de traitement. Le routeur doit aussi être hautement disponible. Ces deux exigences nécessitent de répartir et répliquer les composants de l'architecture. Ainsi, chaque composant est mis en œuvre par un ensemble (potentiellement grand) de nœuds. Chaque nœud étant un processus, on peut avoir plusieurs nœud sur une même machine physique et les noeuds doivent se coordonner entre eux, si nécessaire. Cependant, tous les composants ne demandent pas le même niveau de coordination. C'est pourquoi, nous distinguons plusieurs organisations des nœuds, plus ou moins structurées, selon le composant à mettre en œuvre (voir la figure 4.2) :

- Application : indépendance totale entre les différentes instances du composant application. Il n'y a pas d'échange d'informations entre les applications, donc pas de liens entre applications.
- Gestion des transactions : structuration forte des nœuds représentant le routeur, organisation en anneau car leur nombre est relativement faible, et il est nécessaire d'échanger très fréquemment des informations entre les différentes instances du composant de gestion de transactions puisqu'elles peuvent gérer des transactions touchant les mêmes données.
- Catalogue : structuration forte en anneau. Les informations contenues dans le catalogue sont répliquées pour éviter leur perte. De plus, les informations sont réparties sur plusieurs nœuds pour paralléliser les accès disjoints, raison pour laquelle nous l'appelons aussi catalogue réparti.
- Base de données : structuration faible car leur nombre est très élevé et il n'est pas nécessaire pour une instance de base de données de connaître toutes les autres bases de données (c'est

le rôle du catalogue et pas celui des bases de données). Toutefois, certains échanges d'informations directs entre les bases de données sont nécessaires pour valider des transactions.

Pour des soucis de présentation, nous utilisons par la suite le terme nœud applicatif (NA) pour désigner une instance du composant "Application", gestionnaire de transaction (GT) pour une instance du "Gestionnaire de transaction", nœud catalogue (NC) pour une instance du "Catalogue" et enfin nœud de données (ND) pour une instance du composant "Base de données".

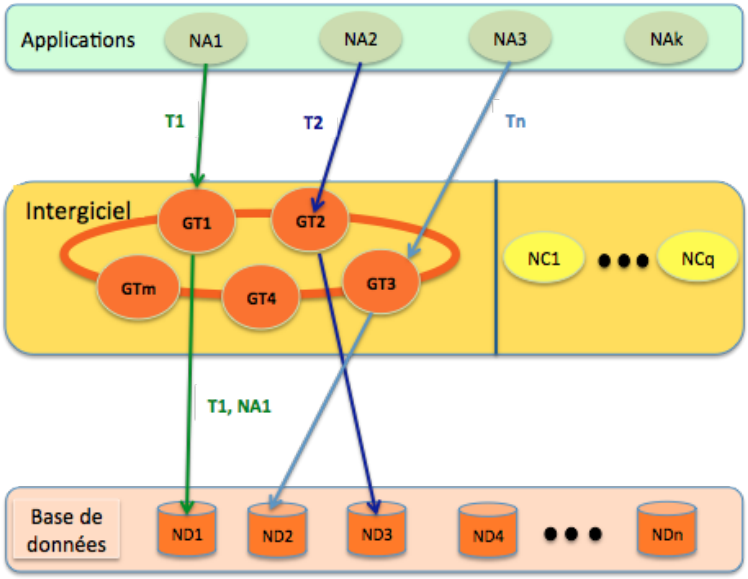

FIGURE 4.2 – Architecture détaillée globale du système

4.2.2 Modèle de communication

Pour garantir la communication des différents nœuds logiques qui composent notre système (GT, NC, NA et ND), nous nous appuyons sur les primitives de communication des systèmes P2P. Sur la figure 4.3, la couche haute concentre les différents noeuds de notre système et la couche basse un réseau logique de P2P (ou d'un système de grille) construit sur des machines physique connectées par Internet ou un réseau WAN. La couche basse fournit des primitives de connexions et déconnexions au système et des services tels que la localisation d'un nœud, le support d'identification unique des nœuds et de la communication asynchrone entre nœuds. Un nœud logique de la couche basse peut regrouper plusieurs nœuds logiques de la couche haute. Ce faisant, un noeud GT peut communiquer avec un noeud ND en passant par le réseau logique P2P sous-jacent. Ce choix d'implémentation nous épargne la tâche de développer des protocoles de communication via des sockets ou autres. Il nous permet aussi de pouvoir intégrer notre solution sur n'importe quel système offrant des services de communication et de localisation des ressources.

L'ensemble des nœuds de notre système communiquent par des messages. Chaque message envoyé, doit être acquitté par le destinataire. Notre modèle de communication est fortement lié au

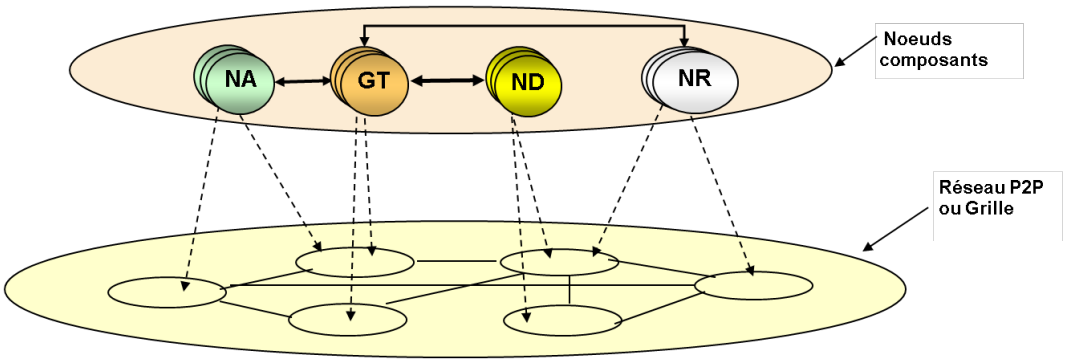

FIGURE 4.3 – Communication via un système P2P ou grille

protocole de routage décrit dans le chapitre prochain et qui permet à un ND d'envoyer directement les résultats au noeuds NA. En fait, pour exécuter une transaction T, un NA contacte un GT qui à son tour va contacter un ND. Le ND envoie directement les résultats aux NA et un message de notification au GT. Ce message de notification permet au GT de marquer T comme exécutée au moins sur un nœud du système. Pour faciliter la collaboration des nœuds, des informations sont

FIGURE 4.4 – Format des messages

ajoutées à chaque message envoyé. Les messages envoyés ont le format décrit dans la figure 4.4 et sont composés de quatre champs. Le premier champ définit le type de message, le deuxième contient l'identifiant de l'émetteur du message, le troisième garde le contenu du message alors que le quatrième champ est optionnel et son contenu est fortement tributaire du type du message. Les types de messages sont au nombre de dix dont les plus utilisés sont : les demandes de traitement de transactions (entre NA, GT et ND), les messages de notification de fin d'exécution (ND et GT), les messages d'acquittement, les messages véhiculant les résultats et les messages de détection de pannes. Le type d'un message est appelé aussi tag. Pour envoyer un message à un nœud (respectivement recevoir un message d'un nœud), un nœud utilise la primitive *sendMsg()* (respectivement *getMsg()*).

4.2.3 Architecture détaillée

Cette section présente l'architecture interne de chaque composant. Elle définit aussi la coordination de plusieurs instances d'un même composant.

Nœud Applicatif (NA)

Interfaces. Chaque NA est constitué de l'instance de l'application et de quelques modules ajoutés pour des besoins spécifiques du routeur (voir figure 4.5). Ces modules additionnels jouent alors le rôle d'interface entre l'application et le gestionnaire de transaction et nous les appelons interface cliente. Les transactions envoyées par l'application sont encapsulées par l'interface cliente dans des primitives sous forme de messages et sont envoyées au GT. En effet l'application envoie sa transaction via une interface standard JDBC et n'a pas connaissance de l'existence des GT. La transaction est interceptée par l'interface cliente qui l'inclut dans un message avec des informations destinées aux GT. L'interface cliente qui a une connaissance de l'adresse de certains GT en choisit un et l'envoie le message pour traiter la transaction qui y est incluse. A titre illustratif, voici un exemple de requête encapsulée dans un message à destination d'un GT.

[msgToRoute : 123 : Update table where attribut = val ; : <123,15>]

Le premier champ contient le tag "MsgToRoute" qui signifie que c'est une transaction à traiter. Le second champ permet d'identifier l'identifiant de l'émetteur du message (123). Le champs "Contenu" contient le code de la transaction à exécuter alors que le champs "Options" stocke l'identifiant de la transaction (<123,15>). De même les réponses aux transactions sont décapsulées par l'interface cliente avant de transmettre le résultat à l'application.

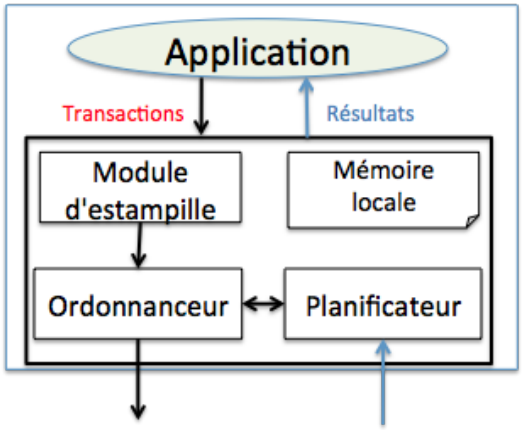

FIGURE 4.5 – Structure interne d'un NA

Rôle des composants internes. Un NA associe à chaque transaction un identifiant global unique (GId) qui est nécessaire pour éviter toute ambigüité durant son exécution. Le GId d'une transaction est la paire <Id, SeqN>, où Id représente l'identifiant du client (adresse IP) et SeqN le numéro de séquence local de la transaction. Il correspond à <123,15> pour l'exemple précédent. Le GId est généré par un module spécial appelé *Module d'estampille* qui doit assurer son unicité et sa monotonie. Les transactions sont envoyées par le module *Ordonnanceur*. Ce dernier a une connaissance

partielle des différents GT existants et les choisit suivant l'algorithme du tourniquet. Toute transaction T envoyée est journalisée dans une structure appelée *Mémoire locale (ML)* au niveau du NA jusqu'à la réception des résultats de son exécution. Le *ML* stocke aussi tous les TMs connus par le nœud applicatif tout en pointant à chaque instant le dernier TM qui a été contacté. Dés réception des résultats de l'exécution d'une transaction, le module *Planificateur* efface du *ML* l'entrée correspondante à T. En cas de non réception des résultats de T au bout d'un délai fixé, le Planificateur charge le module Ordonnanceur de renvoyer la transaction avec le même GId. L'objectif de réémettre une transaction avec un même GId est d'une part, d'éviter qu'elle soit exécutée plusieurs fois et d'autre part, qu'elle soit prise en compte si elle ne l'est pas encore. Les détails de cette approche seront donnés dans les prochains chapitres. Les nœuds NA n'ont aucune connaissance de la localisation et encore moins de l'état des nœuds de la couche de données, ce qui permet d'assurer une transparence totale de la distribution des données.

Pour se connecter au système, un nouveau NA a besoin de connaître un GT qui lui renvoie par la suite les identifiants de ses voisins.

Gestionnaire de transaction(GT)

Interfaces. Les nœuds gestionnaire de transactions (GT) transmettent les transactions pour exécution sur les nœuds de données (ND) tout en préservant la cohérence globale. Les GT utilisent les métadonnées stockées sur le catalogue réparti pour choisir le nœud vers lequel envoyer une transaction. C'est pourquoi, le GT a deux interfaces : une pour accéder au catalogue et une autre pour dialoguer avec les nœuds de données. L'interface utilisée pour communiquer avec les ND est similaire à celle utilisée pour interagir avec un NA. Le message ci-après correspond au message envoyé par un GT pour demander l'exécution d'une transaction sur un ND.

[MsgToPerform : 005 : Update table where attribut = val ; : <123,15> ; *Ordre*]

Le champs "Options", contient cette fois des informations pour garder la cohérence : *Ordre* contient les transactions qui doivent précéder la transaction entrante identifiée par <123 ,15> (voir section 5.1). L'interface utilisée pour accéder aux métadonnées utilise des primitives offertes par des systèmes tiers qui sont utilisés pour stocker les métadonnées (voir section 4.3).

Rôle des composants interne. Les GT sont organisés sous forme d'anneau logique [LAF99]. Cette structuration a pour but de faciliter leur collaboration et leur communication afin de maintenir les performances du système (disponibilité, cohérence, ...). Pour ce faire, chaque GT est relié à k prédécesseurs et k successeurs. L'ensemble des successeurs de GT_i est obtenu par la formule $Suc(GT_i) = \{GT_{((i+j) \ mod \ n)}, 1 \leq j \leq k\}$, avec n le nombre total de GT. Chaque GT est composé de quatre modules (voir figure 4.6) :

- Le *Module de routage (MR)* est chargé de réceptionner les transactions provenant des NA et de choisir un nœud ND pour l'exécution de celle-ci. Le choix du nœud qui traitera la transaction se fait suivant des algorithmes que nous présenterons dans le prochain chapitre. La plupart de ses algorithmes sont basés sur un modèle de coût qui permet de choisir un

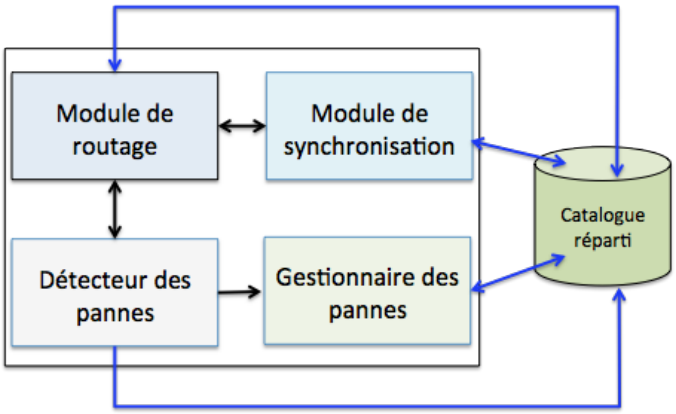

FIGURE 4.6 – Structure interne d'un GT

nœud parmi plusieurs candidats potentiels. Ainsi lors du processus de routage, le *MR* fait appel à la fonction de coût, qui permet de retrouver le nœud optimal.

- Le *Module de synchronisation (MS)* est le module qui permet la convergence des copies vers un même état. Concrètement, il initialise la propagation des modifications faites sur un nœud ND vers d'autres nœuds ND. Cette propagation se fait soit à la demande d'un GT ou d'un ND, soit périodiquement. Pour déterminer les mises à jour à propager, certaines informations stockées dans le catalogue sont utilisées à savoir la dernière transaction exécutée sur un nœud, la transaction la plus récente, etc.
- Le *Module de détection des pannes (MDP)* permet de prendre en compte la volatilité des ressources du système, particulièrement les nœuds ND et GT. En effet, il est responsable de la détection des pannes et de leur notification auprès des GT pour les pannes de nœuds ND. Quand un nœud GT tombe en panne, ces prédécesseurs en seront informés pour maintenir l'anneau à jour. Chaque nœud ND en panne est inscrit dans une file appelée file des nœuds en panne (*FNP*). Cette file est stockée sur le catalogue et est utilisée lors des processus de routage pour éviter qu'un nœud en panne soit choisi par un GT. Ceci augmente la chance d'envoyer une transaction sur un nœud ND non défaillant et donc les chances d'obtenir une réponse rapidement.
- Le *Gestionnaire des pannes (GP)* est chargé de détecter le retour de tout nœud précédemment déclaré comme étant en panne par le *MDP*. S'il s'agit d'un ND, il l'enlève de la file (*FNP*) et dans le cas d'un GT, il initialise la réorganisation de l'anneau.

Les GT sont sans état et toutes les informations dont ils ont besoin sont toujours stockées dans le catalogue réparti. Ainsi, un GT défaillant peut être remplacé par n'importe quel autre GT sans perdre des informations. Un GT a besoin de connaître au moins un autre GT pour s'insérer dans l'anneau et recevoir les informations nécessaire à l'accès au catalogue.

Nœud de données (ND)

Interfaces. Les nœuds ND utilisent un système de gestion de base de données (SGBD) local pour stocker les données et exécuter les transactions envoyées par les GT. Après l'exécution d'une transaction, les résultats sont directement envoyés au NA à l'origine de la transaction, sans repasser par les GT. Ce mécanisme permet de s'éloigner du fonctionnement Client/Serveur. Par conséquent, le ND a une interface pour communiquer avec le GT et le ND et une autre interface pour communiquer avec le SGBD local. La première interface est identique à celle utilisée par le GT pour communiquer avec le ND et consiste donc à un message encapsulant des informations de routage. Un exemple d'un message envoyé via cette interface est :

[MsgToNotify : 025 : Positive EOT : <123,15>]

Le champ "Options" contient le GId de la transaction que le ND vient d'exécuter. Le champs "Contenu" indique l'état de l'exécution de la transaction. On rappelle que la fin d'une transaction (EOT) peut être positive si l'exécution a réussi et négative dans le cas contraire.

Par contre pour communiquer avec le SGBD local, le ND utilise l'interface standard JDBC pour envoyer la transaction et recevoir les résultats.

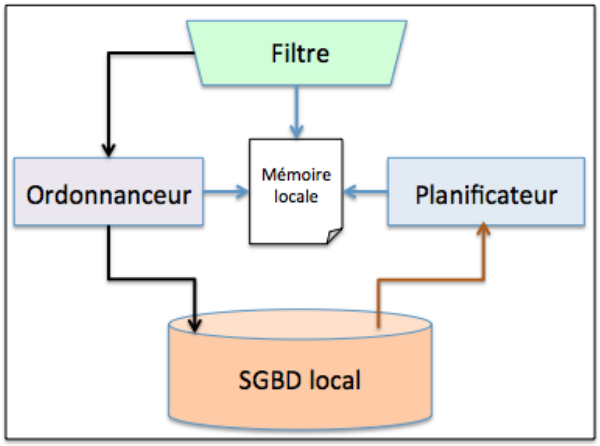

FIGURE 4.7 – Structure d'un ND

Rôle des composants internes. Des informations de routage (Identifiant émetteur, GId, ...) sont ajoutées à chaque transaction dans l'optique de garder une traces des nœuds qui ont participé à l'exécution de la transaction. Pour dissocier le code de la transaction à exécuter des informations de routage, un nœud ND utilise un module dénommé *Filtre*. Ainsi, les instructions de la transaction à traiter sont transmises au SGBD local via un pilote JDBC par le Filtre et les informations de routage sont gardées jusqu'à la fin de l'exécution pour informer les GT de la bonne terminaison de celle-ci mais aussi pour renvoyer les résultats directement au NA à l'origine de la transaction. Par ailleurs, nous mentionnons qu'avec les systèmes à large échelle, que deux messages envoyés avec un ordre

peuvent être reçus dans un ordre différent et par conséquent, engendrer quelques problèmes de cohérence. Pour prendre en compte ce genre de problème, nous avons un module *Ordonnanceur* qui est chargé de transmettre la transaction au SGBD tout en garantissant que l'ordre de précédence soit respecté (les détails de cette garantie seront donnés dans le chapitre suivant). Ce module permet également à un nœud ND de ne pas exécuter une même transaction deux fois de suite pendant une période supérieure à la période de synchronisation totale des répliques. Les informations de routage collectées par le Filtre sont stockées dans une zone tampon sur le ND local. Enfin, un dernier module *Planificateur* est chargé de l'envoi des résultats aux clients (NA) et les notifications de fin de traitement des transactions aux GT.

Contrairement aux nœuds GT, un nœud ND ne connaît pas les autres NDs même si ces derniers stockent la même copie que lui. Toutefois, si un nœud ND a besoin de collaborer avec un autre ND pour l'exécution d'une transaction T, les informations nécessaires pour cela (identifiant, et adresse IP) sont fournies lors du routage de T. Cette absence de connaissance globale du système permet à un ND de ne pas avoir besoin de faire des mises à jour lors des connexions ou déconnexions d'autres NDs.

4.3 Description de la structure des métadonnées

Dans cette section nous présentons la structure de métadonnées. Nous décrivons d'abord le contenu des métadonnées avant de détailler leur implémentation.

4.3.1 Description et structure des métadonnées

Cette partie décrit les informations qui sont stockées dans le catalogue réparti et comment elles sont structurées.

Besoin de garder des informations multiples

Les métadonnées sont les informations relatives à la distribution des données et des transactions exécutées, autrement dit, les métadonnées contiennent les informations sur les ND. Elles sont stockées dans le catalogue réparti à travers des nœuds appelés nœuds catalogue (NC). L'objectif de garder des métadonnées est de faciliter la recherche des ressources du système et particulièrement l'état des bases de données afin de router efficacement une transaction. Plus les informations utilisées pour décrire les ressources du systèmes sont nombreuses plus la recherche d'une ressource satisfaisant un certain nombre de critère est rapide, précise et fructueuse. Ainsi, pour chaque relation R^i de la base de données, nous gardons plusieurs informations : (1) les identifiants de l'ensemble des ND qui stockent une copie de R^i ; (2) pour chaque ND_j stockant R^i, nous gardons l'état courant de la base *i.e.* la liste des transactions courantes et/ou déjà exécutées sur ND_j ; (3) pour chaque ND, nous gardons son statut *i.e.* s'il est connecté ou déconnecté ; etc. Nous mentionnons que nous gardons dans le catalogue toutes les transactions en cours et celles qui sont déjà validées mais non encore propagées sur toutes les répliques. Le choix de garder qu'une partie des transactions déjà

exécutée a pour objectif de minimiser le volume des métadonnées. En effet un volume de métadonnées très important peut entrainer des latences lors des accès . En outre, les transactions déjà exécutées sur un quelconque ND sont propagées périodiquement sur l'ensemble des ND distants et sont immédiatement éffacées du catalogue. Le catalogue réparti stocke aussi, pour chaque transaction T, le temps estimé pour exécuter T, qui est une moyenne variable obtenue par l'exécution des précédentes exécutions de T. Il est initialisé par une valeur par défaut en exécutant T sur un nœud non chargé. Cette mesure est indispensable pour effectuer le routage (voir chapitre 5).

Besoin de fragmenter le catalogue

Comme nous l'avions mentionné ci-avant, les GT ont besoin des métadonnées pour assurer le traitement des transactions envoyées par les noeuds NA et contrôler la cohérence globale. De ce fait, le catalogue devient indispensable et doit être disponible d'autant plus que les GT peuvent simultanément solliciter l'accès aux métadonnées.

Pour garantir la disponibilité des métadonnées et leur utilisation simultanée par plusieurs GT, les métadonnées sont fragmentées et répliquées sur plusieurs sites. La fragmentation augmente les accès disjoints et favorise ainsi les accès parallèles, ce qui améliorer les performances du système, particulièrement le débit du routage. La fragmentation est faite de telle sorte que chaque fragment ne contient que les métadonnées relatives à une relation R^i $(Meta(R^i)$. Ceci facilite la recherche par nom de relation et fournit des accès indépendants de chaque $GSG(R^i)$. Pour trouver le graphe global ($GSG(T)$) relatif à la transaction T, il faut récupérer les $GSG(R^i)$ tel que $R^i \in Rel(T)$ et donc $GSG(T) = \bigcup GSG(R^i) | R^i \in Rel(T)$.

Cependant, la réplication peut entraîner quelques problèmes dans la gestion des métadonnées, notamment leur cohérence. Ainsi, il devient important de gérer les accès concurrents au catalogue de manière efficace afin de garantir le cohérence et de réduire la latence (cf. section 5.2).

4.3.2 Implémentation du catalogue

Pour implémenter les nœuds NC, nous avons utilisé des systèmes tiers qui fournissent des services de gestion et de stockage de données dans un environnement à grande échelle. En premier lieu, nous utilisons JuxMem [ABJ05] qui fournit un service transparent et cohérent de partage de données sur une grille informatique. En second lieu, nous utilisons une DHT pour un meilleur passage à l'échelle et une disponibilité plus importante du catalogue.

Architecture du catalogue avec un système à mémoire partagée : JuxMem

JuxMem fournit un service transparent et cohérent de partage de données sur une grille informatique. Il permet le partage d'un ensemble de blocs de données dans un système à mémoire partagé. Ainsi, il offre des primitives d'écriture et de lecture de données stockées sur ces blocs tout en garantissant leur cohérence et leur disponibilité. Par conséquent, pour stocker les informations du catalogue, le problème consiste à travers JuxMem de demander des blocs de mémoire de blocs nécessaire sans pour autant se soucier des endroits où le stockage physique se fera proprement fait. Cependant, comme les blocs de JuxMem sont de taille limitée, il faut éviter que les informations

d'un fragment de métadonnées se trouvent dans deux blocs, qui peuvent être éloignés et donc ralentir l'accès. C'est une motivation de plus à notre choix de ne pas garder dans le catalogue les transactions déjà propagées afin de réduire la taille des métadonnées.

Interfaces. JuxMem offre à travers une interface plusieurs primitives pour manipuler les données parmi lesquelles nous pouvons citer :

- *alloc (size, attributes)*, une primitive qui permet d'obtenir un nouveau bloc d'une taille donnée (*size*) avec un degré de redondance et un protocole de contrôle de cohérence mutuelle spécifiés par le paramètre *attributes*.
- *put (id, value)* permet de modifier la valeur d'une donnée d'un bloc identifié par *id*.
- *get(id)* permet de récupérer la valeur du bloc identifié par *id*.
- *lock(id)* et *unlock(id)* respectivement pour verrouiller et déverrouiller le bloc d'identifiant *id*.

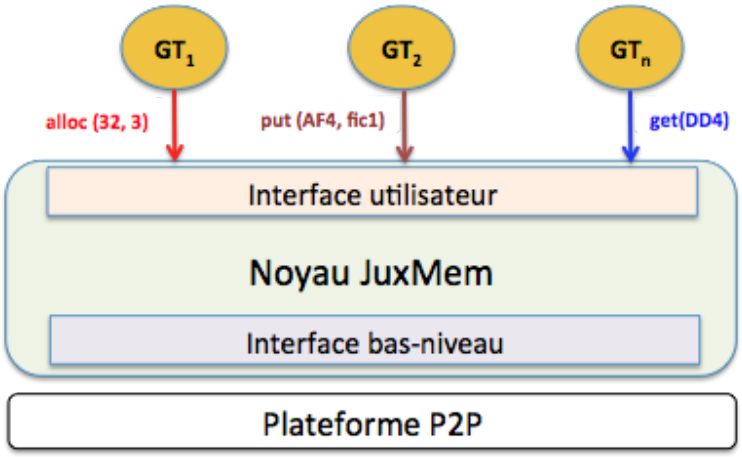

FIGURE 4.8 – Méthodes d'accès au catalogue avec JuxMem

Pour manipuler les métadonnées, les nœuds GT se comportent comme des clients dans l'architecture de JuxMem (voir figure 4.8). Par conséquent, un GT demande à JuxMem un nouveau bloc lors de la création d'un fragment de métadonnées, puis utilise la primitive *put (id, value)* pour insérer la structure *Meta(R)*. Pour lire le contenu du *Meta(R)*, il utilise la primitive *get(id)*.

Accès concurrents avec JuxMem. La lecture et la modification des blocs avec Juxmem se fait via l'utilisation de verrous. Pour assurer la cohérence lors des routages, nous avons utilisé les primitives de verrous de JuxMem pour reproduire le schéma classique de verrouillage à deux phases (2PL). En d'autres termes, un GT garde un verrou sur le bloc de données requis durant tout le processus de routage. Ceci ne dégrade pas les performances du système, puisque : (1) le processus de routage est très rapide comparé à l'exécution de la transaction et des opérations de rafraîchissement, et (2) le catalogue est découpé de tel sorte qu'un seul bloc est souvent sollicité par une transaction.

