Data Replication in three Contexts: Data Warehouse, Cluster and P2P Systems

Esther Pacitti

Université de Nantes

Ecole Doctorale

Sciences et Technologies de l'Information et des Matériaux

Rapport scientifique pour l'obtention de l'

Habilitation à Diriger les Recherches en Informatique

Réplication asynchrone des données dans trois contextes: entrepôts, grappes et systèmes pair-à-pair

Esther PACITTI

8 juillet 2008

Rapporteurs :

Prof. Anne Doucet, Université Pierre et Marie Curie, Paris 6

Prof. Philippe Pucheral, Université de Versailles Saint-Quentin

Prof. Michel Scholl, Conservatoire National des Arts et Métiers, Paris

Examinateurs :

Dr. Serge Abiteboul, Directeur de Recherche, INRIA-Saclay (*Président du Jury*)

Prof. Noureddine Mouaddib, Université de Nantes

Prof. Josep Lluís Larriba Pey, Universitat Politècnica de Catalunya, Barcelone

ACKNOWLEDGEMENTS

I want to thank all the people I met (so many friends) at all the institutions I worked for, which motivated my whole research work since I started in Brazil in 1985: Federal University of Rio de Janeiro (NCE and COPPE), Pontificia Universidade Catolica do Rio de Janeiro (PUC-Rio), University of Paris 5, INRIA (many many thanks to the INRIA teams Rodin, Caravel and Atlas) and University of Nantes. I am very grateful to Brazilian university leaders who always encouraged my research and pushed me to come to France for my Ph.D., providing me a fellowship and believing in me. I very much appreciated this. More recently, the people involved in the management of LINA, Prof. Frédéric Benhamou, Prof. Noureddine Mouaddib and Prof. Pierre Cointe encouraged my research on data replication. I am also very thankful to INRIA, with which I have been involved since 1995 (as a Ph.D. student) until today, as a member of the Atlas team, for providing an excellent environment to do research. In particular, my recent "delegation" at INRIA was crucial to help me produce this HDR report.

Many thanks to the committee members: Prof. Anne Doucet (rapporteur), Prof. Philippe Pucheral (rapporteur), Prof. Michel Scholl (rapporteur), Dr. Serge Abiteboul (exminateur), Prof. Noureddine Mouaddib (examinateur), Prof. (examinateur) for their interest in my work. Special thanks to Anne Doucet for helping in organizing the HDR defense.

I also want to express my intense gratitude to the researchers who helped me during critical times of my research path: Prof. Tamer Özsu, Dr. Eric Simon, Prof. Dennis Shahsa, Prof. Ricardo Jimenez-Peris and Dr. Patrick Valduriez. I am also very thankful to Prof. Bettina Kemme who kindly helped in reviewing this report.

I am very thankful to the members of the Atlas team who contributed to the research in data replication during our thursday meetings, providing useful suggestions and coming up with very good questions. In particular, I am grateful to the Ph.D. students whom I had the chance and pleasure to work with: Cédric Coulon, Vidal Martins, Reza Akbarinia, Manal-El-Dick, Wence Palma. I would also like to thank Elodie Lize for her kind assistance on the defense organization.

And last but not least, I would like to thank my Brazilian and French families for believing and loving me, especially my husband Patrick and daughter Anna.

Making research is sometimes very hard but we also can have much fun !

CONTENTS

Selected Papers:

Résumé Etendu

Motivations

Dans une base de données distribuée, la réplication de données permet d'augmenter la fiabilité et la disponibilité des données ainsi que les performances d'accès [SS05]. En général, l'unité (ou l'objet) de réplication est une table relationnelle (ou un fragment de table), un document ou un fichier. La réplication consiste alors à placer plusieurs *copies* (ou répliques) de l'objet sur différents nœuds du réseau. Cela fournit une grande disponibilité des données. Si un nœud devient non opérationnel à la suite d'une panne par exemple, une autre copie est toujours accessible sur un autre nœud. La réplication permet aussi d'améliorer les performances d'accès en augmentant la localité de référence. Lorsque le coût de communication est un facteur dominant, le placement d'une copie sur le nœud où elle est le plus souvent accédée favorise les accès locaux et évite les accès réseaux.

Les avantages apportés par la réplication sont à comparer avec la complexité et les coûts supplémentaires de maintenance des copies qui doivent, en théorie rester identiques à tout moment. La mise-à-jour d'une copie doit être répercutée automatiquement sur toutes ses répliques. Le problème est compliqué par la présence de pannes de nœud ou réseau. Le compromis recherché entre performance d'accès en consultation et en mise-à-jour des données rend difficile le choix du niveau de réplication. Celui-ci est très dépendant de la charge de travail demandée par les applications. Le problème de la réplication de données reste donc un vaste thème de recherche et les solutions doivent être adaptées au contexte afin d'offrir un bon compromis entre des objectifs conflictuels tels que disponibilité, cohérence, performances, etc. Dans mon travail de recherche, je me suis concentrée sur le maintien de la cohérence des données répliquées dans trois contextes majeurs: les entrepôts de données, les grappes de bases de données, et les applications collaboratives en pair-à-pair (P2P).

Entrepôts de Données

Dans les entrepôts de données [Cod95], la configuration mono-maître est souvent utilisée, avec diverses variantes possibles: diffusion, mono-consolidation (consolidation avec un nœud), multi-consolidation (avec plusieurs nœuds), triangulaire. La gestion de la cohérence est difficile pour certaines configurations comme la consolidation avec plusieurs nœuds, la configuration triangulaire ou leurs généralisations en combinant les configurations de base.

Grappes de bases de données

Les grappes de bases de données (*database clusters*) sont typiquement utilisées par des applications de lectures intensives, ce qui facilite l'exploitation du parallélisme. Cependant, les grappes [ABKW98] peuvent également être utilisées par des applications avec beaucoup de mises-à-jour, par ex. par un ASP (*Application Service Provider)*. Dans un contexte ASP, les applications et les bases de données des clients sont stockées chez le fournisseur et sont disponibles, typiquement depuis Internet, aussi efficacement que si elles étaient locales pour les clients. Pour améliorer les performances, les applications et les données peuvent être répliquées sur plusieurs noeuds. Ainsi, les clients peuvent être servis par n'importe quel noeud en fonction de la charge. Cette architecture fournit une haute disponibilité: dans le cas de la panne d'un noeud, d'autres noeuds peuvent effectuer le même travail. Le défi est alors de gérer la réplication multi-maître, totale et partielle, en assurant la cohérence forte et le passage à l'échelle en nombres de nœuds.

Applications Collaboratives en P2P

Les systèmes P2P adoptent une approche complètement décentralisée [AMPV06a] au partage des ressources. En distribuant données et traitements sur tous les pairs du réseau, ils peuvent passer à très grande échelle sans recourir à des serveurs très puissants. La réplication de données dans les systèmes P2P [AMPV04] est un enjeu majeur pour les applications collaboratives, comme les forums de discussion, les calendriers partagés, ou les catalogues de e-commerce, etc. En effet, les données partagées doivent pouvoir être mises-à-jour en parallèle par différents pairs. Les premiers systèmes P2P existants supposent que les données sont statiques et n'intègrent aucun mécanisme de gestion des mises-à-jour et de réplication. Une mise-à-jour d'une donnée par le pair qui la possède implique une nouvelle version non propagée à ceux répliquant cette donnée. Il en résulte diverses versions sous le même identifiant et l'utilisateur accède à celle stockée par le pair qu'il contacte. Aucune forme de cohérence entre les répliques n'est alors garantie. Le défi est de gérer la cohérence éventuelle face au dynamisme des pairs tout en passant à l'échelle.

Contributions

Les contributions de recherche présentées dans ce rapport scientifique correspondent à la période 1999-2008 :

1. **Réplication mono-maître dans les entrepôts de données (1999-2001).** Nous avons proposé des algorithmes efficaces pour le maintien de la cohérence des données répliquées

dans les entrepôts de données [PSM98,PV98,PMS99,PS00 PMS01]. Ce travail a été validé dans le cadre du projet européen DWQ (Data Warehouse Quality).

2. **Réplication multi-maître dans les grappes de bases de données (2002-2005).** Nous avons proposé un nouvel algorithme de réplication de données, dit *préventif*, asynchrone et multi-maître qui assure la cohérence forte dans les grappes de bases de données [POC03, CGPV04,CPV05a,CPV05b,PCVO05]. Ce travail a été validé dans le cadre du projet RNTL Leg@net avec le prototype RepDB*.
3. **Réconciliation de données dans les applications collaboratives en P2P (2006-2008).** Nous avons proposé des algorithmes efficaces pour la gestion de données répliquées en P2P, notamment pour la réconciliation de données répliquées en mode optimiste [AMPV04, MPV05,MPJV06,MAPV06,MP06,AMPV06a,MPV06a,MPV06b,EPV07,MPEJ08]. Nous avons aussi proposé des optimisations qui exploitent la localité offerte par certains réseaux P2P. Nous avons validé ces algorithmes avec le prototype APPA, dans le cadre des ACI Masses de Données MDP2P et Respire, le projet européen STREP Grid4All et le projet RNTL Xwiki Concerto.
4. **Gestion de données courantes dans les DHTs répliqués (2007-2008).** Nous avons proposé une solution complète pour déterminer les données courantes (les plus à jour) parmi les données répliquées dans les tables de hachage distribuées (DHTs) [APV07a]. Nous avons validé notre solution par une implémentation du DHT Chord sur un cluster de 64 nœuds et par une simulation jusqu'à 10.000 pairs en utilisant SimJava. Ce travail a été réalisé dans le cadre de l'ACI Masses de Données Respire.

Ces travaux ont fait l'objet de trois thèses de doctorat soutenues : **Cédric Coulon** (2005) sur la réplication préventive de données dans les grappes de bases de données ; **Vidal Martins** (2007) sur la réplication sémantique des données en P2P, et **Reza Akbarinia** (2007) sur les techniques d'accès aux données en P2P. Aujourd'hui, il y a deux thèses en cours (en seconde année) dans le prolongement de ces travaux : **Manal El-Dick** sur la gestion de cache en P2P et **Wenceslao Palma** sur la gestion des flux de données en P2P.

Ces travaux ont été aussi validés par deux prototypes majeurs qui implémentent les techniques proposées :

1. **RepDB*: http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB.** Service de réplication de bases de données autonomes dans une grappe de PC. Implémenté en Java sous Linux (20K lignes). Déposé à l'APP en 2004 (INRIA et U. Nantes), logiciel libre sous licence GPL.

2. **APPA (Atlas P2P Architecture): http://www.sciences.univ-nantes.fr/lina/gdd/appa.** Système de gestion de données pair-à-pair pour des applications collaboratives, en cours de réalisation en Java sous OpenChord. En particulier, nous développons le nouveau service P2P-LTR (P2P Log and Timestamping for Reconciliation) qui permet la réconciliation pour l'édition collaborative.

Réplication Mono-Maître dans les Entrepôts de Données

Dans le contexte des entrepôts de données, qui servent principalement à la prise de décision, un objectif majeur est de concevoir des algorithmes de maintien de la cohérence des données répliquées : en minimisant le degré de fraîcheur ; en minimisant la surcharge de messages (éviter les votes) ; en étant assez extensible pour différentes configurations ; en tolérant les pannes des nœuds.

La minimisation du degré de fraîcheur compense la perte de la cohérence mutuelle due à la réplication asynchrone (car il y a perte de cohérence mutuelle). Les solutions existantes pour la gestion de la cohérence pour la réplication mono-maître sont spécifiques pour certaines configurations. La gestion de la cohérence est une tâche difficile lorsque le placement des données détermine le routage des messages et qu'un nœud tombe en panne. L'idée d'utiliser, de façon contrôlée, les services du réseau est alors très utile pour la réplication asynchrone. C'est la voie de recherche que nous avons choisie. Au-delà des algorithmes proposés, nous avons aussi proposé une architecture qui respecte l'autonomie des SGBD (vus comme des boites noires) et donc fonctionne avec tout type de SGBD.

Gestion de la Fraîcheur

Pour certaines applications, par ex. boursières, il est très important de minimiser le dégré de fraîcheur entre copies primaires (les copies en mise-à-jour) et copies secondaires (les copies en lecture seule). C'est pourquoi l'approche *push*, où les mises-à-jour sont envoyées par les noeuds maîtres (stockant des copies primaires), a été choisie. Un premier objectif est alors de minimiser le degré de fraîcheur pour les configurations de type diffusion et mono-consolidation.

Dès qu'une transaction *T* de mise-à-jour est validée sur le maître, ses mises-à-jour sont propagées vers les nœuds *cibles* dans un message, en *différé*. On appelle cette stratégie de propagation *différé-différé* Pour minimiser le degré de fraîcheur, nous avons proposé la propagation *immédiate*. L'idée est de propager les opérations de mises-à-jour d'une transaction *T*, qui met a jour une relation *R*, dès que la première opération d'écriture (noté *w*) sur R est détectée. Pour garantir la sérialisabilité des opérations d'écriture, nous utilisons la primitive de réseau FIFO [HT93]. Pour le nœud cible, trois stratégies de réception sont possibles [PSM98]:

- *immédiate* : dés que la première opération d'écriture *w* arrive, la transaction de mise-à-jour sur la copie secondaire *r* est déclenchée, mais seulement validée après la réception du *commit de T.*
- *différé*: le nœud cible attend la réception complète de la transaction (toutes les opérations une par une) puis déclenche la transaction sur *r.*
- *à posteriori* : le nœud cible attend la réception complète de la transaction, puis (à la réception) déclencher la transaction sur *r.*

La combinaison de la stratégie de propagation immédiate avec les stratégies de réception produisent trois stratégies: *immédiate-immédiate, immédiate-différé*, *immédiate-posteriori.* Nos résultats de performance (obtenus par simulation) ont montré que les stratégies immédiates peuvent améliorer jusqu'à 5 fois le degré de fraîcheur comparé à la stratégie *différé-différé.* De plus, le traitement de la tolérance aux pannes des nœuds peut être facilement implémenté en utilisant des journaux, sans bloquer les protocoles de réplication. L'Annexe A présente les détails de ces contributions qui correspond à la publication [PS00].

Gestion de la Cohérence des Données

Certaines architectures d'entrepôts de données mettent en œuvre des configuration de réplication de type multi-consolidation et triangulaire. Un problème important est alors d'assurer la cohérence sans imposer un chemin de routage pour les transactions de mise-à-jour. Pour les configurations triangulaires, la propriété de causalité doit aussi être assurée. La diffusion FIFO ne suffit plus pour assurer la cohérence. Pour résoudre ce problème, nous avons proposé d'utiliser des estampilles avec les services de réseau.

Bien que les services de réseau soient très attractifs, le nombre de messages augmente considérablement pour augmenter les garanties, limitant ainsi le passage à l'échelle. C'est pourquoi nous avons évité la primitive *ordre total* (très chère en messages) et utilisé la primitive FIFO (beaucoup plus efficace). Pour garantir l'ordre total, des estampilles globales *C* sont données aux transactions de mise-à-jour au moment de la validation de chaque transaction sur les nœuds maîtres. Pour cela, on suppose un système semi-asynchrone où les horloges sont ε-synchronisés et le temps de transmission maximal d'un message (*Max*) est connu [BGM04]. Pour gérer la cohérence des copies sur les nœuds cible, nous avons proposé l'algorithme *refresher* [PMS99, PMS01].

Les transactions de mises-à-jour sont exécutées sur le noeud maître puis sont propagées à toutes les copies sous forme de transactions de rafraîchissement. À la réception de l'une de ces transactions, le noeud cible place la transaction dans une file d'attente (il y a une file d'attente par maître que possède la copie primaire). La transaction de rafraîchissement attend alors un temps *Max* avant d'être élue et placée dans une file d'exécution. En attendant un temps *Max* après son départ, on s'assure alors qu'aucun message n'a été émis auparavant et transite encore sur le réseau (le réseau est fiable et un message met au maximum un temps *Max* pour arriver à destination). Le moment exact où le message sera élu pour exécution est donc: $C + Max + \varepsilon$. Cet algorithme assure l'ordre total des transactions de rafraîchissement. L'Annexe B présente les détails de ces contributions qui correspond à la publication [PMS01]

Réplication Multi-Maître dans les Grappes de Bases de Données

Pour les grappes de bases de données, nous avons adopté la réplication multi-maître qui permet de mettre à jour en parallèle (sur différents nœuds) les copies d'un même objet et ainsi de maximiser les performances. Nous avons proposé un nouvel algorithme de *réplication préventif*, en partant de l'algorithme *refresher*. Le principe de l'algorithme *refresher* est de soumettre les transactions dans un ordre total sur les noeuds cibles en fonction de leur estampille d'arrivée. Pour accomplir ceci, le nœud cible retarde l'exécution des transactions pour s'assurer qu'aucune transaction plus ancienne n'est en route vers le noeud. Cependant, cet algorithme n'autorise que la mise-à-jour sur un seul nœud (mono-maître). Nous abordons donc le problème où plusieurs maîtres peuvent faire des mises jour en parallèle.

Réplication Preventive

La réplication préventive [POC03, PCVO05] assure la cohérence forte (il n'y a jamais d'incohérences) pour les configurations multi-maîtres (en réplication totale et partielle). Pour la réplication totale (les données sont répliquées sur tous les nœuds), lorsqu'une transaction *T* arrive dans la grappe elle est diffusé en FIFO à *tous les nœuds* de la grappe y compris le nœud qui a reçu la transaction. Chaque nœud de la grappe retarde l'exécution de *T*, comme auparavant, et la cohérence forte est assurée pour les mêmes raisons. En autorisant ainsi davantage de noeuds multi-maîtres, nous supprimons le goulot d'étranglement que représente un seul noeud maître. Cependant, le surcoût en mise-à-jour sur tous les nœuds peut être important, d'où la nécessité de la réplication partielle (les données sont répliquées sur certains nœuds).

Pour la réplication partielle [CPV05a, CPV05b], nous ne faisons pas de restrictions ni sur le type de copies (copie primaire ou multi-maître) ni sur le placement des données (un noeud peut posséder une partie seulement des copies et plusieurs types de copie: primaire, multi-maître, secondaire). Cependant, un noeud peut ne pas être en mesure d’exécuter toutes les transactions car il ne possède pas toutes les copies nécessaires. L'algorithme place alors la transaction en attente des jeux d'écritures qui sont diffusés par le noeud d'origine. L'algorithme de réplication partielle autorise donc un plus grand nombre de configurations mais introduit la diffusion d'un message de rafraîchissement.

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

Optimisations

Afin de mieux supporter les applications à fortes charges transactionnelles où les mise-à-jour sont majoritaires, nous avons amélioré l'algorithme de réplication préventive. Nous avons dans un premier temps éliminé le délai introduit par l'ordonnancement des transactions en les exécutant de manière optimiste dès leur réception dans le noeud et non plus après le délai $Max + \varepsilon$. Si les transactions n'ont pas été exécutées dans l'ordre correct (celui de leurs estampilles), alors elles sont annulées et ré-exécutées après ordonnancement. Le nombre d'abandons reste faible car dans un réseau rapide et fiable les messages sont naturellement ordonnés [PS98]. Avec cette optimisation, la cohérence forte est garantie car nous retardons la validation des transactions (et non plus la totalité de la transaction) exécutées de façon optimiste. Les transactions sont ordonnancées pendant leur exécution et non plus avant, supprimant ainsi les délais d'ordonnancement.

La seconde optimisation concerne la soumission des transactions. Dans les algorithmes précédents, les transactions sont soumises à exécution une par une pour garantir la cohérence. Le module de soumission représente donc un goulot d'étranglement dans le cas où le temps moyen d'arrivée des transactions est supérieur au temps moyen d'exécution d'une transaction. Pour supprimer ce problème, nous avons autorisé l'exécution parallèle des transactions non conflictuelles. Cependant, pour garantir la cohérence des données, nous ordonnançons toujours le démarrage et la validation des transactions, ceci afin de garantir que toutes les transactions soient exécutées dans le même ordre sur tous les noeuds malgré le parallélisme. Nous avons prouvé que l'exécution en parallèle des transactions est équivalente à une exécution séquentielle. L’Annexe C présente les détails de ces contributions qui correspond à la publication [PCVO05].

Validation

Nous avons validé l'algorithme de réplication préventive en testant trois propriétés recherchées pour un algorithme de réplication asynchrone en grappe:

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

De plus, nous avons montré que nos optimisations permettent un meilleur support face aux fortes charges. En effet, en exécutant les transactions en parallèle et en éliminant le délai d'ordonnancement, notre système supporte mieux l'émission massive de transactions. Les expérimentations ont prouvé que l'exécution optimiste des transactions n'entraînait un taux très faible d’abandons (1%), ce qui rend notre optimisation viable. Finalement, nous avons développé le prototype RepDB* [CGPV04] qui implémente l'algorithme de réplication préventive avec toutes les optimisations.

Réconciliation de Données pour les Application Collaboratives en P2P

Les wikis sont maintenant très utilisés pour l’édition collaborative de documents sur le Web mais s’appuient sur un site central, qui peut être un goulot d’étranglement et un point critique en cas de panne. Une approche P2P permet de pallier à ces problèmes et offre d’autres avantages comme un meilleur contrôle des données privées (qui restent locales) et le support du travail en mode déconnecté. A partir d’une application de Wiki en P2P [Wik07], nous pouvons résumer les besoins de réplication pour les applications collaboratives comme suit : haut niveau d’autonomie, réplication multi-maître, détection et résolution de conflits, cohérence éventuelle des répliques, et indépendance des types de données.

La réplication optimiste supporte la plupart de ces besoins en permettant la mise-à-jour asynchrone des répliques de sorte que les applications puissent progresser même si quelques nœuds sont déconnectés ou en panne. En conséquence, les utilisateurs peuvent collaborer de manière asynchrone. Cependant, les solutions optimistes existantes sont peu applicables aux réseaux P2P puisqu'elles sont centralisées ou ne tiennent pas compte des limitations du réseau. Les approches centralisées sont inadéquates en raison de leur disponibilité limitée et de leur vulnérabilité aux fautes et aux partitions du réseau. D'autre part, les latences variables et les largeurs de bande, typiques des réseaux P2P, peuvent fortement influencer sur les performances de réconciliation puisque les temps d'accès aux données peuvent changer de manière significative de nœud à nœud. Par conséquent, afin d'établir une solution appropriée de réconciliation P2P, des techniques optimistes de réplication doivent être revues.

Motivé par ce besoin, nous avons proposé une solution hautement disponible de réconciliation et qui passe à l'échelle pour des applications de collaboration P2P. Pour ce faire, nous proposons des protocoles de réconciliation basés sur la sémantique qui assurent la cohérence éventuelle des répliques et tiennent compte des coûts d'accès aux données.

Réconciliation Sémantique Distribuée (DSR)

L'algorithme DSR (*Distributed Semantic Reconciliation*) [MPV05] utilise le modèle *action-contrainte* proposé pour le système IceCube [KRSD01, PSM03, SBK04] afin de capturer la sémantique de l'application et résoudre les conflits de mise-à-jour. Cependant, DSR est tout à fait différent d'IceCube car il adopte des hypothèses différentes et fournit des solutions distribuées. Dans IceCube, un seul nœud centralisé prend des actions de mise-à-jour de tous les autres nœuds pour produire un ordonnancement global. Ce nœud peut être un goulot d'étranglement. D'ailleurs, si le nœud qui fait la réconciliation tombe en panne, le système entier de réplication peut être bloqué jusqu'au rétablissement. En revanche, DSR est une solution repartie qui tire profit du traitement parallèle pour fournir la haute disponibilité et le passage à l'échelle.

Nous avons structuré l'algorithme DSR en 5 étapes reparties pour maximiser le traitement parallèle et pour assurer l'indépendance entre les activités parallèles. Cette structure améliore les performances et la disponibilité de la réconciliation (c.-à-d. si un nœud tombe en panne, l'activité qu'il était en train d'exécuter est attribuée à un autre nœud disponible).

Avec DSR, la réplication de données se passe comme suit. D'abord, les nœuds exécutent des actions locales pour mettre à jour une réplique d'un objet tout en respectant des contraintes définies par l'utilisateur. Puis, ces actions (avec les contraintes associées) sont stockées dans une table de hachage distribuée (DHT) en se basant sur l'identifiant de l'objet. Enfin, les nœuds réconciliateurs retrouvent les actions et les contraintes dans la DHT et produisent un ordonnancement global en réconciliant les actions conflictuelles. Cette réconciliation est complètement distribuée et l'ordonnancement global est localement exécuté dans chaque nœud, assurant de ce fait la cohérence éventuelle [SBK04, SS05].

Dans cette approche, nous distinguons trois types de nœuds : le *nœud de réplique*, qui tient une réplique locale ; le *nœud réconciliateur*, qui est un nœud de réplique qui participe à la réconciliation distribuée ; et le *nœud fournisseur,* qui est un nœud dans la DHT qui stocke des données consommées ou produites par les nœuds réconciliateurs (par ex., le nœud qui tient l'ordonnancement s'appelle le *fournisseur d'ordonnancement*).

Nous concentrons le travail de réconciliation dans un sous-ensemble de nœuds (les nœuds réconciliateurs) pour maximiser les performances. Si nous ne limitons pas le nombre de nœuds réconciliateurs, les problèmes suivants peuvent survenir. D'abord, les nœuds fournisseurs et le réseau entier deviennent surchargés à cause d'un grand nombre de messages visant à accéder au même sous-ensemble d'objets dans la DHT pendant un intervalle très court de temps. Ensuite, les nœuds avec de hautes latences et de faibles bandes passantes peuvent gaspiller beaucoup de temps avec le transfert de données, compromettant de ce fait le temps de réconciliation. Notre stratégie ne crée pas des déséquilibres dans la charge des nœuds réconciliateurs car les activités de réconciliation ne sont pas des processus intensifs.

Réconciliation P2P (P2P-Reconciler)

P2P-reconciler transforme l'algorithme DSR en protocole de réconciliation en développant des fonctionnalités additionnelles que DSR ne fournit pas. D'abord, il propose une stratégie pour calculer le nombre de nœuds qui devraient participer à la réconciliation afin d'éviter des surcharges de messages et assurer de bonnes performances [MAPV06, MPV06a]. En second lieu, il propose un algorithme distribué pour choisir les meilleurs nœuds réconciliateurs basés sur les coûts d'accès aux données, qui sont calculés selon les latences de réseau et les taux de transfert. Ces coûts changent dynamiquement pendant que les nœuds rejoignent et partent du réseau, mais notre solution fait face à un tel comportement dynamique. Troisièmement, il garantit la cohérence éventuelle des répliques en dépit de jonctions et départs autonomes des nœuds [MAPV06, MP06, MPV06a, MPJV06]. En outre, nous avons formellement montré que P2P-reconciler assure la cohérence éventuelle, est fortement disponible, et fonctionne correctement en présence des fautes.

Réconciliation consciente de la topologie (P2P-reconciler-TA)

P2P-reconciler-TA [EMP07] est une optimisation du protocole P2P-reconciler qui vise à exploiter les réseaux P2P conscients de leurs topologies (en anglais, *topology-aware P2P networks*) pour améliorer les performances de réconciliation. Les réseaux P2P conscients de leurs topologies établissent les voisinages parmi les nœuds basés sur des latences de sorte que les nœuds qui sont proches les uns des autres en termes de latence dans le réseau physique soient aussi des voisins dans le réseau P2P logique. Pour cette raison, des messages sont routés plus efficacement sur les réseaux conscients de leurs topologies. L'algorithme DSR n'est pas affecté par la topologie du réseau. Cependant, un autre algorithme est nécessaire pour le choix des nœuds qui participent à la réconciliation.

Plusieurs réseaux P2P conscients de leurs topologies peuvent être employés pour valider notre approche telle que Pastry [RD01a], Tapestry [ZHSR+04], CAN [RFHK+01], etc. Nous avons choisi CAN parce qu'il permet de construire le réseau P2P logique conscient de sa topologie d'une façon assez simple. De plus, il est facile de mettre en œuvre son mécanisme de routage, bien que moins efficace que d'autres réseaux P2P conscients de leurs topologies (par ex., le chemin de routage moyen dans CAN est habituellement plus long que dans d'autres réseaux P2P structurés).

Les protocoles P2P-reconciler et P2P-reconciler-TA tirent profit de l'algorithme DSR pour réconcilier des actions conflictuelles. Cependant, ils sont très différents par rapport à l'allocation de nœuds réconciliateurs. P2P-reconciler-TA choisit d'abord les nœuds fournisseurs qui sont proches les uns des autres et sont entourés par un nombre acceptable de réconciliateurs potentiels. Puis, il transforme des réconciliateurs potentiels en réconciliateurs candidats. Au fur et à mesure que la topologie du réseau change suite à des jonctions, départs, et échecs de nœuds, P2P-reconciler-TA change également les nœuds fournisseurs choisis et les réconciliateurs candidats associés. Ainsi, les fournisseurs et les réconciliateurs candidats choisis changent d'une façon dynamique et auto-organisée selon l'évolution de la topologie du réseau. P2P-reconciler-TA choisit des nœuds réconciliateurs à partir de l'ensemble de réconciliateurs candidats en appliquant une approche heuristique qui réduit rigoureusement l'espace de recherche et préserve les meilleures options. En outre, ce protocole également assure la cohérence éventuelle des répliques, rend la réconciliation hautement disponible même pour les réseaux très dynamiques, et fonctionne correctement en présence d'échecs. L'Annexe D présente les détails de toutes ces contributions qui correspond à la publication [MPEJ08].

Validation

Nous avons validé nos algorithmes par la création d'un prototype et d'un simulateur. Le prototype sur la plateforme distribuée Grid5000 nous a permis de vérifier l'exactitude de notre solution de réplication et de calibrer le simulateur. D'autre part, le simulateur a permis d'évaluer le comportement de notre solution sur des réseaux plus grands. L'évaluation de performances de DSR a montré qu'il surpasse la réconciliation centralisée en réconciliant un grand nombre d'actions. En outre, il fournit un plus grand degré de disponibilité, de passage à l'échelle, et de tolérance aux fautes. D'ailleurs, il passe à l'échelle très bien jusqu'à 128 nœuds réconciliateurs. Puisque le nombre de nœuds réconciliateurs ne limite pas le nombre de nœuds de répliques, il s'agit d'un excellent résultat.

P2P-reconciler a été évalué avec des méthodes distinctes d'allocation de nœuds réconciliateurs. Les résultats expérimentaux ont prouvé que la réconciliation avec l'allocation basée sur le coût surpasse l'approche aléatoire par un facteur de 26. De plus, le nombre de nœuds connectés n'est pas important pour déterminer les performances de réconciliation. Ceci est du au fait que la DHT passe à l'échelle et les réconciliateurs sont aussi proches que possible des objets de réconciliation. Par ailleurs, la taille des actions affecte le temps de réconciliation dans une échelle logarithmique. En conclusion, P2P-reconciler restreint la surcharge du système puisqu'il calcule des coûts de communication en employant des informations locales et limite la portée de la propagation des événements (par ex., jonction ou départ).

Nos résultats expérimentaux ont prouvé que P2P-reconciler-TA sur CAN surpasse P2P-reconciler par un facteur de 2. C'est un excellent résultat si nous considérons que P2P-reconciler est déjà un protocole efficace et CAN n'est pas le réseau P2P conscient de topologie le plus efficace (par ex., Pastry et Tapestry sont plus efficaces que CAN). P2P-reconciler-TA exploite d'une manière très appropriée les réseaux conscients de topologie puisque ses meilleures performances sont obtenues quand le degré de proximité parmi les nœuds en termes de latence est le plus élevé. De plus, il passe à l'échelle au fur et à mesure que le nombre de nœuds connectés augmente. En conclusion, l'approche heuristique de P2P-reconciler-TA pour choisir les nœuds réconciliateurs est très efficace.

Gestion de Données Courantes dans les DHTs Répliquées

Les DHTs, comme CAN [RFHKS01] et Chord [SMKK+01], fournissent une solution efficace pour la recherche de données dans les systèmes P2P. Une DHT réalise une conversion entre une clef *k* et un pair *p*, appelé le responsable pour *k*, en utilisant une fonction de hachage, et permet ainsi de trouver efficacement le pair qui est responsable pour une clef. Les DHTs fournissent typiquement deux opérations de base [HHHL+02]: put (*k*, data) stocke une clef *k* et une donnée *data* dans la DHT en utilisant une fonction de hachage; *get (k)* recherche la donnée stockée dans la DHT avec la clef *k*. Cependant, la disponibilité des données stockées n'est pas garantie. Pour améliorer la disponibilité des données, nous pouvons répliquer la paire (k, data) sur plusieurs pairs. Toutefois, la cohérence des répliques après mise-à-jour peut être compromise en raison des pairs qui ont quitté le réseau ou des mises-à-jour concurrentes.

Nous avons proposé une solution complète pour déterminer les données courantes (les plus à jour) parmi les données répliquées dans les DHTs [APV07a]. Cette solution consiste en un service de gestion des mises-à-jour, UMS (*Update Management Service*), qui permet de gérer des données répliquées et de retrouver les copies courantes (les plus à jour). Pour ce faire, UMS s'appuie sur un service d'estampillage basé sur clef, KTS (*Key-based Timestamp Service*), qui permet de générer des estampilles logiques d'une façon complément distribuée.

Gestion des Mises-à-jour (UMS)

UMS permet d'insérer (opération insert) une donnée et sa clef en la répliquant dans la DHT et de retrouver (opération retrieve) la donnée répliquée la plus courante correspondant à une clef donnée. Le fonctionnement de UMS peut être résumé comme suit. Soit *H* un ensemble de fonctions de hachage. Pour chaque clef *k* et chaque fonction de hachage *h*, il y a un pair *p* responsable pour *k*. Nous appelons *p* le responsable de *k* par rapport à *h*, et le dénotons par *rsp(k, h)*. Un pair peut être responsable pour *k* par rapport à une fonction h_1 mais non responsable pour *k* par rapport à une autre fonction h_2. Pour améliorer la disponibilité des données, le service UMS stocke chaque donnée sur plusieurs pairs en utilisant un ensemble de fonctions de hachage $H_r \subset H$. l'ensemble H_r s'appelle *l'ensemble de fonctions de réplication*. Le nombre de fonctions de réplication, c.-à-d. $|H_r|$, peut être différent pour différents réseaux P2P.

Des répliques peuvent devenir *obsolètes*, par exemple en raison de l'absence de certains pairs au moment de la mise-à-jour. Afin de distinguer entre les répliques *courantes* (à jour) et obsolètes, avant de stocker les données, UMS leur ajoute une estampille logique qui est produite par le service KTS. En prenant une clef *k* et une donnée *data*, pour chaque $h \in H_r$, UMS stocke (*k, {data, estampille}*) sur le pair *rsp(k, h)*. Suite à une demande pour une donnée qui est stockée avec une clef sur la DHT, UMS renvoie l'une des répliques qui a une estampille récente.

Estampillage (KTS)

L'opération principale de KTS est l'opération *gen_ts (k)* qui, en prenant une clef *k*, produit un nombre réel en tant qu'estampille pour *k*. Les estampilles produites par KTS ont la propriété de monotonie, c.-à-d. que les estampilles produites pour la même clef sont monotonement croissantes.

KTS produit les estampilles d'une façon complètement distribuée, en utilisant les compteurs locaux. A chaque moment, il produit au maximum une estampille pour une clef *k*. Donc, en considérant la propriété de monotonie, il y a un ordre total sur l'ensemble des estampilles produites pour une clef. Cependant, il n'y a aucun ordre total sur les estampilles produites pour des clefs différentes. En plus de *gen_ts(k)*, KTS a une autre opération dénotée par *last_ts (k)* qui en prenant une clef *k*, renvoie la dernière estampille produite pour *k*.

L'opération *gen_ts(k)* produit des estampilles monotonement croissantes pour les clefs. Une solution centralisée pour produire les estampilles n'est évidemment pas viable dans un système P2P car le pair central serait un goulot d'étranglement et un point d'échec. Et les solutions distribuées à l'aide d'horloges synchronisées ne s'appliquent pas non plus dans un système P2P. Nous avons proposé alors une technique distribuée pour produire des estampilles dans les DHTs. Elle utilise des compteurs locaux pour produire des estampilles et des algorithmes d'initialisation des compteurs qui garantissent la monotonie des estampilles, même en cas de pannes. L'Annexe E présente les détails de toutes ces contributions qui correspond à la publication [APV07a].

Validation

Nous avons validé notre solution par une implémentation sur un cluster de 64 nœuds de Grid5000 et par une simulation jusqu'à 10.000 pairs en utilisant SimJava [HM98]. Nous avons réalisé les services UMS et KTS sur notre propre implémentation de Chord [SMKK+01], une DHT simple et efficace.

Nous avons comparé les performances de notre service UMS avec celles de BRK (un service réalisé dans le projet BRICK [KWR05]). Les résultats d'expérimentation et de simulation montrent que KTS et UMS donnent des gains majeurs, en termes de temps de réponse et de coût de communication, par rapport à BRK. Le temps de réponse et le coût de communication d'UMS ont une tendance logarithmique en nombre de pairs de la DHT. L'augmentation du nombre de répliques n'a pas d'impact sur le temps de réponse et le coût de communication de UMS. Mais cette augmentation a un grand impact négatif sur BRK. Nous avons aussi testé UMS et KTS dans des situations où la DHT est très dynamique, c.-à-d. les pairs rejoignent et quittent le système fréquemment. Les résultats ont montré que même dans ces situations, UMS et KTS fonctionnent très bien. En résumé, notre validation a montré que la gestion des données courantes peut être supportée efficacement dans les DHTs.

Conclusions et Travaux Futurs

Dans le contexte des *entrepôts de données* nous avons proposé des algorithmes pour la gestion de la cohérence pour la réplication asynchrone mono-maître. Le principe de l'algorithme *refresher* est de soumettre les transactions dans un ordre total sur tous les nœuds cibles en fonction de leur estampille d'arrivée. Pour ce faire, l'algorithme refresher retarde (avec un délai d'attente) l'exécution des transactions pour s'assurer qu'aucune transaction plus ancienne n'est en route pour le noeud. L'algorithme refresher impose une restriction sur le placement des données (le graphe de dépendances des noeuds ne doit pas former de cycles). Nos résultats de performance (obtenus par simulation) ont montré que des stratégies immédiates peuvent améliorer jusqu'à 5 fois le degré de fraîcheur comparé à des stratégies différées.

Dans le contexte des *grappes de bases de données***,** nous avons présenté un nouvel algorithme de réplication dite *préventive*, basés sur l'algorithme *refresh*. Nous avons ajouté le support de la réplication totale et partielle ainsi que d'importantes optimisations pour relâcher les délais d'attente. Nous avons montré que ces algorithmes avaient de très bonnes performances et passaient à l'échelle. Nous avons réalisé ces algorithmes dans le prototype RepDB* avec les SGBD PostGreSQL et BerkeleyDB.

Dans le contexte des *applications collaboratives en P2P*, nous avons proposé une solution de réconciliation fortement disponible qui passe à l'échelle et assure la cohérence éventuelle. Nous avons proposé l'algorithme DSR qui peut être exécuté dans différents environnements distribués (grappe, grille, ou P2P). Nous avons étendu DSR en un protocole de réconciliation P2P appelé P2P-reconciler. Puis nous avons proposé le protocole P2P-reconciler-TA, qui exploite les réseaux P2P conscients de leur topologie afin d'améliorer les performances de la réconciliation. Nous avons validé nos solutions et évalué leurs performances par un prototype que nous avons déployé sur la plateforme Grid5000 et simulation. Les résultats ont montré que notre solution de réplication apporte haute disponibilité, excellent passage à l'échelle, avec des performances acceptables et surcharge limitée.

Nous avons aussi proposé une solution complète au problème d'accès à des données courantes dans les DHTs [APV07a]. Nous avons proposé le service UMS qui permet de mettre à jour les données répliquées et l'accès efficace à des répliques courantes en utilisant une approche basée sur l'estampillage. Après la récupération d'une réplique, UMS détecte si elle est courante ou pas, c.-à-d. sans devoir la comparer avec les autres répliques. Contrairement aux travaux existants, par exemple [KWR05], UMS n'a pas besoin de rechercher toutes les répliques pour trouver une réplique courante. En outre, les mises-à-jour concurrentes ne posent aucun problème pour UMS. Nous avons également proposé le service KTS qui produit des estampilles monotonement croissantes, de façon distribuée en utilisant les compteurs locaux. Nous avons validé UMS et KTS par une combinaison d'implémentation avec la DHT Chord sur un cluster de Grid5000 et simulation. Les résultats ont montré l'efficacité de ces deux services.

Nos techniques de réplication en P2P ont été conçues dans le contexte du projet APPA [AMPV04 AMPV06a] qui fournit des services avancés de gestion de données dans les systèmes P2P. Dans le prolongement de nos travaux en réplication, nous poursuivons trois directions de recherche nouvelles dans APPA : un service de réconciliation basé sur des estampilles distribuées continues, un service de gestion de cache, et un service de gestion de flux en P2P.

Réconciliation P2P

Nous proposons une solution nouvelle à la réconciliation en P2P pour l'édition collaborative (dans le cadre du projet Xwiki Concerto). Elle combine l'approche à base de transformées opérationnelles (OT) [SCF98, FVC04 MOSI03] qui permet de réaliser la réconciliation de façon très efficace et simple, et un nouveau service appelé P2P-LTR (Logging and Timestamping for Reconciliation). P2P-LTR réalise la journalisation des actions en P2P et intègre un service d'estampillage continu inspiré de KTS et un algorithme pour retrouver l'ordre total des actions stockées dans le journal P2P. Un défi est la gestion d'estampilles continues qui doit être tolérante aux fautes, afin d'ordonnancer les actions (les modifications sur les documents) stockées dans les journaux.

Gestion de Cache en P2P

Bien que largement déployés pour le partage de fichiers, les systèmes P2P non structurés surexploitent les ressources réseaux. Le trafic P2P monopolise la bande passante, notamment à cause de l'inefficacité des mécanismes de recherche aveugle qui inondent le réseau de messages redondants.

Puisque les requêtes dans ces systèmes exhibent une forte localité temporelle, les techniques de mise en cache des réponses de requêtes (c.à.d. la localité des fichiers) pourraient optimiser la recherche et limiter le trafic redondant. En revanche, la gestion de caches constitue un défi majeur pour éviter le surcoût de stockage. De plus, la majorité des approches existantes ne prennent pas en compte la proximité réseau entre le client et le fournisseur du fichier, alors que les fichiers populaires sont naturellement répliqués en différentes localités. Ce problème contribue également à la surcharge du réseau et à la dégradation des temps de réponse. Nous proposons d'étudier des techniques de recherche qui profitent de la mise en cache des réponses des requêtes. L'idée est d'exploiter la réplication désordonnée de fichiers en considérant leur localité, pour limiter la consommation de bande passante dans les systèmes P2P non structurés.

Nous nous intéressons aussi à gestion de cache web en P2P. Plusieurs travaux sur les caches web comme Akamai redistribuent le contenu des serveurs web pour un public plus large. Ces techniques absorbent la surcharge des serveurs webs, limitent les coûts de bande passante et optimisent le temps de latence perçus par les clients. Cependant, ces services sont coûteux en termes de maintenance et d'administration. La technologie pair-à-pair est une nouvelle alternative pour redistribuer le contenu à une grande échelle et à bas coûts, tout en exploitant les ressources non utilisées des clients. Plusieurs défis se présentent lors de la conception d'un système de cache web P2P avec des performances comparables à celles des techniques traditionnelles, alors que l'on compte essentiellement sur des nœuds autonomes et dynamiques.

Plus précisément, nous abordons les questions suivantes : quelles données doivent être mises en cache, sur quels nœuds placer le cache, comment gérer les mises-à-jour (et assurer la cohérence des données répliquées), et comment traiter efficacement le routage de requêtes en exploitant le système de caches distribués .

Gestion de Flux de Données en P2P

Des applications modernes comme la surveillance de réseau, l'analyse financière ou les réseaux de capteurs requièrent des requêtes continues pour traiter des flux de données (*data streams*). La gestion des flux de tuples produits en continu et de taille non bornée est un domaine de recherche important. La nature continue et non bornée des données a pour conséquence qu'il n'est pas possible de stocker les données sur disque. De plus, cela empêche l'application des optimisations développées sur les opérateurs traditionnels de gestion de données. Il n'est donc pas concevable de traiter les données avec une approche classique.

D'autre part, le succès des applications P2P et ses protocoles a motivé d'aller au-delà des applications de partage de fichiers. C'est ainsi que le paradigme P2P s'est récemment imposé comme la clé du passage à l'échelle dans les systèmes distribués. Dans cette direction, nous intéressons à la conception et l'implémentation d'algorithmes efficaces pour le traitement de requêtes sur des flux de données reposant sur le paradigme P2P.

1 Introduction

In this chapter, we present the main structure and goals of this research work on lazy data replication in different contexts: small, large and very large scale systems. We first introduce the main concepts, relevant terms and some related work. Then we present the context of our research which includes the projects we have been involved. Finally, we present the report organization.

1.1 Data Replication

Data replication is important in the context of distributed systems for several reasons [SS05]. First, replication improves system availability by removing single points of failures (objects are accessible from multiple nodes). Second, it enhances system performance by reducing the communication overhead (objects can be located closer to their access points) and increasing the system throughput (multiple nodes serve the same object simultaneously). Finally, replication improves system scalability as it supports the growth of the system with acceptable response times.

Data replication consists of managing reads and writes over multiple copies of a single object, called *replicas*, stored in set of interconnected nodes. An *object* is the minimal unit of replication in a system. For instance, in a replicated relational database, if tables are entirely replicated then tables correspond to objects; however, if it is possible to replicate individual tuples, then tuples correspond to objects. Other examples of objects include XML documents, typed files, multimedia files, etc.

A major issue concerning data replication is how to manage updates. Gray et al. [GHOS96] classify the replica control mechanisms according to two parameters: *where* updates take place (i.e. which replicas can be updated), and *when* updates are propagated to all replicas. According to the first parameter (i.e. where), replication protocols can be classified as *single-master* or *multi-master* solutions. With single master replication, updates on a replicated object are performed at a single master node, which holds the primary copy (read/write) of the replicated object. Slave nodes, holds secondary copies (read only) of the replicated object. With multi-master replication, all involved nodes holds primary copies of replicated objects. According to the second parameter (i.e. when), update propagation strategies are divided into *synchronous* and *asynchronous* approaches. More specifically, in distributed database systems, data access is done via transactions. A *transaction* is a sequence of read and write operations followed by a *commit*. If the transaction does not complete successfully, we say that it *aborts*. The updates of a transaction that updates a replicated object must be propagated to all nodes that hold replicas of this object in order to keep these replicas consistent. Such update propagation can be done within the transaction boundaries or after the transaction commit. The former is called *synchronous* (henceforth eager) *replication*, and the latter, *asynchronous* (hence forth lazy) replication.

The replica control solutions are also affected by the way replicas are distributed over the network. Replica placement over the network directly affects the replica control mechanisms. We discuss the basic alternative approaches for replica placement: full replication and partial replication.

Full replication consists of storing a copy of every replica of an object at all participating nodes. This approach provides simple load balancing since all nodes have the same capacities, and maximal availability as any node can replace any other node in case of failure. Figure 1.1 presents the full replication of two objects R and S over three nodes.

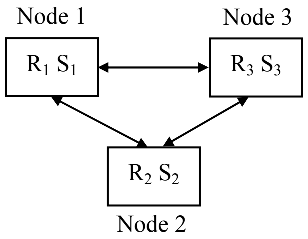


Figure 1.1. Example of full replication with two objects *R* and *S*

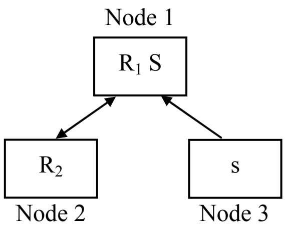


Figure 1.2. Example of partial replication with two objects *R* and *S*

With *partial replication*, each node may hold a subset of replicated copies, so that the objects replicated at one node may be different of the objects replicated at another node, as shown in Figure 1.2. This approach incurs less storage space and reduces the number of messages needed to update replicas since updates are only propagated towards some nodes. Thus, updates produce less load for the network and nodes. However, if related objects are stored at different nodes, the propagation protocol becomes more complex as the replica placement must be taken into account. In addition, this approach limits load balance possibilities since certain nodes are not able to execute a particular set of transactions.

One of the crucial problems of data replication is updating a replica. The *eager* replication approaches apply updates to all replicas within the context of the transaction that initiates the updates, as shown in Figure 1.3. As a result, when the transaction commits, all replicas have the same state and *mutual consistency* is assured. This is achieved by using concurrency control mechanisms like distributed two-phase-locking (D-2PL) [OV99] or timestamp based algorithms. In addition, a commitment protocol like two-phase-commit (2PC) can be used to provide atomicity (either all transaction's operations are completed or none of them are). Thus, eager replication enforces mutual consistency among replicas. Bernstein et al. [BHG87] define this consistency criteria as *one-copy-serializability*, i.e. despite the existence of multiple copies, an object appears as one logical copy (*one-copy-equivalence*), and a set of accesses to the object on multiple nodes is equivalent to serially execute these accesses on a single node.

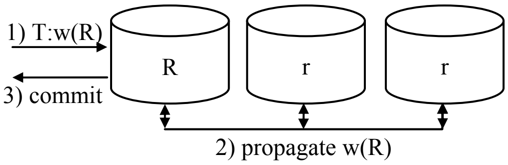


Figure 1.3. Principle of lazy replication

Early solutions [Sto79] use synchronous single-master approaches to assure one-copy-serializability. However, later solutions avoid this centralized solution and follow the multi-master approach. For instance, in the ROWA (*read-one/write-all*) approach [BHG87], read operations are done locally while write operations access all copies. ROWA is not fault-tolerant since the update processing stops whenever a copy is not accessible. ROWAA (*read-one/write-all-available*) [BG84,GSC+83] overcomes this limitation by updating only the available copies. Another alternative are quorum protocols [Gif79,JM87,PL88,Tho79], which can succeed as long as a quorum of copies agrees on executing the operation. Other solutions combine ROWA/ROWAA with quorum protocols [ET89].

More recently, Kemme and Alonso [KAa00] proposed new protocols for eager replication that take advantage of group communication systems to avoid some performance limitations introduced by the standard eager solutions when using D-2PL and 2PC. Group communication systems [CKV01] provide group maintenance, reliable message exchange, and message ordering primitives between groups of nodes. The basic mechanism behind these protocols is to first perform a transaction locally, deferring and batching writes to remote replicas until transaction commit time. At commit time all updates (called the write set) are sent to all replicas using a total order multicast primitive which guarantees that all nodes receive all write sets in exactly the same order. As a result, no two-phase commit protocol is needed and no deadlock can occur. Following this approach, Jiménez-Peris et al. [JPAK03] show that the ROWAA approach, instead of quorums, is the best choice for a large range of applications requiring data replication in cluster environments. Next, in [LKPJ05] the most crucial bottlenecks of the existing protocols are identified, and optimizations are proposed to alleviate these problems, making *one-copy-serializability* feasible in wide-area-networks (WAN) environments of medium size. Finally, [LKPJ07] proposes a replication solution adapted to edge computing which provides higher scalability with out the use of group communication at the WAN.

The main advantage of eager replication is to provide mutual consistency within transaction management. This enables local queries to read consistent values. The drawback is that the transaction has to update all replicas before committing, thus increasing transaction response times, with limited scalability (up to ten nodes). In addition, in the presence of failures these protocols may block. The use of group communication improves these limitations somewhat. A group communication system enables a node to multicast a message to all nodes of a group with a delivery guarantee, i.e. the message is eventually delivered to all nodes. Furthermore, it can provide multicast primitives with different delivery orders. In total order multicast, all messages sent by different nodes are delivered in the same total order at all nodes. In eager replication, this primitive was proposed [KA00a] to be used to guarantee that all nodes receive the write operations in exactly the same order, thereby ensuring identical serialization order at each node. In fact, the combination of the use reliable delivery (if one node receives the message all nodes receive the message unless they fail) and total order guarantees atomicity.

With *lazy replication*, a transaction commits as soon as possible at the master node, and afterwards the updates are propagated towards all other replicas, as shown in Figure 1.4, and replicas are then updated in a separatetransactions. As a consequence mutual consistency is relaxed. The concept of *freshness* is used to measure the deviation between replicas (primary copies and secondary copies) when using lazy single master replication.

Multi-master lazy replication (update anywhere) solutions can be classified as *optimistic* or *non-optimistic* according to their way of handling conflicting updates. In general, *optimistic replication* relies on the optimistic assumption that conflicting updates will occur only rarely, if at all. Tentative updates are applied locally, and later on conflicts are detected and resolved by some reconciliation engine [SS05]. As a result tentative updates may be aborted if necessary, and *eventually* mutual consistency is assured. Between two reconciliations, inconsistent states are allowed to be read by queries.

In contrast, *non-optimistic replication* assumes that update conflicts are likely to occur and implements propagation mechanisms that prevent conflicting by establishing some transaction execution strategy, in the fly, to assure total order. These protocols are said to assure *strong consistency* since an inconsistent state is never seen by queries. However, *unfresh* states are allowed to be read. This is why strong consistency is different from mutual consistency.

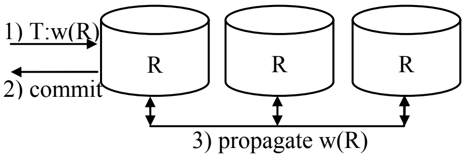


Figure 1.4. Principle of asynchronous propagation

An advantage of lazy propagation is that the replica protocol does not block transaction processing due to unavailable replicas, which improves system *liveness* in the presence of failures and dynamicity. In addition, less communication overhead is needed to coordinate concurrent updates, thereby reducing the transaction response times and improving the system scalability. In particular, *optimistic lazy* replication is more flexible than other approaches as the application can choose the appropriate time to perform reconciliation. Thus, applications may progress even over a dynamic network, in which nodes can connect and disconnect at any time. The main drawback is that replicas may be inconsistent or unfresh in the presence of queries, which in some cases may be unacceptable. *Non-optimistic lazy* replication is not as flexible as the optimistic approach, but completely avoids inconsistent reads, even if unfresh reads are accepted.

1.2 Contributions

In this report, we present our contributions to *improve data freshness* and to *manage strong consistency* in single master and multi-master lazy replication configurations, respecting the autonomy of the database internals. This means that all the components necessary to support our protocols are implemented outside de DBMS. These contributions were motivated by distributed database system applications such as small scale Oneline Analysis Processing (OLAP) and small and large scale Oneline Transaction Processing (OLTP) in database cluster system.

The motivations of these applications are given in chapters 2 and 3, respectively.

Let us now motivate the needs for replication in peer to peer (P2P) systems since it's a recent research challenge. Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, P2P, and mobile computing). As an example of such applications, consider a second generation Wiki [Wik07] that works over a peer-to-peer (P2P) network and supports users on the elaboration and maintenance of shared documents in a collaborative and lazy manner.

P2P systems adopt a completely decentralized approach to resource management [SMKK+01,RFHK+01,RD01,Gnu06,Kaz06]. By distributing data storage, processing, and bandwidth across autonomous peers in the network, they can scale without the need for powerful servers. All P2P systems rely on a P2P network to operate. This network is built on top of the physical network (typically the Internet), and therefore is referred to as an *overlay network*. The degree of centralization and the topology of the overlay network tightly affect the properties of the P2P system, such as fault-tolerance, self-maintainability, performance, scalability, and security. For simplicity, we consider three main classes: unstructured, structured, and super-peer networks.

P2P systems allow decentralized data sharing by distributing data storage across all peers of a P2P network. Since these peers can join and leave the system at any time, the shared data may become unavailable. To cope with this problem, a solution is to replicate data over the P2P network. Several data replication solutions have been proposed in P2P systems to improve availability within the overlay network [CMHS+02,ACDD+03, KBCC+00,AHA03]. Most of these solutions are based on lazy single master replication built for file systems. On the other hand, very few replication solutions have been proposed to handle the application requirements.

Our research contributions for managing lazy multi-master data replication in P2P systems are related to specific collaborative applications. We adopt optimistic replication due to its flexibility which is necessary in dynamic and large scale environments. The first contribution in this subject is a P2P Topology Aware Semantic Reconciliation engine. Another important contribution is related to the improvement of performance on providing data availability in structured P2P systems in the presence of multiple replicas of a given object, given dynamicity and failures. In this context, we proposed a data replication service that uses the concept of currency over a distributed timestamp protocol.

To conclude this section, we present a summary of the different contexts of our research on replication:

Distributed OLTP and OLAP (1999 to 2001) [PMS99, PS00, PMS01]:

- Mainly Lazy Single Master configurations
- Small scale
- Partial Replication
- Database heterogeneity
- Strong Consistency

Database Clusters (2002 to 2006) [POC03, CGPV04, PCVO05,CPV05a, CPV05b]:

- Lazy Multi-Master and Single Master configurations
- Large scale
- Full and Partial Replication
- Database heterogeneity
- Strong Consistency

P2P Data Management Systems (2003 until today) [AMPV04, MAPV06, MP06, EMP07, APV07a]:

- Lazy Single Master and Multi-Master Replication
- Dynamicity, high heterogeneity (DB, network, peers)
- Large scale or very large scale
- Full and Partial Replication
- Eventual consistency

Besides data replication, we also have important contributions on query processing in distributed P2P systems [AMPV06b, APV07b, APV07] but they are beyond the scope of this report.

1.3 Research Projects

Our research on data replication is motivated by several important research projects presented below:

- ESPRIT Long Term Research (DWQ) Data Warehouse Quality. I worked on DWQ as a Ph.D. student in the Rodin team at Inria Rocquencourt and afterwards, as a research collaborator in the Caravel team at Inria Rocquencourt. Our contribution was the development of algorithms to improve data freshness and managing consistency in lazy single master replication configurations for small scale heterogeneous distributed and replicated data base systems.
- RNTL Leg@net (Legacy applications on the Net). I worked on Leg@net first at University of Paris 5 and continued as assistant professor at University of Nantes, in the ATLAS team, a joint team between INRIA and LINA. I supervised the Ph.D. thesis of Cedric Coulon and 3 master theses. Our contribution was the development of lazy muti-master non-optimistic

replication algorithms and architectures for large scale database clusters for OLTP applications. We developed a significant research prototype which was released as open software (RepDB*).

- STREP Grid4All and RNTL XWiki Concerto, as an associated professor at University of Nantes and member the ATLAS team. I supervised the Ph.D. these of Vidal Martins and Reza Arkbarinia and 4 master these. I'am still working on these projects and supervising 2 engineers for prototyping our solutions. A significant research prototype, Atlas P2P Architecture (APPA) is being developed and incorporates the results of our work. Our contributions are P2P data replication solutions and architectures for lazy multi-master replication for collaborative applications over P2P systems.

1.4 Research Prototypes

The main results of my research have been the basis for the major research prototypes:

RepDB*: 2002-2005

http://www.sciences.univ-nantes.fr/lina/atlas/repdb/

We have initially designed it in the context of the Leg@net RNTL project and further developed it in the context of the ARA Masses de données MDP2P project. RepDB* supports preventive data replication capabilities (multi-master modes, partial replication, strong consistency) which are independent of the underlying DBMS. It employs general, non intrusive techniques. It is implemented in Java on Linux and supports various DBMS: Oracle, PostgreSQL and BerkeleyDB. We validated RepDB* on the Atlas 8-node cluster at LINA and another 64-node cluster at INRIA-Rennes. In 2004, we registered RepDB* to the APP (Agence pour la Protection des Programmes) and released it as Open Source Software under the GPL licence. Since then, RepDB* has been available for downloading (with more than a thousand downloads in the first three months).

Atlas Peer-to-Peer Architecture (APPA): 2006-now

http://www.sciences.univ-nantes.fr/lina/atlas/appa/

APPA is a P2P data management system that provides scalability, availability and performance for applications which deal with semantically rich data (XML, relational, etc.).

APPA provides advanced services such as queries, replication and load balancing. It is being implemented on top of various P2P networks such as JXTA and OpenChord and tested on GRID5000. Three services related to data replication have been implemented so far: KTS, P2P-Reconciler, and P2P-LTR. KTS (Key-based Timestamp Service) is a distributed service to manage timestamps in DHTs (distributed hash tables). It is useful to solve various DHT problems which need a total order on operations performed on each data, e.g. data currency. P2P-Reconciler is a P2P semantic optimistic reconciliation engine useful for managing multi-master replication for P2P collaborative applications. We are currently developing another service P2P Log and Timestamper for Reconciliation (P2P-LTR) in the Strep Grid4All and RNTL Xwiki Concerto projects as the basis to perform reconciliation of replicated documents in a P2P wiki system.

1.5 Report Organisation

This document is organized as follows. Chapter 2, summarizes and discusses the main contributions related to lazy master replication in the context of OLAP and OLTP applications. Next, in Chapter 3, we present our contributions on lazy multi-master replication for OLTP applications in database clusters. Chapter 4 presents our contributions on Topology Aware Semantic P2P Reconciliation and Chapter 5 presents our contributions on data availability in structured P2P systems. Finally, Chapter 6 concludes and discusses our current and future work. In addition to the 6 chapters that resumes our research work, we also present four annexes. Each annex corresponds to a published paper related to a specific Chapter: Annex A and B (Chapter 2), Annex C (Chapter 3), Annex D (chapter 4) and Annex E (Chapter 5) , and details the contributions of these chapters.

2 Lazy Single Master Replication in Data Warehouses

In this chapter, we first present the main applications that motivated our research on lazy single master replication. Next, we summarize: (i) the proposed framework for lazy replication, (ii) two update propagation strategies to improve data freshness based on immediate propagation, (iii) consistency criteria's in terms of group communication protocols for some important configurations used in OLAP and OLTP (iv) the refresher algorithms used to enforce consistency for acyclic configurations. Next, we compare the contributions with relevant related work. Finally we conclude the section.

2.1 Motivations

Over years companies have built operational database to support their day-to-day operations with On-Line Transaction Processing (OLTP) applications which are transactions oriented (airline reservation, banking, etc). They need extensive data control and availability, high multi-user throughput and predicable, fast response times. The users are clerical. Operational databases are medium to large (up to several gigabytes). In effect, distributed databases have been used to provide integrated access to multiple operational databases and data replication is used to improve response time and availability. To improve response times and scalability up to ten nodes in the internet, lazy single master replication is proposed as an alternative solution to the early eager replication solution. The concept of freshness is then necessary to measure the deviation of primary and secondary copies. In addition, in partial lazy single master replication, consistency management is necessary to express in which order transactions that update replicated data must be executed to avoid inconsistent reads.

Decision support applications have been termed On-Line Analytical Processing (OLAP) [Cod95] to better reflect their different requirements. OLAP applications, such as trend analysis or forecasting, need to analyze historical, summarized data coming from operational databases. They use complex queries over potentially very large tables and read intensive. Because of their strategic nature, response time is important. Performing OLAP queries directly over distributed operational databases raises two problems. First, is hurts the OLTP application performance by competing for local sources. Second, the overall response time of the OLAP queries can be very poor because large quantities of data need to be transferred over the network. Furthermore, most OLAP applications do not need the most current versions of the data and thus do not need direct access to operational data. Data warehousing is the solution to this problem, which extracts and summarizes data from operational data bases in a separate database, dedicated to OLAP. Data warehousing is often considered an alternative to distributed databases, but in fact these are complementary technologies. Lazy single master data replication is used to built distributed data warehouse and needs high degree of fresh data to improve OLAP analysis quality. In addition, several distributed data warehousing architectures have been proposed and again, consistency management is necessary to express in which order transactions that update replicated data must be executed to avoid inconsistent reads.

2.2 Lazy Single Master Context

With lazy single master replication, updates are performed on a primary copy are first committed at the master node. Afterwards, each secondary copy is updated, in a separate transactions, *called refresh transaction*. Primary copied are updatable and secondary copies are read-only. In addition, there is a single primary copy for each secondary copy. Lazy single master replication have been widely implemented by current database systems [GHOS96, Lad90]. When we started on this topic, several manuals, documents and few papers provided informal definitions which were general and not precise enough to address the problems related to the applications that motivated our research. To express our contributions, we proposed a formal framework for lazy single master replication based in five basic parameters: *ownership, configuration, transaction model, propagation and refreshment*. Even though we focused on lazy single master replication, our framework is also valid for lazy multi-master replication. We will use the terms defined in this framework also to compare our work with related ones, later on in this chapter. We pay special attention to the propagation parameter since it expresses important contributions in the field. Finally, we present the way we manage consistency in different lazy single master configurations.

2.3 Basic Parameters

The *ownership* parameter is inspired from [GHOS96] and defines the nodes capabilities for updating replica copies (primary and secondary copies). Three types of nodes are identified: Master, Slave and MasterSlave. A node is called Master if it stores only primary copies (upper case letter). Similarly if a nodes store only secondary copies (lower case letter), it is called slave node. Finally if a node stores primary and secondary copies, it is called MasterSlave.

The *configuration* parameter defined the components nodes of a replication configuration used in distributed OLTP and data warehousing architectures. We focus on three main configurations: 1Master-nSlaves (Figure 2.1.a), 1Slave-NMaster (Figure 2.1.b), mMaster-nSlave (Figure 2.1.c) and Master-MasterSlave-Slave (Figure 2.1.d).