JuxMem permet d'obtenir de bonnes performances si le nombre de GT concurrents est faible. L'utilisation des verrous entraîne la dégradation des performances si le nombre de GT concurrents est important puisqu'il faut attendre toujours le relâchement d'un verrou pour pouvoir traiter (router) une transaction. Dans un environnement à volatilité très importante la panne d'un noeud cause de sérieux problème à l'utilisation des verrous puisqu'un nœud détenant un verrou peut tomber à tout moment en panne. En outre, JuxMem cache les détails de stockage et de réplication des données. Ainsi, il devient impossible de modifier le protocole de réplication pour des besoins spécifiques de gestion du catalogue réparti.

Architecture du catalogue avec une DHT

Pour garantir que le catalogue réparti soit plus disponible et que son utilisation simultanée par plusieurs GT ne constitue pas une source de congestion, nous avons utilisé une DHT qui permet d'avoir des services d'indexation passant à l'échelle. La DHT distribue et réplique les métadonnées sur plusieurs noeuds NC pour assurer la disponibilité. Dans la suite de cette section, nous décrivons comment nos métadonnées sont intégrés dans une DHT.

Interfaces. Pour manipuler les métadonnées dans la DHT, nous avons besoin d'opérations de base pour les insérer, les retrouver et les modifier. Pour ce faire et compte tenu de notre contexte, nous avons d'une part les primitives natives offertes par la plupart des implémentations des DHT et d'autre part des primitives additionnelles développées pour nos propres besoins (cf. figure 4.9).

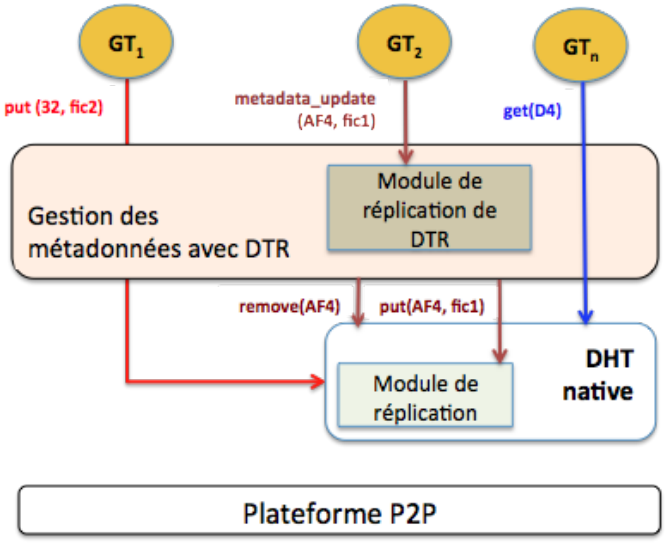

FIGURE 4.9 – Méthodes d'accès au catalogue avec DHT

Méthodes d'accès avec les primitives natives des DHT. En général, une DHT offre deux primitives d'opérations très usuelles : *put(k, v)* pour insérer une valeur *v* associée à une clé *k*, et *get(k)*

pour récupérer la valeur *v* associée à *k*. Dans notre cas, *k* est nom d'une relation R et *v* est la structure *Meta(R)*. En outre, pour tolérer la panne ou la déconnexion des noeuds, la DHT réplique chaque couple (*k, v*) sur plusieurs nœuds. Une opération *put(k, v)* crée n répliques $(k, v_i)0 < i < n$. La valeur de n est initialisée au moment de l'insertion de la valeur et en fonction du niveau de disponibilité sollicité. En plus toute réplique peut être utilisée par la DHT sans distinction. La seule chose à assurer est qu'une opération *get* retourne toujours une des copies de la valeur associée à la clé utilisée. Ainsi, deux opérations *get* concurrentes peuvent trouver des répliques différentes gérées par deux nœuds distincts. La DHT ne prend pas en compte la détection d'opérations *get* concurrentes puisque les DHT sont conçues de tel sorte que les données qui y sont stockées ne soient accessibles qu'en lecture seule. C'est la raison pour laquelle il n'y a que deux situations dans lesquelles, les primitives natives des DHT peuvent être utilisées. Premièrement, quand une GT route une transaction de lecture seule, il utilise la primitive *get(R)* pour obtenir la structure *Meta(R)*. Le GT ne modifie pas *Meta(R)*, il le traverse uniquement dans l'objectif de calculer la séquence de rafraîchissement. Deuxièmement, quand une structure *Meta(R)* vient d'être créée, on utilise la primitive *put* pour l'insérer une première fois dans le catalogue réparti. Tout autre accès nécessite les primitives personnalisées ci dessous.

Personnalisation des méthodes d'accès d'une DHT Il existe deux cas dans lesquels nous avons besoin de primitives autre que celles proposées par une DHT et correspondent à des modifications du contenu du catalogue. En effet, quand un GT route une transaction, il lit le graphe de précédence avec l'intention de le modifier (*get_for_update*), ainsi il a besoin d'être tenu informé des autres accès concurrents pour ne pas faire diverger les copies des métadonnées. A la fin de l'exécution d'une nouvelle transaction, le GT a besoin de marquer sur le catalogue que la transaction a été bien exécutée et pour ce faire il modifie le graphe de précédence et l'état du nœud sur la quelle la transaction est validée. Cette dernière opération est appelée *metadata_update*. L'implémentation des opérations *metadata_update* et *get_for_update* est basée simplement sur les primitives d'accès des DHT. De manière plus précis, nous modifions légèrement le protocole de réplication d'une DHT de tel sorte que tous les noeuds stockant une copie d'une même portion des métadonnées ne jouent pas le même rôle. Les noeuds qui stockent une copie du couple (k, v) sont appelés successeurs de la clé k. Ainsi, le premier successeur (nœud dont l'identifiant est le plus proche de la clé k) est appelé nœud NC maître et est utilisé pour le routage des transactions de mises à jour. Les autres successeurs sont appelés secondaires et sont utilisés pour le routage des transactions de lecture et donc sont accessibles avec les primitives d'opérations natives de la DHT. Le NC maître est utilisé pour mettre à jour les métadonnées et est donc responsable de la synchronisation avec les autres successeurs.

Accès concurrent avec une DHT. Pour gérer l'accès concurrent au catalogue lors du routage, nous ajoutons une entrée dans le catalogue appelée *Last(R)* . *Last(R)* est un pointeur sur le dernier GT qui a accédé à la structure *Meta(R)*. Ceci permet d'ordonner les écritures des GT accédant simultanément la structure et donc de préserver la cohérence. Le choix de garder le dernier GT à accéder à la structure *Meta(R)* est guidé par notre principe de conception qui consiste à ne jamais utiliser de verrous durant l'accès aux métadonnées. Nous argumentons cela par le fait que les

mécanismes de verrous ne passent jamais à l'échelle notamment parce qu'un noeud détenant un verrou peut quitter le système et donc bloquer tous les autres nœuds qui souhaitent accéder aux même métadonnées. Cependant, nous avons besoin de contrôler l'accès simultané de plusieurs GT afin d'éviter des incohérences au niveau du graphe de précédence. En bref, *Last(R)* permet aux GT de reconstruire le graphe de précédence complet de manière cohérent. Nous donnerons plus de détails sur la gestion de la cohérence des métadonnées dans la section 5.2

4.4 Conclusion

Dans ce chapitre nous avons présenté, l'architecture de notre solution en spécifiant ses différents composants, leur rôle et leur modèle de communication. Le choix d'une architecture hybride à mi-chemin entre les systèmes P2P structurés et ceux non structurés nous permet de tirer profit des avantages des uns et des autres. En fait, la structuration des noeuds GT autour d'un anneau logique permet de faciliter leur collaboration pour assurer le traitement cohérent des transactions alors que la structuration faible des nœuds ND leur confère une grande autonomie. Notre intergiciel redondant permet de faire face à la volatilité d'un environnement à large échelle puisqu'à chaque fois qu'un nœud GT ou ND tombe en panne, nous utilisons un autre nœud disponible pour continuer le traitement ou récupérer les données. L'utilisation d'un catalogue réparti facilite l'exploitation des ressources disponibles et un contrôle global de l'état du système. Pour rendre disponible les informations stockées à l'intérieur du catalogue, nous avons utilisé JuxMem, puis une DHT qui sont des systèmes de gestion de données à large échelle. Pour exploiter avec efficacité, les ressources du systèmes (équilibrer les charges, identifier rapidement le nœud optimal, etc.), nous collectons plusieurs informations dans le catalogue comme métadonnées et nous les avons structurés logiquement pour que leur manipulation (lecture et modification) soit simple en se basant sur les interfaces offertes par JuxMem ou par la DHT.

Dans le prochain chapitre, nous décrivons comment les différents composants de notre architecture interagissent pour garantir un traitement de transaction adapté à large échelle et tolérant aux pannes.

Chapitre 5

Routage des transactions

Nous considérons un environnement constitué d'une base de données répartie à très grande échelle, et dont les fragments sont répliqués sur plusieurs nœuds de données. Dans un tel environnement, le routage d'une transaction consiste tout d'abord à localiser les répliques contenant les données à manipuler. Ensuite, il s'agit de choisir, parmi les répliques candidates, celle qui sera accédée. Le choix de la réplique vise un objectif de performance : la réplique minimisant le temps de réponse de la transaction, est choisie. Plus précisément, le choix de la réplique s'appuie sur une fonction de coût qui estime la durée de traitement d'une transaction sur un nœud de données, en tenant compte des pré-traitements nécessaires au maintien de la cohérence des données. Le coût d'une transaction est l'estimation de son temps de réponse, en fonction de la charge du nœud et de la latence des communications entre les différents nœuds impliqués dans le routage de la transaction.

Dans ce chapitre, nous détaillons les deux principales tâches du routage :

1. Répertorier les répliques candidates et leur état. Nous décrivons l'accès au catalogue réparti et la gestion des métadonnées qu'il contient. Les solutions proposées garantissent la cohérence des métadonnées. Deux variantes sont décrites et leurs avantages respectifs sont présentés.

2. Gérer le traitement d'une transaction sur une réplique. Les solutions proposées ordonnent les transactions de telle sorte que les ordres, sur chaque réplique, soient tous compatibles entre eux, *i.e.* , qu'ils soient tous conformes au graphe de sérialisation définit dans le chapitre précédent. Il existe plusieurs façons d'exécuter une transaction sans compromettre la cohérence des données. Nous distinguons deux algorithmes d'exécution des transactions. Le premier algorithme appelé routage pessimiste consiste à exécuter au préalable l'ensemble des transactions pouvant être en conflit avec la transaction demandée, puis à exécuter ensuite la transaction proprement dite. Le deuxième algorithme appelé routage hybride consiste à omettre certains traitements préalables afin de gagner du temps. Une tentative d'exécution optimiste de la transaction est effectuée au risque de compromettre la cohérence des données. Il s'en suit une vérification de l'état de la base de données afin de rejeter les tentatives qui introduiraient des incohérences. Les risques de conflits entre transactions étant faibles avec les applications Web 2.0, alors le routage hybride s'avèrera plus performant que le routage

pessimiste.

Le chapitre s'organise comme suit. La section 5.1 présente le routage des transactions en détaillant tout d'abord l'algorithme de routage de manière générique 5.1.2. Puis l'algorithme pessimiste 5.1.3 et enfin l'algorithme hybride 5.1.4 sont détaillés. La section 5.2 présente la gestion des métadonnées et l'accès au catalogue réparti.

5.1 Routage des transactions

Cette section présente les détails de l'algorithme de routage. Le routage est effectué par un gestionnaire de transaction (GT). Le routage a pour objectif de déterminer pour chaque transaction, un nœud de données (ND) apte à traiter rapidement la transaction tout en assurant que les différentes répliques restent cohérentes. La solution étant entièrement décentralisée, l'algorithme de routage se déroule sur chaque nœud GT. Ainsi, un GT doit tenir compte des autres GT lorsqu'il effectue le routage, afin d'éviter que des choix de routage contradictoires se produisent. La condition suffisante pour éviter un routage contradictoire est que chaque GT ordonne les transactions qu'il reçoit dans l'ordre défini par le graphe GSG.

Par conséquent, on suppose ici que chaque GT peut lire et compléter le GSG de manière cohérente. Les mécanismes garantissant la cohérence du GSG, sont décrits dans la section suivante. L'interface d'accès au GSG pendant le routage de la transaction T consiste en deux méthodes *getMetaData(T)* et *putMetaData(G)* qui ont été définis dans la section 4.3.2 du chapitre architecture.

5.1.1 Définition du graphe de rafraîchissement et du plan d'exécution

L'algorithme de routage a pour rôle de déterminer, à chaque fois qu'une nouvelle transaction T arrive, le plan d'exécution de T. Le plan d'exécution de T sur un nœud de donnée ND, nommé $P(T, ND)$ est la procédure suivie pour exécuter T sur ND. Dans notre approche, c'est l'ordonnancement de l'ensemble des transactions qui précèdent T, appelé graphe de rafraîchissement (GR^T) suivi de T elle même.

Définition 12. *Graphe de rafraîchissement pour une transaction T*
Un graphe de rafraîchissement d'une transaction T pour un nœud ND_i est un graphe orienté et acyclique noté GR_i^T <$G, \rightarrow$> tel que :
i) GR_i^T est un sous graphe du GSG ;
ii) $\forall\, T' \subset G$, T' est une transaction de mise à jour ;
iii) $\forall\, T' \subset G, T' \rightarrow T$;
iv) l'exécution de toutes les transactions de G sur ND_i est suffisante et nécessaire pour rendre ND_i suffisamment frais vis-à-vis de T, autrement dit, T peut démarrer son exécution.

GR_i^T est le plus petit sous-graphe du GSG tel que l'exécution des nœuds de G dans l'ordre rend ND_i suffisamment frais pour l'exécution de T. En d'autres termes, après application de GR^T sur ND_i, l'obsolescence de ce dernier par rapport à chaque relation lue par T est inférieure à l'obsolescence tolérée par T pour la relation correspondante.

Nous définissons de manière formelle, le plan d'exécution de T sur un ND_i comme suit :

Définition 13. *Plan d'exécution*
Le plan d'exécution de T sur un nœud ND_i est : $P_i(T, ND_i) = GR_i^T \cup \{T\}$, i.e. *$GR_i^T$ avec tous les arcs allant de GR_i^T vers T.*

Après avoir déterminé le plan d'exécution, il faut l'exécuter. L'exécution de P peut se faire de deux manières : *1*) soit on exécute totalement le plan sur un ND_i, on parle d'algorithme de routage pessimiste ; *2*) soit on exécute uniquement T et à la fin on vérifie s'il existe des conflits réels entre T et les transactions de GR_i^T, on parle d'algorithme de routage optimiste. Le second algorithme fait l'hypothèse que le plan d'exécution est obtenu à partir des conflits potentiels puisqu'il est un sous-graphe de GSG et ne correspond pas totalement aux conflits réels. Avant de décrire ces deux algorithmes qui définissent comment les transactions sont exécutées et validées sur les ND, nous présentons d'abord le scénario global de routage qui est identique pour les deux algorithmes.

5.1.2 Algorithme générique de routage

Le processus de routage débute dès la réception de la transaction et prend fin après exécution de la transaction. L'algorithme de routage est schématisé par la figure 5.1.

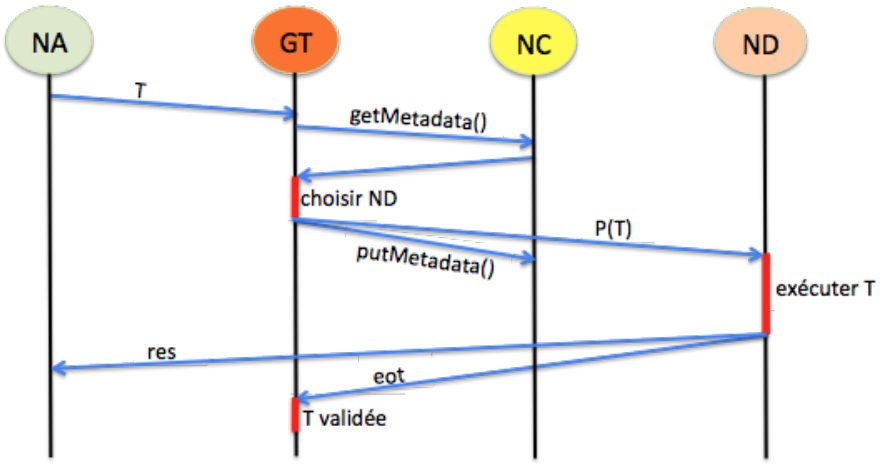

FIGURE 5.1 – Scénario de routage

Au début, un nœud applicatif NA envoie une transaction T à un nœud GT. Si T est une transaction de mise à jour, alors les étapes suivantes sont suivies :

1. Le GT identifie d'abord les classes de conflits, *i.e.* les relations sollicitées par T ($Rel(T)$). Pour chaque classe de conflit, le GT demande au répertoire réparti deux types d'informations : les contraintes de précédence existantes et relatives à la transaction T, et la localisation de tous les sites dont le contenu inclut toutes les données sollicitées par T. Le GT reçoit du répertoire réparti, une description de tous les sites candidats avec leur état mentionnant les transactions déjà validées ou en cours sur chaque réplique.

2. Ensuite, le GT choisit le ND le plus optimal, *i.e.* le nœud de données qui minimise le temps estimé par une fonction de coût pour exécuter T. Une fois un ND choisi, le GT lui envoie le plan d'exécution.
3. Le ND qui reçoit le plan d'exécution, exécute soit toutes les transactions de $P(T, ND)$, soit T seulement en fonction de l'algorithme de routage choisi. Une fois que le ND a validé T, il envoie une notification au GT et au NA. Pour NA, la notification correspond aux résultats de l'exécution de T alors que pour GT, elle consiste en information sur la fin de la transaction (validée ou annulée). A la réception de la notification du ND, le GT mentionne dans le répertoire que la transaction T est validée avec succès sur le ND ou la route de nouveau sur un autre ND.

Les détails du calcul du nœud optimal ainsi que les mécanismes d'exécution de $P(T, ND)$ seront donnés dans la section 5.1.3 (resp. section 5.1.4) pour l'algorithme de routage pessimiste (resp. optimiste).

Si T est une requête, alors la première étape est identique à celle d'une transaction de mise à jour. A l'étape 2, le plan $P(T, ND)$ inclut uniquement les transactions à propager pour rendre le ND correspondant frais vis-à-vis des exigences de la requête. Le GT choisit le ND qui minimise le coût du plan d'exécution. A l'étape 3, le ND exécute toujours le plan complètement.

5.1.3 Algorithmes de routage pessimiste

Dans cette section nous décrivons la mise en œuvre du scénario décrit dans la section précédente avec un modèle d'exécution pessimiste. Le principe de base est que les transactions sont exécutées dans l'ordre défini par le plan d'exécution. Nous présentons d'abord l'algorithme pour choisir un nœud ND afin d'exécuter une transaction T. Puis, nous détaillons l'exécution de T sur ND.

Choix du nœud optimal

Nous définissons par $Card(G)$ le nombre de sommets d'un graphe, autrement dit, le nombre de transactions. La fonction $getFirst(G)$ retourne le premier sommet (*i.e.* sommet sans prédécesseur) d'un graphe G. Le choix du nœud sur lequel exécuter une transaction est basée sur le coût et utilise la synchronisation à la demande. Il tient compte du coût de rafraîchissement d'un nœud dans le calcul du coût global d'un nœud.

Dans un premier temps, l'algorithme évalue, pour chaque nœud ND_j de la liste des candidats potentiels :

- la charge de ND_j. Cette composante du coût du nœud est obtenue en évaluant le temps d'exécution restant de toutes les transactions actives sur ND_j ;
- le coût de rafraîchissement de ND_j afin que celui-ci soit suffisamment frais par rapport à l'obsolescence tolérée par la transaction T (on rappelle que si T fait des mises à jours, cette tolérance est forcément nulle). A cet effet, l'algorithme calcule le graphe de rafraîchissement GR_j^T pour ND_j. La procédure de calcul de GR_j^T est donnée par l'algorithme 1.

Algorithme 1: *CalculRafraichissement (ND$_j$, Obs)*

entrées : *ND$_j$* nœud de données, Obs obsolescence tolérée, GSG.
sorties : GR_j^T graphe de rafraichissement
variables : GV fraîcheur du nœud *ND$_j$ i.e.* graphe de transactions déjà validées et/ou en cours sur *ND$_j$*.

```
1 begin
2     GR_j^T = ∅ ;
3     while Card(GSG) − card(GV) ≤ Obs do
4         T' = getFirst(GSG − GV) ;
5         GR_j^T = GR_j^T ∪ T');
6         GV = GV ∪ T' ;
7     return GR_j^T
```

Le coût de rafraîchissement est donc le temps total estimé pour exécuter toutes les transactions dans GR_j^T. Nous mentionnons également que GR_j^T peut contenir des transactions déjà validées et des transactions en cours.

– le coût d'exécution de T elle-même. Ce coût est obtenu en calculant la moyenne glissante du temps d'exécution des dernières transactions exécutées sur le système ;
– le coût total de l'exécution de T sur *ND$_j$* appelé $CoutExec(ND_j, T)$ est obtenu en faisant la somme des trois coûts précédents.

Ensuite le ND avec un coût d'exécution le plus faible est choisi comme nœud optimal et correspond à *ND$_k$* défini par notre fonction de coût **CoutExec**$(ND_k, T) = Min(\{(CoutExec(ND_i, T)\})$ tel que pour tout *ND$_i$* appartenant aux nœuds candidats. La fonction *ChoisirNoeud()* décrite par l'algorithme 2 donne les détails du calcul du nœud optimal.

Exécution cohérente des transactions au niveau du ND

Dans cette section nous présentons comment un ND assure la cohérence mutuelle en exécutant une transaction conformément aux exigences du plan d'exécution. Nous montrons d'abord qu'une exécution suivant le plan d'exécution est cohérente puis nous présentons l'algorithme utilisé pour exécuter le plan.

Maintien de la cohérence. Ce paragraphe décrit pourquoi le plan d'exécution généré par l'algorithme 5.1.3 est suffisant pour garantir la cohérence mutuelle. Pour ce faire nous énonçons la propriété suivante.

Propriété 1. *Si l'ordre d'exécution des transactions sur chaque nœud est compatible avec l'ordre global (GSG), alors la cohérence à terme est garantie puisque toutes les transactions seront exécutées sur tous les nœuds dans un ordre compatible.*

Preuve: Soient deux transactions $T1$ et $T2$ (avec $debut(T1) < debut(T2)$) routées respectivement

Algorithme 2: *ChoisirNoeud (LC, Obs)*

entrées : LC liste des nœuds candidats, Obs obsolescence tolérée.
sorties : N_{opt} nœud ND choisi, P plan d'exécution.
variables : GC : Graphe de transaction en cours sur le ND considéré, GR le graphe de rafraichissement, $AVG(T)$: temps moyen d'exécution des transactions, Min : réel, CoutExec : coût d'exécution de T

```
1 begin
2     Min = ∞ ;
3     foreach N ∈ LC do
4         GR = CalculRafraichissement(N, Obs);
5         CoutExec(N, T) = AVG(T) * [(Card(GR) + Card(GC)) + 1];
6         if CoutExec(N, T) < Min then
7             Min = CoutExec(N, T);
8             N_opt = N;
9     return N_opt
```

par $GT1$ et $GT2$ sur *ND*1 et *ND*2. Supposons que $T1$ et $T2$ soient exécutées dans un ordre différent sur les deux ND, par exemple, $T1 \rightarrow T2$ sur *ND*1 et $T2 \rightarrow T1$ sur *ND*2.

Cela veut dire que $GT1$ et $GT2$ ont utilisé chacun un GSG différent pour calculer les plans d'exécution de $T1$ et $T2$, autrement dit, $GT2$ n'a pas lu les modifications faites sur le GSG par $GT1$. Ceci n'est pas possible puisque le GSG est unique et est géré de tel sorte que tout accès concurrent soit sérialisé (cf. section 5.2). Ainsi $GT2$ utilise forcément le même GSG que $GT1$ vient de modifier, ce qui veut dire que $GT2$ inclut $T1$ dans le graphe de rafraîchissement requis pour exécuter $T2$ et dès lors $T1 \rightarrow T2$.

De plus comme toutes les transactions de mise à jour et de rafraîchissement sont routées avec cet algorithme 5.1.3, alors la convergence des copies ne peut pas être compromise puisque les ND exécutent les transactions dans le même ordre qu'ils les reçoivent (cf. paragraphe ci-après).

Exécution des transactions. Pour exécuter les transactions sans compromettre la cohérence mutuelle, il faut garantir la propriété 2, qui à son tour exige une application parfaite de ($P(ND, T)$). Pour respecter l'ordre de $P(ND, T)$, l'*Ordonnanceur* d'un nœud ND ne transmet au SGBD local la transactions T que lorsque toutes les transactions qui l'ont précédées ont été transmises et validées avec succès. Si une ou plusieurs transactions précédant T ne peuvent pas être exécutées pour une raison quelconque, l'*Ordonnanceur* avise le *Planificateur* qui à son tour renseigne le GT de la situation en lui faisant parvenir l'état de rafraîchissement du nœud si toutefois il a été entamé. En d'autres mots, si une partie des transactions de rafraîchissement a été validée avec succès, alors le GT est tenu au courant pour qu'il mette à jour les informations du catalogue. L'algorithme 3 définit la procédure déroulée par l'*Ordonnanceur* pour s'assurer de la bonne exécution de P.

La fonction $delivrer(T_c)$ permet d'envoyer la transaction courante au SGBD local pour exécu-

Algorithme 3: *ExecuterP* (P)

```
entrées  : P plan d'exécution.
sorties  : booléen
1  begin
2      while Card(P) ≠ 0 do
3          T_c = (getFirst(P));
4          delivrer(T_c) ;
5          repeat
6              attendre();
7          until fin transaction;
8          if T_c.etat=validee then
9              P = P − T_c ;
10         else
11             return Echec ;
12     return Succes;
```

tion. De plus, l'*Ordonnanceur* via la fonction $attendre()$ est bloqué jusqu'à ce que T_c soit validée ou annulée. Si T_c est annulée alors la transaction T échoue puisque le nœud ne sera pas suffisamment frais pour l'exécuter. Une fois le GT averti de l'échec de l'exécution de T ou d'une exécution en partie de la séquence de transaction qui devrait la précéder, il choisit un autre nœud pour reprendre l'exécution de T mais en recalculant la nouvelle séquence de précédence relative au nouveau ND choisi. Ce processus est itéré jusqu'à ce que la transaction puisse être exécutée sur un ND. En cas de non exécution de la transaction après avoir sollicité toutes les répliques candidates, le client est informé.

Exemple. La figure 5.2 illustre un système composé de deux nœuds de données $ND1$ et $ND2$. Le catalogue réparti indique que $ND1$ a reçu $T1$ et $T2$ alors que $ND2$ n'a exécuté que $T1$ avec la contrainte $T1$ précède $T2$. Supposons qu'à partir de cet état du système, il arrive une troisième

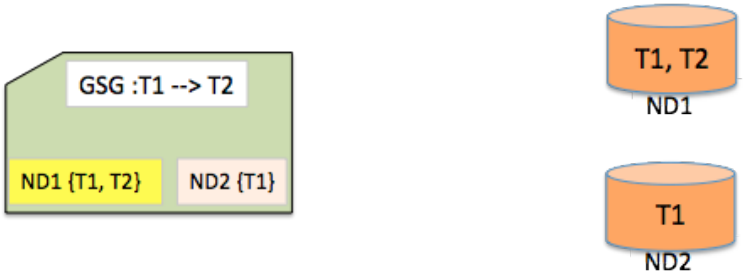

FIGURE 5.2 – GSG avant routage

transaction $T3$ avec un niveau de fraîcheur maximal. En d'autres mots, $T3$ requiert l'état le plus récent des données. La figure 5.3 montre que le GT ayant reçu $T3$, par l'intermédiaire de l'algorithme décrit précédemment désigne le nœud $ND2$ comme nœud optimal. Ce qui fait que le GR correspondant est $T2$ puisque c'est l'unique transaction dans le GSG à ne pas être exécutée sur $ND2$. Enfin, le GT envoie le GR et $T3$ à $ND2$ puis met à jour le catalogue. Ainsi, comme le montre la figure 5.3, le GSG est complété avec $T3$ et l'état de $ND2$ inclut désormais les deux dernières transactions qui viennent d'être envoyées. L'envoi de GR suivi de $T3$ signifie et sera interprété par ailleurs par $ND2$ que $T2 \rightarrow T3$.

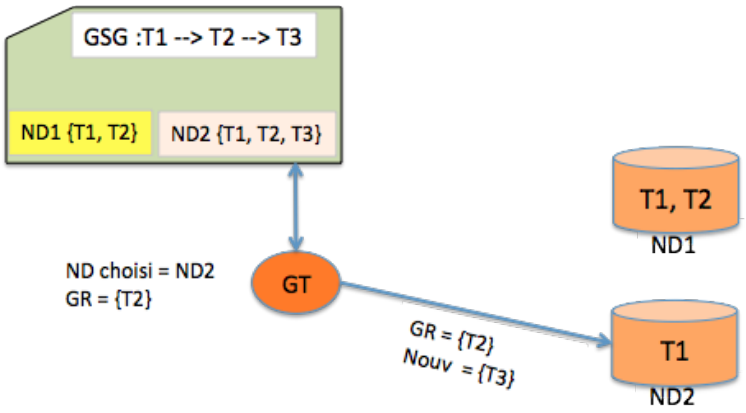

FIGURE 5.3 – GSG après routage

Nous mentionnons que le GSG est utilisé que pour garder la cohérence par conséquent, il ne contient que les transactions de mises à jour qui peuvent introduire des incohérences.

Discussion

Le mécanisme de routage décrit dans cette section assure une sérialisation globale de manière pessimiste. Il est utilisé aussi bien pour router les transactions de mise à jour que les requêtes. Chaque transaction est associée avec ses classes de conflits, qui contiennent les données que la transaction peut potentiellement lire (resp. modifier). En fonction des classes de conflits, les transactions sont ordonnées dans un GSG en s'appuyant sur leur ordre d'arrivée. Bien que cette approche assure une sérialisation globale, il réduit malheureusement la parallélisation du traitement des transactions puisqu'elle s'appuie sur des sur-ensembles potentiels de données réellement accédées. Par exemple, si les transactions T et T' écrivent sur la relation R, alors elles seront exécutées sur chaque nœud dans le même ordre. Pourtant, si T et T' n'accèdent pas aux même tuples, elles pourraient être exécutées dans n'importe quel ordre, ce qui accroît le parallélisme et donc le débit transactionnel.

Par conséquent, pour accroître la parallélisation du traitement des transactions, nous proposons dans la prochaine section, un second algorithme hybride qui combine une approche pessimiste et optimiste.

5.1.4 Algorithme de routage hybride

L'objectif du routage hybride est de traiter une transaction plus rapidement en réduisant les pré-traitements appliqués juste avant le traitement de la transaction proprement dite.

Le routage hybride se déroule comme suit : Un GT reçoit une nouvelle transaction entrante, nommée T. Puis le GT consulte le catalogue et obtient le graphe de rafraichissement contenant toutes les transactions courantes (notées TC) ou validées (notées TV) qui précèdent T. Il obtient aussi la description de toutes les répliques candidates pour traiter T. Pour chaque réplique candidate, le GT construit un plan d'exécution constitué des TV et de T, de telle sorte que chaque transaction dans TV précède T. On rappelle que l'idée est de ne pas inclure les TC dans le plan. Le coût de chaque plan est estimé, celui de moindre coût est choisi pour être exécuté. Le GT demande au nœud de donnée (ND) de traiter le plan d'exécution. Lorsque le ND reçoit le plan, il exécute et valide toutes les TV. Puis il exécute T sans la valider. Il compare les données accédées par T avec celles accédées par les transactions TC afin de détecter un conflit. Deux issues peuvent se produire : (1) S'il n'y a pas de conflit, alors T est validée. Le ND notifie ensuite le GT de l'absence conflit entre T et TC. Le GT répercute cette information au niveau du catalogue afin de simplifier le graphe de précédence : toutes les précédences entre T et les TC sont supprimées. (2) En cas de conflit, T est abandonnée. Le ND bascule en mode pessimiste. Il traite l'ensemble des TC puis finit par traiter T.

Nous exposons ci-après les principales raisons qui justifient le routage hybride. L'avantage du routage hybride repose sur notre aptitude à déterminer la présence d'un conflit réel entre deux transactions. Plus précisément, si on peut prouver que le graphe de rafraichissement est disjoint de la transaction à traiter, alors le traitement de T sans rafraichissement est acceptable, et évidemment plus rapide.

Or, dans notre contexte, la preuve que le rafraichissement n'est pas nécessaire, se base sur la connaissance des conflits réels, ce qui nécessite d'avoir déjà traité la transaction. C'est pourquoi le routage hybride consiste tout d'abord à tenter, de manière optimiste, de traiter la transaction sans son rafraichissement, puis à vérifier, à postériori, que cela est correct.

Naturellement, la probabilité que la transaction accède aux mêmes données qu'une autre transaction est proportionnelle au nombre de transactions considérées. Par conséquent, nous sommes confrontés à un compromis : d'une part ne pas trop rafraichir si cela n'est pas nécessaire, d'autre part réduire le risque d'échec en rafraichissant malgré tout

Afin de trouver une issue à ce compromis, nous décidons de conserver les pré-traitements qui peuvent être appliqués rapidement, l'objectif principal étant d'accélérer le traitement d'une transaction. Ainsi, toutes les transactions déjà validées (TV) sur un autre nœud seront appliquées car elles ont l'avantage de pouvoir être appliquées immédiatement sans être ré-exécutées, donc beaucoup plus rapidement que l'exécution initiale. L'application immédiate d'une transaction validée consiste à propager ses effets sans la ré-exécuter. Notons que l'application immédiate est plus rapide que la ré-exécution du fait de notre contexte applicatif où une transaction passe beaucoup de temps à lire des données et faire des calculs et peu de temps à écrire des données. En fin de compte, le "pari optimiste" ne portera que sur les quelques transactions courantes (TC) qu'on ne peut pas appliquer immédiatement car elles n'ont pas encore validé. Les transactions courantes et potentiellement conflictuelles, étant peu nombreuses par hypothèse (faible taux de conflit des ap-

plications, comme expliqué en section 2.1), les chances de réussite du routage optimiste s'avèrent très élevées.

Dans cette section, nous détaillons comment le scénario décrit dans 5.1.2 est mise en œuvre avec un modèle d'exécution optimiste des transactions. Nous présentons d'abord l'algorithme pour choisir un nœud ND qui exécutera T. Puis, nous détaillons l'exécution de T sur ND.

Choix du nœud optimal

La fonction *ChoisirNoeudhyb()* définie par l'algorithme 4 détaille le calcul du nœud optimal.

Contrairement à l'algorithme présenté dans la section 5.1.3, le coût de rafraîchissement des transactions en cours n'est pas pris en compte dans le coût global, autrement dit, seul le coût de rafraichissement des transactions déjà validées avant $debut(T)$ sont prises en compte. Toute transaction T' tel que $debut(T) \leq fin(T') < fin(T)$ est considérée comme transaction courante. Ainsi, nous dissocions les transactions validées de celles en cours dans la séquence des transactions qui ont précédées T. Les transactions déjà validées doivent être appliquées sur le ND choisi avant de commencer l'exécution de T. Par contre, les transactions en cours ne sont pas reprises sur le ND mais leurs modifications doivent être confrontées avec celles de T pour garder la cohérence. Ce choix de comparer avec T que les transactions courantes et non celles déjà validées avant son arrivée se justifie par le souci d'une exécution rapide de T. De fait, comparer les modifications avec toutes les transactions courantes et validées est beaucoup plus coûteux et long que l'application des transactions validées sur le ND qui exécute T.

Pour calculer le graphe de rafraîchissement GR d'un ND lors de l'exécution de T, nous utilisons la même procédure de l'algorithme 1 mais en enlevant toutes les transactions non encore validées. La fonction $CalculConcurrent()$ (ligne 4) retourne toutes les transactions courantes et qui sont en conflits avec T. Ainsi, le graphe de rafraîchissement GR est obtenu en appliquant la différence entre $CalculRafraichissement$ et $CalculConcurrent()$. L'estimation du coût prend aussi en compte la charge des nœuds ND et du coût d'exécution de la transaction (obtenu par étalonnage). La fonction $CalculConcurrent()$ retourne un graphe dont les sommets ne sont pas des transactions mais des couples *(T, ND)*, qui signifie que T a comme nœud initial ND. Autrement dit, ND est le nœud sur lequel T est exécutée une première fois par conséquent, il est capable de fournir $DataSet(T)$. La ligne 6 de l'algorithme 5 montre comment on ajoute un nouveau sommet au graphe GR^T. Le terme $(getFirst(GSG - GV))$ permet de retrouver une transaction précédant T alors que la fonction $getInitial(T')$ renvoie le nœud ND sur lequel T' a été exécutée. Le code de la fonction $CalculConcurrent()$ est donné par l'algorithme 5.

Le coût obtenu avec l'algorithme 4 est sous-estimé quand un conflit réel advient puisque toutes les transactions conflictuelles courantes (GC) doivent être exécutées avant T. Cependant ce cas est rare et donc a un impact négligeable sur le fonctionnement du système. Une fois le nœud ND choisi, le GT lui envoie le plan d'exécution composé des transactions de GR, de GC et de T.

Validation des transactions

Pour accélérer le traitement, le ND qui reçoit un plan d'exécution exécute GR puis T. Une fois T en phase de validation, le ND compare les modifications de T avec celles des transactions

Algorithme 4: *ChoisirNoeudHyb (LC, Obs)*

entrées : LC liste des nœuds candidats, Obs obsolescence tolérée.
sorties : N_{opt} nœud ND choisi.
variables : GC : Graphe de transaction en cours sur le ND considéré, $AVG(T)$: temps moyen d'exécution des transactions, Min : réel, GR graphe de rafraîchissement (de transactions validées sur les autres ND)

```
1 begin
2     Min = ∞ ;
3     foreach N ∈ LC do
4         GR = CalculRafraichissement(N, Obs) − CalculConcurrent(N, Obs)
          CoutExec(N, T) = AVG(T) ∗ [Card(GC) + Card(GR) + 1];
5         if CoutExec(N, T) < Min then
6             Min = CoutExec(N, T);
7             N_opt = N;
8     return N_opt
```

Algorithme 5: *CalculConcurrent (ND_j, Obs)*

entrées : ND_j nœud de données, Obs obsolescence tolérée, GSG.
sorties : GC graphe de concurrence
variables : GV fraîcheur du nœud ND_j *i.e.* graphe de transactions déjà validées et/ou en cours sur ND_j.

```
1 begin
2     GC = ∅ ;
3     while Card(GSG) − card(GV) ≤ Obs do
4         T' = getFirst(GSG − GV) ;
5         if debut(T) ≤ fin(T') < fin(T) then
6             GC = GC ∪ (T', getInitial(T'));
7             GV = GV ∪ T' ;
8     return GC
```

en cours GC. En effet, le ND contacte tous les ND qui ont exécutée des transactions comprises dans GC. Nous rappelons que GC est un graphe dont les sommets sont composés des transactions conflictuelles en cours et les ND sur lesquels elles s'exécutent.

Par exemple, soient deux nœuds de données *ND1* et *ND2*. Supposons que $GR = \emptyset$, *i.e.* il n'existe pas de transaction validée non encore propagée sur *ND2* et $GC = (T_1,ND_1) \rightarrow (T_2, ND_2)$. Cette séquence signifie que T_1 est en train d'être exécutée sur *ND1* et que $debut(T_1) < debut(T_2)$.

Après avoir exécuté T_2, *ND2* demande à *ND1* les données réellement manipulées par T_1 (*i.e.* les $DataSet(T_1)$) pour toute relation $R \in Rel(T_1)$. Une fois les $DataSet(T_1)$ reçus par le *ND2*, alors ce dernier compare les $DataSet(T_2)$ avec ceux de T_1. A la suite de cette comparaison deux cas peuvent se présenter :

- soit $DataSet(T_1) \cap DataSet(T_2) = \emptyset$, *i.e.* il n'y a pas d'intersection entre les données modifiées et lues par T_1 et T_2, par conséquent, T_2 peut valider. De plus, *ND2* notifie le GT en lui informant que le conflit potentiel entre T_1 et T_2 n'est pas un conflit réel, ce qui veut dire que $T_1 \not\rightarrow T_2$.
- soit $DataSet(T_1) \cap DataSet(T_2) \neq \emptyset$, *i.e.* T_2 a lu ou modifié des données modifiées par T_1, alors T_2 est annulée et *ND2* exécute dans l'ordre $T1$ puis $T2$. Mais comme les $DataSet(T_1)$ contiennent les modifications de T_1, alors *ND2* ne ré-exécute pas $T1$ mais plutôt il se contente d'appliquer les modifications. Le conflit potentiel prédéfini sur T_1 et T_2 est réel et donc T_1 et T_2 seront exécutées sur toute les répliques en respectant l'ordre de précédence.

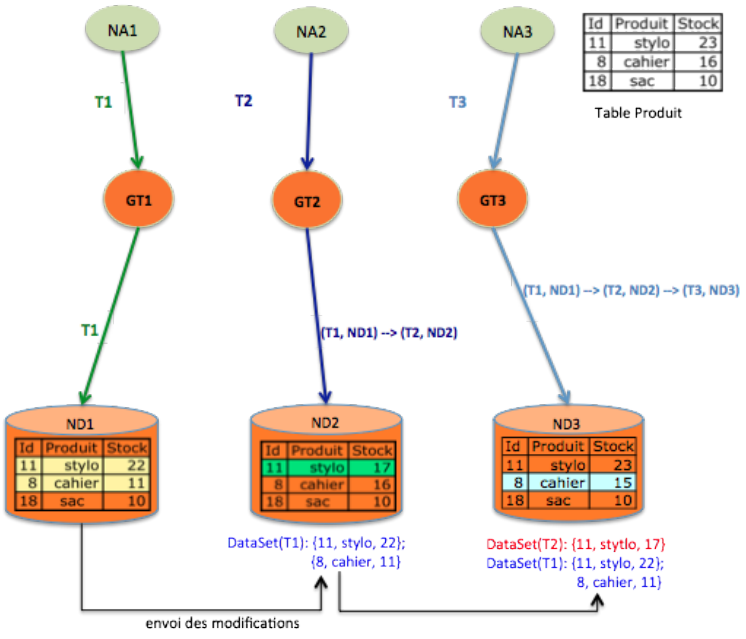

FIGURE 5.4 – Validation des transactions

Nous notons par ailleurs que si le nombre de transactions concurrentes à T est supérieur à un (*i.e.* Card(GC) > 1), alors le ND est obligé de comparer les $DataSet(T)$ avec tous $DataSet(T_i)$

tels que $T_i \in GC$. Cependant, pour récupérer tous les $DataSet(T_i)$, un seul ND est contacté. Les détails de cette stratégie de communication seront données dans les prochaines sections.

Exemple. La figure5.4 illustre la phase de validation de trois transactions concurrentes $T1$, $T2$ et $T3$ (avec $debut(T1) < debut(T2) < debut(T3)$), chacune manipulant la même relation $Produit$. Supposons que $GT1$ reçoit $T1$ et la route sur *ND1*, $GT2$ (respectivement $GT3$) route $T2$ vers *ND2* (respectivement $T3$ vers $GT3$).

Quand $GT2$ route $T2$ vers *ND2*, il indique que $T1$ qui précède $T2$ est exécutée sur *ND1* : ($T1$, *ND1*) $\rightarrow$ ($T2$, *ND2)*.

De même, $GT3$ indique à *ND3* les nœuds sur lesquels s'exécutent $T1$ et $T2$: ($T1$, *ND1*) $\rightarrow$ ($T2$, *ND2*) $\rightarrow$ ($T3$, *ND3*).

Quand *ND1* veut valider $T1$, cela se passe sans problème car il n'y a aucune transaction qui précède $T1$.

Par contre, quand $ND2$ tente de valider $T2$, il récupère les informations de $T1$. En faisant la comparaison des modifications de $T1$ et $T2$, il trouve que :$DataSet(T1) \cap DataSet(T2) = \{11, stylo, 23\}$. Il annule alors $T2$, applique les $DataSet(T1)$ et reprend $T2$.

$GT3$ effectue aussi la même procédure et trouve que : (1) $DataSet(T1) \cap DataSet(T3) = \{8, cahier, 16\}$; et (2) $DataSet(T2) \cap DataSet(T3) = \emptyset$. Il conclut que $T1 \rightarrow T3$ par contre $T2 \not\rightarrow T3$. Il annule ainsi $T3$, applique les modifications de $T1$ et reprend $T3$. L'information d'absence de précédence entre $T2$ et $T3$ sera remontée au GT pour qu'il corrige le GSG.

L'avantage majeur de cette vérification de conflits en fin d'exécution de la transaction esÒt qu'il favorise plus de parallélisme et donne l'opportunité de différer les propagations. Avec l'exemple précédent, l'exécution parallèle de $T2$ et $T3$ favorise un temps de réponse plus faible que $T2$ suivi de $T3$. Cette stratégie est très adaptée à notre contexte dans lequel les conflits sont rares.

Calcul d'intersection des DataSets. Dans ce paragraphe, nous définissons comment les conflits sont détectés lors de la validation. Les $DataSet(T)$ sont un sous-ensemble des tuples de $Rel(T)$ et sont obtenus avec des règles actives. Chaque tuple à un identifiant unique appelé *Id* et correspond à la clé de la relation. Nous supposons que la clé d'une relation n'est jamais modifiée par une transaction de mise à jour. Lors de l'extraction des $DataSet(T)$, les valeurs de tous les attributs d'un tuple t modifié ou lu par T sont associées avec leur *Id* pour former un nouveau tuple t_{acc}. A cela s'ajoute le type d'opération (lecture ou écriture) permettant d'obtenir t_{acc}. Ainsi les $DataSet(T)$ sont similaires à une vue obtenue par sélection sur $Rel(T)$ en gardant les mêmes noms des attributs. Nous associons alors à chaque $DataSet(T)$ un schéma relationnel appelé $DataSet_{Sc}(T)$. Le schéma relationnel d'un $DataSet(T)$ est composé des noms des attributs de $Rel(T)$ alors que $DataSet(T)$ est la relation associé à tous les tuples modifiés ou lus.
Supposons la relation **Produit** (*Id, nomproduit, stock*) et les deux transactions suivantes liées par la contrainte de précédence $T_i \rightarrow T_j$.

- T_i : SELECT NOMPRODUIT FROM PRODUIT WHERE ID=25 ;
- T_j : UPDATE PRODUIT SET STOCK=17 WHERE ID=25 ;

Alors les schémas des $DataSet(T_i)$ et $DataSetT_j$ sont respectivement $DataSet_{Sc}(T_i)$={Id, nom-produit} et $DataSet_{Sc}(T_j)$={Id, stock}. L'algorithme 6 détaille le calcul des intersections. Pour faire la comparaison des $DataSet(T_i)$ et $DataSet(T_j)$, les schémas sont d'abord comparés (ligne 2-5). S'il n'existe aucun attribut en commun en dehors des *Id*, cela est suffisant pour conclure que $T_i \not\rightarrow T_j$. Par contre, si un attribut figure en même temps dans les deux schémas, les valeurs des attributs sont comparées pour identifier s'il y a réellement un conflit (ligne 8-12).

Algorithme 6: *CalculIntersection (*$DataSet(T_i)$, $DataSet(T_j)$

```
   entrées    : DataSet(T_k) les datasets à comparer.
   sorties    : Booleén
   variables  : Tab : tableau de chaine de caractères
 1 begin
 2     foreach attr1 ∈ DataSet_Sc(T_i) do
 3         foreach attr2 ∈ DataSet_Sc(T_j) do
 4             if (attr2 = attr1) et (attr1 ≠ Id) then
 5                 Ajouter attr2 dans Tab;
 6     if Tab est vide then
 7         return false;
 8     else
 9         foreach tupleA ∈ DataSet(T_i) et tupleB ∈ DataSet(T_j) do
10             foreach attr ∈ Tab do
11                 if (tupleA.Id = tupleB.Id) et (tupleA.attr ≠ tupleB.attr) then
12                     S'il existe au moins une écriture return true;
13         return false;
```

Nous remarquons que l'algorithme 6 permet de vérifier l'intersection des DataSet(T) que si les transactions concernées ne font que des modifications. Lorsque l'une des deux transactions T_i et T_j liées par le conflit potentiel $T_i \rightarrow T_j$ fait des insertions ou des suppressions, alors l'algorithme 6 conclut que $T_i \not\rightarrow T_j$ puisqu'il n'existe pas de tuples accédés en commun. Pourtant, ceci n'est pas toujours vrai comme le montrent les cas suivants :

- T_i fait des insertions et T_j des modifications (ou des suppressions), alors les tuples insérés par T_i peuvent satisfaire les conditions requises par T_i pour faire une modification. Donc, l'exécution de T_i et de T_j dans n'importe quel ordre ne donne pas le même résultat ;
- T_i et T_j font des insertions. Si les insertions dépendent d'une lecture préalable de la base, alors l'ordre de T_i et de T_j a une influence sur l'état de la base.

Pour éviter ce genre de problème, nous maintenons le même ordre défini grâce aux conflits potentiels. Autrement dit, nous concluons que les conflits réels sont identiques aux conflits potentiels à chaque fois que l'une des transactions concernées contiennent des insertions. Cependant ce cas est rare car la plupart des applications Web 2.0 et des ASP maintiennent des serveurs pour *insert-*

only dans le souci de faciliter le passage à l'échelle, de maintenir plusieurs versions, d'assurer la cohérence à terme, etc. [FJB09].

Gestion des surcharges induites par la validation

La procédure de validation utilisée pour terminer l'exécution d'une transaction T peut engendrer quelques surcharges. En effet la récupération des $DataSet(T)$ peut allonger le temps de réponse des transactions puisque T n'est validée qu'après comparaison de $DataSet(T)$ avec tous les $DataSet(T_i)$ correspondant aux modifications des transactions concurrentes. Ainsi, les messages envoyés peuvent être important si le nombre de transactions concurrentes est important. Dans ce paragraphe, nous étudions les mécanismes utilisés pour borner le temps de réponse et le nombre de messages. Nous détaillons aussi comment les $DataSet(T)$ peuvent être utilisés pour rendre plus frais les répliques même en situation de conflit.

Gestion des messages. Pour valider une transaction, un ND a besoin de contacter tous les ND qui ont exécuté une transaction conflictuelle et courante. Ainsi, le coût de la communication devient très significatif quand le nombre de ND à contacter est important. Pour limiter le nombre de message, un ND cache les $DataSet(T)$ qu'il a reçu d'un autre ND pour pouvoir les transférer à d'autres ND qui en auront besoin. En d'autres mots, un ND qui valide T contacte toujours le ND qui a exécuté la dernière transaction pour chacune des classes de conflits dans lesquelles T est impliquée. Un ND qui doit fournir les $DataSet(T')$, transmet également tous $DataSet(T_i)$ tel que T_i soit dans la séquence des transactions concurrentes qui précède T'.

Par exemple, soit la séquence de précédence suivante $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ routée respectivement sur $ND1$, $ND2$, $ND3$ and $ND4$. Quand $ND2$ veut valider T_2, il demande les $DataSet(T_1)$ qu'il met en cache à leur réception. Ensuite, quand $ND3$ cherche à valider T_3, il contacte seulement $ND2$ qui est capable de lui délivrer à la fois $DataSet(T_1)$ et $DataSet(T_2)$. Ainsi, en cas d'absence de panne , deux messages sont largement suffisants pour décider de valider ou d'annuler une transaction.

Propagation indirecte des mises à jour. Pour récupérer les $DataSet(T')$ d'une transaction T' précédant une transaction T en cours de validation, nous utilisons des triggers et l'interface JDBC avec laquelle nous accédons au SGBD local d'un ND. Cela génère un coût supplémentaire non négligeable quand le volume des données lues est importante. Par conséquent, une fois que les $DataSet(T)$ sont obtenus, les ND en font une exploitation maximale. En effet, à la réception des $DataSet(T)$ pour valider T', nous avons décrit que deux cas pouvaient se présenter à savoir l'existence de conflit ou l'absence de conflit entre T et T'. Dans tous les deux cas, le ND applique les modifications de T. De fait, si T et T' sont en conflit réel, le ND est obligé d'exécuter d'abord T puis T'. Si heureusement, $T \not\rightarrow T'$, le ND valide d'abord T' et ensuite applique les modifications de T puisqu'il a déjà à sa possession les $DataSet(T)$. Ainsi, même en cas d'annulation de transaction la récupération des $DataSet(T)$ n'est pas une perte en soit puisqu'ils ont permis à rendre plus frais un ND, ce qui fera de lui un candidat potentiel pour les prochaines exécution. Le gain obtenu avec cette approche est comparable à celui obtenu avec la synchronisation précoce décrite dans

[GNPV07]. Il permet de réduire la séquence de transaction nécessaire pour rendre un nœud plus frais pour les futures transactions et donc diminue le temps de réponse global.

Majoration du temps de réponse. Pour valider T, un ND est obligé d'attendre les modifications de toutes les transactions conflictuelles et courantes. Pour éviter des temps réponses élevés, nous bornons le temps d'attente pour l'obtention des $DataSet(T_i)$. Ce temps d'attente est appelé *Temps de Décision (Δ_{TD})*. A l'expiration de Δ_{TD}, le ND contacte le GT en lui envoyant le $DataSet(T)$ pour qu'il prenne une décision finale. Cette situations est très délicate car plusieurs scénarios sont envisageables. Par exemple, une transaction T_i précédant T peut être déjà exécutée sur un ND qui tombe en panne tout juste avant d'envoyer les $DataSet(T_i)$, ou même que T_i n'a pu être traitée et donc il n'existe aucun $DataSet(T_i)$ à envoyer. La détection et la gestion des pannes sont détaillées dans le chapitre 6.

De plus, si le nœud *ND1* qui veut valider T n'arrive pas à récupérer les $DataSet(T_i)$ des transactions qui précèdent T, alors tous les ND qui attendaient les $DataSet(T)$ hériteront du même problème. Pour éviter que tous ces ND bloqués contactent les GT, *ND1* les avisent afin que ces derniers puissent allonger leur temps de décision.

Ce temps de décision est choisi en prenant en compte la latence du réseau et correspond à $\Delta_{TD} \geq 2n * (Avg(T) + \lambda_N)$, avec ($\lambda_N$), la latence du réseau, $Avg(T)$, le temps de traitement d'une transaction, et n le nombre de transaction précédent T_i.

Pour trouver cette borne, supposons que $t^i(T_j)$ soit la date d'envoi de T_j à un ND et λ_N le délai écoulé avant la réception de T_j. A tout instant, le temps d'exécution restant pour exécuter T_j peut être obtenu par :

$$RT(T_j) = Avg(T) - (\delta_{GT_{sys}} - t^i(T_j))$$

avec $\delta_{GT_{sys}}$ la date système du nœud GT qui a routé T_j. Pour obtenir les datasets , chaque ND fait un aller-retour qui est équivalent à $2 * \lambda_N$. Ainsi, en supposant que le temps d'extraction des datasets est identique au temps d'exécution de la transaction, le temps total de récupération de tous les datasets est :

$$RT(T_j) = 2 * Avg(T) - (\delta_{GT_{sys}} - t^i(T_j)) + 2 * \lambda_N$$

Si le nombre de transaction précédent T_i est égal à n, alors

$$\Delta_{TD} \geq \sum_{k=0}^{k=n} (2 * Avg(T) - (\delta_{GT_{sys}} - t^i(T_k)) + 2 * \lambda_N)$$

D'où :

$$\Delta_T \geq 2n * (Avg(T) + \lambda_N)$$

5.1.5 Discussion

Dans cette section, nous avons présenté notre protocole de routage. Les applications spécifient leurs exigences en termes de besoin de cohérence, puis l'intergiciel honore ces exigences en

contrôlant la cohérence des données. Nous avons défini deux protocoles pour maintenir la cohérence globale, en fonction de la connaissance des données manipulées par les transactions. Le premier protocole ordonne les transactions à partir de la définition *a priori* des données accédées. Le deuxième protocole détermine un ordre plus souple, en comparant les données accédées, le plus tardivement possible, juste avant la validation des transactions. La deuxième solution offre plus d'opportunité de parallélisme surtout dans le contexte des applications web2.0 avec lesquelles les conflits sont rares. De plus, l'utilisation du temps de décision permet de borner le temps de validation des transactions, puisque son expiration entraîne l'intervention du GT qui a une connaissance plus détaillée de l'état des ND, ce qui peut accélérer le traitement. Cette majoration associée à une gestion des pannes permet de garde de bonnes performances en garantissant l'exécution d'une transaction quelque soient les problèmes de pannes et de latence. Nous détaillons la gestion des pannes dans le chapitre 6. Dans la prochaine section nous étudions comment les GT accèdent et manipulent réellement les métadonnées afin de garantir leur cohérence en cas d'accès concurrent.