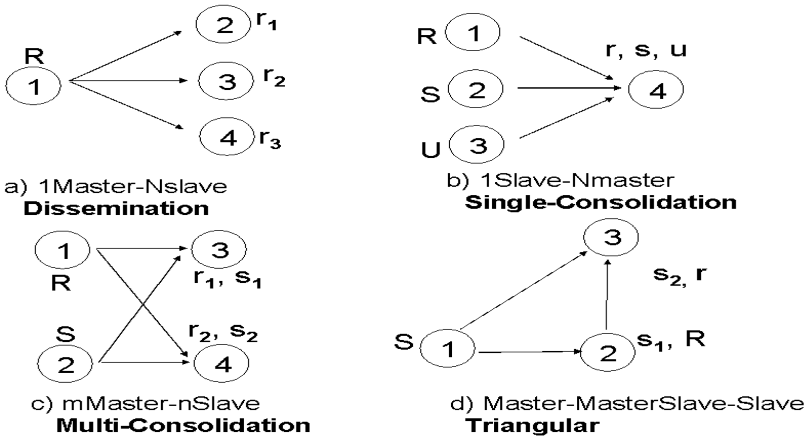


Figure 2.1: Lazy Single Master Configuration

The transaction model parameter defines the properties of the transactions that access the replica copies at each node. We focus on three types of transactions: *update transaction* (noted *T*) that update primary copies, *refresh transactions* (notes *RT*) which carries the sequence of write operations of a specific update transaction used to refresh secondary copies and *queries* that read secondary copies at the slave nodes.

2.4 Propagation and Refreshment

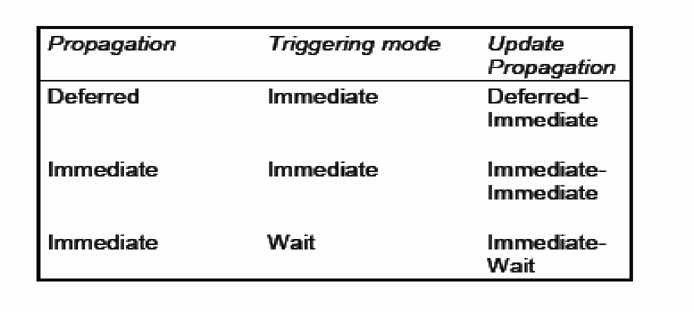
The propagation parameter defines "when" the updates to a primary copy must be multicast towards the nodes storing its secondary copies.

We focus on two types of propagation: *deferred* and *immediate* [PSM98]. D*eferred* propagation is found in commercial database systems and *immediate* propagation is our proposal. When using deferred propagation strategy, the serial sequence of writes on a set of primary copies performed by an update transaction *T* is multicast together within a message *M,* after the commitment of *T.* When using an immediate propagation, each write operation performed by a transaction, is immediately multicast inside a message *m*, without waiting for the commitment of the original update transaction *T.*

The refreshment parameter is defined by the triggering and ordering of a refresh transaction ate the slave node. The triggering component defines when the delivery (transaction submission) of a refresh transactions starts with respect to the propagation strategy. Three triggering modes are proposed: *deferred*, *immediate* and *wait*. The couple formed by the propagation and trigger mode determines a specific update propagation strategy. We proposed two update propagation strategies: *immediate-immediate* and *immediate-wait*.

With *deferred-immediate*, as soon as a transaction is committed at the master node the corresponding refresh transaction is multicast towards the slaves nodes and, at the slave node, as soon as *RT* is received it is submitted to execution. This strategy corresponds to the ones available in some commercial systems.

We proposed the following strategies to improve data freshness since mutual consistency is relaxed. *immediate-immediate* propagation, involves the sending of each write operation performed by *T* towards each slave node, without waiting for the commitment of the original update transaction. At the slave node, a refresh transaction is started as soon as the first write operation is received from the master or MasterSlave node. Finally, *immediate-wait* is similar to immediate-immediate. However, a refresh transaction is submitted for execution only after the complete reception of all write operations of the original update transaction. These *immediate* strategies allow parallelism between the propagation of updates and the execution of the associated refresh transactions. Figure 2.2 shows these update propagation strategies.



Propagation	*Triggering mode*	*Update Propagation*
Deferred	Immediate	Deferred-Immediate
Immediate	Immediate	Immediate-Immediate
Immediate	Wait	Immediate-Wait

Figure 2.2: Update Propagation Strategies

2.5 Validation of the Update Propagation Strategies

To implement these strategies, we proposed a lazy single master (1Master-nSlave) replication architecture which respects the underlying DBMS heterogeneity and autonomy as a *black box*. Using this architecture, we simulated these propagation strategies over Oracle 7.3 to measure the gains on data freshness. More formally, given a replica X, which is either a secondary or primary copy, we define *n(X,t)* as the number of committed update transactions on X at global time *t*. It was assumed that update transactions can have different sizes but their occurrence is uniformly distributed over time. Then, the degree of freshness of a secondary copy *r* at global time *t* is: *f(r,t) = n(r,t)/n(R,t)*, where *R* is the primary copy of *r*. A degree of freshness close to 0 means bad freshness while close to 1 means excellent.

Based on this measure, we studied exhaustively the behaviour of the three update propagation strategies and the results were very good (for complete performance results see Annex A). The results indicate that, for short transactions, the deferred approach performs almost as well as *immediate-immediate* and *immediate-wait*. The strategies exhibit different freshness results when long transactions occur. In these cases, the *immediate* strategies show much better results and the *immediate-immediate* strategy provides the best freshness results. For some important kinds of workloads, freshness may be five times better than that of the deferred strategy. On the other hand, *immediate-wait* only improves freshness when the update transaction arrival rate at the master is *bursty* (i.e. there are short periods (bursts) with high numbers of arriving transactions). The down side of *immediate-immediate* is the increase of query response times due to transactions blocking when there are conflicts between refresh transactions and queries. However, using a multi-version concurrency control protocol at the slave node, this drawback can be drastically reduced without a significant loss of freshness. The improvement shown by the immediate strategies are beneficial for both OLTP and OLAP applications.

2.6 Consistency Management

The consistency parameter enables to express how to achieve refresh transaction total order at the slave nodes with respect to the corresponding update transactions at the master nodes, considering different types of configurations. We express consistency in terms of acceptable group communication primitives. This is an important contribution because it was one of the first approaches to use group communication services [CKV01, Kemm00]] together with distributed transaction management for lazy replication. Briefly, group communication systems provide *multicast services* which may differ in the final order in which message are delivered at each node *N* [Kemm00]. Below we present the main services at were useful to define our consistency criteria's:

1. *Basic service*: A message is delivered whenever it is physically received from the network. Thus, each node receives the message in arbitrary order.

2. *Reliable service*: If a message is delivered in a node then all nodes receive the message unless they fail.
3. *FIFO service*: If a node sends message *m* before message *m'*, then no node receives *m'* unless it has previously received *m*.
4. *Causal order service*: If a message *m* causally precedes a message *m'* then no node receives *m'* until it has previously received *m*.
5. *Total order service*: All messages are delivered in the same total order at all sites, i.e. if any two nodes *N* and *N'* receive some messages *m* and *m'*, then either both receive m before *m'* or both receive *m'* before *m*.

Using these services, we express consistency as follow:

- In 1Master-nSlave and nMaster-1Slave configurations, the message delivery order at each slave node must follow the same delivery order that would be obtained when using reliable FIFO multicast service.
- In mMaster-nSlave configuration, the message delivery order at each slave node must follow the same delivery order that would be obtained when using a reliable FIFO Total order multicast service.
- In Master-MasterSlave-Slave configuration, the message delivery order at each slave node must follow the same delivery order that would be obtained when using reliable Causal order multicast service.

2.7 Refresher Algorithms

We proposed three Refresher algorithms that assure consistency (for all above configurations) [PMS99, PMS01] (see Annex B), each one related to a specific update propagation strategy. The three algorithms implement refresh transaction ordering in the following way.

It was assumed that the underlying network implements reliable FIFO multicast and has a known upper bound *Max*. In addition, it was also assumed that clocks are synchronized, such that the difference between any two clocks is not higher than the precision ε. All these assumption are acceptable and in the next chapter we relax some of them in the context of database clusters. The timestamp of a message corresponds to the real time value at *T*'s commitment time.

To assure consistency for mMaster-nSlave and Master-MasterSlave-Slave configurations, the refresher algorithm works as follows. A refresh transaction RT is committed at a Slave or MasterSlave node (1) once all its write operations have been done, (2) according to the order given by the timestamp *C* of its associated update transaction, and (3) at the earliest, at real time *C* + *Max* + ε. After this delay period, all older refresh transactions are guaranteed to be received at node. Thus total order is assured among update transactions and refresh transaction at all involved slave nodes meeting the consistency requirements.

A key aspect of the refresher algorithm is to rely on the upper bound *Max* on the transmission of a message by the global FIFO reliable multicast. Therefore, it is essential to have a value *Max* that is not overestimated. The computation of *Max* resorts to scheduling theory. Recent work in P2P systems accepts that bounds may be known in *relaxed-synchronous systems* [BGM04] which is valid for distributed database systems. Thus this assumption is acceptable. The value of *Max* usually takes into account four kind of parameters. First, there is a global reliable multicast itself. Second, are the characteristics of the message to multicast (e.g. arrival laws, size). For instance in [GM98], an estimation of *Max* is given for sporadic message arrivals. Third, are the failures to be tolerated by the multicast algorithm (e.g medium access protocol). It is possible to compute an upper bound Max_i for each type *i* of message to multicast. In that case, the refreshment algorithm at node N waits $max_{i∈J}$ Max_i , where *J* is the set of message types that can be received by node *N*.

Thus, an accurate estimation of *Max* depends on the accurate knowledge of the above parameters. However, accurate values of the application dependent parameters can be obtained in performance sensitive replicated database applications. For instance, in the case of data warehouse applications that have strong requirements on freshness, certain characteristics of messages can be derived from the characteristics of the operational data sources (usually, transaction processing systems). Furthermore, in a given application, the variations in the transactional workload of the data sources can often be predicted.

There are many alternatives to implement a total order primitive: based on tokens [MMS+96], sequencing [KT96], consensus [CT96], etc. All these solutions require several message rounds [Kem00]. In contrast, for basic FIFO order multicast, a message is simply broadcast to all nodes [CT96,SR96]. Each node tags its message with sequence numbers. Clearly, reliable FIFO multicast is cheaper than the total order service which requires further communication in order to determine the total order.

In summary, the approach taken by the refresher algorithm to enforce total order over an algorithm that implements a global FIFO reliable multicast trades the use of a worst case multicast time at the benefit of reducing the number of messages exchanged on the network, for instance when using atomic broadcast. This is a well known tradeoff. This solution brings simplicity and ease of implementation.

2.8 Related Work

There are several propagation strategies for lazy single master replication configurations based on *push* and *pull* approaches [Dav94, PA99]. Whenever the master node is the one that initiates update propagation then the update propagation strategy follows the push approach. The push approach makes it possible to perform event-driven propagation (implemented by Sybase, Informix and Ingres). In contrast, when the slave requests update propagation, then it follows the pull approach. The drawback of the pull approach is the difficulty to achieve near-real-time update. The main advantage of pull approaches is that it provides scalability, reduces network load by propagating only the last value or aggregated data instead of the whole sequence of updates performed at the master node. Capturing updates on primary copied is done using different capturing mechanisms such as *Log Sniffing* (ex: Sybase Replication Server) , *using triggers* (ex: Oracle) to start update propagation, or propagating *SQL statements*.

Concerning freshness, it has been suggested [ABG90, SR90, AA95, PV98, GNPV07, BFGR06] that users should be allowed to specify freshness constraints. The types of freshness constraints that can be specified are the following:

- Time-bound constraints. Users may accept divergence of physical copy values up to a certain time: x_i may reflect the value of an update at time t while x_j may reflect the value at $t - \Delta$ and this may be acceptable.
- Value bound constraints. It may be acceptable to have values of all physical data items within a certain range of each other. The user may consider the database to be mutually consistent if the values not converge more than a certain amount (or percentage).
- Drift constraints on multiple data items. For transactions that read multiple data items, the users may be satisfied if the time drift between the update timestamps of two data items is less then a threshold.

In [CRR96] the authors proposes a solution that defines a set of allowed configurations using configuration graphs where nodes are sites, and there is non-directed edge between two sites if one has the primary copy and the other a secondary copy for a give data item. In order to provide serializability the resulting configuration must be strongly acyclic. Clearly, this approach does not provide a solution for some important cyclic configurations such as the

Master-MasterSlave-Slave considered in our work. In [BKRS+99], the authors proposed an alternative solution by requiring the directed configuration graph (edges are directed from primary copy to secondary copy) to have no cycles. One strategy transforms the graph into a tree where a primary copy is not necessarily directly connected with all its secondary copies but there exists a path from the primary to each secondary. Update propagation is then performed along the paths of the graph (edges are directed from primary to secondary copy to have no cycles). This also requires to introduce more sophisticated update propagation strategies. Again, no algorithm is provided to refresh secondary copies in the cases of cyclic configurations, such as the Master-MasterSlave-Slave considered in our work.

2.9 Conclusion

In this chapter, we presented our contributions on lazy single master replication [PSM98, PV98,PMS99, PS00, PMS01]. We first addressed the problem related to freshness improvement in lazy single master replicated schemes. More specifically, we dealt with update propagation from primary copy to secondary copies. A complete framework and functional architecture is proposed [PMS01] to define update propagation strategies. We proposed two new strategies called *immediate-immediate* and *immediate-wait*, which improve over the deferred strategy of commercial. These strategies allow parallelism between the propagation of updates and the execution of the associated refresh transactions. *Immediate-Immediate* is the best strategy (by a factor of 5) but may slow down queries. *Immediate-wait* is almost as good but never slows down queries.

Next, we discussed the contributions related to refreshment algorithms which addresses the central problem of maintaining replicas' consistency. It delays the execution of a refresh transaction until its deliver time. An observer of a set of replicas at some node never observes a state which is never seen by another observer of the same set of replicas at another node. The basic refresher algorithm may be implemented using the *deferred-immediate* propagation strategy as well as the immediate strategies to improve *fresheness* in data warehousing applications. Another important contribution is related to the notion of correct refreshment algorithm acceptable for some important configurations in terms of group communication protocols, which is a very important contribution. Annex A and B presents [PS00] and [PMS01] that provides all formalizations and performance evaluation results resumed in this chapter.

3 Lazy Multi-Master Replication in Database Clusters

In this chapter we present preventive replication algorithms used to manage consistency in lazy multi-master replication configurations. This is done in the context of database clusters where scalability is the main concern. We first present the type of applications that motivated this research. Next, as the main contributions of the chapter, we present the type of lazy multi-master configurations we consider and the corresponding preventive replication algorithms necessary to assure consistency with some important optimizations. An important contribution of this work is RepDB* which is an open source software available for downloading. Finally, we compare our approach with important related work and conclude the chapter.

3.1 Introduction

High-performance and high-availability of database management have been traditionally achieved with parallel database systems [Val93], implemented on tightly-coupled multiprocessors. Parallel data processing is then obtained by partitioning and replicating the data across the multiprocessor nodes in order to divide processing. Although quite effective, this solution requires the database system to have full control over the data and is expensive in terms of software and hardware.

Clusters of PC servers now provide a cost-effective alternative to tightly-coupled multiprocessors. They have been used successfully by, for example, Web search engines using high-volume server farms (e.g., Google). However, search engines are typically read-intensive, which makes it easier to exploit parallelism. Cluster systems can make new businesses such as Application Service Providers (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at the provider site and need to be available, typically through the Internet, as efficiently as if they were local to the customer site. Notice that due to autonomy, it is possible that the DBMS at each node are heterogeneous. To improve performance, applications and data can be replicated at different nodes so that users can be served by any of the nodes depending on the current load [ABKW98]. This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. However, managing data replication in the ASP context is far more difficult than in Web search engines since applications can be update-intensive and both applications and databases must remain autonomous. The solution of using a parallel DBMS is not appropriate as it is expensive, requires heavy migration to the parallel DBMS and hurts database autonomy.

We consider a database cluster with similar nodes, each having one or more processors, main memory (RAM) and disk. Similar to multiprocessors, various cluster system architectures are possible: shared-disk, shared-cache and shared-nothing [Val93]. Shared-disk and shared-cache require a special interconnect that provides a shared space to all nodes with provision for cache coherence using either hardware or software. Shared-nothing (or distributed memory) is the only architecture that supports our autonomy requirements without the additional cost of a special interconnect. Furthermore, shared-nothing can scale up to very large configurations. Thus, we strive to exploit a shared-nothing architecture.

To improve performance in a database cluster, an effective solution is to replicate databases at different nodes so that users can be served by any of them depending on the current load [GNPV92, GNPV07] . This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. However, the major problem of data replication is to manage the consistency of the replicas in the presence of updates [GHOS96]. The basic solution in distributed systems that enforces strong replica consistency[1] is synchronous (or eager) replication (typically using the Read-One-Write All – ROWA protocol [OV99]). Whenever a transaction updates a replica, all other replicas are updated inside the same distributed transaction. Therefore, the mutual consistency of the replicas is enforced. However, synchronous replication is not appropriate for a database cluster for two main reasons. First, all the nodes would have to homogeneously implement the ROWA protocol inside their local transaction manager, thus violating DBMS autonomy. Second, the atomic commitment of the distributed transaction typically relies on the two-phase commit (2PC) protocol which is known to be blocking (i.e., does not deal well with nodes' failures) and has poor scale up.

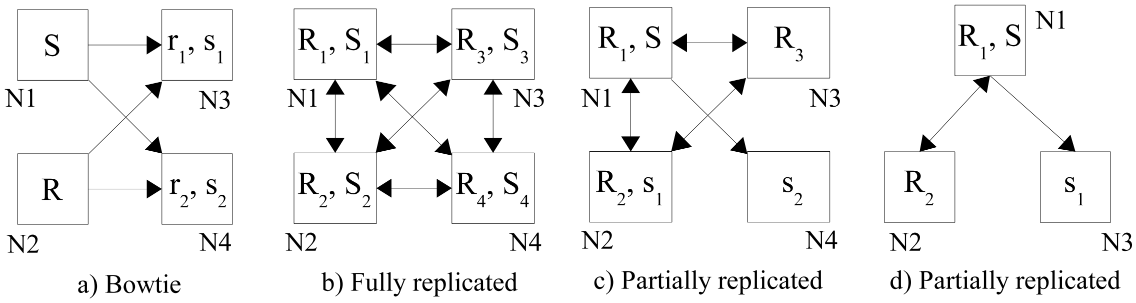
A better solution that scales up is lazy replication. Lazy replication allows for different replication configurations. A useful configuration is *lazy master* where there is only one primary copy. Although it relaxes the property of mutual consistency, strong consistency is assured. However, it hurts availability since the failure of the master node prevents the replica to be updated. A more general configuration is (*lazy*) *multi-master* where the same primary copy, called a multi-owner copy, may be stored at and updated by different master nodes, called multi-owner nodes. The advantage of multi-master is high-availability and high-performance since replicas can be updated in parallel at different nodes. However, conflicting updates of the same primary copy at different nodes can introduce replica incoherence. In the rest of this chapter, we present our solution to this problem using *preventive replication* for *full* and *partial* [PCVO95, CPV05a, CPVb05] replication configurations.

3.2 Lazy Multi Master Configurations

A *primary copy*, denoted by *R*, is stored at a *master* node where it can be updated while a *secondary copy*, denoted by r_i, is stored at one or more *slave* nodes *i* in read-only mode. A *multi-master copy*, denoted by R_i, is a primary copy that may be stored at several multi-master nodes *i*. Figure 3.1 shows various replication configurations, using two tables *R* and *S*.

Figure 3.1a shows a bowtie (lazy master) configuration where there are only primary copies and secondary copies. This configuration is useful to speed-up the response times of read-only queries through the slave nodes, which do not manage the update transaction load. However, availability is limited since, in the case of a master node failure, its primary copies can no longer be updated.

[1] For any two nodes, the same sequence of transactions is executed in the same order.



a) Bowtie b) Fully replicated c) Partially replicated d) Partially replicated

Figure 3.1 Replication configurations

Figure 3.1.b shows a fully replicated configuration. In this configuration, all nodes manage the update transaction load because whenever *R* or *S* is updated at one node, all other copies need be updated asynchronously at the other nodes. Thus, only the read-only query loads are different at each node. Since all the nodes perform all the transactions, load balancing is easy because all the nodes have the same load (when the specification of the nodes is homogeneous) and availability is high because any node can replace any other node in case of failure.

Figure 3.1.c and Figure 3.1.d illustrate partially replicated configurations where all kinds of copies may be stored at any node. For instance, in Figure 3.1.c, node N_1 carries the multi-master copy R_1 and the primary copy *S*, node N_2 carries the multi-master copy R_2 and the secondary copy s_1, node N_3 carries the multi-master copy R_3, and node N_4 carries the secondary copy s_2. Compared with full replication, only some of the nodes are affected by the updates on a multi-master copy (only those that hold common multi-master copies or one of their corresponding secondary copies in a separate step). Therefore, transactions do not have to be multicast to all the nodes. Thus, the nodes and the network are less loaded and the overhead for refreshing replicas is significantly reduced.

With partial replication a transaction *T* may be composed of a sequence of read and write operations followed by a commit (as produced by the SQL statement in Figure 3.2) that updates multi-master copies. This is more general than in [POC03] where only write operations are considered. We define a *refresh transaction (RT)* as the sequence of write operations of a transaction *T* that updates a set of multi-master copies (R_1,R_2..R_n) at node N_i . *RT* is extracted from the Log History of N_i and propagated only to the nodes that holds secondary copies of R_1,R_2..R_n. A *refreshment algorithm* is the algorithm that manages, asynchronously, the updates on a set of multi-master and secondary copies, once one of the multi-master (or primary) copies is updated by *T* for a given configuration.

Given a transaction *T* received in the database cluster, there is an *origin node* chosen by the load balancer that triggers refreshment, and a set of *target nodes* that carries replicas involved with *T*. For simplicity, the origin node is also considered a target node. For instance, in Figure 3.1.b whenever node N_1 receives a transaction that updates R_1, then N_1 is the origin node and N_1, N_2, N_3 and N_4 are the target nodes.

To refresh multi-master copies in the case of full replication, it is sufficient to multicast the incoming transactions to all target nodes. But in the case of partial replication, even if a transaction is multicast towards all nodes, it may happen that the nodes are not able to execute it because they do not hold all the replicas necessary to execute *T* locally. For instance, Figure 3.1.c allows an incoming transaction at node N_1, such as the one in Figure 3.2 to read *S* in order to update R_1. This transaction can be entirely executed at N_1 (to update R_1) and N_2 (to update R_2). However it cannot be executed at node N_3 (to update R_3) because N_3 does not hold a copy of *S*. Thus, refreshing multi-master copies in the case of partial replication needs to take into account replica placement.

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

Figure 3.2 Incoming transaction at node N1

3.3 Consistency Management

Informally, a correct refreshment algorithm guarantees that any two nodes holding a common set of replicas, R_1, R_2, …, R_n, must always produce the same sequence of updates on R_1, R_2, …, R_n. We provide a criterion that must be satisfied by the refreshment algorithm in order to be correct:

Multi-Master Full Replication: For any cluster configuration that meets a multi-master configuration requirement, the refresh algorithm is correct if and only if the algorithm enforces total order for update transaction execution at each node.

Multi-Master- Partial Replication: For any cluster configuration meets partially replicated configuration requirement, then the refresh algorithm is correct if and only if the algorithm enforces total order for update and refresh transactions execution at each involved node.

3.4 Preventive Refresher for Full Replication

The preventive refresher algorithm comes from the one presented for lazy single master configurations. In this section, wee briefly recall how its works here for multi-master configurations. For all details (including partial replication configurations) see Annex C that corresponds to [PCVO05].

We assume that the network interface provides reliable FIFO multicast [HT93]. We denote by *Max*, the upper bound of the time needed to multicast a message from a node *i* to any other node *j*. We also assume that each node has a local clock. For fairness, clocks are assumed to have a drift and to be ε-synchronized. This means that the difference between any two correct clocks is not higher that ε (known as the precision).

Each transaction is associated with a chronological timestamp value *C*. The principle of the preventive refreshment algorithm is to submit a sequence of transactions in the same chronological order at each node. Before submitting a transaction at node *i*, we must check whether there is any older transaction en route to node *i*. To accomplish this, the submission time of a new transaction at node *i* is delayed by *Max* + ε. Thus the earliest time a transaction is submitted is *C* + *Max* + ε (henceforth *delivery time*).

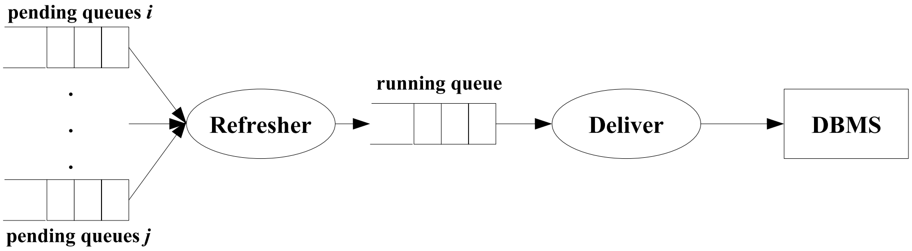


Figure 3.3 Refreshment Architecture

Whenever a transaction T_i is to be triggered at some node *i*, node *i* multicasts T_i to all nodes 1, 2, …, *n*, including itself. This is the main difference compared to the refresher algorithm used to enforce consistency for NMaster-NSlave (bowtie) configurations. Once T_i is received at some other node *j* (*i* may be equal to *j*), it is placed in the pending queue in FIFO order with respect to the triggering node *i*. Therefore, at each multi-master node *i*, there is a set of queues, q_1, q_2, …, q_n, called pending queues, each of which corresponds to a multi-master node and is used by the refreshment algorithm to perform chronological ordering with respect to the delivery times. Figure 3.3 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the top of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction is ordered, the refresher writes it to the *running que*ue in FIFO order, one after the other. Finally, *Deliver* keeps checking the head of the running queue to start transaction execution, one after the other, in the local *DBMS*.

3.5 Preventive Replication for Partial Configurations

With partial replication, some of the target nodes may not be able to perform a transaction *T* because they do not hold all the copies necessary to perform the read set of *T* (recall the discussion on Figure 3.2). However the write sequence of *T*, which corresponds to its refresh transaction, denoted by *RT*, must be ordered using *T*'s timestamp value in order to ensure consistency. So *T* is scheduled as usual but not submitted for execution. Instead, the involved target nodes wait for the reception of the corresponding *RT*. Then, at origin node *i*, when the commitment of *T* is detected (by sniffing the

DBMS' log), the corresponding *RT* is produced and node *i* multicasts *RT* towards the target nodes. Upon reception of *RT* at a target node *j*, the content of *T* (still waiting) is replaced with the content of incoming *RT* and *T* can be executed.

Let us now illustrate the algorithm with an example of execution. In Figure 3.4, we assume a simple configuration with 4 nodes (N_1, N_2, N_3 and N_4) and 2 copies (*R* and *S*). N_1 carries a multi-owner copy of *R* and a primary copy of *S*, N_2 a multi-owner copy of *R*, N_3 a secondary copy of *S*, and N_4 carries a multi-owner copy of *R* and a secondary copy of *S*. The refreshment proceeds in 5 steps. In step 1, N_1 (the origin node) receives *T* from a client which reads *S* and updates R_1. For instance, *T* can be the resulting read and write sequence produced by the transaction of Figure 3.2. Then, in step 2, N_1 multicasts *T* to the involved target nodes, i.e. N_1, N_2 and N_4. N_3 is not concerned with *T* because it only holds a secondary copy *s*. In step 3, *T* can be performed using the refreshment algorithm at N_1 and N_4. At N_2, *T* is also managed by the Refresher and then put in the running queue. However, *T* cannot yet be executed at this target node because N_2 does not hold *S*. Thus, the Deliver needs to wait for its corresponding *RT* in order to apply the update on *R* (see step 4). In step 4, after the commitment of *T* at the origin node, the *RT* is produced and multicast to all involved target nodes. In step 5, N_2 receives *RT* and the Receiver replaces the content of *T* by the content of *RT*. The Deliver can then submit *RT*.

Partial replication may be blocking in case of failures. After the reception of *T*, some target nodes would be waiting for *RT*. Thus, if the origin node fails, the target nodes are blocked. However, this drawback can be easily solved by replacing the origin node by an equivalent node, a node that holds all the replicas necessary to execute *T*. Once the target node detects the failure of the origin node, it can request an equivalent node *j* to multicast *RT* given *T*'s identifier. At node *j*, *RT* was already produced in the same way as at the origin node: transaction *T* is executed and, upon detection of *T*'s commitment, an *RT* is produced and stored in a *RT* log, necessary to handle failure of the origin node. In the worst case where no other node holds all the replicas necessary to execute *T*, *T* is globally aborted.

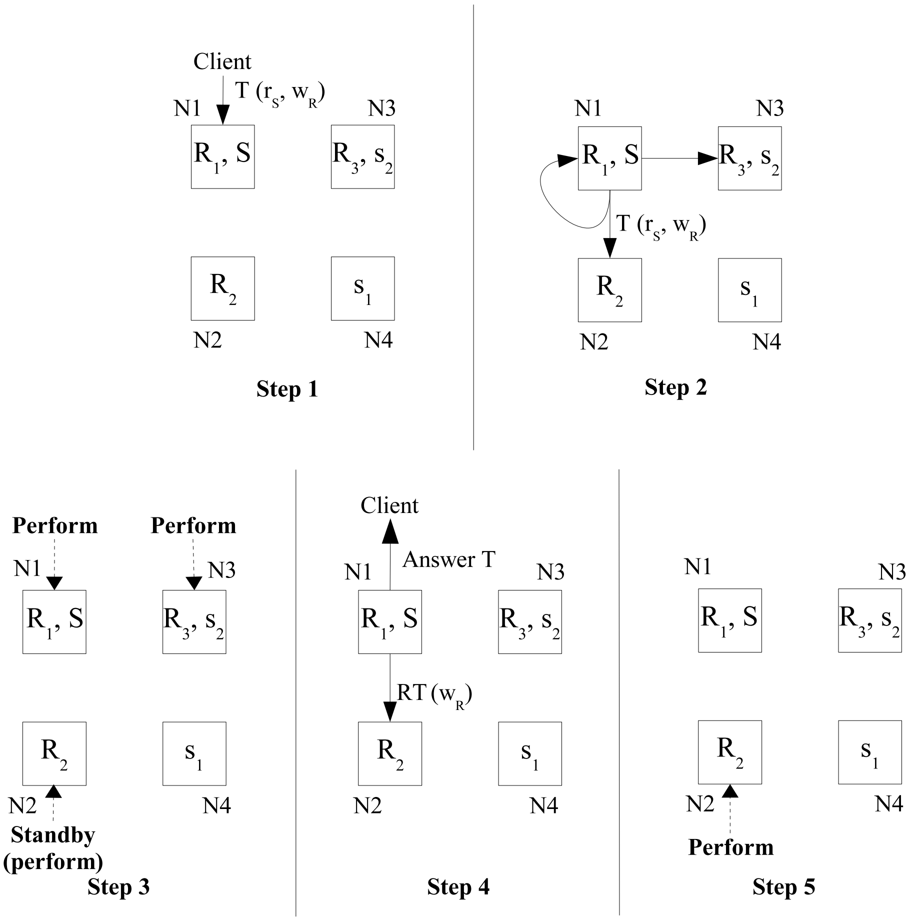


Figure 3.4 Example of preventive refreshment with partial configurations

3.6 Optimizations

In a cluster network (which is typically fast and reliable), in most cases messages are naturally chronologically ordered [PS98]. Only a few messages can be received in an order that is different than the sending order. Based on this property, we can improve our algorithm by submitting a transaction to execution as soon as it is received, thus avoiding the delay before submitting transactions. Yet, we still need to guarantee strong consistency. In order to do so, we schedule the commit order of the transactions in such a way that a transaction can be committed only after *Max* + *ε*. Recall that to enforce strong consistency, all the transactions must be performed according to their timestamp order. So, a transaction is out-of-order when its timestamp is lower than the timestamps of the transactions already received. Thus, when a transaction *T* is received out-of-order, all younger transactions must be aborted and re-submitted according to their correct timestamp order with respect to *T*. Therefore, all transactions are committed in their timestamp order.

Thus, in most cases the delay time (*Max* + ε) is eliminated. Let *t* be the time to execute transaction *T*. In the previous basic preventive algorithm, the time spent to refresh a multi-master copy, after reception of *T*, is *Max* + ε + *t*. Now, a transaction *T* is ordered while it is executed. So, the time to refresh a multi-master copy is max[(Max + ε), t]. In most cases, t is higher than the delay Max + ε. Thus, this simple optimization can well improve throughput as we show in our performance study.

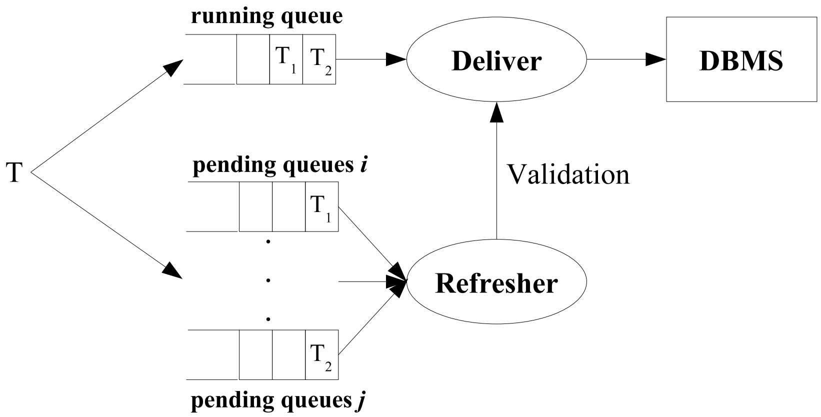
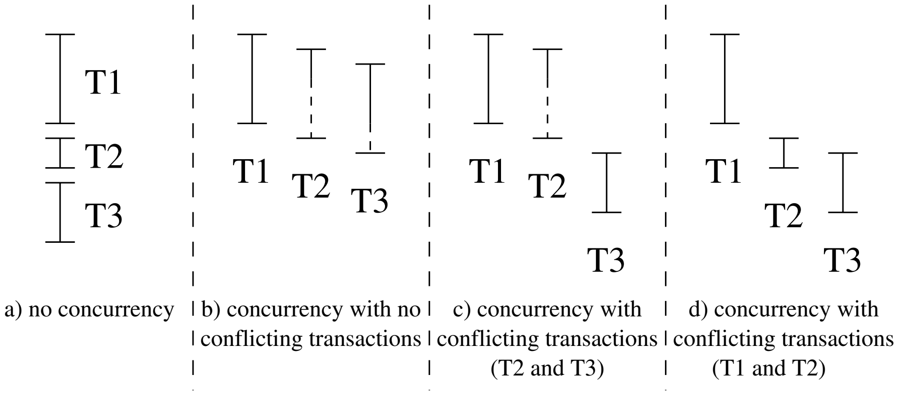


Figure 3.5 Refreshment Architecture

The average Figure 3.5 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the head of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction *T* is ordered, the Refresher notifies the Deliver that *T* is ordered and ready to be committed. Meanwhile, the Deliver keeps checking the head of the running queue to start transaction execution optimistically, one after the other, inside the local *DBMS*. However, to enforce strong consistency the Deliver only commits a transaction when the Refresher has signaled it.

To improve throughput, we introduce concurrent replica refreshment. In the previous section, the Receiver writes transactions directly into the running queue (optimistically), and afterwards the Deliver reads the running queue contents in order to execute the transaction. On the other hand, to assure consistency, the same transactions are written as usually in the pending queues to be ordered by the Refresher. Hence, the Deliver extracts the transactions from the running queue and performs them one by one in serial order. So, if the Receiver fills the running queue faster than the Deliver empties it, and if arrival rate is higher than the average running rate of a transaction (typically in bursty workloads), the response time increases exponentially and performance degrades.



a) no concurrency | b) concurrency with no conflicting transactions | c) concurrency with conflicting transactions (T2 and T3) | d) concurrency with conflicting transactions (T1 and T2)

Figure 3.6 Example of concurrent execution of transactions

To improve response time in bursty workloads, we propose to trigger transactions concurrently. In our solution, concurrency management is done outside the database to preserve autonomy (different from [POC03]). Using the existing isolation property of database systems, at each node, we can guarantee that each transaction sees a consistent database at all times. To maintain strong consistency at all nodes, we enforce that transactions are committed in the same order in which they are submitted. In addition, we guarantee that transactions are submitted in the order in which they have been written to the running queue. Thus, total order is always enforced.

However, without access to the DBMS concurrency controller (for autonomy reasons), we cannot guarantee that two conflicting concurrent transactions obtain a lock in the same order at two different nodes. Therefore, we do not trigger conflicting transactions concurrently. To detect that two transactions are conflicting, we determine a subset of the database items accessed by the transaction according to the transaction. If the subset of a transaction does not intersect with a subset of another transaction, then the transactions are not conflicting. For example, in the TPC-C benchmark, the transactions' parameters allow us to define a subset of tuples that could be read or updated by the transaction. If the subset of the transaction cannot be determined, then we consider the transaction to be conflicting with all other transactions. This solution is efficient if most transactions are known, which is true in OLTP environments.

We can now define two new conditions to be verified by the Deliver before triggering and before committing a transaction:

i. Start a transaction iff the transaction is not conflicting with transactions already started (but not committed) and iff no older transaction waits for the commitment of a conflicting transaction to start.

ii. Commit a transaction iff no older transactions are still running.

Figure 3.6 shows examples of concurrent executions of transactions. Figure 3.6.a illustrates a case where the transactions are triggered sequentially, which is equivalent to the case where all the transactions are conflicting. Figure 3.6.b, Figure 3.6.c and Figure 3.6.d show parallel executions of transaction T_1, T_2 and T_3. In Figure 3.6.b and Figure 3.6.c, transaction T_2 finishes before T_1 but waits for commit because T_1 is still running (this is represented by a dashed line in the figure). In Figure 3.6.b, T_1, T_2 and T_3 are not conflicting, so they can run concurrently. On the other hand, in Figure 3.6.c, T_2 is conflicting with T_3, so T_3 must wait for the end of T_2 before starting. Finally, in Figure 3.6.d, T_1 and T_2 are conflicting, so T_2 cannot start before the commitment of T_1 and T_3 cannot start before T_2 because transactions must be executed in the order they are in the running queue.

3.7 Validation

We implemented our Preventive Replication Manager in our RepDB* prototype [CGPV04, RepDB] on a cluster of 64 nodes (128 processors) on a Grid5000[Gri06]. RepDB* is a data management component for replicating autonomous databases or data sources in a cluster system. RepDB* supports preventive data replication capabilities (multi-master modes, partial replication, strong consistency) which are independent of the underlying DBMS. It employs general, non intrusive techniques. It is implemented in Java on Linux and supports various DBMS: Oracle, PostGreSQL and BerkeleyDB. We validated RepDB* on the Atlas 8-node cluster at LINA and another 64-node cluster at INRIA-Rennes. In 2004, we registered RepDB* to the APP (Agence pour la Protection des Programmes) and released it as Open Source Software under the GPL licence. Since then, RepDB* has been available for downloading (with more than a thousand downloads in the first three months).

In addition, we did an extensive performance validation based on the implementation of Preventive Replication in our RepDB* prototype over a cluster of 64 nodes running PostgreSQL. Our experimental results using the TPC-C benchmark show that our algorithm scales up very well and has linear response time behavior. We also showed the impact of the configuration on transaction response time. With partial replication, there is more inter-transaction parallelism than with full replication because of the nodes being specialized to different tables and thus transaction types. Thus, transaction response time is better with partial replication than with full replication (by about 15%). The speed-up experiment results showed that the increase of the number of nodes can well improve the query throughput. Finally, we showed that, with our optimistic approach, unordered transactions introduce very few aborts (at most 1%) and that the waiting delay for committing transactions is very small (and reaches zero as transaction time increases).

3.8 Related Work

Synchronous (eager) replication can provide strong consistency for most configurations including multi-master but its implementation, typically through 2PC, violates system autonomy and does not scale up. In addition, 2PC may block due to network or node failures. The synchronous solution proposed in [KA00b] reduces the number of messages exchanged to commit transactions compared to 2PC. It uses, as we do, group communication services to guarantee that messages are delivered at each node according to some ordering criteria. However, DBMS autonomy is violated because the implementation must combine concurrency control with group communication primitives. In addition solutions based on total order broadcast is not well suited for large scale replication because as the number of nodes increases the overhead of messages exchanged may dramatically increase to assure total order. The Database State Machine [SPOM01, SPMO02] supports partial replication for heterogeneous databases and thus does not violate autonomy. However, its synchronous protocol uses two-phase locking that is known for its poor scalability, thus making it inappropriate for database clusters.

Lazy replication typically trades consistency for performance. Early papers provide the user with a way to control inconsistency. A couple of weak consistency models have been constructed that provide correctness criteria weaker than 1-copy-serializability. For instance, Epsilon-serializability [PL91] measures the distance between database objects like the difference in the value or the number of updates applied. The application can therefore specify the amount of inconsistency tolerated by a transaction. N-Ignorance [KB91] is based on quorums. It relaxes the requirement that quorums must intersect in such a way that the inconsistencies introduced by concurrent transactions are bounded. In Mariposa [SAS+96] the frequency of update propagation depends on how much the maintainer of a replica is willing to pay. Also the degree of data freshness in a query is determined by the price a user wants to pay. However, different from preventive replication, these approaches implement the protocols in the database internals, what normally imposes DBMS homogeneity. Besides, making the choice of the right bound of inconsistency is a non-trivial problem and users must have a good understanding of the inconsistency metrics.

A refreshment algorithm that assures correctness for lazy master configurations is proposed in [PMS01]. This work does not consider multi-master and partial replication as we do.

The synchronous replication algorithms proposed in [JPKA02] provides one-copy-serializability for multi-master and partial replication while preserving DBMS autonomy. However, they require that transactions update a fixed primary copy: each type of transaction is associated with one node so a transaction of that type can only be performed at that node. Furthermore, the algorithm uses 2 messages to multicast the transaction, the first is a reliable multicast and the second is a total ordered multicast. The cost of these messages is higher than the single FIFO multicast message we use. However, one advantage of this algorithm is that it avoids redundant work: the transaction is performed at the origin node and the target nodes only apply the write set of the transaction. In our algorithm, all the nodes that hold the resources necessary for the transaction perform it entirely. We could also remove this redundant work to generalize the multicast of refresh transactions for all nodes instead of only for the nodes that do not hold all the necessary replicas. However, the problem is to decide whether it is faster to perform the transaction entirely or to wait for the corresponding write set from the origin node for short transactions. [JPKA02] also proposes similar optimizations that permit to overlap transaction processing with the time it takes to deliver total order. Finally their experiments do not show scale-up with more than 15 nodes while we go up to 64 in our experiments.

More recent work has focused on snapshot isolation to improve the performance of read-only transactions. The RSI-PC [PA04] algorithm is a primary copy solution which separates update transactions from read-only transactions. Update transactions are always routed to a main replica, whereas read-only transactions are handled by any of the remaining replicas, which act as read-only copies. Postgres-R(SI) [WK05] proposes a smart solution that does not need to declare transactions properties in advance. It uses the replication algorithm of [JPKA02] which must be implemented inside the DBMS. The experiments are limited to at most 10 nodes. SI-Rep [YKPJ05] provides a solution similar to Postgres-R(SI) on top of PostgreSQL which needs the write set of a transaction before its commitment. Write sets can be obtained by either extending the DBMS, thus hurting DBMS autonomy, or using triggers.

3.9 Conclusion

In this chapter, we proposed two algorithms for preventive replication in order to scale up to large cluster configurations [POC03, CGPV04,CPV05a,CPV05b,PCVO05]. The first algorithm supports fully replicated configurations where all the data are replicated on all the nodes, while the second algorithm supports partially replicated configurations, where only a part of the data are replicated. Both algorithms enforce strong consistency. Moreover, we presented two optimizations that improve transaction throughput; the first optimization eliminates optimistically the delay introduced by the preventive replication algorithm while the second optimization introduces concurrency control features outside the DBMS in which non conflicting incoming transactions may execute concurrently.

We did an extensive performance validation based on the implementation of Preventive Replication in our RepDB* prototype over a cluster of 64 nodes running PostgreSQL. Our experimental results using the TPC-C benchmark show that our algorithm scales up very well and has linear response time behavior. The performance gains strongly depend on the types of transactions and of the configuration. Thus, an important conclusion is that the configuration and the placement of the copies should be tuned to selected types of transactions. Annex C presents [PCVO05] to give a complete view of our contributions (formalizations, proofs, performance evaluation, etc).

4 Semantic Reconciliation in P2P Systems

In this chapter we present our contributions on large scale lazy multi-master master optimistic replication in P2P systems. We first present the kind of applications that motivated this research (P2P wiki). To enable the deployment of such applications in P2P networks, it is required a mechanism to deal with data sharing in a dynamic, scalable and available way. We chose the lazy multi-master optimistic approach. Previous work on optimistic replication has mainly concentrated on centralized systems. Centralized approaches are inappropriate for a P2P setting due to their limited availability and vulnerability to failures and partitions from the network. We focus on the design of a reconciliation algorithm to be deployed in large scale cooperative applications, such as P2P Wiki. The main contribution is a distributed reconciliation algorithm designed for P2P networks (P2P-reconciler).

4.1 Introduction

Peer-to-peer (P2P) systems adopt a completely decentralized approach to data sharing and thus can scale to very large amounts of data and users. Popular examples of P2P systems such as Gnutella [Gnu06] and KaaZa [Kaz06] have millions of users sharing petabytes of data over the Internet. Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems, such as Gnutella and KaaZa, which rely on flooding. This work led to structured solutions based on distributed hash tables (DHT), *e.g.* CAN [SMKK+01], Chord [SMKK+01], and Pastry [RD01b].

Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, P2P, and mobile computing). As an example of such applications, consider a second generation Wiki that works over a peer-to-peer (P2P) network and supports users on the elaboration and maintenance of shared documents in a collaborative and asynchronous manner. Consider also that each document is an XML file possibly linked to other documents. Wiki allows collaboratively managing a single document as well as composed, integrated documents (e.g. an encyclopedia or a knowledge base concerning the use of an open source operating system). Although the number of users that update in parallel a document *d* is usually small, the size of the collaborative network that holds *d* in terms of number of nodes may be large. For instance, the document *d* could belong to the knowledge base of the Mandriva Club, which is maintained by more than 25,000 members [Man07] or it could belong to Wikipedia, a free content encyclopedia maintained by more than 75,000 active contributors [Wik07].

Many users frequently need to access and update information even if they are disconnected from the network, e.g. in an aircraft, a train or another environment that does not provide good network connection. This requires that users hold local replicas of shared documents. Thus, a P2P Wiki requires optimistic multi-master replication to assure data availability at anytime. With this approach, updates made offline or in parallel on different replicas of the same data may cause replica divergence and conflicts, which should be reconciled. In order to resolve conflicts, the reconciliation solution can take advantage of application semantic to avoid inconsistent edition wrt. to the document contents. This motivated our research in semantic reconciliation. A detailed example of semantic reconciliation for text edition can be found in [MPEJ08].

Optimistic replication is largely used as a solution to provide data availability for these applications. It allows asynchronous updating of replicas such that applications can progress even though some nodes are disconnected or have failed. This enables asynchronous collaboration among users. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled. In most existing solutions [PSM03, SS05] reconciliation is typically performed by a single node (reconciler node) which may introduce bottlenecks. In addition, if the reconciler node fails, the entire replication system may become unavailable.

A theory for semantic reconciliation was set in IceCube [KRSD01, PSM03] to run in a single node. According to this theory, the application semantics can be described by means of constraints between update actions. A *constraint* is an application invariant, *e.g.* a *parcel* constraint establishes the "all-or-nothing" semantics, *i.e.* either all *parcel*'s actions execute successfully in any order, or none does. For instance, consider a user that improves the content of a shared document by producing two *related* actions a_1 and a_2 (*e.g.* a_1 changes a document paragraph and a_2 changes the corresponding translation); in order to assure the "all-or-nothing" semantics, the application should create a parcel constraint between a_1 and a_2. These actions can conflict with other actions. Therefore, the aim of reconciliation is to take a set of actions with the associated constraints and produce a *schedule*, *i.e.* a list of ordered actions that do not violate constraints.

Different from IceCube, we propose a fully distributed approach to perform P2P reconciliation in a distributed hash table (DHT) overlay [DZDKS03] . DHTs typically provide two basic operations [DZDKS03]: *put(k, data)* stores a key *k* and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with *k* in the DHT.

Variable latencies and bandwidths, typically found in P2P networks, may strongly impact the reconciliation performance once data access times may vary significantly from node to node. Therefore, in order to build a suitable P2P reconciliation solution, communication and topology awareness must be considered.

4.2 P2P semantic reconciliation

IceCube describes the application semantics by means of constraints between actions. An *action* is defined by the application programmer and represents an application-specific operation (*e.g.* a write operation on a file or document, or a database transaction). A *constraint* is the formal representation of an application invariant (*e.g.* an update cannot follow a delete). Constraints are classified as follows:

User-defined constraint[2]**.** Users and applications can create user-defined constraints to make their intents explicit. The *predSucc*(a_1, a_2) constraint establishes causal ordering between actions (*i.e.* action a_2 executes only after a_1 has succeeded); the *parcel*(a_1, a_2) constraint is an atomic (all-or-nothing) grouping (*i.e.* either a_1 and a_2 execute successfully or none does); the *alternative*(a_1, a_2) constraint provides choice of at most one action (*i.e.* either a_1 or a_2 is executed, but not both).

System-defined constraint[3]. It describes a semantic relation between classes of concurrent actions. The *bestOrder*(a_1, a_2) constraint indicates the preference to schedule a_1 before a_2 (*e.g.* an application for account management usually prefers to schedule credits before debits); the *mutuallyExclusive*(a_1, a_2) constraint states that either a_1 or a_2 can be executed, but not both.

Let us illustrate user- and system-defined constraints with the example that appears in Figure 4.1. In this example, an action is noted a_n^i, where n indicates the node that has executed the action and i is the action identifier. T is a replicated object, in this case, a relational table; K is the key attribute for T; A and B are any two other attributes of T. T_1, T_2, and T_3 are replicas of T. And *parcel* is a user-defined constraint that imposes atomic execution for a_3^1 and a_3^2. Consider that the actions in Figure 4.1 (with the associated constraints) are concurrently produced by nodes n_1, n_2 and n_3, and should be reconciled.

a_1^1: update T_1 set A=a1 where K=k1
a_2^1: update T_2 set A=a2 where K=k1
a_3^1: update T_3 set B=b1 where K=k1
a_3^2: update T_3 set A=a3 where K=k2
Parcel(a_3^1, a_3^2)

Figure 4.1 . Conflicting actions on T

In Figure 4.1, actions a_1^1 and a_2^1 try to update a copy of the same data item (*i.e.* T's tuple identified by *k1*). The IceCube reconciliation engine realizes this conflict and asks the application for the semantic relationship involving a_1^1 and a_2^1. As a result, the application analyzes the intents of both actions, and, as they are really in conflict (*i.e.* n_1 and n_2 try to set the same attribute with distinct values), the application produces a *mutuallyExclusive*(a_1^1, a_2^1) *system-defined constraint* to properly represent this semantic dependency. Notice that from the point of view of the reconciliation engine a_3^1 also conflicts with a_1^1 and a_2^1 (*i.e.* all these actions try to update a copy of the same data item). However, by analyzing actions' intents, the application realizes that a_3^1 is semantically independent of a_1^1 and a_2^1 as a_3^1 tries to update another attribute (*i.e.* B). Therefore, in this case no system-defined constraints are produced. Actions a_3^1 and a_3^2 are involved in a *parcel user-defined constraint*, so they are semantically related.

The aim of reconciliation is to take a set of actions with the associated constraints and produce a *schedule*, *i.e.* a list of ordered actions that do not violate constraints. In order to reduce the schedule

[2] *User-defined* constraint is called *log* constraint by IceCube. We prefer *user-defined* to emphasize the user intent.

[3] *System-defined* constraint is called *object* constraint by IceCube. We use *system-defined* to contrast with user intents.

production complexity, the set of actions to be ordered is divided into subsets called *clusters*. A cluster is a subset of actions related by constraints that can be ordered independently of other clusters. Therefore, the *global schedule* is composed by the concatenation of clusters' ordered actions. To order a cluster, IceCube performs iteratively the following operations:

Select the action with the highest merit from the cluster and put it into the schedule. The merit of an action is a value that represents the estimated benefit of putting it into the schedule (the larger the number of actions that can take part in a schedule containing a_i^n is, the larger the merit of a_i^n will be). If more than one action has the highest merit (different actions may have equal merits), the reconciliation engine selects randomly one of them.

1. Remove the selected action from the cluster.
2. Remove from the cluster the remaining actions that conflict with the selected action.

This iteration ends when the cluster becomes empty. As a result, cluster's actions are ordered. In fact, several alternative orderings may be produced until finding the best one.

4.3 Overview

We assume that P2P-reconciler is used in the context of a virtual community which requires a high level of collaboration and relies on a reasonable number of nodes (typically hundreds or even thousands of interacting users) [WI097]. The P2P network we consider consists of a set of nodes which are organized as a distributed hash table (DHT) [RFHK+01,SMKK+01]. A DHT provides a hash table abstraction over multiple computer nodes. Data placement in the DHT is determined by a hash function which maps data identifiers into nodes. In the remainder of this chapter we assume that the network is reliable and the lookup service of the DHT does not behave incorrectly.

We have structured the P2P reconciliation in 6 distributed steps in order to maximize parallel computing and assure independence between parallel activities. This structure improves reconciliation performance and availability (*i.e.* if a node fails, the activity it was executing is assigned to another available node).

With P2P-reconciler, data replication proceeds basically as follows. First, nodes execute local actions to update a replica of an object while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the DHT based on the object's identifier. Finally, reconciler nodes retrieve actions and constraints from the DHT and produce the global schedule, by reconciling conflicting actions in 6 distributed steps based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency [SBK04, SS05]. The replicated data is eventually consistent if, when all nodes stop the production of new actions, all nodes will eventually reach the same value in their local replicas.

In this protocol, we distinguish three types of nodes: the *replica node*, which holds a local replica; the *reconciler node*, which is a replica node that participates in distributed reconciliation; and the *provider node*, which is a node in the DHT that holds data consumed or produced by the reconcilers (*e.g.* the node that holds the schedule is called *schedule provider*).

We concentrate the reconciliation work in a subset of nodes (the reconciler nodes) in order to maximize performance. If we do not limit the number of reconcilers, the following problems take place. First, provider nodes and the network as a whole become overloaded due to a large number of messages aiming to access the same subset of DHT data in a very short time interval. Second, nodes with high-latencies and low-bandwidths can waste a lot of time with data transfer, thereby hurting the reconciliation response time. Our strategy does not create improper imbalance in the load of reconciler nodes as reconciliation activities are not processing intensive.

4.4 Reconciliation objects

Data managed by P2P-reconciler during reconciliation are held by reconciliation objects that are stored in the DHT giving the object identifier. To enable the storage and retrieval of reconciliation objects, each reconciliation object has a unique identifier. P2P-reconciler uses the following reconciliation objects:

Action log *R* (noted L_R): it holds all actions that try to update any replica of the object *R*

Clusters set (noted *CS*): recall that a cluster contains a set of actions related by constraints, and can be ordered independently from other clusters when producing the global schedule. All clusters produced during reconciliation are stored in the *clusters set* reconciliation object.

Action summary (noted *AS*): it captures semantic dependencies among actions, which are described by means of constraints. In addition, the action summary holds relationships between actions and clusters, so that each relationship describes an action membership (an action is a *member* of one or more clusters).

Schedule (noted *S*): it contains an ordered list of actions, which is composed from the concatenation of clusters' ordered actions.

Reconciliation objects are guaranteed to be available using known DHT replication solutions [KWR05]. P2P-reconciler's liveness relies on the DHT liveness.

4.5 P2P-reconciler Distributed Steps

P2P-reconciler executes reconciliation in 6 distributed steps as showed in Figure 4.2. Any connected node can start reconciliation by inviting other available nodes to engage with it. In the 1st step (node allocation), a subset of engaged nodes is allocated to step 2, another subset is allocated to step 3, and so forth until the 6th step (details about node allocation are provided in the next sections). Nodes at step 2 start reconciliation. The outputs produced at each step become the input to the next one. In the following, we describe the activities performed in each step, and we illustrate parallel processing by explaining how these activities could be executed simultaneously by two reconciler nodes, n_1 and $n_{2.}$

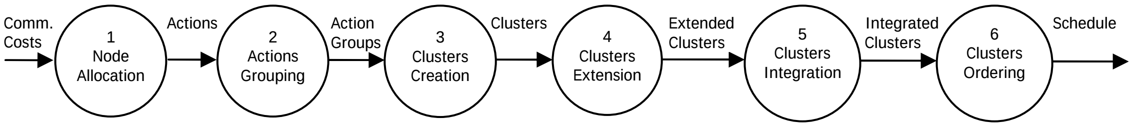


Figure 4.2 P2P-reconciler steps

Step 1 – node allocation: a subset of connected replica nodes is selected to proceed as reconciler nodes.

Step 2 – actions grouping: reconcilers take actions from the action log and put actions that try to update common object items into the same group.

Step 3 – clusters creation: reconcilers take action groups from the action log and split them into clusters of semantically dependent conflicting actions; system-defined constraints are created to represent the semantic dependencies detected in this step; these constraints and the action memberships that describe the association between actions and clusters are included in the action summary; clusters produced in this step are stored in the clusters set.

Step 4 – clusters extension: user-defined constraints are not taken into account in clusters creation (e.g. although a_3^1 and a_3^2 belong to a *parcel*, the previous step does not put them into the same cluster, because they do not update a common object item). Thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints. These extensions lead to new relationships between actions and clusters, which are represented by new action memberships; the new memberships are included in the action summary.

Step 5 – clusters integration: clusters extensions lead to cluster overlapping (an overlap occurs when the intersection of two clusters results a non-null set of actions); in this step, reconcilers bring together overlapping clusters. At this point, clusters become mutually-independent, i.e. there are no constraints involving actions of distinct clusters.

Step 6 – clusters ordering: in this step, reconcilers take clusters from the clusters set and order clusters' actions; the ordered actions associated with each cluster are stored in the *schedule* reconciliation object (*S*); the concatenation of all clusters' ordered actions makes up the global schedule that is executed by all replica nodes.

At every step, the P2P-reconciler algorithm takes advantage of data parallelism, i.e. several nodes perform simultaneously independent activities on a distinct subset of actions (e.g. ordering of different clusters). No centralized criterion is applied to partition actions. Indeed, whenever a set of reconciler nodes requests data from a provider, the provider node naively supplies reconcilers with about the same amount of data (the provider node knows the maximal number of reconcilers because it receives this information from the node that launches reconciliation).

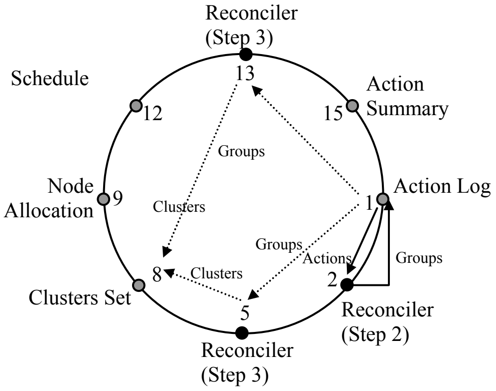


Figure 4.3 P2P-reconciler at work

4.5.1 P2P-reconciler at work

We now illustrate the execution of the P2P-reconciler algorithm over a Chord DHT [SMKK+01] network. For simplicity, we consider only its first 3 steps and a few nodes at work. Figure 4.3 shows 8 nodes and their respective roles in the reconciliation protocol. All of them are replica nodes. Reconciliation objects are stored at provider nodes according to the hashed values associated with the reconciliation object identifiers (e.g. Chord maps a hashed value *v* to the first node that has an identifier equal to or greater than *v* in the circle of ordered node identifiers). In this example, we assume that Chord maps the hashed value of the action log identifier to node 1; using the same principle, the clusters set, the schedule and the action summary are mapped respectively to nodes 8, 12 and 15. Finally, node 9 is responsible for allocating reconcilers.

Any node can start the reconciliation by triggering the step 1 of P2P-reconciler at the appropriate node (*e.g.* node 9), which selects the best reconcilers and notifies them the steps they should perform. In our example, node 9 selects node 2 to execute step 2, and selects nodes 5 and 13 to perform step 3 (details about node allocation are provided in the next sections).

Node 2 starts the step 2 of reconciliation by retrieving actions from the action log (stored at node 1) in order to arrange them in groups of actions on common object items. At the same time, nodes 5 and 13 begin step 3 by requesting action groups to node 1; these requests are held in a queue at node 1 while action groups are under construction. When node 2 stores action groups at action log, node 1 replies the requests previously queued by nodes 5 and 13. At this moment, step 2 is terminated and step 3 proceeds. Notice that each reconciler works on independent data (*e.g.* nodes 5 and 13 receive distinct action groups from node 1). To assure this independence, provider nodes segment the data they hold based on the number of reconcilers (*e.g.* node 1 creates two segments of action groups, one for node 5 and another for node 13).

When step 2 terminates, nodes 5 and 13 receive action groups from node 1 and produce the corresponding clusters of actions, which are stored at node 8. In turn, node 8 replies requests for clusters that reconcilers of step 4 have previously queued; and so forth, until the end of step 6.

4.6 Dealing with dynamism

Whenever distributed reconciliation takes place, a set of nodes N_d may be disconnected. As a result, the global schedule is not applied by the nodes of N_d. Moreover, actions produced by N_d nodes and not yet stored in the P2P network are not reconciled. We need a new reconciliation object to assure eventual consistency in the presence of disconnections. Thus, we define *schedule history*, noted H, as a reconciliation object that stores a chronological sequence of schedules' identifiers produced by reconciliations ($H = (S_{id1}, \ldots, S_{idn})$). A replica node can check whether it is *up to date* by comparing the identifier of the last schedule it has locally executed with S_{idn}. Fault-tolerance aspects relies on the DHT, that normally achieved by replicating each peer data at its neighbor.

4.7 DHT cost model

A DHT network is usually built on top of the Internet, which consists of nodes with variable latencies and bandwidths. As a result, the network costs involved in DHT data accesses may vary significantly from node to node and have a strong impact in the reconciliation performance. Thus, network costs should be considered to perform reconciliation efficiently. We propose a basic cost model for computing communication costs in DHTs based on *look-up*, *direct-cost* and *transfer costs*. The *lookup cost*, noted $lc(n, id)$, is the latency time spent in a lookup operation launched by node n to find the data item identified by *id*. Similarly, *direct cost*, noted $dc(n_i, n_j)$, is the latency time spent by node n_i to directly access n_j. And the *transfer cost*, noted $tc(n_i, n_j, d)$, is the time spent to transfer the data item d from node n_i to node n_j, which is computed based on d's size and the bandwidth between n_i and n_j. On top of it, we can build customized cost models.

In the basic cost model, we define communication costs (henceforth costs) in terms of latency and transfer times, and we assume links with variable latencies and bandwidths. In order to exploit bandwidth, the application behavior in terms of data transfer should be known. Since this behavior is application-specific, we exploit bandwidth in higher-level customized models.

4.8 P2P-reconciler node allocation

The first step of P2P-reconciler aims to select the best nodes to proceed as reconcilers in order to maximize performance. The number of reconcilers has a strong impact on the reconciliation time. Thus, this section concerns the estimation of the optimal number of reconcilers per step as well as the allocation of the best nodes. We first determine the maximal number of reconciler nodes using polynomial regression (for details see [MPV06a]). Then, we use the P2P-reconciler cost model for computing the cost of each reconciliation step. Next, the cost provider node selects reconcilers based on P2P-reconciler cost model. We proposed an approach for managing the dynamic behavior of P2P-reconciler costs.

4.8.1 P2P-reconciler cost model

The P2P-reconciler cost model is built on top of the DHT cost model by taking into account each reconciliation step and defining a new metric: node step cost. The *node step cost*, noted *cost*(*i, n*), is the sum of lookup (noted *lc*) , direct access (noted *dc*), and transfer costs (noted *tc*) estimated by node *n* for executing step *i* of P2P-reconciler algorithm.

By analyzing the P2P-reconciler behavior in terms of lookup, direct access, and data transfer operations at every step, we produced a cost formula for each step of P2P-reconciler, which are shown in Table 4.1. There is no formula associated with step 1 because it is not performed by reconciler nodes.

Step *i*	***Cost(i, n)***
2	$lc(n, L_R) + 2\times dc(n, n_{LR}) + tc(n_{LR}, n, actSet) + lc(n, L_R) + dc(n, n_{LR}) + tc(n, n_{LR}, grpSet)$
3	$lc(n, L_R) + 3\times dc(n, n_{LR}) + tc(n_{LR}, n, grpSet) + lc(n, CS) + 2\times dc(n, n_{CS}) +$ $tc(n, n_{CS}, [cluSet + cluIds]) + lc(n, AS) + dc(n, n_{AS}) + tc(n, n_{AS}, [sdcSet + m_3Set])$
4	$lc(n, CS) + 3\times dc(n, n_{CS}) + tc(n_{CS}, n, cluSet) + 2\times lc(n,AS) + 3\times dc(n, n_{AS}) + tc(n, n_{AS}, m_4Set)$
5	$lc(n, AS) + 3\times dc(n, n_{AS}) + tc(n_{AS}, n, mSet) + lc(n, CS) + dc(n, n_{CS}) + tc(n, n_{CS}, ovlCluSet)$
6	$lc(n, CS) + 3\times dc(n, n_{CS}) + tc(n_{CS}, n, itgCluSet) + lc(n, AS) + 2\times dc(n, n_{AS}) +$ $tc(n_{AS}, n, sumActSet) + lc(n, S) + dc(n,n_S) + tc(n, n_S, ordActSet)$

Table 4.1 P2P-reconciler cost model

For instance in formula 2, *lc(n,* L_R*)* is the lookup cost from node *n* to the node that holds L_R, necessary to locate the provider of *R*. Due to space limitations we do not go further on the cost model discussion. Annex D, that corresponds to [MPEJ08], presents all the precisions on how these formulas are estimated and used.