5.2 Gestion des métadonnées

Le scénario de routage décrit dans la section 5.1.2 se déroule à chaque fois qu'un GT reçoit une transaction d'un client. Ainsi, deux ou plusieurs GT peuvent avoir des accès concurrent sur le même nœud NC si leurs transactions partagent une même classe de conflit, ou un même ND si le nœud avec le plus faible coût choisi est le même pour tous les deux GT. Soient $GT1$ et $GT2$ deus routeurs voulant accéder à $Meta(R^1)$ pour router respectivement deux transactions T_1 et T_2 qui tentent de modifier la relation R^1. Chaque GT obtient une copie de $Meta(R^1)$ via le catalogue. Supposons que $GT1$ route T_1 vers le nœud de données $ND1$ et met à jour $Meta(R^1)$ en écrivant (T_1, *ND*1) qui signifie que T_1 est entrain de s'exécuter sur $ND1$. Si $GT2$ route T_2 vers un second nœud de données ND_2 sans prendre en compte T_1, la cohérence mutuelle est compromise puisqu'une transaction de mise à jour doit lire toujours des données totalement fraîches.

Pour éviter cette situation, il faut des mécanismes d'accès concurrents au catalogue réparti pour maintenir la cohérence des métadonnées lors des opérations de routage.

Pour ce faire, nous proposons deux approches : une approche utilisant le verrouillage et une autre sans verrouillage. La gestion avec verrouillage garantit la cohérence des métadonnées et en particulier le GSG. La gestion avec le verrouillage est implémentée via JuxMem dans le but d'intégrer nos travaux dans le cadre du projet ANR Respire [Prod]. Malheureusement, nous avons remarqué que le verrouillage ne facilite pas le passage à l'échelle puisque l'attente de l'obtention d'un verrou peut être longue et rallonger les temps de réponse. C'est pourquoi, nous avons opté pour une solution sans verrouillage lors de l'accès au catalogue. De plus, nous nous sommes rendu compte qu'il existait des systèmes tels que les DHT qui permettent de gérer des données répliquées sans utilisation de mécanismes de verrouillage. Ainsi, nous avons implémenté la gestion du catalogue à travers une DHT d'autant plus que celle-ci favorise le passage à l'échelle et une grande disponibilité, deux caractéristiques très importantes pour notre système de routage.

5.2.1 Gestion des métadonnées avec verrouillage

Nous décrivons dans cette section les détails de la gestion du catalogue avec JuxMem. Comme nous l'avons déjà mentionné, un GT a besoin d'accéder au catalogue pour lire les métadonnées et les modifier. Avec l'utilisation de JuxMem, un GT se comporte comme un client via-à-vis de JuxMem. Par conséquent, toute opération de lecture ou d'écriture nécessite un verrou. Quand il s'agit d'opération de lecture, un verrou partagé est accordé au lecteur tandis qu'un verrou exclusif est fourni lors d'une opération d'écriture. Autrement dit, un GT fait une demande de verrou partagé quand il veut lire les métadonnées et un verrou exclusif lorsqu'il veut écrire dans le catalogue.

Conformément à la spécification du routage, à chaque fois qu'une transaction veut manipuler une donnée, le routeur doit accéder au catalogue pour retrouver l'ensemble des nœuds qui stockent une copie de la donnée, puis la modifier en cas de besoin. Supposons que l'on ait deux transactions $T1$ et $T2$ qui manipulent simultanément la même relation R^i. Les deux transactions sont interceptées respectivement par $GT1$ et $GT2$. $GT1$ accède à la partie du catalogue qui contient $Meta(R^i)$ pour avoir la liste des nœuds candidats en lecture. Pour ce faire, Juxmem va allouer un verrou partagé à $GT1$ qui le relâchera dès qu'il finira sa lecture. A ce moment $GT2$ acquiert un verrou partagé sur $Meta(R^i)$ et va lire la même chose que $GT1$. Si un des deux GT finit son traitement et met à jour le catalogue, la copie de $Meta(R^i)$ qui est en train d'être manipulée par l'autre GT est erronée. Pour éviter ce problème, on utilise une méthode simple qui consiste à demander et garder un verrou exclusif durant tout le choix du nœud optimal. Autrement dit, on fait toujours une lecture avec intention d'écriture. Ainsi, à chaque fois qu'un GT détient le verrou exclusif il fait toutes ses opérations de lecture et d'écriture avant de relâcher les verrous.

Pour éviter des situations d'inter-blocages entre GT, les $Meta(R^i)$ sont ordonnés suivant le nom de la relation. Ainsi, le GT qui gère T récupère les $GSG(R^i)$ en respectant cet ordre. Par exemple, si T a besoin de manipuler R^1 et $R2$, le GT demande d'abord un verrou exclusif sur $Meta(R^1)$ et tant que le verrou ne lui sera pas accordé, le GT ne peut demander un verrou exclusif sur $Meta(R^2)$. Cette situation ou un GT doit demander et obtenir tous les verrous pour commencer le processus du routage est comparable à la méthode de verrouillage à deux phases.

Ce mécanisme a la lourde conséquence d'augmenter le nombre de transactions en attente d'être routées au niveau d'un GT puisque l'attente d'un verrou peut être long ou indéfini si toutefois le GT qui le détient est en panne. Pour éviter ces problèmes nous avons proposé une solution sans verrouillage à l'aide d'une DHT.

5.2.2 Gestion des métadonnées sans verrouillage

Bien que le GSG ne soit pas verrouillé par le NC, il faut s'assurer que lorsqu'un GT modifie un GSG, aucun autre GT ne modifie le GSG simultanément, sinon il y a risque d'incohérence du GSG. Un GT doit obtenir l'accès exclusif au GSG avant de pouvoir le compléter. Cela est réalisé par un accord entre les GT concurrents et en ordonnant les accès au GSG

Pour atteindre ce but, les GT déclarent leur intention de modifier les métadonnées à chaque fois qu'ils tentent de router une transaction de mise à jour. Ce faisant, un GT utilise la primitive *get_for_update* pour récupérer un GSG. Toute demande d'accès faite à partir de cette primitive ne peut être résolue que par le NC maître de la relation correspondante. Ainsi, le NC maître détecte

les accès concurrents et peut informer les GT qui en ont besoin. Concrètement, à chaque fois qu'un GT sollicite un GSG, le NC maître lui donne une copie mais aussi dans le cas échéant lui indique le dernier GT qui a accédé simultanément au même GSG. Bien que le NC ne maintienne pas l'ordre des GT effectuant des demandes concurrentes, cet ordre existe sous la forme d'un chaînage entre les GT, car chaque GT connait son prédécesseur ($Last$). Un GT doit compléter le GSG qu'il reçoit en contactant le dernier GT ayant lu le GSG avant lui. Les détails de la gestion cohérente du GSG sont donnés ci-après.

Pour obtenir le GSG le plus cohérent, un GT contacte d'abord la DHT et particulièrement le ou les NC maîtres de la portion des métadonnées sollicités. Chaque NC maître contacté, envoie au GT le $Meta(R)$ qu'il gère. Le $Meta(R)$ contient les informations comme le $GSG(R)$, l'état de chaque réplique ($State(R_i)$) et le pointeur $Last(R)$. A la réception de la réponse du NC maître, le GT a deux alternatives :

- *i*) si le pointeur $Last(R)$ est vide, alors le GSG reçu est le plus récent et donc le GT effectue son choix, *i.e.* détermine le ND qui va exécuter la transaction ;
- *ii*) si le pointeur n'est pas vide, cela signifie qu'il y a des accès concurrents, ainsi le GT contacte le GT pointé par $Last(R)$ pour obtenir des dernières modifications faites sur le GSG avant de router la transaction.

L'utilisation du pointeur permet de connaître à tout moment le dernier GT qui a accédé au GSG et donc de pouvoir retrouver les récentes modifications qui y sont annexées. Quand un GT pointé ($Last(R)$) est contacté par un autre GT qui veut recevoir les dernières modifications, il envoie uniquement le sous-graphe qu'il a obtenu des GT qui l'ont précédé et auquel il ajoute la dernière transaction qu'il vient de router. En d'autres termes, un GT pointé n'envoie pas le GSG global mais uniquement le sous-graphe manquant au GT qui l'a contacté. L'envoi que du sous-graphe manquant à un GT minimise les informations à transférer et par conséquent la surcharge.

A titre illustratif, la figure 5.5 schématise le scénario pour construire un GSG cohérent en cas d'accès concurrent aux métadonnées. Trois applications clientes soumettent trois transactions $T1$, $T2$ et $T3$ aux routeurs GT_1, GT_2, and GT_3 respectivement. Les trois transactions mettent à jour la même relation R. Pour router $T1$, GT_1 demande au NC maître le $Meta(R)$ via l'opération $getMetadata(T_1, R_1)$: le GSG et le pointeur $Last$ sont initialement vide. Puis, GT_2 voulant router T_2, demande au NC le $Meta(R)$, et reçoit $Last$=GT_1. Ainsi, GT_2 contacte GT_1 pour les mises à jour manquantes sur le GSG ($getPreced(R)$) ; GT_1 lui envoie le sommet $T1$. Par conséquent, GT_2 reconstruit le GSG complet qui devient : $T1 \rightarrow T2$. Enfin, GT_3 déroule le même procédure que GT_2 et obtient $Last$=GT_2. Alors, GT_3 contacte à son tour GT_2 pour récupérer le sous-graphe manquant et obtient $T1 \rightarrow T2$. Ce faisant, GT_3 reconstruit le GSG complet : $T1 \rightarrow T2 \rightarrow T3$.

Quand une transaction T accède à plusieurs relations, le GSG correspondant est obtenu en réunissant les différents $GSG(R_i)$. Cependant, l'accès à plusieurs relations peut engendrer des problèmes. Soient deux transactions concurrentes T_1 et T_2. T_1 souhaite modifier la relation R_1 puis R_2 alors que T_2 décide de modifier R_2 avant R_1. Supposons que $GT1$ se charge de router T_1 alors que GT_2 s'occupe de T_2. $GT1$ demande au NC maître de R_1 le $GSG(R_1)$ et $GT2$ fait la même chose pour R_2. Alors quand chacun des GT cherchera de récupérer le second GSG,

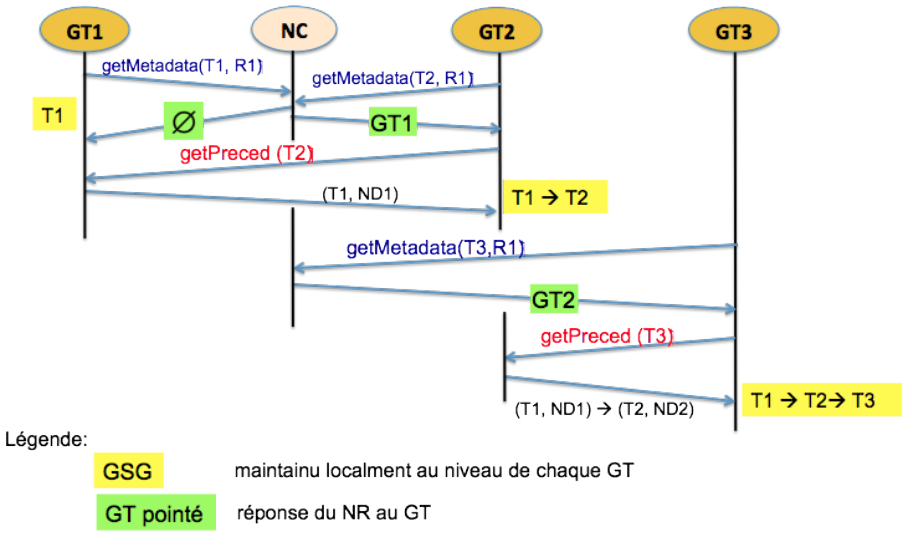

FIGURE 5.5 – Routage concurrent

voici la situation qui arrive : pour $GT1$, $Last(R_2)$ = $GT2$ alors que pour $GT2$, $Last(R_1)$=$GT1$. Ceci amène une situation de blocage des deux GT puisque chacun attend la réponse de l'autre pour pouvoir lui donner une réponse à son tour, on parle de précédence croisée. Pour éviter ce problème, les $Meta(R^i)$ sont ordonnés suivant le nom de la relation. Ainsi, le GT qui gère T récupère les $GSG(R_i)$ en respectant cet ordre. Par exemple, si T a besoin de manipuler R_1 et R_2, le GT récupère d'abord $GSG(R_1)$ via $Meta(R_1)$ avant de récupérer $GSG(R_2)$ dans $Meta(R_2)$. Le graphe globale de T sera obtenu par $GSG(T) = GSG(R_1) \cup GSG(R_2)$. Cette stratégie permet d'éviter l'occurrence des précédence croisée et donc de pourvoir construire toujours le GSG d'une transaction.

En conclusion, malgré le fait que les NC délivrent des versions obsolètes du GSG, les GT sont capable de reconstruire un GSG complet et cohérent avec une surcharge limitée par un aller-retour de communication. Le bénéfice de cette approche est qu'un GT arrive à router une transaction sans attendre que les modifications faites sur les métadonnées soient inscrites dans la DHT. Les modifications sur les métadonnées sont propagées d'un GT à un autre, donc regroupées avant d'être insérées dans la DHT. L'écriture en bloc des modifications constituent également un gain de taille. En outre, nous mentionnons que le GSG reste de taille faible puisque que toute transaction terminée 'validée sur l'ensemble des répliques) est soustraite du GSG.

5.2.3 Etude comparative des deux méthodes de gestion du catalogue

Intuitivement, la solution sans verrouillage est plus rapide que celle avec verrouillage car elle demande moins de communication avec le catalogue. Plus précisément, nous souhaitons quantifier la différence de coût entre les deux solutions. Le coût pour obtenir l'accès exclusif au GSG est exprimé en nombre de messages. Nous distinguons deux types de messages dont les coûts diffèrent. Un message GT $\rightarrow$ GT (ou NC $\rightarrow$ GT) a un coût unitaire m correspondant à une communication

directe entre deux noeuds. Un message GT $\rightarrow$ NC a un coût plus élevé car il doit effectuer plusieurs sauts avant d'atteindre sa destination. Le nombre de sauts (noté r) dépend de l'organisation des nœuds NC. La valeur de r est souvent supérieure à 2, par exemple, avec notre implémentation de JuxMem, r vaut 3, car il faut contacter un groupe local et ou un groupe global. De plus, notons que r augmente avec le nombre de nœuds gérant le catalogue. Nous détaillons le coût pour que $GT2$ obtienne l'accès exclusif lorsque $GT1$ détient le GSG.

1. Coût avec verrouillage
   - $GT1$ demande et obtient le GSG ;
   - $GT2$ demande le GSG au NC : $r * m$;
   - NC met en attente $GT2$;
   - $GT1$ renvoie le GSG modifié au NC : $r * m$;
   - NC accorde leGSG à $GT2$: m ;
   - le nombre de message total est : $C = (2r + 1)m$
2. Coût sans verrouillage
   - $GT1$ demande et obtient le GSG ;
   - $GT2$ demande le GSG au NC : $r * m$;
   - $GT2$ demande le GSG à $GT1$: m
   - $GT1$ envoie le GSG à $GT2$: m
   - le nombre de message total est : $C' = (r + 2)m$

On ne prend pas en compte dans la deuxième solution le coût de transmettre le GSG modifié au NC car cela est fait de manière découplée et peu fréquente. En faisant la comparaison des deux coûts, nous observons avec la méthode sans verrouillage, un bénéfice de $C - C' = (2r+1)m-(r+2)m = (r - 1)m$.

Ce résultat montre que si tous les NC se trouvent dans un même réseau LAN et que chaque NC connait tous les autres NC, alors les solutions avec verrouillage ou sans verrouillage sont équivalentes car $r = 1$. Par contre, quand les NC sont sur différents LAN, comme dans le cas de JuxMem, ou bien chaque NC ne connait que quelques NC (par exemple le cas des DHT), alors la gestion des métadonnées sans verrouillage est toujours meilleure puis que r est largement supérieur à un.

5.3 Conclusion

Nous avons décrit dans ce chapitre, notre approche pour gérer les traitements des transactions et le catalogue réparti en cas d'accès concurrents. Nous proposons deux approches pour sérialiser et router les transactions : une approche pessimiste basée sur les conflits potentiels et une seconde qui est plutôt hybride puisqu'il fait un premier ordonnancement des transactions en se basant sur les conflits potentiels, puis corrige cet ordonnancement en se basant sur les conflits réels. Notre seconde approche favorise plus le parallélisme puisque les transactions sont exécutées de manière optimiste, ce qui est très bénéfique dans le contexte des applications web 2.0 où les conflits sont rares. Pour maintenir la cohérence globale, nous avons conçu un catalogue pour stocker les métadonnées. Le catalogue est maintenu de tel sorte qu'il soit disponible et passant à l'échelle.

Chapitre 6

Tolérance à la dynamicité des noeuds

La dynamicité, qui peut être définie comme étant l'attitude des nœuds à joindre ou quitter volontairement ou non le système, est un aspect important à prendre en compte lors d'une conception d'un système réparti à grande échelle. Ceci est vrai pour la simple raison que la dynamicité a deux impacts négatifs dans le fonctionnement du système : (1) une diminution de la capacité du système (quand plusieurs nœuds quittent le système) et (2) des pertes d'informations quand un nœud quitte le système durant le traitement d'une tâche, ce qui peut entrainer des incohérences du système. Nous dissocions deux situations dans la gestion de la dynamicité : la déconnexion intempestive d'un nœud que nous appelons panne d'un nœud et la déconnexion prévue d'un nœud. Le problème des déconnexions prévues est assez simple à résoudre (cf. section 6.1). C'est pour cela ce chapitre se concentre sur le cas des pannes.

Dans le chapitre précédent, nous avons présenté les algorithmes de routage pour contrôler l'exécution cohérente des transactions. Cependant, ces algorithmes supposent que tous les nœuds impliqués dans l'exécution d'une transaction ne tombent pas en panne. Cette hypothèse est loin d'être plausible dans le contexte d'un système à grande échelle où les nœuds qui composent le système peuvent tomber en panne à tout moment. Pourtant, l'occurrence des pannes peut entraîner des situations d'incohérences même si le routage est fait de manière correcte. Pour illustrer le problème introduit par les pannes, supposons qu'une application de vente en ligne A envoie une transaction T pour acheter un objet. L'objet se trouve dans deux répliques R_1^i et R_2^i et il y a deux routeurs $GT1$ et $GT2$. Supposons que GT_1 reçoit T et l'envoie sur R_1^i qui l'exécute mais tombe malheureusement en panne avant d'envoyer les résultats à A. Au bout d'un moment, A n'ayant pas reçu de réponse, renvoie T. Supposons cette fois ci que T soit interceptée par $GT2$. $GT2$, n'ayant aucune connaissance de la première exécution de T sur R_1^i, route T sur R_2^i. Si l'exécution de T se déroule avec succès sur R_2^i, alors T est exécutée à deux reprises et A aurait acheté deux fois le même objet. Pour éviter ce genre de problème, il faut contrôler l'occurrence des pannes des nœuds durant l'exécution des transactions.

Ce chapitre aborde la tolérance à la dynamicité des nœuds pour l'algorithme de routage défini dans le chapitre précédent. Le système de routage que nous proposons (cf chapitre 4) est redondant : il contient plusieurs nœuds GT, et NC, et les données sont répliquées sur plusieurs nœuds ND. Il y a deux raisons à cette redondance :

1. accroitre les ressources permet de traiter les transactions plus rapidement ;

2. disposer de plusieurs nœuds identiques permet d'assurer une continuité de service malgré la panne de certains nœuds.

Cette dernière raison nécessite de concevoir un algorithme de routage capable de réagir correctement face à l'occurrence d'une panne. L'objectif principal de ce chapitre est de rendre l'algorithme de routage tolérant aux pannes. Le deuxième objectif de ce chapitre est d'étudier de manière théorique la disponibilité de notre système dans le temps : quelles conditions garantissent qu'il y a toujours au moins un nœud actif (i.e. en service) pour traiter une transaction, autrement dit, comment définir le nombre de répliques optimal pendant une période donnée afin d'avoir toujours un nœud disponible pour traiter la transaction.

Nous précisons que nous étudions la détection et la résolution des pannes afin de maintenir la cohérence et de borner le temps de réponse uniquement. Autrement dit, nous ne nous intéressons pas à restaurer les nœuds en panne.

Le reste de ce chapitre est organisé comme suit. La section 6.1 présente la gestion des déconnexions prévues. La section 6.2 décrit les mécanismes utilisés pour faire face aux pannes. La section 6.3 étudie le nombre minimal de répliques en dessous duquel, le système n'assure plus la disponibilité du système.

6.1 Gestion des déconnexion prévue

Une déconnexion prévue survient si un nœud décide volontairement de quitter le système et en informe aux GT responsable du système. Nous rappelons que nous intéressons qu'aux déconnexions des GT et des ND qui peuvent compromettre la cohérence du système.

Déconnexion prévue d'un GT

Nous supposons que deux nœuds GT adjacents (liés par la relation prédécesseur - successeur) ne quittent jamais l'anneau en même temps. Ils le font l'un à la suite de l'autre. La figure 6.1 décrit sommairement le processus de déconnexion d'un GT. Quand GT_2 qui a comme prédécesseur GT_1 et comme successeur GT_3 décide de quitter le système, il informe GT_1 de son intention de quitter l'anneau. Pour ce faire, GT_2 envoie un message *Quitte Anneau (QA)* à GT_1 en précisant qui était son successeur. Cette information permet à GT_1 de savoir qui va devenir son nouveau successeur. Par la suite, GT_2 envoie un message *Maj Anneau (MA)* à GT_3 qui, à la réception de ce message, met à jour sa vue (*i.e.* définit GT_1 comme son nouveau prédécesseur). Durant la phase de déconnexion, GT_2 ignore simplement les messages entrants et particulier les transactions entrantes.

Déconnexion prévue d'un ND

Si un ND veut se déconnecter, il envoie un message appelé *Requête Déconnexion (RD)* au dernier GT qui lui a envoyé une transaction. Supposons comme le montre la figure 6.2, GT_i est le dernier GT qui ait contacté ND_1 et GT_j l'avant dernier GT. Donc ND_1 contacte d'abord GT_i et attend pendant une période pour recevoir l'aval de ce dernier. Si ce GT_i n'arrive pas à répondre au bout de cette période, ND_1 contacte GT_j. Si toutefois le GT_j n'est pas aussi disponible, ND_1

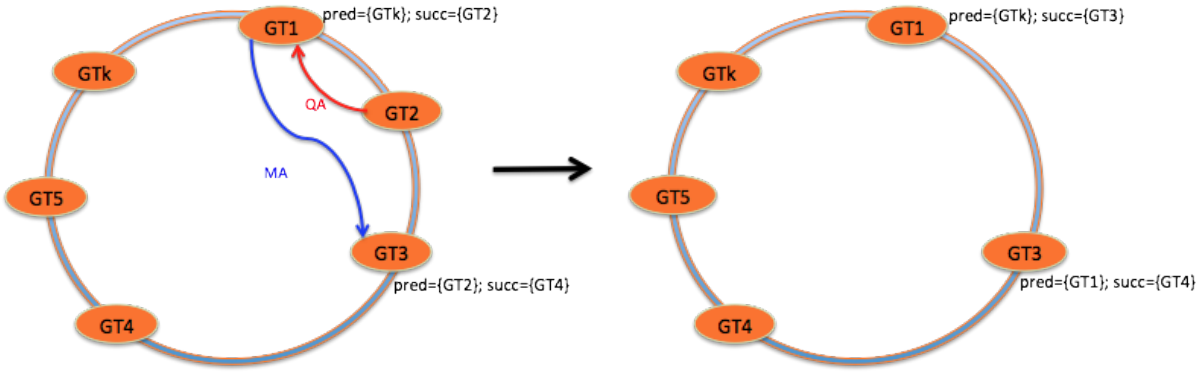

FIGURE 6.1 – Déconnexion d'un GT

contactera le GT qui l'a contacté avant GT_j, et ainsi de suite. Quand GT_j reçoit le message *RD*, il enlève ND_1 de la liste des ND disponibles pour éviter qu'un autre GT route une transaction vers ce nœud en cours de déconnexion. Par la suite, GT_j envoie à ND_1 un message appelé *Déconnexion Reçue (DR)*, ce qui permet à ce dernier de se déconnecter. Si après plusieurs tentatives de contacter un GT, ND_1 n'y arrive pas, il se déconnecte et son départ sera interprété comme une panne par tout GT qui tentera de lui envoyer une transaction implicite (cf. section suivante).

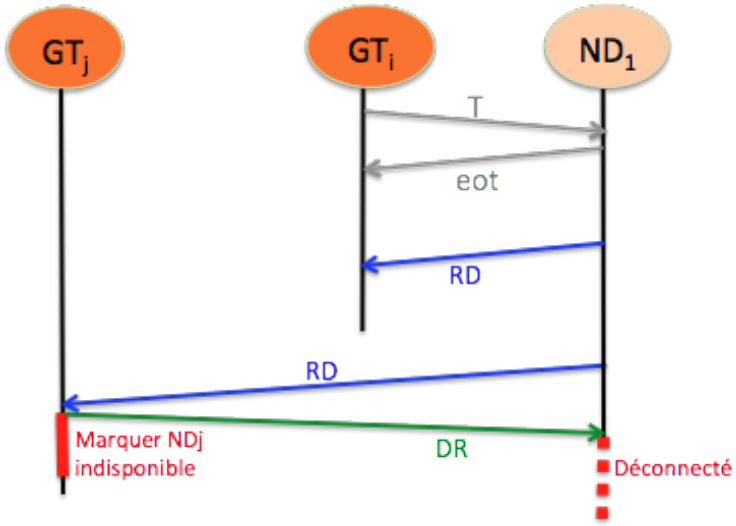

FIGURE 6.2 – Déconnexion d'un ND

6.2 Gestion des pannes

Dans cette section nous étudions les mécanismes de gestion des pannes dans le but de maintenir la cohérence des données et de borner le temps de réponse en cas de pannes des nœuds du système. Pour détailler nos protocoles de gestion des pannes, nous présentons les types de pannes que nous prenons en compte dans cette étude. Puis nous décrivons comment ces pannes sont détectées puis résolues.

6.2.1 Modèle et détection de pannes

Dans cette section, nous présentons les types de pannes que nous prenons en compte et comment elles sont détectées.

Modèle de pannes

Dans cete section, nous considérons les systèmes constitués uniquement de deux types de composants : les nœuds qui traitent les transactions (NA, ND et GT), et les canaux de communications. Chacun de ces types de composants peut tomber en panne durant le fonctionnement du système, engendrant ainsi une panne de nœud ou de communication. Les nœuds du catalogue NC, sont exclus car ils sont gérés via une DHT ou JuxMem qui ont leur propre mécanisme de gestion des pannes. Dans la suite de ce travail, nous axons notre réflexion sur les types de pannes suivants :

- Panne d'un nœud. Quand un nœud tombe en panne, ses traitements s'arrêtent anormalement, ce qui peut conduire à des incohérences. Nous supposons qu'un nœud fonctionne correctement ou s'arrête complètement (il est en panne). En d'autres mots, nous ne prenons en compte que les pannes franches et non les pannes byzantines ;
- Panne de communication. Une panne de communication survient quand un nœud N_i est incapable de contacter le nœud N_j, bien qu'aucun d'entre eux ne soit en panne. Si une telle panne survient, aucun message n'est délivré. Dans notre contexte, la communication est asynchrone et chaque message reçu par un nœud doit être acquitté. Sans cet acquittement, nous supposons que le message est perdu à cause d'une panne de communication ou d'un nœud.