4.9 Node allocation

Node allocation is the first step of P2P-reconciler protocol as shown in Figure 4.2. It aims to select for every succeeding step a set of reconciler nodes that can perform reconciliation with good performance. We define a new reconciliation object needed in node allocation. Then we describe how reconciler nodes are chosen, and we illustrate that with an example.

We define *communication costs*, noted *CC*, as a reconciliation object that stores the *node step costs* estimated by every replica node and used to choose reconcilers before starting reconciliation.

The node that holds *CC* in the DHT at a given time is called *cost provider*, and it is responsible for allocating reconcilers. The allocation works as follows. Replica nodes locally estimate the costs for executing every P2P-reconciler step, according to the P2P-reconciler cost model, and provide this information to the cost provider. The node that starts reconciliation computes the maximal number of reconcilers per step (*maxRec*), and asks the cost provider for allocating at most *maxRec* reconciler nodes per P2P-reconciler step. As a result, the cost provider selects the best nodes for each step and notifies these nodes of the P2P-reconciler steps they should execute.

In our solution, the cost management is done in parallel with reconciliation. Moreover, it is network optimized since replica nodes do not send messages to cost providers, informing their estimated costs, if the node step costs overtake the *cost limit*. For these reasons, the cost provider does not become a bottleneck.

4.9.1 Managing the dynamic costs

The costs estimated by replica nodes for executing P2P-reconciler steps change as a result of disconnections and reconnections. To cope with this dynamic behavior and assure reliable cost estimations, a replica node n_i works as follows:

Initialization: whenever n_i joins the system, n_i estimates its costs for executing every P2P-reconciler step. If these costs do not overtake the cost limit, n_i supplies the cost provider with this information.

Refreshment: while n_i is connected, the join or leave of another node n_j may invalidate n_i's estimated costs due to routing changes. Thus, if the join or leave of n_j is relevant to n_i, n_i recomputes its P2P-reconciler estimated costs and refreshes them at the cost provider.

Termination: when n_i leaves the system, if its P2P-reconciler estimated costs are smaller than cost limit (*i.e.* the cost provider holds n_i's estimated costs), n_i notifies its departure to the cost provider.

P2P-reconciler computes the cost limit based on these parameters: the expected average latency of the network (*e.g.* 150ms for the Internet), and the expected average number of hops to lookup a reconciliation object (*e.g.* $log(n)/2$ for a Chord DHT, where n represents the number of connected nodes and can be established as 15% of the community size).

4.10 P2P-reconciler-TA protocol

P2P-reconciler-TA is a distributed protocol for reconciling conflicting updates in *topology-aware* P2P networks. Given a set of nodes, we exploit topological information to select the “best” nodes to participate in the different steps of an algorithm, in a way that achieves an optimal performance. A P2P network is classified as topology-aware if its topology is established by taking into account the physical distance among nodes (e.g. in terms of latency times).

Several topology-aware P2P networks could be used to validate our approach such as Pastry [RD01a], Tapestry [ZHSR+04], CAN [RFHK+01], etc. We chose to construct our P2P-reconciler-TA over optimized CAN because it allows building the topology-aware overlay network in a relatively simple manner. In addition, its routing mechanism is easy to implement, although less efficient than other topology-aware P2P networks (e.g. the average routing path length in CAN is usually grater than in other structured P2P networks).

Basic CAN [RFHK+01] is a virtual Cartesian coordinate space to store and retrieve data as (*key, value*) pairs. At any point in time, the entire coordinate space is dynamically partitioned among all nodes in the system, so that each node owns a distinct zone that represents a segment of the entire space. To store (or retrieve) a pair (k_1, v_1), key k_1 is deterministically mapped onto a point *P* in the coordinate space using a uniform hash function, and then (k_1, v_1) is stored at the node that owns the zone to which *P* belongs. Intuitively, routing in CAN works by following the straight line path through the Cartesian space from source to destination coordinates.

Optimized CAN aims at constructing its logical space in a way that reflects the topology of the underlying network. It assumes the existence of well-known landmarks spread across the network. A node measures its round-trip time to the set of landmarks and orders them by increasing latency (i.e. network distance). The coordinate space is divided into bins such that each possible landmarks ordering is represented by a bin. Physically close nodes are likely to have the same ordering and hence will belong to the same bin.

Briefly, P2P-reconciler-TA works as follows. Based on the network topology, it selects the best provider and reconciler nodes. These nodes then reconcile conflicting updates and produce a schedule, which is an ordered list of non-conflicting updates. We proposed a dynamic distributed algorithm for efficiently selecting provider and reconciler nodes.

4.11 Validation

We validated and evaluated the performance of our reconciliation solutions through experimentation using the APPA prototype [AM07]. The validation of the APPA replication service took place over the Grid5000 platform [Gri06]. Grid5000 aims at building a highly reconfigurable, controllable and monitorable experimental Grid platform, gathering 9 sites geographically distributed in France featuring a total of 5000 nodes. Within each site, the nodes are located in the same geographic area and communicate through Gigabit Ethernet links as clusters. Communications between clusters are made through the French academic network (RENATER). Grid5000's nodes are accessible through the OAR batch scheduler from a central user interface shared by all the users of the Grid. A cross-clusters super-batch system, OARGrid, is currently being deployed and tested. The home directories of the users are mounted with NFS on each of the infrastructure's clusters. Data can thus be directly accessed inside a cluster. Data transfers between clusters have to be handled by the users. The storage capacity inside each cluster is a couple of hundreds of gigabytes. Now more than 600 nodes are involved in Grid5000. Additionally, in order to study the scalability of the APPA replication service with larger numbers of nodes that are connected by means of links with variable latencies and bandwidths, we implemented simulators using Java and SimJava [HM98], a process based discrete event simulation package. Simulations were executed on an Intel Pentium IV with a 2.6 GHz processor, and 1 GB of main memory, running the Windows XP operating system. The performance results obtained from the simulator are consistent with those of the replication service prototype.

In the implementation intended for the Grid5000 platform, each peer manages multiple tasks in parallel (e.g. routing DHT messages, executing a DSR step, etc.) by using multithreading and other associated mechanisms (e.g. semaphores); in addition, peers communicate with each other by means of sockets and UDP depending on the message type. To have a topology close to real P2P overlay networks in this Grid platform, we determine the peers' neighbors and we allow that every peer communicate only with its neighbors in the overlay network. Although the Grid5000 provides fast and reliable communication, which usually is not the case for P2P systems, it allows to validate the accuracy of APPA distributed algorithms and to evaluate the scalability of APPA services. We have deployed APPA over this platform because it was the largest network available to perform our experiments in a controllable manner. On the other hand, the implementation of the simulator conforms to the SimJava model with respect to parallel processing and peers communication. It is important to note that, in our simulator, only the P2P network topology and peer communications are simulated; full-fledged APPA services are deployed on top of the simulated network.

The experimental results showed that our cost-based reconciliation outperforms the random approach by a factor of 26. In addition, the number of connected nodes is not important to determine the reconciliation performance due to the DHT scalability and the fact that reconcilers are as close as possible to the reconciliation objects. The action size impacts the reconciliation time in a logarithmic scale. Our algorithm yields high data availability and excellent scalability, with acceptable performance and limited overhead.

In the same way, we also validated P2P-reconciler-TA. The experimental results show that our topology-aware approach achieves a performance improvement of 50 % in comparison with the P2P-reconciler. In addition, P2P-reconciler-TA has proved to be scalable with limited overhead and thereby suitable for P2P environments. Our topology-aware approach is conceived for distributed reconciliation; however our metrics, costs functions as well as our selection approach are useful in several contexts.

4.12 Related work

In the context of P2P networks, there has been little work on managing data replication in the presence of updates. Most of data sharing P2P networks consider the data they provide to be very static or even read-only. Freenet [CMHS+02] partially addresses updates which are propagated from the updating peer downward to close peers that are connected. However, peers that are disconnected do not get updated. P-Grid [ACDD0+3, AHA03] is a structured P2P network that exploits epidemic algorithms to address updates. It assumes that conflicts are rare and their resolution is not necessary in general. In addition, P-Grid assumes that probabilistic guarantees instead of strict consistency are sufficient. Moreover, it only considers updates at the file level. In OceanStore [KBCC+00] every update creates a new version of the data object. Consistency is achieved by a two-tiered architecture: a client sends an update to the object's primary copies and some secondary replicas in parallel. Once the update is committed, the primary copies multicast the result of the update down the dissemination tree. OceanStore assumes an infrastructure comprised of servers that are connected by high-speed links. Different from the previous works, we propose to distribute the reconciliation engine in order to provide high availability.

Table 4.2 compares the replication solutions provided by different types of P2P systems. Clearly, none of them provide eventual consistency among replicas along with weak network assumptions, which is the main concern of this work.

The distributed log-based reconciliation algorithms proposed by Chong and Hamadi [CH06] addresses most of our requirements, but this solution is unsuitable for P2P systems as it does not take into account the dynamic behavior of peers and network limitations. Operational transformation [VCFS00, MOSI03, FVC04] also addresses eventual consistency among replicas specifically for text edition at the character level with no flexibility wrt. to semantic constraint specification.

P2P System	**P2P Network**	**Data Type**	**Autonomy**	**Replication Type**	**Conflict Detection**	**Consistency**	**Network Assump.**
Napster	Super-peer	File	Moderate	Static data	–	–	Weak
JXTA	Super-peer	Any	High	–	–	–	Weak
Gnutella	Unstructured	File	High	Static data	–	–	Weak
Chord	Structured (DHT)	Any	Low	Single-master Multi-master	Concurrency None	Probabilistic Probabilistic	Weak
CAN	Structured (DHT)	Any	Low	Static data Multi-master	– None	– Probabilistic	Weak
Tapestry	Structured (DHT)	Any	High	–	–	–	Weak
Pastry	Structured (DHT)	Any	Low	–	–	–	Weak
Freenet	Structured	File	Moderate	Single-master	None	No guarantees	Weak
PIER	Structured (DHT)	Tuple	Low	–	–	–	Weak
OceanStore	Structured (DHT)	Any	High	Multi-master	Concurrency	Eventual	Strong
PAST	Structured (DHT)	File	Low	Static data	–	–	Weak
P-Grid	Structured	File	High	Multi-master	None	Probabilistic	Weak

Table 4.2 Comparing replication solutions in P2P systems

In the context of APPA (Atlas Peer-to-Peer Architecture), a P2P data management system which we are building [AMPV06b, [MAPV06], we proposed the *DSR-cluster* algorithm [MPJV06, MPV05], a distributed version of the semantic reconciliation engine of IceCube [KRSD01, PSM03] for cluster networks. However, *DSR-cluster* does not take into account network costs during reconciliation. A fundamental assumption behind *DSR-cluster* is that the communication costs among cluster nodes are negligible. This assumption is not appropriate for P2P systems, which are usually built on top of the Internet. In this case, network costs may vary significantly from node to node and have a strong impact on the performance of reconciliation.

4.13 Conclusion

Our main contribution related to optimistic lazy multi-master replication is P2P-reconciler, a distributed protocol for semantic reconciliation in P2P networks [MPV05,MPJV06,MAPV06,MP06,AMPV06a,MPV06a,MPV06b,EPV07,MPEJ08]. Other relevant related contributions are: the cost model for computing communication costs in DHTs and an algorithm that takes into account these costs and the P2P-reconciler steps to select the best reconciler nodes. For computing communication costs, we use local information and we deal with the dynamic behavior of nodes.

Another important contribution is the topology-aware approach to improve response times in P2P distributed semantic reconciliation. The P2P-reconciler-TA algorithm dynamically takes into account the physical network topology combined with the DHT properties when executing reconciliation. We proposed topology-aware metrics and cost functions to be used for dynamically selecting the best nodes to execute reconciliation, while considering dynamic data placement.

We validated P2P-reconciler and P2P-reconciler-TA through implementation and simulation. The experimental results showed that our cost-based reconciliation has very impressive results in scaling up, showing the relevance of our work for P2P collaborative applications. In addition, experimental results show that our topology-aware approach achieves a performance improvement of 50 % in comparison with the P2P-reconciler. In addition, P2P-reconciler-TA has proved to be scalable with limited overhead and thereby suitable for P2P environments. Our topology-aware approach is conceived for distributed reconciliation; however our metrics, costs functions as well as our selection approach are useful in several contexts. Annex D presents [MPEJ08] to give a larger view of our contributions (formalization, detailed algorithms, performance evaluation, etc).

5 Data Currency in Structured P2P Networks

Distributed Hash Tables (DHTs) provide a scalable solution for data sharing in P2P systems. To ensure high data availability, DHTs typically rely on data replication, yet without data currency guarantees. Supporting data currency in replicated DHTs is difficult as it requires the ability to return a current replica despite peers leaving the network or concurrent updates. In this chapter, we describe an Update Management Service (UMS) to deal with data availability and efficient retrieval of current replicas based on timestamping. For generating timestamps, we proposed a Key-based Timestamping Service (KTS) which performs distributed timestamp generation using local counters.

5.1 Introduction

While there are significant implementation differences between DHTs [SMKK+1, SMKK+1, RD01b], they all map a given key k onto a peer p using a hash function and can lookup p efficiently, usually in O(*log n)* routing hops where n is the number of peers. DHTs typically provide two basic operations [SMKK+01]: *put(k, data)* stores a key k and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with k in the DHT.

One of the main characteristics of P2P systems is the dynamic behavior of peers which can join and leave the system frequently, at anytime. When a peer gets offline, its data becomes unavailable. To improve data availability, most DHTs rely on data replication by storing (*k, data*) pairs at several peers, *e.g.* using several hash functions [RFHK+1]. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. However, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Let us illustrate the problem with a simple update scenario in a typical DHT. Let us assume that the operation *put*(k, d_0) (issued by some peer) maps onto peers p_1 and p_2 which both get to store the data d_0. Now consider an update (from the same or another peer) with the operation *put*(k, d_1) which also maps onto peers p_1 and p_2. Assuming that p_2 cannot be reached, *e.g.* because it has left the network, then only p_1 gets updated to store d_1. When p_2 rejoins the network later on, the replicas are not consistent: p_1 holds the *current* state of the data associated with k while p_2 holds a *stale* state. Concurrent updates also cause inconsistency. Consider now two updates *put*(k, d_2) and *put*(k, d_3) (issued by two different peers) which are sent to p_1 and p_2 in reverse order, so that p_1's last state is d_2 while p_2's last state is d_3. Thus, a subsequent *get(k)* operation will return either stale or current data depending on which peer is looked up, and there is no way to tell whether it is current or not. For some applications (*e.g.* agenda management, bulletin boards, cooperative auction management, reservation management, etc.) which could take advantage of a DHT, the ability to get the current data is very important.

Many solutions have been proposed in the context of distributed database systems for managing replica consistency [OV99] but the high numbers and dynamic behavior of peers make them no longer applicable to P2P [KWR05]. Supporting data currency in replicated DHTs requires the ability to return a current replica despite peers leaving the network or concurrent updates. The problem is partially addressed in [DGY03] using data versioning. Each replica has a version number which is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica. In the remainder of this chapter we assume that the network is reliable and the lookup service of the DHT does not behave incorrectly.

In this chapter, we present our solution to data currency in DHT [APV07a] (see Annex E for all details).

5.2 Data Currency in DHT using UMS and KTS

We provided a complete solution to data availability and data currency in replicated DHTs [APV07a]. This solution is the basis for a service called Update Management Service (UMS) which deals with efficient insertion and retrieval of current replicas based on timestamping. Experimental validation has shown that UMS incurs very little overhead in terms of communication cost. After retrieving a replica, UMS detects whether it is current or not, *i.e.* without having to compare with the other replicas, and returns it as output. Thus, UMS does not need to retrieve all replicas to find a current one. UMS only requires the DHT's lookup service with *put* and *get* operations.

To provide high data availability, the data is replicated in the DHT using a set of independent hash functions H_r, called replication hash functions. The peer that is responsible for *k* wrt *h* at the current time is denoted by *rsp(k,h)*. To be able to retrieve a current replica, each pair *(k, data)* is "stamped" with a logical timestamp, and for each $h \in H_r$, the pair *(k, newData)* is replicated at *rsp(k,h)* where *newData={data, timestamp}*, i.e. newData is a data composed of the initial data and the timestamp. Upon a request for the data associated with a key, we can thus return one of the replicas which are stamped with the latest timestamp. The number of replication hash functions, *i.e.* $|H_r|$, can be different for different DHTs. For instance, if in a DHT the availability of peers is low, for increasing data availability a high value of $|H_r|$ (*e.g.* 30) is used.

To generate timestamps, UMS uses a distributed service called *Key-based Timestamping Service (KTS)*. The main operation of KTS is *gen_ts(k)* which given a key *k* generates a real number as a *timestamp for k*. The timestamps generated by KTS have the *monotonicity* property, *i.e.* two timestamps generated for the same key are monotonically increasing. This property allows ordering the timestamps generated for the same key according to the time at which they have been generated. KTS has another operation denoted by *last_ts(k)* which given a key *k* returns the last timestamp generated for *k* by KTS.

At anytime, *gen_ ts (k)* generates at most one timestamp for *k*, and different timestamps for *k* have the monotonicity property. Thus, in the case of concurrent calls to insert a pair *(k, data)*, *i.e.* from different peers, only the one that obtains the latest timestamp will succeed to store its data in the DHT.

Through probabilistic analysis, we compute the expected number of replicas which UMS must retrieve for finding a current replica. Except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of available replicas are current then the expected number of retrieved replicas is less than 3.

5.3 Validation

We validated our solution through implementation and experimentation over a 64-node cluster of GRID 5000 and evaluated its scalability through simulation over 10,000 peers using SimJava. We compared the performance of UMS and BRK (from the BRICK project [KWR05]) which we used as baseline algorithm. The experimental and simulation results show that using KTS, UMS achieves major performance gains, in terms of response time and communication cost, compared with BRK. The response time and communication cost of UMS grow logarithmically with the number of peers of the DHT. Increasing the number of replicas, which we replicate for each data in the DHT, increases very slightly the response time and communication cost of our algorithm. In addition, even with a high number of peer fails, UMS still works well. In summary, this demonstrates that data currency, a very important requirement for many applications, can now be efficiently supported in replicated DHTs.

The current implementation of our prototype in APPA is based on Open Chord [OCH] which is an open source implementation of the Chord protocol. Open Chord is distributed under the GNU General Public License (GPL). It provides all DHT functionalities which are needed for implementing UMS and KTS, *e.g.* lookup, get and put functions.

In our prototype, peers are implemented as Java objects. They can be deployed over a single machine or several machines connected together via a network. Each object contains the code which is needed for implementing UMS and KTS. To communicate between peers, we use Java RMI [JRMI]which allows an object to invoke a method on a remote object.

The prototype provides a GUI that enables the user to manage the DHT network (*e.g.* create the DHT, add/remove peers to/from the system, etc.), store/retrieve data in/from the DHT, monitor the data stored at each peer, the keys for which the peer has generated a timestamp, the set of its initiated counters, etc.

5.4 Related Work

In the context of distributed systems, data replication has been widely studied to improve both performance and availability. Many solutions have been proposed in the context of distributed database systems for managing replica consistency [OV99], in particular, using eager or lazy (multi-master) replication techniques. However, these techniques either do not scale up to large numbers of peers or raise open problems, such as replica reconciliation, to deal with the open and dynamic nature of P2P systems.

Data currency in replicated databases has also been widely studied [BFGR06, GLR05, RBSS02]. However, the main objective is to trade currency and consistency for performance while controlling the level of currency or consistency desired by the user. Our objective in this paper is different, *i.e.* return the current (most recent) replica as a result of a get request.

Most existing P2P systems support data replication, but without consistency guarantees. For instance, Gnutella [Gnu06] and KaZaA [Kaz06], two of the most popular P2P file sharing systems allow files to be replicated. However, a file update is not propagated to the other replicas. As a result, multiple inconsistent replicas under the same identifier (filename) may co-exist and it depends on the peer that a user contacts whether a current replica is accessed.

PGrid is a structured P2P system that deals with the problem of updates based on a rumor-spreading algorithm. It provides a fully decentralized update scheme, which offers probabilistic guaranties rather than ensuring strict consistency. However, replicas may get inconsistent, *e.g.* as a result of concurrent updates, and it is up to the users to cope with the problem.

The Freenet P2P system [CMH02] uses a heuristic strategy to route updates to replicas, but does not guarantee data consistency. In Freenet, the query answers are replicated along the path between the peers owning the data and the query originator. In the case of an update (which can only be done by the data's owner), it is routed to the peers having a replica. However, there is no guarantee that all those peers receive the update, in particular those that are absent at update time.

The BRICKS project [KWR05] deals somehow with data currency by considering the currency of replicas in the query results. For replicating a data, BRICKS stores the data in the DHT using multiple keys, which are correlated to the key *k* by which the user wants to store the data. There is a function that, given *k*, determines its correlated keys. To deal with the currency of replicas, BRICKS uses versioning. Each replica has a version number which is increased after each update. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica. In addition, to return a current replica, all replicas need be retrieved in order to select the latest version. In our solution, concurrent updates raise no problem, *i.e.* this is a consequence of the monotonicity property of timestamps which are generated by KTS. In addition, our solution does not need to retrieve all replicas, and thus is much more efficient.

5.5 Conclusion

We proposed an Update Management Service (UMS) to deal with data availability and efficient retrieval of current replicas based on timestamping. For generating timestamps, we propose a Key-based Timestamping Service (KTS) which performs distributed timestamp generation using local counters. The validation of UMS-KTS shows excellent in scalability and outperforms significantly the BRICK solution. Annex E presents [APV07a] which provides all details of our contributions (formalizations, detailed algorithms, proofs, performance evaluation, etc).

Several important P2P applications can take advantage of KTS. For instance a reconciliation engine can exploit the use KTS to provide timestamps on tentative actions. This actions may be then reconciled based on the timestamp numbers. This research topic is presented as part of future work.

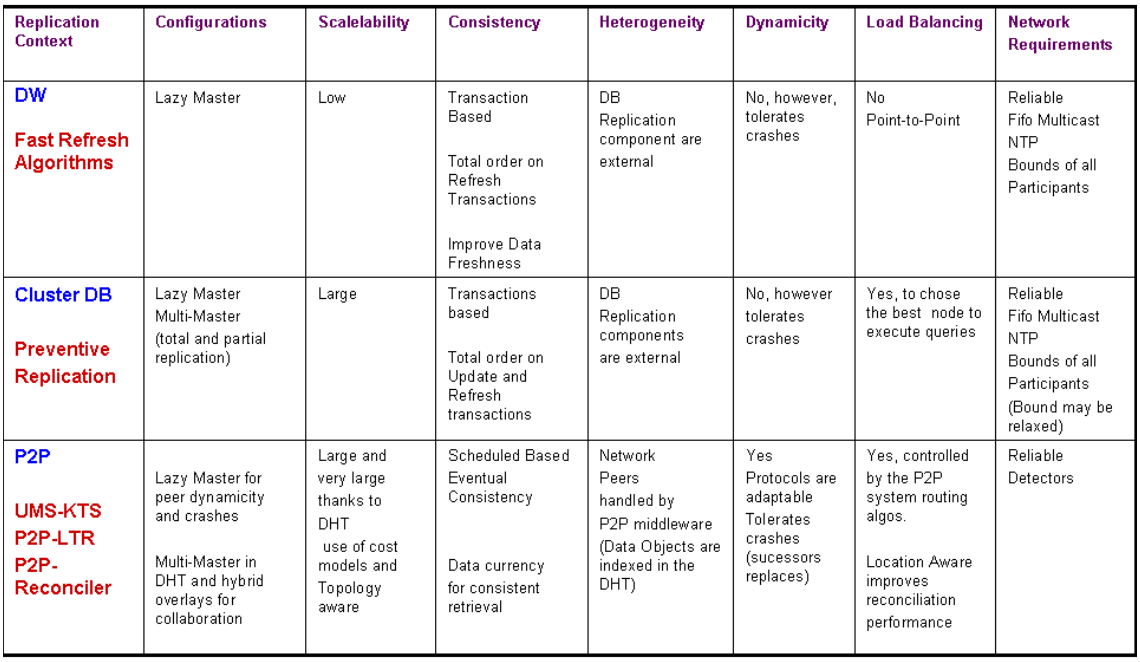
6 Conclusions and Future Work

6.1 General Conclusions

In this report, we presented our contributions on lazy data replication in different contexts (data warehouse, database clusters, P2P Systems). Since we presented the main related work in each chapter, we now summarize the main lessons learned. We learned that important parameters should be considered when using lazy replication for distributed applications: *configuration, scalability, application consistency requirements, database visibility, node dynamicity, node and network load balancing, and network characteristics*. These parameters define the relevant requirements for consistency management. Table 6.1 summarizes our research contributions considering all parameters, which we discuss below.

The configuration parameter is related to where replicas can be updated. In a small scale distributed systems, normally lazy single master configurations are sufficient because for the type of application we consider (small scale OLAP and OLTP), the single master is rarely a bottleneck. Furthermore load balancing is not an issue. This context corresponds to that of Chapter 2. With respect to network characteristics, since nodes are static we can safely employ FIFO reliable multicast without degrading the overall performance. In addition, for clock synchronization we can rely on network time protocols [Mil06]. With respect to consistency management, we expressed total order requirements in terms of group communication primitives. In this context, our solution for managing replica consistency is to improve data freshness by using immediate propagation strategies. To assure total order we proposed a refresher algorithm which delays the execution of refresh transactions until its deliver time. Given our parameter setting, this solution is quite acceptable and simple to implement while respecting database autonomy.

In database clusters, from the user perspective, the system should appear as a single unit. Adaptability plays a crucial role in providing the user with the desired quality of service. Thus, several nodes must have the right to update replicated data to enable the system to adapt to manage load balance and failures. This context corresponds to that of Chapter III. In this context, nodes are static and the network is reliable and fast. Again, we can safely employ FIFO reliable multicast without degrading the overall performance. In addition, for clock synchronization we can rely on network time protocols. Again, with respect to consistency management, we expressed total order requirements in terms of group communication primitives. In this context, our solution for managing replica consistency is to delay the execution of all update and refresh transactions (for partial replication) submitted to the cluster, until its deliver time at a specific node. The optimizations we proposed for preventive replication on submitting updated transactions optimistically yield excellent gains that almost compensate the delay time. In most cases, it eliminates the delay time. In this context, the use of total order primitives could be an alternative solution to multicast transactions towards all nodes in the cluster. However, the network performance could be degraded in *bursty* workloads, due to the overhead of message exchange, necessary to implement the total order service. In a large scale lazy multi-master replication architecture, the load balancing of transactions may improve the overall performance by choosing the best node to execute a transaction or query [GNPV02].



Replication Context	Configurations	Scalelability	Consistency	Heterogeneity	Dynamicity	Load Balancing	Network Requirements
DW Fast Refresh Algorithms	Lazy Master	Low	Transaction Based Total order on Refresh Transactions Improve Data Freshness	DB Replication component are external	No, however, tolerates crashes	No Point-to-Point	Reliable Fifo Multicast NTP Bounds of all Participants
Cluster DB Preventive Replication	Lazy Master Multi-Master (total and partial replication)	Large	Transactions based Total order on Update and Refresh transactions	DB Replication components are external	No, however tolerates crashes	Yes, to chose the best node to execute queries	Reliable Fifo Multicast NTP Bounds of all Participants (Bound may be relaxed)
P2P UMS-KTS P2P-LTR P2P-Reconciler	Lazy Master for peer dynamicity and crashes Multi-Master in DHT and hybrid overlays for collaboration	Large and very large thanks to DHT use of cost models and Topology aware	Scheduled Based Eventual Consistency Data currency for consistent retrieval	Network Peers handled by P2P middleware (Data Objects are indexed in the DHT)	Yes Protocols are adaptable Tolerates crashes (sucessors replaces)	Yes, controlled by the P2P system routing algos. Location Aware improves reconciliation performance	Reliable Detectors

Table 6.1: Lazy Replication Parameters

Node fault tolerance is easy to implement [PS00] in the replication solutions presented in Chapters 2 and 3 because the algorithms rely on logs to keep track of propagated and received messages. Thus, in case of failures, the system is not blocked because there is no voting scheme and, upon recovery, the logs are used to resynchronize the recovering node.

P2P applications make data management much more difficult. Peers can leave and join the system dynamically. As the scale of P2P system gets very large, the probability of failures increases. P2P data replication must also be more general and we do not focus specifically on relational tables and transaction consistency requirements, as in preventive replication. Instead, we focus on P2P collaborative applications with some semantics, and general data objects. However, to enable the deployment of such applications in P2P networks, it is required a mechanism to deal with their high data sharing in dynamic, scalable and available way. Previous work on optimistic replication has mainly concentrated on centralized systems. Centralized approaches are inappropriate for a P2P setting due to their limited availability and vulnerability to failures and partitions from the network. We focused on the design of a reconciliation algorithm designed to be deployed in large scale cooperative applications, such as P2P Wiki. Our main contribution was a distributed reconciliation algorithm designed for P2P networks (P2P-reconciler). Other important contributions are: a basic cost model for computing communication costs in a DHT overlay network; a strategy for computing the cost of each reconciliation step taking into account the cost model; and an algorithm that dynamically selects the best nodes for each reconciliation step. Furthermore, since P2P networks are built independently of the underlying topology, which may cause high latencies and large overheads degrading performance, we also propose a topology-aware variant of our P2P-reconciler algorithm and show the important gains on using it. Our P2P-reconciler solution enables high levels of concurrency thanks to semantic reconciliation and yields high availability, excellent scalability, with acceptable performance and limited overhead. In addition, our topology aware approach improves the basic P2P reconciler by 50%, which is a very impressive result. This last result encouraged us to use topology awareness in current research work. This context corresponds to that of Chapter 4.

Distributed Hash Tables (DHTs) provide a scalable solution for data sharing in P2P systems. To ensure high data availability, DHTs typically rely on data replication, yet without data currency guarantees. Supporting data currency in replicated DHTs is difficult as it requires the ability to return a current replica despite peers leaving the network or concurrent updates. We give a complete solution to this problem. We propose an Update Management Service (UMS) to deal with data availability and efficient retrieval of current replicas based on timestamping. For generating timestamps, we propose a Key-based Timestamping Service (KTS) which performs distributed timestamp generation using local counters. Through probabilistic analysis, we compute the expected number of replicas which UMS must retrieve for finding a current replica. Except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of available replicas are current then the expected number of retrieved replicas is less than 3. This context corresponds to that of Chapter 5.

We chose to develop our replication solutions of chapter 4 and 5 on top of DHT networks which enable to index data objects through a *key* value that may have some semantic significance. Using a DHT, data is uniformly distributed over the network, which is a good basis for scaling up and load balancing. In addition, fault-tolerance relies on the DHT fault tolerance. In our solutions, we assumed that the lookup service of the DHT does not behave incorrectly, that is, given a key *k*, it either finds correctly the responsible for *k* or reports an error, *e.g.* in the cases of network partitioning where the responsible is not reachable. To support this last assumption, network must be reliable and bounds must be approximatively known in order to have a reliable DHT stabilization algorithm. Finally, we also considered the use of some kind of detector [LCGS05]. These assumptions are quite acceptable in P2P dynamic networks [LCGS05]. To guarantee 100% of consistency, we would need to implement complex fault tolerance protocols [OV99] based on quorums, or some specific group communication service on the DHT overlay level. However, these solutions are known not to scale up well, and do not behave well in dynamic environments. This lessons was learned in the context of Chapters 4 and 5.

6.2 Current and Future Work

Based on our previous work we just described and faced with some new challenges, we now present our perspectives in current and future work.

6.2.1 P2P Logging and Timestamping for Reconciliation (P2P-LTR)

As part of our current and future work, we consider the use of KTS (see Chapter 5) as a distributed P2P timestamper for general collaborative applications. For instance, consider several Xwiki users who update documents in parallel in a multi-master replication approach. The reconciliation of these updates needs to establish a total order on the updates to be applied at each master peer to provide eventual consistency. Alternatively to semantic reconciliation, we adopt a text editing reconciliation algorithm based on operational transforms such as So6 [MOSI03, SCF98]. For the reconciliation engine, one solution to provide total order is to rely on a centralized timestamper node that also stores the log of reconciled updates. However, the timestamper node may be a single point of failure and performance bottleneck. In this context, we now address three new requirements:

- Continuous P2P Timestamping; such that difference between the timestamps of any two consecutive updates be one;
- P2P-Log service, necessary to store the timestamped updates (log) in the DHT;
- Retrieval service to safely find the continuous timestamps updates stored in P2P-Log, used later on to perform document reconciliation.

We propose P2P-LTR service that addresses these new requirements. Concerning the Timestamps must be continuous (the difference between the timestamps of any two consecutive updates is one) instead of in monotonically increasing order. This allows Xwiki users to safely *get* the *next* missing patches of updates in the DHT (stored at log peers). So given the last timestamp value on a document *d*, seen locally, the master asks the timestamper *responsible peer* of *d,* to provide the last timestamp it gave to any other multi-master peer of *d* . If this value is greater than the master local timestamp value, then the master of *d* gets all missing timestamped updates stored previously in the DHT. Thus the DHT acts as a highly available distributed log. Hence updates performed by masters are indexed using the document key and its timestamp value. We are currently working on this research topic, addressing some fault-tolerance issues and developing a prototype of P2P-LTR.

To measure the reliability of P2P-LTR we plan to implement and run exhaustively our solution over the PlanetLab platform [Pla]. PlanetLab makes it easy to deploy the DHT we use (OpenChord) [OCH] as it lets Java RMI call cross firewalls. We expect that in practice eventual consistency may be violated rarely. However, we also accept that for the type of applications we consider a low level of inconsistency is acceptable (missing updates, out of order updates) and that the users can manage it. We plan to investigate the management of quality of data in such contexts.

6.2.2 P2P Caching

Motivated by the cost models proposed in our previous work on the P2P-reconciler, we currently focus on reducing P2P network traffic by combining different techniques: search strategies, caching placement and location awareness for cache management in unstructured and structured systems.

Index Caching

Despite the emergence of sophisticated overlay structures, unstructured P2P systems remain highly popular and widely deployed. They exhibit many simple yet attractive features, such as low-cost maintenance and high flexibility in data placement and node neighborhood. Unstructured P2P systems are heavily used for file-sharing communities due to their capacity of handling keyword queries i.e. instead of lookup on entire filenames. However, some studies [SGDGL02] observed that the P2P traffic is mainly composed of query messages and contributes the largest portion of the Internet traffic.

The principal cause of the heavy P2P traffic is the inefficient search mechanism, blind flooding, which is commonly employed in unstructured P2P networks. Many researchers have focused on this critical issue that may compromise the benefits of such systems by drastically limiting their scalability. In fact, inefficient searches cause unnecessary messages that overload the network, while missing requested files. Several analyses [P02, LBBS02, Scrip] found the P2P file-sharing traffic highly repetitive because of the temporal locality of queries. They actually observed that most queries request a few popular files and advocated the potential of caching query responses, to efficiently answer queries without flooding over the entire network (a query response holds information about the location of the requested file).

However, searches in general, suffer significantly from the dynamic nature of P2P systems where nodes are run by users with high autonomy and low availability. In fact, it is rather impossible to ensure the availability of a single file copy and thereby to satisfy queries for this file. In P2P file sharing, a node that requested and downloaded a file, can provide its copy for subsequent queries. As a consequence, popular files, which are frequently requested, become naturally well-replicated [CRBL+03]. Hence, search techniques should leverage the natural file replication to efficiently find results with minimum overhead.

Another important factor that worsens the traffic issue consists in constructing P2P overlays without any knowledge of the underlying topology. A typical case that illustrates this problem is the following: a query can be directed to a copy of the desired file which is hosted by a physically distant node, while other copies may be available at closer nodes. Hence, file downloads can consume a significant amount of bandwidth and thereby overload the network. In addition, the user experience dramatically degrades due to the relatively high latency perceived during transfer.

Our proposal to reduce P2P network traffic is based on DiCAS [WXLZ06], an index caching and adaptive search algorithm. In DiCAS, query responses are cached, in the form of file indexes, in specific groups of nodes based on a specific hashing of the filenames. Guided by the predefined hashing, queries are then routed towards nodes which are likely to have the desired indexes. However, DiCAS is not optimized for keyword searches which are the most common in the context of P2P file sharing. Moreover, caching a single index per file does not solve the problem of file availability given the dynamic nature of P2P systems, while it may overload some nodes located by previous queries. DiCAS also lacks topological information to efficiently direct queries to close nodes.

Aiming at reducing the P2P redundant traffic and addressing the limitations of existing solutions, we propose a solution [EPV07] that leverages the natural file replication and incorporates topological information in terms of file physical distribution, when answering queries. To support keyword searches, a Bloom filter is used to express keywords of filenames cached at each node, and is then propagated to neighbors. A node routes a query by querying its neighbors' Bloom filters. To deal with issues concerning availability and workload, a node caches several indexes per file along with topological information. As a consequence, a node answers a query by providing several possibilities, which significantly improves the probability of finding an available file. In addition, based on the topological information, we expect that queries are satisfied in a way that optimizes the file transfer and thus the bandwidth consumption. Our simulations shows that the cache hit and download distance is improved by using our approach with acceptable overhead.

P2P Web Caching

In the next step of our research in P2P caching, we propose to exploit a DHT to share content storage [RCFB07]. Much research work has been on addressed the search efficiency issue, including the designs of Chord, Tapestry, Can, etc. In these approaches, each peer and its stored contents are organized using a DHT. For a system with *N* nodes, the search cost (i.e. number of lookup hops) is bounded by *O(LogN)*. For the current search solutions in the structured P2P, all peers are assumed to submit queries to uniformly search the contents stored in all peers. However, this assumption is not valid in practice. Often, the popularities of the contents are skewed. For example, web requests on the Internet space are highly skewed with a Zipf-like distribution. Therefore, this skewed popularity causes unbalanced workload. In addition, due to the limited bandwidth, even though the search cost to some hot peers is still bounded by *O(logN)* hops, there can be a long latency for getting the data.

One solution to improve the workload balance is to replicate these popular contents in the best nodes which requires to address two key problems related to caching the replicas in structured P2P systems: where and how to cache the replicas of popular contents.

6.2.3 P2P Data Streaming

Recent years have witnessed major research interest in data stream management systems. A data stream is a continuous and unbounded sequence of data items. There are many applications that generate streams of data including financial applications, network monitoring, telecommunication data management, sensor networks, etc. Processing a query over a data stream involves running the query continuously over the data stream and generating a new answer each time a new data item arrives. Due to the unbounded nature of data streams, it is not possible to store the data entirely in a bounded memory. This makes difficult the processing of queries that need to compare each new arriving data with past ones. We are interested in systems which have limited main memory but that can tolerate an approximate query result which has a maximum subset of the result.

A common solution to the problem of processing join queries over data streams is to execute the query over a sliding window [GO03] that maintains a restricted number of recent data items. This allows queries to be executed in a finite memory and in an incremental manner by generating new answers when a new data item arrives.

In current research, we address the problem of computing approximate answers to windowed stream joins over data streams. Our solution [PAPV08] involves a scalable distributed sliding window that takes advantage of the free computing power of DHT networks and can be equivalent to thousands of centralized sliding windows. Then, we propose a mechanism called DHTJoin, which deals with efficient execution of join queries over all data items which are stored in the distributed sliding window. DHTJoin combines hash-based placement of tuples in the DHT and dissemination of queries using a gossip style protocol.

Bibliography

[AA95] G. Alonso, A. Abbadi. Partioned Data Objects in Distributed Databases, *Distributed and Parallel Databases*, 3(1):5-35, 1995.

[ABKW98] T. Anderson, Y. Breitbart, H. Korth, A. Wool: Replication, Consistency, and Practicality: Are These Mutually Exclusive? In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 484-495, Seattle, Washington, June, 1998.

[ABCM+03] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML documents with distribution and replication. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 527-538, San Diego, California, June 2003.

[ACDD0+3] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *ACM SIGMOD Record*, 32(3):29-33, September 2003.

[ABG90] R. Alonso, D.Barbara, H. Garcia-Molina. Data Caching Issues in an Information Retrieval System, *ACM Transactions on Database Systems*, 15(3):359-384,, September 1990.

[AD76] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proc. of the Int. Conf. on Software Engineering*, pages 562-570, San Francisco, California, October 1976.

[AHA03] D. Anwitaman, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 76-85, Washington, District of Columbia, May 2003.

[AL01] P. Albitz and C. Liu. *DNS and BIND*. 4th Ed., O'Reilly, January 2001.

[AM06] R. Akbarinia and V. Martins. Data management in the APPA P2P system. In *Proc. of the Int. Workshop on High-Performance Data Management in Grid Environments (HPDGrid)*, Rio de Janeiro, Brazil, July 2006.

[AM07] R. Akbarinia and V. Martins. Data management in the APPA system. *Journal of Grid Computing*, pages 303-317, 2007.

[AMPV04] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Replication and query processing in the APPA data management system. In *Proc. of the Int. Workshop on Distributed Data and Structures (WDAS)*, Lausanne, Switzerland, July 2004.

[AMPV06a] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. *Global Data Management* (Chapter Design and implementation of Atlas P2P architecture). 1st Ed., IOS Press, July 2006.

[AMPV06b] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Top-k query processing in the APPA P2P system. In *Proc. of the Int. Conf. on High Performance Computing for Computational Science (VecPar)*, pages 158-171, Rio de Janeiro, Brazil, July 2006.

[APV06] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured P2P systems using top-k queries. *Distributed and Parallel Databases,* 19(2-3):67-86, May 2006.

[APV07a] R. Akbarinia, E. Pacitti, P. Valduriez. Data Currency in Replicated DHTs. *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD),* pages 211-222, Pékin, Chine, June 2007.

[APV07b] R. Akbarinia, E. Pacitti and P. Valduriez. Processing top-k queries in distributed hash tables. *European Conf. on Parallel Computing (Euro-Par)*, 489-502, 2007.

[APV07c] R. Akbarinia, E. Pacitti and P. Valduriez. Best position algorithms for top-k queries. *Int. Conf. on Very Large Databases (VLDB)*, 495-506, 2007.

[AT02] Y. Amir, C. Tutu. From Total Order to Database Replication. *In Proc. ICDCS*, pages 494-, July, Vienna, Austria, 2002.

[ATS+05] F. Akal, C. Turker, H-J. Schek, Y. Breibart, T. Grabs, L. Veen. Fine-grained Replication and Scheduling with Freshness and Correctness Guarantees. *In Int. Conf. on Very Large Data Bases (VLDB)*, pages 565-576, 2005.

[BFGR06] P.A. Bernstein, A. Fekete, H. Guo , R. Ramakrishnan, P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD*), pages 599-610*,* Chicago, USA, June, 2006.

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

[BGM04] M. Bawa, A. Gionis, H. Garcia-Molina, R. Motwani. The Price of Validity in Dynamic Netwoks. In *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD*), pages 515-526*,* Paris, France, June, 2004.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. 1st Ed., Addison-Wesley, February 1987.

[BKRS+99] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databases. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 97-108, Philadelphia, Pennsylvania, June 1999.

[Bri06] BRITE. http://www.cs.bu.edu/brite/.

[CGPV04] C. Coulon, G. Gaumer, E. Pacitti, P. Valduriez: The RepDB* prototype: Preventive Replication in a Database Cluster, *Base de Données Avancées*, pages 333-337, Montpellier, France, 2004.

[CH06] Y.L. Chong and Y. Hamadi. Distributed log-based reconciliation. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, pages 108-112, Riva del Garda, Italy, September 2006.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427-469, December 2001.

[CMHS+02] I. Clarke, S. Miller, T.W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40-49, January 2002.

[CPV05a] C. Coulon, E. Pacitti, P. Valduriez: Scaling up the Preventive Replication of Autonomous Databases in Cluster Systems, *Int. Conf. on High Performance Computing for Computational Science (VecPar 2004)*, Valencia, Spain, 2004, Lecture Notes in Computer Science 3402, Springer 2005, pages 174-188.

[CPV05b] C. Coulon, E. Pacitti, P. Valduriez: Consistency Management for Partial Replication in a High Performance Database Cluster, *IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS 2005)*, Fukuoka, Japan, 2005.

[COD95] E. Codd. *Twelve rules for on-line analytical processing*. Computer world, April, 1995.

[CRBL+03] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P systems scalable. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407-418, Karlsruhe, Germany, August 2003.

[CRR96] P. Chundi, D.J. Rosenkrantz, and S.S. Ravi. Deferred updates and data placement in distributed databases. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 469-476, New Orleans, Louisiana, February 1996.

[CT96] T.D. Chandra, S. Toueg. Unreliable failure detectors for asynchronous systems. *In Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 325-340, Montreal, Canada, August, 1991.

[Dav94] J. Davis. Data Replication, *Distributed Computing Monitor*, 1994

[DGY03] N. Daswani, H. Garcia-Molina, B. Yang. Open problems in data-sharing peer-to-peer systems. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, pages 1-15, Siena, Italy, January 2003.

[DHA03] A. Datta, M. Hauswirth, K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proc. Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 76-, Providence, RI, USA, May, 2003.

[DZDKS03] F. Dabek , B. Y. Zhao, P. Druschel , J. Kubiatowicz , I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. Proc. of Int. Workshop on Peer-to-Peer Systems (IPTPS), 2003.

[EMP07] M. El-Dick, V. Martins, E. Pacitti, A Topology-Aware Approach for Distributed Data Reconciliation in P2P Networks, In *Proc. of the European Conf. on Parallel Computing (Euro-Par),* pages 318-327, Rennes, France, August, 2007.

[EPV07] M. El-Dick, E. Pacitti, P. Valduriez. Location-Aware Index Caching and Searching for P2P Systems. *In VLDB Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2,)*, Vienna, Austria, 2007, selected papers LNCS Springer, à paraître.

[ET89] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264-290, June 1989.

[FI03] I.T. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. *In Proc. of the Int. Workshop on P2P Systems (IPTPS)*, pages 118-128, Berkeley, California, February 2003.

[Fid88] C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of the Australian Computer Science Conference*, pages 55-66, University of Queensland, Australia, February 1988.

[Fip95] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, April 1995.

[FKT01] I.T. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: enabling scalable virtual organizations. *Journal of High Performance Computing Applications*, 15(3):200-222, Fall, 2001.

[FVC04] J. Ferrié, N. Vidot, M. Cart. Concurrent undo operations in collaborative environments using operational transformation. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 155-173, Agia Napa, Cyprus, October 2004.

[Gen06] Genome@Home. http://genomeathome.stanford.edu/.

[GHOS96] J. Gray, P. Helland, P.E. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 173-182, Montreal, Canada, June 1996.

[Gif79] D.K. Gifford. Weighted voting for replicated data. In *Proc. of the ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 150-162, Pacific Grove, California, December 1979.

[GLR05] H. Guo, P.A. Larson, R. Ramakrishnan. Caching with 'Good Enough' Currency, Consistency, and Completeness. *In Int. Conf. on Very Large Data Bases (VLDB)*, pages 457-468, Trondheim, Norway, August , 2005.

[GLRG04] Guo, H., Larson, P.Å., Ramakrishnan, R., and Goldstein, J. Relaxed Currency a Consistency: How to Say "Good Enough in SQL". In *ACM SIGMOD Int. Conf. on Management of Data (SIGMOD*), pages 815-826, Paris, France, June, 2004.

[GM98] L. George, P. Minet. A Uniform Reliable Multicast Protocol with Guaranteed Response Times, *In Int.Proc. ACM SIGPLAN Workshop on Languages Compilers and Tools for Embedded Systems*, Montreal, pages 65-82, June, 1998.

[Gnu06] Gnutella. http://www.gnutelliums.com/.

[GO03] Golab L., Özsu T. Processing Sliding Windows Multi-Joins in Continuous Queries over Data Streams. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 500-511, Berlin, Germany, September, 2003.

[Gol92] R.A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, California, December 1992.

[Gri06] Grid5000 Project. http://www.grid5000.fr.

[GNPV02] S. Gançarski, H. Naacke, E. Pacitti, P. Valduriez: Parallel Processing with Autonomous Databases in a Cluster System, *In Proc. Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 410-428, Irvine, California, October, 2002.

[GNPV07] S. Gançarski, H. Naacke, E. Pacitti P. Valduriez. The leganet System: Freshness-Aware Transaction Routing in a Database Cluster. *Information Systems*, Vol. 32, N° 2, 2007, 320-343.

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

[GSC+83] N. Goodman, D. Skeen, A. Chan, U. Dayal, S. Fox, and D.R. Ries. A recovery algorithm for a distributed database system. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 8-15, Atlanta, Georgia, March 1983.

[Har06] Harmony. http://www.seas.upenn.edu/~harmony/.

[HHHL+02] M. Harren, J.M. Hellerstein, R. Huebsch, B.T. Loo, S. Shenker and I. Stoica. Complex queries in DHT-based peer-to-peer networks. *Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 242-259, 2002.

[HIMT03] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proc. of the Int. World Wide Web Conference (WWW)*, pages 556-567, Budapest, Hungary, May 2003.

[HM98] F. Howell and R. McNab. SimJava: a discrete event simulation package for Java with applications in computer systems modeling. In *Proc. of the Int. Conf. on Web-based Modeling and Simulation*, San Diego, California, January 1998.

[HT93] V. Hadzilacos, S. Toueg: *Fault-Tolerant Broadcasts and Related Problems*. Distributed Systems, 2nd Edition, S. Mullender (ed.), Addison-Wesley, 1993.

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

[JAB01] M. Jovanovic, F. Annexstein, and K. Berman. Scalability issues in large peer-to-peer networks: a case study of Gnutella. Technical report, ECECS Department, University of Cincinnati, Cincinnati, Ohio, January 2001.

[JM87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 227-238, San Francisco, California, May 1987.

[Jov00] M. Jovanovic. *Modelling large-scale peer-to-peer networks and a case study of Gnutella*. Master's thesis, Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio, June 2000.

[JPKA02] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, G. Alonso: Improving the Scalability of Fault-Tolerant Database Clusters: Early Results. *In Proc. Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 477-484, Vienna, Austria, July, 2002.

[JPAK03] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257-294, September, 2003.

[JRMI] Java RMI. http://www.sun.com.

[Jxt06] JXTA. http://www.jxta.org/.

[KA00a] B. Kemme, G. Alonso. Don't Be Lazy, Be Consistent : Postgres-R, A New Way to Implement Database Replication. *In Proc. Int. Conf. on VLDB*, pages 134-143, Cairo, Egypt, September, 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333-379, September, 2000.

[Kaz06] Kazaa. http://www.kazaa.com/.

[KB91] N. Krishnakumar, J. Bernstein. Bounded Ignorance in Replicated Systems. *In Proc. Of the ACM Symp. On Principles of Database Systems (PODS)*, pages 63-74, Denver, Colorado, May, 1991.

[KB94] N. Krishnakumar, J. Bernstein. Bounded Ignorance: A Technique for Increasing Concurrency in a Replicated System. *ACM Transactions on Database Systems (TOD)*, 19(4):586-625, 1994.

[KBCC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proc. of the ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190-201, Cambridge, Massachusetts, November 2000.

[KBHO+88] L. Kawell Jr., S. Beckhart, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 205-216, Portland, Oregon, September 1988.

[Kemm00] B. Kemme. *Data Replication for Clusters of Workstation*. PhD Dissertation. Swiss Federal Institute of Technology Zurich, 2000.

[KKMN98] D.G. Kleinbaum, L.L. Kupper, K.E. Muller, and A. Nizam. *Applied Regression Analysis and Multi-variable Methods*. 3rd Ed., Duxbury Press, January 1998.

[KLLL+97] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the ACM Symp. on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[KRSD01] A-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 210-218, Newport, Rhode Island, August 2001.

[KT96] M. F. Kaashoek, A. S. Tanenbaum. An Evaluation of the Amoeba group communication system. *In Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 436-448, Hong Kong, May, 1996.

[KWR05] P. Knezevic, A. Wombacher, and T. Risse. Enabling high data availability in a DHT. In *Proc. of the Int. Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05)*, pages 363-367, Copenhagen, Denmark, August 2005.

[Lad90] R. Ladin. Lazy Replication: Exploiting the semantics of distributed services. *In Proc. of the Int. Workshop on Management of Replicated Data*, Houston, November, 1990.

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

[LCCL+02] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. *In Proc. of the ACM Int. Conf. on Supercomputing (ICS)*, pages 84-95, New York, New York, June 2002.

[LCGS05] P.Linga, A. Crainiceaunu, J. Gehrke, J. Shanmugasudaram. Guaranteeing Correctness and Availability in P2P Range Indices*, In Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 323-334, Baltimore, Maryland, USA, 2005.

[LKPJ05] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Consistent data replication: is it feasible in WANs? In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 633-643, Lisbon, Portugal, September 2005.

[LKPJ07] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Enhancing Edge Computing with Database Replication. In IEEE *Symp. on Reliable Distributed Systems (SRDS),* Beijing, China, September, 2007.

[MAPV06] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P system. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 401-410, Minneapolis, Minnesota, July 2006.

[MPEJ08] V. Martins, E. Pacitti, M. El-Dick, R. Jiménez-Peris. Scalable and Topology-Aware Reconciliation on P2P Networks, accepted with minor revisions, 2008.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proc. of the Int. Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B.V., pages 216-226, North-Holland, October 1989.

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

[Mil06] D.L. Mills. *Computer Network Time Synchronization: the Network Time Protocol*, CRC Press, 2006, 304 pages.

[MOSI03] P. Molli, G. Oster, H. Skaf-Molli, A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proc. of the ACM SIGGROUP Int. Conf. on Supporting Group Work (GROUP)*, pages 212-220, Sanibel Island, Florida, November 2003.

[MMS+96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. Communication of the ACM, 39(4):54-63, 1996.

[MP06] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in P2P-DHT networks. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 337-349, Dresden, Germany, September 2006.

[MPJV06] V. Martins, E. Pacitti, R. Jimenez-Peris, and P. Valduriez. Scalable and available reconciliation in P2P networks. In *Proc. of the Journées Bases de Données Avancées (BDA)*, Lille, France, October 2006.

[MPV05] V. Martins, E. Pacitti, and P. Valduriez. Distributed semantic reconciliation of replicated data. *IEEE France and ACM SIGOPS France - Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, Paris, France, November 2005.

[MPV06a] V. Martins, E. Pacitti, and P. Valduriez. A dynamic distributed algorithm for semantic reconciliation. In *Proc. of the Int. Workshop on Distributed Data & Structures (WDAS)*, Santa Clara, California, January 2006.

[MPV06b] V. Martins, E. Pacitti, and P. Valduriez. Survey of data replication in P2P systems. Technical Report 6083, INRIA, Rennes, France, December 2006.

[Nap06] Napster. http://www.napster.com/.

[OCH] Open Chord. http://open-chord.sourceforge.net.

[OGSA06a] Open Grid Services Architecture. http://www.globus.org/ogsa/.

[OGSA06b] Open Grid Services Architecture Data Access and Integration. http://www.ogsadai.org.uk/.

[Ora95] Oracle Corporation. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*. White paper, Oracle Corporation, 1995.

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

[OST03] B. Ooi, Y. Shu, and K-L. Tan. Relational data sharing in peer-based data management systems. *ACM SIGMOD Record*, 32(3):59-64, September 2003.

[OV99] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. 2nd Ed., Prentice Hall, January 1999.

[P02] E. P. Markatos. Tracing a large-scale P2P System: an hour in the life of Gnutella. *Proc. of CCGRID*, 2002.

[PA04] C. Plattner, G. Alonso: Ganymed: Scalable Replication for Transactional Web Applications, *In Proc. of the 5th International Middleware Conference*, pages 155-174 Toronto, Canada, october 2004.

[PA99] E. Pacitti. *Improving Data Freshness in Replicated Databases.* Thesis and Research Report No 3617, INRIA, 114 pages, 1999.

[Pal02] PalmSource. Introduction to conduit development. Available at http://www.palmos.com/ dev/support/docs/.

[PAPV08] W. Palma, R. Akbarinia, E. Pacitti, P. Valduriez, Efficient Processing of Continuous Join Queries using Distributed Hash Tables, submitted paper, 2008.

[PC98] C. Palmer, G. Cormack. Operation transforms for a distributed shared spreadsheet. *In Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW*), pages 69-78, Seattle, Washington, November 1998.

[PCVO05] E. Pacitti, C. Coulon, P. Valduriez, T. Özsu . Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, Vol. 18, No. 3, 2005, 223-251.

[PGS96] F. Pedone, R. Guerraoui, A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. *Proc. of the European Conf. on Parallel Computing (Euro-Par),* pages 513-520, Southampton, UK, september, 1998.

[Pla] http://www.planet-lab.org/node

[PL91] C. Pu, A. Leff. Replica Control in Distributed Systems : An asynchronous approach. *In Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 377-386, Denver, Colorado, May, 1991.

[POC03] E. Pacitti, T. Özsu, C. Coulon: Preventive Multi-Master Replication in a Cluster of Autonomous Databases. *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, 318-327, Klagenfurt, Austria, august, 2003.

[PL88] J.F. Pâris and D.E. Long. Efficient dynamic voting algorithms. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 268-275, Los Angeles, California, February, 1988.

[PJKA00] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, G. Alonso. Scalable Replication in Database Clusters. *In Int. Symposium on Distributed Computing (DISC)*, pages 315-329, Toledo, Spain, October, 2000.

[PJKA05] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, G. Alonso. Middle-R : Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375-423, 2005.

[Pos05] PostgresSQL Global Development Group. PostgresSQL 8.1, http.//www.postgressql.org, 2005.

[PMS99] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 126-137, Edinburgh, Scotland, September 1999.

[PMS01] E. Pacitti, P. Minet, E. Simon. Replica Consistency in Lazy Master Replicated Databases. *Distributed and Parallel Databases*, Vol . 9, No. 3, 2001, 237-267.

[PRR97] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures* (*SPAA)*, pages 311-320, Newport, Road Island, June 1997.

[PS00] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *VLDB Journal*, 8(3-4):305-318, 2000.

[PS98] F. Pedonne, A Schiper: Optimistic Atomic Broadcast. *In Int. Symposium on Distributed Computing (DISC)*, pages 318-332, Andros, Greece, september,1998.

[PSG04] B.C. Pierce, A. Schmitt, and M.B. Greenwald. *Bringing Harmony to optimism: an experiment in synchronizing heterogeneous tree-structured data*. Technical report MS-CIS-03-42, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, February 2004.

[PSJ04] S. PalChaudhuri, A.K. Saha and D.B. Johnson. Adaptive clock synchronization in sensor networks. *Int. Symp. on Information Processing in Sensor Networks (IPSN)*, pages 340-348, Berkeley, California, April, 2004.

[PSM03] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 38-55, Catania, Italy, November 2003.

[PSM98] E. Pacitti, E. Simon, R.N. Melo. Improving data freshness in lazy master schemes. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 164-171, Amsterdam, The Netherlands, May 1998.

[PSTT+97] K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 288-301, St. Malo, France, October 1997.

[PV04] B.C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania, February 2004.

[PV98] E. Pacitti, P. Valduriez. Replicated Databases: concepts, architectures and techniques. *Network and Information Systems Journal*, Vol. 1, No. 3, 1998, 519-546

[RA93] F. Raab: TPC-C - The Standard Benchmark for Online transaction Processing (OLTP). The Benchmark Handbook for Database and Transaction Systems (2nd Edition), Morgan Kaufmann, 1993.

[RBSS02] Röhm, U., Böhm, K., Schek, H., and Schuldt,H. FAS: a Freshness- Sensitive Coordination Middleware for a Cluster of OLAP Components. In *Proc. of the Int. Conf. on Very Large Data Bases (VLDB)*, pages 754-765, Hong Kong, China, August, 2002.

[RC01] N. Ramsey, E. Csirmaz. An algebraic approach to file synchronization. In *Proc. of the ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (FSE)*, pages 175-185, Vienna, Austria, September 2001.

[RCFB07] W. Rao, L. Chen, A. Fu, Y. Bu. Optimal Proactive Caching in Peer-to-Peer Network Analysis and Application, *In Proc. Int. Conf. on Information and Knowledge Management (CIKM)*, pages 633-672, Lisbon, Portugal, november 2007.

[RD01a] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329-350, Heidelberg, Germany, November 2001.

[RepDB] RepDB*: Data Management Component for Replicating Autonomous Databases in a Cluster System (released as open source software under GPL). http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 188-201, Banff, Canada, October 2001.

[RFHK$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technolo-*

gies, Architectures, and Protocols for Computer Communications, pages: 161-172, San Diego, California, August 2001.

[RGK96] M. Rabinovich, N.H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 207-222, Avignon, France, March 1996.

[RHKS01] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. of the Int. Workshop on Networked Group Communication (NGC)*, pages 14-29, London, United Kingdom, November 2001.

[SAS+96] J. Sidell, P.M. Aoki, A. Sah, C. Staelin, M. Stonebraker, A. Yu. Data Replication in Mariposa. *In Proc. of the Int. Conf. on Data Engineering (ICDE)*, pages 485-494, New Orleans, Louisiana, February, 1996.

[SBK04] M. Shapiro, K. Bhargavan, N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. of the Int. Conf. on Principles of Distributed Systems (OPODIS)*, pages 331-345, Grenoble, France, December 2004.

[Scrip] Sripanidkulchai K.:The popularity of Gnutella queries and its implication on scaling. http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html.

[SCF98] M. Suleiman, M. Cart, J. Ferrié. Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *Proc. IEEE Int. Conf. on Data Engineering (ICDE)*, pages 36-45, Orlando, USA, February, 1998.

[SE98] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 59-68, Seattle, Washington, November 1998.

[Set06] Seti@home. http://setiathome.ssl.berkeley.edu.

[SGDGL02] S. Saroiu, P. K. Gummadi, R. J. Dunn, S. D. Gribble, H. Levy. An analysis of Internet content delivery systems. In *Proc.5th International Symposium on Operating System Design and Implementation (OSDI)*, Boston, Massachusetts, Decemebr, 2002.

[SJZY+98] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63-108, March 1998.

[SM02] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of the Int. Workshop on P2P Systems (IPTPS)*, pages 261-269, Cambridge, Massachusetts, March 2002.

[SMKK+01] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149-160, San Diego, California, August 2001.

[SPOM01] A. Sousa, F. Pedone, R. Oliveira, F Moura: Partial Replication in the Database State Machine. *IEEE Int. Symposium on Network Computing and Applications (NCA)*, 2001.

[SPMO02] A. Sousa, J. Pereira, F. Moura, R. Oliveira. Optimistic total order in wide area networks. *Proc. 21st ieee symposium on reliable distributed systems*, pages 190-199. 2002.

[SOTZ03] W. Siong Ng, B. Ooi, K-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, March 2003.

[SR90] A. Sheth, M. Rusinkiewicz. Management of Interdependent Data: Specifying Dependency and Consistency Requirements, *In Proc. Int. Workshop on Management of Replicated Data*, Houston, November, 1990.

[SR96] A. Schiper, M. Raynal. From group communication to transactions in distributed systems. *Communication of the ACM, 39(4):84-87*, April, 1996.

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

[Sta95] D. Stacey. Replication: Db2, Oracle or Sybase ?. In ACM Sigmod, 24(4), pages 95-101, 1995.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, 5(3):188-194, May 1979.

[SWBC+97] W. Sullivan III, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A new major SETI project based on project Serendip data and 100,000 personal computers. In *Proc. of the Int. Conf. on Bioastronomy*, Bologna, Italy, 1997.

[SYZC96] C. Sun, Y. Yang, Y. Zhang, and D. Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Proc. of the Australian Computer Science Conference*, pages 582-591, Melbourne, Australia, January 1996.

[TC94] K. Tindel, J. Clark. Holistic Schedulability Analysis for distributed hard real-time systems. *Microproces, Microprogram*, 40(2-3):117-134,1994.

[The04] N. Théodoloz. *DHT-based routing and discovery in JXTA*. Master Thesis, École Polytechnique Fédérale de Lausanne, 2004.

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

[TIMH+03] I. Tatarinov, Z.G. Ives, J. Madhavan, A. Halevy, D. Suciu, N. Dalvi, X. Dong, Y. Kadiyska, G. Miklau, and P. Mork. The Piazza peer data management project. *ACM SIGMOD Record*, 32(3):47-52, September 2003.

[TTPD+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 172-183, Cooper Mountain, Colorado, December 1995.

[Uni06] Unison. http://www.cis.upenn.edu/~bcpierce/unison/.

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

[VCFS00] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 171-180, Philadelphia, Pennsylvania, December 2000.

[Ves03] J. Vesperman. *Essential CVS*. 1st Ed., O'Reilly, June 2003.

[Wik07] Wikipedia. http://wikipedia.org/.

[WIO97] S. Whittaker, E. Issacs, and V. O'Day. Widening the net: workshop report on the theory and practice of physical and network communities. *ACM SIGCHI Bulletin*, 29(3):27-30, July 1997.

[WK05] S. Wu, B. Kemme: Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. IEEE Int. Conference on Data Engineering (ICDE), 2005.

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

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

[WXLZ06] Wang, C., Xiao, L., Liu, Y., Zheng, P..DiCAS: an efficient distributed caching mechanism for P2P systems. *IEEE Trans. Parallel Distrib. Syst.*, 2006.

[XCK06] Y. Xia, S. Chen, and V. Korgaonkar. Load balancing with multiple hash functions in peer-to-peer networks. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 411-420, Minneapolis, Minnesota, July 2006.

[YKPJ05] Y. Lin, B. Kemme, M. Patino-Martinez, R. Jimenez-Peris. Middleware based Data Replication providing Snapshot Isolation. *ACM SIGMOD Int. Conf. on Management of Data*. Baltimore, USA, Jun. 2005

[ZHSR$^+$04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC),* 22(1):41-53, January 2004.

ANNEX A

Update Propagation Strategies to Improve Freshness in
Lazy Master Replicated Databases.
E. Pacitti, E. Simon.
***The VLDB Journal*, Vol. 8, No. 3-4, 2000, 305-318.**

The VLDB Journal
© Springer-Verlag 2000

Update propagation strategies to improve freshness in lazy master replicated databases

Esther Pacitti[1], **Eric Simon**[2] ⋆

[1] NCE-UFRJ Brazil and INRIA Rocquencourt; e-mail: Esther.Pacitti@inria.fr
[2] INRIA Rocquencourt; e-mail: Eric.Simon@inria.fr

Edited by A.P. Buchmann. Received April 24, 1998 / Revised June 7, 1999

Abstract. Many distributed database applications need to replicate data to improve data availability and query response time. The two-phase commit protocol guarantees mutual consistency of replicated data but does not provide good performance. Lazy replication has been used as an alternative solution in several types of applications such as on-line financial transactions and telecommunication systems. In this case, mutual consistency is relaxed and the concept of freshness is used to measure the deviation between replica copies. In this paper, we propose two update propagation strategies that improve freshness. Both of them use immediate propagation: updates to a primary copy are propagated towards a slave node as soon as they are detected at the master node without waiting for the commitment of the update transaction. Our performance study shows that our strategies can improve data freshness by up to five times compared with the deferred approach.

Key words: Data replication – Distributed databases – Performance evaluation

1 Introduction

In a distributed database system, data is often replicated to improve query performance and data availability. Performance is improved because replica copies are stored at the nodes where they are frequently needed. Availability is improved because replica copies are stored at nodes with independent failure modes. An important goal of concurrency control for a replicated database system is to achieve replica transparency. That is, as far as the users are concerned, a replicated database system should behave like a single copy database. Therefore, the interleaved execution of transactions on a replicated database should be equivalent to a serial execution of these transactions on a single copy database. Hence, *one-copy serializability* is enforced and replica consistency is acheived. Transaction processing in replicated databases has been well studied in [BHG87]. Several protocols such as two-phase commit have been proposed to achieve replica consistency [Gif79, Tho79, BHG87]. Almost all commercially available distributed database systems provide *two-phase commit* as an option [Sch76] to guarantee the consistency of replicated data.

A central problem of several database applications with real-time constraints, such as telecommunication systems, operational data stores [Inm96] or on-line financial transactions [Sha97], is to guarantee a high level of performance. For example, in a global trading system distributed over a wide-area network where exchange rates are replicated, it is crucial to have fast access to replicated data from any trader location. In addition, a change to a rate by a trader at a location must be propagated as fast as possible to all other locations to refresh replicated data. With a two-phase commit protocol (henceforth 2PC), an *update transaction* that updates a replica of an object is committed only after all nodes containing a corresponding replica copy agree to commit the transaction, thereby enforcing that all replica copies are *mutually consistent*. This approach is often referred to as *tight consistency* or *synchronous replication* [CRR96]. Synchronous replication has some drawbacks in practice [Moi96, Sch76]. Many database system vendors indicate that in numerous applications such as the ones we are interested in 2PC is impractical. The major argument is that its reliance on 100% system availability makes maintaining a productive level of transaction throughput for distributed replication impossible.

A good overview of replication capabilities of some commercially available distributed database systems can be found in [Gol94]. Systems like Sybase 10 [Moi96], Oracle 7 [Dav94b] and IBM Datapropagator Relational [Dav94a] support synchronous replication as well as an *optional* protocol that *defers* the updates of replicas. The deferred update protocols support *loose consistency* [Moi96] by allowing some replica copies to be inconsistent for some time. This approach is often referred to as *asynchronous* or *lazy replication*. Lazy replication provides better responsiveness since the waiting operations associated with multisite commit protocols are avoided.

⋆ This work was partially supported by Esprit project EP 22469DWQ "Foundations for Data Warehouse Quality".

With *lazy replication*, a transaction can commit only after updating at least one replica at some node. After the transaction commits, the updates are propagated towards the other replicas which are then updated in separate refresh transactions. Thus, this scheme relaxes the mutual consistency property of 2PC. Furthermore, the interval of time between the execution of the original update transaction and the corresponding refresh transactions may be large due to the propagation and execution of the refresh transactions. The degree of *freshness* indicates the proportion of updates that are reflected by a given replica but have nevertheless been performed on the other replica copies.

In this paper, we address the problem of freshness in lazy replication schemes. We present a lazy master replication framework and assume a functional architecture with one master and multiple slave nodes. We propose an update propagation strategy, called *immediate-propagation*, which works as follows: updates to a replica at some master node are immediately propagated towards the other replica copies held by slave nodes without waiting for the commitment of the original update transaction. Specifically, we propose two variants: *immediate-immediate* and *immediate-wait*. With *immediate-immediate*, a refresh transaction is started at a slave node as soon as the first update operation is received from the master node. With *immediate-wait*, a refresh transaction is started at a slave node after the complete reception of all updates (of the same transaction) from the master node. We present experimental results that demonstrate the improvement of freshness brought by these strategies with respect to a *deferred* strategy, as used by several commercial relational database systems.

This paper is a significantly extended version of [PSdM98]. The extensions are the following. First, we present in more details the update propagation algorithms and introduce the recovery procedures to deal with nodes failures. Second, we extend our performance evaluation by measuring query response times, the impact of *freshness* on the *immediate* strategies when transactions abort and the impact of *freshness* for different network delays. Finally, we include a comparison with related work.

The rest of this paper is structured as follows. Section 2 introduces our replication framework and basic definitions. Section 3 presents the system architecture of master and slave nodes used by our update propagation strategies. Section 4 describes the deferred strategy and the immediate strategies. Section 5 presents our performance evaluation. Section 6 discusses related work. Finally, Sect. 7 concludes.

2 Lazy master framework

With lazy master replication, updates on a primary copy are first committed at the master node. Then, each secondary copy is updated *asynchronously*, in a separate refresh transaction. A replication design typically includes the definition of the data to be replicated, the number of replica copies (i.e., primary and secondary copies), the nodes at which replica copies must be placed and others characteristics. In [PV98], we use a variety of terms to define a replication design. However, several terms are ambiguous and some others are still missing to make clear the characterization of a replication design. In this section, we define a framework that can precisely specify a lazy master replication design.

Four parameters characterize our framework: ownership, propagation, refreshment and configuration. However, the parameters we present are also valid for lazy group replication. We use the term *lazy master replication scheme* to refer to a replication design that fits in our framework. Therefore, whenever the four parameters are set, a replication scheme is established. We do not consider how to choose the data to be replicated since we consider that this involves the knowledge of the semantics of the distributed application.

The *ownership* parameter [GHOS96] defines the node capabilities for updating replica copies. A replica copy that is updatable is called a *primary* copy (denoted by capital letters), otherwise it is called a *secondary* copy (denoted by lowercase letters). In the remainder of this paper, we assume that replica copies are relations. For each primary copy, say R, there is a set of secondary copies $r_1, r_2, \ldots$. We refer to the set of primary and secondary copies as *replica copies*. In addition, we sometimes use the term *replicated data* instead of replica copies. We identify three types of nodes: **Master, Slave** and **MasterSlave**. Whenever a node stores only primary copies, it is be referred to as a master node. Similarly, whenever a node stores only secondary copies it is called a slave node. Finally, a node that stores both primary and secondary copies is called a MasterSlave node.

We assume familiarity with transactional concepts [BHG87]. We focus on three types of transactions that read or write replica copies: *update transactions*, *refresh transactions* and *queries*. An update transaction (denoted by T) updates a set of primary copies $R, S, \ldots$. Each time a transaction is committed, a timestamp (denoted by C) is generated.

A refresh transaction (denoted by RT) is composed by the serial sequence of write operations performed by an update transaction T. A refresh transaction is used to update (henceforth refresh) at least one secondary copy. Finally, a query, Q, consists of a sequence of read operations on secondary copies. We assume that once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol.

The *configuration* parameter defines the component nodes of a replication scheme. For instance, a *1Master-nSlaves* configuration, typically called data dissemination, consists of a replication scheme with a *single* master node, i, and n slaves of i. In this paper, we focus on *1Master-nSlaves* configurations.

The propagation parameter defines "when" the updates to a primary copy must be multicast (henceforth propagated) towards the nodes storing its secondary copies. We focus on two types of propagation: *deferred* and *immediate*. When using a *deferred* propagation strategy, the serial sequence of writes on a set of primary copies performed by an update transaction is propagated together within a message M, after the commitment of T. When using an *immediate* propagation, each write operation performed by a transaction, for instance T, is immediately propagated inside a message m, without waiting for the commitment of the original update transaction T.

The *refreshment* parameter defines when the submission of a refresh transaction starts with respect to a propagation

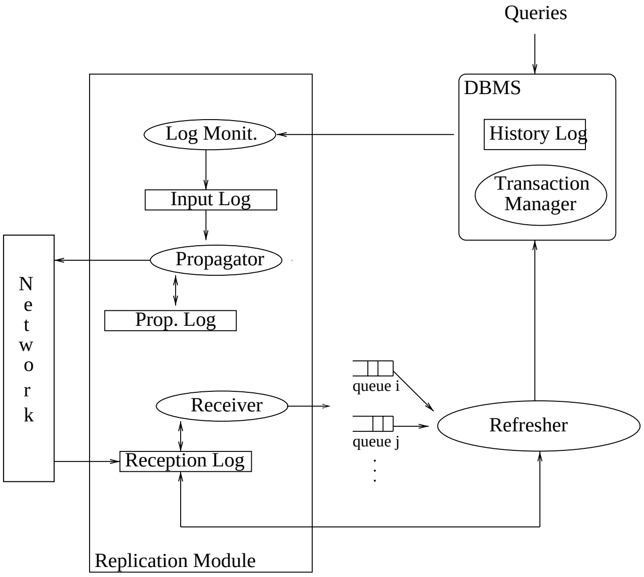


Fig. 1. Architecture of a node: *square boxes* represent persistent data repositories and *oval boxes* represent system components

Table 1. Update propagation strategies

Propagation	*Triggering mode*	*Update propagation*
Deferred	Immediate	Deferred-Immediate
Immediate	Immediate	Immediate-Immediate
Immediate	Wait	Immediate-Wait

strategy. We consider three trigger modes: *deferred*, *immediate* and *wait*. The couple formed by the propagation and the trigger mode determines a specific update propagation strategy. For instance, with a *deferred-immediate* strategy, propagation involves the sending of an RT towards each slave node which, as soon as it is received at a slave node, is submitted for execution. With *immediate-immediate*, propagation involves the sending of each write performed by a T towards each slave node, without waiting for the commitment of the original update transaction. At a slave node, a refresh transaction is started as soon as the first write operation is received from the master or MasterSlave node. Finally, *immediate-wait* is similar to *immediate-immediate*, except that a refresh transaction is submitted for execution only after the complete reception of all write operations of the original update transaction. Table 1 shows the update propagation strategies we focus on.

In our framework, we assume that messages are exchanged among the nodes of the replicated system through a reliable FIFO multicast protocol[1] [HT94] because it is one of the most simple multicast protocols.

3 System architecture

The objective of our architecture is to maintain the autonomy of each node. This means that neither the local transaction management protocols nor query processing are changed to support lazy master replication. Each node, whether master or slave, supports a database system and three components. The first component is the replication module, which itself consists of three components: *Log Monitor*, *Propagator* and *Receiver*. The second component is the *Refresher*, which provides different qualities of service by implementing different refreshment strategies to update secondary copies. The last component, the *Network Interface*, is used to propagate and receive messages on the network. For simplicity, it is not shown in Fig. 1 and need not be discussed in this paper. We now present in more details the functionality of these components:

Log Monitor. It implements *log sniffing* [SKS86, KR87, Moi96], which is a procedure used to extract the changes to a primary copy by reading sequentially the contents of a local history log (denoted by H). We do not consider any particular log file format in our study. However, we safely assume that a log record contains all the information we need such as $timestamp, primary\_id$, and other relevant attributes (see Chap. 9 of [GR93] for more details). When the log monitor finds a write operation on R, it reads the corresponding log record from H and writes it into a stable storage called *input log* that is used by the Propagator. The reason we impose copying the history log entries to the input log is discussed below, when we present the *Propagator's* functionality. We do not deal with conflicts between the write operations on the history log and the read operations performed by the Log Monitor since this procedure is well known and available in commercial systems [Moi96].

Receiver. It implements message reception at the slave node. Messages coming from different masters are received through a network interface that stores them in a *reception log*. The receiver reads messages from the *reception log* and stores them in pending queues. The contents of these queues form the input to the *Refresher*. A slave node detects its master node failure using *timeout* procedures implemented in the network interface. The node recovery procedures related to the Receiver are discussed in Sect. 4.3.

Propagator. It implements the propagation of messages that carry log records issued by the Log Monitor. Both types of messages are written in the *input log*. After correctly propagating a message, its contents is written inside the *propagation log*. If a node failure occurs during message propagation (i.e., before writing the message contents inside the propagation log), the input log is used to re-propagate the lost message at node recovery time. The Propagator continuously reads the records of the *input log* and propagates messages through the network interface. A master node is able to detect failures of its slave nodes using the network interface. The node recovery procedures related to the Propagator are discussed in Sect. 4.3.

Refresher. It implements refreshment strategies that define *when* refresh transactions are executed. For each slave node, there is a pending queue $i, j, ...$. Each pending queue stores a sequence of messages (write opetations) comming from a specific master node (see Fig. 1) . The Refresher reads the contents of each pending queue and, following some refreshment strategy, executes refresh

[1] Messages are received in the same order as they are propagated.

transactions to update the set of secondary copies. A refresh transaction execution is performed by submitting each write operation to the *local transaction manager*. Since we focus on *1Master-nSlaves* configuration, we consider a single pending queue.

4 Update propagation strategies

We consider three update propagation strategies: *deferred-immediate*, which is based on the common approach used by lazy master replication schemes [Moi96], and our *immediate-immediate* and *immediate-wait* strategies. In this section, we present them with respect to our architecture. We first present the deferred and immediate propagation strategies that establish the basis for other strategies. Without loss of generality, we present our algorithms for a *1Master-1Slave* configuration for a single primary copy R.

4.1 Propagation

To preserve serializability, the Log Monitor reads log records from H in the order they were written. It is worth mentioning that the algorithm used to manage H in the local database system is orthogonal to our strategies and has no impact on the way they function. The Propagator also reads log records from the input log in the order they were written and propagates them in serial order. Each log record stored in both H and the input log carries the information necessary to perform an operation, that is, the following attributes [GR93]: *timestamp, primary_id,tuple_id, field_id,operation, new_value*.

The master identifier, *primary_id*, identifies the primary copy R that is updated at the master node. Tuples are identified by their primary key. In addition, the updated field within a tuple is identified by $field\_id$. Next, $operation$ identifies the type of operation (update, delete, insert, abort or commit) performed. In case of an update operation, new_value contains the new value of the field being updated. When there is no ambiguity, we sometimes use the term operation instead of log record.

The Propagator implements the algorithm given in Fig. 2. With *immediate* propagation, each write operation, w_i, of R, by transaction T is read from the input log and forwarded by a *propagate* function in a message m_i containing the input log record to the slave holding a copy r. Thus, for every slave node, there will be as many messages as write operations in T. The performance impact of the possibly large number of messages generated is analyzed in Sect. 5. It is important to note that concurrent update transactions at the master produce an interleaved sequence of write operations in H for different update transactions. However, write operations are propagated in the same order as they were written in the input log to preserve the master serial execution order. Update transaction's *abort* and *commit* at a master are also detected by the Log Monitor.

With *deferred* propagation, the sequence $w_1, w_2, ...w_n$ of operations on R performed by transaction T is packaged within a single message (denoted by M_i) that is propagated to the slave holding r, after reading T's *commit* from the *input log*. The message M_i will form the body of the refresh transaction RT that updates r at the slave node. In case of an *abort* of T, the corresponding M_i under construction is discarded using the *discard* function.

```
Propagator
input: Input Log
output: messages sent to slave nodes
variables:
    $o$: a record read from the Input Log
    $m$: carries $o$
    $M_i$: carries a seq. of $o$ associated with the same transaction
    begin
        repeat
            read(Input Log,$o$);
            if propagation = immediate
                then
                    wrap $o$ into a message $m$;
                    propagate ($m$);
            else /* propagation = deferred */
                if $o$ = $write$ for new transaction $T$
                    then
                        create a message $M$ associated with $T$;
                if $o$ = $write$ for transaction $T$
                    then
                        add $o$ to $M$;
                if $o$ = $commit$
                    then
                        propagate($M$);
                else if $o$ = $abort$
                    then
                        discard($M$);
        for ever;
    end.
```

Fig. 2. Propagator algorithm

```
Immediate-Wait
input: a queue $q_r$
output: submit refreshment transactions
variables:
    $o$: the content of a message stored in $q_r$
    $RV$: vector of write operations for $RT$
    begin
        repeat
            read($q_r$,$o$);
            if $o$ corresponds to a new transaction
                then
                    Create a new vector $RV$;
            if ($o \neq$ commit) and ($o \neq$ abort)
                then
                    add $o$ to its associated vector $RV$;
            if $o$ = commit
                then
                    add $o$ to its vector $RV$;
                    submit $RV$ as an $RT$;
            if $o$ = abort
                then
                    discard the vector associated with $o$;
        for ever.
    end.
```

Fig. 3. Immediate-wait algorithm for queue q_r

4.2 Reception and refreshment

Each Receiver reads the messages M_i in their reception order from the reception log. In addition, each Receiver checks the master identifier *primary_id* to select in which queue to store the log record(s) that each message carries, whenever there is more than one master node. Since we consider a single master node and primary copy R, we simply use q_r to denote the pending queue that stores the messages to update r. We assume that when a message carries a sequence, this sequence is stored as a single record in the queue.

To update a secondary copy r_i, the Refresher continously reads the pending queue, q_r, seeking for new incoming records. With *deferred-immediate*, the Refresher reads a sequence $w_1, w_2, ..., w_n$ within a single message from q_r and subsequently submits it as a refresh transaction to the local transaction manager. The effect of the serial execution order of update transactions $T_1, T_2..., T_k$ performed at the master is preserved at the slave because the corresponding refresh transactions $RT_1, RT_2..., RT_k$ are performed in the same order.

With *immediate-immediate*, each time an operation is read from a queue it is subsequently submitted to the local transaction manager as part of some refresh transaction. Here again, for the same reasons as before, the effect of the serial execution order of the update transactions performed at the master is preserved. When an abort operation for a transaction T is read by the Refresher, it is also submitted to the local transaction manager to abort RT.

With *immediate-wait*, refreshment is done in two steps. In the first step, a message is read from q_r exactly as with *immediate-immediate*. However, each operation associated with a transaction T is stored into an auxiliary data structure, called a reception vector (denoted by RV). Thus, there is one vector per transaction and each element of a vector is an operation. When a *commit* operation for a transaction T is read from queue q_r, it is appended to the corresponding vector RV and a refresh transaction RT is formed with the sequence of operations contained in RV, and submitted to the local transaction manager.

The period of time delimited by the reading, in a queue q_r, of the first operation for transaction T, and the reading of the *commit* operation for T, is called the *wait period* for RT. Figure 3 summarizes the algorithm executed by the Refresher for a given queue q_r using *immediate-wait*.

For all the three strategies, when a refresh transaction RT is committed, the Refresher marks all the messages M_i or m_i in the reception log as *processed*.

4.3 Dealing with node failures

We now present the connection and node recovery protocols used to recover from node failures. They are based on those used for transaction recovery (see [GR93]). Thus, we introduce only the additional features needed for our replication scheme. We first present how a master node initializes a connection with a specific slave node. Then, we show how slave and master node failures are handled. Since we assume that network omissions are bounded and taken into account by the multicast protocol, we can ignore network failures. For pedagogical reasons, whenever necessary, we make clear the distinction between master and slave node procedures.

4.3.1 Initialization

To start update propagation towards a slave node, a master node must first initiate a connection. When update propagation is completed, the master can close the connection. In order to preserve the autonomy of each DBMS, connection and disconnection requests are captured by our replication module through the local history log (denoted by H). The result of a connection or disconnection request is done using the *connection table*, which we describe below. Connection and disconnection are requested using the following functions.

Connect($Master\_id$**,** $Slave\_id$**,** $Replica\_id$**):** $Master\_id$ requests a connection to $Slave\_id$ in order to start update propagation on $Replica\_id$. A corresponding log record is generated and written in the local history log of $Master\_id$ with the following information:

$$<''\ Connect'', Master\_id, Slave\_id, Replica\_id >.$$

Disconnect($Master\_id$**,** $Slave\_id$**,** $Replica\_id$**):** $Master\_id$ requests a disconnection to $Slave\_id$ in order to stop update propagation on $Replica\_id$. A corresponding log record is generated and written in $Master\_id$ local history with the following information:

$$<''\ Disconnect'', Master\_id, Slave\_id, Replica\_id >.$$

Each node i (master or slave node) keeps control of its connections using a *connection table* stored in the local DBMS. Each entry of this table corresponds to a established connection. The main attributes of this table are $Node\_id$, $Replica\_id$ and $Status$. $Node\_id$ identifies a node that is connected to node i and $Replica\_id$ identifies the local replica copy involved in a specific connection. The $status$ attribute indicates the current status of the connection. The interpretation of each attribute at a master or slave node is as follows.

Master: $Replica\_id$ is a primary-copy identifier. The Log Monitor reads the connection table to check which primary copies it must monitor. $Node\_id$ is a slave node identifier associated with a specific connection. By reading the set of slave node identifiers from the connection table, the Propagator is able to request a connection or disconnection and detect a slave node failure. The $status$ attribute indicates whether a connection is *active* or *inactive*. The connection table may be read and updated by the Propagator and Log Monitor, in addition, it may be read by the local DBMS users.

Slave: $Replica\_id$ is a secondary-copy identifier. $Node\_id$ identifies a master node associated with a specific connection. A slave node uses the connection table to identify its masters in order to create and destroy pending queues and detect master node failures. The $status$ attribute indicates whether a connection is *active* or *inactive*. The connection table may be read and updated by the Receiver and read by the local DBMS users.

Node initialization starts by the creation of a connection table a each node. Recall that at each slave node, the

Refresher manages a set of pending queues. Each pending queue may store a sequence of write operations used to update a set of secondary copies. When a master node is inactive for propagating the updates on a specific primary copy, a $Disconnect$ record is stored in the corresponding pending queue at each slave node indicates that the queue is not available for refreshment.

Connection and disconnection are handled as follows. When the Log Monitor reads a $Connect$ (or $Disconnect$) record from H, it writes it in the input log. Then, the Propagator reads the Connect (or Disconnect) record from the input log, and propagates the Connect (or Disconnect) message towards $Slave\_id$. In case of connection, it creates a new connection entry in the local connection table. Otherwise, it deletes the corresponding connection from the connection table.

When a slave node receives the Connect message, its Receiver creates a new connection entry in the local connection table and records the contents of the Connect message into the pending queue corresponding to the master node. Thus, a Connect record informs the Refresher that the queue is available for refreshment. When a slave node receives the Disconnect message, its Receiver deletes the corresponding connection entry from the local connection table and records the contents of the Disconnect message in the corresponding pending queue. Thus, a Disconnect record informs the Refresher that the queue is not available for refreshment.

Disconnection can be granted only after all the messages sent by the master have been received by the slave. This is easy because messages are propagated using a FIFO multicast protocol. Thus, when a slave receives a Disconnect message, it is necessarily the last one for the corresponding connection.

4.3.2 Recovery

We now present how master and slave nodes recover after a node failure.

Slave failure

Slave failure is managed as follows. When a slave node fails, all the connected masters must stop sending messages to it. Failure detection is done by each master Propagator by periodically checking whether the slave is up using the network interface.

Slave recovery is performed in two steps. First, after system recovery, the Receiver writes in each pending queue a $Reconnect$ record of the form:

$$<''\ Reconnect'', Master\_id, Slave\_id, Replica\_id,\\ Message\_id >.$$

$Message\_id$ indicates the last message M_i or m_i (this is easily read by the Receiver based on the *reception log*) and is used as re-synchronization point to re-start the master propagation and the slave refreshment activities. The Refresher reads a Reconnect record and is aware that the connection with $Master\_id$ is being re-established. In the second step, the Receiver re-establishes its pending connections. It does so by searching in the reception log for all messages received from *Master_id* but not yet processed by the Refresher, and stores each message in the correct queue q.

Master failure

Master failure is handled as follows. When a master node fails, all the connected slave nodes detect the failure through their Receiver, which periodically checks for master node availability using the network interface. As soon as a master failure is detected, the slave Receiver writes a $Fail$ record of the form

$$<''\ Fail'', Master\_id >$$

in the correct pending queue. This record informs the Refresher of $Master\_id$ failure. When a master recovers, it re-establishes its connections by propagating a Reconnect message towards each slave node it was connected to. In this case, the $Message\_id$ field is nil. When the slave node receives a Reconnect message, it stores its contents in the correct queue q. Thus, the Reconnect record indicates the Refresher of $Master\_id$ recovery. The Receiver re-starts reception activities for $Master\_id$. As in any database system, the propagation log and reception log are supposed to have enough disk space to deal with recovery.

Our node recovery protocol is non-blocking, because in case of a master failure, the slave refreshment activities on other secondary copies are not interrupted. Using the connection table, a user at a slave node can detect the failure of a master and make a choice on how to proceed with its replication activities. For instance, the user may decide to ignore the master failure at the expense of freshness.

5 Validation

To validate our update propagation strategies, we need to demonstrate their performance improvement. Performance evaluation of such strategies is difficult since several factors such as node speed, multiprogramming level, network bandwitch and others, have a major impact on performance. Some update propagation strategies have been evaluated analytically [GN95, GHOS96]. However, analytical evaluation is typically very complex and hard to understand. In addition, the results may not reflect the real behavior of the strategies under various workloads. Therefore, instead of doing performance analysis of our strategies, we prefer to use a simulation environment that reflects as much as possible a real replication context.

5.1 Simulation environment

There are many factors that may influence the performance of our strategies, for instance, the processing capability of the slave nodes and the speed of the network medium. We isolate the most important factors such as: (i) transaction execution time, reflected by the time spent to log the updates made by a transaction into the history log (denoted by H), (ii) refresh transaction (denoted by RT) propagation time, that is the time needed to propagate the necessary messages from a master to a slave, and (iii) refresh transaction execution time. Consequently, our simulation environment only focuses on the components of a node architecture that determine these three factors. In the following, we present the component modules of the simulation environment.

5.1.1 Modules

Our simulation environment is composed by the following modules: *Master*, *Network*, *Slave* and a database server. The Master module simulates all relevant functionalities of a master node such as log monitoring and message propagation. The Network module implements and simulates the most significant factors that may impact our update propagation strategies such as the delay to propagate a message. The Slave module implements the most relevant components of the slave node architecture such as Receiver, Refresher and Deliverer. In addition, for performance evaluation purposes, we add the Query component in the slave module, which implements the execution of queries that read replicated data. We do not consider node failures in our performance evaluation because we focus on freshness improvement. Therefore, the reception log as well as the recovery procedures are not taken into account. Finally, a database server is used to implement refresh transactions and query execution.

Our environment is implemented on a Sun Solaris workstation in C language using pipes for inter-process communication. We use Oracle 7.3 to implement the database functionalities such as transaction execution, query processing and others.

5.2 Performance model

In this section, we formalize the concept of *freshness* needed for our performance evaluation. Then we present the parameters we considered in our experimentations, as well as the terms used to explain our results.

Freshness is formalized as follows. We may have different transactions sizes but we assume that their occurrence is uniformly distributed over time. Using this assumption of uniformity, we define the concept of $shift$ to simplify the definition of degree of freshness. The $shift$ of r at time t_Q with respect to R when a query Q reads r at a slave node is the difference between the number of committed update transactions of R (denoted by $n(R)$), and the number of committed refresh transactions on r (denoted by $n(r)$):

$$shift(t_Q, r) = n(R) - n(r) .$$

Thus, we define the *degree of freshness*, f, as

$$f = 1 - shift(t_Q, r)/n(R); f \in [0, 1] .$$

Therefore, a degree of freshness close to 0 indicates that data freshness is bad, while a degree of freshness close to 1 indicates that data freshness is excellent. In the remainder of this paper, we use the terms freshness and degree of freshness interchangeably.

Update transactions may be executed in different ways. The *density* of an update transaction T is the average interval of time (denoted by ϵ) between write operations in T as it is reflected in the history log. If, on average, $\epsilon \geq c$, where c is a predefined system parameter, then T is said to be *sparse*. Otherwise, T is said to be *dense*. We focus on *dense* update transactions, i.e., transactions with a small time interval between each two writes. In addition, we vary the transactions arrival rate distribution (denoted by λ_t), which is exponential and reflected in the history log. Updates are done on the same attribute (denoted by $attr$) of a different tuple. Furthermore, we take into account that transactions may *abort*. Therefore, we define an abort transaction percentage (denoted by *abr*) of 0, 5%, 10%, 20% that corresponds to the percentage of transactions that abort in an experimentation. Furthermore, we assume that a transaction abort occurs after half of its execution. For instance abr = 10% means that 10% of the update transactions abort after the execution of half of its write operations.

Table 2. Definition of parameters

ϵ	Density of a T_i
λ_t	mean time interval between trans.
λ_q	mean time interval between queries
$nbmaster$	Number of Master nodes
$\|Q\|$	Query size
$\|RT\|$	Refresh transaction size
ltr	Long transaction ratio
abr	Abort ratio
Protocols	Concurrency control protocols
t_{short}	Prop. time of a single record
δ	Net. delay to prop. a message
t_p	Total propagation time

Network delay is calculated by $\delta + t$, where δ is the network delay introduced to propagate each message and t is the *on-wire* transmission time. In general, δ is considered to be non-significant, and t is calculated by dividing the message size by the network bandwidth [CFLS91]. In our experiments, we use a short message transmission time (denoted by t_{short}), which represents the time needed to propagate a single log record. In addition, we consider that the time spent to transmit a sequence of log records is linearly proportional to the number of log records it carries. The network delay to propagate each message, δ, is implicity modeled by the system overhead to read from and write to sockets. The *total propagation time* (denoted by t_p) is the time spent to propagate all log records associated with a given transaction. Thus, if n represents the size of the transaction with immediate propagation, we have $t_p = n \times (\delta + t_{short})$, while, with deferred propagation, we have $t_p = (\delta + n \times t_{short})$. *Network contention* occurs when δ increases due to the increase of network traffic. In this situation, the delay introduced by δ may impact the total propagation time, especially with immediate propagation.

Refreshment time is the time spent to execute an RT. *Update propagation time* is defined as the time delay between the commitment of RT at the slave and the commitment of its corresponding T at the master. Query arrival rate distribution (denoted by λ_q) is exponential. Query size is supposed to be small and thus fixed to 5.

Refresh transaction execution time is a relevant factor in our simulation environment (see Sect. 5.1) because it may delay by δ a query Q execution time at time t_Q. Meanwhile the degree of freshness may be increased. Therefore, to measure freshness, we fix a 50% conflict rate for each

Table 3. Performance model

ϵ	$mean$ = 100 ms
λ_t	*low: ($mean$ = 10 s),* *bursty: ($mean$ = 200 ms)*
λ_q	Exponential: *low ($mean$ = 15 s)*
$\vert Q\vert$	5
$\vert RT\vert$	5; 50
$nbmaster$	1 to 8
Conflict	50%
Protocols	S2PL, Multiversion
t_{short}	20 ms and 100 ms
ltr	0; 30%; 60%; 100%
abr	0; 5%; 10%; 20%

secondary copy because it gives high chances to have conflicting queries and refresh transactions. This means that each refresh transaction coming from the master node master nodes updates 50% of the tuples read by a query. Each time an update transaction commits, a database variable, called $version\_master$, is incremented. Similarly, each time a refresh transaction commits another database variable, called $version\_slave$, is incremented. For each query, the degree of freshness is computed by subtracting $version\_slave$ from $version\_master$.

We compare the impact of using two concurrency control protocols on query response time. One protocol is the strict two-phase locking protocol (henceforth, S2PL), which may increase response time when a query conflicts with a refresh transaction. Since Oracle 7.3 does not implement S2PL, we simulate it using Oracle's SELECT-FROM-WHERE statement followed by the FOR UPDATE option.

The second protocol is a multiversion protocol, which is implemented in different ways [BHG87] by several commercial database systems. The main idea of a multiversion protocol is to increase the degree of concurrency between transactions through a mechanism that permits the execution of both a query and a transaction in a conflict situation. Here, we focus on the *Snapshot-Isolation*-based multiversion protocol available in Oracle 7.3. In this protocol, a transaction T executing a read operation on a data item always reads the most recent version of that data item that has been committed before the beginning of T, later called *Start_Timestamp* of T. Therefore, T reads a *snapshot* of the database as at the time it started. Updates performed by transactions that are active after the time *Start_Timestamp* are invisible to T. An important point is that, with a *Snapshot Isolation* multiversion protocol, queries never conflict with refresh transactions.

We define two types of update transactions. Small update transactions have size 5 (i.e., five write operations), while long transactions have size 50. To understand the behavior of each strategy in the presence of short and long transactions, we define four scenarios. Each scenario determines a parameter called *long transaction ratio* (denoted by ltr). We set ltr as follows:

- scenario 1: ltr = 0 (all update transactions are short),
- scenario 2: ltr = 30 (30 % of the executed update transactions are long),
- scenario 3: ltr = 60 (60 % of the executed update transactions are long),
- scenario 4: ltr = 100 (all transactions are long).

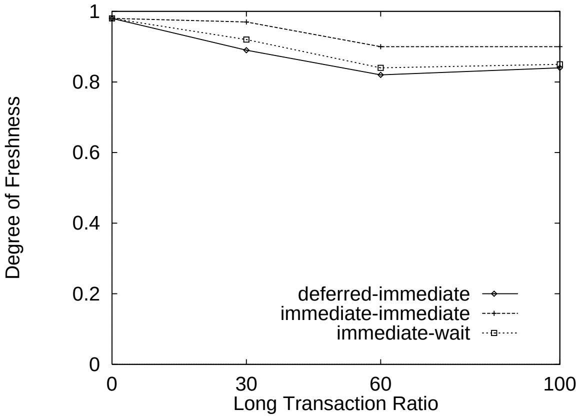


Fig. 4. Low workload - degree of freshness

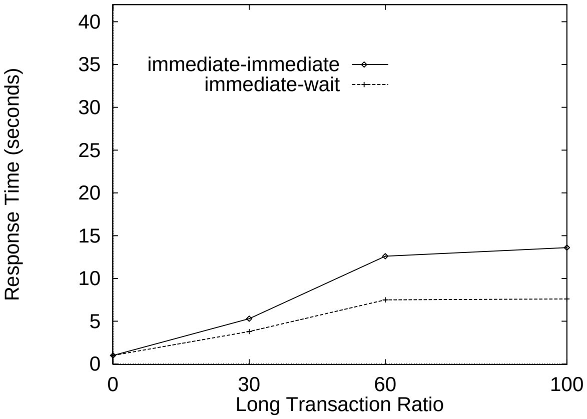


Fig. 5. Low workload - response time

When $ltr > 0$, the value of Max is calculated using the average time spent to propagate a long transaction ($50 \times t_{short}$). On the other hand, when ltr = 0, the value of Max is calculated using the average time spent to propagate a short transaction ($5 \times t_{short}$).

Refresh transaction execution is performed on top of Oracle 7.3 using C/SQL. For simulation purposes, each write operation corresponds to an UPDATE command that is submitted to the server for execution. The definition and values of the parameters of the performance model are summarized in Tables 2 and 3. The results are average values obtained for the execution of 40 update transactions.

5.3 Performance evaluation

The goal of our experimentations is to understand the behavior of the three propagation strategies on *low* and *bursty* scenarios, since these workloads are typical of advanced applications. The first experiment presents our results for the *low* workload and the second one for the *bursty* workload. In the third experiment, we study the impact on freshness when update transactions abort. In the fourth experiment, we verify the freshness improvement of each strategy when the network delay to propagate a message increases. Finally, we discuss our results. Table 4 summarizes each experiment.

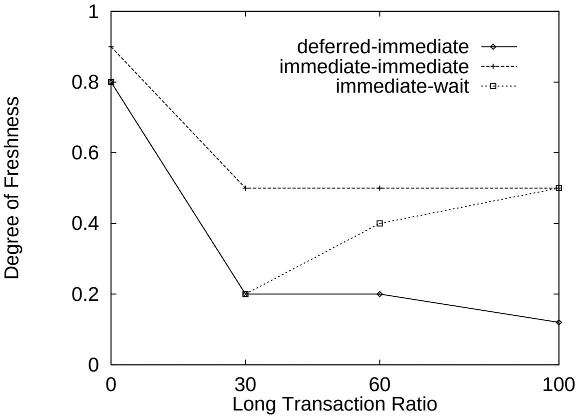


Fig. 6. Bursty workload – degree of freshness

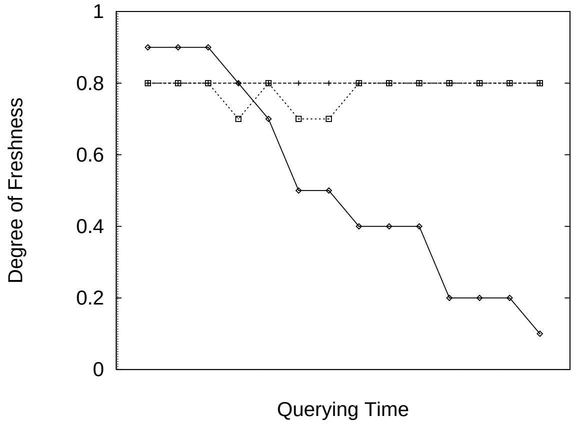


Fig. 7. Bursty workload - freshness behavior (ltr = 100)

5.3.1 Experiment 1

The goal of this experiment is to analyze the query response times and the degree of freshness obtained for a *low* update transaction arrival rate at a master node.

As despicted in Fig. 4, when ltr = 0, the degree of freshness is almost 1, i.e., replicas are almost mutually consistent with the three strategies. The reason is that, on average, $\lambda_t \simeq t_p$, that is the time interval between the execution of a T_i and a subsequent T_{i+1} is sufficiently high to enable completion of T_i's update propagation before the commitment of T_{i+1}. However, for higher ltr values, we have $\lambda_t < t_p$ for the three strategies. Thus, during T_i's update propagation, some transactions $T_{i+1}...T_{i+n}$ may be committed, thereby decreasing the degree of freshness. For all ltr values, the degree of freshness obtained with *deferred-immediate* and *immediate-wait* are close, because the refreshment time is near equal for these two strategies. Furthermore, the total propagation times are also close since there is no network contention.

With *immediate-immediate*, refreshment time is greater than that of other strategies because the time interval between the execution of two write operations, w_j and w_{j+1}, of RT_i is impacted by t_{short}, δ and ϵ, thereby slowing down refreshment time. However, compared to the other two strategies, *immediate-immediate* update propagation time is smaller, because propagation and refreshment are done in parallel. That is, RT_i execution starts after the reception of the first write done by T_i. Therefore, *immediate-immediate* is the strategy that always presents the best degree of freshness. For all strategies, the degree of freshness does not vary linearly with ltr since we are mixing transaction sizes and our freshness measure is based on transaction size.

Table 4. Experiments goal

Experiment	*Measure*	*Vary*	*Workload*
1	Freshness & Response Times	ltr	Low
2	Freshness & Response Times	ltr	Bursty
3	Freshness	abr	Bursty
4	Freshness & Response Times	Net. Delay	Bursty

With *immediate-immediate*, query response time may increase whenever a query conflicts with a refresh transaction, because propagation and refreshment are done in parallel. Therefore, refreshment time is impacted by the total propagation time. However, the chance of conflicts is reduced because $\lambda_t \simeq \lambda_q$. That is the reason why the mean query response times are not seriously affected when ltr = 30 (see Fig. 5). However, with ltr = 60 and ltr = 100, lock-holding times are longer due to the transaction size that increases the number of propagated messages, causing the increase in query response times. Figure 5 shows a situation (with ltr = 100) where response time may be doubled compared to *immediate-wait*. We only show the *immediate-wait* curve since response times for the *deferred-immediate* strategy are very close. When using a multiversion protocol, query response time for the three strategies in all cases is reduced to an average of 1.2 s.

5.3.2 Experiment 2

The goal of this experiment is to analyze the degree of freshness and the query response times obtained for the three strategies for a *bursty* transaction arrival rate.

As depicted in Fig. 6, when ltr = 0 (short transactions), the degree of freshness is already impacted because on average, $\lambda_t < t_p$. Therefore, during T_i's update propagation, $T_{i+1}, T_{i+2}...T_{i+n}$ may be committed. It is important to note that *deferred-immediate* may give a better degree of freshness compared to *immediate-wait*, in bursty workloads, because δ increases sufficiently to increase the *immediate* total propagation time. Therefore, the total propagation time of a short transaction using *deferred* propagation may be less than the total propagation time using *immediate* propagation. On the other hand, even with network contention, *immediate-immediate* yields a better degree of freshness than both *deferred-immediate* and *immediate-wait*, because refreshment begins after the reception of the first write.

When long update transactions are executed, the degree of freshness decreases because t_p increases and $\lambda_t << t_p$. *Immediate-wait* begins to improve and becomes better than *deferred-immediate* when ltr = 30, ltr = 60 and ltr = 100. This is because q_r is quickly filled with a large number of

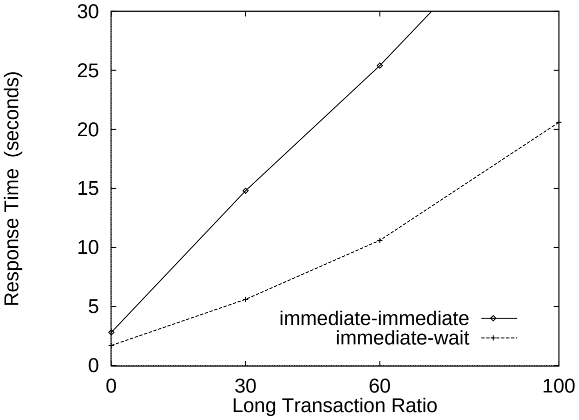


Fig. 8. Bursty workload - response time

operations such that, when the Refresher reads q_r, seeking for a new RT_i, it may happen that all operations associated with RT_i may have been already received and stored in q_r. In this case, the additional wait period of *immediate-wait* may be reduced to 0. This is clearly seen when ltr = 100 (all refresh transactions have the same size). Figure 7 shows a snapshot of the degrees of freshness for a sequence of queries Q_i, Q_{i+1}... at times $t_{Q_i}, t_{Q_{i+1}}$, ... when ltr = 100. For simplicity, we omit the time each query occurred since we are intrested in the behavior of the degree of freshness. The results show that the degrees of freshness obtained with *immediate-immediate* and *immediate-wait* are close to equal. On the other hand, the degree of freshness of *deferred-immediate* decreases rapidly because there is no parallelism of tasks as with the immediate strategies. So, when update transaction sizes increase, update propagation times rise much more compared to the immediate strategies. This impacts the degrees of freshness much more seriously.

Response time is clearly impacted by the *immediate-immediate* strategy (see Fig. 8) for the same reasons as in Experiment 1. The difference here is that $\lambda_t << \lambda_q$, therefore, the chances of conflicts between a query and a refresh transaction augment and the mean query response time increases much more, compared to the results obtained in the *low workload* case. This is well perceived when ltr increases. Using a multiversion protocol, query response time for the three strategies in all cases is also reduced to an average of 1.2 s without a significant decrease of the degree of freshness.

5.3.3 Experiment 3

The goal of this experiment is to show the effects of transaction*aborts* on the degree of freshness. We consider a single master node in a bursty workload.

As shown in Fig. 9, for ltr = 0 and various values of abr (5, 10, 20), the decrease of freshness introduced by update transactions that abort with *immediate-immediate* is insignificant. In the worst case, it achieves 0.2. This behavior is the same for other values of ltr (30, 60, 100). The same behavior is also observed for *immediate-wait*. These results show that the time spent to discard the reception vectors and *undo* refresh transactions at a slave node are insignificant compared to propagation time. Therefore, for the immediate strategies, we may safely state that freshness is not affected in case of update transaction aborts.

With *deferred-immediate*, the degrees of freshness may even increase, because no processing is initiated at the slave node until the complete commitment and propagation of an update transaction. Therefore, while a transaction is aborting at the master node, the slave node may be catching up.

5.3.4 Experiment 4

The goal of this experiment is to show the impact of network delay for message propagation on the degree of freshness and query response times. We consider a single master node in a bursty workload.

Figure 10 compares the freshness results obtained when δ = 100 ms and δ = 20 ms (δ denotes the network delay to propagate a message). When δ = 20 and ltr = 100 *immediate-immediate* improves 1.1 times better than *deferred-immediate* and when δ = 100, *immediate-immediate* improves 5 times better. The improvements of *immediate-wait* compared to *deferred-immediate* are close to these two results. Similar results are obtained when ltr = 60. In addition, in the presence of long transactions, the decrease of freshness when δ = 100 ms is significantly higher for *immediate-wait* and *deferred-immediate* compared to the results obtained when δ = 20 ms. This is because the higher the value of δ is, the longer it takes to propagate a refresh transaction and in average the values of t_p may be much higher than λ_t. Finally, these experiments confirm the benefits of having tasks being performed in parallel when using *immediate-immediate*.

Figure 11 compares the response times obtained when δ = 100 ms and δ = 20 ms. Whenever the value of δ increases, the locking time in conflict situations augments because the refreshment time is proportional to propagation time. When δ = 100 ms, the increase of query response is high when ltr = 30 but insignificant when δ = 20 ms.

5.4 Discussion

We now summarize and discuss the major observations of these experiments. With low workloads, the degree of freshness for one master node is slightly impacted if update transactions are dense and long. In this case, *immediate-wait* and *deferred-immediate* give similar degrees of freshness because their total propagation times are close. *Immediate-immediate* is the strategy that gives the best degree of freshness because propagation and refreshment are done in parallel. *Immediate-immediate* is the only strategy that may introduce an increase in query response time because propagation and refreshment are done in parallel. Therefore, in conflict situations, the increase of query response times may depend on t_{short}, ϵ and δ. However, the mean query response time is not seriously affected because $\lambda_q > \lambda_t$. Therefore, the chances of conflicts are small.

When update transactions arrive in burst, the degree of freshness decreases much more in the presence of long transactions. The *immediate-immediate* strategy still yields

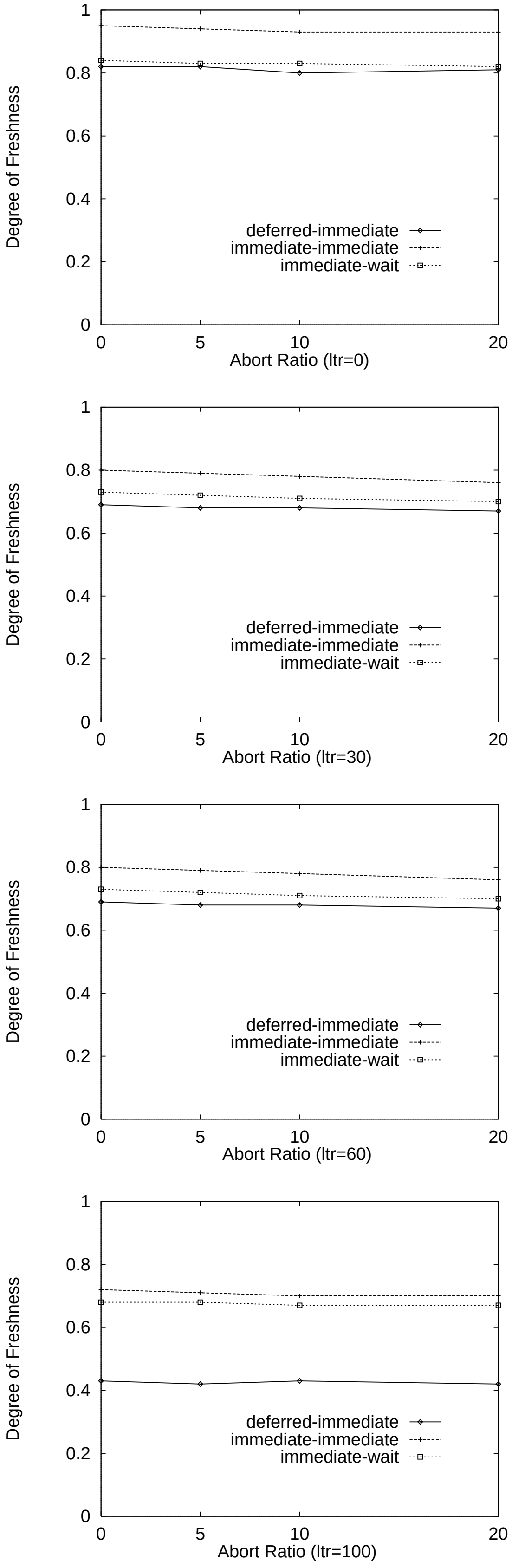


Fig. 9. Bursty workload - *abort* effects

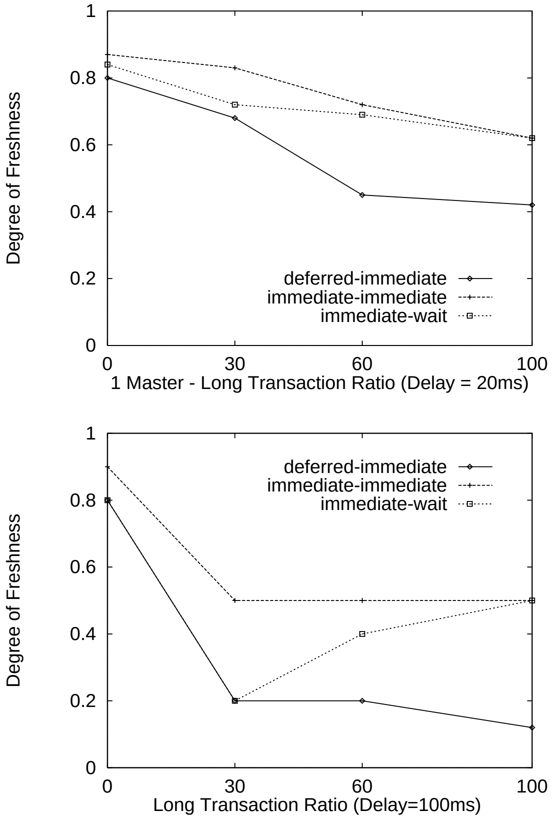


Fig. 10. Increase of network delay - degree of freshness

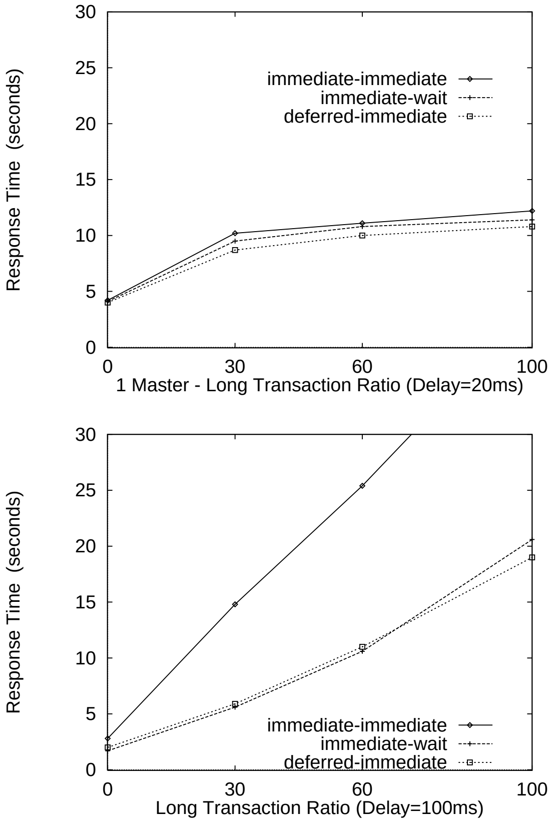


Fig. 11. Increase of network delay - response time (1 master)

the best degree of freshness, even with network contention. When the value of ltr grows, *immediate-wait* gives results close to those of *immediate-immediate*. When all transactions are long, *immediate-wait* performs like *immediate-immediate* because each pending queue is quickly filled with operations due to the immediate propagation. Thus, the effect of the *wait period* of *immediate-wait* may be eliminated because the operations that compose a refresh transaction RT may be already available in q_r whenever the refresher seeks for a new refresh transaction. With *deferred-immediate* the degree of freshness is much more impacted, compared to *immediate-immediate* and *immediate-wait* because there is no parallelism advantage in improving the degree of freshness.

With *immediate-immediate*, the mean query response time may be seriously impacted by the parallelism of propagation and refreshment. When $\lambda_q >> \lambda_t$, the chances of conflicts increase much more compared to the case of low workload. Query response times are much lower with *immediate-wait* because there is no parallelism between propagation and refreshment. Here, the network delay to propagate each operation has an important role and the higher its value, the higher are the query response times in conflict situations. In any case, the use of a multiversion protocol on the slave node may significantly reduce query response times, without a significant decrease in the degree of freshness.

The *abort* of an update transaction with *immediate-immediate* and *immediate-wait* does not impact the degree of freshness since the delay introduced to undo a refresh transaction or discard a reception vector, respectively, are insignificant compared to the propagation time. Finally, the gains obtained with *immediate-immediate* and *immediate-wait* are much more significant when the network delay to propagate a single operation augments. This clearly shows the advantages of having tasks being executed in parallel.

6 Related work

To discuss related work, we use Table 5 which summarizes the two major lazy replication schemes and their basic parameters.

Replication scheme A corresponds to lazy replication where all replica copies are updatable (*update anywhere*). In this case, there is *group ownership* of the replicas. The common update propagation strategy for this scheme is *deferred-immediate*. However, *conflicts* may occur if two or more nodes update the same replica copy. Policies for conflict detection and resolution [Gol95, Bob96] can be based on timestamp ordering, node priority and other strategies. The problem with conflict resolution is that during a certain period of time, the database may be in an inconsistent state. Conflicts cannot be avoided, but their detection may happen earlier by using an immediate propagation.

Replication scheme B is the focus of our work. There are several refreshment strategies for this replication scheme. With *on-demand* refreshment, each time a query is submitted for execution, secondary copies that are read by the query are refreshed by executing all the refresh transactions that have been received. Therefore, a delay may be introduced in query response time. When *group* refresh is used, refresh transactions are executed in groups according to the application's freshness requirements. With the *periodic* approach, refreshment is triggered at fixed intervals. At refreshment time, all received refresh transactions are executed. Finally, with *periodic propagation*, changes performed by update transactions are stored in the master and propagated periodically towards the slaves. Immediate propagation may used be with all refreshment strategies.

Table 5. Replication schemes

Scheme	*Ownership*	*Propagation*	*Refreshment*
A	Group	Deferred Immediate	Immediate (Reconciliation)
B	Master	Deferred Immediate Periodic	Immediate On demand Group Periodic

Incremental agreement is an update propagation strategy [CHKS95] that has some features in common with our proposed strategies. However, it focuses on managing network failures in replicated databases and does not address the problem of improving freshness. Refreshment is performed using the slave log, whereas we use the local transaction manager.

The goal of epidemic algorithms [TTP+95] is to ensure that all replicas of a single data item converge to a single final value in a lazy group replication scheme. Updates are executed locally at any node. Later, nodes communicate to exchange up-to-date information. In our approach, updates are propagated from each primary copy towards all its secondary copies instead.

In [AA95, ABGM90], authors propose weak consistency criteria based on time and space, e.g., a replica should be refreshed after a time interval of after ten updates on a primary copy. There, the concern is not anymore on fast refreshment, and hence these solutions are not adequate to our problem.

In [PMS99], we have formally analyzed and extended the configurations introduced here and focused on replica consistency in different lazy-master-replicated database configurations. For each configuration, we defined sufficient conditions that must be satisfied by a refreshment algorithm in order to be correct. We proposed a refreshment algorithm, which we proved to be correct for a large class of acyclic configurations. In this paper, we focus on freshness improvement for 1Master-nSlaves configurations only and provide extensive experimental results.

The timestamp message delivery protocol found in [Gol92] implements eventual delivery for a lazy group replication scheme [GHOS96]. It uses periodic exchange of messages between pairs of servers that propagate messages to distinct groups of master nodes. At each master node, incoming messages are stored in a history log (as initially proposed in [KR87]) and later delivered to the application in a defined order. Eventual delivery is not appropriate in our framework, since we are interested in improving data freshness.

The stability and convergence of replication schemes A and B are compared in [GHOS96] through an analytical model. The authors show that scheme A has unstable behavior as the workload scales up and that using scheme B reduces the problem. They introduce several concepts such as lazy master and ownership, which we use. They also explore the use of mobile and base nodes. However, immediate propagation is not considered.

Formal concepts for specifying coherency conditions for replication scheme B in large-scale systems are introduced in [GN95]. The authors focus on the *deferredimmediate* strategy. The proposed concepts enable computing an independent measure of relaxation, called *coherency index*. In this context, the concept of *version* is closely related to our notion of freshness.

Freshness measures are related to *coherency conditions* that are widely explored in [AA95, ABGM90] and used in information retrieval systems to define when *cached data* must be updated with respect to changes performed on the central object.

[AKGM96] proposes several derived data refresh strategies such as no-batching, on-demand, periodic and others for derived data refreshment. Replica refreshment and derived data refreshment are done in separate transactions. The authors address freshness improvement, although they focus on the incoherency between derived data and the secondary copy.

Oracle 7.3 [HHB96] implements *event-driven* replication. Triggers on the master tables make copies of changes to data for replication purposes, storing the required change information in tables called *queues* that are periodically propagated. Sybase 10 replication server replicates transactions, not tables, across nodes in the network. The *Log Transfer Manager* implements log monitoring like our approach. However, these systems do not implement immediate propagation and there is no multi-queue scheme for refreshment.

7 Conclusion

In this paper, we addressed the problem of improving freshness in lazy master replication schemes. More specifically, we dealt with update propagation from primary copy to secondary copies. We presented a framework and a functional architecture for master and slave nodes to define update propagation strategies. Focusing on *1Master-nSlave* configurations, we proposed two new strategies called *immediate-immediate* and *immediate-wait*, which improve over the deferred strategy of commercial systems.

To validate our strategies, we performed a thorough performance evaluation through a simulation using Oracle 7.3. The results indicate that, for short transactions, the deferred approach performs almost as well as *immediate-immediate* and *immediate-wait*. The strategies exhibit different freshness results when long transactions occur. In these cases, our strategies show much better results and the *immediate-immediate* strategy provides the best freshness results. For some important kinds of workloads, freshness may be five times better than that of the deferred strategy. On the other hand, *immediate-wait* only improves freshness when the update transaction arrival rate at the master is bursty. The downside of *immediate-immediate* is the increase of query response time due to the transaction blocking when there are conflicts between refresh transactions and queries. However, we argue that, using a multiversion concurrency control protocol at the slave node, this drawback can be drastically reduced without a significant loss of freshness.

The improvement shown by our *immediate* strategies should be beneficial to distributed applications with real-time constraints. For instance, in a global on-line financial trading application exchange rates are replicated. At each slave node, traders will always have a *fresher* "view of the world" and in many cases, the improvement will help avoiding wrong decisions.

Acknowledgements. We would like to thank Dennis Shasha for early discussions which motivated this work and Tamer Özsu and Patrick Valduriez for their final reviews.

References

[AA95] Alonso G, Abbadi A (1995) Partitioned data objects in distributed databases. Distrib Parallel Databases 3(1): 5–35

[ABGM90] Alonso R, Barbara D, Garcia-Molina H (1990) Data-caching issues in an information retrieval system. ACM Trans Database Syst 15(3): 359–384

[AKGM96] Adelberg B, Kao B, Garcia-Molina H (1996) Database support for efficiently maintaining derived data. In: P. Apers, M. Bouzeghoub G. Gardarin (eds) Proc. Int. Conf. on Extending Database Technology (EDBT), March 1996, Avignon, France. Springer, Berlin Heidelberg New York, pp 223–240

[BHG87] Bernstein PA, Hadzilacos V, Goodman N (1987) Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading, Mass.

[Bob96] Bobrowski S (1996) Oracle 7 server concepts, release 7.3. Oracle Corporation, Redwood City, Calif.

[CFLS91] Carey MJ, Franklin MJ, Livny M, Shekita EJ (1991) Data-caching tradeoffs in client-server DBMS architectures. In: J. Clifford, R. King (eds) Proc. ACM SIGMOD Int. Conf. on Management of Data, June 1991, Denver, Colo. ACM Press, New York, pp 357–366

[CHKS95] Ceri S, Houstma MAW, Keller AM, Samarati P (1995) Independent updates and incremental agreement in replicated databases. Distrib Parallel Databases 3(3): 225–246

[CRR96] Chundi P, Rosenkrantz DJ, Ravi SS (1996) Deferred updates and data placement in distributed databases. In: Y. Stanley, W. Su (eds) Proc Int. Conf. on Data Engineering (ICDE), February 1996, Louisiana. Computer Society Press, Los Alamitos, pp 469–476

[Dav94a] Davis J (1994) Data replication. Distrib Comput Monit 9(10): 3–24

[Dav94b] Davis J (1994) Oracle delivers. Open Inf Syst (July)

[GHOS96] Gray J, Helland P, O'Neil P, Shasha D (1996) The danger of replication and a solution. In: H.V Jagadish, I. S. Mumick (eds) Proc. ACM SIGMOD Int. Conf on Management of Data, June 1996, Montreal, Canada. ACM Press, New York, pp 173–182

[Gif79] Gifford DK (1979) Weighted voting for replicated data. In: Proc. ACM SIGOPS Symposium on Operating Systems Principles, December 1979, Pacific Grove, Calif. ACM Press, New York , pp 150–162

[GN95] Gallersdorfer R, Nicola M (1995) Improving performance in replicated databases through relaxed coherency. In: U. Dayal, P. Gray, S. Nishio (eds) Proc. Int. Conf. on VLDB, September 1995, Zurich, Switzerland. Morgan Kaufmann, San Francisco, pp 445–456

[Gol92] Goldring R (1992) Weak-consistency group communication and membership. PhD Thesis. University of Santa Cruz, Calif.

[Gol94] Goldring R (1994) A discussion of relational database replication technology. InfoDB 8(1): 21–26

[Gol95] Golding R (1995) Things every update replication customer should know. In: M. Carey, D. Schneider (eds) Proc. ACM SIGMOD Int. Conf. On Management of Data, June 1995, San Jose, Calif. ACM Press, New York, pp 439–440

[GR93] Gray JN, Reuter A (1993) Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Mateo, Calif.

[HHB96] Helal AA, Heddaya AA, Bhargava BB (1996) Replication Techniques in Distributed Systems. Kluwer Academic, Dordrecht

[HT94] Hadzilacos V, Toueg S (1994) A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR-94-1425. Dept. of Computer Science, Cornell University, Ithaca, N.Y.

[Inm96] Inmon WH (1996) Building the Data Warehouse. John Wiley & Sons, Chichester

[KR87] Kahler B, Risnes O (1987) Extending logging for database snapshot refresh. In: P. M. Stocker, W. Kent (eds) Proc. Int. Conf on VLDB, September 1987, Brighton, UK. Morgan Kaufmann, Los Altos, pp 387–398

[Moi96] Moissis A (1996) Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information. Technical document. Sybase Inc.

[PMS99] Pacitti E, Minet P, Simon E (1999) Fast algorithms for maintaining replica consistency in lazy- master replicated databases. In: M. Atkinson, M. E. Orlowska, P. Valduriez, S. Zdonik, M. Brodie (eds) Proc. Int. Conf on VLDB, September 1999, Edinburgh, UK. Morgan Kaufman, Orlando, pp 126–137

[PSdM98] Pacitti E, Simon E, Melo R de (1998) Improving data freshness in lazy-master schemes. In: M. P. Papazoglou, M. Takizawa, B. Kramer, S. Chanson (eds) Proc. Int. Conf. on Distributed Computing Systems (ICDCS), May 1998, Amsterdam, Netherlands. Computer Society Press, Los Alamitos , pp 164–171

[PV98] Pacitti E, Valduriez P (1998) Replicated databases: concepts, architectures and techniques. Networking Inf Syst Journal 1(3): 519–546

[Sch76] Schussel G (1976) Database replication: Playing both ends against the middleware. Client /Server Today (November): 10–14

[Sha97] Shasha D (1997) Lessons from Wall Street: case studies in configuration, timing and distribution. In: J. M. Peckman (eds) Proc. ACM SIGMOD Int. Conf on Management of Data, June 1997, Tucson, Ariz. ACM Press, New York, pp 498–501

[SKS86] Kahler87 Sarin SK, Kaufman CW, Somers JE (1986) Using history information to process delayed database updates. In: W. Chu, G. Gardarin S. Ohsuga, (eds) Int. Conf. on VLDB, August 1986, Kyoto, Japan. Morgan Kaufman, San Francisco, pp 71–78

[Tho79] Thomas RH (1979) A majority consensus approach to concurrency control for multiple-copy databases. ACM Trans Database Syst 4(2): 180–209

[TTP+95] Terry DB, Theimer MM, Petersen K, Demers AJ, Spreitzer MJ, Hauser CH (1995) Managing update conflicts in bayou, a weakly connected replicated storage system. In Symposium on Operating System Principles (SIGOPS), December 1995, Colorado. Operating System Review, ACM Press, 29(5), pp 172–183

ANNEX B

Replica Consistency in Lazy Master Replicated Databases.