Par ailleurs, chaque noeud (NA, GT, ND) qui rejoint le système est capable de contacter un noeud GT disponible. Le GT contacté est alors responsable d'inclure le nouveau nœud en mettant à jour l'anneau (arrivée d'un GT) ou le répertoire partagé (connexion d'un NA ou ND). Comme notre principal objectif est de préserver la cohérence quand un nœud quitte le système, nous restreignons notre étude aux pannes des GT et ND (si un NA quitte le système, la cohérence du système n'est nullement menacée puisque le NA délègue l'exécution des transactions aux GT). De plus, nous supposons qu'il y a toujours au moins un nœud disponible (GT ou ND) sur lequel on peut s'appuyer à chaque fois que l'on détecte une panne de nœud. Autrement dit, il n'y a pas d'occurrence simultanée de pannes de tous les nœuds du système.

Détection des pannes

En général, les pannes sont détectées soit par des messages périodiques de type *heartbeat* [ACT99], soit à la demande par des messages *ping-pong* [LAF99]. [CT96] présentent les principes de détection collaborative pour les systèmes à large échelle. Nous nous inspirons de ces travaux en combinant l'utilisation des messages *heartbeat* et *ping-pong*. En effet, nous utilisons une détection de pannes à la demande pour les ND et une détection périodique pour les GT. L'utilisation des messages périodiques pour détecter les pannes des GT se justifie par le nombre réduit de GT comparé à celui des ND, générant moins de messages. En outre la détection des pannes de ND se fait par collaboration entre les GT, d'où l'importance de détecter le plus tôt possible une panne de GT. Par contre l'utilisation de cette technique pour détecter les pannes des ND engendrerait beaucoup de messages du fait de leur nombre important. Ainsi, pour détecter les pannes des ND sans surcoût significatif, on intègre la détection dans le protocole de routage utilisant la méthode *ping-pong*. De ce fait, un ND en panne n'est détecté que si un GT essaie de lui envoyer une transaction. Pour prendre en compte des pannes survenant lors de l'accès au répertoire réparti, nous nous appuyons soit sur JuxMem qui empêche qu'un lecteur (ou écrivain) en panne verrouille l'accès aux données indéfiniment, soit sur la DHT qui offre des primitives de récupération ou de localisation de ressources même en présence de pannes, autrement dit, elle a ses propres mécanismes de gestion de panne.

Les déconnexions prévues n'ont pas besoin d'être détectées car elles sont déclarées avant leur occurrence par les nœuds qui l'expérimentent. Nous présentons d'abord les mécanismes de gestion des déconnexions prévues avant de présenter ceux des pannes.

6.2.2 Tolérance aux pannes

Dans l'optique de gérer les pannes, nous reprenons notre protocole de routage décrit dans 5.1.2 en faisant abstraction de l'accès aux métadonnées. En effet l'accès aux métadonnées est considéré dans cette section comme étant une action interne d'un GT. Ce choix découle du fait que nous déléguons la gestion des pannes des nœuds stockant les métadonnées à JuxMem ou à la DHT. Le protocole de routage peut être découpé en trois phases. La figure 6.3 représente le déroulement du processus d'exécution de la transaction T en absence de pannes.

1. *phase d'initialisation* : Durant cette phase, un NA envoie T à un GT ;
2. *phase de routage* : Le GT exécute un des algorithmes de routage (voir section 5.1.4 et section 5.1.3) et envoie ainsi la transaction à un ND ;
3. *phase d'exécution* : Pendant cette phase, un ND reçoit une transaction T, l'exécute localement et envoie le résultat au NA qui avait initié la transaction. Il informe aussi le GT qui l'a contacté que T a été correctement exécutée.

Dans la suite, nous utilisons les noms de messages définis dans la figure 6.3. A chacune des phases de notre protocole, des pannes peuvent survenir, empêchant alors l'exécution correcte des tâches qui sont allouées aux GT et ND participant à l'exécution d'une transaction. Suivant la phase en cours, un nœud (NA, GT ou ND) peut détecter la panne d'un nœud participant à l'exécution

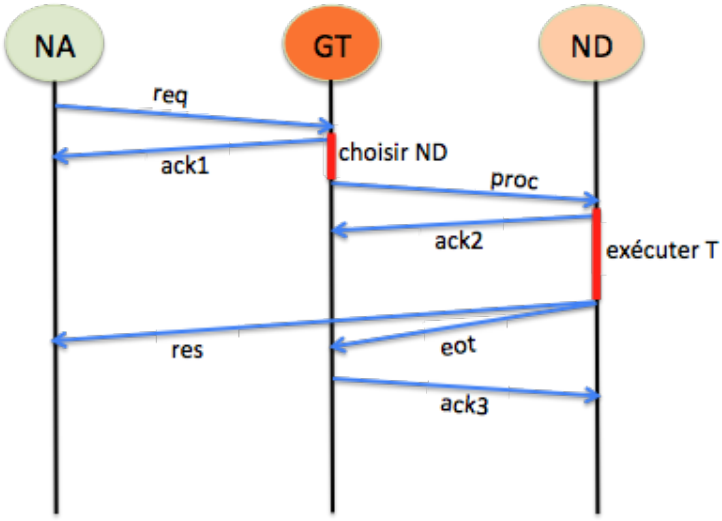

FIGURE 6.3 – Les phases d'exécution d'une transaction

de la transaction et donc essayer de la résoudre. Ainsi, lors de la première phase, un NA peut expérimenter la panne d'un GT et tente de résoudre celle-ci, on parle alors de gestion des pannes faite par le NA. De même durant la phase de routage ou d'exécution, un GT peut détecter la panne d'un ND qu'il résoudra : il s'agit de la gestion des pannes faites par le GT. Enfin, durant la phase d'exécution, un ND peut détecter la panne d'un GT ou d'un NA et on parle de gestion faite par un ND. Dans les trois prochaines sections nous présentons comment un nœud arrive à détecter ou à suspecter une panne et comment il le prend en compte.

Gestion des pannes faite par le NA

Un NA envoie un message *req* à un GT et initialise ensuite un temporisateur δ_a (cf. figure 6.4). Quand δ_a expire, le NA conclut que le message *req* ou l'acquittement *ack1* est perdu à cause d'une panne de communication ou du GT contacté. Alors, NA retransmet le message *req* avec le même identifiant global. Pour accroître les chances de réussite de la retransmission, le NA incrémente le nombre de GT cibles. Plus précisément, le NA ajoute un GT de plus sur la liste des destinataires à chaque fois qu'il retransmet *req*. Les GT candidats sont choisis parmi les GT connus par le NA en utilisant l'algorithme du tourniquet. Remarquons que la cohérence ne peut être compromise puisque la transaction n'est transmise à aucun ND pour exécution. Même si plusieurs GT reçoivent la même transaction, cette dernière n'est transmise qu'à un seul ND grâce à l'utilisation de l'identifiant global et de l'accès exclusif au répertoire partagé.

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

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

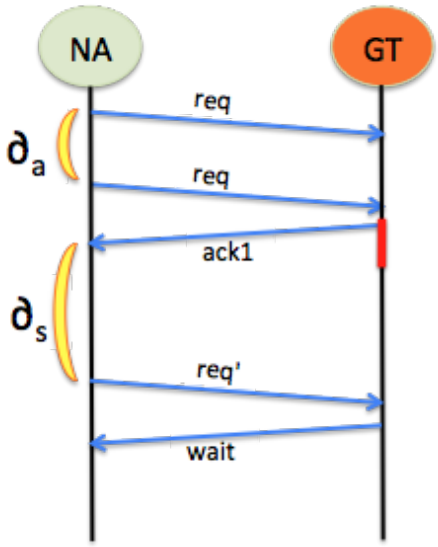

FIGURE 6.4 – Comportement du NA en fonction des temporisateur

Gestion des pannes faite par le GT

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

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

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

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

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

A chaque évaluation de l'algorithme de routage, le GT garde la liste des ND candidats triés suivant l'ordre croissant du coût. Par la suite, le GT envoie le message *proc* au premier candidat ND_i, et initialise un temporisateur δ_a. Une fois que le message *ack2* est reçu, le GT inscrit dans le

répertoire partagé que T est en cours d'exécution. A partir de ce moment, il initialise un second temporisateur δ_r.

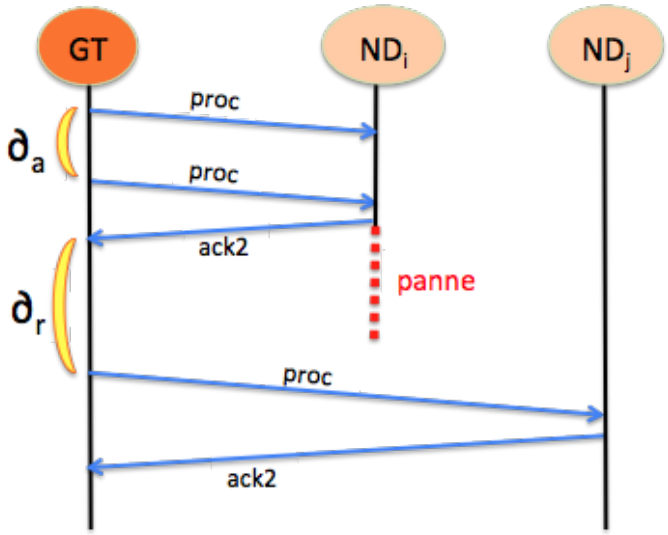

FIGURE 6.5 – Comportement du GT en fonction des temporisateur

Quand δ_a et δ_r expirent, le GT conclut qu'une panne de communication ou du ND est survenue. Il envoie alors le message *proc* au prochain candidat ND_j sur la liste (cf. figure 6.5). En parallèle, il invoque le module de détection de pannes (cf. figure 4.6) qui vérifie si ND_i est disponible ou non. Pour cela, il contacte son prédécesseur et son successeur et chacun d'entre eux essaie d'entrer en contact avec ND_i en lui envoyant un message. Les résultats de ces échanges sont envoyés au nœud initial. Si tous les résultats sont négatifs, il conclut que ND_i est en panne et l'ajoute dans la file des nœuds en panne, appelée *FNP* et stockée dans le répertoire partagé. Par contre, si au moins un des résultats est positif, le GT suppose qu'il y a une panne de communication temporaire entre lui et ND_i. Il peut donc considérer ND_i comme un candidat potentiel lors des prochains routages.

Le GT utilise également cette même procédure quand un nœud ND voulant valider une transaction T le contacte à l'expiration du temps de décision Δ_{TD} (cf. section 13).

Quelle que soit la cause de ce retard, le GT vérifie que les transactions correspondantes aux $DataSet(T_i)$ manquants sont déjà exécutées en consultant le catalogue réparti.

Dans l'affirmative, il essaie de récupérer les $DataSet(T_i)$ manquants et de faire la vérification nécessaire. Si malgré cette tentative, il n'arrive pas à récupérer tous les $DataSet(T_i)$ manquants ou même que les transactions correspondantes ne sont pas encore exécutées, alors le GT envoie directement au ND les transactions manquantes en tenant compte des vérifications positives faites avec $DataSet(T_k)$ qui ont pu être récupérés. Précisément, si avec les $DataSet(T_k)$ déjà reçus, le GT détecte des conflits il demande au ND de reprendre la transaction T. Par contre, s'il existe des transactions qui ne sont pas réellement en conflit avec T, le GT avise le ND en lui donnant une nouvelle séquence de contraintes de précédence.

Si un nœud ND_j exécutant une transaction T_i qui précède T n'arrive pas à être joint par le GT,

ce dernier lance le module de détection de pannes.

Le gestionnaire des pannes, appelé module de reprise sur panne (cf. figure 4.6), est chargé de vérifier la reprise de chaque nœud en panne et de l'enlever de la file *FNP* comme décrit dans [CT96].

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

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

Gestion des pannes faite par le ND

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

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

6.2.3 Majoration du temps de réponse

Comme décrit précédemment, notre protocole de routage se termine malgré la présence de pannes. Néanmoins, le délai pour exécuter totalement une transaction augmente proportionnellement avec le nombre de pannes. Plus le nombre de retransmissions nécessaires pour exécuter une transaction est grand, plus le temps d'exécution s'élève. Pour démontrer l'efficacité de notre approche, nous montrons que le nombre d'essais requis pour exécuter une transaction est souvent faible. Dans cette perspective, nous notons $avg(T)$, le temps moyen d'exécution d'une transaction, λ_N, la latence moyenne du réseau, λ_D, le temps d'accès moyen au répertoire partagé et $\bar{S}$, la taille de la séquence de rafraîchissement. En absence de toute panne, le temps nécessaire pour exécuter T est :

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

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

requises lors de l'exécution de T, alors le temps total d'exécution de T est :

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

Supposons que p est la probabilité qu'une transaction tombe en panne (*i.e.* NC n'a reçu aucun résultat) et X une variable aléatoire représentant le nombre de tentatives.

$P(X = i) = (1-p) * (p)^{i-1}$ est la probabilité que les *(i-1)* premières tentatives ont échoué et que la i^{me} a réussi. Le nombre de tentatives exécutées est obtenu avec la formule suivante :

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

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

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

Par conséquent, nous pouvons estimer le nombre de tentatives : $k \approx \lceil \frac{1}{1-p} \rceil$. Par exemple, k=2 pour une probabilité de panne inférieure à 50 %. A partir de ces résultats, nous concluons que le nombre de tentatives d'exécuter une transaction est borné et il est faible.

6.3 Gestion contrôlée de la disponibilité

Dans la section précédente, nous avons décrit les mécanismes pour faire face aux pannes des nœuds lors d'un traitement d'une transaction. Cet algorithme suppose qu'il existe toujours un nœud sur lequel une transaction peut être reprise. Nous allons étudier comment peut on garantir cette hypothèse en contrôlant le degré de la réplication. Comme nous l'avons spécifié dans la section 2.3.3, la réplication permet de masquer les pannes, puisqu'à chaque fois qu'un nœud tombe en panne, un autre nœud peut être utilisé pour poursuivre les traitements que le nœud en panne effectuait. Cela suppose qu'il existe au moins un nœud sur le système capable de faire le traitement du nœud en panne, ce qui nous amène à la définition suivante.

Définition 14. *Un système est dit disponible, s'il existe au moins un nœud non en panne (appelé remplaçant) qui peut continuer à faire les traitements que le nœud en panne faisait.*

S'il s'agit d'un nœud ND, le nœud remplaçant doit être capable de faire les transactions du nœud en panne et s'il s'agit d'un GT, il doit être capable de router les transactions.

Avec cette définition, il apparaît que pour rendre un système disponible il suffit de garantir la présence d'au moins un nœud remplaçant à tout moment pendant une période fixée.

Dans cette section nous présentons un modèle qui nous permet de savoir le nombre de répliques minimal pour garder le système disponible pendant un intervalle de temps. Pour ce faire, nous proposons une approche basée sur la probabilité (ou fréquence) des pannes. En effet, connaissant la

probabilité d'occurrence des pannes, nous définissons le nombre minimal de répliques nécessaires pour rendre le système disponible pendant un intervalle de temps. Pour ce faire, nous supposons que les pannes sont franches (cf. section 6.2.1).

Le calcul de la probabilité de pannes dans un système distribué est un problème résolu de longue date. Dans [OV99], les auteurs arguent que l'occurrence des pannes suit une distribution de Poisson. De cette hypothèse, la probabilité des pannes est donnée par :

$$P_k = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \tag{6.1}$$

P_k représente la probabilité que k pannes surviennent durant l'intervalle de temps t et λ le taux d'occurrence des pannes pendant t.

Partant de cette probabilité, nous déterminons le nombre minimal de répliques nécessaire pour que l'occurrence de k pannes ne rend pas indisponible le système. Pour ce faire, supposons λ, le taux d'arrivée des pannes, P_{tol}, la probabilité de panne des nœuds que l'on veut tolérer pendant l'intervalle de temps t (*i.e.* le nombre de pannes qui ne doit pas compromettre la disponibilité). Pour assurer la disponibilité durant t, nous avons besoin de définir K le nombre de répliques tel que $P_K > P_{tol}$.

Soit m tel que : $\forall m < K, P_K < P_{tol} \leq P_m$.
Pour $m = 2, P_2 \geq P_{tol}$ et nous pouvons obtenir par addition de termes $P_1 + 2P_2 \geq P_{tol} + 2P_{tol}$.
Pour $m = 3$, $P_3 \geq P_{tol}$ et $P_1 + 2P_2 + 3P_3 \geq P_{tol} + 2P_{tol} + 3P_{tol}$.
En continuant ce raisonnement jusqu'au rang m, nous obtenons l'inégalité suivante dont la partie droite est une suite arithmétique puisque P_{tol} est constante.

$$P_1 + 2P_2 + ... + mP_m \geq P_{tol} + 2P_{tol} + ... + mP_{tol} \tag{6.2}$$

En faisant le calcul de la somme de la suite arithmétique, nous obtenons :

$$\sum_{n=1}^{m} nP_n \geq \frac{m(m+1)}{2} P_{tol} \tag{6.3}$$

Ensuite, en remplaçant P_n par sa valeur dans 6.3, nous obtenons :

$$e^{-\lambda t} \sum_{n=1}^{m} \frac{(\lambda t)^n}{(n-1)!} \geq \frac{m(m+1)}{2} P_{tol} \tag{6.4}$$

L'ajout de quelques termes positifs dans la partie gauche de l'équation 6.4 ne change pas le sens de l'inégalité et donne :

$$\lambda t.e^{-\lambda t}[\sum_{n=0}^{m-1} \frac{(\lambda t)^n}{(n)!} + \sum_{n=m}^{\infty} \frac{(\lambda t)^n}{(n)!}] \geq \frac{m(m+1)}{2} P_{tol} \tag{6.5}$$

Ceci est équivalent à :

$$\lambda t.e^{-\lambda t} \sum_{n=0}^{\infty} \frac{(\lambda t)^n}{(n)!} \geq \frac{m(m+1)}{2} P_{tol} \tag{6.6}$$

Par ailleurs, comme $\sum_{n=0}^{\infty} \frac{(\lambda t)^n}{(n)!} = e^{\lambda t}$ alors, l'inégalité 6.6 devient :

$$\lambda t \geq \frac{m(m+1)}{2} P_{tol} \tag{6.7}$$

En posant m = K - 1, nous obtenons l'inéquation suivante $m^2 + m - \frac{2\lambda t}{P_{tol}} \leq 0$ dont la résolution aboutit à :

$$K > \lceil \sqrt{\frac{1}{4} + \frac{2\lambda t}{P_{tol}}} - \frac{1}{2} \rceil \tag{6.8}$$

La formule 6.8 permet de déterminer le nombre de réplique suffisant pour tolérer une probabilité de pannes P_{tol} dont le taux d'occurrence des pannes est λ. Par exemple, soit un système avec un taux d'arrivée des pannes égal à 0.005 ($\lambda = 0.005$) et un intervalle de temps t égal à 1000 secondes. Nous calculons le nombre de pannes moyen durant t par $\lambda * t = 5$. Par conséquent, en utilisant la formule 6.8, le nombre de répliques est $K > \lceil 7.06575 \rceil$. Intuitivement, il est clair qu'avec 7 répliques, il existe au moins un nœud disponible durant t malgré l'occurrence de 5 pannes.

6.4 Conclusion

Dans ce chapitre, nous avons présenté un mécanisme de gestion de la dynamicité. Ce mécanisme est basé sur la détection sélective des fautes et sur un algorithme de reprise. Contrairement à la plupart des autres approches, notre mécanisme n'implique pas l'utilisation de nœuds qui ne participent pas à l'exécution de la transaction en cours, ce qui le permet de passer à l'échelle. Pour cela, nous adaptons des approches existantes de détection des pannes afin de les rendre opérationnelles pour chaque type de nœud (gestionnaire de transaction et nœud de données) de notre système. Nous avons proposé un protocole permettant de gérer toutes les situations lorsqu'un nœud quitte le système pendant le traitement d'une transaction. Ceci est nécessaire et suffisant pour contrôler la cohérence du système, surtout en cas de déconnexions intempestives.

Cependant, pour garder le débit transactionnel constant en cas de fréquentes pannes, il faut être capable d'ajouter de nouvelles ressources en fonction des déconnexions. Pour ce faire, nous avons proposé un modèle pour déterminer et contrôler le nombre de répliques requises pour garder le système disponible. Autrement dit, ce modèle permet de déterminer le nombre minimum de répliques nécessaires au bon fonctionnement du système et donc de minimiser les surcoûts liés à la gestion des répliques. Cette étude nécessite d'être complétée pour déterminer la manière d'ajouter ou de réduire le nombre de répliques en fonction de l'évolution de la charge du système ou de sa dynamicité.

Chapitre 7

Validation

Pour valider la faisabilité de nos approches, nous avons implémenté et expérimenté deux prototypes nommés respectivement DTR (*Distributed Transaction Routing*) et TRANSPEER (*TRANSaction on PEER-to-peer*).

Puis, nous avons effectué des simulations pour étudier le passage à l'échelle et la tolérance aux pannes de notre solution. Nous mentionnons que l'implémentation de deux prototypes est liée au besoin de gérer le catalogue avec verrouillage ou sans verrouillage. De fait, DTR constitue le prototype développé avec verrouillage en s'appuyant JuxMem. TRANSPEER est conçu pour la gestion du catalogue sans verrouillage et pour un modèle de communication de type P2P entre les composants du routeur et s'appuie sur FreePastry [Fre]. De plus, notre algorithme de routage pessimiste est implémenté dans DTR alors que l'approche hybride l'est avec TRANSPEER.

De plus, notre choix d'utiliser à la fois de l'expérimentation et de la simulation se justifie par le fait que : (1) l'expérimentation permet d'évaluer un système dans des conditions réelles ; et (2) la simulation est une représentation simplifiée du système, facile à réaliser et requiert moins de ressources que l'implémentation, ce qui favorise l'évaluation d'un système à grande échelle. Nous avons mené une série d'expériences sur nos deux prototypes pour étudier les performances de notre système : débit transactionnel, temps de réponse, passage à l'échelle et tolérance aux pannes.

Notons dores et déjà que nous n'avons fait qu'une validation partielle du passage à l'échelle de notre solution, autrement dit, notre solution n'assure pas un grand passage à l'échelle. Néanmoins les expériences effectuées nous ont permis de savoir les impacts de nos différents choix sur le passage à l'échelle. Nous décrivons dans le section 8.2 (Perspectives), comment nous comptons assurer ce passage à l'échelle.

Pour ce faire, nous étudions d'abord la surcharge liée à la gestion du catalogue dans la section 7.1, puis la section 7.2 les performances globales du routage et enfin les apports de la gestion des pannes sont décrits dans la section 7.3.

7.1 Evaluation de la gestion du catalogue

Dans cette section nous évaluons les performances de la gestion du catalogue. Nous vérifions que l'accès au catalogue lors du routage n'est pas une source de congestion, *i.e.* il ne ralentit pas le

Charge applicative	nombre de NA ($10 \rightarrow 80$)
GT concurrents	Nombre de GT en concurrence ($1 \rightarrow 2$)
Taux de conflit	Nb total d'accès concurrents / Nb total d'accès ($0\% \rightarrow 100\%$)
Granularité	Relation
Degré de réplication	Nombre de répliques d'une relation R^i

TABLE 7.1 – Paramètres d'évaluation de l'accès au catalogue

routage des transactions.

Pour atteindre ces objectifs, nous utilisons chacun de nos deux prototypes pour mener des séries d'expériences. Pour chaque expérience, nous mesurons le débit du routage en faisant varier les paramètres de notre système qui sont regroupés dans le tableau 7.1. Nous terminons par une analyse de nos approches de gestion du catalogue, en nous plaçant dans le contexte des applications Web 2.0.

La charge applicative est constituée aléatoirement de requêtes et de transactions envoyées par les NA. Un NA envoie une transaction puis attend la réponse avant d'envoyer une autre. Tous les NA ont la même priorité *i.e.* leurs transactions sont traitées sans tenir compte du type de l'application. Toutes les transactions ont la même granularité (Relation) et accèdent aux données de manière aléatoire. La base de données est répartie sur les ND de sorte qu'une transaction s'exécute sur un ND. Le taux de conflit, noté τ_C est défini par le nombre total de transactions en conflit potentiel sur le nombre total de transactions courantes. Soit T_T l'ensemble des transactions en cours et T_C, l'ensemble des transactions en conflits, alors $\tau_C = \frac{|T_C|}{|T_T|}$.

7.1.1 Surcharge de la gestion du catalogue dans DTR

Cette section présente une série d'expériences destinées à valider le modèle de gestion des métadonnées avec verrouillage présenté dans la section 4.3.2. Ces résultats ont fait l'objet de publications en 2008 dans la conférence nationale BDA [SNG08b] et dans le workshop international HPDGRID [SNG08a] et en 2010, dans la revue nationale RSTI -ISI [SNG10a].

Environnement expérimental

DTR est implémenté avec le langage C et s'appuie sur les services de JuxMem, qui est conçu au-dessus de la plate-forme JXTA de Sun. JuxMem fournit des accès à une mémoire virtuelle partagée à travers une grille. Néanmoins, nous remarquons que notre système est faiblement dépendant de JuxMem, puisque JuxMem est utilisé comme une API (une librairie). Nous pourrions donc utiliser tout logiciel qui fournit une API d'accès à une mémoire virtuelle partagée.

Nous avons effectué toutes les expériences sur un cluster de 20 nœuds sur lesquels s'exécutent les GT, les NC et les ND avec Postgresql comme SGBD sous-jacent pour stocker les données. Nous avons utilisé les ordinateurs personnels des membres du laboratoire pour faire tourner les NA. Ceci constitue au total un environnement de 40 machines physiques. Tous les nœuds (P4, 3GHz, 2Gb RAM) sont interconnectés par un réseau local Fast-Ethernet 1 Gbit/s.

Impact de l'accès au catalogue partagé

Les premières expérimentations s'intéressent au routage proprement dit. Elles mesurent la surcharge engendrée par l'utilisation d'un répertoire partagé pour stocker les métadonnées. La charge applicative est générée par un nombre croissant d'applications, chacune d'entre elles ne peut pas envoyer plus d'une transaction par seconde à un routeur. Nous mesurons le débit (en transactions / seconde) qu'un routeur peut assurer. La figure 7.1 montre que chacun des routeurs peut traiter jusqu'à 40 transactions/seconde. Par ailleurs, la figure montre qu'au delà de 40 applications (ou 40 transactions par secondes) le débit du routage reste constant. Cela est dû par le fait que chaque processus de routage requiert un accès au catalogue et un calcul du nœud optimal qui est dure environ 24 millisecondes. Par conséquent, au delà d'une charge applicative 40 transactions par seconde, le routeur n'est plus capable de les traiter toutes en moins d'une seconde. Cependant, les résultats montrent que le temps d'accès au catalogue est faible, autrement dit, le temps d'accès au catalogue est acceptable.

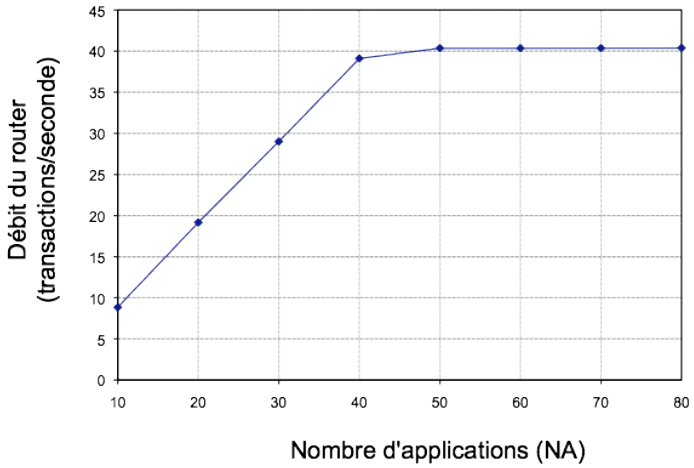

FIGURE 7.1 – Débit d'un seul routeur avec DTR

Pour confirmer ce résultat et évaluer davantage le coût de cet accès, nous augmentons la taille du répertoire en ajoutant de nouvelles répliques, puisque plus de répliques (ND) impliquent plus de métadonnées et donc plus de temps pour les récupérer et les manipuler. De plus, un grand nombre de réplique requiert plus de temps pour déterminer le nœud optimal. Nous reportons dans la figure 7.2, le débit traité dans trois situations : petite, moyenne et grande taille du répertoire (respectivement 5, 50, 100 répliques). Nous mesurons une baisse des performances inférieure à 20 % pour une grande quantité de métadonnées (100 répliques). Pour un degré de réplication moyen (par exemple 50 répliques), la baisse est uniquement de 5 %. Ceci montre que le répertoire pénalise peu les performances. Ce résultat est une conséquence de notre choix architectural de fragmenter les métadonnées et de ne garder dans le catalogue que les transactions non encore propagées sur tous les ND. De plus, ce résultat est important si on se réfère à notre contexte où le nombre de

transactions est si important que les garder toutes nécessiterait un espace de stockage considérable pouvant ralentir la recherche d'informations dans le catalogue.

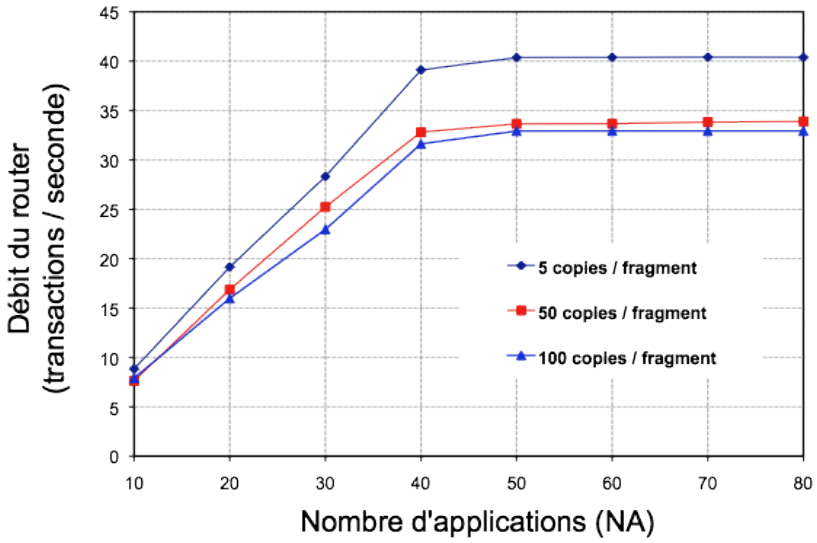

FIGURE 7.2 – Surcharge du répertoire

Impact de l'accès concurrent

Nous étudions l'impact de l'accès concurrent de plusieurs routeurs au répertoire partagé. La charge applicative est générée de la même manière que les premières expérimentations, cependant les transactions sont envoyées à 2 routeurs de telle sorte que la moitié de la charge va sur chacun des nœuds. Les résultats de la figure 7.3 sont obtenus dans le pire des cas (*i.e.* toutes les transactions accèdent à la même relation, conduisant les routeurs à accéder au même Meta(R)). Ces résultats montrent un débit maximal de 21 transactions/seconde, c'est-à-dire la moitié d'un seul routeur. En effet, l'attente d'un verrou dégrade considérablement les performances. Néanmoins, dans le pire des cas où chaque accès au répertoire est retardé par un autre accès concurrent sur la même donnée, le routeur est encore capable de fournir de bons résultats. Bien que ce résultat soit acceptable, nous tentons de l'améliorer en réduisant l'attente au niveau de l'accès aux métadonnées par suppression de l'utilisation des verrous. Pour ce faire, nous utilisons à la section suivante notre second prototype et faisons varier le degré de concurrence entre 0 % et 100 % afin de mesurer réellement l'impact de l'attente au niveau du catalogue.

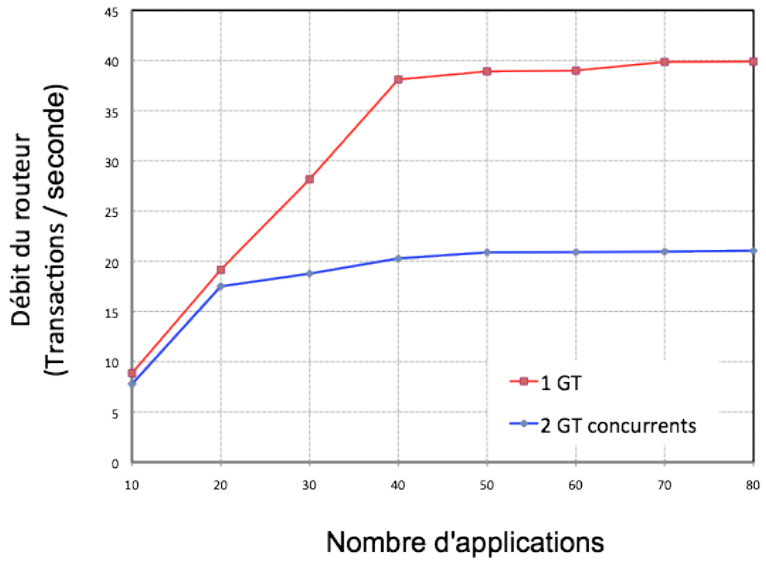

FIGURE 7.3 – Accès concurrent avec DTR

7.1.2 Surcharge de la gestion du catalogue dans TRANSPEER

Cette section présente une série d'expériences destinées à valider la gestion des métadonnées avec une DHT. Ces résultats ont fait l'objet de publications en 2010 dans la conférence internationale SAC [SNG10b].

Environnement expérimental

TRANSPEER est implémenté avec Java 1.6 (5000 lignes de code), et peut tourner sur n'importe quel système qui supporte la machine virtuelle JVM 1.6. Chaque composant est développé comme une seule application Java et par conséquent on peut le répliquer autant de fois que l'on souhaite. La couche de communication P2P entre les nœuds est conçu avec la version libre de Pastry [RD01a] à savoir FreePastry. Pour stocker les métadonnées, nous utilisons la DHT PAST[RD01b]. Nous utilisons PostgreSQL comme SGBD pour stocker les données, puis nous passons par JDBC et des règles actives (triggers) pour extraire les modifications faites par les transactions. Les données sont partitionnées dans des relations et chaque relation est répliquée sur plus de la moitié des ND. Nous effectuons nos expériences dans un environnement réel composé de 20 machines (P4, 2.4GHz, 2Gb RAM) connectées par un réseau qui supporte 100 autres machines en même temps. Les expériences sont faites pendant que les autres machines sont utilisées afin de se placer dans des conditions plus réalistes.

Impact de l'accès au catalogue partagé

Comme avec DTR, nous évaluons la surcharge lié à l'utilisation du catalogue pour router les transactions. La charge applicative est envoyée par un nombre croissant de NA ; chacun envoie une transaction par seconde à un routeur. Nous mesurons le débit (en transactions / seconde) que le routeur peut assurer. La figure 7.4 montre qu'un seul routeur de TRANSPEER peut exécuter plus de 50 transactions par seconde, ce qui représente 25% de plus que le débit d'un routeur de DTR. Cette amélioration du débit du routage découle de la suppression des verrous lors de l'accès au catalogue. En effet, pour écrire sur le catalogue via JuxMem, un GT (client de JuxMem) initie toujours une communication avec les leaders du groupe de données sollicitées (*Local Data Group* si l'accès local est possible ou *Global Data Group* dans le cas ou la donnée est répliquée sur des clusters distants). Cette communication n'est rien d'autre qu'une forme de synchronisation entre les leaders des différents sites sur lesquels la donnée est répliquée afin de fournir un verrou exclusif au GT. Par contre, avec TRANSPEER, le GT contacte directement le NC maître pour récupérer directement les métadonnées, ce qui réduit le temps de routage et donc augmente le débit transactionnel.

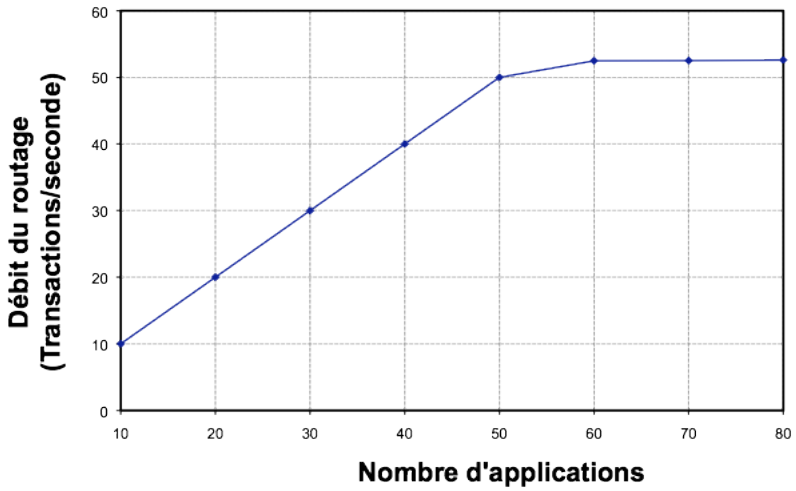

FIGURE 7.4 – Débit d'un seul routeur avec TransPeer

Impact de l'accès concurrent

En restant dans les mêmes conditions d'expérimentation, nous évaluons l'impact de l'accès concurrent de plusieurs routeurs. La charge applicative est envoyée à deux GT de tel sorte que chacun reçoit la moitié des transactions. Dans la figure 7.5, nous observons que le débit du routage maximal est de 32 transactions/seconde quand le taux de conflit est de 100 %. Cependant, contrairement à DTR, la dégradation du débit n'est que de 40% par rapport à un seul routeur. Cette diminution s'explique par le fait que lors des accès au catalogue, les GT contactent d'abord la DHT,

puis s'échangent des informations pour retrouver le GSG complet . De plus, cette échange d'informations entre GT est aussi source d'une surcharge qui est une conséquence directe de l'accès concurrent au catalogue. En effet, puisque nous avons utilisé Pastry [RD01a] pour implémenter TRANSPEER, alors le coût d'accès à la DHT est de $\log_{2^b}(N)$, avec N le nombre de nœud NC. Ainsi, pour our construire le GSG le plus cohérent, le nombre de messages envoyés par GT_1 est de $2^{m+1} + \log_{2^b}(N)$ avec m le nombre de GT qui ont précédé GT_1. Précisément, l'opération $get(k)$ coûte 2 messages alors que les échanges avec les GT précédant GT_1 correspondent à 2^m messages. Cependant comme nous l'avons décrit dans le chapitre 5, le GT ne contacte que le dernier GT parmi tous les GT qui l'ont précédé, ce qui fait que le coût devient $2^2 + \log_{2^b}(N)$. Ce coût est indépendant du nombre de GT et donc du nombre de transactions courantes, raison pour laquelle nous confirmons que la surcharge de la gestion du catalogue sans verrouillage est négligeable et qu'elle surpasse celle avec verrouillage en favorisant un débit plus grand.

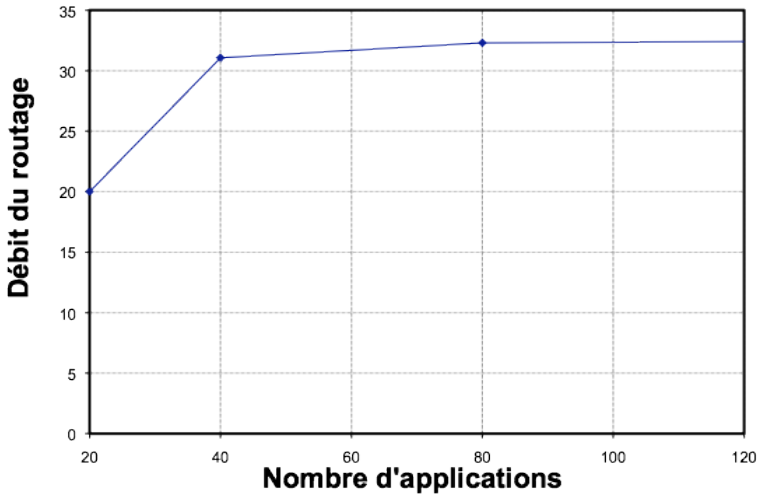

FIGURE 7.5 – Accès concurrent avec TransPeer

7.1.3 Analyse de la surcharge du catalogue

Les résultats présentés dans les deux sections précédentes montrent que l'utilisation du catalogue lors du routage ne crée pas trop de surcharge. Ce résultat est d'autant plus vrai que dans le cas d'accès concurrent le débit minimal est de 1260 transactions/minute pour un seul routeur (avec 2 GT). Ce débit minimal est obtenu dans le pire des cas (*i.e.* quand le taux de conflit est à 100% de conflit). Pourtant, un taux de conflit à 100% est peu probable. De plus, même si les transactions touchent les mêmes données, l'accès aux métadonnées ne se fait pas toujours simultanément à cause de la latence du réseau et de la position des GT par rapports aux NC dans le système. En d'autres termes, deux transactions en conflits peuvent être interceptées par deux GT qui à leur tour n'accéderont pas simultanément aux métadonnées, mais plutôt de manière séquentielle.

Pour vérifier cette affirmation, nous avons mené une série d'expérience avec TRANSPEER pour mesurer le taux de conflit réel au niveau des métadonnées. Nous voulons savoir si deux transactions conflictuelles envoyées à deux GT au même moment engendrent toujours un accès concurrent au niveau du catalogue. Pour ce faire, nous utilisons une charge applicative générée par deux NA et envoyée à deux GT. Nous avons fait varier le taux de conflit au niveau des NA, *i.e.* taux de conflits des transactions τ_C envoyées par les NA, de 0% à 100%. Puis nous avons mesuré le taux de conflit observé au niveau du catalogue. Le taux de conflit au niveau du catalogue τ_{AC}, est défini comme étant le nombre total de GT qui sollicite un même $Meta(R)$, N_C sur l'ensemble des GT, N_T qui ont accédé au catalogue, autrement dit, $\tau_{AC} = \frac{|N_C|}{|N_T|}$.

Les résultats de la figure 7.6 montrent que même si le taux de conflits τ_C est de 100%, le taux de conflits τ_C n'est que n'est que de 70%. Par conséquent, les conflits aux niveau des transactions n'entraînent pas forcément des conflits au niveau du catalogue, ce qui confirme notre précédente affirmation sur les conflits réels au niveau du catalogue.

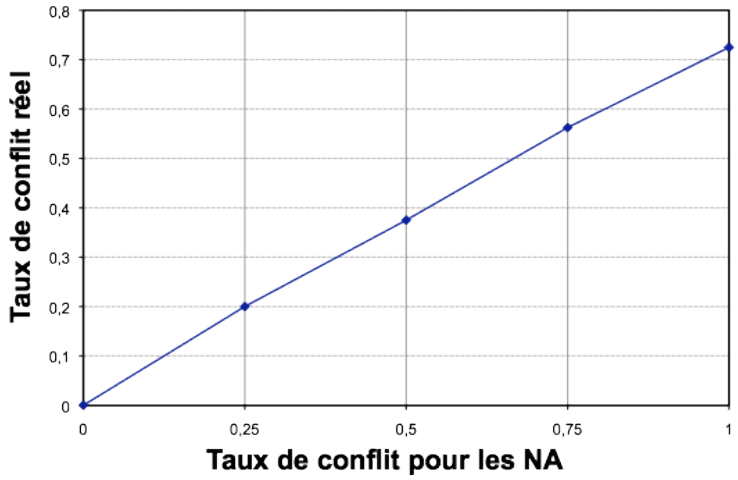

FIGURE 7.6 – Débit d'un seul routeur avec TransPeer

Parallèlement nous étudions la diminution du débit de routage quand le taux de conflit des NA augmente. Les résultats de la figure 7.7 montrent une diminution faible et progressive du débit de routage quand le taux de conflit varie. Par exemple avec un taux de conflit de 25%, la débit n'a baissé que de 5%. Cette diminution progressive est une conséquence directe de la rapidité de notre processus de routage qui fait que les GT libèrent très vite l'accès au catalogue.

De plus, dans le contexte des applications Web 2.0 où les utilisateurs ne modifient que leurs propres données, les taux de conflits sont encore plus faibles. Ceci nous pousse à affirmer que notre solution de gestion de catalogue, qui a un faible impact sur le débit du routage, s'adapte bien aux applications Web2.0 d'autant plus que l'utilisation d'un catalogue dans de tels systèmes permet de contrôler la cohérence des données mais aussi l'état du système.

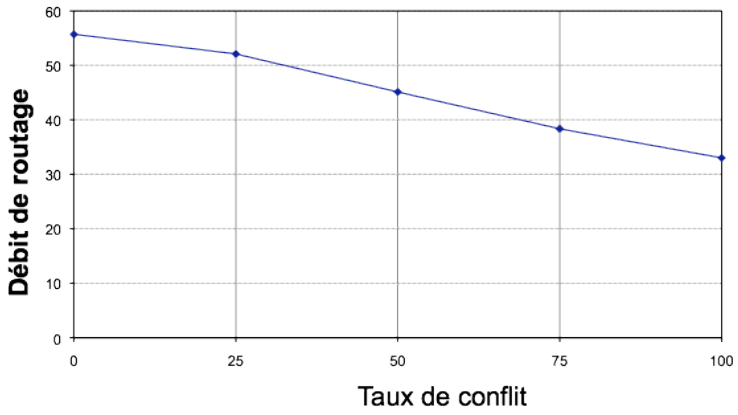

FIGURE 7.7 – Accès Concurrent avec TransPeer

7.2 Evaluation des performances globales du routage

Dans cette section nous évaluons les performances de notre routage. Comme nous prenons en compte la fraîcheur lors de notre processus de routage nous étudions très brièvement l'impact du relâchement de la fraîcheur dans notre système. Nous présentons ensuite les performances de notre processus de routage et enfin, nous concluons cette section par une analyse de notre approche.

7.2.1 Impact du relâchement de la fraîcheur

Dans cette section, nous étudions et mesurons l'influence du relâchement de la fraîcheur sur les performances en termes de temps de réponse et d'équilibrage de charge. Pour ce faire, nous nous plaçons dans le même environnement expérimental décrit dans la section 7.1.1. Nous avons choisi une taille intermédiaire : 40 applications (20 pour des mises à jour et 20 pour des lectures) et 20 ND. Chaque application envoie 40 transactions durant toute l'expérience, ce qui donne un nombre total de transaction égal à 1600. Nous faisons varier l'obsolescence tolérée des transactions de lecture. La figure 7.8 montre le temps de réponse des transactions en fonction de l'obsolescence tolérée (exprimée en nombre de mises à jour manquantes). Les résultats révèlent qu'augmenter l'obsolescence tolérée diminue considérablement le temps de réponse. Cela est principalement dû au fait que relâcher la fraîcheur donne plus de souplesse pour retarder la synchronisation, et donc l'exécution des transactions se fait de manière plus rapide. Nous notons que si le degré d'obsolescence dépasse 15, le temps de réponse ne s'accroît plus. La raison est que le temps de réponse minimal d'exécution d'une transaction est atteint. La valeur précise 15 découle de la configuration de la taille de notre système, *i.e.* un nombre d'applications et de ND différents de celui que nous avons considéré engendrerait une valeur différente de 15.

Bien entendu, le relâchement de la fraîcheur s'accompagne d'une perte de cohérence mutuelle des répliques. La figure 7.9 montre l'obsolescence des données à la fin de l'expérience, en fonction de l'obsolescence tolérée des transactions de lecture. Heureusement, le nombre de transac-

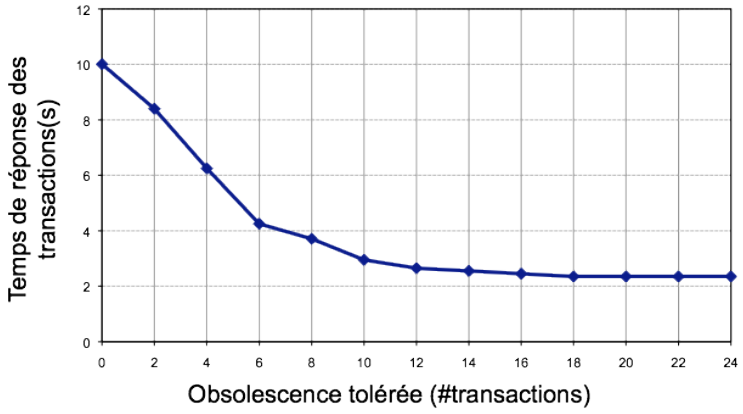

FIGURE 7.8 – Temps de réponse vs. obsolescence tolérée

tions manquantes croit faiblement (avec une tendance logarithmique) quand l'obsolescence tolérée augmente. Pour une obsolescence tolérée de 24 transactions, le nombre cumulé de transactions manquantes sur tous les ND n'est que de 315 soit moins de 20 % du nombre total de transactions envoyées (1600) durant l'expérience avec cependant une diminution du temps de réponse de 80% (cf. figure 7.8).

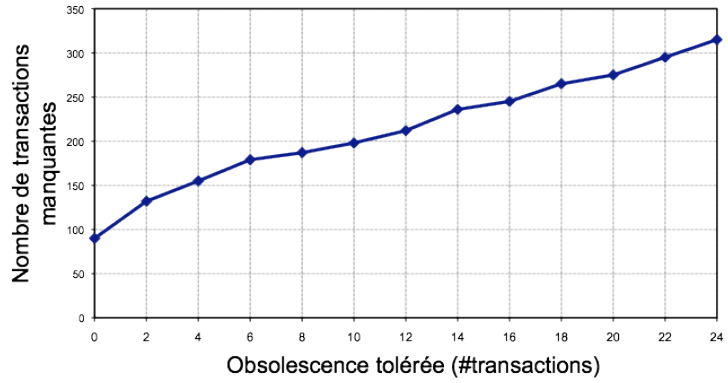

FIGURE 7.9 – Nombre de mises à jour manquantes vs. obsolescence tolérée

Ensuite, nous évaluons l'impact du relâchement de la fraîcheur sur l'équilibrage des charges. Pour atteindre ce but, nous mesurons le taux de déséquilibre de la répartition des charges (ζ). Le taux de déséquilibre montre l'imperfection ou la déviation de la répartition des charges par rapport à un équilibrage parfait. Pour obtenir la déviation, nous divisons l'écart type (σ) par la moyenne de la charge applicative (E) : $\zeta = \frac{\sigma}{E}$. Les résultats de la figure 7.10 montrent comment l'obsolescence tolérée réduit par un facteur de 2 la déviation initiale, *i.e.* quand une fraîcheur maximale est requise. En d'autres mots, nous obtenons un meilleur équilibrage quand l'obsolescence tolérée croît. En

effet, l'accroissement de l'obsolescence tolérée augmente le nombre de candidats et par conséquent les choix possibles.

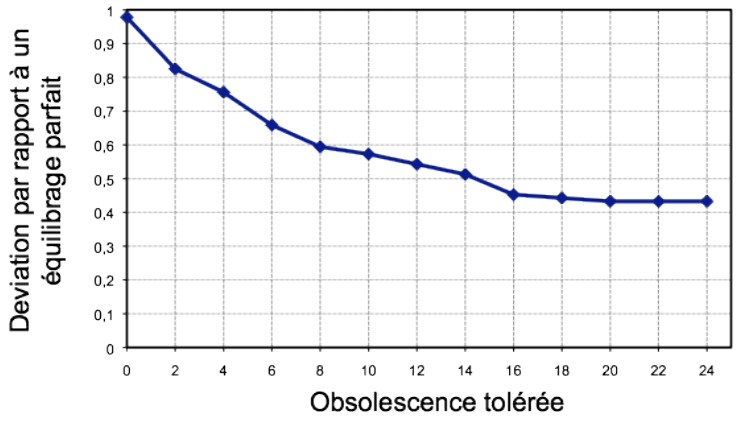

FIGURE 7.10 – Equilibrage des charges vs. obsolescence tolérée

En conclusion, l'introduction du relâchement de la fraîcheur permet d'accroître les performances en réduisant le temps de réponses surtout des requêtes et en améliorant l'équilibrage des charges. Le contrôle du relâchement de la fraîcheur est effectué de manière simple en s'appuyant sur le catalogue qui stocke l'état des nœuds et donc n'engendre pas de surcharge ni ne compromet l'autonomie des SGBD.

7.2.2 Apport du routage décentralisé

Après avoir montré que la gestion du catalogue donne de bonnes performances en termes de débit et de contrôle de la fraîcheur des nœuds. Nous évaluons à présent l'apport du routage distribué, autrement dit nous mesurons le gain introduit par la redondance des gestionnaires de transactions.

En effet, nous mesurons les améliorations du routage distribué en ce qui concerne le débit, comparées à la version centralisée de [GNPV07]. La charge est générée par N applications classées dans 3 catégories en proportion égale. Chaque type d'application accède à une partie distincte de la base de données et donc est connecté à un routeur spécifique. En d'autres termes, il n y a aucune concurrence entre routeurs au niveau de l'accès au catalogue, *i.e.* $\tau_{AC} = 0$. Nous mesurons le débit du traitement quand N varie de 15 à 150 applications. Sur la figure 7.11, nous comparons ces résultats avec le cas où un seul routeur reçoit cette même charge. Plus N s'accroît, plus la différence entre le routage centralisé et distribué devient importante. Pour une forte charge de 150 applications, le gain avec le routage distribué atteint un facteur de 3. La raison principale est que le routage centralisé atteint ses limites très rapidement car s'appuie sur un catalogue stocké sur un seul nœud. Nous observons un gain égal au nombre de routeurs, ce qui démontre une montée en charge linéaire. De plus, la répartition des métadonnées sur plusieurs sites permet au GT de fonctionner de manière totalement parallèle, ce qui maintient les performances.

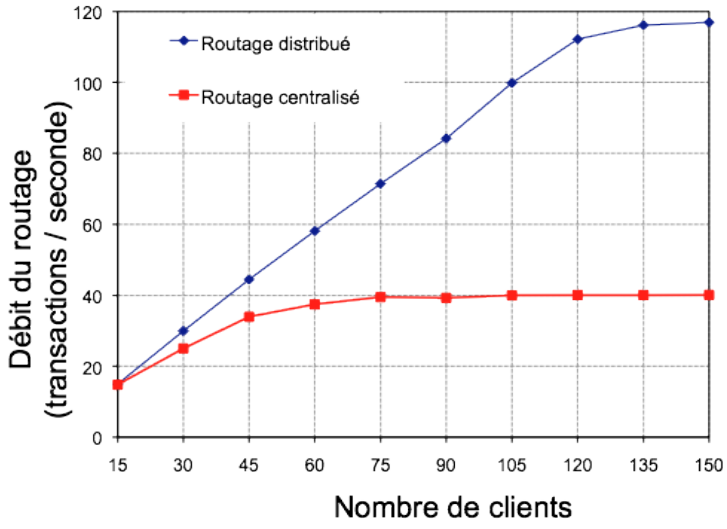

FIGURE 7.11 – Routage réparti vs. routage centralisé

7.2.3 Passage à l'échelle

Dans cette section nous étudions le passage à l'échelle de notre solution. Pour ce faire, nous utilisons de la simulation afin d'avoir plusieurs milliers de nœuds.

Nous utilisons notre prototype TRANSPEER décrit dans 7.1.2. Notre choix d'utiliser ce prototype découle du fait que FreePastry offre un environnement de simulation sans ré-écriture du code de nos différents composants (GT, NA, ND et NC). Seule la couche communication qui fonctionnait avec les *sockets* de Java doit être simulée avec des invocations de messages conçus sous forme de *thread*.