E. Pacitti, P. Minet, E. Simon

***Distributed and Parallel Databases,* Vol . 9, No. 3, 2001, 237-267**

Distributed and Parallel Databases, 9, 237–267, 2001

Replica Consistency in Lazy Master Replicated Databases

ESTHER PACITTI
NCE-UFRJ Rio de Janeiro, Brazil

PASCALE MINET
ERIC SIMON
INRIA Rocquencourt, France

Recommended by: Ahmed Elmagarmid

Abstract. In a lazy master replicated database, a transaction can commit after updating one replica copy (primary copy) at some master node. After the transaction commits, the updates are propagated towards the other replicas (secondary copies), which are updated in separate refresh transactions. A central problem is the design of algorithms that maintain replica's consistency while at the same time minimizing the performance degradation due to the synchronization of refresh transactions. In this paper, we propose a simple and general refreshment algorithm that solves this problem and we prove its correctness. The principle of the algorithm is to let refresh transactions wait for a certain "deliver time" before being executed at a node having secondary copies. We then present two main optimizations to this algorithm. One is based on specific properties of the topology of replica distribution across nodes. In particular, we characterize the nodes for which the deliver time can be null. The other improves the refreshment algorithm by using an immediate update propagation strategy.

Keywords: replicated data, distributed database, data mart, data warehouse, replica consistency, lazy replication, refreshment algorithm, correctness criteria

1. Introduction

Lazy replication (also called asynchronous replication) is a widespread form of data replication in (relational) distributed database systems [27]. With lazy replication, a transaction can commit after updating one replica copy.[1] After the transaction commits, the updates are propagated towards the other replicas, and these replicas are updated in separate refresh transactions. In this paper, we focus on a specific lazy replication scheme, called *lazy master* replication [18] (also called Single-Master-Primary-Copy replication in [4]). There, one replica copy is designated as the *primary copy*, stored at a *master* node, and update transactions are only allowed on that replica. Updates on a primary copy are distributed to the other replicas, called *secondary copies*. A major virtue of lazy master replication is its ease of deployment [4, 18]. In addition, lazy master replication has gained considerable pragmatic interest because it is the most widely used mechanism to refresh data warehouses and data marts [8, 27].

However, lazy master replication may raise a consistency problem between replicas. Indeed, an observer of a set of replica copies at some node at time t may see a state I of these copies that can never be seen at any time, before or after t, by another observer of the same copies at some other node. We shall say that I is an *inconsistent* state. As a first example, suppose that two data marts S_1 and S_2 both have secondary copies of two primary copies stored at two different data source nodes.[2] If the propagation of updates coming from different transactions at the master nodes is not properly controlled, then refresh transactions can be performed in a different order at S_1 and S_2, thereby introducing some inconsistencies between replicas. These inconsistencies in turn can lead to inconsistent views that are later almost impossible to reconciliate [20].

Let us expand the previous example into a second example. Suppose that a materialized view V of S_1, considered as a primary copy, is replicated in data mart S_2. Now, additional synchronization is needed so that the updates issued by the two data source nodes *and* the updates of V issued by S_1 execute in the same order for all replicas in S_1 and S_2.

Thus, a central problem is the design of algorithms that maintain replica's consistency in lazy master replicated databases, while minimizing the performance degradation due to the synchronization of refresh transactions. Considerable attention has been given to the maintenance of replicas' consistency. First, many papers addressed this problem in the context of lazy group replicated systems, which require the reconciliation of updates coming from multiple primary copies [1, 15, 18, 30, 33]. Some papers have proposed to use weaker consistency criterias that depend on the application semantics. For instance, in the OSCAR system [10], each node processes the updates received from master nodes according to a specific weak-consistency method that is associated with each secondary copy. However, their proposition does not yield the same notion of consistency as ours. In [2, 3, 31], authors propose some weak consistency criterias based on time and space, e.g., a replica should be refreshed after a time interval or after 10 updates on a primary copy. There, the concern is not anymore on fast refreshment and hence these solutions are not adequate to our problem. In [9], the authors give conditions over the placement of secondary and primary copies into sites under which a lazy master replicated database can be guaranteed to be globally serializable (which corresponds to our notion of consistency). However, they do not propose any refreshment algorithm for the cases that do not match their conditions, such as our two previous examples. Finally, some synchronization algorithms have been proposed and implemented in commercial systems, such as Digital's Reliable Transaction Router [4], where the refreshment of all secondary copies of a primary copy is done in a distributed transaction. However, to the best of our knowledge, these algorithms do not assure replica consistency in cases like our second above example.

This paper makes three important contributions with respect to the central problem mentionned before. First, we analyze different types of configurations of a lazy master replicated system. A configuration represents the topology of distribution of primary and secondary copies accross the system nodes. It is a directed graph where a directed arc connects a node N to a node N' if and only if N holds a primary copy of some secondary copy in N'. We formally define the notion of correct refreshment algorithm that assures database consistency. Then, for each type of configuration, we define sufficient conditions that must be satisfied by a refreshment algorithm in order to be correct. Our results generalize already published results such as [9].

As a second contribution, we propose a simple and general refreshment algorithm, which is proved to be correct for a large class of acyclic configurations (including for instance, the two previous examples). We show how to implement this algorithm using system components that can be added to a regular database system. Our algorithm makes use of a reliable multicast with a known upper bound, that preserves a global FIFO order. Our algorithm also uses a deferred update propagation strategy, as offered by all commercial replicated database systems. The general principle of the algorithm is to make every refresh transaction wait a certain "deliver time" before being executed.

As a third contribution, we propose two main optimizations to this algorithm. First, using our correctness results on configurations types, we provide a static characterization of nodes that do not need to wait. Second, we give an optimized version of the algorithm that uses an immediate update propagation strategy, as defined in [28]. We give a performance evaluation based on simulation that demonstrates the value of this optimization by showing that it significantly improves the freshness of secondary copies.

This paper is a significantly extended version of [29]. The extensions are the following. First, we provide all the proofs of the propositions defining our correctness criterias. Second, we extend our performance evaluation by measuring freshness and the impact of update transactions aborts for up to 8 master nodes. Third, we expand the related work section.

The rest of this paper is structured as follows. Section 2 introduces our lazy master replication framework, and the typology of configurations. Section 3 defines the correctness criteria for each type of configuration. Section 4 describes our refreshment algorithm, how to incorporate it in the system architecture of nodes, and proves its correctness. Section 5 presents our two main optimizations. Section 6 introduces our simulation environment and presents our performance evaluation. Section 7 discusses some related work. Finally, Section 8 concludes.

2. Lazy master replicated databases

We define a (relational) lazy replicated database system as a set of n interconnected database systems, henceforth called *nodes*. Each node N_i hosts a relational database whose schema consists of a set of pairwise distinct relational schemas, whose instances are called relations. A replication scheme defines a partitioning of all relations of all nodes into partitions, called *replication sets*. A replication set is a set of relations having the same schema, henceforth called *replica copies*.[3] We define a special class of replicated systems, called *lazy master*, which is our framework.

2.1. Ownership

Following [18], the *ownership* defines the node capabilities for updating replica copies. In a replication set, there is a single updatable replica copy, called *primary* copy (denoted by a capital letter), and all the other relations are called *secondary* copies (denoted by lower-case letters). We assume that a node never holds the primary copy and a secondary copy of the same replication set. We distinguish between three kinds of nodes in a lazy master replicated system.

Definition 2.1 (Types of nodes).

1. A node M is said to be a *master* node iff : $\forall m \in M$ m is a primary copy.
2. A node S is said to be a *slave* node iff : $\forall s \in S$ s is a secondary copy of a primary copy of some master node.
3. A node MS is said to be a *master/slave* node iff: $\exists ms$ and $ms' \in MS$, such that ms is a primary copy and ms' is a secondary copy.

Finally, we define the following slave and master dependencies between nodes. A node M is said to be a *master node* of a node S iff there exists a secondary copy r in S of a primary copy R in M. We also say that S is a *slave node of* M.

2.2. Configurations

Slave dependencies define a DAG, called configuration.

Definition 2.2 (Configuration). A configuration of a replicated system is defined by a directed graph, whose nodes are the nodes of the replicated system, and there is a directed are from a node N to a node N' iff N' is a slave node of N. Node N is said to be a predecessor of N'.

In the following, we distinguish different types of configurations. Intuitively, to each configuration will correspond a correctness criterion to guarantee database consistency. In the figures illustrating the configurations, we use integers to represent nodes in order to avoid confusion with the names of the relations that are displayed as annotation of nodes.

Definition 2.3 (1 master-per-slave configuration). An acyclic configuration in which each node has at most one predecessor is said to be a *1master-per-slave* configuration.

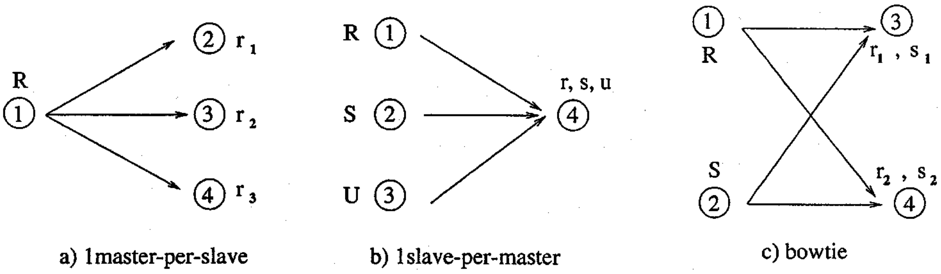


Figure 1. Examples of configurations.

This configuration, illustrated in figure 1(a), corresponds to a "data dissemination" scheme whereby a set of primary copies of a master or master/slave node is disseminated towards a set of nodes. It characterizes for instance the case of several data marts built over a centralized corporate data warehouse.

Definition 2.4 (1slave-per-master configuration). An acyclic configuration in which each node has at most one successor is said to be a *1slave-per-master* configuration.

This configuration, illustrated in figure 1(b), corresponds to what is often called a "data consolidation" scheme, whereby primary copies coming from different nodes are replicated into a single node. It characterizes for instance a configuration wherein a data warehouse node (or even, an operational data store node) holds a set of materialized views defined over a set of relations stored by source nodes. In this context, replicating the source relations in the data warehouse node has two main benefits. First, one can take advantage of the replication mechanism to propagate changes from the source towards the data warehouse. Second, it assures the self-maintainability of all materialized views in the data warehouse, thereby avoiding the problems mentioned in [35].

Definition 2.5 (bowtie configuration). An acyclic configuration in which there exist two distinct replicas X_1 and X_2 and four distinct nodes M_1, M_2, S_1 and S_2 such that (i) M_1 holds the primary copy of X_1 and M_2 the primary copy of X_2, and (ii) both S_1 and S_2 hold secondary copies of both X_1 and X_2.

Such configuration, illustrated in figure 1(c), generalizes the two previous configurations by enabling arbitrary slave dependencies between nodes. This configuration characterizes, for instance, the case of several data marts built over several data sources. The benefits of a replication mechanism are the same as for a data consolidation configuration.

Definition 2.6 (triangular configuration). An acyclic configuration in which there exist three distinct nodes M, MS and S such that (i) MS is a successor of M, and (ii) S is a successor of both M and MS, is said to be a *triangular* configuration. Nodes M, MS and S are said to form a triangle.

This configuration, illustrated in figure 2 (a), slightly generalizes the two first configurations by enabling a master/slave node to play an added intermediate role between a master node and a slave node. This configuration was also considered in [9].

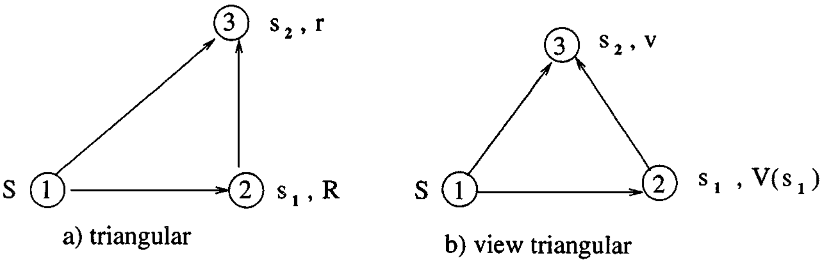


Figure 2. Examples of configurations.

Definition 2.7 (materialized view). A primary copy of a master/slave node *MS* which is defined as the result of the query over a set of secondary copies of *MS* is called a *materialized view*.

Definition 2.8 (view triangular configuration). A derived configuration in which all the primary copies hold by any node *MS* of any triangle are materialized views of local secondary copies, is said to be a *view triangular* configuration.

This configuration, illustrated in figure 2(b), characterizes, for instance, the case of two independent data marts defined over the same data warehouse in which one of the data mart replicates some materialized view of the other data mart. Although they overlap, the bowtie and the view triangular configurations are incomparable (none is included into the other).

2.3. Transaction model

The *transaction model* defines the properties of the transactions that access the replica copies at each node. Moreover, we assume that once a transaction is submitted for execution to a local transaction manager at a node, all conflicts are handled by the local concurrency control protocol, in such a way that serializability of local transactions is ensured.

We focus on three types of transactions that read or write replica copies: *update transactions*, *refresh transactions* and *queries*. All these transactions access only local data.

An update transaction is a local user transaction (i.e., executing on a single node) that updates a set of primary copies. Updates performed by an update transaction T are made visible to other transactions only after T's commitment. We denote T_{R_1,R_k} an update transaction T that updates primary copies R_1, R_k. We assume that no user transaction can update a materialized view.

A refresh transaction associated with an update transaction T and a node N, is composed by the serial sequence of write operations performed by T on the replica copies hold by N. We denote RT_{r_1,r_k} a refresh transaction that updates secondary copies r_1, r_k. Finally, *a query transaction*, noted Q, consists of a sequence of read operations on replica copies.

2.4. Propagation

The propagation parameter defines “when” the updates to a primary copy must be multicast towards the nodes storing its secondary copies. The multicast is assumed to be reliable and to preserve the global FIFO order [22]: the updates are received by the involved nodes in the order they have been multicast by the node having the primary copy.

Following [28], we focus on two types of propagation: *deferred* and *immediate*. When using a *deferred* propagation strategy, the sequence of operations of each refresh transaction associated with an update transaction T is multicast to the appropriate nodes within a single message M, after the commitment of T. When using an *immediate* propagation, each operation of a refresh transaction associated with an update transaction T is immediately multicast inside a message m, without waiting for the commitment of T.

2.5. Refreshment

The *refreshment algorithm* defines: (i) the *triggering parameter* i.e., when a refresh transaction is started, and (ii) the *ordering parameter* i.e., the commit order of refresh transactions.

We consider three triggering modes: *deferred*, *immediate* and *wait*. The combination of a propagation parameter and a triggering mode determines a specific update propagation strategy. With a *deferred-immediate* strategy, a refresh transaction RT is submitted for execution as soon as the corresponding message M is received by the node. With an *immediate-immediate* strategy, a refresh transaction RT is started as soon as the first message m corresponding to the first operation of RT is received. Finally, with an *immediate-wait* strategy, a refresh transaction RT is submitted for execution only after the last message m corresponding to the commitment of the update transaction associated with RT is received.

3. Correctness criteria

In this section, we first formally define the notion of a correct refreshment algorithm, which characterizes a refreshment algorithm that does not allow inconsistent states in a lazy master replicated system. Then for each type of configuration introduced in Section 2, we provide criteria that must be satisfied by a refreshment algorithm in order to be correct.

We now introduce useful preliminary definitions similar to those used in [19] in order to define the notion of a consistent replicated database state. We do not consider node failures, which are out of the scope of this paper. As a first requirement, we impose that any committed update on a primary copy must be eventually reflected by all its secondary copies.

Definition 3.1 (Validity). A refreshment algorithm used in a lazy master replicated system is said valid iff any node that has a copy of a primary copy updated by a committed transaction T is guaranteed to commit the refresh transaction RT associated with T.

Definition 3.2 (Observable State). Let N be any node of a lazy master replicated system, the observable state of node N at local time t is the instance of the local data that reflects all and only those update and refresh transactions committed before t at node N.

In the next definitions, we assume a global clock so that we can refer to global times in defining the notion of consistent global database state. The global clock is used for concept definition only. We shall also use the notation $I_t[N](Q)$ to denote the result of a query transaction Q run at node N at time t.

Definition 3.3 Quiescent State). A lazy master replicated database system is in a quiescent state at a global time t if all local update transactions submitted before t have either aborted or committed, and all the refresh transactions associated with the committed update transactions have committed.

Definition 3.4 Consistent Observable State). Let N be any node of a lazy master replicated system D. Let t be any global time at which a quiescent state of D is reached. An observable state of node N at time $t_N \leq t$ is said to be consistent iff for any node N' holding a non-empty set X of replica copies hold by N and for any query transaction Q over X, there exists some time $t_{N'} \leq t$ such that $I_{t_N}[N](Q) = I_{t_{N'}}[N'](Q)$.

Definition 3.5 Correct Refreshment Algorithm for a node N). A refreshment algorithm used in a lazy master replicated system D, is said to be correct for a node N of D iff it is valid and for any quiescent state reached at time t, any observable state of N at time $t_N \leq t$ is consistent.

Definition 3.6 Correct Refreshment Algorithm). A refreshment algorithm used in a lazy master replicated system D, is said to be correct iff it is correct for any node N of D.

In the following, we define correctness criteria for acyclic configurations that are sufficient conditions on the refreshment algorithm to guarantee that it is correct.

3.1. Global FIFO ordering

For 1master-per-slave configurations, inconsistencies may arise if slaves can commit their refresh transactions in an order different from their corresponding update transactions. Although in 1slave-per-master configurations, every primary copy has a single associated secondary copy, the same case of inconsistency could occur between the primary and secondary copies. The following correctness criterion prevents this situation.

Definition 3.7 Global FIFO order). Let T_1 and T_2 be two update transactions committed by the same master or master/slave node M. If M commits T_1 before T_2, then at every node having a copy of a primary copy updated by T_1, a refresh transaction associated with T_2 can only commit after the refresh transaction associated with T_1.

Proposition 3.1. *If a lazy master replicated system D has an acyclic configuration which is neither a bowtie nor a triangular configuration, and D uses a valid refreshment algorithm meeting the global FIFO order criterion, then this refreshment algorithm is correct.*

See the proof in the Section 9.1 of the appendix. A similar result was shown in [9] using serializability theory.

3.2. Total ordering

Global FIFO ordering is not sufficient to guarantee the correctness of refreshment for bowtie configurations. Consider the example in figure 1(c). Two master nodes, node 1 and node 2, store relations $R(A)$ and $S(B)$, respectively. The updates performed on R by some transaction T_R: insert $R(A : a)$, are multicast towards nodes 3 and 4. In the same way, the updates performed on S by some transaction T_S: insert $S(B : b)$, are multicast towards nodes 3 and 4. With the correctness criterion of Proposition 3.1, there is no ordering among the commits of refresh transactions RT_r and RT_s associated with T_R and T_S. Therefore, it might happen that RT_r commits before RT_s at node 3 and in a reverse order at node 4. In which case, a simple query transaction Q that computes $(R - S)$ could return an empty result at node 4, which is impossible at node 3. The following criterion requires that RT_r and RT_s commit in the same order at nodes 3 and 4.

Definition 3.8 (Total order). Let T_1 and T_2 be two committed update transactions. If two nodes commit both the associated refresh transactions RT_1 and RT_2, they both commit RT_1 and RT_2 in the same order.

Proposition 3.2. *If a lazy master replicated system D that has a bowtie configuration but not a triangular configuration, uses a valid refreshment algorithm meeting the global FIFO order and the total order criteria, then this refreshment algorithm is correct.*

See the proof in the Section 9.2 of the appendix.

3.3. Master/slave induced ordering

We first extend the model presented in Section 2 to deal with materialized views as follows. From now on, we shall consider that in a master/slave node *MS* having a materialized view, say $V(s_1)$, any refresh transaction of s_1 is understood to encapsulate the update of some virtual copy $\hat{V}$. The actual replica copies V and v are then handled as if they were secondary copies of $\hat{V}$. Hence, we consider that the update of the virtual copy $\hat{V}$ is associated with:

- at node *MS*, a refresh transaction of V, noted RT_V,
- ¤ at any node *S* having a secondary copy v, a refresh transaction of V noted RT_V.

With this new modeling in mind, consider the example of figure 2(b). Let $V(A)$ be the materialized view defined from the secondary copy $s1$. Suppose that at the initial time t_o of the system, the instance of $V(A)$ is: $\{V(A:8)\}$ and the instance of $S(B)$ is: $\{S(B:9)\}$. Suppose that we have two update transactions T_s and $T_{\hat{V}}$, running at nodes 1 and 2 respectively: T_s: [delete $S(B:9)$; insert $S(B:6)$], and $T_{\hat{V}}$: [if exists $S(B:x)$ and $x \leq 7$ then delete $V(A:8)$; insert $V(A:5)$]. Finally, suppose that we have the query transaction Q over V and S, Q: [if exists $V(A:x)$ and $S(B:Y)$ and $y < x$ then *bool* = *true* else *bool* = *false*], where *bool* is a variable local to Q.

Now, a possible execution is the following. First, T_s commits at node 1 and its update is multicast towards nodes 2 and 3. Then, RT_{s_1} commits at node 2. At this point of time, say t_1, the instance of s_1 is $\{s_1(B:6)\}$. Then the update transaction $T_{\hat{V}}$ commits, afterwards the refresh transaction RT_V commits. The instance of V is $\{V(A:5)\}$. Then at node 3, RT_v commits (the instances of v and s_2 are $\{v(A:5)\}$ and $\{s_2(B:9)\}$), and finally, RT_{s_2} commits (the instances of v and s_2 are $\{v(A:5)\}$ and $\{s_2(B:6)\}$). A quiescent state is reached at this point of time, say t_2.

However, there exists an inconsistent observable state. Suppose that Q executes at time t_1 on node 2. Then, Q will return a value *true* for *bool.* However, for any time between t_0 and t_2, the execution of Q on node 3 will return a value *false* for *bool*, which contradicts our definition of consistency.

The following criterion imposes that the commit order of refresh transactions must reflect the commit order at the master/slave node.

Definition 3.9 (Master/slave induced order). If *MS* is a node holding a secondary copy s_1 and a materialized view V, then any node N_i, $i > 1$, having secondary copies s_i and v_i must commit its refresh transactions RT_{s_i} and RT_{v_i} in the same order as RT_V and RT_{s_1} commit at *MS*.

Proposition 3.3. *If a lazy master replicated system D that has a view triangular configuration but not a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order and the master/slave induced order criteria then this refreshment algorithm is correct.*

See the proof in the Section 9.3 of the appendix.

As said before, a configuration can be both a bowtie and a view triangular configuration. In this case, the criteria for both configurations must be enforced.

Proposition 3.4. *If a lazy master replicated system D having both a view triangular configuration and a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order, the master/slave induced order and the total order criteria, then this refreshment algorithm is correct.*

See the proof in the Section 9.3 of the appendix.

4. Refreshment algorithm

We start this section by presenting the system architecture assumed by our algorithms. Then, we present our refreshment algorithm that uses a deferred update propagation strategy and prove its correctness. Finally we discuss the rationale for our algorithm.

4.1. System architecture of nodes

To maintain the autonomy of each node, we assume that four components are added to a regular database system, that includes a transaction manager and a query processor, in order to support a lazy master replication scheme. Figure 3 illustrates these components for a node having both primary and secondary copies. The first component, called *Replication Module*, is itself composed of three sub-components: a Log Monitor, a Propagator and a Receiver. The second component, called *Refresher*, implements a refreshment strategy. The third component, called *Deliverer*, manages the submission of refresh transactions to the local transaction manager. Finally, the last component, called *Network Interface*, is used to propagate and receive update messages (for simplicity, it is not portrayed on figure 3). We now detail the functionality of these components.

We assume that the *Network Interface* provides a global FIFO reliable multicast [22] with a known upper bound [13]: messages multicast by a same node are received in the order

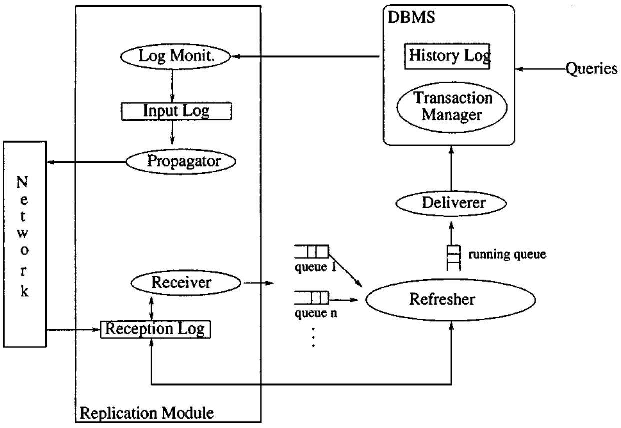


Figure 3. Architecture of a node.

they have been multicast. We also assume that each node has a local clock. For fairness reasons, clocks are assumed to have a bounded drift and to be ε synchronized. This means that the difference between any two correct clocks is not higher than the precision ε.

The *Log Monitor* uses *log sniffing* [23, 30] to extract the changes to a primary copy by continuously reading the content of a local History Log (noted H). We safely assume (see Chap. 9 of [17]) that a log record contains all the information we need such as the timestamp of a committed update transaction, and other relevant attributes that will be presented in the next section. Each committed update transaction T has a timestamp (henceforth denoted C), which corresponds to the real time value at T's commitment time. When the log monitor finds a write operation on a primary copy, it reads the corresponding log record from H and writes it into a stable storage, called *Input Log*, that is used by the Propagator. We do not deal with conflicts between the write operations on the History Log and the read operations performed by the Log Monitor.

The *Receiver* implements update message reception. Messages coming from different masters or master/slaves are received and stored into a *Reception Log*. The receiver then reads messages from this log and stores them in FIFO *pending queues*. We denote *Max*, the upper bound of the time needed to multicast a message from a node and insert it into a pending queue at a receiving node. A node N has as many pending queues $q_1, \ldots, q_n$ as masters or master/slaves nodes from which N has a secondary copy. The contents of these queues form the input to the Refresher.

The *Propagator* implements the propagation of update messages constructed from the Log Monitor. Such messages are first written into the *Input Log*. The propagator then continuously reads the *Input Log* and propagates messages through the network interface.

The *Refresher* implements the refreshment algorithm. First, it reads the contents of the pending queues, and based on its refreshment parameters, submits refresh transactions by inserting them into a *running queue*. The running queue contains all ordered refresh transactions not yet entirely executed.

Finally, the *Deliverer* submits refresh transactions to the local transaction manager. It reads the content of the running queue in a FIFO order and submits each write operation as part of a refresh transaction to the local transaction manager. The local transaction manager ensures serializability of local transactions. Moreover, it executes the operations requested by the refresh transactions according to the submission order given by the *Deliverer*.

4.2. Refreshment algorithm

As described in Section 2, the refreshment algorithm has a triggering and an ordering parameters. In this section, we present the refreshment algorithm in the case of a *deferred-immediate* update propagation strategy (i.e., using an immediate triggering), and focus on the ordering parameter.

The principle of the refreshment algorithm is the following. A refresh transaction *RT* is committed at a slave or master/slave node (1) once all its write operations have been done, (2) according to the order given by the timestamp *C* of its associated update transaction, and (3) at the earliest, at real time $C + Max + \varepsilon$, which is called the deliver time, noted *deliver_time*. Therefore, as clocks are assumed to be ε synchronized, the effects of updates on secondary copies follow the same chronological order in which their corresponding primary copies were updated.

```
Deferred-Immediate Refresher
input: pending queues $q_1 ... q_n$
output: running queue
variables:
  $curr\_M$, $new\_M$: messages from pending queues;
  timer: local reverse timer whose state is either active or inactive;
begin
  timer.state = inactive;
  $curr\_M = new\_M = \emptyset$;
  repeat
    on message arrival or change of timer's state to inactive do
      Step 1:
      $new\_M \leftarrow$ message with min $C$ among top messages of $q_1, q_n$;
      Step 2:
      if $new\_M \neq curr\_M$ then
        $curr\_M \leftarrow new\_M$;
        calculate $deliver\_time(curr\_M)$;
        timer.value $\leftarrow deliver\_time(curr\_M) - local\_time$
        timer.state $\leftarrow$ active;
      endif
    on timer.value = 0 do
      Step 3:
      write $curr\_M$ into running queue;
      dequeue $curr\_M$ from its pending queue;
      timer.state $\leftarrow$ inactive;
  for ever
end
```

Figure 4. Deferred-immediate refreshment algorithm.

We now detail the algorithm given in figure 4. Each element of a pending queue is a message that contains: a sequence of write operations corresponding to a refresh transaction RT, and the timestamp C of the update transaction associated with RT. Since messages successively multicast by a same node are received in that order by the destination nodes, in any pending queue, messages are stored according to their multicast order (or commitment order of their associated update transactions).

Initially, all pending queues are empty, and *curr_M* and *new_M* are empty too. Upon arrival of a new message M into some pending queue signaled by an event, the Refresher assigns variable *new_M* with the message that has the smallest C among all messages in the top of all pending queues. If two messages have equal timestamps, one is selected according to the master or master/slave identification priorities. This corresponds to Step 1 of the algorithm. Then, the Refresher compares *new_M* with the currently hold message *curr_M*. If the timestamp of *new_M* is smaller than the timestamp of *curr_M*, then *curr_M* gets the value of *new_M*. Its deliver time is then calculated, and a local reverse timer is set with value *deliver_time – local_time*. This concludes Step 2 of the algorithm. Finally, whenever the timer expires its time, signaled by an event, the Refresher writes *curr_M* into the running queue and dequeues it from its pending queue. Each message of the running queue will yield a different refresh transaction. If an update message takes *Max* time to reach a pending queue, it can be processed immediately by the Refresher.

4.3. Refreshment algorithm correctness

We first show that the refreshment algorithm is valid for any acceptable configuration. A configuration is said *acceptable* iff (i) it is acyclic, and (ii) if it is a triangular configuration, then it is a view triangular configuration.

Lemma 4.1. *The Deferred-immediate refreshment algorithm is valid for any acceptable configuration.*

Lemma 4.2 (**Chronological order**). *The Deferred-immediate refreshment algorithm ensures for any acceptable configuration that, if T_1 and T_2 are any two update transactions committed respectively at global times t_1 and t_2 then:*

- *if $t_2 - t_1 > \varepsilon$, the timestamps C_2 for T_2 and C_1 for T_1 meet $C_2 > C_1$.*
- *any node that commits both associated refresh transactions RT_1 and RT_2, commits them in the order given by C_1 and C_2.*

Lemma 4.3. *The Deferred-immediate refreshment algorithm satisfies the global FIFO order criterion for any acceptable configuration.*

Lemma 4.4. *The Deferred-immediate refreshment algorithm satisfies the total order criterion for any acceptable configuration.*

Lemma 4.5. *The Deferred-immediate refreshment algorithm satisfies the master/slave induced order criterion for any acceptable configuration.*

From the previous lemmas and propositions, we have:

Theorem 4.1. *The Deferred-immediate refreshment algorithm is correct for any acceptable configuration.*

4.4. *Discussion*

A key aspect of our algorithm is to rely on the upper bound *Max* on the transmission time of a message by the global FIFO reliable multicast. Therefore, it is essential to have a value of *Max* that is not overestimated. The computation of *Max* resorts to scheduling theory (e.g., see [34]). It usually takes into account four kinds of parameters. First, there is the global reliable multicast algorithm itself (see for instance [22]). Second, are the characteristics of the messages to multicast (e.g. arrival laws, size). For instance, in [14], an estimation of *Max* is given for sporadic message arrivals. Third, are the failures to be tolerated by the multicast algorithm, and last are the services used by the multicast algorithm (e.g. medium access protocol). It is also possible to compute an upper bound Max_i for each type i of message to multicast. In that case, the refreshment algorithm at node N waits until $max_{i \in J}$ Max_i where J is the set of message types that can be received by node N.

Thus, an accurate estimation of *Max* depends on an accurate knowledge of the above parameters. However, accurate values of the application dependent parameters can be obtained in performance sensitive replicated database applications. For instance, in the case of data warehouse applications that have strong requirements on freshness, certain characteristics of message can be derived from the characteristics of the operational data sources (usually, transaction processing systems). Furthermore, in a given application, the variations in the transactional workload of the data sources can often be predicted.

In summary, the approach taken by our refreshment algorithm to enforce a total order over an algorithm that implements a global FIFO reliable multicast trades the use of a worst case multicast time at the benefit of reducing the number of messages exchanged on the network. This is a well known tradeoff. This solution brings simplicity and ease of implementation.

5. Optimizations of the refreshment

In this section, we present two main optimizations for the refreshment algorithm presented in Section 4. First, we show that for some configurations, the deliver time of a refresh transaction needs not to include the upper bound (*Max*) of the network and the clock precision (ε), thereby considerably reducing the waiting time of a refresh transaction at a slave or master/slave node. Second, we show that without sacrificing correctness, the principle of our refreshment algorithm can be combined with immediate update propagation strategies, as they were presented in [28]. Performance measurements, reported in Section 6, will demonstrate the value of this optimization.

5.1. *Eliminating the deliver time*

There are cases where the waiting time associated with the deliver time of a refresh transaction can be eliminated. For instance, consider a multinational investment bank that has traders in several cities, including New York, London, and Tokyo. These traders update a local database of positions (securities held and quantity), which is replicated using a lazy master scheme (each site is a master for securities of that site) into a central site that warehouses the common database for all traders. The common database is necessary in order for risk management software to put limits on what can be traded and to support an internal market. A trade will be the purchase of a basket of securities belonging to several sites. In this context, a delay in the arrival of a trade notification may expose the bank to excessive risk. Thus, the time needed to propagate updates from a local site to the common database must be very small (e.g., below a few seconds).

This scheme is a 1slave-per-master configuration, which only requires a global FIFO order to ensure the correctness of its refreshment algorithm (see proposition 3.1). Since, we assume a reliable FIFO multicast network, there is no need for a refresh transaction to wait at a slave node before being executed. More generally, given an arbitrary acceptable configuration, the following proposition characterizes those slave nodes that can process refresh transactions without waiting for their deliver time.

Proposition 5.1. *Let N a node of a lazy master replicated system D. If for any node N' of D, X being the set of common replicas between N and N', we have*:

- *cardinal* $(X) \leq 1$, *or*
- $\forall X_1, X_2, \in X$, *the primary copies of X_1 and X_2 are hold by the same node, then any valid refreshment algorithm meeting the global FIFO order criterion is correct for node N.*

Figure 5 illustrates a configuration meeting Proposition 5.1 for any node of the configuration. For instance, let us consider node 1. The set of replicas hold by both node 1 and node 2 is the singleton r. The set of replicas hold by both node 1 and node 3 is the set r, s, whose primaries are hold by node 1. Node 1 holds no replica in common with another node except node 2 and node 3. Hence node 1 meets Proposition 5.1.

Proof: We proceed by contradiction assuming that an inconsistent state of N can be observed. There is a time t at which a quiescent state of D is reached. There exist a node $N' \in D$, a non-empty set X of replicas hold by both N and N', a time $t_N \leq t$ and a query transaction Q over X such that, for any time $t'_N \leq t$, we have $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. We distinguish two cases:

- *Case 1*: *cardinal (X)* $= 1$. Let $X1$ be the unique replica of X and N'' be the node holding the primary copy of $X1$. By definition, a valid refreshment protocol ensures that any

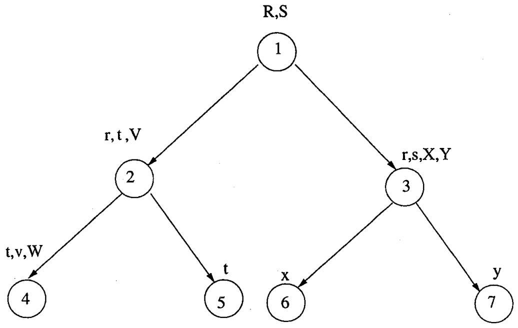


Figure 5. An example of a configuration where global FIFO order is sufficient.

node having a secondary copy of $X1$, commits the refresh transaction associated with a committed transaction updating $X1$ at node N''. The global FIFO order criterion forces nodes N and N' to commit the refresh transactions in the same order as their associated update transactions have committed at node N''. Hence, a contradiction.

- *Case 2*: X contains at least two distinct replicas. In the previous case, we have shown that any secondary copy commits refresh transactions according to the commit order of their associated update transactions at the primary copy. It follows that the different results obtained by a query transaction Q at nodes N and N' come from a misordering of two transactions commits. Let $X1$ and $X2$ the two distinct replicas of X such that node N commits an update/refreshment of $X1$ before an update/refreshment of $X2$ and node N' commits first the update/refreshment of $X2$ and then the update/refreshment of $X1$. By assumption, the primary copies of $X1$ and $X2$ are hold by the same node N''. The global FIFO order criterion forces nodes N and N' to reproduce the same commit order as node N''. Hence, a contradiction. □

From an implementation point of view, the same refreshment algorithm runs at each node. The behavior of the refreshment algorithm regarding the need to wait or not, is simply conditioned by a local variable. Thus, when the configuration changes, only the value of the variable of each node can possibly change.

5.2. Immediate propagation

We assume that the Propagator and the Receiver both implement an immediate propagation strategy as specified in [28], and we focus here on the Refresher. Due to space limitations, we only present the *immediate-immediate* refreshment algorithm. We have chosen the *immediate-immediate* version because it is the one that provides the best performance compared with *deferred-immediate*, as indicated in [28].

5.2.1. Immediate-immediate refreshment. We detail the algorithm of figure 6. Unlike deferred-immediate refreshment, each element of a pending queue is a message m that carries an operation o of some refresh transaction, and a timestamp C. Initially, all pending queues are empty. Upon arrival of a new message m in some pending queue, signaled by an event, the Refresher reads the message and if m does not correspond to a *commit*, inserts it into the running queue. Thus, any operation carried by m other than commit can be immediately submitted for execution to the local transaction manager. If m contains a *commit* operation then *new_m* is assigned with the commit message that has the smallest C among all messages in the top of all pending queues. Then, *new_m* is compared with *curr_m*. If *new_m* has a smallest timestamp than *curr_m*, then *curr_m* is assigned with *new_m*. Afterwards, the Refresher calculates the *deliver_time* for *curr_m*, and timer is set as in the *deferred-immediate* case. Finally, when the timer expires, the Refresher writes *curr_m* into the running queue, dequeues it from its pending queue, sets the timer to inactive and re-executes Step 1.

```
Immediate_immediate Refresher
input: pending queues $q_1 ... q_n$
output: running queue
variables:
   $curr\_m$, $new\_m$: messages from pending queues;
   timer: local reverse timer whose state is either active or inactive;
begin
   timer.state = inactive;
   $curr\_m = new\_m = \emptyset$;
   repeat
      on message arrival or change of timer's state to inactive do
         if $m \neq commit$ then
            write $m$ into the running queue;
            dequeue $m$ from its pending queue;
         else
            $new\_m$ ← commit message with min $C$
               among top messages of $q_1 ... q_n$;
            if $new\_m \neq curr\_m$ then
               $curr\_m \leftarrow new\_m$;
               calculate $deliver\_time(curr\_m)$;
               timer.value ← $deliver\_time(curr\_m) - local\_time$
               timer.state ← active;
            endif
         endif
      on timer.value = 0 do
         write $curr\_m$ into running queue;
         dequeue $curr\_m$ from its pending queue;
         timer.state ← inactive;
   for ever
end
```

Figure 6. Immediate-immediate refreshment algorithm.

5.2.2. Algorithm correctness. Like the deferred-immediate refreshment algorithm, the immediate-immediate algorithm enforces refresh transactions to commit in the order of their associated update transactions. Thus, the proofs of correctness for any acceptable configuration are the same for both refreshment algorithms.

6. Performance evaluation

In this section, we summarize the main performance gains obtained by an *immediate-immediate* refreshment algorithm against a *deferred-immediate* one. More extensive performance results are reported in [28]. We use a simulation environment that reflects as much as possible a real replication context. We focus on a bowtie configuration which requires the use of a $Max + \varepsilon$ deliver time, as explained in Section 5.2. However, once we have fixed the time spent to reliably multicast a message, we can safely run our experiments with a single slave and several masters.

Our simulation environment is composed of *Master, Network, Slave* modules and a database server. The Master module implements all relevant capabilities of a master node such as log monitoring and message propagation. The Network module implements the most significant factors that may impact our update propagation strategies such as the delay to reliably multicast a message. The Slave module implements the most relevant components of the slave node architecture such as Receiver, Refresher and Deliverer. In addition, for performance evaluation purposes, we add the Query component in the slave module, which implements the execution of queries that read replicated data. Since, we do not consider node failures, the reception log is not taken into account. Finally, a database server is used to implement refresh transactions and query execution.

Our environment is implemented on a Sun Solaris workstation using Java/JDBC as the underlying programming language. We use sockets for inter-process communication and Oracle 7.3 to implement refresh transaction execution and query processing. For simulation purposes, each write operation corresponds to an UPDATE command that is submitted to the server for execution.

6.1. Performance model

The metrics used to compare the two refreshment algorithms is given by the freshness of secondary copies at the slave node. More formally, given a replica X, which is either a secondary or a primary copy, we define $n(X, t)$ as the number of committed update transactions on X at global time t. We assume that update transactions can have different sizes but their occurrence is uniformly distributed over time. Using this assumption, we define the degree of freshness of a secondary copy r at global time t as:

$$f(r, t) = n(r, t)/n(R, t);$$

Therefore, a degree of freshness close to 0 means bad data freshness while close to 1 means excellent. The mean degree of freshness of r at a global time T is defined as:

$$mean_f = 1/T \int_0^T f(r, t)\, dt$$

Table 1. Performance parameters.

Parameters	Definition	Values
λ_t	mean time interval between Trans.	*bursty: (mean* $= 200$*ms)*
λ_q	mean time interval between Queries	*low (mean* $= 15$*s)*
nbmaster	Number of Master nodes	1 to 8
$\|Q\|$	Query Size	5
$\|RT\|$	Refresh Transaction Size	5; 50
ltr	Long Transaction Ratio	0; 30%; 60%; 100%
abr	Abort Ratio	0; 5%; 10%; 20%
t_{short}	Multicast Time of a single record	20 ms and 100 ms

We now present the main parameters for our experimentations summarized in Table 1. We assume that the mean time interval between update transactions, noted λ_t, as reflected by the history log of each master, is bursty. Updates are done on the same attribute (noted *attr*) of a different tuple. We focus on *dense* update transactions, i.e., transactions with a small time interval between each two writes. We define two types of update transactions.

Small update transactions have size 5 (i.e., 5 write operations), while long transactions have size 50. We define four scenarios in which the proportion of long transactions, noted *ltr*, is set respectively to 0, 30, 60, and 100. Thus, in a scenario where $ltr = 30$, 30% of the executed update transactions are long. Finally, we define an abort transaction ratio, noted *abr*, of 0, 5%, 10%, 20%, that corresponds to the percentage of transactions that abort in an experiment. Furthermore, we assume that a transaction abort always occurs after half of its execution. For instance $abr = 10\%$ means that 10% of the update transactions abort after the execution of half of their write operations.

Network delay is calculated by $\delta + t$, where δ is the time between the insertion of a message in the input queue of the Network module and the multicast of the message by that module, and t is the reliable multicast time of a message until its insertion in the pending queue of the Refresher. Concerning the value of t used in our experiments, we have a short message multicast time, noted t_{short}, which represents the time needed to reliably multicast a single log record. In addition, we consider that the time spent to reliably multicast a sequence of log records is linearly proportional to the number of log records it carries. The network overhead delay, δ, takes into account the time spent in the input queue of the Network, it is implicity modeled by the system overhead to read from and write to sockets. The *Total propagation time* (noted t_p) is the time spent to reliably multicast all log records associated with a given transaction. Thus, if n represents the size of the transaction with immediate propagation, we have $t_p = n \times (\delta + t_{short})$, while with deferred propagation, we have $t_p = (\delta + n \times t_{short})$. Network contention occurs when δ increases due to the increase of network traffic. In this situation, the delay introduced by δ may impact the total propagation time, especially with immediate propagation. Finally, when $ltr > 0$, the value of *Max* is calculated using the maximum time spent to reliably multicast a long transaction $(50 * t_{short})$. On the other hand, when $ltr = 0$, the value of *Max* is calculated using the maximum time spent to reliably multicast a short transaction $(50 * t_{short})$.

The refresh transaction execution time is influenced by the existence of possible conflicting queries that read secondary copies at the slave node. Therefore, we need to model queries. We assume that the mean time interval between queries is low, and the number of data items read is small (fixed to 5). We fix a 50% conflict rate for each secondary copy, which means that each refresh transaction updates 50% of the tuples of each secondary copy that are read by a query.

To measure the mean degree of freshness, we use the following variables. Each time an update transaction commits at a master, variable *version_master* for that master, is incremented. Similarly, each time a refresh transaction commits at the slave, variable *version_slave*, is incremented. Whenever a query conflicts with a refresh transaction we measure the degree of freshness.

6.2. *Experiments*

We present three experiments. The results are average values obtained from the execution of 40 update transactions. The first experiment shows the mean degree of freshness obtained for the *bursty* workload. The second experiment studies the impact on freshness when update transactions abort. In the third experiment, we verify the freshness improvement of each strategy when the network delay to propagate a message increases.

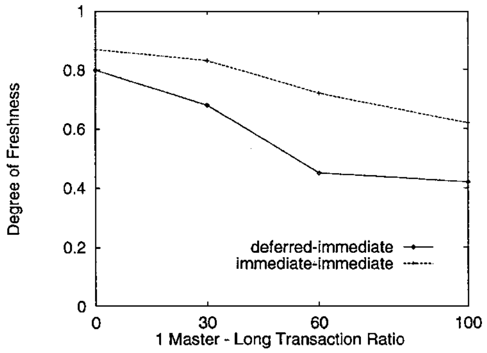


Figure 7. Bursty workload—response time.

We now summarize and discuss the major observations of our experiments. As depicted in figure 7, when $ltr = 0$ (short transactions), the mean degree of freshness is already impacted because on average, $\lambda_t < t_p$. Therefore, during T_i's update propagation, T_{i+1}, $T_{i+2} \cdots T_{i+n}$ may be committed. Notice that even with the increase of network contention, *immediate-immediate* yields a better mean degree of freshness. With 2, 4, and 8 masters, the results of *immediate-immediate* are much better than those of *deferred-immediate*, as ltr increases (see figure 8). For instance, with 4 masters with $ltr = 30$, the mean degree of freshness is 0.62 for *immediate-immediate* and 0.32 for *deferred-immediate*. With 6 masters and $ltr = 60$, the mean degree of freshness is 0.55 for *immediate-immediate*, and 0.31 for *deferred-immediate.* In fact, *immediate-immediate* always yields the best mean degree of freshness even with network contention due to the parallelism of log monitoring, propagation, and refreshment.

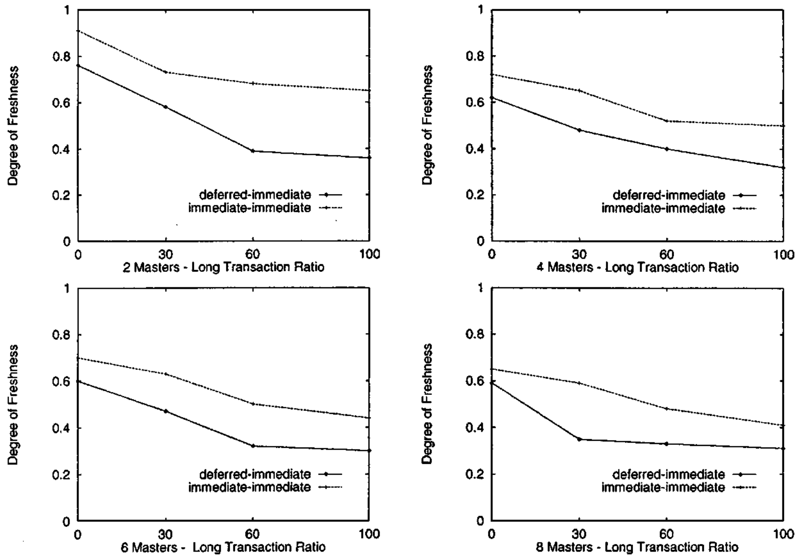


Figure 8. Bursty workload—mean degree of freshness.

With *immediate-immediate*, the mean query response time may be seriously impacted because each time a query conflicts with a refresh transaction, it may be blocked during a long period of time since the propagation time may be added to the refresh transaction execution time. When $\lambda_q \gg \lambda_t$, the probability of conflicts is quite high. There, the network delay to propagate each operation has an important role and the higher its value, the higher are the query response times in conflict situations. However, we verified that the use of a multiversion protocol on the slave node may significantly reduces query response times, without a significant decrease in the mean degree of freshness.

The *abort* of an update transaction with *immediate-immediate* does not impact the mean degree of freshness since the delay introduced to undo a refresh transaction is insignificant compared to the propagation time. As shown in figure 9, for $ltr = 0$ and various values of *abr* (5, 10, 20), the decrease of freshness introduced by update transactions that abort with *immediate-immediate* is insignificant. In the worst case, it achieves 0.2. This behavior is the same for other values of *ltr* (30, 60, 100).

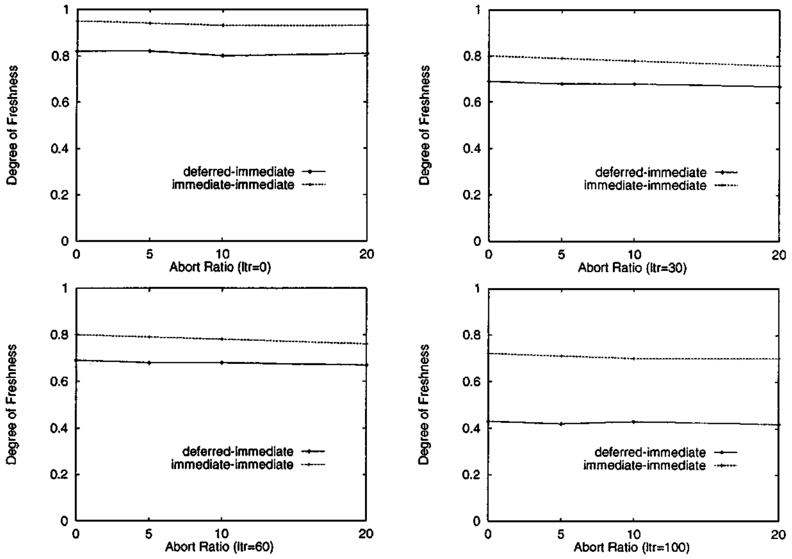


Figure 9. Bursty workload—*Abort* Effects

Finally, the improvements brought by *immediate-immediate* are more significant when the network delay to propagate a single operation augments. Figure 10 compares the freshness results obtained when $\delta = 100$ ms and $\delta = 20$ ms. For instance, when $\delta = 20$ and $ltr = 100$ *immediate-immediate* improves 1.1 times better than *deferred-immediate* and when $\delta = 100$, *immediate-immediate* improves 5 times better. This clearly shows the advantages of having tasks being executed in parallel.

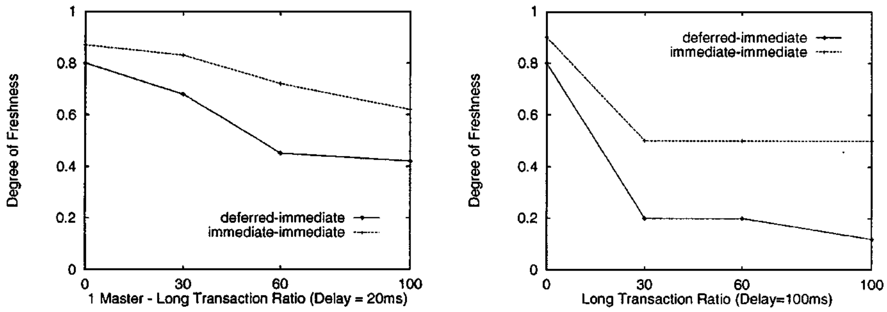


Figure 10. Increase of network delay—mean degree of freshness.

7. Related work

Apart from the work cited in Section 1, the closest work to ours is in [9]. The authors show that for any strongly acyclic configuration a refreshment algorithm which enforces a global FIFO ordering, guarantees a global serializability property, which is similar to our notion of correction. Their result is analogous to our Proposition 3.1. They also propose an algorithm, which assigns, when it is possible, a site to each primary copy so that the resulting configuration is strongly acyclic. However, no algorithm is provided to refresh secondary copies in the cases of non strongly acyclic configurations.

In [6] the authors propose two new lazy update protocols (DAG(WT) and DAG(T)) that ensure serializabity for acyclic configurations and imposes a much weaker requirement on data placement than [9]. Close to our approach, the DAG(T) protocol is based on timestamp ordering. Notice, however that there timestamp is based on logical clocks.

Much work has been devoted to the maintenance of integrity constraints in federated or distributed databases, including the case of replicated databases [7, 11, 19, 21]. These papers propose algorithms and protocols to prevent the violation of certain kind of integrity constraints by local transactions. However, their techniques are not concerned with the consistent refreshment of replicas.

Table 2. Replication schemes.

Replication scheme	Ownership	Propogation	Refreshment
A	Group	Deferred Immediate	Immediate (Reconcilation)
B	Master	Deferred Immediate Periodic	Immediate On Demand Group Periodic

Table 2 summarizes the two major lazy replication schemes and their basic parameters.

Replication scheme A corresponds to lazy replication where all replica copies are updatable (*update anywhere*). In this case, there is *group ownership* of the replicas. The common update propagation strategy for this scheme is *deferred-immediate.* However, *conflicts* may occur if two or more nodes update the same replica copy. Policies for conflict detection and resolution [16, 5] can be based on timestamp ordering, node priority and other strategies. The problem with conflict resolution is that during a certain period of time, the database may be in an inconsistent state.

For instance, in [30], the authors describe a lazy group replication scheme in which the update propagation protocol applies updates to replicated data in their arrival order, possibly restoring inconsistencies when arrivals violate the timestamp ordering of transactions. Notice that conflicts cannot be avoided but their detection may happen earlier by using an immediate propagation.

The timestamp message delivery protocol in [15] implements eventual delivery for a lazy group replication scheme [18]. It uses periodic exchange of messages between pairs of servers that propagate messages to distinct groups of master nodes. At each master node incoming messages are stored in a history log (as initially proposed in [23]) and later delivered to the application in a defined order. Eventual delivery is not appropriate in our framework since we are interested in improving data freshness.

Replication scheme B is the focus of our work. There are several refreshment strategies for this replication scheme. With *on demand* refreshment, each time a query is submitted for execution, secondary copies that are read by the query are refreshed by executing all the refresh transactions that have been received. Therefore, a delay may be introduced in query response time. When *group* refresh is used, refresh transactions are executed in groups according to the application's freshness requirements. With the *periodic* approach, refreshment is triggered at fixed intervals. At refreshment time, all received refresh transactions are executed. Finally, with *periodic propagation*, changes performed by update transactions are stored in the master and propagated periodically towards the slaves. Immediate propagation may be used with all refreshment strategies.

The stability and convergence of replication schemes A and B are compared in [18] through an analytical model. The authors show that scheme A has unstable behavior as the workload scales up and that using scheme B reduces the problem. They introduce several concepts such as lazy master and ownership which we use. They also explore the use of mobility.

The goal of epidemic algorithms [33] is to ensure that all replicas of a single data item converge to the same value in a lazy group replication scheme. Updates are executed locally at any node. Later, nodes communicate to exchange up-to-date information. In our approach, updates are propagated from each primary copy towards its secondary copies.

Formal concepts for specifying coherency conditions in a replicated distributed database have been introduced in [12]. The authors focus on a *deferred-immediate* update propagation strategy and propose concepts for computing a measure of relaxation.[4] Their concept of *version* is closely related to our notion of freshness.

Oracle 7.3 [24] implements *event-driven* replication. Triggers on the master tables make copies of changes to data for replication purposes, storing the required change information in tables called *queues* that are periodically propagated. Sybase 10 replication server replicates transactions, not tables, across nodes in the network. The *Log Transfer Manager* implements log monitoring like our approach. However, these systems do not implement immediate propagation and there is no multi-queue scheme for refreshment.

8. Conclusion

In a lazy master replicated system, a transaction can commit after updating one replica copy (primary copy) at some node. The updates are propagated towards the other replicas (secondary copies), and these replicas are refreshed in separate refresh transactions.

We proposed refreshment algorithms which address the central problem of maintaining replicas' consistency. An observer of a set of replicas at some node never observes a state which is never seen by another oberver of the same set of replicas at another node.

This paper has three major contributions. Our first contribution is a formal definition of (i) the notion of correct refreshment algorithm and (ii) correctness criteria for any acceptable configuration.

Our second contribution is an algorithm meeting these correctness criteria for any acceptable configuration. This algorithm can be easily implemented over an existing database system. It is based on a deferred update propagation, and it delays the execution of a refresh transaction until its deliver time.

Our third contribution concerns optimizations of the refreshment algorithm in order to improve the data freshness. With the first optimization, we characterized the nodes that do not need to wait. The second optimization uses *immediate-immediate* update propagation strategy. This strategy allows parallelism between the propagation of updates and the execution of the associated refresh transactions.

Finally, our performance evaluation shows that the *immediate-immediate* strategy always yields the best mean degree of freshness for a bursty workload.

9. Appendix

In this appendix, we detail the proofs of the four propositions given respectively in Sections 3.1, 3.2 and 3.3. These propositions define correctness criteria of a refreshment algorithm for different acyclic configurations. Section 9.1 deals with an acyclic configuration which is neither a bowtie nor a triangular configuration. Section 9.2 deals with a bowtie configuration which is not a triangular one. Finally, Section 9.3 deals with a view triangular configuration.

9.1. Correctness criterion for an acyclic configuration which is neither a bowtie nor a triangular configuration

As said in Section 3.1, the correctness criterion for an acyclic configuration which is neither a bowtie nor a triangular configuration, is the following one:

Proposition 3.1. *If a lazy master replicated system D has an acyclic configuration which is neither a bowtie nor a triangular configuration and D uses a valid refreshment algorithm meeting the global FIFO order criterion, then this refreshment algorithm is correct.*

To prove this proposition, we proceed by step. We first establish three preliminary lemmas. The first one deals with each replica copy taken individually. The second one shows that if the correctness criterion is violated, then necessarily there are two replicas such their updates have been committed in a different order by two nodes. The last lemma enounces all the possible cases for two replicas shared by two nodes. Finally, we prove the proposition.

We now enounce the first lemma. We consider each replica copy individually and show that at each replica copy, updates are committed in the order they have been committed at the primary copy.

Lemma 9.1. *In a lazy master replicated system D, using a valid refreshment protocol meeting the global FIFO order criterion, let t be any global time at which a quiescent state of D is reached. For any node N, for any replica copy X hold by N, for any node N' holding X, for any query transaction Q over the only replica copy X, for any time $t_N \leq t$, there exists a time $t_{N'} \leq t$ such that $I_{t_N}[N](Q) = I_{t_{N'}}[N'](Q)$.*

Proof: We proceed by contradiction. We assume that there is a time $t_N \leq t$ such that for any time $t_{N'} \leq t$ we have:$I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. We distinguish two cases:

- either N holds the primary copy of X. Hence the query over X at node N reflects all the update transactions committed before t_N. According to the validity property, all the associated refresh transactions will be committed at nodes having a secondary copy of X. Moreover the global FIFO order criterion forces the refresh transactions to be committed in the order of their associated update transactions. Hence a contradiction.
- or N holds a secondary copy of X. If N' holds the primary copy of X, by analogy with the previous case, we obtain a contradiction. If now N' holds a secondary copy of X, then by the validity property and by the global FIFO order criterion, all the nodes having a secondary copy of X must reflect all the updates transactions committed before t_N and in the order they have been committed on the primary copy of X. Hence a contradiction.

□

We now show that if a query transaction over a common set X of replicas gives different results at two nodes N and N', then there exist two replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

Lemma 9.2. *In a lazy master replicated system D, using a valid refreshment protocol meeting the global FIFO order criterion, let t be any global time at which a quiescent state of D is reached. If there are nodes N and N', a non-empty set X of replica copies hold by N and N', a query transaction Q over X, a time $t_N \leq t$ such that for any time $t_{N'} \leq I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$, then there are two distinct replicas $X1$ and $X2$ in X such that their updates/refreshes have been committed in a different order by nodes N and N'.*

Proof: We assume that there are nodes N and N', a non-empty set X of replica copies hold by N and N', a query transaction Q over X, a time $t_N \leq t$ such that for any time $t_{N'} \leq t$ $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. From lemma 9.1, this is impossible if the cardinal of X is one. Hence we assume that X contains at least two distinct replicas. If the results of Q over X

differ at nodes N and N', it means that the transactions having updated/refreshed the replicas in X have committed in a different order at nodes N and N'. Hence there are two distinct replicas $X1$ and $X2$ in X, with $X1 \neq X2$, such that node N commits an update/refresh of $X1$ before an update/refresh of $X2$, and node N' commits an update/refresh of $X2$ before an update/refresh of $X1$. Hence the lemma. □

We now consider all the possible cases for two nodes of an acyclic configuration, holding both at least two replica copies.

Lemma 9.3. *In an acyclic configuration of a lazy master replicated system D, for any two distinct nodes N and N' holding both two distinct replica copies $X1$ and $X2$, the only possible cases are*:

1. *either N has the primary copies of both $X1$ and $X2$; N' is a slave of N;*
2. *or N' has the primary copies of both $X1$ and $X2$; N is a slave of N';*
3. *or both N and N' have secondary copies of both $X1$ and $X2$.*
4. *or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$.*
5. *or N' has the primary copy of $X1$ and a secondary copy of $X2$; N has secondary copies of both $X1$ and $X2$.*
6. *or N has the primary copy of $X2$ and a secondary copy of $X1$; N' has secondary copies of both $X1$ and $X2$.*
7. *or N' has the primary copy of $X2$ and a secondary copy of $X1$; N has secondary copies of both $X1$ and $X2$.*

Proof: We consider all the possible cases for two distinct nodes N and N' holding both a replica copy of $X1$ and a replica copy of $X2$ with $X1 \neq X2$. We have 16 possible cases for the attribution of the primary/secondary copy of $X1$ and $X2$ to N and N'. Each case can be coded with four bits with the following meaning:

- the first bit is one if N holds the primary copy of $X1$ and zero otherwise;
- the second bit is one if N holds the primary copy of $X2$ and zero otherwise;
- the third bit is one if N' holds the primary copy of $X1$ and zero otherwise;
- the fourth bit is one if N' holds the primary copy of $X2$ and zero otherwise;

Among them, seven are impossible, because for any replica, only one node holds the primary copy. Hence, the impossible cases are 1010, 1110, 1011, 1111, 0101, 0111, 1101.

We now prove that the cases 1001 and 0110 are impossible. Let us consider the case 1001 where N holds the primary copy of $X1$ and N' holds the primary copy of $X2$. We then have N is a slave of N' (because of $X2$) and N' is a slave of N (because of $X1$). The configuration is then cyclic: a contradiction with our assumption. By analogy, the case 0110 is impossible.

Hence there are $16 - 7 - 2 = 7$ only possible cases, which are given in the lemma. □

We can now prove the proposition given at the beginning of Section 9.1.

Proof of Proposition 3.1: We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query transaction Q over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](Q)$. From Lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by Lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.
2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.
3. or both N and N' have secondary copies of both $X1$ and $X2$. Notice that this case is impossible in a 1slave-per-master configuration (both N and N' would be slave of the node holding the primary copy of X1). Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. Since the configuration is not a bowtie configuration, we necessarily have $N1 = N2$. The global FIFO order criterion enforces both N and N' to commit the refresh transactions in the order their associated update transactions have been committed by $N1 = N2$. Hence a contradiction.
4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. This case and all the following ones would lead to a triangular configuration. Hence, a contradiction. □

9.2. *Correctness criterion for an acyclic configuration which is a bowtie configuration but not a triangular one*

As said in Section 3.2, the correctness criterion for an acyclic configuration which is a bowtie configuration but not a triangular one, is the following one:

Proposition 3.2. *If a lazy master replicated system D that has a bowtie configuration but not a triangular configuration, uses a valid refreshment algorithm meeting the global FIFO order and the total order criteria, then this refreshment algorithm is correct.*

Proof of Proposition 3.2: We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](P) \neq I_{t_{N'}}[N'](P)$. From Lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by Lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.
2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.

3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. The total order criterion enforces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the same order. Hence, a contradiction.
4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. This case and all the following ones would lead to a triangular configuration. Hence, a contradiction. □

9.3. *Correctness criterion for an acyclic configuration which is a view triangular one*

As said in Section 3.3, the correctness criterion for an acyclic configuration which is a view triangular one but not a bowtie configuration, is the following one:

Proposition 3.3. *If a lazy master replicated system D that has a view triangular configuration but not a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order and the master/slave induced order criteria*, *then this refreshment algorithm is correct.*

Proof of Proposition 3.3: We proceed by contradiction. There exist a node N and a node N' with $N = N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](P) \neq I_{t_{N'}}[N'](P)$. From Lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by Lemma 9.3.

1. either N has the primary copies of both $X1$ and X2; N' is a slave of N; The global FIFO order criterion enforces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.
2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.
3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. Since the configuration is not a bowtie configuration, we necessarily have $N1 = N2$. The global FIFO order criterion forces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the order their associated update transactions have been committed by $N1 = N2$. Hence, a contradiction.
4. or N has the primary copy of $X1$ and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. As the configuration is a view triangular one, $X1$ is a materialized view from local secondary copies. The master/slave induced order criterion enforces nodes N and N' to commit the refresh transactions of $X1$ and $X2$ in the same order. Hence, a contradiction. □

If now, the lazy master replicated system has both a view triangular configuration and a bowtie configuration, then the correctness criterion becomes:

Proposition 3.4. *If a lazy master replicated system D having both a view triangular configuration and a bowtie configuration, uses a valid refreshment algorithm meeting the global FIFO order, the master/slave induced order and the total order criteria then this refreshment algorithm is correct.*

Proof of Proposition 3.4: We proceed by contradiction. There exist a node N and a node N' with $N \neq N'$ such that X the set of replica hold by both N and N' is non empty, there exist a query program P over X and a time $t_N \leq t$ such that $\forall t'_N \leq t$, we have $I_{t_N}[N](Q) \neq I_{t_{N'}}[N'](P)$. From Lemma 9.2, there exist two distinct replicas $X1$ and $X2$ in X such that their updates have been committed in a different order by nodes N and N'.