Une fois notre environnement de simulation configurée, nous mesurons le temps de réponse global quand le nombre d'applications varie de 100 à 1000 et que le nombre de GT varie de 2 à 8. Nous fixons le taux de conflits des NA égal à zéro, *i.e.* les GT accèdent de manière disjoints au catalogue. Ce choix d'absence de conflits se justifie par le fait que nous voulons mesurer si le débit théorique correspond au débit réel, autrement dit, l'ajout de n GT permet-il d'obtenir un débit égal au débit d'un seul GT multiplié par le facteur n.

La première série d'expérience est faite pour évaluer le bénéfice obtenu en ajoutant des GT quand la charge applicative augmente. Nous utilisons 100 ND pour toute l'expérience pour s'assurer qu'il sont toujours disponible et non surchargés (10 NA / ND). Par conséquent seuls les ND et le catalogue peuvent être source de congestion.

La figure 7.12 montre deux résultats. Premièrement, pour une charge de 1000 NA, le temps de réponse diminue si le nombre de GT augmente. Pour un nombre d'applications égal à 1000, le temps de réponse diminue de 42% si le nombre de GT passe de 2 à 8.

Cette amélioration est obtenue puisque l'accroissement du nombre de GT réduit le temps d'attente d'un client pour voir sa requête être prise en compte.

Deuxièmement, la figure 7.12 détermine le nombre minimal de GT nécessaire quand la charge

applicative croît. Par exemple, avec une charge applicative entre 400 et 600 applications, 8 à 10 GT sont suffisants pour assurer un temps de réponse inférieur à 2,5 secondes.

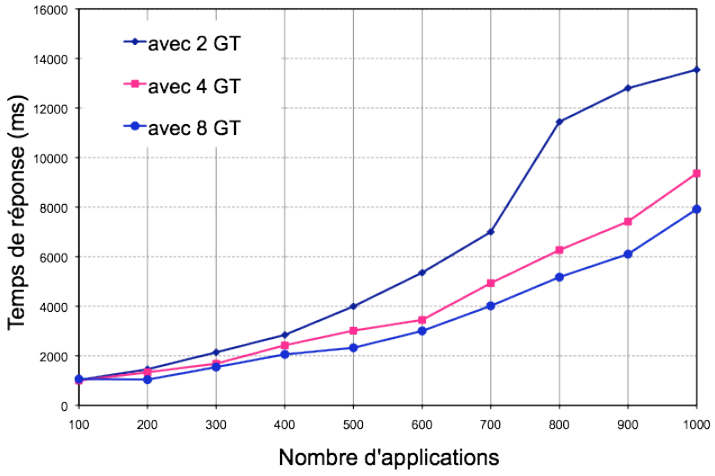

FIGURE 7.12 – Temps de réponse vs. Nombre de NA

Ensuite, nous évaluons l'impact du nombre de ND quand le nombre de GT est judicieusement choisi en fonction des résultats de l'expérience précédente. Ainsi, nous fixons à 100 le nombre de GT suffisant pour qu'ils ne soient pas source de congestion et nous faisons varier le nombre de ND de 50 à 1200.

Nous présentons les résultats obtenus quand le le nombre d'applications varie de 1000 à 4000. La figure 7.13 montre que le temps de réponse diminue si le nombre de ND augmente. Pour une charge applicative de 1000 clients, le temps de réponse est acceptable puisqu'il est inférieur à 2500 millisecondes. De plus, il reste constant même si le nombre de ND varie. Pour les charges applicatives plus importantes (plus de 2000 applications), l'ajout de nouvelles répliques améliore significativement le temps de réponse jusqu'à ce que le nombre de répliques atteigne quelques centaines. A partir de ce degré l'ajout de nouvelles répliques n'augmente guère le débit global pour deux raisons : (1) chaque ND est surchargé par la propagation et l'applications des mises à jour et (2) le nombre de ND est très important et les GT perdent trop de temps à choisir le ND optimal. Dans ce cas, l'ajout de nouveaux GT devient nécessaire pour donner plus de choix aux NA et donc réduire le temps de réponse.

Certes, des expériences avec une charge applicative envoyée par plus de 10.000 NA permettront de mieux situer les limites de notre système. Cependant des contraintes environnementales nous empêchent de les faire. Ces contraintes sont entre autre liées à notre modèle d'implémentation et aux nombres de *thread* maximum que l'on peut tourner sur nos machines physiques. Nos travaux en cours tentent de repousser ces obstacles.

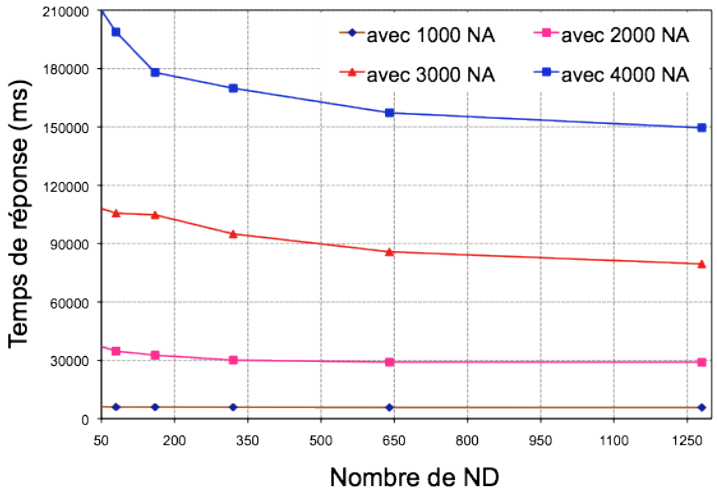

FIGURE 7.13 – Temps de réponse vs. Nombre de ND

7.2.4 Conclusion sur les performance du routage

Le relâchement de la fraîcheur des données lus par les transactions permet d'améliorer le temps de réponse et un meilleur équilibrage des charges. Ce résultat est très important dans le contexte des applications web 2.0 où les transactions peuvent lire des données non fraîches. Par exemple, la consultation d'un profil d'ami sur facebook ou la participation à une vente d'un objet sur eBay. De plus, l'équilibrage des charges permet de garder un certain niveau de disponibilité en éliminant ou réduisant la surcharge d'un nœud. Dans le contexte des applications Web 2.0 où les données sont réparties sur plusieurs data centres, ce relâchement de fraîcheur permettrait une meilleure mutualisation des ressources puisqu'une requête d'un client est traité sur le data centre le plus proche de sa localisation. Ce même constat est aussi valable pour les clouds d'autant plus qu'il permettrait des modèles de coût plus économiques. Par ailleurs, nous avons étudié l'introduction de la redondance des GT. Les résultats montrent que cela est indispensable pour le passage à l'échelle. Limités par nos environnements expérimentaux, nous ne pouvons pas affirmer de manière catégorique que notre solution fonctionne bien quand le nombre de GT passe à 1.000 ou que le nombre de ND dépasse les 10.000. Les raisons de ce doute découle du fait que l'ajout de GT génère plus d'accès au catalogue qui peut devenir source de congestion. Néanmoins dans l'optique de résoudre ce problème, nous sommes en train de développer une approche où les GT n'auront pas besoin d'accéder à un catalogue et par conséquent le passage à une échelle d'une dizaine de milliers de clients pourra être garanti.

7.3 Evaluation des performances de la tolérance aux pannes

Dans cette section, nous évaluons l'impact de la gestion des pannes dans le routage des transactions. Les résultats de ces expériences ont fait l'objet de publications en 2010 dans la revue

nationale RSTI-ISI [SNG10a] et dans la conférence internationale DBKDA [SNG10c]. Notre objectif est d'abord d'évaluer l'impact de la gestion des pannes en évaluant la surcharge engendrée et le gain obtenu. Puis nous évaluons notre mécanisme de détection ciblée des pannes en le comparant avec une technique existante de détection ciblée passant à l'échelle.

Nous nous plaçons dans les mêmes conditions expérimentales de la section 7.2.3, autrement dit nous utilisons notre prototype TRANSPEER pour évaluer la faisabilité.

7.3.1 Configuration du temporisateur

Comme nos algorithmes de détection et de gestion des pannes sont basés sur des temporisateurs, nous commençons d'abord par bien paramétrer leur valeur. Pour cela, nous faisons varier le temporisateur de 10 millisecondes à 10 secondes. Nous reportons dans la figure 7.14 le temps de réponse quand le nombre de 2 NA, 2 GT et 10 ND constituent notre système.

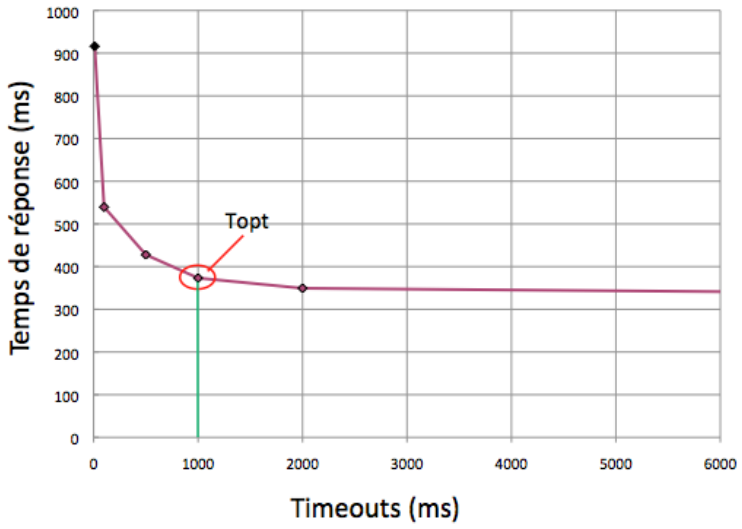

FIGURE 7.14 – Temps de réponse vs. temporisateurs

Nous identifions T_{opt}, la valeur optimale du temporisateur qui ne nécessite aucune retransmission. Nous observons que le temps de réponse diminue de 900 millisecondes jusqu'à environ 330 millisecondes. T_{opt} correspond au seuil de la valeur du temporisateur à partir duquel le temps de réponse reste quasi constante et correspond à 1 seconde.

Nous affirmons que cette valeur est un bon compromis entre d'une part, une faible surcharge de la gestion des pannes des pannes et d'autre part une détection rapide des pannes qui favorise la réduction du temps de réponse des transactions concernées par une panne. Nous mentionnons aussi que nous pouvons dynamiquement ajuster la valeur de T_{opt} en fonction de l'augmentation du taux d'arrivée des pannes.

7.3.2 Surcharge de la gestion des pannes

Une fois que nous avons bien défini la valeur du temporisateur, nous mesurons la surcharge engendrée par la prise en compte des pannes lors du processus de routage. Nous comparons notre solution tolérante aux pannes par apport à notre approche basique qui ne prend pas en compte les pannes. En l'absence de pannes, nous mesurons les débits de chacune des solutions.

Nous utilisons 20 PC du laboratoire connecté par un réseau LAN, chacun n'hébergeant qu'un seul nœud (NA, ND, GT ou NC) pour éviter toute dégradation de performances dues à des partages de ressources. La charge applicative provient des NA qui envoient des transactions sous forme d'instructions SQL. Chaque ND est connecté à un SGBD Postgresql qui exécut les transactions. Chaque expérience est répétée cinq fois afin de s'assurer de la précision des résultats. Alors, nous mesurons le temps de réponse moyen en variant le nombre de ND de 2 à 10.

La figure 7.15 montre que notre solution non tolérante aux pannes produit un temps de réponse quasi constant autour de 150 millisecondes (seul 20 millisecondes de plus avec 10 ND qu'avec 2 ND), tandis que la solutions tolérante aux pannes offre des temps de réponses très élevés.

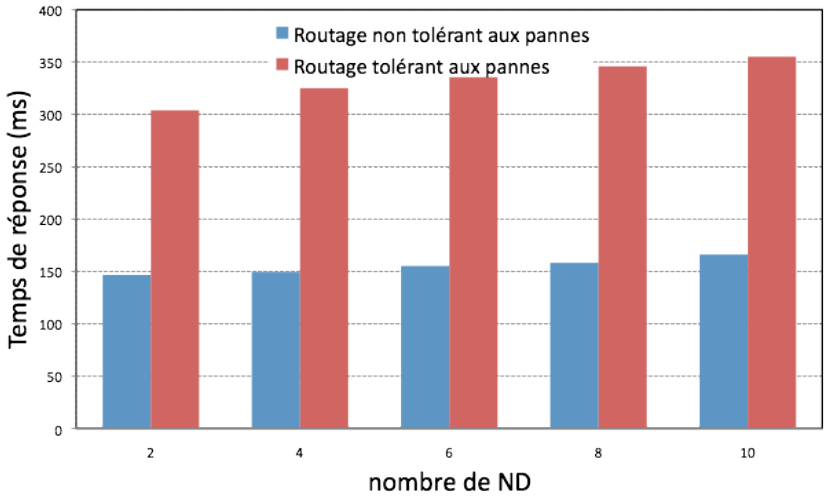

FIGURE 7.15 – Surcharge routage tolérant vs. routage non toléran

En effet, la solution tolérante aux pannes fonctionne deux fois plus lentement que la solution non tolérante puisqu'elle requiert des processus supplémentaire pour gérer les pannes. Par exemple, chaque nœud est couplé avec un chronomètre qui signale chaque fin de temporisateur et le coût supplémentaire découle des tests faits par les GT, ND, NA et NC pour s'assurer que la transaction n'est exécutée qu'une seule fois. Ceci nécessite que les GT communiquent entre eux pour savoir si une transaction est en cours ou déjà exécutée. En plus durant le processus de routage le GT teste si le ND choisi est en panne ou non ce qui rallonge le temps de routage et donc le temps de réponse. Dans cette expérience, notre solution non tolérante est meilleure puisqu'il n'existe aucune panne,ce qui ne reflète pas une situation réelle. En situation de panne, notre solution tolérante

présente plus davantage puisqu'il assure toujours qu'une transaction soit exécutée, ce qui n'est pas le cas avec notre solution non tolérante. Nous évaluons le gain de la solution tolérante dans la prochaine expérience.

7.3.3 Performances de la gestion des pannes

Nous évaluons dans cette section les bénéfices de notre solution en évaluant d'une part l'amélioration du débit de routage et la rapidité de notre protocole à détecter les pannes.

Gain de la gestion des pannes

Dans cette section, nous évaluons l'apport positif de notre solution tolérante. Ainsi, nous instancions plusieurs nœuds par machine (500 NA/ 500 ND / 50 GT). Nous faisons varier le débit de panne des ND de 0 à 100%. Les pannes sont uniformément réparties sur l'ensemble de période d'expérimentation, *i.e.* un débit de panne de 100% signifie que le premier nœud tombe en panne au débit et la prochaine panne arrive tf plus tard et ainsi de suite, avec tf = (taux d'arrivée des pannes * temps d'expérimentation total) / nombre de nœuds. Nous reportons les résultats sur la figure 7.16.

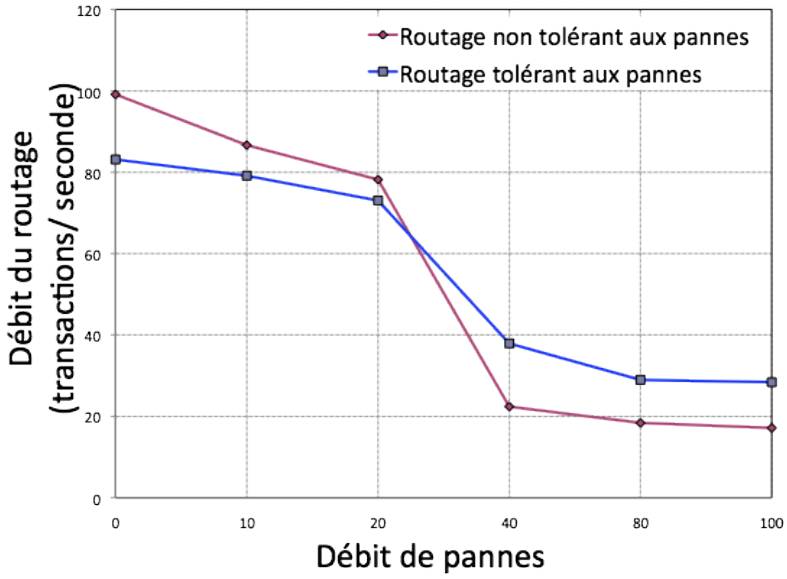

FIGURE 7.16 – Débit routage tolérant vs. routage non tolérant

Les résultats montrent qu'avec un taux de panne faible, la solution non tolérante dépasse de 18% la solution tolérante en débit de routage. Ceci est la conséquence de la surcharge engendrée par la solution tolérante évaluée précédemment .

Par contre dès que le taux de panne dépasse 30%, la solution tolérante dépasse de 3% la solution non tolérante. Ce gain découle du fait que la solution tolérante est capable de continuer les tâches

d'un ND en panne sur un autre ND disponible. De plus, chaque ND en panne est enregistré afin de ne pas l'utiliser pour les prochaines transactions entrantes.

Certes, le débit de panne au delà duquel la solution tolérante surpasse la solution non tolérante est très élevé (30 %). Cependant, nous mentionnons que ce débit inclut aussi bien les pannes que les déconnexions prévues sans oublier les pannes de communication, ce qui augmente le nombre de situations interprétées comme des pannes par notre solution.

Notre objectif dans cette expérience était uniquement de montrer l'apport de la gestion de la dynamicité dans le routage des transactions. Les résultats obtenus sont somme toute intéressants bien qu'ils peuvent être améliorer en dissociant la gestion des pannes de réseau de celle des pannes afin de réduire la surcharge de notre solution tolérante. Cette amélioration inscrite dans nos perspectives, nécessite des techniques de détection des pannes dans un réseau et peuvent être délégués à un système tiers.

Coût de la détection des pannes

Après avoir montré l'intérêt de la gestion des pannes nous nous intéressons au coût de la détection des pannes. Nous comparons notre approche de détection ciblée avec celle proposée dans SWIM [DGM02], en mesurant le coût de communication (*i.e.* le nombre total de messages nécessaires pour détecter les pannes). SWIM est une solution de détection des pannes conçue pour les systèmes à large échelle et s'appuie sur l'envoie de message "ping-pong" sur des ensembles de nœuds choisis aléatoirement. Dans cette expérience, nous nous intéressons uniquement à la détection des GT puisque la détection d'un ND est faite uniquement au cours de l'exécution d'une transaction et ne nécessite pas plus de 5 messages (cf. section 6.2.2).

Dans SWIM, la détection est faite en envoyant périodiquement des messages de "ping-pong" à un ensemble aléatoire de nœuds (nous avons choisi 4 dans notre cas). Dans notre solution, chaque GT collabore avec les autres GT (4 dans cette expérience) pour détecter la panne. Nous commençons la simulation avec 18 GT disponibles et 2 GT en panne. Ensuite nous faisons varier le nombre de GT en panne de 2 à 18. Nous reportons sur la figure 7.17 le nombre de pannes détectées.

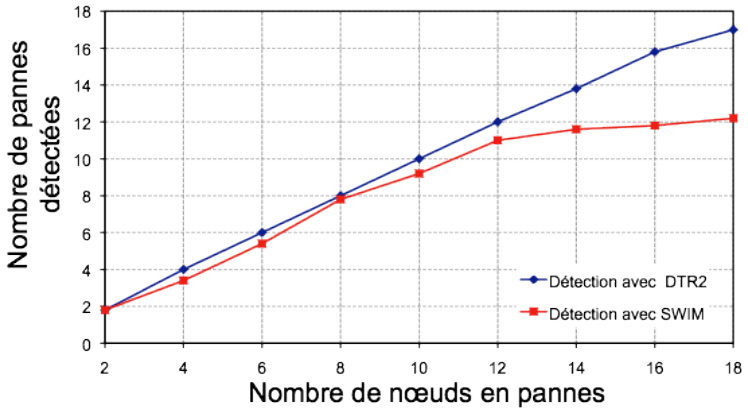

FIGURE 7.17 – Pannes détectées vs. pannes survenues

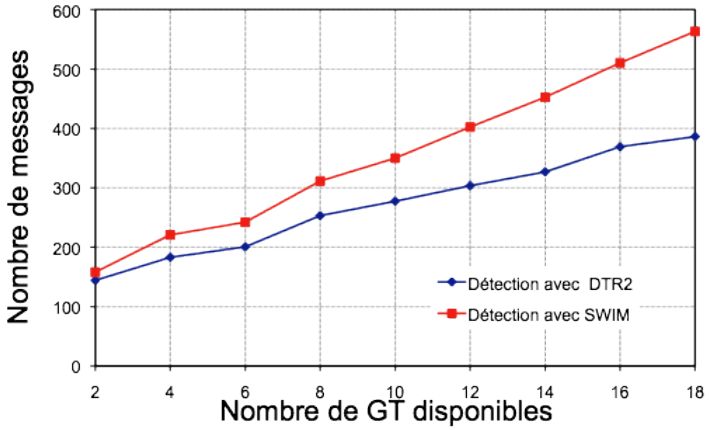

FIGURE 7.18 – Nombre de messages vs. pannes survenues

La figure 7.17 montre qu'avec un nombre de pannes élevé, notre solution génère de meilleures performances que SWIM en ce qui concerne le nombre de pannes détectées. La principale raison est que chaque GT en panne est détecté par au moins un de ses successeurs, donc il est exclu de l'anneau logique. Au contraire, avec SWIM, un GT peut tomber en panne sans pour autant être détecté puisqu'il peut rester longtemps sans être choisi aléatoirement par les autres GT.

En outre, nous comparons le coût de communication engendré par la détection des pannes de notre solution à celui de SWIM. Pour cela, nous calculons le nombre total de messages envoyés par les nœuds pour détecter l'occurrence des pannes. Les résultats présentés sur la figure 7.18 montrent que le nombre total de messages pour détecter les pannes augmente si le nombre de nœuds disponibles est important. En fait, avec 20 GT le nombre de messages est supérieur à 400 durant une simulation, puisqu'un GT contacte ses successeurs périodiquement. En plus, nous soulignons que notre solution requiert toujours moins de messages que SWIM lors de la détection. En effet l'anneau logique est restructuré dès qu'une panne est détectée, évitant ainsi de contacter un nœud en panne plus d'une fois. Avec SWIM, un nœud en panne peut être contacté par les autres nœuds qui ne sont pas encore au courant de la panne, puisque la notification de panne n'est pas immédiate.

7.3.4 Conclusion sur l'évaluation de la gestion des pannes

Les expériences effectuées dans cette section montrent l'importance de la gestion des pannes dans le protocole de routage des transactions. Les résultats montrent qua la prise en compte des pannes réduit la dégradation des performances en présence de pannes. La tolérance aux pannes borne les temps de réponse qui sont des critères forts pour les applications. Par exemple, les applications Web 2.0 fonctionnent avec les revenus générés par les publicités qui sont versés que s'il y a des visiteurs. Ainsi, des temps de réponses longs engendrés par les indisponibilités du service réduisent la présence de visiteurs et donc de fonds pour les fournisseurs d'applications.

7.4 Conclusion

Dans ce chapitre, nous avons présenté la validation de notre approche. Nous avons développé deux prototypes DTR et TRANSPEER pour implémenter les composants de notre routeur. DTR est conçu au dessus de JuxMem et donc nous a permis d'implémenter notre approche de gestion du catalogue avec verrouillage. TRANSPEER est implémenté au dessus d'un réseau P2P afin de gérer le catalogue sans verrouillage. Nous les avons utilisé pour mener une série d'expériences et les résultats obtenus démontrent la faisabilité de nos approches. Les résultats obtenus sont les suivants :

- la surcharge induite par la gestion d'un catalogue réparti est acceptable ;
- la redondance du routeur accroît le débit de routage et réduit le temps de réponse tout en introduisant plus de disponibilité ;
- le choix d'une gestion de catalogue sans verrouillage facilite le passage à l'échelle ;
- le relâchement de la fraîcheur améliore le temps de réponse et donne au routeur plus de choix. Le calcul de la fraîcheur est effectué avec l'utilisation du catalogue sans besoin de contacter les SGBD.
- la prise en compte de la dynamicité du système est indispensable et permet de borner le temps de réponse. Les méthodes de détection et de résolution des pannes utilisées sont simples à mettre en œuvre et s'avèrent bien adaptées pour un système à large échelle.

Chapitre 8

Conclusion et Perspectives

8.1 Synthèse

Les travaux présentés dans cette thèse montrent que l'utilisation d'un catalogue réparti associé à un modèle de routage réparti permet d'améliorer le débit transactionnel d'une base de données répliquées à large échelle tout en préservant la cohérence à terme et en contrôlant la fraîcheur des données lues par les requêtes. La prise en compte de la dynamicité des nœuds du système permet de maintenir les performances du système en cas d'occurrence de pannes.

Architecture du système de routage. Notre architecture se présente comme un intergiciel pour contrôler l'accès à la base de données et peut être divisée en deux parties : une partie assurant le service de médiation entre les différents composants du système (intergiciel) et une autre chargée de la gestion des métadonnées qui sont les données nécessaires au fonctionnement du système en entier. Le choix d'une architecture hybride à mi-chemin entre les systèmes P2P structurés et ceux non structurés nous permet de tirer profit des avantages des uns et des autres. En fait, la structuration des nœuds GT autour d'un anneau logique permet de faciliter leur collaboration pour assurer le traitement cohérent des transactions alors que la structuration faible des nœuds ND leur confère une grande autonomie. Notre intergiciel redondant permet de faire face à la volatilité d'un environnement à large échelle puisqu'à chaque fois qu'un nœud GT ou ND tombe en panne, nous utilisons un autre nœud disponible pour continuer le traitement ou récupérer les données. L'utilisation d'un catalogue réparti facilite l'exploitation des ressources disponibles et un contrôle global de l'état du système. Pour rendre disponible les informations stockées à l'intérieur du catalogue, nous les avons répliquées par l'intermédiaire de systèmes garantissant le passage à l'échelle et la disponibilité, notamment JuxMem et une DHT. Pour exploiter avec efficacité, les ressources du systèmes (équilibrer les charges, identifier rapidement le nœud optimal, etc.), nous collectons plusieurs informations dans le catalogue comme métadonnées et nous les structurons logiquement pour que leur manipulation (lecture et modification) soit simple.

Protocole de routage. Nous avons proposé un mécanisme de routage des transactions pour garantir une exécution rapide et cohérente, des transactions. Les algorithmes de routage proposés

requièrent des accès au catalogue réparti pour maintenir la cohérence mutuelle des répliques et ils définissent l'ordre dans lequel les transactions doivent être exécutées pour ne pas compromettre la cohérence. Le premier algorithme est dit pessimiste et ordonne toutes les transactions conflictuelles en s'appuyant sur les conflits potentiels. En d'autres mots, le protocole de routage assure une sérialisation globale définie de manière pessimiste et qui est utilisé pour router les transactions. Chaque transaction est associée avec ses classes de conflits, qui contiennent les données que la transaction peut potentiellement lire (resp. modifier). En fonction des classes de conflits, les transactions sont ordonnées dans un GSG en s'appuyant sur leur ordre d'arrivée. Bien que cette approche assure une sérialisation globale, elle réduit malheureusement la parallélisation du traitement des transactions puisqu'elle s'appuie sur des sur-ensembles potentiels de données réellement accédées.

Pour améliorer le parallélisme du traitement des transactions, nous avons proposé un second algorithme qui combine une approche pessimiste et optimiste. Ce second algorithme s'appuie sur une tentative d'exécution des transactions afin deréduire le temps de réponse des transactions. Autrement dit, les transactions conflictuelles sont exécutées de manière optimiste et une phase de validation est utilisée à la fin pour garantir la cohérence. Dans le contexte des applications Web 2.0 où les transactions courantes et potentiellement conflictuelles sont peu nombreuses, les chances de réussite du routage optimiste s'avèrent très élevées et donc il apparaît plus adapté.

Catalogue réparti. Pour maintenir la cohérence globale, nous avons conçu un catalogue pour stocker les métadonnées (GSG). Le catalogue est utilisé à chaque processus de routage et peut faire alors l'objet d'accès concurrents qui peuvent être source d'incohérence. Pour garantir la cohérence du catalogue lors de l'accès au métadonnées, nous avons proposé deux approches : une approche utilisant le verrouillage et une autre sans verrouillage. La gestion avec verrouillage est implémentée via JuxMem dans le but d'intégrer nos travaux dans le cadre du projet ANR Respire [Prod]. Malheureusement, nous avons remarqué que le verrouillage ne facilite pas le passage à l'échelle. C'est pourquoi nous avons opté pour une solution sans verrouillage lors de l'accès au catalogue. De plus, nous nous sommes intéressés à des systèmes tels que les DHT qui permettent de gérer des données répliquées sans utilisation de mécanismes de verrouillage. Ainsi, nous avons implémenté la gestion du catalogue à travers une DHT d'autant plus que celle-ci facilite le passage à l'échelle et une grande disponibilité, deux caractéristiques très importantes pour notre système de routage.

Gestion des pannes. Nous avons présenté un mécanisme de gestion de la dynamicité. Ce mécanisme est basé sur la détection sélective des fautes et sur un algorithme de reprise. Contrairement à la plupart des autres approches, notre mécanisme n'implique pas l'utilisation de nœuds qui ne participent pas à léxécution de la transaction en cours, ce qui permet de passer à l'échelle. Pour cela, nous adaptons des approches existantes de détection des pannes afin de les rendre opérationnelles pour chaque type de nœud (gestionnaire de transaction et nœud de données) de notre système. Nous avons proposé un protocole permettant de gérer toutes les situations lorsqu'un nœud quitte le système pendant le traitement d'une transaction. Ceci est nécessaire et suffisant pour contrôler la cohérence du système, surtout en cas de déconnexions intempestives.

Cependant, pour maintenir le débit transactionnel en cas de fréquentes pannes, il faut être

capable d'ajouter de nouvelles ressources en fonction des déconnexions. Pour ce faire, nous avons proposé un modèle permettant de déterminer et contrôler le nombre de répliques requises pour garder le système disponible. Ce modèle permet de déterminer le nombre minimum de répliques nécessaires au bon fonctionnement du système et donc de minimiser les surcoûts liés à la gestion des répliques.

Validation. Pour valider la faisabilité de nos approches, nous avons implémenté deux prototypes nommés respectivement DTR (*Distributed Transaction Routing*) et TRANSPEER (*TRANSaction on PEER-to-peer*). L'implémentation de deux prototypes est liée au besoin de gérer le catalogue avec verrouillage ou sans verrouillage. De fait, DTR constitue le prototype développé avec verrouillage du catalogue alors que TRANSPEER est conçu pour une gestion du catalogue sans verrouillage et un modèle de communication de type P2P. Puis, nous avons effectué quelques simulations pour étudier le passage à l'échelle et la tolérance aux pannes de notre solution. Notre choix d'utiliser à la fois de l'expérimentation et de la simulation se justifie par le fait que : (1) l'expérimentation permet d'évaluer un système dans des conditions réelles ; et (2) la simulation est une représentation simplifiée du système, facile à réaliser et requiert moins de ressources que l'implémentation, ce qui favorise l'évaluation d'un système à grande échelle. Nous avons mené une série d'expériences sur nos deux prototypes pour étudier les performances de notre système : débit transactionnel, temps de réponse, passage à l'échelle et tolérance aux pannes. Les résultats obtenus montrent que l'utilisation d'un catalogue pour stocker les métadonnées permet de router les transactions en contrôlant le niveau de fraîcheur sollicité par les applications. De plus, la surcharge induite par la gestion d'un catalogue réparti est acceptable et donc n'a pas trop d'impact négatif sur le débit du routage. Les expériences ont montré que le relâchement de la fraîcheur des données améliore le temps de réponse des requêtes et l'équilibrage des charges, ce qui est économiquement important vis-à-vis l'utilisation totale des ressources disponibles. Les résultats montrent également que la redondance du routeur accroît le débit de routage et réduit le temps de réponse tout en introduisant plus de disponibilité. Les résultats obtenus avec notre prototype TRANSPEER démontrent le gain de la gestion des métadonnées sans verrouillage, ce qui favorise la réduction du temps de réponse. Enfin, nous avons montré que la prise en compte de la dynamicité du système est indispensable et permet de borner le temps de réponse. Les méthodes de détection et de résolution des pannes utilisées sont simples à mettre en œuvre et s'avèrent bien adaptées pour un système à large échelle.

8.2 Perspectives

Nos principales perspectives visent à améliorer la capacité de notre solution à passer à l'échelle surtout et à la rendre plus générique de telle sorte qu'elle soit indépendante de la définition des répliques et de leur placement.

Amélioration du passage à l'échelle. Notre solution présente l'avantage de s'adapter dynamiquement aux évolutions du système. Cependant, son comportement en présence de forts et nombreux pics de charge n'a pas été étudié. C'est pourquoi nous nous proposons d'étudier un algorithme de synchronisation gérant les pics de charge en présence de pannes. Cette perspective a pour

but de proposer un algorithme distribué permettant de résoudre le problème de synchronisation en soulageant la charge des GT dédiés à la gestion du graphe et en réduisant les communications nécessaires entre ces noeuds. L'accent sera mis, principalement, sur l'aptitude à gérer les pics de charge. Ce problème est d'autant plus important que dans le type d'applications visées, on cherche davantage à garantir, au plus grand nombre, une réponse dans un temps donné, qu'à réduire le temps d'accès moyen.

Par ailleurs, dans l'implémentation actuelle de notre approche, la synchronisation des accès au graphe repose sur l'utilisation d'algorithmes centralisés exécutés par quelques pairs dédiés. Cette solution présente l'inconvénient de traiter toutes les requêtes en parallèle et donc d'être particulièrement sensible aux pics de charge. Parallèlement l'équipe REGAL du LIP6 a développé plusieurs algorithmes distribués de synchronisation [SALAS09, SAS06] qui offrent une faible sensibilité aux variations de charge, ainsi que de très bonnes performances (i.e. le temps d'accès conserve un faible écart type et une faible moyenne). Cependant ces algorithmes ne prévoient pas de garantie sur la latence maximale d'obtention de l'accès. Ils doivent être améliorés afin d'avertir le demandeur qu'un accès sera trop long à obtenir plutôt que de le faire patienter.

Pour ce faire, nous proposons de composer les approches développées par les deux équipes pour obtenir un algorithme tolérant aux fautes et restant efficace sous une forte charge momentanée. Le principe consiste à effectuer un accès en deux étapes : on demande l'accès à l'algorithme distribué pour pouvoir, ensuite, demander l'accès à l'algorithme centralisé. On peut alors voir l'algorithme distribué comme une simple file d'attente répartie. Cette approche compositionnelle présente plusieurs avantages :

- La base de donnée utilisant un algorithme centralisé (ou répliqué), verra ses pics de charge absorbés par l'algorithme distribué qui se comportera en file d'attente distribuée.
- L'algorithme distribué, ne gérant pas directement l'accès au graphe (ressource critique), pourra traiter un problème de synchronisation un peu simplifié de façon à gérer plus simplement la dynamicité du système.

Transactions multi-pairs. Actuellement nos solutions se limitent à des transactions dites "monosite", qui modifient une seule base (toutes les données manipulées par la transaction sont sur une même réplique). Cette limitation s'avère contraignante car elle empêche de définir librement les répliques indépendamment des transactions. En effet, une réplique devant contenir toutes les données accédées par une transaction, cela peut aboutir à une réplication totale des données, ce qui est contraire au contexte de réplication partielle dans lequel se situe cette thèse. Par conséquent, le problème se pose de traiter des transactions réparties (ou multi sites) qui ont besoin de modifier plusieurs bases de manière atomique. Plusieurs solutions ont été proposées mais elles ne fonctionnent pas dans le contexte des applications ciblées. Certaines solutions sont limitées à un cluster (de l'ordre de 100 nœuds) donc en faisant l'hypothèse que le réseau sous-jacent est fiable [PMJPKA05, ATS$^+$05], d'autres brisent l'autonomie des sites en imposant de modifier le gestionnaire de transaction local du SGBD [CPV05].

L'objectif de cette perspective est de concevoir une solution pour router les transactions à très large échelle, en contrôlant les traitements locaux (sous-transactions) effectuées sur les pairs. L'approche suivie consiste à utiliser au mieux les SGBD existant sur chaque pair et capable de traiter

des sous- transactions localement. La difficulté est de définir un ordre global de traitement des transactions, de déterminer un ordre compatible avec l'ordre global et qui apportera un gain de performance significatif, et finalement de garantir que cet ordre sera toujours respecté par l'exécution des sous- transactions sur les répliques malgré les pannes probables d'un ou plusieurs pairs.

Bibliographie

[ABJ05] G. Antoniu, L. Bougé, and M. Jan. JuxMem : An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing : Practice and Experience*, 6(3) :45–55, 2005.

[ACT99] M. Aguilera, W. Chen, and S. Toueg. Using the Heartbeat Failure Detector for Quiescent Reliable Communication and Consensus in Partitionable Networks. *Theoretical Computer Science*, 220(1) :3–30, 1999.

[ACZ03] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning : consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware '03 : Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 282–304, 2003.

[ADM06] G. Antoniu, J. Deverge, and S. Monnet. How to Bring Together Fault Tolerance and Data Consistency to Enable Grid Data Sharing. *Concurrency and Computation : Practice and Experience*, 18(13) :1705–1723, 2006.

[APV07] R. Akbarinia, E. Pacitti, and P. Valduriez. Data Currency in Replicated DHTs. In *Int. Conf. on Management of Data (SIGMOD)*, pages 211–222, 2007.

[AT89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. Database Syst.*, 14(2) :264–290, 1989.

[ATS+05] F. Akal, C. Türker, H. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Int. Conf. on Very Large DataBase (VLDB)*, pages 565–576, 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. In *SIGMOD '95 : Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, 1995.

[BFG+08] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD '08 : Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 251–264, 2008.

[BGL+06] R. Baldoni, R. Guerraoui, R. R. Levy, V. Quéma, and S. T. Piergiovanni. Unconscious eventual consistency with gossips. In *SSS*, pages 65–81, 2006.

[BGRS00] K. Böhm, T. Grabs, U. Röhm, and H. Schek. Evaluating the coordination overhead of replica maintenance in a cluster of databases. In *Euro-Par 2000 Parallel Processing*, pages 435–444, 2000.

[BHG87] P. A. 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. *SIGMOD Rec.*, 28(2) :97–108, 1999.

[Bre00] E. A. Brewer. Towards robust distributed systems (abstract). In *PODC '00 : Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, page 7, 2000.

[BYV08] R. Buyya, C. S. Yeo, and S. Venugopal. Market-oriented cloud computing : Vision, hype, and reality for delivering it services as computing utilities. In *HPCC '08 : Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 5–13, 2008.

[CDKR02] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC*, 20 :2002, 2002.

[CL02] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4) :398–461, 2002.

[CMZ05] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC : Flexible Database Clustering Middleware. Technical report, ObjectWeb, Open Source Middleware, 2005.

[CPV05] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. In *ICPADS '05 : Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pages 809–815, 2005.

[CPW07] L. Camargos, F. Pedone, and M. Wieloch. Sprint : A Middleware for High-Performance Transaction Processing. In *ACM European Conf. on Computer Systems (EuroSys)*, pages 385–398, 2007.

[CRF09] J. M. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4) :1–42, 2009.

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

[CVL10] M. Correia, G. S. Veronese, and L. C. Lung. Asynchronous byzantine consensus with 2f+1 processes. In *SAC '10 : Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 475–480, 2010.

[DGM02] A. Das, I. Gupta, and A. Motivala. SWIM : Scalable Weakly Consistent Infection-style Process Group Membership Protocol. In *Int. Conf. on Dependable Systems and Networks (DSN)*, 2002.

[DS06] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB '06 : Proceedings of the 32nd international conference on Very large data bases*, pages 715–726, 2006.

[DSS10] O. Diallo, M. Sene, and I. Sarr. Freshness-aware metadata management : Performance evaluation with swn. In *8th IEEE/ACS International Conference on Computer Systems and Applications, 2009 (AICCSA 2009)*, 2010.

[EGA08] C. Emmanuel, C. George, and A. Anastasia. Middleware-based database replication : the gaps between theory and practice. In *SIGMOD '08 : Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 739–752, 2008.

[EZP05] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS '05 : Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 73–84, 2005.

[Fac] Facebook. www.facebook.com.

[FDMBGJM+09] noz-Escoí F. D. Mu J. M. Bernabé-Gisbert, R. Juan-Marín, nigo J. E. Armendáriz-Í and J. R. González De Mendívil. Revising 1-copy equivalence in replicated databases with snapshot isolation. In *OTM '09 : Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems*, pages 467–483, 2009.

[FJB09] S. Finkelstein, D. Jacobs, and R. Brendle. Principles for inconsistency. In *CIDR*, 2009.

[FLO+05] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2) :492–528, 2005.

[Fre] FreePastry. http ://www.freepastry.org/freepastry/.

[Gan06] S. Gançarski. *Cohérence et Fraîcheur dans les bases de données réparties*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, Paris 6, France, October 2006.

[Gee09] J. Geelan. Twenty one experts define cloud computing. electronic magazine. http ://virtualization.sys-con.com/node/612375, 2009.

[GGL03] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96 : Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, 1996.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *SOSP '79 : Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, 1979.

[GL02] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2) :51–59, 2002.

[GLRG04] H. Guo, P. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency : how to say "good enough" in sql. In *SIGMOD '04 : Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM, 2004.

[GN95] R. Gallersdörfer and M. Nicola. Improving performance in replicated databases through relaxed coherency. In *VLDB '95 : Proceedings of the 21th International Conference on Very Large Data Bases*, pages 445–456, 1995.

[GNPV07] S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez. The leganet system : Freshness-aware transaction routing in a database cluster. *Journal of Information Systems*, 32(2) :320–343, 2007.

[Gnu] Gnutella. http ://www.gnutella.com/.

[GR92] J. Gray and A. Reuter. *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[Gro] Groove. http ://office.microsoft.com/en-us/groove/default.aspx.

[GS97] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(40) :68–74, 1997.

[GSN09] M. Gueye, I. Sarr, and S. Ndiaye. Database replication in large scale systems : optimizing the number of replicas. In *EDBT/ICDT '09 : Proceedings of the 2009 EDBT/ICDT Workshops*, pages 3–9. ACM, 2009.

[HAA99] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication using atomic broadcast group communication, 1999.

[HAA00] J. Holliday, D. Agrawal, and A. El Abbadi. Database Replication Using Epidemic Communication. In *Euro-Par 2000 Parallel Processing*, pages 427–434, 2000.

[HAA02] J. Holliday, D. Agrawal, and A. El Abbadi. Partial database replication using epidemic communication. In *ICDCS '02 : Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 485, 2002.

[HCH+05] R. Huebsch, B. Chun, J. M. Hellerstein, B. Thau Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi. The architecture of pier : an internet-scale query processor. In *IN CIDR*, pages 28–43, 2005.

[HIM+04] A. Y. Halevy, Z. G. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The piazza peer data management system, 2004.

[HSAA03] J. Holliday, R. Steinke, D. Agrawal, and A. El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(5) :1218–1238, 2003.

[Jan06] Mathieu Jan. *JuxMem : un service de partage transparent de données pour grilles de calculs fondé sur une approche pair-à-pair*. Thèse de doctorat, Université de Rennes 1, IRISA, Rennes, France, November 2006.

[JEAIMGdMFDM08] nigo J. E. Armendáriz-I A. Mauch-Goya, J. R. González de Mendívil, and noz-Escoí F. D. Mu˙ SIPRe : a partial database replication protocol with SI replicas. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 2181–2185, 2008.

[JPMPAK03] R. Jiménez-Peris, no-Martínez M. Pati G. Alonso, and B. Kemme. Are quorums an alternative for data replication ? *ACM Trans. Database Syst.*, 28(3) :257–294, 2003.

[JWZ03] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra : A peer-to-peer approach to network intrusion detection and prevention, 2003.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent : Postgres-r, a new way to implement database replication. In *VLDB '00 : Proceedings of the 26th International Conference on Very Large Data Bases*, pages 134–143, 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. Database Syst.*, 25(3), 2000.

[KaZ] KaZaA. http ://www.kazza.com/.

[Kra09] S. Krakowiak. *Middleware Architecture with Patterns and Frameworks*. Creative Commons License, 2009.

[LAF99] M. Larrea, S. Arévalo, and A. Fernández. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In *Proceedings of the 13th International Symposium on Distributed Computing*. Springer-Verlag, 1999.

[LFVM09] A. A. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel olap query processing in database clusters with data replication. *Distrib. Parallel Databases*, 25(1-2) :97–123, 2009.

[LG06] C. Le Pape and S. Gançarski. Replica Refresh Strategies in a Database Cluster. In *High-Performance Data Management in Grid Environments (HPDGrid VECPAR Workshop)*, 2006.

[LGV04] C. Le Pape, S. Gançarski, and P. Valduriez. Refresco : Improving Query Performance Through Freshness Control in a Database Cluster. In *Int. Conf. On Cooperative Information Systems (CoopIS)*, 2004.

[LKJP+09] Y. Lin, B. Kemme, R. Jiménez-Peris, M . Patiño-Martínez, and nigo J. E. Armendáriz-I˙ Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2) :1–49, 2009.

[LKMPJP05] Y. Lin, B. Kemme, no-Martínez M. Pati and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05 : Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, 2005.

[LM09] A. Lakshman and P. Malik. Cassandra : structured storage system on a p2p network. In *PODC '09 : Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 5–5, 2009.

[MM02] P. Maymounkov and D. Mazières. A peer-to-peer information system based on the xor metric. In *1st Int. Workshop on Peer-to-Peer Systems(IPTPS)*, 2002.

[MN09] T. Mishima and H. Nakamura. Pangea : an eager database replication middleware guaranteeing snapshot isolation without modification of database servers. *Proc. VLDB Endow.*, 2(1) :1066–1077, 2009.

[MSN] MSN. www.msn.com.

[Nap] Napster. http ://www.napster.com/.

[OV99] M. T. Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[PA04] C. Plattner and G. Alonso. Ganymed : scalable replication for transactional web applications. In *Middleware '04 : Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, 2004.

[Pap05] C. Le Pape. *Contrôle de qualité des données répliquées dans les clusters*. Thèse de doctorat, Université Pierre et Marie Curie, Paris 6, France, December 2005.

[Pau02] Pragyansmita Paul. Seti @ home project and its website. *Crossroads*, 8(3) :3–5, 2002.

[PCVO05] E. Pacitti, C. Coulon, Patrick Valduriez, and T. Ozsu. Preventive replication in a database cluster. *Distributed and Parallel Databases*, 18(3) :223–251, 2005.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *SRDS '97 : Proceedings of the 16th Symposium on Reliable Distributed Systems*, page 175, 1997.

[PGS03] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1) :71–98, 2003.

[PMJPKA05] M. Patino-Martinez, R. Jimenez-Peres, B. Kemme, and G. Alonso. MIDDLE-R, Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems*, 28(4) :375–423, 2005.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. *Int. Conf. on Very Large DataBases (VLDB)*, 1999.

[proa] ApGrid project. http ://www.apgrid.org/.

[prob] China National Grid project. http ://www.cngrid.org/web/guest/home.

[Proc] Grid'5000 Project. www.grid5000.org.

[Prod] Respire Project. http ://www.respire.lip6.fr.

[proe] TeraGrid project. https ://www.teragrid.org/web/about/index.

[prof] The Folding@home project. www.folding.stanford.edu.

[Prog] The Hadoop Project. www.hadoop.apache.org.

[PRS07] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for distributed wikis on structured overlays. In *Managing Virtualization of Networks and Services*, pages 256 – 267, 2007.

[PST+97] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *SOSP '97 : Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 288–301, 1997.

[Pu91] Calton Pu. Generalized transaction processing with epsilon-serializability. In *In Proceedings of Fourth International Workshop on High Performance Transaction Systems, Asilomar*, 1991.

[RBSS02] U. Rohm, K. Bohm, H. Sheck, and H. Schuldt. FAS - a Freshness-Sensitive Coordination Middleware for OLAP Components. *Int. Conf. on Very Large DataBases (VLDB)*, 2002.

[RC96] K. Ramamritham and P. K. Chrysanthis. A taxonomy of correctness criteria in database applications. *The VLDB Journal*, 5(1) :085–097, 1996.

[RD01a] A. Rowstron and P. Druschel. Pastry : Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, 2001.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM '01 : Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172, 2001.

[Rui] Jean-Francois Ruiz. Web 2.0 - quelles-applications ?

[SALAS09] J. Sopena, L. Arantes, F. Legond-Aubry, and P. Sens. Building effective mutual exclusion services for grids. *The Journal of Supercomputing*, 49(1) :84–107, 2009.

[SAS06] J. Sopena, L. Arantes, and P. Sens. Performance evaluation of a fair fault-tolerant mutual exclusion algorithm. In *SRDS*, pages 225–234, 2006.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : a tutorial. *ACM Comput. Surv.*, 22(4) :299–319, 1990.

[Sch93] F.B. Schneider. *Replication Management Using the State-Machine Approach*, pages 169–197. Distributed Systems (2nd Ed.). ACM Press, 1993.

[SGMB01] L. Serafini, F. Giunchiglia, J. Mylopoulos, and P. A. Bernstein. The local relational model : Model and proof theory, 2001.

[Sho07] R. Shoup. eBay Marketplace Architecture : Architectural Strategies, Patterns and Forces. In *InfoQueue Conf. on Enterprise Software Development*, 2007.

[SJPPMK06] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *SRDS '06 : Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pages 377–390, 2006.

[Sky] Skype. www.skype.com.

[SMK+01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord : A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, 2001.

[SNG08a] I. Sarr, H. Naacke, and S. Gançarski. Dtr : Distributed transaction routing in a large scale network. In *VECPAR*, pages 521–531, 2008.

[SNG08b] I. Sarr, H. Naacke, and S. Gançarski. Routage décentralisé de transactions avec gestion des pannes dans un réseau à large échelle. In *BDA*, 2008.

[SNG10a] I. Sarr, H. Naacke, and S. Gançarski. Routage décentralisé de transactions avec gestion des pannes dans un réseau à large échelle. *Ingénierie des Systèmes d'Information*, 15(1) :87–111, 2010.

[SNG10b] I. Sarr, H. Naacke, and S. Gançarski. Transpeer : Adaptive distributed transaction monitoring for web2.0 applications. In *SAC*, pages 423–430, 2010.

[SNG10c] Idrissa Sarr, Hubert Naacke, and Stéphane Gançarski. Failure-tolerant transaction routing at large scale. In *DBKDA*, pages 165–172, 2010.

[SOMP01] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *NCA '01 : Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, page 298, 2001.

[SPMJPK07] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC '07 : Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, pages 290–297, 2007.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Commun. ACM*, 39(4) :84–87, 1996.

[SSP10] N. Schiper, P. Sutra, and F. Pedone. P-store : Genuine partial replication in wide area networks. Technical Report 2010/03, University of Lugano, 2010.

[TS99] A. S. Tanenbaum and M. VAN STEEN. *Distributed Systems : Principles and Paradigms*. Prentice Hall, 1999.

[VATS04] V. Vlachos, S. Androutsellis-Theotokis, and D. Spinellis. Security applications of peer-to-peer networks. *Comput. Netw.*, 45(2), 2004.

[VBB+03] R. VanRenesse, K. Birman, A. Bozdog, D. Dimitriu, M. Singh, and W. Vogels. Heterogeneity-aware peer-to-peer multicast. In *17th International Symposium on Distributed Computing (DISC2003)*, 2003.

[VBLM07] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP '07 : Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 59–72, 2007.

[VMR02] A. K. Vishal, V. Misra, and D. Rubenstein. Sos : Secure overlay services. In *In Proceedings of ACM SIGCOMM*, pages 61–72, 2002.

[Vog09] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1) :40–44, 2009.

[VRMCL09] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds : towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1) :50–55, 2009.

[VS05] D. Del Vecchio and S. H. Son. Flexible update management in peer-to-peer database systems. In *IDEAS '05 : Proceedings of the 9th International Database Engineering & Application Symposium*, pages 435–444, 2005.

[WK05] S. Wu and B. Kemme. Postgres-r(si) : Combining replica control with concurrency control based on snapshot isolation. In *ICDE '05 : Proceedings of the 21st International Conference on Data Engineering*, pages 422–433, 2005.

[WLG+08] P. Watson, P. L., F. Gibson, P. Periorellis, and G. Pitsilis. Cloud computing for e-science with carmen. In *In 2nd Iberian Grid Infrastructure Conference Proceedings*, pages 3–14, 2008.

[WYP97] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence control algorithms for epsilon serializability. *IEEE Trans. Knowl. Data Eng.*, 9(2) :262–274, 1997.

[Yah] Yahoo. www.yahoo.com.

[YV00] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *IN INT. CONF. ON VERY LARGE DATABASES (VLDB)*, pages 123–133, 2000.

Résumé

La réplication dans les bases de données a été largement étudiée, au cours des trois dernières décennies. Elle vise à améliorer la disponibilité des données et à augmenter la performance d'accès aux données. Un des défis majeurs de la réplication est de maintenir la cohérence mutuelle des répliques, lorsque plusieurs d'entre elles sont mises à jour, simultanément, par des transactions. Des solutions qui relèvent partiellement ce défi pour un nombre restreint de bases de données reliées par un réseau fiable existent. Toutefois, ces solutions ne sont pas applicables à large échelle. Par ailleurs, l'antinomie entre les besoins de performances et ceux de cohérence étant bien connue, l'approche suivie dans cette thèse consiste à relâcher les besoins de cohérence afin d'améliorer la performance d'accès aux données. Or, dans le contexte du web2.0, de nombreuses applications tolèrent une cohérence relâchée et acceptent de lire des données qui ne sont pas nécessairement les plus récentes ; cela ouvre la voie vers de nouvelles solutions offrant de meilleures performances en termes de débit transactionnel, latence, disponibilité des données et passage à l'échelle. Par exemple, il est possible de gérer des transactions de vente aux enchères (sur eBay ou Google Adsense) sans nécessairement accéder à la dernière proposition de prix, puisque l'enchère est sous pli cacheté. Dans cette thèse, nous considérons des applications transactionnelles déployées à large échelle et dont les données sont hébergées dans une infrastructure très dynamique telle qu'un système pair-à-pair. Nous cherchons à améliorer les performances des applications en contrôlant la cohérence des données accédées, en équilibrant la charge des répliques et en tenant compte de la disponibilité des ressources (SGBD, gestionnaire de transactions). Nous proposons une solution intergicielle qui rend transparente la distribution et la duplication des ressources mais aussi leur indisponibilité temporaire. Notre solution préserve l'autonomie des applications qui demeurent inchangées, sans qu'aucune modification interne du SGBD ne soit nécessaire. Les applications spécifient leurs exigences en termes de besoin de cohérence, puis l'intergiciel honore ces exigences en contrôlant le routage des transactions et l'état des ressources. Nous définissons deux protocoles pour maintenir la cohérence globale, en fonction de la connaissance des données manipulées par les transactions. Le premier protocole ordonne les transactions à partir de la définition *a priori* des données accédées. Le deuxième protocole détermine un ordre plus souple, en comparant les données accédées, le plus tardivement possible, juste avant la validation des transactions. De plus, nous avons complété notre solution en concevant un catalogue entièrement décentralisé et passant à l'échelle pour gérer les métadonnées nécessaires au routage des transactions. Toutes les solutions proposées tolèrent les pannes franches, fonctionnalité essentielle pour que les résultats de cette thèse puissent être mis en œuvre à très large échelle. Finalement, nous avons implémenté nos solutions pour les valider expérimentalement. Les tests de performances montrent que la gestion

des métadonnées est efficace et améliore le débit transactionnel. Nous montrons également que la redondance de l'intergiciel diminue le temps de réponse face aux situations de pannes.

Mots-clés : Bases de Données, Réplication asynchrone, Routage de transactions, disponibilité, passage à l'échelle, cohérence mutuelle.