We will now consider all the cases given by Lemma 9.3.

1. either N has the primary copies of both $X1$ and $X2$; N' is a slave of N; The global FIFO order criterion forces N' to commit the refresh transactions in the order their associated update transactions have been committed by N. Hence a contradiction.
2. or N' has the primary copies of both $X1$ and $X2$; N is a slave of N'; This is the symmetrical case of the previous one. We then obtain a contradiction.
3. or both N and N' have secondary copies of both $X1$ and $X2$. Let $N1$ be the node holding the primary copy of $X1$. Let $N2$ be the node holding the primary copy of $X2$. The total order criterion forces both N and N' to commit the refresh transactions on $X1$ and $X2$ in the same order. Hence, a contradiction.
4. or N has the primary copy of X1 and a secondary copy of $X2$; N' has secondary copies of both $X1$ and $X2$. There exists a node $N2$ holding the primary copy of $X2$ such that N and N' are slaves of $N2$ and N' is slave of N. As the configuration is a view triangular one, $X1$ is a materialized view from local secondary copies. The master/slave induced order criterion enforces nodes N and N' to commit the refresh transactions of $X1$ and $X2$ in the same order. Hence, a contradiction.
5. this case and all the following ones are symmetrical to the previous one. □

Notes

[1] From now on, we suppose that replicas are relations.

[2] This frequent situation typically arises when no corporate data warehouse has been set up between data sources and data marts. Quite often, each data mart, no matter how focused, ends up with views of the business that overlap and conflict with views held by other data marts (e.g., sales and inventory data marts). Hence, the same relations can be replicated in both data marts [20].

[3] A replication set can be reduced to a singleton if there exists a single copy of a relation in the replicated system.

[4] called *coherency index*.

References

1. D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi, “Exploiting atomic broadcast in replicated databases”, in EURO-PAR Int. Conf. on Parallel Processing, August 1997.

2. R. Alonso, D. Barbara, and H. Garcia-Molina, “Data caching issues in an information retrieval system,” ACM Transactions on Database Systems, vol. 15, no. 3, pp. 359–384, 1990.
3. G. Alonso and A. Abbadi, “Partitioned data objects in distriduted databases,” Distributed and Parallel Databases, vol. 3, no. 1, pp. 5–35, 1995.
4. P.A. Bernstein and E. Newcomer, Transaction Processing, Morgan Kaufmann, 1997.
5. S. Bobrowski, Oracle 7 server Concepts, release 7.3, Oracle Corporation, Redwood City, CA, 1996.
6. Y. Breitbart, R. Komondoor, R. Rastogi, and S. Seshadri, “Update propagation protocols for replicated databases,” in ACM SIGMOD Int. Conference on Management of Data, Philadelphia, PA, May 1999.
7. S. Ceri and J. Widom, “Managing semantic heterogeneity with production rules and persistent queues,” in Int. Conf. on VLDB, 1993.
8. S. Chaudhuri and U. Dayal, “An overview of data warehousing and OLAP technology,” ACM SIGMOD Record, vol. 26, no. 1, 1997.
9. P. Chundi, D.J. Rosenkrantz, and S.S. Ravi, “Deferred updates and data placement in distributed databases,” in Int. Conf. on Data Engineering (ICDE), Louisiana, February 1996.
10. A. Downing, I. Greenberg, and J. Peha, “OSCAR: A system for weak-consistency replication,” in Int. Workshop on Management of Replicated Data, Houston, November 1990, pp. 26–30.
11. L. Do and P. Drew, “The management of interdependent asynchronous transactions in heterogeneous database environments,” in Int. Conf, on Database Systems for Advanced Applications, 1995.
12. R. Gallersdorfer and M. Nicola, “Improving performance in replicated databases through relaxed coherency,” in Int. Conf. on VLDB, Zurich, September 1995.
13. L. George and P. Minet, “A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints,” in Int. Conf. on Distributed Computing Systems (ICDCS97), Baltimore, May 1997.
14. L. George and P. Minet, “A uniform reliable multicast protocol with guaranteed responses times,” in ACM SIGPLAN Workshop on Languages Compilers and Tools for Embedded Systems, Montreal, June 1998.
15. R. Goldring, “Weak-consistency group communication and membership,” Ph. D. Thesis, University of Santa Cruz, 1992.
16. R. Goldring, “Things every update replication customer should know,” in ACM SIGMOD Int. Conf. on Management of Data, San Jose, June 1995.
17. J.N. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan-Kaufmann, 1993.
18. J. Gray, P. Helland, P. O’Neil, and D. Shasha, “The danger of replication and a solution,” in ACM SIGMOD Int. Conf. on Management of Data, Montreal, June 1996.
19. P. Grefen and J. Widom, “Protocols for integrity constraint checking in federated databases,” Distributed and Parallel Databases, vol. 5, no. 4, 1997.
20. P.S. Group, “The Data mart option,” Open Information Systems, vol. 11, no. 7, 1996.
21. H. Gupta, I.S. Mumick, and V.S. Subrahmanian, “Maintaining views incremently,” in ACM SIGMOD Int. Conference on Management of Data, Washington DC, May 1993.
22. V. Hadzilacos and S. Toueg, “A modular approach to fault-tolerant broadcasts and related problems,” Dept. of Computer Science, Cornell University, Ithaca, NY 14853, Technical Report TR 94–1425, 1994.
23. B. Kahler and O. Risnes, “Extending logging for database snapshot refresh,” in Int. Conf. on VLDB, Brighton, September 1987.
24. A.A. Helal, A.A. Heddaya, and B.B. Bhargava, Replication Techniques in Distributed Systems, Kluwer Academic Publishers, 1996.
25. A. Moissis, Sybase Replication Server: A Pratical Architecture for Distributing and Sharing Corporate Information, replication server white paper, http://www.sybase.com 1996.
26. M. Nicola and M. Jarke, “Increasing the expressiveness of analytical performance models for replicated databases,” in Int. Conference on Database Theory (ICDT), Jerusalem, January 1999.
27. M.T. Özsu and P. Valduriez, Principles of Distributed Database Systems, 2nd edn, Prentice Hall, 1999.
28. E. Pacitti, E. Simon, and R. de Melo, “Improving data freshness in lazy master schemes,” in Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.
29. E. Pacitti, P. Minet, and E. Simon, “Fast algorithms for maintaning replica consistency in lazy master replicated databases,” in Int. Conf. on VLDB, 1999.
30. S.K. Sarin, C.W. Kaufman, and J.E. Somers, “Using history information to process delayed database updates,” in Int. Conf. on VLDB, Kyoto, August 1986.

31. A. Sheth and M. Rusinkiewicz, “Management of interdependent data: Specifying dependency and consistency requirements,” in Int. Workshop on Management of Replicated Data, Houston, November 1990.
32. I. Stanoi, D. Agrawal, and A. El Abbadi, “Using broadcast primitives in replicated databases,” in Int. Conf. on Distributed Computing Systems (ICDCS98), Amsterdam, May 1998.
33. D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser, “Managing update conflicts in Bayou, a weakly connected replicated storage system,” in Symposium on Operating System Principles (SIGOPS), Colorado, December 1995.
34. K. Tindell and J. Clark, “Holistic schedulability analysis for distributed hard real-time systems,” Microprocessors and Microprogramming, Euromicro Journal (Special issue on parallel embedded real-time systems, vol. 40, pp. 117–134, 1994.
35. Y. Zhuge, H. Garcia-Molina, and J.L. Wiener, “View maintenance in a warehouse environment,” in ACM SIGMOD Int. Conf. on Management of Data, 1995.

ANNEX C

Preventive Replication in a Database Cluster

E. Pacitti, C. Coulon, P. Valduriez, T. Özsu

***Distributed and Parallel Databases*, Vol. 18, No. 3, 2005, 223-251**

Distributed and Parallel Databases, 18, 223–251, 2005

DOI: 10.1007/s10619-005-4257-4

Preventive Replication in a Database Cluster*

ESTHER PACITTI pacitti@lina.univ-nantes.fr
CÉDRIC COULON coulon@lina.univ-nantes.fr
PATRICK VALDURIEZ patrick.valduriez@inria.fr
INRIA and LINA, University of Nantes, France

M. TAMER ÖZSU tozsu@uwaterloo.ca
University of Waterloo, Canada

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

Keywords: database cluster, partial replication, preventive replication, strong consistency, TPC-C benchmarking

1. Introduction

High-performance and high-availability of database management have been traditionally achieved with parallel database systems [23], implemented on tightly-coupled multiprocessors. Parallel data processing is then obtained by partitioning and replicating the data across the multiprocessor nodes in order to divide processing. Although quite effective, this solution requires the database system to have full control over the data and is expensive in terms of software and hardware.

Clusters of PC servers now provide a cost-effective alternative to tightly-coupled multiprocessors. They have been used successfully by, for example, Web search engines using high-volume server farms (e.g., Google). However, search engines are typically read-intensive, which makes it easier to exploit parallelism. Cluster systems can make new businesses such as Application Service Providers (ASP) economically viable. In the ASP model, customers' applications and databases (including data and DBMS) are hosted at

*Work partially funded by the MDP2P project of the ACI "Masses de Donniées" of the French Ministry of Research.

the provider site and need be available, typically through the Internet, as efficiently as if they were local to the customer site. Notice that due to autonomy, it is possible that the DBMS at each node are heterogeneous. To improve performance, applications and data can be replicated at different nodes so that users can be served by any of the nodes depending on the current load [1]. This arrangement also provides high-availability since, in the event of a node failure, other nodes can still do the work. However, managing data replication in the ASP context is far more difficult than in Web search engines since applications can be update-intensive and both applications and databases must remain autonomous. The solution of using a parallel DBMS is not appropriate as it is expensive, requires heavy migration to the parallel DBMS and hurts database autonomy.

In this paper, we consider a database cluster with similar nodes, each having one or more processors, main memory (RAM) and disk. Similar to multiprocessors, various cluster system architectures are possible: shared-disk, shared-cache and shared-nothing [23]. Shared-disk and shared-cache require a special interconnect that provide a shared space to all nodes with provision for cache coherence using either hardware or software. Shared-nothing (or distributed memory) is the only architecture that supports our autonomy requirements without the additional cost of a special interconnect. Furthermore, shared-nothing can scale up to very large configurations. Thus, we strive to exploit a shared-nothing architecture.

The major problem of data replication is to manage the consistency of the replicas in the presence of updates [6]. The basic solution in distributed systems that enforces strong replica consistency[1] is synchronous (or eager) replication (typically using the Read-One-Write All—ROWA protocol [11]). Whenever a transaction updates a replica, all other replicas are updated inside the same distributed transaction. Therefore, the mutual consistency of the replicas is enforced. However, synchronous replication is not appropriate for a database cluster for two main reasons. First, all the nodes would have to homogeneously implement the ROWA protocol inside their local transaction manager, thus violating DBMS autonomy. Second, the atomic commitment of the distributed transaction should rely on the two-phase commit (2PC) protocol [11] which is known to be blocking (i.e. does not deal well with nodes' failures) and has poor scale up.

A better solution that scales up is lazy replication [14], where a transaction can commit after updating a replica, called *primary copy*, at some node, called *master node*. After the transaction commits, the other replicas, called *secondary copies*, are updated in separate refresh transactions at *slave nodes*. Lazy replication allows for different replication configurations [12]. A useful configuration is *lazy master* where there is only one primary copy. Although it relaxes the property of mutual consistency, strong consistency is assured. However, it hurts availability since the failure of the master node prevents the replica to be updated. A more general configuration is (*lazy*) *multi-master* where the same primary copy, called a multi-owner copy, may be stored at and updated by different master nodes, called multi-owner nodes. The advantage of multi-master is high-availability and high-performance since replicas can be updated in parallel at different nodes. However, conflicting updates of the same primary copy at different nodes can introduce replica incoherence.

Preventive replication [13] is an asynchronous solution that enforces strong consistency. Instead of using atomic broadcast, as in synchronous group-based replication [9], preventive replication uses First-In First-Out (FIFO) reliable multicast which is a weaker constraint.

It works as follows. Each incoming transaction is submitted, via a load balancer, to the best node of the cluster. Each transaction T is associated with a chronological timestamp value C, and is multicast to all other nodes where there is a replica. At each node, a *delay time d* is introduced before starting the execution of T. This delay corresponds to the upper bound of the time needed to multicast a message. When the delay expires, all transactions that may have committed before C are guaranteed to be received and executed before T, following the timestamp chronological order (i.e. total order). Hence, this approach prevents conflicts and enforces consistency. Its implementation over a cluster of 8 nodes showed good performance [13].

However, the original proposal has two main limitations. First, it assumes that databases are fully replicated across all cluster nodes and thus propagates each transaction to each cluster node. This makes it unsuitable for supporting large databases and heavy workloads on large cluster configurations. Second, it has performance limitations since transactions are performed one after the other, and must endure waiting delays before starting. Thus, refreshment is a potential bottleneck, in particular, in the case of bursty workloads where the arrival rates of transactions are high at times. This paper addresses these important limitations. It is based on the solution initially proposed in [4] with significant extensions regarding replication configurations, concurrency management, proofs of algorithms and performance evaluation.

In this paper, we provide support for partial replication, where databases are partially replicated at different nodes. Unlike full replication, partial replication can increase access locality and reduce the number of messages for propagating updates to replicas. To increase transaction throughput, we propose a refreshment algorithm that potentially eliminates the delay time, and we introduce concurrent replica refreshment. We describe the implementation of our algorithm in our RepDB∗ prototype [19] over a cluster of 64 nodes running the PostgreSQL DBMS. Our experimental results using the TPC-C Benchmark show that it yields excellent scale-up and speed-up.

The rest of the paper is organized as follows. Section 2 introduces the global architecture for processing user requests against applications into the cluster system. Section 3 defines the basic concepts for fully and partial replication. Section 4 describes preventive refreshment for partially replication, including the algorithm and architecture. Section 5 proposes some important optimizations to the refreshment algorithm that improves transaction throughput. Section 6 describes our validation and experimental results. Section 7 discusses related work. Section 8 concludes.

2. Database cluster architecture

In this section, we introduce the architecture for processing user requests against applications into the cluster system and discuss our general solutions for placing applications, submitting transactions and managing replicas. Therefore, the replication layer is identified together with all other general components.

In this paper, we exploit a shared-nothing architecture. This is the only architecture that allows sufficient node autonomy without the additional cost of special interconnects. In our shared-nothing architecture, each cluster node is composed of five layers (see

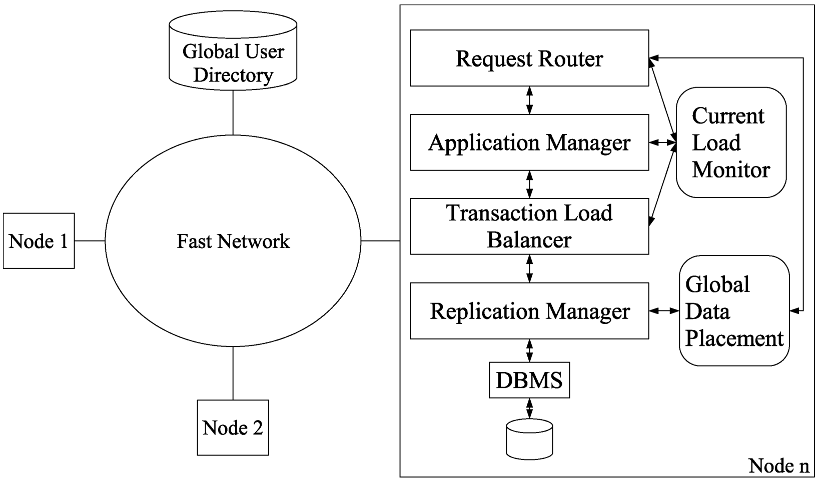


Figure 1. Peer-to-peer cluster architecture.

figure 1): Request Router, Application Manager, Transaction Load Balancer and Replication Manager. A user request may be a query or update transaction on a specific application. The general processing of a user request is as follows.

When a user request arrives at the cluster, traditionally through an access node, it is sent randomly to a cluster node *i*. There is no significant data processing at the access node, avoiding bottlenecks. Within that cluster node, the user is authenticated and authorized through the Request Router, available at each node, using a multi-threaded *global user directory* service. Notice that user requests are managed completely asynchronously. Next, if a request is accepted, then the Request Router chooses a node *j*, to submit the request. The choice of node *j* involves selecting all nodes in which the required application is available, and, among these nodes, the node with the lightest load. Therefore, eventually *i* may be equal to *j*. The *Request Router* then routes the user request to an application node using a traditional load balancing algorithm.

Notice, however, that the database accessed by the user request may be placed at another node *k* since applications and databases are both replicated and not every node hosts a database system. In this case, the choice regarding node *k* will depend on the cluster configuration and the database load at each node.

A node load is computed by a *current load monitor* available at each node. For each node, the load monitor periodically computes application and transaction loads using traditional load balancing strategies. For each type of load, it establishes a load grade and multicasts the grades to all the other nodes. A high grade corresponds to a high load. Therefore, the Request Router chooses the best node for a specific request using the node grades (light node is better as discussed below).

The *Application Manager* is the layer that manages application instantiation and execution using an application server provider. Within an application, each time a transaction is to be executed, the *Transaction Load Balancer* layer is invoked which triggers transaction execution at the best node, using the load grades available at each node. The "best" node is defined as the one with lighter transaction load. The Transaction Load Balancer ensures that each transaction execution obeys the ACID (atomicity, consistency, isolation, durability) properties [14], and then signals to the Application Manager to commit or abort the transaction.

The *Replication Manager layer* manages access to replicated data and assures strong consistency in such a way that transactions that update replicated data are executed in the same serial order at each node. We employ data replication because it provides database access parallelism for applications. Our preventive replication approach avoids conflicts at the expense of a forced waiting time for transactions, which is negligible due to the fast cluster network system.

3. Replication model

In this section, we define all the terms and concepts of lazy replication for fully and partially replicated databases necessary to understand our solutions. Then, we present the consistency criteria for the three types of configurations: Lazy-Master, Multi-master and Partially replicated.

3.1. Configurations

We assume that a replica is an entire relational table. Given a table R, we may have three kinds of copies: primary, secondary and multi-master. A *primary copy*, denoted by R, is stored at a *master* node where it can be updated while a *secondary copy*, denoted by r_i, is stored at one or more *slave* nodes i in read-only mode. A *multi-master copy*, denoted by R_i, is a primary copy that may be stored at several multi-master nodes i. Figure 2 shows various replication configurations, using two tables R and S.

Figure 2(a) shows a bowtie (lazy master) configuration where there are only primary copies and secondary copies. This configuration is useful to speed-up the response times of read-only queries through the slave nodes, which do not manage the update transaction load. However, availability is limited since, in the case of a master node failure, its primary copies can no longer be updated.

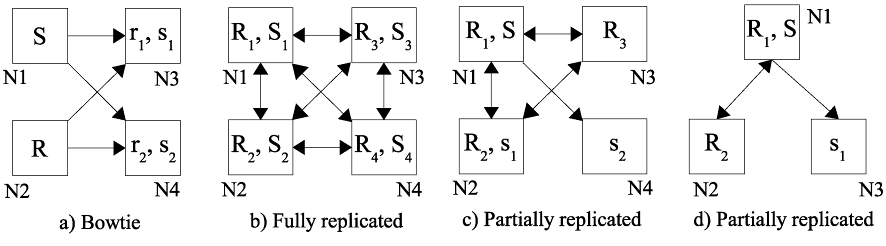


Figure 2. Replication configurations.

Figure 2(b) shows a fully replicated configuration. In this configuration, all nodes manage the update transaction load because whenever *R* or *S* is updated at one node, all other copies need be updated asynchronously at the other nodes. Thus, only the read-only query loads are different at each node. Since all the nodes perform all the transactions, load balancing is easy because all the nodes have the same load (when the specification of the nodes is homogeneous) and availability is high because any node can replace any other node in case of failure.

Figures 2(c) and (d) illustrate partially replicated configurations where all kinds of copies may be stored at any node. For instance, in figure 2(c), node N_1 carries the multi-master copy R_1 and the primary copy *S*, node N_2 carries the multi-master copy R_2 and the secondary copy s_1, node N_3 carries the multi-master copy R_3, and node N_4 carries the secondary copy s_2. Compared with full replication, only some of the nodes are affected by the updates on a multi-master copy (only those that hold common multi-master copies). Therefore, transactions do not have to be multicast to all the nodes. Thus, the nodes and the network are less loaded and the overhead for refreshing replicas is significantly reduced.

With partial replication a transaction *T* may be composed of a sequence of read and write operations followed by a commit (as produced by the SQL statement in figure 3) that updates multi-master copies. This is more general than in [13] where only write operations are considered. We define a *refresh transaction* as the sequence of write operations of a transaction, as written in the Log History. In addition, a *refreshment algorithm* is the algorithm that manages, asynchronously, the updates on a set of multi-master and secondary copies once one of the multi-master (or primary) copies is updated by *T* for a given configuration.

Given a transaction *T* received in the database cluster, there is an *origin node* chosen by the load balancer that triggers refreshment, and a set of *target nodes* that carries replicas involved with *T*. For simplicity, the origin node is also considered a target node. For instance, in figure 2(b) whenever node N_1 receives a transaction that updates R_1, then N_1 is the origin node and N_1, N_2, N_3 and N_4 are the target nodes. In figure 2(c), whenever N_3 receives a transaction that updates R_3, then the origin node is N_3 and the target nodes are N_1, N_2 and N_3.

To refresh multi-master copies in the case of full replication, it is sufficient to multicast the incoming transactions to all target nodes. But in the case of partial replication, even if a transaction is multicast towards all nodes, it may happen that the nodes are not be able to execute it because they do not hold all the replicas necessary to execute *T* locally. For instance, figure 2(c) allows an incoming transaction at node N_1, such as the one in figure 3 to read s_1 in order to update R_1. This transaction can be entirely executed at N_1 (to update R_1) and N_2 (to update R_2). However it cannot be executed at node N_3 (to update R_3)

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

Figure 3. Incoming transaction at node N1.

because N_3 does not hold a copy of S. Thus, refreshing multi-master copies in the case of partial replication needs to take into account replica placement.

3.2. Consistency criteria

Informally a correct refreshment algorithm guarantees that any two nodes holding a common set of replicas, R_1, R_2, … , R_n, must always produce the same sequence of updates on $R_1, R_2, \ldots, R_n$. For each configuration and its sub-configurations, we provide a criterion that must be satisfied by the refreshment algorithm in order to be correct. Group communication systems provide multicast services that differ in the final order in which messages are delivered at each node. We use these known orders [14] as a guide to express our correctness criteria. An example of each configuration is presented in Section 3.1.

Lazy-Master configuration (figure 2(a)). In Lazy-Master configurations, inconsistency may arise if slave nodes can commit their refresh transactions in an order different than their corresponding master nodes. The following correctness criterion prevents this situation.

Definition 3.1 (Total order). Two refresh transactions RT_1 and RT_2 are said to be in *total order* if any slave node that commits RT_1 and RT_2, commits them in the same order.

Proposition 3.1. *For any cluster configuration C that meets a lazy-master configuration requirement, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

Multi-Master configuration (Figure 2(d)). In Multi-Master configurations, inconsistencies may arise whenever the serial execution orders of two transactions at two nodes are not equal. Therefore, transactions must be executed in the same serial order at any node. Thus, Global FIFO Ordering is not sufficient to guarantee the correctness of the refreshment algorithm. Hence the following correctness criterion is necessary:

Definition 3.2 (Total order). Two transactions T_1 and T_2 are said to be executed in *Total Order* if all multi-owner nodes that commit both T_1 and T_2 commit them in the same order.

Proposition 3.2. *For any cluster configuration C that meets a multi-master configuration requirement, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

Partially-Replicated configurations (Figures 2(c) and (d)). In a Partially-Replicated configuration, the inconsistency issues are similar to those found in each component sub-configuration, namely multi-master and lazy-master. That is, two transactions T_1 and T_2 must be executed in the same order at the multi-owner nodes, and, in addition, their corresponding refresh transactions RT_1 and RT_2 must commit in the same order in which the origin node commit T_1 and T_2. Therefore, the following correctness criterion prevents inconsistencies:

Proposition 3.3. *If a cluster configuration C meets partially replicated configuration requirement, then the refresh algorithm that C uses is correct if and only if for each sub-configuration SC correctness is enforced (see Propositions 3.1 and 3.2).*

Proposition 3.4. *For any cluster configuration C that meets the partially replicated requirements, the refresh algorithm that C uses is correct if and only if the algorithm enforces total order.*

4. Preventive refreshment

In this section, we first present the basic refreshment algorithm originally designed for full replication. Then we present the extension of the algorithm to manage partial replication. Afterwards we show the correctness of the algorithm for both fully and partially replicated configurations. Finally, we describe the Replication Manager architecture that implements these algorithms.

4.1. Full replication

We assume that the network interface provides global FIFO reliable multicast: messages multicast by one node are received at the multicast group nodes in the order they have been sent [7]. We denote by *Max*, the upper bound of the time needed to multicast a message from a node i to any other node j. It is essential to have a value of *Max* that is not over estimated. The computation of *Max* resorts to scheduling theory [22] and takes into account several parameters such as the global reliable network itself, the characteristics of the messages to multicast and the failures to be tolerated. We also assume that each node has a local clock. For fairness, clocks are assumed to have a drift and to be ε-synchronized. This means that the difference between any two correct clocks is not higher that ε (known as the precision).

To define the refreshment algorithm, we need the formal correctness criterion presented in Section 3.2 to define strong copy consistency. Inconsistencies may arise whenever the serial orders of two transactions at two nodes are not equal. Therefore, they must be executed in the same serial order at any two nodes. Thus, global FIFO ordering is not sufficient to guarantee the correctness of the refreshment algorithm.

Each transaction is associated with a chronological timestamp value C. The principle of the preventive refreshment algorithm is to submit a sequence of transactions in the same chronological order at each node. Before submitting a transaction at node i, we must check whether there is any older transaction en route to node i. To accomplish this, the submission time of a new transaction at node i is delayed by $Max+\varepsilon$. Thus the earliest time a transaction is submitted is $C + Max + \varepsilon$ (henceforth *delivery time*).

Whenever a transaction T_i is to be triggered at some node i, node i multicasts T_i to all nodes 1, 2, ... , n, including itself. Once T_i is received at some other node j (i may be equal to j), it is placed in the pending queue in FIFO order with respect to the triggering node i. Therefore, at each multi-master node i, there is a set of queues, q_1, q_2, ... , q_n, called pending queues, each of which corresponds to a multi-master node and is used by the

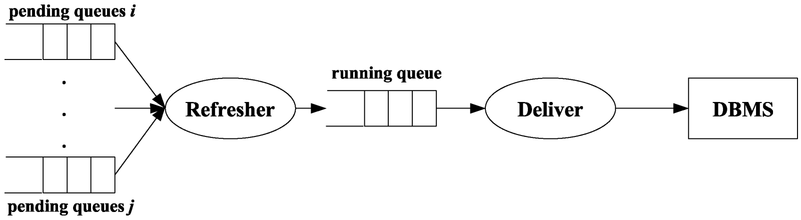


Figure 4. Refreshment architecture.

refreshment algorithm to perform chronological ordering with respect to the delivery times. Figure 4 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the top of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction is ordered, then the refresher writes it to the *running que*ue in FIFO order, one after the other. Finally *Deliver* keeps checking top of the running queue to start transaction execution, one after the other, in the local *DBMS*.

Let us illustrate the algorithm by an example. Suppose we have two nodes *i* and *j*, masters of the copy *R*. So at node *i*, there are two pending queues: *q*(*i*) and *q*(*j*) corresponding to multi-master nodes *i* and *j*. T_1 and T_2 are two transactions which update *R*, respectively on node *i* and on node *j*. Let us suppose that *Max* is equal to 10 and ε is equal to 1. So, on node *i*, we have the following sequence of execution:

– At time 10: T_2 arrives at node *i* with a timestamp $C_2 = 5$

- $q(i) = [T_2\ (5)]$, $q(j) = []$
- T_2 is chosen by the Refresher to be the next transaction to perform at *delivery_time* 16 $(5 + 10 + 1)$, and the time is set to expire at time 16.

– At time 12: T_1 arrives from node *j* with a timestamp $C_1 = 3$

- $q(i) = [T_2\ (5)]$, $q(j) = [T_1\ (3)]$
- T_1 is chosen by the Refresher to be the next transaction to perform at *delivery_time* 14 $(3 + 10 + 1)$, and the time is re-set to expire at time 14.

– At time 14: the timeout expires and the Refresher writes T_1 into the running queue.

- $q(i) = [T_2\ (5)]$, $q(j) = []$
- T_2 is selected to be the next transaction to perform at *delivery_time* 16 $(5 + 10 + 1)$

– At time 16: the timeout expires. The Refresher writes T_2 into the running queue.

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

```
Multi-Master Refresher

Input:
 pending queues $q_1,…q_n$
Output:
 running queue
Variables:
 curr_T: currently selected transaction to be executed;
 first_T: transaction with the lowest timestamp in the
          pending queue
 timer: local reverse timer whose state is either active or
        inactive

Begin
 timer.state = inactive;
 curr_T = first_T = 0;
 Repeat
  On arrival of a new message
  or when (timer.state = active and timer.value = 0) do

  Step1:
   first_T <- message with min C among top messages
              of the pending queues;

  Step2:
   If first_T <> curr_T then
    curr_M <- first_T;
    calculate delivery_time(curr_T);
    timer.value <- delivery_time(curr_T) - local_time
    timer.state <- active;
   end if

  Step 3:
   If timer.state = active and timer.value = 0 then
    append curr_T to the running queue;
    dequeue curr_T from its pending queue;
    timer.state <- inactive;
   end if
 for ever
end
```

Figure 5. Multi-master refresher algorithm.

Although the transactions are received in wrong order with respect to their timestamps (T_2 then T_1) they are written into the running queue in chronological order according to their timestamps (T_1 then T_2). Thus, the total order is enforced even if messages are not sent in total order.

In figure 5, we can see the three steps of the algorithm used in the Refresher module. In step 1, at the reception of a new message in a pending queue, we choose the most recent message from the pending queues. In step 2, we calculate the *delivery_time* according to the timestamp of the message and the $Max + \varepsilon$, and then we set a local reverse timer that will expire at the *delivery_time*. Finally, in step 3, when the timer is over, the message is submitted to the running queue for execution.

4.2. *Partial replication*

With partial replication, some of the target nodes may not be able to perform a transaction *T* because they do not hold all the copies necessary to perform the read set of *T* (recall the discussion on figure 3). However the write sequence of *T*, which corresponds to its refresh transaction, denoted by *RT*, must be ordered using *T*'s timestamp value in order to ensure consistency. So *T* is scheduled as usual but not submitted for execution. Instead, the involved target nodes wait for the reception of the corresponding *RT*. Then, at origin node *i*, when the commitment of *T* is detected (by sniffing the DBMS' log—see Section 4.3), the corresponding *RT* is produced and node *i* multicasts *RT* towards the target nodes. Upon reception of *RT* at a target node *j*, the content of *T* (still waiting) is replaced with the content of incoming *RT* and *T* can be executed.

Let us now illustrate the algorithm with an example of execution. In figure 6, we assume a simple configuration with 4 nodes (N_1, N_2, N_3 and N_4) and 2 copies (*R* and *S*). N_1 carries a multi-owner copy of *R* and a primary copy of *S*, N_2 a multi-owner copy of *R*, N_3 a secondary copy of *S*, and N_4 carries a multi-owner copy of *R* and a secondary copy of *S*.

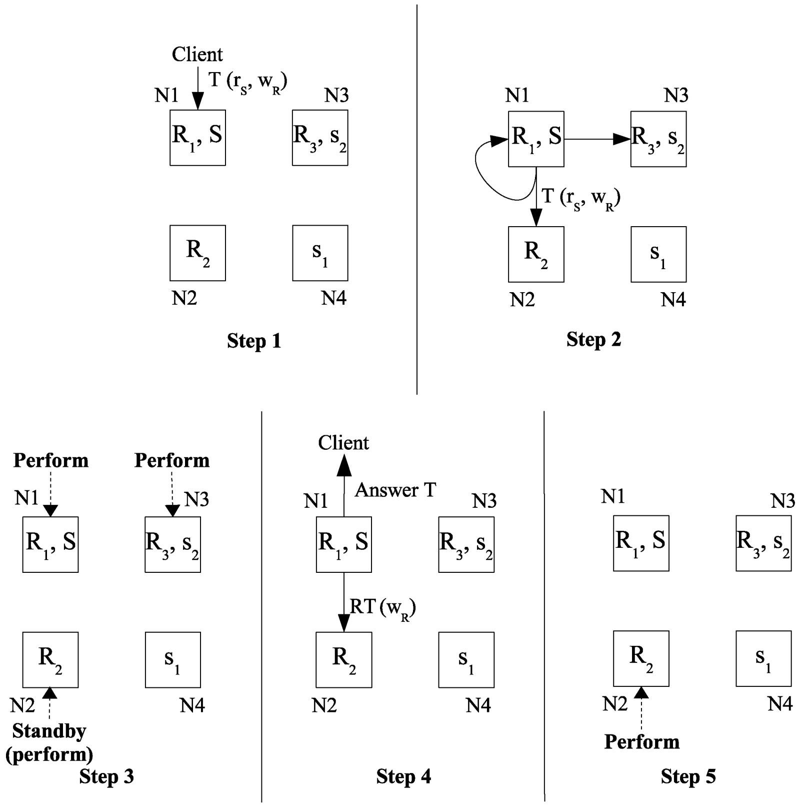


Figure 6. Example of preventive refreshment with partial configurations.

The refreshment proceeds in 5 steps. In step 1, N_1 (the origin node) receives T from a client which reads S and updates R_1. For instance, T can be the resulting read and write sequence produced by the transaction of figure 3. Then, in step 2, N_1 multicasts T to the involved target nodes, i.e. N_1, N_2 and N_4. N_3 is not concerned with T because it only holds a secondary copy s. In step 3, T can be performed using the refreshment algorithm at N_1 and N_4. At N_2, T is also managed by the Refresher and then put in the running queue. However, T cannot yet be executed at this target node because N_2 does not hold S. Thus, the Deliver needs to wait for its corresponding *RT* in order to apply the update on R (see step 4). In step 4, after the commitment of T at the origin node, the *RT* is produced and multicasts it to all involved target nodes. In step 5, N_2 receives *RT* and the Receiver replaces the content of T by the content of *RT*. The Deliver can then submit *RT*.

Partial replication may be blocking in case of failures. After the reception of T, some target nodes would be waiting for *RT*. Thus, if the origin node fails, the target nodes are blocked. However, this drawback can be easily solved by replacing the origin node by an equivalent node, a node that holds all the replicas necessary to execute T. Once the target nodes detect the failure of the origin node, it can request an equivalent node j to multicast *RT* given T's identifier. At node j, *RT* was already produced in the same way that at the origin node: transaction T is executed and, upon detection of T's commitment, an *RT* is produced and stored in a *RT* log (see Section 4.4), necessary to handle failure of the origin node. In the worst case where no other node holds all the replicas necessary to execute T, T is globally aborted. Reconsider the example in figure 6: if N_1 fails at Step 3, N_2 can not receive the *RT* corresponding to the waiting T. So, once N_2 detects that N_1 is out of service, it can identify that N_4 has all copies necessary for T (remember that the global data placement is known) and request the transfer of *RT* to N_4. So, we assume that *RT*'s logs are kept at each node (see Section 4.4). In addition, if N_4 is also out of service, then no node can perform T. Thus, N_2 would abort transaction T. Consistency is enforced because none of the active nodes has performed the transaction. In this case, at recovery time, the failed nodes would undo T.

4.3. Correctness of the refresher algorithm

In this section we show that the refresher algorithm is correct. The proofs for the lazy master based configurations appear in [12] and we do not re-discuss them here. The proofs for partial configurations we consider come directly from those of lazy-master and multi-master configurations, as we will show.

Lemma 4.1. *The refreshment algorithm is correct for multi-master configurations.*

Proof: Let us consider any node N of a multi-master configuration holding multi-owner copies. Let T be any transaction committed by node N. The propagator located at node N will propagate the operations performed by T by means of a message using reliable multicast. Hence any node involved in the execution of the transaction receives the update message. Since (i) the message containing the timestamp of any transaction T is the last one related to that transaction, and (ii) the reliable multicast preserves the global FIFO order, when a node *N'* receives the message containing the timestamp of *T* (i.e., at delivery time $C + Max + \varepsilon$), it has previously received all operations related to *T* and involving that node. Hence the transaction can be committed when all its operations are done and earliest at delivery time $C + Max + \varepsilon$. □

Lemma 4.2. *The refreshment algorithm is correct for partial configurations.*

Proof: Let us consider any node *N* of a partial configuration holding at least one multi-owner copy. Let *T* be any transaction submitted to node *N*, so *N* is the origin node of *T*. When the update message is received by any node involved in the execution of the transaction, by Lemma 4.1, transaction *T* can be committed when all its operations are done and earliest at delivery time $C + Max + \varepsilon$. But in the case where the node does not hold all the copies necessary to the transaction, *T* waits. Since an origin node must hold all the copies necessary to the transaction submitted by a client, the node *N* can perform *T*. Then, node *N* produces and multicasts *RT* which contains the write set associated to *T* to all waiting target nodes. So, the waiting target nodes can perform *T* by replacing the content of the transaction by its write set. Hence the transaction is still committed earliest at delivery time $C + Max + \varepsilon$. □

Lemma 4.3 (Transaction chronological order). *The refreshment algorithm ensures that, if T_1 and T_2 are any two transactions that start execution at global times t_1 and t_2, respectively, then: if $t_2 - t_1 > \varepsilon$, the timestamps C_2 for T_2 and C_1 for T_1 satisfy $C_2 > C_1$; any node that commits both T_1' and T_2', commits them in the order given by C_1 and C_2.*

Proof: Let us assume that $t_2 – t_1 > \varepsilon$ Even if the clock of the node committing T_1 is ε ahead with regard to the clock of the node committing T_2, we have $C_2 > C_1$. We now assume that we have $C_2 > C_1$ and we consider a node *N* that commits first T_1' and then T_2'. According to the algorithm, T_2' is not committed before local time $C_2 + Max + \varepsilon$. At that time, if *N* commits T_2' before T_1', it means that *N* has not received the message related to T_1. Since clocks are ε synchronised, that message would have experienced a multicast delay higher than *Max*. □

Lemma 4.4 (Total order). *The refreshment algorithm satisfies the total order criterion for any configurations.*

Proof: If the refreshment algorithm is correct (Lemmas 4.1 and 4.2) and the transactions are performed in chronological order on each node (Lemma 4.3), then the total order is enforced. □

Lemma 4.5 (Deadlock). *The refreshment algorithm ensures that no deadlock appears.*

Proof: Let us consider a transaction T_1 which has for origin node N_1 and waits for its write set at node N_2 and a transaction T_2 which has for origin node N_2 and waits for its write set at N_1. A deadlock appears if and only if T_1 is performed before T_2 on N_2 and if T_2 is performed before T_1 on N_1. Hence, the total order is not enforced. This contradicts Lemma 4.3 since transactions are always performed in their chronological order at all the nodes. □

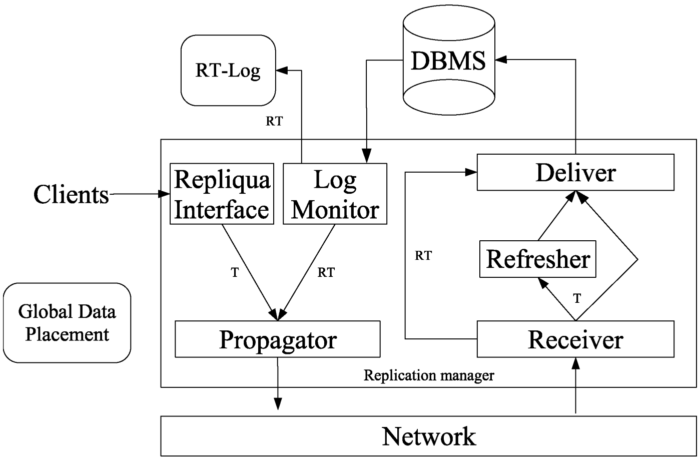


Figure 7. Replication Manager architecture.

4.4. Replication Manager architecture

In this section, we present the Replication Manager architecture to implement the Preventive Partial Replication algorithm (see figure 7). We add several components to a regular DBMS while preserving node autonomy, i.e. without requiring the knowledge of system internals. The Replica Interface receives transactions coming from the clients. The Propagator and the Receiver manage the sending and reception (respectively) of transactions and refresh transactions inside messages within the network.

Whenever the Receiver receives a transaction, it places it in the appropriate pending queue, used by the Refresher, and in the running queue used by the Deliver to start its execution. Next, the Refresher executes the refreshment algorithm to ensure strong consistency. The Deliver submits transactions, read from the running queue, to the DBMS and commits them only when the Refresher ensures that the transactions have been performed in chronological order.

With partial replication, when a transaction *T* is composed of a sequence of reads and writes, the Refresher at the target nodes must assure correct ordering. However, in case where the node does not hold all the necessary copies, *T* 's execution must be delayed until its corresponding refresh transaction *RT* is received. This is because *RT* is produced only after the commitment of the corresponding *T* at the origin node. At the target node, the content of *T* (sequence of read and write operations) is replaced by the content of the *RT* (sequence of write operations) in the Deliver. Thus, at the target node, when the Receiver receives *RT*, it interacts directly with Deliver.

The Log Monitor constantly checks the content of the DBMS log to detect whether replicas have been updated. For each transaction *T* that updated a replica, it produces a corresponding refresh transaction. At the origin node, whenever the corresponding transaction is composed of reads and writes and some of the target nodes do not hold all the necessary replicas, the Log Monitor submits the refresh transaction to the propagator, which multicasts it to those nodes. Then, upon receipt of the refresh transaction, the target nodes can perform the corresponding waiting transaction. To provide fault-tolerance in case of failure of the origin node (see Section 4.2), Log Monitor stores *RT*, in addition to the origin node, in all the nodes that are able to perform the transaction (nodes which hold all necessary replicas to perform a transaction *T*). Thus, in case of failure of the origin node, one of these nodes can replace the origin node and multicast the *RT* to the target nodes that can not perform the corresponding *T*.

5. Improving response time

In this section, we present optimizations for both Full and Partial Replication that improve transaction throughput. First, we modify the algorithm to eliminate partially the delay times ($Max + \varepsilon$) before submitting transactions. Then, we introduce concurrency control features in the algorithm to improve transaction throughput. Finally, we show the correctness of these optimizations.

5.1. Eliminating delay time

In a cluster network (which is typically fast and reliable), in most cases messages are naturally chronologically ordered [16]. Only a few messages can be received in an order that is different than the sending order. Based on this property, we can improve our algorithm by submitting a transaction to execution as soon as it is received, thus avoiding the delay before submitting transactions. Yet, we still need to guarantee strong consistency. In order to do so, we schedule the commit order of the transactions in such a way that a transaction can be committed only after $Max + \varepsilon$. Recall that to enforce strong consistency, all the transactions must be performed according to their timestamp order. So, a transaction is out-of-order when its timestamp is lower than the timestamps of the transactions already received. Thus, when a transaction *T* is received out-of-order, all younger transactions must be aborted and re-submitted according to their correct timestamp order with respect to *T*. Therefore, all transactions are committed in their timestamp order.

Thus, in most cases the delay time ($Max + \varepsilon$) is eliminated. Let *t* be the time to execute transaction *T*. In the previous algorithm [13], the time spent to refresh a multi-master copy, after reception of *T*, is $Max + \varepsilon + t$. Now, a transaction *T* is ordered while it is executed. So, the time to refresh a multi-master copy is $\max[(Max + \varepsilon), t]$. In most cases, *t* is higher than the delay $Max + \varepsilon$. Thus, this simple optimization can well improve throughput as we show in our performance study.

Figure 8 shows part of the components necessary to run our algorithm. The *Refresher* reads transactions from the head of *pending queues* and performs chronological ordering with respect to the delivery times. Once a transaction *T* is ordered, the refresher notifies *Deliver* that *T* is ordered and ready to be committed. Meanwhile, Deliver keeps checking the head of the running queue to start transaction execution optimistically, one after the other, inside the local *DBMS*. However, to enforce strong consistency Deliver only commits a transaction when the Refresher has signaled it.

Let us illustrate the algorithm with an example from figure 8. Suppose we have a node *i* that holds the master of the copy *R*. Node *i* receives T_1 and T_2, two transactions that update

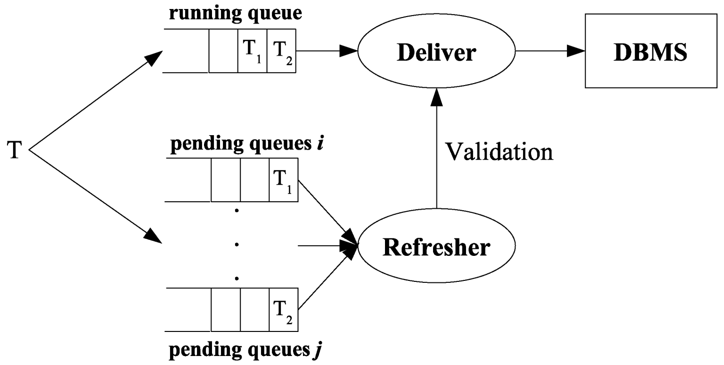


Figure 8. Refreshment architecture.

R, respectively from node i with a timestamp $C_1 = 10$ and from node j with a timestamp $C_2 = 15$. T_1 and T_2 must be performed in chronological order, T_1 then T_2. Let us see what happens when the messages are not received chronologically ordered at node i. In our example, T_2 is received before T_1 at node i and immediately written into the running queue and the corresponding pending queue. Thus, T_2 is submitted to execution by the Deliver but must wait the Refresher's decision to commit T_2. Meanwhile, T_1 is received at node i, it is similarly written into both pending and running queues. However the Refresher detects that the younger transaction T_2 has already been submitted before T_1. So, T_2 is aborted and re-started, causing it to be re-inserted into the running queue (after T_1). T_1 is chosen to be the next transaction to commit. Finally, T_2 is performed and elected to commit by Refresher. Thus, the transactions are committed in their timestamp order, even if they have been received unordered.

Preventive algorithm details. We can define three different states for a transaction T represented in figure 9. When a transaction T arrives at the Replication Manager, its state is

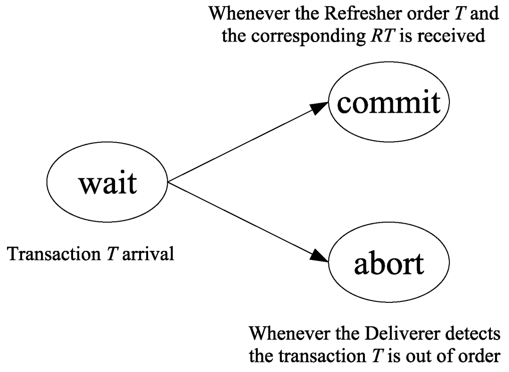

Whenever the Refresher order T and the corresponding RT is received
commit
wait
Transaction T arrival
abort
Whenever the Deliverer detects the transaction T is out of order

Figure 9. Transition state graph for *T*.

initialized to *wait*. Then, when *T* can be executed (a transaction can be executed when the node holds all necessary replicas or when its corresponding *RT* is received), and when the Refresher has ordered the transaction, the state of Transaction *T* is set to *commit*. Finally, when the Deliver receives an out-of-order transaction *T* (its timestamp is lower than the timestamps of the transactions already received), the state of the current running transactions is set to *abort*.

The Preventive algorithm is described in detail in figures 10 and 11. Figure 10 describes the Refresher algorithm. The Refresher selects the next totally ordered transaction. A transaction is guaranteed to be totally ordered at its *delivery_time* ($C + Max + \varepsilon$). Thus, in step 1, on the arrival of a new transaction, the refresher chooses the oldest transaction *T*

```
Partial Replication Refresher

Input:
 pending queues q1,…qn
Variables:
 curr_T:     currently selected transaction to be ordered;
 first_T:    transaction with the lowest timestamp
             in the pending queues;
 running_T:  transactions in the running queue;
 timer:      local reverse timer whose state is
             either active or inactive;

Begin
 timer.state = inactive;
 curr_T = 0
 Repeat

  Step 1:
  On arrival of a new transaction in the pending queue
  or when timer expires do
   first_T <- message with min timestamp among top
              messages of the pending queues;
   If first_T <> curr_T then
    curr_T <- first_T;
    calculate delivery_time(curr_T);
    timer.value <- delivery_time(curr_T) - local_time
    timer.state <- active;
   endif

  Step 2:
   If timer expires then
    For all transactions running_T in the running queue
    such as curr_T = running_T do
     running_M.state = commit;
    end for;
    Dequeue curr_T from its pending queue;
    timer.state <- inactive;
   endif;
 for ever;
end;
```

Figure 10. Partial replication refresher algorithm with elimination of delay times.

```
Partial Replication Deliver (no concurrency)

Input:
 running queue, RT list
Variables:
 new_T: new incoming transactions from the running queue
 curr_T: currently running transaction;
 new_RT: new incoming refresh transactions from the RT list
Begin
 curr_T = 0;
 Repeat
  On arrival of a new transaction new_T
  or at the end of curr_T
  or the state of curr_T changes
  or on arrival of new refresh transaction new_RT do

  Step 1:
   If curr_T finishes performing then
    If curr_T.state = abort then
     Rollback curr_T;
     Dequeue curr_T from the running queue;
     curr_T = 0;
    elseif curr_T.state = commit then
     Commit curr_T;
     Dequeue curr_T from the running queue;
     curr_T = 0;
    endif;
   endif;

  Step 2:
   If new_T <> 0 then
    new_T.state = wait;
    If curr_T <> 0 and curr_T.C > new_T.C then
     curr_T.state = abort;
     Introduce a copy of curr_T in the running queue
     according to its timestamp with a wait state;
    end if;
   end if;

  Step 3:
   If curr_T = 0 and the running queue is not empty then
    curr_T <- the transaction at the top of
              the running queue;
    If node holds all copies necessary to curr_T then
     curr_T.standby = false;
     Perform curr_T;
    else
     curr_T. standby = true;
    end if;
   end if;

  Step 4:
   If curr_T <> 0 and curr_T.standby = true then
    Extract the new_RT corresponding to curr_T from RT_list;
    If new_RT <> 0 then
     new_RT.state = curr_T.state;
     curr_T = new_RT;
     curr_T.standby = false;
     Perform curr_T;
    end if;
   end if;
 for ever;
end;
```

Figure 11. Partial replication deliver algorithm with elimination of delay times.

from the top of the pending queues and computes *T* 's *delivery_time*. Next the Refresher initializes a timer that will expire at *T* 's *delivery_time*. So, if the incoming transaction *T* is not out-of-order according to the current selected transaction, *curr_T*, nothing happens. In the other case, the new *T* 's *delivery_time* is calculated according to *T* 's timestamp. In step 2, when the timer expires, the Refresher looks for the non aborted transactions corresponding to *curr_T*. Then, it sets the state of *curr_T* to commit.

Figure 11 describes the Deliver algorithm in the optimistic arrival approach. Deliver reads transactions from the running queue and executes them. If a transaction *T* is out-of-order, Deliver aborts the current running transaction, *curr_T*, and executes *T* followed by *curr_T*. Deliver commits a transaction when the Refresher sets its state to commit. In step 1, at the end of the execution of the current transaction *curr_T*, Deliver commits or rolls-back *curr_T* according to its state (commit or abort). Since we do not have access to the transaction manager of the DBMS, we cannot abort directly the transactions and we must wait until the end of the transaction to abort it. In step 2, Deliver sets the state of the newly received transaction (*new_T*) to wait and checks whether *new_T* is not an out-of-order transaction. If the transaction is out-of-order, the state of the current transaction (*curr_T*) is set to abort. As the Deliver has to wait the end the transaction to rollback the transaction, a copy of *curr_T* is reintroduced in the running queue and its state is set to wait while the aborted *curr_T* is running. Thus, a transaction *T* aborted due to an unordered message will be re executed from a copy of *curr_T*. In step 3, the Refresher selects the transaction at the top of the running queue and performs it if the node holds all the copies necessary to the transaction, Otherwise, the Refresher set the transaction in stand by. Finally, in step 4, on arrival of a new refresh transaction *new_RT*, the Deliver replaces the content of the waiting *T* by the content of its corresponding *RT*. So, the current transaction can execute.

5.2. Improving transaction throughput

To improve throughput, we now introduce concurrent replica refreshment. In the previous section, the Receiver writes transactions directly into the running queue (optimistically), and afterwards the Deliver reads the running queue contents in order to execute the transaction, and in the other hand, to assure consistency, the same transactions are written as usually in the pending queues to be ordered by the Refresher. Hence, the Deliver extracts the transactions from the running queue and performs them one by one in serial order. So, if the Receiver fills the running queue faster than the Deliver empties it, and if the average arrival rate is higher than the average running rate of a transaction (typically in bursty workloads), the response time increases exponentially and performance degrades.

To improve response time in bursty workloads we propose to trigger transactions concurrently. In our solution, concurrency management is done outside the database to preserve autonomy (different from [9]). Using the existing isolation property of database systems [11], at each node, we can guarantee that each transaction sees a consistent database at all times. To maintain strong consistency at all nodes, we enforce that transactions are committed in the same order in which they are submitted. In addition, we guarantee that transactions are submitted in the order in which they have been written to the running queue. Thus, total order is always enforced.

However, without access to the DBMS concurrency controller (for autonomy reasons), we cannot guarantee that two conflicting concurrent transactions obtain a lock in the same order at two different nodes. Therefore, we do not trigger conflicting transactions concurrently. To detect that two transactions are conflicting, we determine a subset of the database items accessed by the transaction according to the transaction. If the subset of a transaction does not intersect with a subset of another transaction, then the transactions are not conflicting. For example, in the TPC-C benchmark, the transactions' parameters allow us to define a subset of tuples that could be read or updated by the transaction. Notice that if the subset of the transaction cannot be determined, then we consider the transaction to be conflicting with all other transactions. This solution is efficient if most transactions are known, which is true in OLTP environments.

We can now define two new conditions to be verified by the Deliver before triggering and before committing a transaction:

(i) Start a transaction iff the transaction is not conflicting with transactions already started (but not committed) and iff no older transaction waits for the commitment of a conflicting transaction to start.
(ii) Commit a transaction iff no older transactions are still running.

Figure 12 shows examples of concurrent executions of transactions. Figure 12(a) illustrates a case where the transactions are triggered sequentially, which is equivalent to the case where all the transactions are conflicting. Figures 12(b), (c) and (d) show parallel executions of transaction T_1, T_2 and T_3. In figures 12(b) and (c), transaction T_2 finishes before T_1 but waits for commit because T_1 is still running (this is represented by a dashed line in the figure). In figure 12(b), T_1, T_2 and T_3 are not conflicting, so they can run concurrently. On the other hand, in figure 12(c), T_2 is conflicting with T_3, so T_3 must wait for the end of T_2 before starting. Finally, in figure 12(d), T_1 and T_2 are conflicting, so T_2 cannot start before the commitment of T_1 and T_3 cannot start before T_2 because transactions must be executed in the order they are in the running queue.

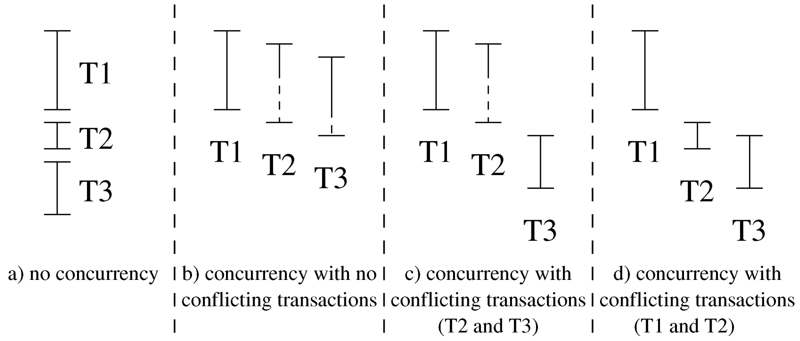

T1
T2
T3
a) no concurrency
T1
T2
T3
b) concurrency with no conflicting transactions
T1
T2
T3
c) concurrency with conflicting transactions (T2 and T3)
T1
T2
T3
d) concurrency with conflicting transactions (T1 and T2)

Figure 12. Example of concurrent execution of transactions.

5.3. Correctness

In this section, we prove that the Preventive Replication algorithm is also correct with the optimizations.

Lemma 5.1. *The elimination of the delay Max + ε does not introduce inconsistency.*

Proof: Let T_1 and T_2 be any two transactions with timestamps C_1 and C_2. If T_1 is older than T_2 ($C_1 < C_2$) and T_2 is received on node i before T_1, then T_2 is managed optimistically. However T_2 cannot be committed before $C_2 + Max + \varepsilon$, and as T_1 is received at node i at the latest at $C_1 + \text{Max} + \varepsilon$, then, T_1 is received before T_2 is committed ($C_1 + Max + \varepsilon < C_2 + Max + \varepsilon$). Therefore, T_2 is aborted, and both transactions are written in the running queue, executed and committed according to their timestamp values. Afterwards, T_1 is executed before T_2, and the strong consistency is enforced even in the case of unordered messages. □

Lemma 5.2. *The parallel execution of transactions does not break the enforcement of strong consistency.*

Proof: Let T_1 and T_2 be any two transactions with timestamps C_1 and C_2 that start execution at times t_1 and t_2, and commit at times c_1 and c_2, respectively. In the case where T_1 and T_2 are received unordered, the transactions are aborted and re-executed in the correct order as described in Lemma 5.2. Now, in the case where the transactions are received correctly ordered, if T_1 and T_2 are conflicting, they start and commit one after the other according to their timestamp values. Hence, if $C_1 < C_2$, then $t_1 < c_1 < t_2 < c_2$. If they are not conflicting, T_2 can start before T_1 commits. However, a transaction is never committed before all older transactions have been committed. If $C_1 < C_2$, then $t_1 < t_2$ and $c_1 < c_2$. Thus, the state of the database viewed by a transaction before its execution and its commitment is the same at all the nodes. Hence, strong consistency is enforced. □

6. Validation

In this section, we describe our implementation and our performance model. Then, we describe two experiments to study scale up and speed-up.

6.1. Implementation

We implemented our Preventive Replication Manager in our RepDB∗ prototype [2, 19] on a cluster of 64 nodes (128 processors). Each node has 2 Intel Xeon 2.4 GHz processors, 1 GB of memory and 40 GB of disk. The nodes are linked by a 1 Gb/s network. We use Linux Mandrake 8.0/Java and CNDS's Spread toolkit that provides a reliable FIFO message bus and high-performance message service among the cluster nodes. We use PostgreSQL Open Source DBMS at each node. We chose PostgreSQL because it is quite complete in terms of transaction support and easy to work with.

Our implementation has four modules: Client, Replicator, Network and Database Server. The Client module simulates the clients. It submits transactions randomly to any cluster node, via RMI-JDBC, which implements the Replica Interface. Each cluster node hosts a Database Server and one instance of the Replicator module. For this validation, we implemented most of the Replicator module in Java outside of PostgreSQL. For efficiency, we implemented the Log Monitor module inside PostgreSQL. The Replicator module implements all system components necessary for a multi-master node: Replica Interface, Propagator, Receiver, Refresher and Deliver. Each time a transaction is to be executed, it is first sent to the Replica Interface that checks whether the incoming transaction updates a replica. Whenever a transaction does not write a replica, it is sent directly to the local transaction manager. Even though we do not consider node failures in our performance evaluation, we implemented all the necessary logs for recovery to understand the complete behavior of the algorithm. The Network module interconnects all cluster nodes through the Spread toolkit.

6.2. Performance model

To perform our experiments, we use the TPC-C Benchmark [18] which is an OLTP workload with a mix of read-only and update intensive transactions. It has 9 tables: Warehouse, District, Customer, Item, Stock, New-order, Order, Order-line and History; and 5 transactions: Order-status, Stock-level, New-order, Payment and Delivery.[2]

The parameters of the performance model are shown in Table 1. The values of these parameters are representative of typical OLTP applications. The size of the database is proportional to the number of warehouses (a tuple in the Warehouse table represents a warehouse). The number of warehouses also determines the number of clients that submit a transaction. As specified in the TPC-C benchmark, we use 10 clients per warehouse. For a client, we fix the transaction arrival rate λ_{client} at 10 s. So with 100 clients (10 warehouses and 10 clients per warehouse), the average transactions' arrival rate λ is 100 ms. In our experiments, we vary the number of warehouses *W* to be either 1, 5 or 10. Then, the different average transactions' arrival rates are 1 s, 200 ms and 100 ms.

During an experiment, each client submits to a random node a transaction among the 4 TPC-C transactions used. In the end, each client must have submitted *M* transactions

Table 1. Performance parameters

Parameter	Definition	Values
W	Number of warehouse	1, 5, 10
Clients	Number of clients by warehouse	10
λ_{client}	Average arrival rate for each client	10 s
λ	Average arrival rate	1 s, 200 ms, 100 ms
Conf.	Replication of tables	FR, PR
M	Number of transactions submittedduring the tests for each client	100
$Max + \varepsilon$	Delay introduced for submitting a Transaction	200 ms

and must have maintained a percentage of mixed transactions: 6% for Order-status, 6% for Stock-level, 45% for New-order and 43% for Payment.

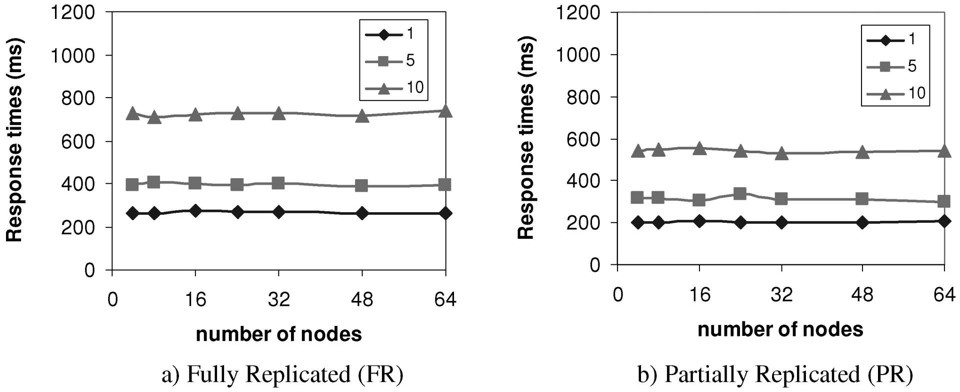
The TPC-C defines a number of different types of transactions. New-order represents a mid-weight, read-write transaction with a high frequency of execution. Payment represents a lightweight, read-write transaction with a high frequency of execution. Order-status represents a mid-weight, read-only transaction with a low frequency of execution. Stock-level represents a heavy, read-only transaction with a low frequency of execution. Thus, we can consider New-order and Payment as multi-master transactions.

Finally, for our experiments, we use two replication configurations. In the *Fully Replicated* (*FR*) configuration all the nodes carry all the tables as multi-master copies. In the *Partially Replicated* (*PR*) configuration, one fourth of the nodes hold tables needed by the Order-status transaction as multi-master copies, another fourth holds tables needed by the New-order transaction as multi-master copies, another fourth holds tables needed by the Payment transaction as multi-master copies and the last fourth holds tables needed by the Stock-level transaction as multi-master copies.

6.3. Scale up experiments

These experiments study the algorithm's scalability. That is, for a same set of incoming transactions (New-order and Payment transactions), scalability is achieved whenever increasing the number of nodes yields the same response times. We vary the number of nodes for each configuration (*FR* and *PR*) and for different numbers of warehouses (1, 5 and 10). For each test, we measure the average response time per transaction. The duration of this experiment is the time to submit 100 transactions for each client.

The experimental results (see figure 13) show that for all tests, scalability is achieved. The performance remains relatively constant according to the number of nodes. Our algorithm has linear response time behavior even when the number of node increases. Let n be the number of target nodes for each incoming transaction, our algorithm requires only the multicast of n messages for the nodes that carry all required copies



a) Fully Replicated (FR)

b) Partially Replicated (PR)

Figure 13. Scale up results.

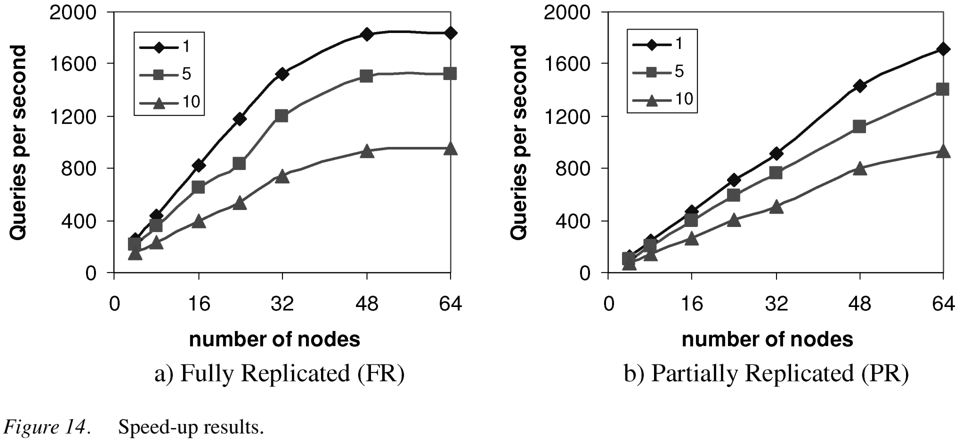
plus $2n$ messages for the nodes that do not carry all required copies. The performance decreases as the number of warehouses increases (which increases the workload). In figure 13(a), although the workload is twice higher for 10 warehouses than for 5 warehouses, the response times remain twice as worse as expected, i.e., 400 ms for 5 warehouses and about 800 ms for 10 warehouses. This demonstrates that our algorithm has good response time when the workload increases and we can expect similar behavior with higher workloads.

The results also show the impact of the configuration on transaction response time. As the number of transactions increases (with the number of nodes that receive incoming transactions), *PR* increases inter-transaction parallelism more than *FR* by allowing different nodes to process different transactions. Thus, transaction response time is slightly better with *PR* (figure 13(a) than with *FR* (figure 13(b)) by about 15%. In *PR*, nodes only hold tables needed by one type of transaction, so they do not have to perform the entire updates of the other type of transactions. Hence, they are less overloaded than in *FR*. Thus the configuration and the placement of the copies should be tuned to selected types of transactions.

6.4. *Speed-up experiments*

These experiments study the performance improvement (speed-up) for read queries when we increase the number of nodes. To test speed-up, we reproduced the previous experiments and we introduced clients that submit queries. We vary the number of nodes for each configuration (*FR* and *PR*) and for different number of warehouses (1, 5 and 10). The duration of this experiment is the time to submit 100 transactions for each client.

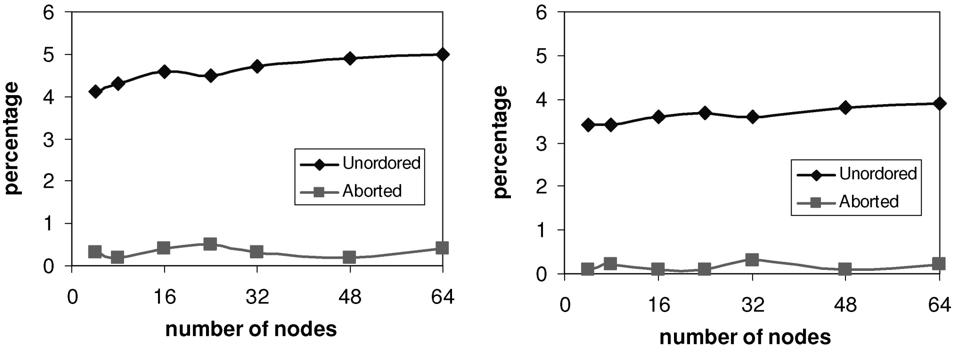
The number of clients that submit queries is 128. The clients submit lightweight queries (Order-status transaction) sequentially while the experiment is running. Each client is associated to one node and we produce an even distribution of clients at each node. Thus, the number of read clients per node is 128 divided by the number of nodes that support the



a) Fully Replicated (FR)

b) Partially Replicated (PR)

Figure 14. Speed-up results.



a) Fully Replicated (FR) b) Partially Replicated (PR)

Figure 15. Percentage of unordered messages and aborted transactions for 10 warehouses.

Order-status transaction. For each test, we measured the throughput of the cluster, i.e. the number of read queries per second.

The experiment results (see figure 14) show that the increase in the number of nodes improves the cluster's throughput. For example in figure 14(a), whatever the number of warehouses, the number of queries per seconds with 32 nodes (1500 queries per seconds) is almost twice that with 16 nodes (800 queries per seconds). However, if we compare *FR* with *PR*, we can see that the throughput is better with *FR*. Although the nodes are less overloaded than in *FR*, performance is half of *FR* because only half of the nodes support the transaction. This is due to the fact that, in *PR*, not all the nodes hold all the tables needed by the read transactions. In *FR*, beyond 48 nodes, the throughput does not increase anymore because the optimal number of nodes is reached, and the queries are performed as fast as possible.

6.5. *Effect of optimistic execution*

Now, we study the effect of optimistically executing transactions as soon as they arrive. Our first study shows the impact of the unordered messages on the number of aborted transactions due to optimistic execution (see Section 5.1). Then, our second study shows the gain of the optimistic approach on the refreshment delay.

In our first experiment, figures 15(a) and (b) show the percentage of the unordered messages and the percentage of the aborted transactions for the scale up experiment (Section 6.3). Below 5% of the messages are unordered, and only 1% of the transactions are aborted. At most only 20% of the unordered messages introduce aborts because two unordered messages are received in a very short period of time (around 2 ms). So, the second message is received before the first message has been processed. Therefore, they are reordered before the execution of the first message.

For *PR*, the Partially Replicated configuration (figure 15(b)), the percentage of the unordered messages is lower than the percentage for *FR*, the Fully Replicated configuration

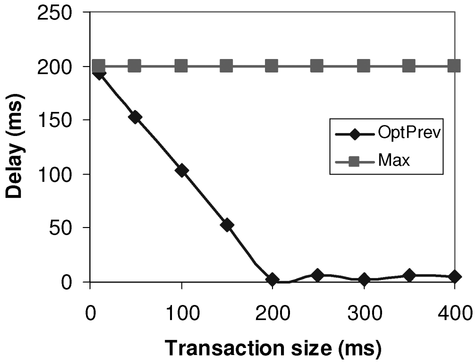


Figure 16. Delay versus transaction size.

(figure 15(a)), because less messages are involved. Thus, the number of aborted transactions is small enough to warrant the gain introduced by the elimination of the delay time.

In our second experiment, we study how the transaction size affects the elimination of the refreshment delay. In the Optimistic Approach, transactions still need to be delayed ($Max + \varepsilon$) before committing. Figure 16 shows the relative importance of the delay time with respect to transaction size. Our test involves only 8 nodes with a FR configuration because the waiting time is not affected by the increase of the number of nodes. We submit 100 transactions in a low workload and we vary the size of the transaction. Then, we measure the delay time introduced by the refreshment algorithm. Recall that the normal delay (Max) value is 200 ms without optimization.

An important observation is that the delay introduced by the refreshment quickly decreases as the transaction time increases. This is due to the fact that, since the transaction is performed as soon as possible, the scheduling of a transaction is performed in parallel with its execution. As the scheduling time is equal to $Max + \varepsilon$, the delay introduced is equal to $\mathrm{Max} + \varepsilon$ minus the size of the transaction. For example, with a transaction size of 50 ms, the delay is 150 ms. Thus, with transactions longer than 200 ms, the delay is almost zero because the scheduling time is included in the execution time. Hence, the gain is almost equal to Max, which is the optimal gain for the elimination of Max. Finally, the number of aborted transactions is not enough significant, so we do not put it on the figure.

7. Related work

Data replication has been extensively studied in the context of distributed database systems [11]. In the context of database clusters, the main issue is to provide scalability (to achieve performance with large numbers of nodes) and autonomy (to exploit black-box DBMS) for various replication configurations such as master-slave, multi-master and partial replication.

Synchronous (eager) replication can provide strong consistency for most configurations including multi-master but its implementation, typically through 2PC, violates system autonomy and does not scale up. In addition, 2PC may block due to network or node failures. The synchronous solution proposed in [9] reduces the number of messages exchanged to commit transactions compared to 2PC. It uses, as we do, group communication services to guarantee that messages are delivered at each node according to some ordering criteria. However, DBMS autonomy is violated because the implementation must combine concurrency control with group communication primitives. In addition solutions based on total order broadcast is not well suited for large scale replication because as the number of nodes increases the overhead of messages exchanged may dramatically increase to assure total order. The Database State Machine [20, 21] supports partial replication for heterogeneous databases and thus does not violate autonomy. However, its synchronous protocol uses two-phase locking that is known for its poor scalability, thus making it inappropriate for database clusters.

Asynchronous (lazy) replication typically trades consistency for performance. A refreshment algorithm that assures correctness for lazy master configurations is proposed in [12]. This work does not consider multi-master and partial replication as we do. The preventive replication solution in [13] is asynchronous and achieves strong consistency for multi-master configurations. However, it introduces heavy message traffic in the network since transactions are multicast to all cluster nodes. In [3], we extended preventive replication to deal with partial replication. However, it also has performance limitations since transactions are forced to wait a delay time before executing. The solution proposed in this paper addresses these important limitations.

The algorithm proposed in [8] provides strong consistency for multi-master and partial replication while preserving DBMS autonomy. However, it requires that transactions update a fixed primary copy: each type of transaction is associated with one node so a transaction of that type can only be performed at that node. This is a problem for update intensive applications. For example, with the TPC-C benchmark, two nodes support 88% of the transactions (45% at one node for the New Order transactions and 43% at another node for the Payment transactions). Furthermore, the algorithm uses 2 messages to multicast the transaction, the first is a reliable multicast and the second is a total ordered multicast. The cost of these messages is higher than the single FIFO multicast message we use. Furthermore, using a logical total order message increases the overhead of physical messages exchanged when increasing the number of nodes. However, one advantage of this algorithm is that it avoids redundant work: the transaction is performed at the origin node and the target nodes only apply the write set of the transaction. In our algorithm, all the nodes that hold the resources necessary for the transaction perform it entirely. We could also remove this redundant work to generalize the multicast of refresh transactions for all nodes instead of only for the nodes that do not hold all the necessary replicas. However, the problem is to decide whether it is faster to perform the transaction entirely or to wait for the corresponding write set from the origin node for short transactions. Finally their experiments do not show scale-up with more than 15 nodes while we go up to 64 in our experiments.

More recent work has focused on snapshot isolation to improve the performance of read-only transactions. The RSI-PC [17] algorithm is a primary copy solution which separates update transactions from read-only transactions Update transactions are always routed to a main replica, whereas read-only transactions are handled by any of the remaining replicas, which act as read-only copies. Postgres-R(SI) [24] proposes a smart solution that does not need to declare transactions properties in advance. It uses the replication algorithm of [8] which must be implemented inside the DBMS. The experiments are limited to at most 10 nodes. SI-Rep [10] provides a solution similar to Postgres-R(SI) on top of PostgreSQL which needs the write set of a transaction before its commitment. Write sets can be obtained by either extending the DBMS, thus hurting DBMS autonomy, or using triggers.

8. Conclusion

In this paper, we introduced two algorithms for preventive replication in order to scale up to large cluster configurations. The first algorithm supports fully replicated configurations where all the data are replicated on all the nodes, while the second algorithm supports partially replicated configurations, where only a part of the data are replicated. Both algorithms enforce strong consistency. Then, we proposed a complete architecture that supports a large numbers of configurations. Moreover, we presented two optimizations that improve transaction throughput; the first optimization eliminates optimistically the delay introduced by the preventive replication algorithm while the second optimization introduces concurrency control features outside the DBMS in which non conflicting incoming transactions may execute concurrently.

We did an extensive performance validation based on the implementation of Preventive Replication in our RepDB$*$ prototype over a cluster of 64 nodes running PostgreSQL. Our experimental results using the TPC-C benchmark show that our algorithm scales up very well and has linear response time behavior. We also showed the impact of the configuration on transaction response time. With partial replication, there is more inter-transaction parallelism than with full replication because of the nodes being specialized to different tables and thus transaction types. Thus, transaction response time is better with partial replication than with full replication (by about 15%). The speed-up experiment results showed that the increase of the number of nodes can well improve the query throughput. Finally, we showed that, with our optimistic approach, unordered transactions introduce very few aborts (at most 1%) and that the waiting delay for committing transactions is very small (and reaches zero as transaction time increases). To summarize, the performance gains strongly depend on the types of transactions and of the configuration. Thus an important conclusion is that the configuration and the placement of the copies should be tuned to selected types of transactions.

Notes

[1] For any two nodes, the same sequence of transactions is executed in the same order.

[2] For our experiments, we do not use the delivery transaction because it is executed in a deferred mode that is not relevant to test the response times on which our measures are based.

References

1. T. Anderson, Y. Breitbart, H. Korth, and A. Wool, "Replication, consistency, and practicality: Are these mutually exclusive"? in SIGMOD Conference, 1998, pp. 484–495.

2. C. Coulon, G. Gaumer, E. Pacitti, and P. Valduriez, The RepDB∗ prototype: Preventive Replication in a Database Cluster, Base de Données Avancées, Montpellier, France, 2004.
3. C. Coulon, E. Pacitti, and P. Valduriez, "Scaling up the preventive replication of autonomous databases in cluster systems," in Int. Conf. on High Performance Computing for Computational Science (VecPar 2004), Valencia, Spain, 2004, Lecture Notes in Computer Science 3402, Springer 2005, pp. 174–188.
4. C. Coulon, E. Pacitti, and P. Valduriez, "Consistency management for partial replication in a high Performance database cluster," in IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS 2005), Fukuoka, Japan, 2005.
5. S. Gançarski, H. Naacke, E. Pacitti, and P. Valduriez, "Parallel processing with autonomous databases in a cluster system," in Int. Conf. on Cooperative Information Systems (CoopIS), 2002.
6. J. Gray, P. Helland, P. O' Neil, and D. Shasha, "The danger of replication and a solution," in ACM SIGMOD Int. Conf. on Management of Data, Montreal, 1996.
7. V. Hadzilacos and S. Toueg, Fault-Tolerant Broadcasts and Related Problems. Distributed Systems, 2nd edition, S. Mullender (Ed.), Addison-Wesley, 1993.
8. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters: early results," in Int. Conf. on distributed Computing Systems (ICDCS), 2002.
9. B. Kemme and G. Alonso, "Don't be lazy be consistent: Postgres-R, a new way to implement database replication," in Int. Conf. on Very Large Databases (VLDB), 2000.
10. Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Middleware based data replication providing snapshot isolation," in ACM SIGMOD Int. Conf. on Management of Data. Baltimore, USA, June 2005.
11. T. Özsu and P. Valduriez, Principles of Distributed Database Systems, 2nd edition, Prentice Hall, 1999.
12. E. Pacitti, P. Minet, and E. Simon, "Replica Consistency in Lazy Master Replicated Databases," Distributed and Parallel Databases, Kluwer Academic, vol. 9, no. 3, 2001.
13. E. Pacitti, T. Özsu, C. Coulon, "Preventive Multi-Master Replication in a cluster of autonomous databases," in Euro-Par Int. Conf., 2003.
14. E. Pacitti and P. Valduriez, "Replicated Databases: Concepts, architectures and techniques," Network and Information Systems Journal, Hermès, vol. 1, no. 3, 1998.
15. Paris Project: http://www.irisa.fr/paris/General/cluster.htm.
16. F. Pedonne, "A schiper: optimistic atomic broadcast," in Distributed Information Systems Conf. (DISC), 1998.
17. C. Plattner and G. Alonso, "Ganymed: scalable replication for transactional web applications," in Proc. of the 5th International Middleware Conference, Toronto, Canada, 2004.
18. F. Raab, TPC-C—The Standard Benchmark for Online transaction Processing (OLTP). The Benchmark Handbook for Database and Transaction Systems, 2nd edition, Morgan Kaufmann, 1993.
19. RepDB∗: Data Management Component for Replicating Autonomous Databases in a Cluster System (released as open source software under GPL). http://www.sciences.univ-nantes.fr/lina/ATLAS/RepDB.
20. A. Sousa, F. Pedone, R. Oliveira, and F. Moura, "partial replication in the database state machine," in IEEE Int. Symposium on Network Computing and Applications (NCA), 2001.
21. A. Sousa, J. Pereira, F. Moura, and R. Oliveira, "Optimistic total order in wide area networks," in *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 190–199.
22. K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," Micro-processors and Microprogramming, vol. 40, 1994.
23. P. Valduriez, "Parallel database systems: Open problems and new issues," Int. Journal on Distributed and Parallel Databases, Kluwer Academic, vol. 1, no. 2, 1993.
24. S. Wu and B. Kemme, "Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation," in IEEE Int. Conference on Data Engineering (ICDE), 2005.

ANNEX D

Scalable and Topology-Aware Reconciliation on P2P Networks

V. Martins, E. Pacitti, M. El-Dick, R. Jiménez-Peris

Accepted paper at Distributed and Parallel Databases, march 2008.

Scalable and Topology-Aware Reconciliation on P2P Networks[1]

Vidal Martins[†], Esther Pacitti[‡], Manal El Dick[‡] and Ricardo Jimenez-Peris [*]

[†]PPGIA/PUCPR – Pontifical Catholic University of Paraná, Brazil
[‡]ATLAS Team, INRIA and LINA, University of Nantes, France
[*] Distributed Systems Lab, Facultad Informatica, Universidad Politécnica de Madrid, Spain
Rua Imaculada Conceição, 1155, Prado Velho, 80.215-901 Curitiba Pr, BRAZIL
2, rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, FRANCE
Campus de Montegancedo s/n, Boadilla del Monte, Madrid 28660, SPAIN
vidal.martins@pucpr.br, {firstname.lastname@univ-nantes.fr}, rjimenez@fi.upm.es

Abstract. Collaborative applications are characterized by high levels of data sharing. Optimistic replication has been suggested as a mechanism to enable highly concurrent access to the shared data, whilst providing full application-defined consistency guarantees. Nowadays, there are a growing number of emerging cooperative applications adequate for Peer-to-Peer (P2P) networks. However, to enable the deployment of such applications in P2P networks, it is required a mechanism to deal with their high data sharing in dynamic, scalable and available way. Previous work on optimistic replication has mainly concentrated on centralized systems. Centralized approaches are inappropriate for a P2P setting due to their limited availability and vulnerability to failures and partitions from the network. In this paper, we focus on the design of a reconciliation algorithm designed to be deployed in large scale cooperative applications, such as P2P Wiki. The main contribution of this paper is a distributed reconciliation algorithm designed for P2P networks (P2P-reconciler). Other important contributions are: a basic cost model for computing communication costs in a DHT overlay network; a strategy for computing the cost of each reconciliation step taking into account the cost model; and an algorithm that dynamically selects the best nodes for each reconciliation step. Furthermore, since P2P networks are built independently of the underlying topology, which may cause high latencies and large overheads degrading performance, we also propose a topology-aware variant of our P2P-reconciler algorithm and show the important gains on using it. Our P2P-reconciler solution enables high levels of concurrency thanks to semantic reconciliation and yields high availability, excellent scalability, with acceptable performance and limited overhead.

1 Introduction

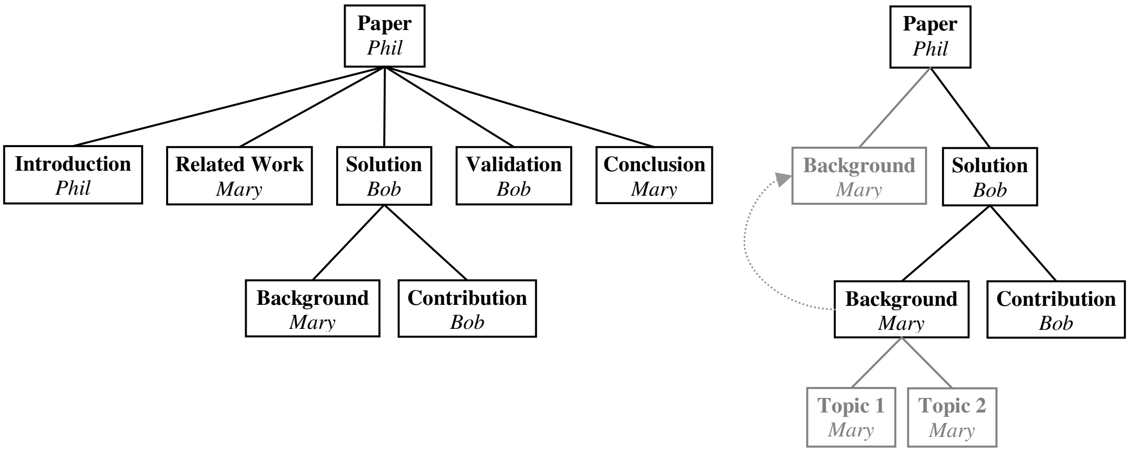
Large-scale distributed collaborative applications are getting common as a result of rapid progress in distributed technologies (grid, P2P, and mobile computing). As an example of such applications, consider a second generation Wiki that works over a peer-to-peer (P2P) network and supports users on the elaboration and maintenance of shared documents in a collaborative and asynchronous manner. Current wiki platforms rely on central servers that are costly and require an organization behind them to buy and maintain the hardware and to maintain the software. A P2P wiki would allow a community of users to collaborate by means of a wiki without having to create an organization to maintain the hardware resources. The computational resources would be provided by the community as a whole. This P2P environment would also allow for people updating the wiki whilst disconnected (e.g. whilst traveling) due to its reconciliation capabilities. In the P2P wiki example, consider that each document is an XML file possibly linked to other documents. Wiki allows collaboratively managing a single document (e.g. a scien-

[1] Work partially funded by ARA “Massive Data” of the French ministry of research (project Respire), the European Strep Grid4All project, the CAPES-COFECUB Daad project and the CNPq-INRIA Gridata project.

tific paper shared by a few of authors) as well as composed, integrated documents (e.g. an encyclopedia or a knowledge base concerning the use of an open source operating system). Although the number of users that update in parallel a document *d* is usually small, the size of the collaborative network that holds *d* in terms of number of nodes may be large. For instance, the document *d* could belong to the knowledge base of the Mandriva Club, which is maintained by more than 25,000 members [Man07] or it could belong to Wikipedia, a free content encyclopedia maintained by more than 75,000 active contributors [Wik07].

Many users frequently need to access and update information even if they are disconnected from the network, e.g. in an aircraft, a train or another environment that does not provide good network connection. This requires that users hold local replicas of shared documents. Thus, a P2P Wiki requires multi-master replication (any replica can perform updates, also known as update-everywhere replication) to assure data availability at anytime. In the multi-master approach, updates made offline or in parallel on different replicas of the same data may cause replica divergence and conflicts, which should be reconciled. In order to resolve conflicts, the reconciliation solution can take advantage of application semantics as illustrated in Example 1. For simplicity, and without loss of generality, this example deals with a single document elaborated by three authors. The document is a scientific paper organized as a tree. Each node (element) in the tree structure corresponds to a section of the paper and holds the name of the responsible author.

Example 1a shows the initial structure of the paper whereas Example 1b shows conflicting updates (in gray) over the initial structure. In Example 1b Phil tries to move the *Background* section under *Paper* thereby changing the *Background* path from *Paper/Solution/Background* to *Paper/Background* while Mary tries to insert two topics under *Background* using the path *Paper/Solution/Background*. If the move operation is accomplished before the insert operations, the *Background*'s path changes so that the insert operations do not find the *Background* element, and therefore such inserts are lost. We can automatically solve this problem by introducing the following application semantics: *update operations precede move operations*. In Example 1, according to this semantics, *Topic 1* and *Topic 2* are inserted in the path *Paper/Solution/Background*, and then the entire subtree under *Background* is moved in such a way that the intents of both users (Phil and Mary) are preserved.



(a) Initial structure (b) Conflicting updates

Example 1. Producing a paper in a collaborative manner

The semantics associated with a P2P collaborative editor can be richer than the simple semantics that we have just discussed. However, we made the example deliberately simple only to show that, by taking advantage of the application semantics on the reconciliation, we can eliminate spurious update conflicts (e.g. insert and move operations over the same element are not really conflicting operations) and we can resolve the real existing conflicts in an automatic manner as users wish.

Optimistic replication is largely used as a solution to provide data availability for these applications. It allows asynchronous updating of replicas such that applications can progress even though some nodes are disconnected or have failed. This enables asynchronous collaboration among users. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled. In most existing solutions [PSM03, SS05] reconciliation is typically performed by a single node (reconciler node) which may introduce bottlenecks. In addition, if the reconciler node fails, the entire replication system may become unavailable.

A theory for centralized semantic reconciliation was set by IceCube [KRSD01, PSM03]. According to this theory, the application semantics can be described by means of constraints between update actions. A *constraint* is an application invariant, *e.g.* a *parcel* constraint establishes the "all-or-nothing" semantics, *i.e.* either all *parcel*'s actions execute successfully in any order, or none does. For instance, consider a user that improves the content of a shared document by producing two *related* actions a_1 and a_2 (*e.g.* a_1 changes a document paragraph and a_2 changes the corresponding translation); in order to assure the "all-or-nothing" semantics, the application should create a parcel constraint between a_1 and a_2. These actions can conflict with other actions. Therefore, the aim of reconciliation is to take a set of actions with the associated constraints and produce a *schedule*, *i.e.* a list of ordered actions that do not violate constraints.

Different from IceCube, we propose a fully distributed approach to perform P2P reconciliation in a distributed hash table (DHT) overlay. The DHT is an efficient distributed architecture to perform key-based information retrieval in a P2P environment. Since our reconciliation solution relies on key-based information retrieval and aims at providing good performance, the choice of DHT seems natural. In addition, variable latencies and bandwidths, typically found in P2P networks, may strongly impact the reconciliation performance once data access times may vary significantly from node to node. Therefore, in order to build a suitable P2P reconciliation solution, topology awareness must be considered. In this paper we also introduce topology awareness to our distributed algorithms, to enhance the reconciliation performance.

In this paper, we propose the *P2P-reconciler*, a new decentralized reconciliation protocol designed for P2P networks that we built upon the reconciliation theory set by IceCube. The main contributions of this paper are: (1) a DHT cost model for computing communication costs of a P2P network using a DHT overlay network; (2) the P2P-reconciler cost model for computing the cost of each reconciliation step based on DHT cost model; (3) the basic *P2P-reconciler* algorithm for semantic reconciliation in P2P networks, which selects the best reconciler nodes based on the P2P-reconciler cost model; (4) a topology-aware approach for the P2P-reconciler (*P2P-reconciler-TA*); (5) experimental results that show: a) that our cost-based approach yields high data availability and excellent scalability, with acceptable performance and limited overhead; b) the important gains of the topology-aware approach (*P2P-reconciler-TA*) compared to the basic *P2P-reconciler.* We also present the validation of P2P-reconciler through its implementation in the APPA architecture [MAPV06] and give the proof of correctness of our protocols.

The rest of this paper is organized as follows. Section 2 describes the basis of the *P2P-reconciler* protocol for semantic reconciliation in P2P networks. Section 3 introduces the DHT cost model. Section 4 describes the P2P-reconciler cost model and the dynamic allocation algorithm for selecting the best reconciler nodes. Section 5 presents the topology aware approach for the *P2P-reconciler.* Section 5 shows implementation and experimental results. Section 6 compares our work with the most relevant related works. Finally, Section 7 concludes this paper. Annex A shows the validation of P2P-reconciler built in the context of APPA and Annex B presents all proofs of correctness of our protocols.

2 P2P semantic reconciliation

In this section, we first introduce the semantic reconciliation as proposed by IceCube. Then, we provide an overview of our P2P-reconciler. Next, we present P2P-reconciler in detail. Finally, we describe our strategy for dealing with the dynamic behavior of the P2P network.

2.1 Representation of application semantics

IceCube describes the application semantics by means of constraints between actions. An *action* is defined by the application programmer and represents an application-specific operation (*e.g.* a write operation on a file or document, or a database transaction). A *constraint* is the formal representation of an application invariant (*e.g.* an update cannot follow a delete). Constraints are classified as follows:

- **User-defined constraints[2]:** user and application can create user-defined constraints to make their intents explicit. The *predSucc*(a_1, a_2) constraint establishes causal ordering between actions (*i.e.* action a_2 executes only after a_1 has succeeded); the *parcel*(a_1, a_2) constraint is an atomic (all-or-nothing) grouping (*i.e.* either a_1 and a_2 execute successfully or none does); the *alternative*(a_1, a_2) constraint provides choice of at most one action (*i.e.* either a_1 or a_2 is executed, but not both).
- **System-defined constraints**[3]: it describes a semantic relation between classes of concurrent actions. The *bestOrder*(a_1, a_2) constraint indicates the preference to schedule a_1 before a_2 (*e.g.* an application for account management usually prefers to schedule credits before debits); the *mutuallyExclusive*(a_1, a_2) constraint states that either a_1 or a_2 can be executed, but not both.

Let us illustrate user- and system-defined constraints with Example 2. In this example, an action is noted a_n^i, where *n* indicates the node that has executed the action and *i* is the action identifier. *T* is a replicated object, in this case, a relational table; *K* is the key attribute for *T*; *A* and *B* are any two other attributes of *T*. T_1, T_2, and T_3 are replicas of *T*. And *parcel* is a user-defined constraint that imposes atomic execution for a_3^1 and a_3^2. Consider that the actions in Example 2 (with the associated constraints) are concurrently produced by nodes n_1, n_2 and n_3, and should be reconciled.

a_1^1: update T_1 set A=a1 where K=k1
a_2^1: update T_2 set A=a2 where K=k1
a_3^1: update T_3 set B=b1 where K=k1
a_3^2: update T_3 set A=a3 where K=k2
Parcel(a_3^1, a_3^2)

Example 2. Conflicting actions on T

In Example 2, actions a_1^1 and a_2^1 try to update a copy of the same data item (*i.e.* *T*'s tuple identified by *k1*). The IceCube reconciliation engine realizes this conflict and asks the application for the semantic relationship involving a_1^1 and a_2^1. As a result, the application analyzes the intents of both actions, and, as they are really in conflict (*i.e.* n_1 and n_2 try to set the same attribute with distinct values), the application produces a *mutuallyExclusive*(a_1^1, a_2^1) *system-defined constraint* to properly represent this semantic dependency. Notice that from the point of view of the reconciliation engine a_3^1 also conflicts with a_1^1 and a_2^1 (*i.e.* all these actions try to update a copy of the same data item). However, by analyzing actions' intents, the application realizes that a_3^1 is semantically independent of a_1^1 and a_2^1 as a_3^1 tries to update another attribute (*i.e.* *B*). Therefore, in this case no system-defined constraints are produced. Actions a_3^1 and a_3^2 are involved in a *parcel user-defined constraint*, so they are semantically related.

[2] *User-defined* constraints are called *log* constraints by IceCube. We prefer *user-defined* to emphasize the user intent.

[3] *System-defined* constraints are called *object* constraints by IceCube. We use *system-defined* in contrast to user intents.

The aim of reconciliation is to take a set of actions with the associated constraints and produce a *schedule*, *i.e.* a list of ordered actions that do not violate constraints. In order to reduce the schedule production complexity, the set of actions to be ordered is divided into subsets called *clusters*. A cluster is a subset of actions related by constraints that can be ordered independently of other clusters. Therefore, the *global schedule* is composed by the concatenation of clusters' ordered actions. To order a cluster, IceCube performs iteratively the following operations:

- Select the action with the highest merit from the cluster and put it into the schedule. The merit of an action is a value that represents the estimated benefit of putting it into the schedule (the larger the number of actions that can take part in a schedule containing a_i^n is, the larger the merit of a_i^n will be). If more than one action has the highest merit (different actions may have equal merits), the reconciliation engine selects randomly one of them.
- Remove the selected action from the cluster.
- Remove from the cluster the remaining actions that conflict with the selected action.

This iteration ends when the cluster becomes empty. As a result, cluster's actions are ordered. In fact, several alternative orderings may be produced until finding the best one. Therefore, our approach does not employ timestamps. Indeed, actions are ordered based on the application semantics in such a way that the number of actions in the final schedule is maximized.

Let us illustrate our ordering procedure with an example. Consider the actions a_1, a_2, a_3 and the associated constraints *predSucc*(a_1, a_2) and *mutuallyExclusive*(a_1, a_3). The constraint *predSucc*(a_1, a_2) states that a_1 must precede a_2 in the schedule according to the application semantics. Similarly, the constraint *mutuallyExclusive*(a_1, a_3) states that either a_1 or a_3 should be scheduled, but not both. In this scenario, a_2 cannot be the first action in the schedule since it should be preceded by a_1. Thus, in the first iteration, we must choose between a_1 and a_3. The merit of a_1 is 1 since a_1 can be followed by a_2, whereas the merit of a_3 is 0. Therefore, the first action we put into the schedule based on actions' merits is a_1. Due to the *mutuallyExclusive*(a_1, a_3) constraint, the selection of a_1 leads to the removal of a_3 and, as a result, a_2 remains alone to the next iteration. Thus, in the second iteration, we put a_2 into the schedule. In this example, the final schedule is ordered as follows: [a_1, a_2].

2.2 Overview on how P2P-reconciler works

We assume that P2P-reconciler is used in the context of a virtual community which requires a high level of collaboration and it is deployed on a reasonable number of nodes (typically hundreds or even thousands of interacting users) [WIO97]. The P2P network we consider consists of a set of nodes which are organized as a distributed hash table (DHT) [RFHK+01, SMKK+01]. A DHT provides a hash table abstraction over multiple computer nodes. Data placement in the DHT is determined by a hash function which maps data identifiers into nodes.

In our solution, the replicated object is generic, i.e. it can be a relational table, an XML document, etc. We call *object item* a component of the object, e.g. a tuple in a relational table or an element in an XML document. A *replica* is a copy of an object (*e.g.* copy of a relational table or an XML document) while a *replica item* is a copy of an object item (*e.g.* a copy of a tuple or XML element). We assume *multi-master* replication, *i.e.* multiple replicas of an object *R*, noted R_1, R_2, …, R_n, are stored in different nodes which can read or write R_1, R_2, …, R_n. Conflicting updates are expected, but it is assumed that the application tolerates some level of replica divergence until reconciliation.

We have structured the P2P reconciliation in 6 distributed steps in order to maximize parallelism and assure independence between parallel activities. This structure improves reconciliation performance and availability (*i.e.* if a node fails, the activity it was executing is assigned to another available node).

With P2P-reconciler, data replication proceeds basically as follows. First, nodes execute local actions to update a replica of an object while respecting user-defined constraints. Then, these actions (with the associated constraints) are stored in the DHT based on the object's identifier. Finally, reconciler nodes retrieve actions and constraints from the DHT and produce the global schedule, by reconciling conflicting actions in 6 distributed steps based on the application semantics. This schedule is locally executed at every node, thereby assuring eventual consistency [SBK04, SS05]. The replicated data is eventually consistent if, when all nodes stop the production of new actions, all nodes will eventually reach the same value in their local replicas.

In this protocol, we distinguish three types of nodes: the *replica node*, which holds a local replica; the *reconciler node*, which is a replica node that participates in distributed reconciliation; and the *provider node*, which is a node in the DHT that holds data consumed or produced by the reconcilers (*e.g.* the node that holds the schedule is called *schedule provider*).

We concentrate the reconciliation work in a subset of nodes (the reconciler nodes) in order to maximize performance. If we do not limit the number of reconcilers, the following problems take place. First, provider nodes and the network as a whole become overloaded due to a large number of messages aiming to access the same subset of DHT data in a very short time interval. Second, nodes with high-latencies and low-bandwidths can waste a lot of time with data transfer, thereby hurting the reconciliation response time. Our strategy does not create improper imbalances in the load of reconciler nodes since reconciliation activities are not computationally intensive.

2.3 Detailed presentation of P2P-reconciler

We now present P2P-reconciler in deeper detail. First, we introduce the reconciliation objects necessary for P2P-reconciler. Then, we describe the six steps of the reconciliation algorithm. Finally, we illustrate this algorithm at work over a Chord DHT.

2.3.1 Reconciliation objects

Data managed by P2P-reconciler during reconciliation are held by reconciliation objects that are stored in the DHT giving the object identifier. To enable the storage and retrieval of reconciliation objects, each reconciliation object has a unique identifier. P2P-reconciler uses the following reconciliation objects:

- **Action log *R* (noted L_R):** it holds all actions that try to update any replica of the object *R* (in the Example 2, all updates on *T*'s tuples performed on T_1, T_2 or T_3 are stored in L_T). Notice that an action is first stored locally in the replica node and afterwards in the provider node that holds L_R. In Example 2, only one action log is involved (L_T) because a single object is replicated (*T*). The action log makes up the input for reconciliation.
- **Clusters set (noted *CS*):** recall that a cluster contains a set of actions related by constraints, and can be ordered independently from other clusters when producing the global schedule. All clusters produced during reconciliation are stored in the *clusters set* reconciliation object.
- **Action summary (noted *AS*)**: it captures semantic dependencies among actions, which are described by means of constraints. In addition, the action summary holds relationships between actions and clusters, so that each relationship describes an action membership (an action is a *member* of one or more clusters). An action membership is a pair of values (a_n^i, C_j), where a_n^i represents an action to be reconciled, and C_j indicates a cluster to which a_n^i belongs.
- **Schedule (noted *S*)**: it contains an ordered list of actions, which is composed from the concatenation of clusters' ordered actions. Thus, we denote a schedule reconciliation object as $S = S_1 \oplus S_2 \dots \oplus S_n$, where each S_i represents the sub-list of ordered actions coming from the cluster C_i and $\oplus$ means concatenation.

Reconciliation objects are guaranteed to be available using known DHT replication solutions [APV07, KWR05]. P2P-reconciler's liveness relies on the DHT liveness.

2.3.2 P2P-reconciler algorithm

P2P-reconciler executes reconciliation in 6 distributed steps as shown in Figure 1. Any connected node *n* can try to start reconciliation by inviting other available nodes to engage with it. Only one reconciliation can run at a time, thus if reconciliation is already running when *n* try to start it, *n* does not succeed and it must try again later. However, if *n* succeeds, the 1st step (node allocation) allocates a subset of engaged nodes to step 2, another subset is allocated to step 3, and so forth until the 6th step (details about node allocation are provided in the next sections). Nodes at step 2 start reconciliation. The output produced at each step becomes the input to the next one. In the following, we describe the activities performed in each step, and we illustrate parallel processing by explaining how these activities can be executed simultaneously by two reconciler nodes, n_1 and n_2.

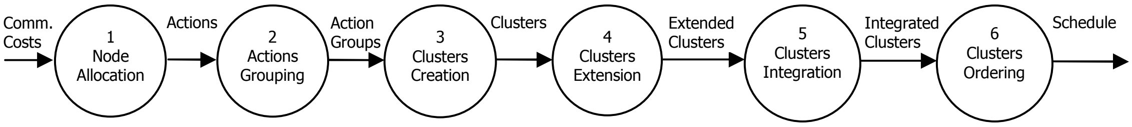


Figure 1. P2P-reconciler steps

- **Step 1 – node allocation:** a subset of connected replica nodes is selected to proceed as reconciler nodes (detailed in Section 4).
- **Step 2 – actions grouping:** reconcilers take actions from the action log and put actions that try to update common object items into the same group. In Example 2, suppose that n_1 takes $\{a_1^1, a_2^1\}$ and n_2, $\{a_3^1, a_3^2\}$ as input. By hashing the identifiers of the replica items handled by these actions (respectively k1, k1, k1, and k2), n_1 puts a_1^1 and a_2^1 into the group G_1 (a_1^1 and a_2^1 handle the same object item identified by k1) whereas n_2 put a_3^1 into G_1 and a_3^2 into G_2 (a_3^1 and a_3^2 handle respectively the object items identified by k1 and k2). Thus, groups $G_1 = \{a_1^1, a_2^1, a_3^1\}$ and $G_2 = \{a_3^2\}$ are produced in parallel and are stored in the *action log* reconciliation object (L_T). Clearly, in order to identify conflicting actions, we need to be able to distinguish *object items* from each other. This can be done by using keys or other means. For instance, two xml elements associated with different paths can be considered different *object items*. In this case, the element path works as an identifier. From the perspective of correctness, it does not matter if two distinct object items are considered the same object item by the reconciliation engine since the responsibility of judging conflicts is assigned to the application, which deeply knows the context. The responsibility of the reconciliation engine is to put together actions that are *potentially* in conflict. Of course, from the perspective of performance, the lack of a good strategy for differentiating object items can lead to larger clusters and, as a result, worse performance.
- **Step 3 – clusters creation:** reconcilers take action groups from the action log and split them into clusters of semantically dependent conflicting actions (two actions a_1 and a_2 are semantically independent if the application judges safe to execute them together, in any order, even if they update a common object item; otherwise, a_1 and a_2 are semantically dependent); system-defined constraints are created to represent the semantic dependencies detected in this step; these constraints and the action memberships that describe the association between actions and clusters are included in the action summary; clusters produced in this step are stored in the clusters set. In Example 2, consider that n_1 takes G_1 and n_2 takes G_2 as input. In this case, n_1 splits G_1 into clusters $C_1 = \{a_1^1, a_2^1\}$ (a *mutuallyExclusive*(a_1^1, a_2^1) system-defined constraint is produced to represent the semantic dependency between a_1^1 and a_2^1) and $C_2 = \{a_3^1\}$ (a_3^1 tries to update the same object item as a_1^1 and a_2^1, but a_3^1 touches a distinct attribute, B; therefore, a_3^1 does not conflict with a_1^1 and a_2^1); at the same time, n_2 turns G_2 into cluster $C_3 = \{a_3^2\}$. All these clusters are stored in the *clusters set* reconciliation object (*CS*). In addition, n_1 stores in the action summary (*AS*) the *mutuallyExclusive*(a_1^1, a_2^1) constraint and the fol-

lowing memberships: {(a_1^1, C_1), (a_2^1, C_1), (a_3^1, C_2)}. Similarly, n_2 stores in *AS* this set of memberships: {(a_3^2, C_3)}.

- **Step 4 – clusters extension:** user-defined constraints are not taken into account in clusters creation (e.g. although a_3^1 and a_3^2 belong to a *parcel*, the previous step does not put them into the same cluster, because they do not update a common object item). Thus, in this step, reconcilers extend clusters by adding to them new conflicting actions, according to user-defined constraints. These extensions lead to new relationships between actions and clusters, which are represented by new action memberships; the new memberships are included in the action summary. In Example 2, assume that n_1 takes C_1 = {a_1^1, a_2^1} as input whereas n_2 takes C_2 = {a_3^1} and C_3 = {a_3^2} (each node deals with 2 actions). Then, n_1 realizes that C_1 does not need extensions, because its actions are not involved in user-defined constraints; in parallel, due to the parcel constraint, n_2 extends C_2 and C_3 as follows: $C_2 \leftarrow C_2 \cup$ {a_3^2}, and $C_3 \leftarrow C_3 \cup$ {a_3^1}. In addition, n_2 updates the action summary with these action memberships: {(a_3^2, C_2), (a_3^1, C_3)}.

- **Step 5 – clusters integration:** clusters extensions lead to cluster overlapping (an overlap occurs when the intersection of two clusters results in a non-null set of actions); in this step, reconcilers bring together overlapping clusters. In Example 2, consider that n_1 takes {(a_3^1, C_2), (a_3^1, C_3), (a_3^2, C_2), (a_3^2, C_3)} as input whereas n_2 takes {(a_1^1, C_1), (a_2^1, C_1)} (each node deals with the memberships of 2 actions). Thus, n_1 realizes that a_3^1 is a member of C_2 and C_3, so n_1 integrates them as follows: $C_4 \leftarrow C_2 \cup C_3$ = {a_3^1, a_3^2}; at the same time, n_2 realizes that a_1^1 and a_2^1 have just one membership, so n_2 does not perform integrations. At this point, clusters become mutually-independent, i.e. there are no constraints involving actions of distinct clusters.

- **Step 6 – clusters ordering:** in this step, reconcilers take clusters from the clusters set and order clusters' actions; the ordered actions associated with each cluster are stored in the *schedule* reconciliation object (*S*); the concatenation of all clusters' ordered actions makes up the global schedule that is executed by all replica nodes. In Example 2, suppose that n_1 takes C_1 as input whereas n_2 takes C$_4$. As a result, n_1 produces the sub-list of ordered actions S_1 = [a_1^1], because C_1 actions are mutually exclusive; in parallel, n_2 produces the sub-list of ordered actions S_4 = [a_3^1, a_3^2], because C_4 actions are involved in a parcel constraint. The global schedule is $S \leftarrow S_1 \oplus S_4$ = [a_1^1, a_3^1, a_3^2].

At every step, the P2P-reconciler algorithm takes advantage of data parallelism, i.e. several nodes perform simultaneously independent activities on a distinct subset of actions (e.g. ordering of different clusters). No centralized criterion is applied to partition actions. Indeed, whenever a set of reconciler nodes requests data from a provider, the provider node naively supplies reconcilers with about the same amount of data (the provider node knows the maximal number of reconcilers because it receives this information from the node that launches reconciliation).

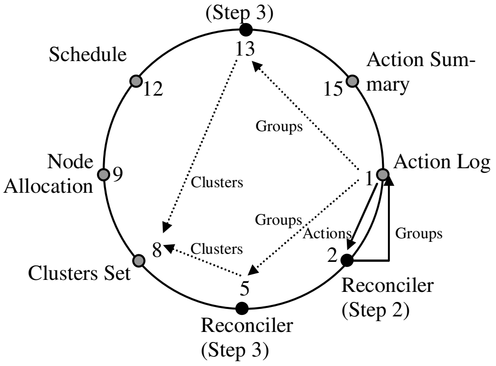


Figure 2. P2P-reconciler at work

2.3.3 P2P-reconciler at work

We now illustrate the execution of the P2P-reconciler algorithm over a Chord DHT network. For simplicity, we consider only its first 3 steps and a few nodes at work. Figure 2 shows 8 nodes and their respective roles in the reconciliation protocol. All of them are replica nodes. Reconciliation objects are stored at provider nodes according to the hashed values associated with the reconciliation object identifiers (e.g. Chord maps a hashed value *v* to the first node that has an identifier equal to or greater than *v* in the circle of ordered node identifiers). In this example, we assume that Chord maps the hashed value of the action log identifier to node 1; using the same principle, the clusters set, the schedule and the action summary are mapped respectively to nodes 8, 12 and 15. Finally, node 9 is responsible for allocating reconcilers.

Any node can start the reconciliation by triggering the step 1 of P2P-reconciler at the appropriate node (*e.g.* node 9), which selects the best reconcilers and notifies them about the steps they should perform. In our example, node 9 selects node 2 to execute step 2, and it selects nodes 5 and 13 to perform step 3 (details about node allocation are provided in the next sections).

Node 2 starts the step 2 of reconciliation by retrieving actions from the action log (stored at node 1) in order to arrange them in groups of actions on common object items. At the same time, nodes 5 and 13 begin step 3 by requesting action groups to node 1; these requests are held in a queue at node 1 while action groups are under construction. When node 2 stores action groups at action log, node 1 replies the requests previously queued by nodes 5 and 13. At this moment, step 2 is terminated and step 3 proceeds. Notice that each reconciler works on independent data (*e.g.* nodes 5 and 13 receive distinct action groups from node 1). To assure this independence, provider nodes segment the data they hold based on the number of reconcilers (*e.g.* node 1 creates two segments of action groups, one for node 5 and another for node 13).

When step 2 terminates, nodes 5 and 13 receive action groups from node 1 and produce the corresponding clusters of actions, which are stored at node 8. In turn, node 8 replies requests for clusters that reconcilers of step 4 have previously queued; and so forth, until the end of step 6.

2.4 Dealing with dynamism

Whenever distributed reconciliation takes place, a set of nodes N_d may be disconnected. As a result, the global schedule is not applied by nodes of N_d. Moreover, actions produced by N_d nodes and not yet stored in the P2P network are not reconciled. In this subsection, we explain how P2P-reconciler assures eventual consistency despite disconnections. Fault-tolerance aspects are not studied in this paper since they are orthogonal to what is presented.

We need a new reconciliation object to assure eventual consistency in the presence of disconnections. Thus, we define *schedule history*, noted *H*, as a reconciliation object that stores a chronological sequence of schedules' identifiers produced by reconciliations ($H = (S_{id1}, \ldots, S_{idn})$). A replica node can check whether it is *up to date* by comparing the identifier of the last schedule it has locally executed with S_{idn}.

P2P-reconciler deals with dynamism as follows. Each node locally stores the identifier of the last schedule it has locally executed (noted S_{last}). In addition, every node knows the schedule history's unique identifier. Thus, when a node *n* of N_d reconnects, it proceeds as follows: (1) *n* checks whether S_{last} is equal to S_{idn}, and, if not (*i.e. n*'s replicas are out of date), *n* locally applies all schedules that follow S_{last} in history *H*; (2) actions locally produced by *n* and not yet stored in the P2P network are put into the action log for later reconciliation.

3 DHT cost model

A DHT network is usually built on top of the Internet, which consists of nodes with variable latencies and bandwidths. As a result, the network costs involved in DHT data accesses may vary significantly from node to node and have a strong impact in the reconciliation performance. Thus, network costs should be considered to perform reconciliation efficiently. In this section, we propose a basic cost model for computing communication costs in DHTs. On top of it, we can build customized cost models (e.g. we elaborated a customized cost model for selecting reconciler nodes to P2P-reconciler – Section 4).

In the basic cost model, we define communication costs (henceforth costs) in terms of latency and transfer times, and we assume links with variable latencies and bandwidths. In order to exploit bandwidth, the application behavior in terms of data transfer should be known. Since this behavior is application-specific, we exploit bandwidth in higher-level customized models.

Most DHT data access operations consist of a lookup for finding the address of the node *n* that holds the requested information followed by direct communication with *n* [HHLT$^+$03]. In the lookup step, several hops may be performed according to nodes' neighborhoods. Therefore, our DHT cost model relies on three metrics: lookup cost, direct cost, and transfer cost. The *lookup cost*, noted $lc(n, id)$, is the latency time spent in a lookup operation launched by node *n* to find the data item identified by *id*. Similarly, *direct cost*, noted $dc(n_i, n_j)$, is the latency time spent by node n_i to directly access n_j. And the *transfer cost*, noted $tc(n_i, n_j, d)$, is the time spent to transfer the data item *d* from node n_i to node n_j, which is computed based on *d*'s size and the bandwidth between n_i and n_j.

3.1.1 Lookup cost

Lookup costs change dynamically as nodes join and leave the P2P network. In this subsection, we show how to compute lookup costs and deal with dynamic changes.

Node *n* could easily compute the lookup cost $lc(n, id)$ by executing the lookup operation and measuring the associated time. However, this approach overloads the node that replies the lookup operation as it receives a lot of lookup messages. Furthermore, the network is overloaded. To avoid these problems, we propose that each node computes its lookup costs incrementally, by taking advantage of cost information held by its neighbors. With this approach, a node *n* only keeps the lookup costs to a few of identifiers (i.e. one identifier for each reconciliation object); in addition, *n* keeps the direct costs to a few of nodes (i.e. *n*'s neighbors). It would be unfeasible and not recommendable to keep information about the full identifier space or all nodes. Our approach is feasible because in a DHT a node *n* looks for an identifier *id* by communicating with the *n*'s neighbor that is closest to *id*.

We illustrate our solution with an example. In Figure 3a, let n_4 be a node that replies lookup operations searching for id=x; let arrows indicate the route of a lookup operation (e.g. if n_2 looks for *x* it makes this route: $n_2 \rightarrow n_3 \rightarrow n_4$); let a number over an arrow be the latency between the associated nodes. In this example, the lookup cost $lc(n_2, x)$ is 100 (i.e. 40 + 60), and $lc(n_1, x)$ is 150 (i.e. 50 + 40 + 60). Instead of executing the lookup operation to compute $lc(n_1, x)$, n_1 can ask n_2 for $lc(n_2, x)$ and add to this cost the latency between n_1 and n_2 (i.e. $lc(n_1, x) \leftarrow lc(n_2, x) + 50$). The advantage of this incremental approach is locality and to avoid network overload.

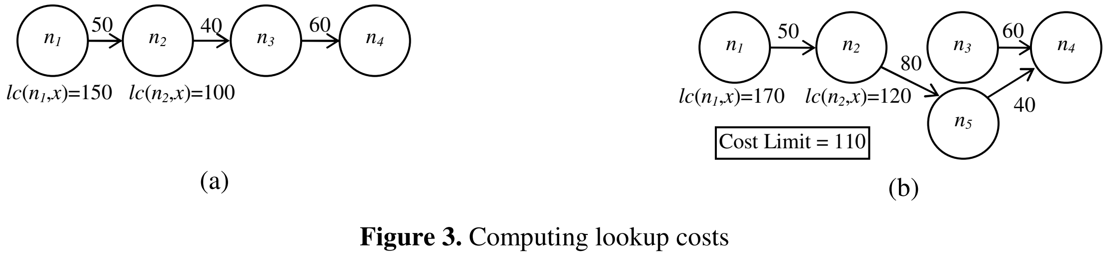


Figure 3. Computing lookup costs

Joins and leaves change the neighborhoods of nodes and, accordingly, the routes of lookup messages. As a result, lookup costs must be refreshed. However, we should avoid the refreshment at distant nodes to avoid network overload. To cope with this problem, we introduce two definitions:

- **Cost limit**: it is the maximal acceptable cost for looking up an identifier. The meaning of *acceptable cost* relies on the application on top of DHT. For instance, in the case of P2P-reconciler, which selects a subset of replica nodes to proceed as reconciler nodes, it is not acceptable that the lookup cost of a particular reconciler overtakes the average lookup cost of the P2P network as a whole, because the number of reconcilers is usually much smaller than the number of replica nodes.

– **Relevant joins and leaves**: a join or leave is *relevant* for a node *n* if it changes the cost for looking up an identifier in which *n* is interested, such that the old or the new lookup cost does not overtake *cost limit*. Nodes refresh their lookup costs only in the presence of relevant joins and leaves.

We illustrate our approach for refreshing lookup costs with an example. In Figure 3b, let cost limit be 110; and consider that n_5 joins the DHT of Figure 3a taking the place of n_3 in the route towards *id*=*x*. The join of n_5 is relevant only to n_2 as n_2 updates $lc(n_2, x)$ from 100 (a value that does not overtake cost limit) to 120. In contrast, the join of n_5 is not relevant to n_3 and n_4 since the associated lookup costs remain unchanged. This join is not relevant to n_1 either, because both, the old lookup cost (i.e. 150) and the new one (i.e. 170), overtake cost limit. Thus, n_1, n_3 and n_4 do not participate in the refresh operation.

3.1.2 Direct cost

Direct costs change dynamically as nodes join and leave the P2P network. In this subsection, we show how to compute direct costs and deal with dynamic changes.

We first define *home*(*id*) as the provider node that holds the identifier *id*. The direct cost *dc*(*n*, *home*(*id*)) represents the latency time spent by node *n* to directly access *home*(*id*). This cost can be exactly computed or estimated. With the exact approach, *n* measures the latency between *n* and *home*(*id*). In contrast, with the estimated approach, *n* measures the latencies between *n* and a subset of nodes and then computes the corresponding average value, which represents the estimated latency between *n* and *home*(*id*). The exact approach is precise, but it can overload *home*(*id*) as it becomes a central point of access for a lot of nodes. On the other hand, the estimated approach does not rely on accessing *home*(*id*), thereby avoiding its overload, but it is not precise. We compare both approaches and, due to the small difference between the corresponding reconciliation times (i.e. 7%), we consider that the estimated approach should be employed in order to avoid overload.

Notice that the estimated approach requires a subset of nodes to estimate the latency between *n* and *home*(*id*). This subset should be *n*'s neighbors for DHTs whose neighborhoods do not rely on physical distances among nodes (e.g. Chord) since, in this case, estimation is not biased and the information needed is already available at *n* (cost zero). However, if the DHT is location-aware, i.e. *n*'s neighbors are closer to *n* than other nodes (e.g. CAN with design improvements), the use of *n*'s neighbors would lead to a biased estimation. Thus, in this case, the subset of nodes should be randomly selected from a bootstrap list (list of nodes that are likely connected).

Joins and leaves may change the *home*(*id*). Thus, direct costs must also be refreshed. In our solution, *dc*(*n*, *home*(*id*)) is refreshed at node *n* whenever *home*(*id*) changes and the associated lookup cost (i.e. *lc*(*n*, *id*)) is smaller than the cost limit. To compute the refreshed value, we use the same strategy employed for computing the initial value. The principle of this approach is to avoid the execution of refreshment operations at far distant nodes, and its advantage is to avoid network overload.

4 P2P-reconciler node allocation

The first step of P2P-reconciler aims to select the best nodes to proceed as reconcilers in order to maximize performance. The number of reconcilers has a strong impact on the reconciliation time. Thus, this section concerns the estimation of the optimal number of reconcilers per step as well as the allocation of the best nodes. We first present how to determine the maximal number of reconciler nodes. Then, we introduce the P2P-reconciler cost model for computing the cost of each reconciliation step. Next, we describe how the cost provider node selects reconcilers based on P2P-reconciler cost model. Finally, we present our approach for managing the dynamic behavior of P2P-reconciler costs.

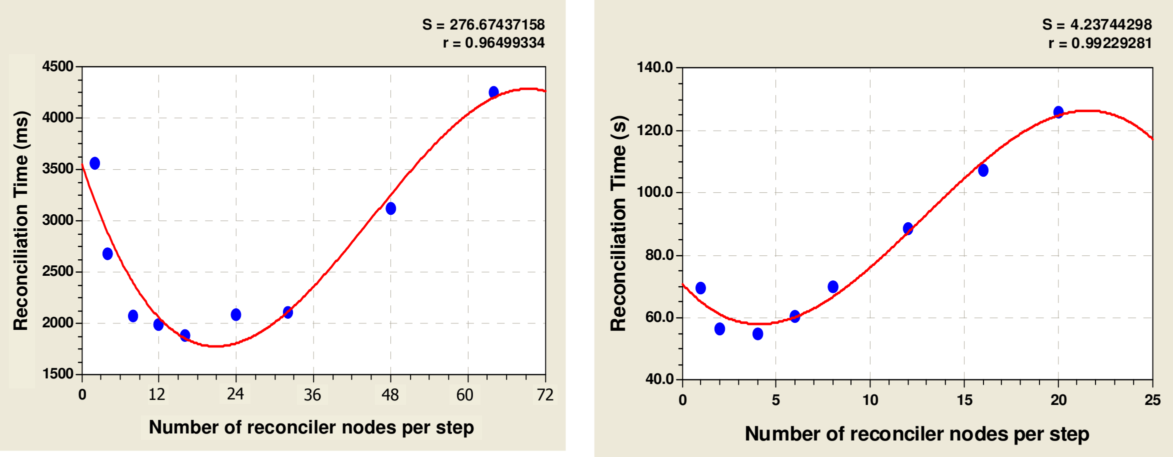
4.1 Determining the number of reconcilers

At the beginning of reconciliation, a subset of replica nodes must be allocated to P2P-reconciler steps in order to proceed as reconciler nodes. This allocation is dynamic as it depends on the reconciliation context (i.e. number of actions to be reconciled, network properties, etc.). Since P2P-reconciler is distributed, we can increase the number of reconciler nodes to reduce the reconciliation time. However, as we increase the number of reconcilers we also increase the number of exchanged messages and the work performed by provider nodes. As a result, beyond a given *bound*, increasing the number of reconcilers yields the opposite effect: the reconciliation time augments. In order to compute this bound, that represents the maximal number of reconcilers per step, we perform the following activities.

- First, we configure the reconciliation context by setting up some parameters (e.g. number of actions, number of connected replica nodes, number of reconciler nodes, minimal and maximal network latencies, network bandwidths), and then we simulate reconciliation several times, taking as a result a reconciliation sample. For each simulation, we change the topology of the physical and overlay networks or the set of actions to be reconciled or both, always respecting the parameters' values. A simulation runs locally in a single node. An important aspect is that only network communication is simulated (everything else is done by the actual P2P-reconciler protocol).
- Second, we search an equation $y = f(x)$ that describes the reconciliation behavior by performing a polynomial regression [KKMN98] with sample's data. This equation allows us to forecast the reconciliation time of any reconciliation in the same context. The independent variable x is the number of reconciler nodes whereas the dependent variable y is the reconciliation time. We have always got an excellent correlation between x and y by using a polynomial of degree 3 as shown in Figure 4 (in Figure 4, r is the correlation coefficient and it can vary from 0 to 1; 1 denotes a perfect correlation).
- Third, we compute the derivative equation $y' = f'(x)$; this derivative equation enables us to find which value of x produces the minimal value of y. The point (x, y) where y is minimal is called *minimal point*. Since $f'(x)$ is a second-order polynomial as shown in Figure 4, the curve described by $f(x)$ has exactly one minimal point and one maximum point, which correspond to the roots of $f'(x)$.
- Finally, we calculate the minimal point, which represents the number of reconcilers that minimizes the reconciliation time in the given context. This optimal number of reconcilers is the same for every step. At the beginning of our research we tried to find a different number of reconcilers for each step, but we realized that, in practice, this approach did not improve our results. This can be explained by the trade-off between the providers' workload and the network traffic. The best performance is achieved with a number of reconcilers that optimize the network traffic without overloading the provider nodes.

The larger the number of actions to be reconciled and the higher the network speed are, the larger the maximal number of reconcilers per step. In order to determine the number of reconcilers, we do not fix parameters like network bandwidth, latency, and others. Everything is variable. Regarding network bandwidth, we provide a list of possible values varying from 64 Kbps to 20 Mbps. The bandwidth values follow a Pareto distribution so that low bandwidths are more frequently assigned than high bandwidths. This means, the network topology of a single experiment has variable bandwidths. With respect to latencies, we provide the minimum and maximum latencies corresponding to the type of network we intend to simulate (e.g. Cluster, Grid, Internet, etc.) and, inspired by BRITE [Bri08], we place nodes in a 2-dimensional Cartesian coordinate space, called *plane*, so that the latency between two nodes n_i and n_j is proportional to the geometrical distance between n_i and n_j on the plane. This approach assures variable latencies among nodes. Details about our network simulator can be found in [Sim07].

We address the problem of obtaining sufficient samples by relying on peers' collaboration. In [Mar07] we show that the only information needed to compute the maximal number of reconcilers per step is the equation $y' = f'(x)$; after determining this equation, sample's data are disposable. Therefore, in order to obtain this equation, a node *n* proceeds as follows. First, *n* locally looks for an existing equation that corresponds to its need. If *n* does not succeed, *n* requests the equation's coefficients from its neighbors. If no neighbor can provide this information, *n* locally produces a reconciliation sample and computes the associated equation, which is stored at *n* for future reuse. With time, a lot of samples are produced and the associated equations are shared among peers. As a result, the simulation is often avoided and, in this case, there is no overhead. For this reason, we did not consider integrating the simulation costs into global reconciliation cost. Our experiments have shown that the time needed to produce a complete sample and compute the associated equation varies from 20s to 60s depending on the number of actions to be reconciled.



y = -0.00035x^3 + 0.237x^2 -37.887x + 3551.567
y' = -0.0011x^2 + 0.4741x – 37.8870

(a) High-speed network

y = -0.026x^3 + 0.985x^2 – 6.740x + 70.803
y' = -0.078x^2 + 1.970x – 6.740

(b) Low-speed network

Figure 4. Examples of polynomial regression

Finally, the relationship between the increase on the provider/reconciler interactions and the optimal number of reconcilers is not straightforward. It depends on the processing power of provider nodes, the amount of data to be transferred, the subjacent bandwidths and latencies, and so forth. However, it is very difficult to compute the optimal number of reconcilers analytically. Therefore, we decided to take advantage of simulation by producing a powerful simulator, which works with variable bandwidths, latencies, number of nodes, and data transfer sizes. In addition, our simulator is very easy to configure.

4.2 P2P-reconciler cost model

The P2P-reconciler cost model is built on top of the DHT cost model by taking into account each reconciliation step and defining a new metrics: node step cost. The *node step cost*, noted *cost*(*i, n*), is the sum of lookup, direct access, and transfer costs estimated by node *n* for executing step *i* of P2P-reconciler algorithm.

By analyzing the P2P-reconciler behavior in terms of lookup, direct access, and data transfer operations at every step, we produced a cost formula for each step of P2P-reconciler, which are shown in Table 1. There is no formula associated with step 1 because it is not performed by reconciler nodes.

As an example, let us explain *cost*(2, *n*). In the second step of P2P-reconciler (*i*=2), node *n* takes actions from the action log *R* (L_R) and arranges them in groups of actions that try to update common object items; these groups are stored at L_R. Thus, the first term in the associated formula ($lc(n,L_R)$) represents the lookup cost for finding L_R provider. The second term ($2 \times dc(n,n_{LR})$) corresponds to the direct costs for taking actions from L_R provider (request and reply). The third term ($tc(n_{LR}, n, actSet)$ stands for the transfer cost of the action set from n_{LR} to *n*. The fourth term ($lc(n,L_R)$) represents the lookup cost for finding again L_R provider. The fifth term ($dc(n,n_{LR})$) corresponds to the direct cost for storing groups in L_R provider (only request). And the last term ($tc(n, n_{LR}, grpSet)$) stands for the transfer cost of the action groups produced in this step from n to n_{LR}. Similarly, all formulas can be explained.

Step *i*	***Cost(i, n)***
2	$lc(n, L_R) + 2\times dc(n, n_{LR}) + tc(n_{LR}, n, actSet) + lc(n, L_R) + dc(n, n_{LR}) + tc(n, n_{LR}, grpSet)$
3	$lc(n, L_R) + 3\times dc(n, n_{LR}) + tc(n_{LR}, n, grpSet) + lc(n, CS) + 2\times dc(n, n_{CS}) +$ $tc(n, n_{CS}, [cluSet + cluIds]) + lc(n, AS) + dc(n, n_{AS}) + tc(n, n_{AS}, [sdcSet + m_3Set])$
4	$lc(n, CS) + 3\times dc(n, n_{CS}) + tc(n_{CS}, n, cluSet) + 2\times lc(n,AS) + 3\times dc(n, n_{AS}) + tc(n, n_{AS}, m_4Set)$
5	$lc(n, AS) + 3\times dc(n, n_{AS}) + tc(n_{AS}, n, mSet) + lc(n, CS) + dc(n, n_{CS}) + tc(n, n_{CS}, ovlCluSet)$
6	$lc(n, CS) + 3\times dc(n, n_{CS}) + tc(n_{CS}, n, itgCluSet) + lc(n, AS) + 2\times dc(n, n_{AS}) +$ $tc(n_{AS}, n, sumActSet) + lc(n, S) + dc(n,n_S) + tc(n, n_S, ordActSet)$

Table 1. P2P-reconciler cost model

4.3 Node allocation

Node allocation is the first step of P2P-reconciler protocol as shown in Figure 1. It aims to select for every succeeding step a set of reconciler nodes that can perform reconciliation with good performance. In this subsection, we define a new reconciliation object needed in node allocation, and then we describe how reconciler nodes are chosen, and we illustrate that with an example.

We define *communication costs*, noted *CC*, as a reconciliation object that stores the *node step costs* estimated by every replica node and used to choose reconcilers before starting reconciliation.

The node that holds *CC* in the DHT at a given time is called *cost provider*, and it is responsible for allocating reconcilers. The allocation works as follows. Replica nodes locally estimate the costs for executing every P2P-reconciler step, according to the P2P-reconciler cost model, and provide this information to the cost provider. The node that starts reconciliation computes the maximal number of reconcilers per step (*maxRec*), as described in Section 4.1, and asks the cost provider for allocating at most *maxRec* reconciler nodes per P2P-reconciler step. As a result, the cost provider selects the best nodes for each step and notifies these nodes of the P2P-reconciler steps they should execute.

In our solution, the cost management is done in parallel with reconciliation. Moreover, it is network optimized since replica nodes do not send messages to the cost provider, informing about their estimated costs, if the node step costs overtake the *cost limit*. For these reasons, the cost provider does not become a bottleneck.

We now illustrate the allocation algorithm using an example. Table 2 shows the lookup and direct costs of 4 nodes belonging to a Chord DHT network [SMKK+01] with 1024 connected nodes. In a DHT, a node that is close to a reconciliation object (*e.g.* n_0 is close to L_R) may be far distant of others (*e.g.* n_0 is far distant of *CS* and *S*). As a result, a node that is suitable for a P2P-reconciler step may not be worth in other steps. For this reason, every P2P-reconciler step has its own set of reconcilers.

Table 3 presents the transfer costs associated with the same nodes of Table 2. For simplicity, we assumed that all links between reconciler nodes and provider nodes have 1Mbps of bandwidth. The sizes of transferred data items are estimated based on the number of actions to be reconciled, the average action size, and the number of reconciler nodes. For space reasons, we do not detail this estimation.

Table 4 shows the estimated costs that the cost provider receives from the replica nodes. These costs are computed by applying on the P2P-reconciler cost model (Table 1) the lookup and direct costs of the DHT cost model (Table 2) and the transfer costs (Table 3). We show in bold the less expensive cost associated with each P2P-reconciler step. Thus, in our example, if the maximal number of reconcilers per step is 1, the cost provider selects as reconciler for each P2P-reconciler step the node of Table 4 whose cost is in bold (*i.e.* $Step_2 = \{n_0\}$, $Step_3 = \{n_0\}$, $Step_4 = \{n_1\}$, $Step_5 = \{n_2\}$, $Step_6 = \{n_3\}$), and notifies its decision to these nodes.

DHT costs per node	**Reconciliation objects**			
	L_R	***AS***	***CS***	***S***
$lc(n_0, id)$	0	685	1085	1036
$dc(n_0, home(id))$	43	162	222	218
$lc(n_1, id)$	832	0	1361	1069
$dc(n_1, home(id))$	163	282	193	185
$lc(n_2, id)$	974	1101	0	1483
$dc(n_2, home(id))$	146	28	351	351
$lc(n_3, id)$	1159	729	976	0
$dc(n_3, home(id))$	163	283	183	175

Table 2. Lookup and direct costs based on the DHT cost model. Each column holds a reconciliation object and each cell provides a specific lookup or direct cost (*e.g.* the cell in the 1st line and 2nd column indicates that n_0 spends 685ms to lookup *AS* whereas the cell in the 2nd line and 2nd column indicates that a direct access between n_0 and *home*(*AS*) costs 162ms.

Data item	**Description**	**Size (Mbits)**	**Cost (ms)**
actSet	Set of actions	1.202	1202
grpSet	Set of action groups	0.343	343
cluSet	Set of clusters	0.336	336
cluIds	Clusters' identifiers	0.120	120
sdcSet	Set of system-defined constraints	0.343	343
m_3Set	Set of memberships (produced at step 3)	0.801	801
m_4Set	Set of memberships (produced at step 4)	0.183	183
mSet	Set of all memberships	0.435	435
ovlCluSet	Set of overlapping clusters	0.336	336
itgCluSet	Set of integrated clusters	0.267	267
sumActSet	Set of summary actions	4.166	4166
ordActSet	Set of ordered actions	0.305	305

Table 3. Transfer costs with 1Mbps of bandwidth.

Nodes	**P2P-reconciler steps (i)**				
	2	**3**	**4**	**5**	**6**
n_0	1674	4449	4126	3249	8752
n_1	3698	5294	3305	3171	8496
n_2	3931	5187	3858	2307	8782
n_3	4352	5946	4351	3508	7733

Table 4. Node step costs

4.4 Managing the dynamic costs

The costs estimated by replica nodes for executing P2P-reconciler steps change as a result of disconnections and reconnections. To cope with this dynamic behavior and assure reliable cost estimations, a replica node n_i works as follows:

- **Initialization**: whenever n_i joins the system, n_i estimates its costs for executing every P2P-reconciler step. If these costs do not overtake the cost limit, n_i supplies the cost provider with this information.
- **Refreshment**: while n_i is connected, the join or leave of another node n_j may invalidate n_i's estimated costs due to routing changes. Thus, if the join or leave of n_j is relevant to n_i, n_i recomputes its P2P-reconciler estimated costs and refreshes them at the cost provider.

- **Termination**: when n_i leaves the system, if its P2P-reconciler estimated costs are smaller than the cost limit (*i.e.* the cost provider holds n_i's estimated costs), n_i notifies its departure to the cost provider.
- **Crash**: two actions follow the crash of a node n_i from the perspective of cost management. First, the cost provider discards the n_i estimated costs. This happens either as a result of an unsuccessful attempt of allocating n_i or due to the expiration of the validity of the n_i estimation (the cost provider associates a *time to live* with each cost estimation). Second, each node n_j that has n_i as neighbor and realizes the n_i absence recomputes n_j's estimated costs, if necessary. This computation is required if and only if n_j looks for some reconciliation object by routing lookup operations through n_i. In this case, n_j additionally propagates the refreshed costs to its neighbors.

P2P-reconciler computes the cost limit based on these parameters: the expected average latency of the network (*e.g.* 150 ms for the Internet), and the expected average number of hops to lookup a reconciliation object (*e.g.* $log(n)/2$ for a Chord DHT, where n represents the number of connected nodes and can be established as 15% of the community size).

Annex A presents the use of P2P-reconciler in the replication service of APPA (Atlas Peer-to-Peer Architecture). Annex B presents all proofs of correctness that P2P-reconciler assures eventual consistency among replicas, providing highly available reconciliation for dynamic networks, and work correctly in the presence of failures.

5 P2P-reconciler-TA protocol

P2P-reconciler-TA is a distributed protocol for reconciling conflicting updates in *topology-aware* P2P networks. Given a set of nodes, we exploit topological information to select the “best” nodes to participate in the different steps of an algorithm, in a way that achieves an optimal performance. A P2P network is classified as topology-aware if its topology is established by taking into account the physical distance among nodes (e.g. in terms of latency times).

Several topology-aware P2P networks could be used to validate our approach such as Pastry [RD01a], Tapestry [ZHSR+04, ZKJ01], CAN [RFHK+01], etc. We chose to construct our P2P-reconciler-TA over optimized CAN because it allows building the topology-aware overlay network in a relatively simple manner. In addition, its routing mechanism is easy to implement, although less efficient than other topology-aware P2P networks (e.g. the average routing path length in CAN is usually greater than in other structured P2P networks).

Basic CAN [RFHK+01] is a virtual Cartesian coordinate space to store and retrieve data as (*key, value*) pairs. At any point in time, the entire coordinate space is dynamically partitioned among all nodes in the system, so that each node owns a distinct zone that represents a segment of the entire space. To store (or retrieve) a pair (k_1, v_1), key k_1 is deterministically mapped onto a point P in the coordinate space using a uniform hash function, and then (k_1, v_1) is stored at the node that owns the zone to which P belongs. Intuitively, routing in CAN works by following the straight line path through the Cartesian space from source to destination coordinates.

Optimized CAN aims at constructing its logical space in a way that reflects the topology of the underlying network. It assumes the existence of well-known landmarks spread across the network. A node measures its round-trip time to the set of landmarks and orders them by increasing latency (i.e. network distance). The coordinate space is divided into bins such that each possible landmarks ordering is represented by a bin. Physically close nodes are likely to have the same ordering and hence will belong to the same bin.

We claim that the switch from Chord (P2P-reconciler) to CAN (P2P-reconciler-TA) does not invalidate our results based on the following argument. The most important DHT property in our experiments from the perspective of performance is the average number of hops needed to find the node that holds a given key. Let N be the number of connected nodes and d be the number of dimensions into which the d-

dimensional CAN space is divided. Chord requires *(log N)/2* hops on average to find a key whereas CAN needs $(d/4)(N^{1/d})$ hops on average. Since $d=2$ in our experiments, $(\log N)/2 < (N^{1/2})/2$. That means, due to the need of a larger number of hops to find a key over CAN, the expected performance over CAN is worse than Chord. This fact makes our results still better.

Briefly, P2P-reconciler-TA works as follows. Based on the network topology, it selects the best provider and reconciler nodes. These nodes then reconcile conflicting updates and produce a schedule, which is an ordered list of non-conflicting updates. In this work, we focus on node allocation by proposing a dynamic distributed algorithm for efficiently selecting provider and reconciler nodes. We first introduce some definitions, and then we present the allocation algorithm in detail.

5.1 Definitions

P2P-reconciler-TA uses the following reconciliation objects: action log (L_R), action summary (*AS*), clusters set (*CS*), and schedule (*S*). The action log contains update actions to be reconciled; the action summary holds constraints among actions; the clusters set stores clusters of conflicting actions; and the schedule holds an ordered list of actions that do not violate constraints. For availability reasons, we produce *k* replicas of each reconciliation object and store these replicas into different providers. We note these terms as follows:

- **RO**: set of reconciliation objects { L_R, *AS*, *CS*, *S*}.
- **ro**: a reconciliation object belonging to *RO* (e.g. *CS*, L_R, etc.).
- $\mathbf{ro_i}$: the replica *i* of the reconciliation object *ro* (e.g. CS_1 is the replica *1* of *CS*), where $1 \le i \le k$; the coordinates (x_i, y_i) are associated with ro_i and determines the ro_i placement over the CAN coordinate space; ro_i is stored at the provider node p_{roi} whose zone includes (x_i, y_i).
- $\mathbf{P_{ro}}$: set of *k* providers p_{roi} that store replicas of the reconciliation object *ro*.
- **best($\mathbf{P_{ro}}$)**: the most efficient provider node holding a replica of *ro*.

We apply various criteria to select the best provider nodes. One of such criteria establishes that a provider node should not be isolated in the network, i.e. it should be close to a certain number of neighbors that can become reconcilers, and therefore are called *potential reconcilers*. The physical proximity in terms of latency is not enough; a potential reconciler should also be able to access provider's data by an acceptable cost. Thus, such a potential reconciler is considered a *good neighbor* of the associated provider node. We now present metrics and terms applied in provider node selection:

- **accessCost(n, p)**: the cost for a node *n* accessing data stored at the provider node *p* in terms of latency and transfer times. The transfer time relies on the message size, which is usually variable. For simplicity, we consider a message of fixed size (e.g. 4 Kb). Equation 1 shows that the *accessCost(n, p)* is computed as the latency between *n* and *p* (noted *latency(n, p)*) plus the time to transfer the message *msg* from *p* to *n* (noted *tc(p, n, msg)*).

$$accessCost(n, p) = latency(n, p) + tc(p, n, msg) \qquad (1)$$

- **maxAccessCost**: the maximal acceptable cost for any node accessing data stored in provider nodes; if *accessCost(n, p) > maxAccessCost*, *n* is considered far away from *p*, and therefore it is not a good neighbor of *p*.
- **potRec(p)**: number of potential reconcilers that are good neighbors of *p*.
- **minPotRec**: minimal number of potential reconcilers required around a provider node *p* in order to accept *p* as a candidate provider; if *potRec(p) < minPotRec*, *p* is considered isolated in the network.

- **candidate provider**: any provider node *p* with *potRec(p)* ≥ *minPotRec* is considered a candidate in the provider selection.
- **cost provider**: node that stores costs used in the node selection.
- **QoN(p)**: quality of network around the provider node *p*. It is defined as the average access cost associate with good neighbors of *p*, and it is computed by equation 2. In this equation, n_i represents a good neighbor of *p*.

$$QoN(p) = \frac{1}{pot\mathrm{Re}c(p)} \sum_{i=1}^{pot\mathrm{Re}c(p)} accessCost(n_i, p) \qquad (2)$$

Another criterion for selecting a provider node is its proximity to other providers. During a reconciliation step, a reconciler node often needs to access various reconciliation objects. By approximating provider nodes we reduce the associated access costs. We now present some terminology for reconciler selection:

- **candidate reconcilers**: set of nodes that are candidate to become reconcilers. This set includes all good neighbors of selected providers.
- **step**: a reconciliation stage.

5.2 Detailed algorithm

P2P-reconciler-TA selects provider nodes and candidate reconcilers as follows. Every provider node regularly evaluates its network quality and, according to the number of potential reconcilers around it, the provider announces or cancels its candidature to the cost provider node. The cost provider, in turn, manages candidatures by monitoring which providers have the best network quality. Whenever the best providers change, the cost provider performs a new selection and notifies its decision to provider nodes. Following this notification, provider nodes inform their good neighbors whether they are candidate reconcilers or not. With the selection of new providers, current estimated reconciliation costs are discarded and new estimations are produced by the new candidate reconcilers. Thus, selected provider nodes and candidate reconcilers are dynamically changing according to the evolution of the network topology. We now detail each step of node allocation.

5.2.1 Computing provider node's QoN

A provider node computes its network quality by using equation 2 and the input data supplied by its good neighbors. Good neighbors introduce themselves to the provider nodes as follows. Consider that node *n* has just joined the network. For each reconciliation object *ro* ∈ *RO*, *n* looks for the closest node that can provide *ro*, noted p_{ro}, and if *accessCost(n,* p_{ro}*)* is acceptable, *n* introduces itself to p_{ro} as a good neighbor by informing *accessCost(n,* p_{ro}*)*. Node *n* finds the closest p_{ro} as follows. First, *n* uses *k* hash functions to obtain the *k* coordinates (x_i, y_i) corresponding to each replica ro_i. Then, *n* computes the Cartesian distance between *n*'s coordinates and each (x_i, y_i). Finally, the closest p_{ro} is the one whose zone includes the closest (x_i, y_i) coordinates. The closest p_{ro} is called the *n*'s reference provider wrt. *ro*. Figure 5 illustrates how node *n* finds its reference provider wrt. the action summary reconciliation object (*AS*).

Provider nodes and the associated potential reconcilers cope with the dynamic behavior of the P2P network as follows. A provider node dynamically refreshes its *QoN* based on its good neighbors' joins, leaves, and failures. Joins and leaves are notified by the good neighbors whereas failures are detected by the provider node based on the expiration of a *ttl* (*time-to-live*) field. On the other hand, a good neighbor dynamically changes a reference provider p_{ro} whenever p_{ro} gives up the responsibility for *ro*. If p_{ro} disconnects or transfers *ro* to another provider, p_{ro} notifies these events to its good neighbors. However, if p_{ro} fails its good neighbors detect such failure and change the corresponding reference provider.

5.2.2 Managing provider candidature

The network quality associated with a provider node dynamically changes as its potential reconcilers join, leave, or fail. Thus, a provider node often refreshes its candidature as follows. When the neighborhood situation of a provider *p* switches from *isolated* (i.e. *p* has a few of potential reconcilers around it) to *surrounded* (i.e. *potRec(p)* ≥ *minPotRec*) *p* announces its candidature to the cost provider. In contrast, when *p* switches from surrounded to isolated, *p* cancels its candidature. Finally, if *p*'s *QoN* varies while it remains surrounded of potential reconcilers, *p* updates its *QoN*. Figure 6 illustrates *AS* candidate providers for *minPotRec* = 4. Since we replicate reconciliation objects, we assume that at least one provider node is available for each reconciliation object. In [Mar07], we prove that our solution assures high availability of replicated objects in the DHT.

5.2.3 Selecting provider nodes

For each reconciliation object, P2P-reconciler-TA must select the best provider node. This selection should take into account the proximity among providers since different providers are accessed in the same reconciliation step. We reduce the search space of best providers by applying the heuristic illustrated in Figure 7. First, we select the *best*(P_{AS}) and the *best*(P_{CS}) (Figure 7a). These nodes must be as close as possible from each other because *AS* and *CS* are the most accessed reconciliation objects and both are often retrieved in the same step. Next, we select the *best*(P_{LR}) and the *best*(P_S) based on the pair (*best*(P_{AS}), *best*(P_{CS})) previously selected (Figure 7b); *best*(P_{LR}) must be as close as possible to the *best*(P_{AS}) since a reconciler accesses both *best*(P_{LR}) and *best*(P_{AS}) in the same step whereas *best*(P_S) must be as close as possible to the *best*(P_{CS}) for the same reason. Figure 7c shows the selected providers of our illustrative scenario (i.e. p_{AS1}, p_{CS3}, p_{S1}, and p_{LR5}).

The candidate providers filtered to participate of the provider selection vary with time. To face this dynamic behavior of candidatures, the cost provider automatically launches a new provider selection whenever the set of filtered candidates changes.

5.2.4 Notifying provider selection

Changing the selected provider leads to changes in the set of candidate reconcilers and invalidates all estimated reconciliation costs. As a result, the cost provider discards estimated costs and notifies the result of provider selection to provider nodes. The provider nodes, in turn, proceed as follows. If the provider *p* switches from selected to unselected, *p* notifies its good neighbors that from now on they are no longer candidate reconcilers. In contrast, if the provider *p* switches from unselected to selected, *p* notifies its good neighbors that from now on they are candidate reconcilers.

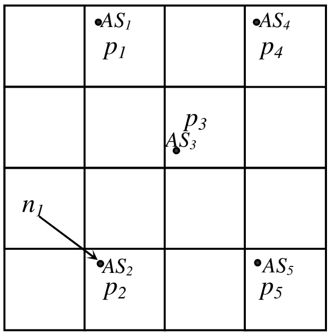


Figure 5. Finding the *AS* reference provider

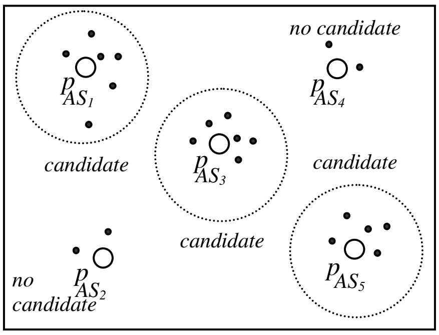
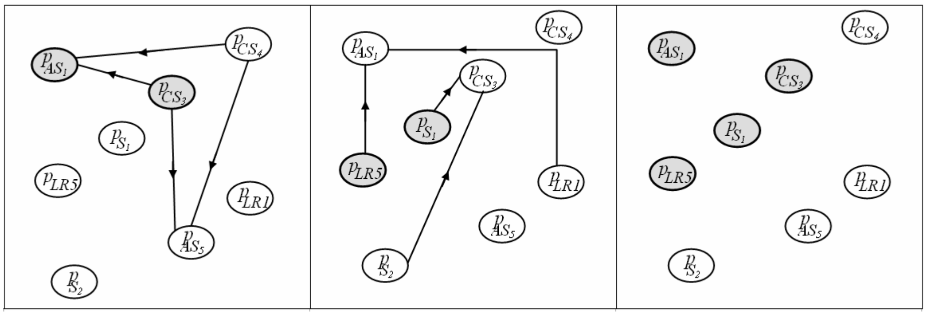


Figure 6. Managing provider candidature



(a) Selecting (*AS*, *CS*) pair (b) Selecting L_R and *S* providers (c) Selected providers

Figure 7. Selecting provider nodes

6 Validation

We validated and evaluated the performance of our reconciliation solutions through experimentation and simulation. The experimentation over Grid5000 (see Annex A) was useful to validate the algorithms and calibrate our simulator. The simulator allowed us to scale up to higher numbers of nodes. In this section, we first introduce our performance model, and then we report the main performance evaluation results. In [Sim07], we describe in detail how we simulate large P2P networks.

6.1 Performance model

We evaluated the performance of P2P-reconciler, and P2P-reconciler-TA. Our performance model takes into account the strategy for selecting provider and reconciler nodes, the action log size (i.e. the number of actions to be reconciled), and the network topology. Some parameters are applicable to all evaluated algorithms whereas other parameters are protocol-specific. Table 5 summarizes such parameters arranging them in three groups: general parameters, parameters that are specific for the P2P-reconciler protocol, and parameters specific for P2P-reconciler-TA.

In all experiments, we need to determine the number of actions to be reconciled, noted *Nb-Actions*. The network topology must also be set before any experiment. The network topology is defined by the number of connected nodes, noted *Nb-Nodes*, the bandwidth of the links among these nodes, noted *Bandwidth*, the average link latency, noted *Avg-Latency*, and the associated standard deviation, noted *Sd-Latency*. Indeed, we provide the minimal and maximal latencies corresponding to the type of network we intend to simulate (e.g. cluster, Grid, Internet, etc.), and after the node placement we compute the resulting average latency and the associate standard deviation. For topologies with variable bandwidths, the bandwidth values follow a Pareto distribution (low bandwidths are more frequently assigned than high bandwidths). We chose the Pareto distribution because it models more realistically the distribution of bandwidths in which the number of sites with higher bandwidths decreases exponentially. We produced 3 different networks for each set of parameter values. We also produced 3 action logs for each action log size. By combining different action logs with different networks for the same set of parameter values, we generate several distinct reconciliation scenarios that avoid over fitted results.

The P2P-reconciler protocol has only one specific parameter, namely the strategy for selecting reconciler nodes; this parameter is called *Allocation*. We define three allocation strategies: random selection (RDM); cost-based selection using *precise* costs for direct communication (CB/P); and cost-based selection using *estimated* costs for direct communication (CB/E). Recall from Section 3 that the precise approach may overload provider nodes and the network as a whole whereas the estimated approach, although not precise, avoids overloads. For every allocation strategy, all experiments use the optimal number of reconcilers.

The P2P-reconciler-TA protocol has specific parameters for node allocation and network simulation. Concerning node *allocation*, three strategies are possible: random selection of provider and reconciler nodes (RDM), cost-based selection of reconciler nodes only (REC), and cost-based selection of both provider and reconciler nodes (PRV-REC). Recall from Section 5 that we replicate each reconciliation object to assure high availability. Hence, each reconciliation object has various candidate provider nodes. In the latter allocation strategy (i.e. PRV-REC), the parameter *Nb-Providers* specifies how many candidate providers should be considered for each reconciliation object in order to select an efficient set of provider nodes. We adopt such a heuristic approach to reduce the search space, thereby avoiding an exhaustive search.

We based our performance evaluation on the IceCube's benchmark [PSM02, PSM03] and we set up application parameters as IceCube. In short, the benchmark is based on a calendar application which consists of an appointment database shared by multiple users. User commands may *request* a meeting, possibly proposing several alternative times, and *cancel* a previous request. Database-level actions *add* or *remove* a single appointment. The user-level *request* command is mapped onto an alternative constraint containing a set of *add* actions; similarly for *cancel*. Each such action contains the time, duration, participants and location of the proposed appointment. The calendar inputs are based on traces from actual Outlook calendar. This was artificially scaled up in size, and were modified to contain conflicts and alternatives and to control the difficulty of reconciliation.

We have shown in [MAPV06] that if we run our P2P reconciliation approach over a high-speed network we improve the reconciliation performance with respect to the centralized counterpart (i.e. IceCube). However, the focus of our work is to provide high data availability in a P2P scenario where individual sites are not very reliable and might be disconnected for variable periods of time, with good scalability, *acceptable* performance, and limited overhead. We have not aimed to achieve optimal performance in any scenario since this might be unfeasible. Since a P2P network usually has a large number of connected nodes which can leave at any time, in the P2P scenario, availability and scalability are the most important properties as far as a reasonable performance is provided.

	Parameter	**Definition**	**Values**
General	*Nb-Actions*	Number of actions to be reconciled	106 – 10000
	Nb-Nodes	Number of connected nodes	1024 – 32768
	Bandwidth	Network bandwidth	Kbps: 64, 128, 256, 512 Mbps: 1, 2, 8, 10, 20
	Avg-Latency	Average latency (in ms)	51 – 263
	Sd-Latency	Standard deviation of latencies (in ms)	15 – 96
P2P-reconciler	*Allocation*	Strategy for selecting reconciler nodes	CB/P; CB/E; RDM
P2P-reconciler-TA	*Allocation*	Strategy for selecting providers and reconcilers	RDM; REC; PRV-REC
	Nb-Providers	Number of candidate providers per rec. object	3 – 8

Table 5. Evaluation parameters

6.2 Experimental results

We now present our main experimental results. We first show the performance of the P2P-reconciler protocol. Then, we present the evaluation of the P2P-reconciler-TA protocol.

The first experiment aims to evaluate the behavior of the cost-based approach as the number of actions increases. In this evaluation, we configured the network with variable latencies, constant bandwidth (1 Mbps), and 1024 connect nodes. The number of actions varies from 106 to 10,000. Figure 8 shows that the reconciliation time using cost-based selection of reconciler nodes (CB/P-1-1024) remains advantageous wrt. the random approach (RDM-1-1024) as the number of actions increases.

The second experiment studies the reconciliation performance with variable bandwidths. Values between 64Kbps and 20 Mbps were assigned to connected nodes according to the Pareto distribution (low bandwidths are more frequently assigned than high bandwidths). We also varied the number of actions to be reconciled in order to observe the scalability of P2P-reconciler. Figure 9 shows that the inclusion of transfer costs in the P2P-reconciler cost model is advantageous in scenarios with variable bandwidths, as is the case of the Internet. The performance improvement provided by the cost-based approaches (CB/P 1024 and CB/E 1024) wrt. the random approach (RDM 1024) achieved a factor of 26 in Figure 9; recall that in Figure 8 we show the same performance improvement varying only latencies, and the corresponding factor is 1.6. The scalability also improved since in Figure 9 the reconciliation times using cost-based approaches (CB/P 1024 and CB/E 1024) are represented by straight lines. In addition, the performance of the precise and the estimated cost-based approaches are quite similar (although the corresponding lines overlap in the scale of Figure 9, there is a difference of about 10%).

Finally, we deepen the investigation of P2P-reconciler scalability by means of two experiments. In the first one, we studied the impact of the number of connected nodes on the reconciliation time (the larger the number of nodes is, the larger the average number of hops to lookup an identifier in the DHT). The network had variable latencies and bandwidths; 10,000 actions were reconciled. We varied the number of connected nodes from 1024 to 32768. Recall from the motivating application (i.e. the P2P Wiki) that, although the number of users updating a single data object in parallel is usually small, the size of the collaborative network to which this object belongs may be large. Figure 10 represents the reconciliation time with a straight line, which means an excellent scalability wrt. the number of connected nodes. In the second experiment, we studied the impact of the action size on the reconciliation time, by varying it from 10 bytes to 1024 bytes. Figure 11 shows that this result is also quite good since an increase of two orders of magnitude on the action size produced a corresponding increase of about 2.6 times on the reconciliation time (from 20s to 52s).

Liveness is an important issue in dynamic systems. P2P-reconciler provides a greater degree of availability, scalability and fault-tolerance than the centralized solution. In addition, the performance of P2P-reconciler is good since it takes 20s to reconcile 10,000 actions in a network with variable latencies and bandwidths (recall that P2P-reconciler depends on network communication). The centralized solution is unsuitable for P2P networks due to its low availability in dynamic environments.

We now present performance results concerning the P2P-reconciler-TA protocol. The first experiment aims to observe the scalability of P2P-reconciler-TA by studying the impact of the number of connected nodes on the reconciliation time (the larger the number of nodes is, the larger the average number of hops needed to lookup an identifier in the DHT). We configured the network with variable bandwidths and we varied the number of connected nodes (*Nb-Nodes*) from 1024 to 4096. The number of reconciled actions (*Nb-Actions*) was 1005. **Erro! A origem da referência não foi encontrada.** represents the reconciliation time with a straight line, which means an excellent scalability wrt. the number of connected nodes.

Recall from Section 5 that reconciliation objects are replicated and stored in the DHT according to multiple hash functions in order to assure high availability. As a result, for each reconciliation object, P2P-reconciler-TA must select the best provider node. Despite the limited number of replicas (typically around 10) the search space is quite large since the combination of provider nodes must be taken into account. We aim at drastically reducing the search space of best providers while preserving the best alternatives in the reduced search space. This allows us to efficiently select provider nodes. So, our second experiment studies the selection of provider nodes by varying the number of candidate providers per reconciliation object. The candidates are chosen according to their network quality (the network quality around the provider node *p* is defined as the average access cost, in terms of latency and transfer times, associated with *p*'s neighbors that are closest to *p*). **Erro! A origem da referência não foi encontrada.** shows that our heuristic achieves the best performance with small numbers of candidates (*Nb-Providers* = 3 or 4). This is an excellent result since the smaller the number of candidates is, the smaller the search space (e.g. *Nb-Providers* = 3 with 4 involved reconciliation objects results a search space of size $3^4 = 81$).

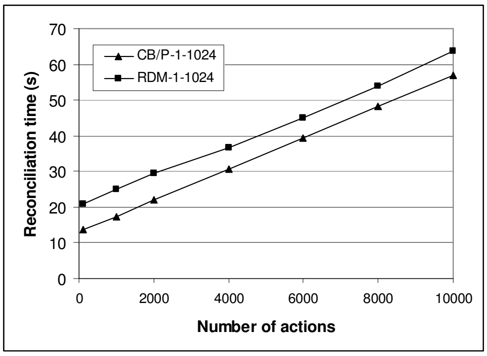


Figure 8. Reconciliation time varying the number of actions

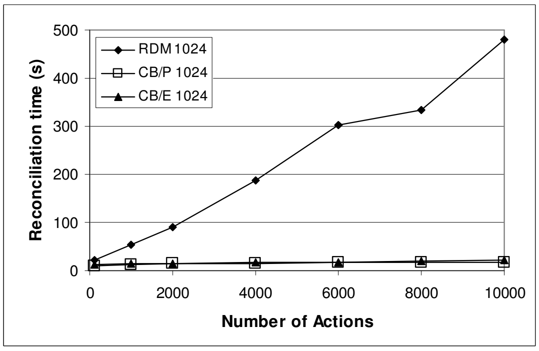


Figure 9. Reconciliation time varying actions and bandwidths

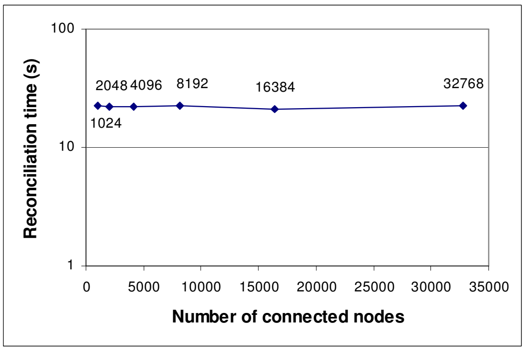


Figure 10. Reconciliation time varying the number of nodes

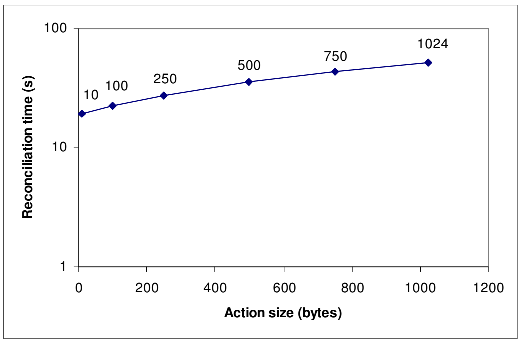


Figure 11. Reconciliation time varying action size

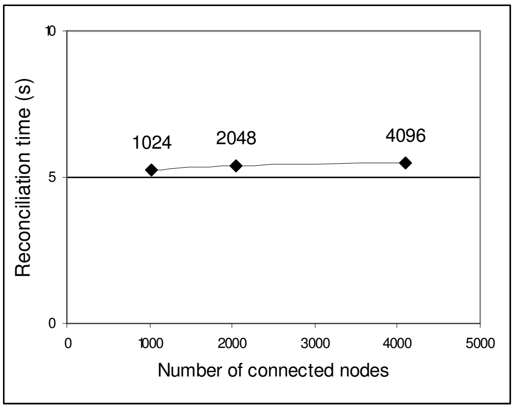


Figure 12. Reconciliation time varying the *Nb-Nodes*

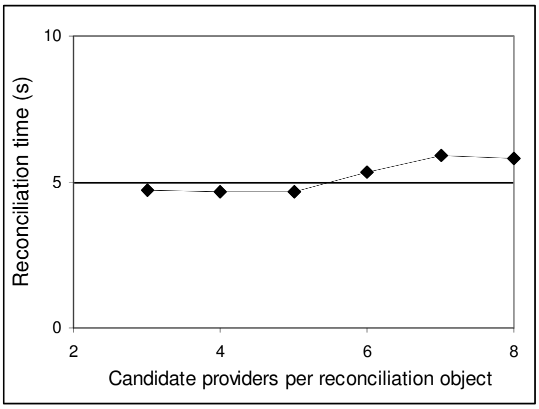


Figure 13. Reconciliation time varying *Nb-Providers*

The main motivation for proposing P2P-reconciler-TA is to improve the performance of P2P-reconciler by taking advantage of topology-aware networks. Thus, our last experiment compares the performance of P2P-reconciler and P2P-reconciler-TA while running both protocols in the same context (i.e. number of actions to reconcile, number of connected nodes, network bandwidths and latencies, etc.). Figure 14 shows that P2P-reconciler-TA over CAN outperforms P2P-reconciler by a factor of 2 (i.e. a performance improvement of 50%). This is an excellent result if we consider that P2P-reconciler is already an efficient protocol and CAN is not the most efficient topology-aware P2P network (e.g. Pastry and Tapestry are more efficient than CAN).

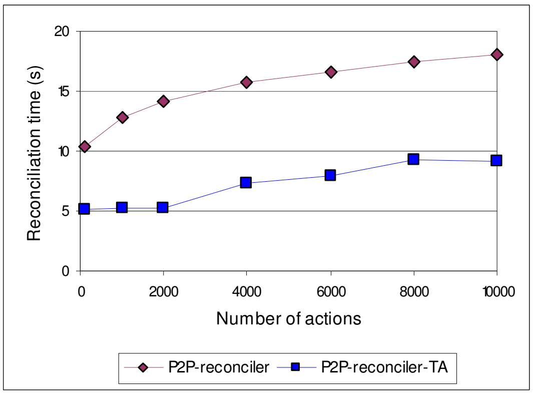


Figure 14. Reconciliation time varying the number of actions

7 Related work

In the context of P2P networks, there has been little work on managing data replication in the presence of updates. Most of data sharing P2P networks consider the data they provide to be very static or even read-only. Freenet [CMHS+02] partially addresses updates which are propagated from the updating peer downward to close peers that are connected. However, peers that are disconnected do not get updated. P-Grid [ACDD+03, AHA03] is a structured P2P network that exploits epidemic algorithms to address updates. It assumes that conflicts are rare and their resolution is not necessary in general. In addition, P-Grid assumes that probabilistic guarantees instead of strict consistency are sufficient. Moreover, it only considers updates at the file level. In OceanStore [KBCC+00] every update creates a new version of the data object. Consistency is achieved by a two-tiered architecture: a client sends an update to the object's primary copies and some secondary replicas in parallel. Once the update is committed, the primary copies multicast the result of the update down the dissemination tree. OceanStore assumes an infrastructure comprised of servers that are connected by high-speed links. Different from the previous works, we propose to distribute the reconciliation engine in order to provide high availability.

Table 6 compares the replication solutions provided by different types of P2P systems. Clearly, none of them provide eventual consistency among replicas along with weak network assumptions, which is the main concern of this work.

The distributed log-based reconciliation algorithms proposed by Chong and Hamadi [CH06] addresses most of our requirements, but this solution is unsuitable for P2P systems as it does not take into account the dynamic behavior of peers and network limitations. Operational transformation [VCFS00] also addresses eventual consistency among replicas, but this approach is specific for collaborative edition and it assumes synchronous collaboration (i.e. concurrent updates of replicas). Our approach assures eventual consistency among replicas, which enables asynchronous collaboration among users. In addition, we provide multi-master replication and we do not assume servers linked by high-speed links.

In the context of APPA (Atlas Peer-to-Peer Architecture), a P2P data management system which we are building [AMPV06b, MAPV06], we proposed the *DSR-cluster* algorithm [MPJV06, MPV05], a distributed version of the semantic reconciliation engine of IceCube [KRSD01, PSM03] for cluster networks. However, *DSR-cluster* does not take into account network costs during reconciliation. A fundamental assumption behind *DSR-cluster* is that the communication costs among cluster nodes are negligible. This assumption is not appropriate for P2P systems, which are usually built on top of the Internet. In this case, network costs may vary significantly from node to node and have a strong impact on the performance of reconciliation.

P2P System	P2P Network	Data Type	Autonomy	Replication Type	Conflict Detection	Consistency	Network Assump.
Napster	Super-peer	File	Moderate	Static data	–	–	Weak
JXTA	Super-peer	Any	High	–	–	–	Weak
Gnutella	Unstructured	File	High	Static data	–	–	Weak
Chord	Structured (DHT)	Any	Low	Single-master Multi-master	Concurrency None	Probabilistic Probabilistic	Weak
CAN	Structured (DHT)	Any	Low	Static data Multi-master	– None	– Probabilistic	Weak
Tapestry	Structured (DHT)	Any	High	–	–	–	Weak
Pastry	Structured (DHT)	Any	Low	–	–	–	Weak
Freenet	Structured	File	Moderate	Single-master	None	No guarantees	Weak
PIER	Structured (DHT)	Tuple	Low	–	–	–	Weak
OceanStore	Structured (DHT)	Any	High	Multi-master	Concurrency	Eventual	Strong
PAST	Structured (DHT)	File	Low	Static data	–	–	Weak
P-Grid	Structured	File	High	Multi-master	None	Probabilistic	Weak

Table 6. Comparing replication solutions in P2P systems

8 Conclusion

In this paper, we proposed the P2P-reconciler, a distributed protocol for semantic reconciliation in P2P networks. Our main contributions are a cost model for computing communication costs in DHTs and an algorithm that takes into account these costs and the P2P-reconciler steps to select the best reconciler nodes. For computing communication costs, we use local information and we deal with the dynamic behavior of nodes. In addition, we limit the scope of event propagation (*e.g.* joins or leaves) in order to avoid network overload.

Furthermore, we proposed a topology-aware approach to improve response times in P2P distributed semantic reconciliation. The P2P-reconciler-TA algorithm dynamically takes into account the physical network topology combined with the DHT properties when executing reconciliation. We proposed topology aware metrics and cost functions to be used for dynamically selecting the best nodes to execute reconciliation, while considering dynamic data placement.

We validated P2P-reconciler through implementation and simulation. The experimental results showed that our cost-based reconciliation outperforms the random approach by a factor of 26. In addition, the number of connected nodes is not important to determine the reconciliation performance due to the DHT scalability and the fact that reconcilers are as close as possible to the reconciliation objects. The action size impacts the reconciliation time in a logarithmic scale. Compared with the centralized solution, which is more efficient but low available, our algorithm yields high data availability and excellent scalability, with acceptable performance and limited overhead.

In the same way, we also validated P2P-reconciler-TA. The experimental results show that our topology-aware approach achieves a performance improvement of 50 % in comparison with the P2P-reconciler. In addition, P2P-reconciler-TA has proved to be scalable with limited overhead and thereby suitable for P2P environments. Our topology-aware approach is conceived for distributed reconciliation; however our metrics, costs functions as well as our selection approach are useful in several contexts.

References

[ACDD+03] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, and R. Schmidt. P-Grid: a self-organizing structured P2P system. *ACM SIGMOD Record*, 32(3):29-33, September 2003.

[AHA03] D. Anwitaman, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, page 76-85, Washington, District of Columbia, May 2003.

[AMPV06a] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. *Global Data Management* (Chapter Design and implementation of Atlas P2P architecture). 1st Ed., IOS Press, July 2006.

[AMPV06b] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Top-k query processing in the APPA P2P system. In *Proc. of the Int. Conf. on High Performance Computing for Computational Science (VecPar)*, Rio de Janeiro, Brazil, July 2006.

[APV06] R. Akbarinia, E. Pacitti, and P. Valduriez. Reducing network traffic in unstructured P2P systems using top-k queries. *Distributed and Parallel Databases,* 19(2-3):67-86, May 2006.

[APV07] R. Akbarinia, E. Pacitti, and P. Valduriez. Data currency in replicated DHTs. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 211-222, Beijing, China, June 2007.

[Bri08] Brite. http://www.cs.bu.edu/brite/

[CH06] Y.L. Chong and Y. Hamadi. Distributed log-based reconciliation. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, pages 108-112, Riva del Garda, Italy, September 2006.

[CJKR+03] M. Castro, M.B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang, and A. Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Proc. of the Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1510-1520, San Francisco, California, April 2003.

[CKV01] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427-469, December 2001.

[CMHS+02] I. Clarke, S. Miller, T.W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40-49, January 2002.

[EMP07] M. El Dick, V. Martins, and E. Pacitti. A topology-aware approach for distributed data reconciliation in P2P networks. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, Rennes, France, August 2007.

[Fip95] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/NIST, National Technical Information Service, Springfield, Virginia, April 1995.

[Gnu06] Gnutella. http://www.gnutelliums.com/.

[Gri06] Grid5000 Project. http://www.grid5000.fr.

[HHLT+03] R. Huebsch, J. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of Int. Conf. on Very Large Databases (VLDB)*, pages 321-332, Berlin, Germany, September 2003.

[HM98] F. Howell and R. McNab. SimJava: a discrete event simulation package for Java with applications in computer systems modeling. In *Proc. of the Int. Conf. on Web-based Modeling and Simulation*, San Diego, California, January 1998.

[JAB01] M. Jovanovic, F. Annexstein, and K. Berman. Scalability issues in large peer-to-peer networks: a case study of Gnutella. Technical report, ECECS Department, University of Cincinnati, Cincinnati, Ohio, January 2001.

[Jov00] M. Jovanovic. Modelling large-scale peer-to-peer networks and a case study of Gnutella. Master's thesis, Department of Electrical and Computer Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio, June 2000.

[Jxt06] JXTA. http://www.jxta.org/.

[KBCC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proc. of the ACM Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190-201, Cambridge, Massachusetts, November 2000.

[KKMN98] D.G. Kleinbaum, L.L. Kupper, K.E. Muller, and A. Nizam. *Applied Regression Analysis and Multivariable Methods*. 3rd Ed., Duxbury Press, January 1998.

[KLLL+97] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the ACM Symp. on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[KRSD01] A-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)*, pages 210-218, Newport, Rhode Island, August 2001.

[KWR05] P. Knezevic, A. Wombacher, and T. Risse. Enabling high data availability in a DHT. In *Proc. of the Int. Workshop on Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE'05)*, pages 363-367, Copenhagen, Denmark, August 2005.

[Man07] Mandriva. http://club.mandriva.com/xwiki/

[Mar07] V. Martins. *Data replication in P2P systems*. PhD thesis, University of Nantes, Nantes, France, May 2007. http://www.sciences.univ-nantes.fr/lina/gdd/members/vmartins/.

[MAPV06] V. Martins, R. Akbarinia, E. Pacitti, and P. Valduriez. Reconciliation in the APPA P2P system. In *Proc. of the IEEE Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 401-410, Minneapolis, Minnesota, July 2006.

[Met06] Meteor. http://meteor.jxta.org/

[MP06] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in P2P-DHT networks. In *Proc. of the European Conf. on Parallel Computing (Euro-Par)*, pages 337-349, Dresden, Germany, September 2006.

[MPJV06] V. Martins, E. Pacitti, R. Jimenez-Peris, and P. Valduriez. Scalable and available reconciliation in P2P networks. In *Proc. of the Journées Bases de Données Avancées (BDA)*, Lille, France, October 2006.

[MPV05] V. Martins, E. Pacitti, and P. Valduriez. Distributed semantic reconciliation of replicated data. *IEEE France and ACM SIGOPS France - Journées Francophones sur la Cohérence des Données en Univers Réparti (CDUR)*, Paris, France, November 2005.

[MPV06a] V. Martins, E. Pacitti, and P. Valduriez. A dynamic distributed algorithm for semantic reconciliation. In *Proc. of the Int. Workshop on Distributed Data & Structures (WDAS)*, Santa Clara, California, January 2006.

[Nap06] Napster. http://www.napster.com/

[PRR97] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 311-320, Newport, Road Island, June 1997.

[PSM02] N. Preguiça, M. Shapiro, and C. Matheson. Efficient semantic-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, Cambridge (UK), May 2002.

[PSM03] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. of the Int. Conf. on Cooperative Information Systems (CoopIS)*, pages 38-55, Catania, Italy, November 2003.

[RD01a] A. Rowstron and P. Druschel. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM Int. Conf. on Distributed Systems Platforms (Middleware)*, pages 329-350, Heidelberg, Germany, November 2001.

[RD01b] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)*, pages 188-201, Banff, Canada, October 2001.

[RFHK+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages: 161-172, San Diego, California, August 2001.

[SBK04] M. Shapiro, K. Bhargavan, N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. of the Int. Conf. on Principles of Distributed Systems (OPODIS)*, Grenoble, France, December 2004.

[Sim07] P2P network simulation. http://www.sciences.univ-nantes.fr/lina/gdd/members/vmartins/.

[SMKK+01] I. Stoica, R. Morris, D.R. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149-160, San Diego, California, August 2001.

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

[VCFS00] N. Vidot, M. Cart, J. Ferrie, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the ACM Int. Conf. on Computer Supported Cooperative Work (CSCW)*, pages 171-180, Philadelphia, Pennsylvania, December 2000.

[Wik07] Wikipedia. http://wikipedia.org/.

[WIO97] S. Whittaker, E. Issacs, and V. O'Day. Widening the net: workshop report on the theory and practice of physical and network communities. *ACM SIGCHI Bulletin*, 29(3):27-30, July 1997.

[ZHSR⁺04] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications (JSAC),* 22(1):41-53, January 2004.

[ZKJ01] B.Y. Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-010-1141, University of California, Berkeley, California, April 2001.

ANNEX A: REPLICATION SUPPORT IN APPA

We propose a solution for data replication in P2P networks that assures eventual consistency among replicas. Such solution is built in the context of APPA (Atlas Peer-to-Peer Architecture). APPA is a data management system that provides scalability, availability and performance for P2P advanced applications, which must deal with semantically rich data (e.g. XML documents, relational tables, etc.) using a high-level SQL-like query language. The replication service is placed in the upper layer of APPA architecture; the APPA architecture provides an application programming interface (API) to make it easy for P2P collaborative applications to take advantage of data replication. The architecture design also establishes the integration of the replication service with other APPA services by means of service interfaces. This section introduces the APPA architecture, and then describes the proposed APPA replication service.

APPA

APPA has a layered service-based architecture. Besides the traditional advantages of using services (encapsulation, reuse, portability, etc.), this enables APPA to be network-independent so it can be implemented over different structured (e.g. DHT) and super-peer P2P networks. The main reason for this choice is to be able to exploit rapid and continuing progress in P2P networks. Another reason is that it is unlikely that a single P2P network design will be able to address the specific requirements of many different applications. Obviously, different implementations will yield different trade-offs between performance, fault-tolerance, scalability, quality of service, etc. For instance, fault-tolerance can be higher in DHTs because no node is a single point of failure. On the other hand, through index servers, super-peer networks enable more efficient query processing. Furthermore, different P2P networks could be combined in order to exploit their relative advantages, e.g. DHT for key-based search and super-peer for more complex searching. Figure 15 shows the APPA architecture, which is composed of three layers of services: P2P network services, basic services and advanced services.

P2P network services. This layer provides network independence with services that are common to different P2P networks:

- **Peer id assignment:** assigns a unique id to a peer using a specific method, e.g. a combination of super-peer id and counter in a super-peer network.
- **Peer linking:** links a peer to some other peers, e.g. by locating a zone in CAN.
- **Key-based storage and retrieval (KSR):** stores and retrieves a (*key*, *object*) pair in the P2P network, e.g. through hashing over all peers in DHT networks or using super-peers in super-peer networks. An important aspect of KSR is that it allows managing data using object semantic. Object semantic means that an object stored in the P2P network consists of a set of data attributes which can be accessed individually for read or write purposes. This approach is appropriate for optimizing object access performance since we do not need to transfer the entire object through the network at each object access operation as the existing P2P networks use to do.
- **Key-based time stamping (KTS):** generates monotonically increasing timestamps which are used for ordering the events occurred in the P2P system.
- **Peer communication:** enables peers to exchange messages (i.e. service calls).

Basic services. This layer provides elementary services for the advanced services using the P2P network layer:

- **Persistent data management (PDM):** provides high availability for the *(key, object)* pairs which are stored in the P2P network.
- **Communication cost management:** estimates the communication costs for accessing a set of objects that are stored in the P2P network. These costs are computed based on latencies and transfer rates, and they are refreshed according to the dynamic connections and disconnections of nodes.
- **Group management:** allows peers to join an abstract *group*, become *members* of the group and send and receive membership notifications. This is similar to group communication systems [CKV01, CJKR+03].

Advanced services. This layer provides advanced services for semantically rich data sharing including schema management, replication [APV07, EMP07, MAPV06, MP06, MPJV06], query processing [AMPV06b, APV06], security, etc. using the basic services.

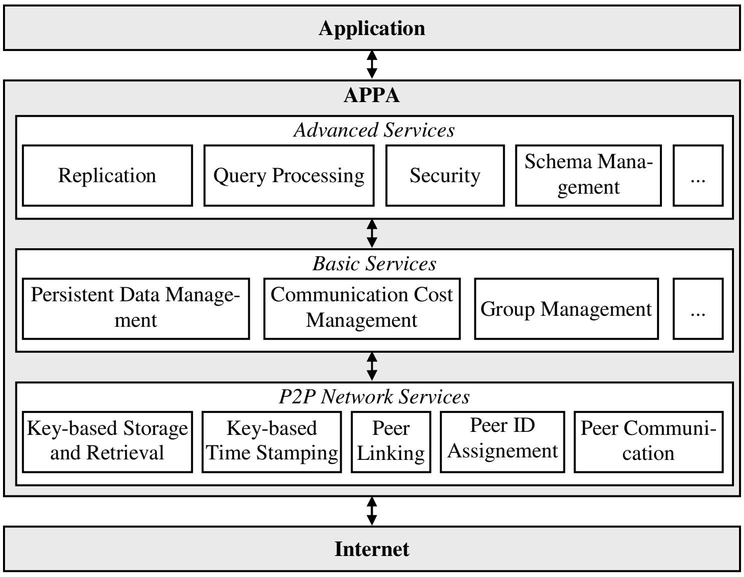


Figure 15. APPA architecture

DATA REPLICATION IN APPA SYSTEM

The APPA replication service [EMP07, MAPV06, MP06, MPJV06] is integrated to the PDM (Persistent Data Management) and KSR (Key-based Storage and Retrieval) services in order to store and retrieve data objects used during reconciliation in a highly available manner. PDM takes advantage of multiple hash functions to precisely place object replicas in the P2P network. With PDM, it is possible to implement the lock and unlock operations over a replicated (*k, object*) pair stored in the P2P network. In addition to PDM, the replication service is integrated to the CCM service (Communication Cost Management), which estimates the communication costs for accessing objects that are stored in the P2P network. These costs are estimated by taking into account latencies and transfer rates as well as the dynamic behavior of nodes that can join and leave the network at any time. The integration of APPA replication service with PDM and CCM is made by means of service interfaces.

In order to make it easy for P2P collaborative applications to take advantage of the APPA replication service, we have defined an application programming interface (API) that abstracts the APPA architecture and works as a façade for the APPA system as a whole by receiving service invocations and internally dispatching such invocations.

We proved the APPA's network-independence by implementing APPA over a super-peer network (JXTA) and two distinct structured networks (Chord and CAN). JXTA provides a good support for the

APPA's P2P Network services. The functionality provided by APPA's peer id assignment, peer linking, and peer communication services are already available in the JXTA core layer. Thus, APPA simply uses JXTA's corresponding functionality. In contrast, JXTA does not provide an equivalent service for key-based storage and retrieval (KSR). Thus, we implemented KSR on top of Meteor [Met06] which is an open-source JXTA service. APPA's advanced services, like replication and query processing, are provided as JXTA community services. The key advantage of APPA implementation is that only its P2P network layer depends on the JXTA platform. Thus, APPA is portable and can be used over other platforms by replacing the services of the P2P network layer.

Chord [SMKK+01] and CAN (Content Addressable Network) [RFHK+01] are two of the most known DHTs. Chord is a simple and efficient DHT that can lookup a data, which is stored at some node in the network, in *O(log n)* routing hops, where *n* is the number of nodes. Its lookup mechanism is provably robust in the face of frequent node failures and re-joins, and it can answer queries even if the system is continuously changing. CAN is based on a logical *d*-dimensional Cartesian coordinate space, which is partitioned into hyper-rectangles, called zones. Each node in the system is responsible for a zone. A data is hashed to a point in the coordinate space, and it is stored at the node whose zone contains the point's coordinates. In CAN, a stored data can be retrieved in $O(dn^{1/d})$ where *n* is the number of nodes.

The validation of the APPA replication service took place over the Grid5000 platform [Gri06]. Grid5000 aims at building a highly reconfigurable, controllable and monitorable experimental Grid platform, gathering 9 sites geographically distributed in France featuring a total of 5000 nodes. Within each site, the nodes are located in the same geographic area and communicate through Gigabit Ethernet links as clusters. Communications between clusters are made through the French academic network (RENATER). Grid5000's nodes are accessible through the OAR batch scheduler from a central user interface shared by all the users of the Grid. A cross-clusters super-batch system, OARGrid, is currently being deployed and tested. The home directories of the users are mounted with NFS on each of the infrastructure's clusters. Data can thus be directly accessed inside a cluster. Data transfers between clusters have to be handled by the users. The storage capacity inside each cluster is a couple of hundreds of gigabytes. Now more than 600 nodes are involved in Grid5000. Additionally, in order to study the scalability of the APPA replication service with larger numbers of nodes that are connected by means of links with variable latencies and bandwidths, we implemented simulators using Java and SimJava [HM98], a process based discrete event simulation package. Simulations were executed on an Intel Pentium IV with a 2.6 GHz processor, and 1 GB of main memory, running the Windows XP operating system. The performance results obtained from the simulator are consistent with those of the replication service prototype.

In the implementation intended for the Grid5000 platform, each peer manages multiple tasks in parallel (e.g. routing DHT messages, executing a DSR step, etc.) by using multithreading and other associated mechanisms (e.g. semaphores); in addition, peers communicate with each other by means of sockets and UDP depending on the message type. To have a topology close to real P2P overlay networks in this Grid platform, we determine the peers' neighbors and we allow that every peer communicate only with its neighbors in the overlay network. Although the Grid5000 provides fast and reliable communication, which usually is not the case for P2P systems, it allows to validate the accuracy of APPA distributed algorithms and to evaluate the scalability of APPA services. We have deployed APPA over this platform because it was the largest network available to perform our experiments in a controllable manner. On the other hand, the implementation of the simulator conforms to the SimJava model with respect to parallel processing and peers communication. It is important to note that, in our simulator, only the P2P network topology and peer communications are simulated; full-fledged APPA services are deployed on top of the simulated network.

ANNEX B: PROOFS

This annex contains the proofs that P2P-reconciler assures eventual consistency among replicas and works correctly in the presence of failures. The proofs for P2P-reconciler-TA are identical to the corresponding proofs of the P2P-reconciler.

EVENTUAL CONSISTENCY

We first prove that P2P-reconciler assures eventual consistency among replicas. This proof assumes that the reconciliation objects stored in DHT are available according to the high availability property of the APPA's PDM service. In addition, we assume that P2P-reconciler is used in the context of a virtual community. Members of a virtual community have common interests and actively participate on collaborative applications. However, they can leave the community at any time thereby ceasing forever their participation. Thus, the *active nodes* involved in a collaborative application may change with time.

Definition B.1 (active node) *A node is active with respect to a collaborative application if it is connected to the application or "temporarily" disconnected. A temporary disconnection can be caused by a failure or a transient pause on the collaboration, and therefore it is followed by at least one more reconnection.*

Lemma B.1 *All active nodes apply reconciled actions to the local replicas in the same order.*
Proof We first show that reconciled actions coming from different executions of the P2P-reconciler protocol are ordered.

- Each execution of the P2P-reconciler produces a schedule. Since a schedule is an ordered list of actions that do not violate constraints, actions of the same schedule are ordered.
- Assume now that $S_1 \rightarrow S_2 \rightarrow \dots \rightarrow S_k$ is a sequence of schedules produced by the P2P-reconciler protocol respectively at times t_1, t_2, …, t_k. Since it is disallowed to launch parallel executions of P2P-reconciler, $t_1 < t_2 < \dots < t_k$, and then we use the execution sequence to order schedules. This ordering is stored in the schedule history reconciliation object in the form of an ordered list of schedule identifiers (i.e. $H = [S_1^{id}, S_2^{id}, \dots, S_k^{id}]$). If schedules are ordered and reconciled actions inside every schedule are also ordered, then all reconciled actions produced by distinct executions of the P2P-reconciler are ordered.

Since all active nodes apply reconciled actions to its local replicas according to the order established in the schedule history H, all active nodes apply reconciled actions in the same order □

Lemma B.2 *All active nodes eventually apply all reconciled actions to their local replicas.*
Proof We have to show that if all active nodes stop the production of update actions so that at time t_i the P2P-reconciler concludes its last reconciliation (i.e. at t_i all actions are reconciled), then there is a time t_j, $t_j > t_i$, at which all active nodes will have applied all schedules produced by the P2P-reconciler protocol. Let H be the schedule history (noted $H = [S_1^{id}, S_2^{id}, \dots, S_k^{id}]$), n be an active node, and S_l^{id} be the identifier of the last schedule locally applied by n (n knows S_l^{id}). P2P-reconciler works as follows. Whenever n connects, it locally applies all schedules that succeed S_l^{id} in the H's ordered list in order to refresh its local replicas with actions that were reconciled while n was disconnected. In addition, n repeats this refreshment operation whenever n disconnects in order to apply actions that were reconciled while it was connected, if any exists. Since n is an active node, it is either connected or temporarily disconnected (i.e. it will reconnect at least one more time) at time t_i. Thus, if n is connected at time t_i, n will apply all schedules produced by the P2P-reconciler when it disconnects at time t_d ($t_d > t_i$). However, if n is disconnected at time t_i, n will apply all schedules when it reconnects at time t_r ($t_r > t_i$). Consider now that the set TFS (Times at which Final Sates were achieved) holds all times t_r and t_d associated with all active nodes. Since no more update actions are produced after t_i, the time t_j at which all active nodes will have applied all schedules produced by the P2P-reconciler protocol is the maximal value belonging to TFS. □

Theorem B.1 *The P2P-reconciler protocol assures eventual consistency among replicas that are stored in active nodes of a collaborative application.*

Proof In this proof we assume that all replicas R_1, R_2, …, R_i, of the object R have the same initial state. Thus, we have to show that the same set of reconciled actions is applied to all such replicas in the same order. If R_1, R_2, …, R_i are held by active nodes of a collaborative application, all reconciled actions are eventually applied to these replicas (Lemma B.2) in the same order (Lemma B.1). □

CORRECTNESS

We prove in this section that P2P-reconciler is correct as it assures eventual consistency among replicas even in the presence of failures. This proof assumes that the reconciliation objects stored in DHT are available according to the high availability property of the APPA's PDM service. It also assumes synchronous network communication for supporting the subset of messages that the P2P-reconciler protocol cannot lose. We use n_{start} to denote the node that starts the reconciliation.

Lemma B.3 *The P2P-reconciler protocol is resilient to failure on the n_{start} node.*

Proof The n_{start} node is responsible for locking the schedule history, notifying the start of reconciliation to provider nodes, and requesting the cost provider for allocating reconciler nodes. Thus, if n_{start} fails while launching the reconciliation, the following problems could happen: (1) the schedule history could remain forever locked; and (2) the provider nodes could wait forever for reconciler requests. We have to show that the P2P-reconciler protocol avoids such problems. In our solution, provider nodes are able to estimate the time required to perform the reconciliation. As a result, if a provider node *n* realizes that it is inactive for a long time wrt. the estimated reconciliation time, *n* infers that the reconciliation has crashed and initiates a recovery procedure, which first notifies the abnormal end of reconciliation to other provider nodes, and then requests that the schedule history provider unlocks the schedule history. Notice that any provider node is able to detect the reconciliation crash and perform the recovery procedure. For this reason, there is no problem if *n* fails while recovering. In this case, another provider node will detect the crash later on and repeat the recovery procedure; duplicated notifications of crash and duplicated requests for unlock the schedule history are discarded. Since provider nodes no longer wait for requests and the schedule history is unlocked, the P2P-reconciler protocol is resilient to failure on the n_{start} node. □

Lemma B.4 *The P2P-reconciler protocol is resilient to failure on the cost provider node.*

Proof The cost provider node is responsible for selecting and notifying reconciler nodes. Thus, if cost provider fails, the following problems could happen: (1) none reconciler node is allocated; or (2) only a subset of selected nodes is notified of allocation. We have to show that reconciliation can be normally restarted after the cost provider failure. In practice, problem 1 is equivalent to n_{start} failure, i.e. if none reconciler is allocated, the schedule history could remain forever locked and the provider nodes could wait forever for reconciler requests. We proved in Lemma B.3 that the P2P-reconciler protocol works properly in this case. On the other hand, if some reconcilers are already notified when the cost provider fails, two scenarios are possible: (a) the reconciliation succeeds even with the reduced number of allocated reconcilers; or (b) the reconciliation crashes at time t_c due to the lack of reconcilers. In the latter case, it is likely that the reconciliation objects have been updated. Thus, the recovery procedure works as follows. The provider node *n* that detects the reconciliation crash notifies this fact to other provider nodes, which, in turn, undo updates performed on reconciliation objects up to time t_c, and then quit the reconciliation. In addition, *n* requests that the schedule history provider unlocks the schedule history. As explained in the proof of Lemma B.3, there is no problem if *n* fails while performing the recovery procedure. Since provider nodes undo updates on reconciliation objects before quitting the reconciliation and the schedule history is unlocked, the reconciliation can be normally restarted and, as a result, the P2P-reconciler protocol is resilient to failure on the cost provider node. □

Lemma B.5 *A reconciliation step "i" terminates properly if at least one reconciler node allocated to step "i" works properly until the end of "i".*

Proof P2P-reconciler protocol is composed of one allocation step (step 1) followed by five reconciliation steps (steps from 2 to 6). We have to show that if at least one reconciler node works properly until the end of each step from 2 to 6, the reconciliation as a whole succeeds. We first show that one reconciler is enough to successfully terminate step 2, and then we generalize the main principles for other steps.

– In step 2, reconciler nodes take actions from the action log providers and store back groups of potentially conflicting actions. On the one side, reconcilers remain requesting actions and storing back groups until the action log provider indicates that there are no more actions to group. On the other side, the action log provider supplies actions to reconcilers and waits for the corresponding acknowledgements that indicate the successful processing of such actions. These acknowledgements are carried by requests for storing groups. After a given delay, actions that were not acknowledged are redistributed to reconcilers that have requested more actions. This redistribution repeats until all actions have been acknowledged. In addition, the action log provider discards duplicated requests for storing groups, if any exists. Suppose now that only a reconciler *n* works properly during step 2. In this case, *n* repeatedly requests actions and stores back the associated groups until the action log provider indicates the end of actions and, as a result, step 2 terminates successfully.

– The general principles applied on step 2 ($i = 2$) are described as follows. Let *maxRec* be the maximal number of reconcilers per step. Step *i* is divided into *k* cycles, where $1 \leq k \leq maxRec$. At each cycle, all reconcilers that still work properly request inputs from provider nodes and give back the associated acknowledgements in order to indicate the successful processing of inputs. This goal is achieved with no additional network traffic as the acknowledgments are inserted in the regular messages of the P2P-reconciler protocol. Provider nodes on the other hand discard duplicated update requests, if any exists, and control the end of step cycles. Because of the number of inputs to be distributed is equal to *maxRec*, if all reconcilers work properly in step *i*, *i* only needs one cycle to successfully terminate. However, if only one reconciler works properly during step *i*, *maxRec* cycles need to be performed until the end of step *i*.

Since all steps from 2 to 6 apply the general principles explained above, every reconciliation step *i* terminates properly if at least one reconciler node works properly until the end of *i*. □

Lemma B.6 *The P2P-reconciler protocol is resilient to failures on reconciler nodes.*

Proof We have to show that after a reconciler failure either the reconciliation terminates correctly or it can be normally restarted later on. Let *n* be the faulty reconciler node. We directly infer from Lemma B.5 that if *n* is not the last alive reconciler of a reconciliation step then the reconciliation terminates correctly. Otherwise, the reconciliation crashes due to the lack of reconcilers for concluding the step to which *n* is allocated. We proved in Lemma B.4 that in this case the reconciliation can be normally restarted. Since after a reconciler failure either the reconciliation terminates correctly or it can be normally restarted, the P2P-reconciler protocol is resilient to failures on reconciler nodes. □

Theorem B.2 *The P2P-reconciler protocol is correct even in the presence of failures.*

Proof The execution of P2P-reconciler protocol involves four types of nodes: the node that starts the reconciliation (n_{start}), the cost provider, the reconciler nodes, and other nodes that hold reconciliation objects in DHT. Since we assume available reconciliation objects, we do not discuss failures at nodes that hold these objects. Thus, we have only to show that the P2P-reconciler protocol is resilient to failures on n_{start}, cost provider, and reconciler nodes. This is proved respectively in Lemmas B.3, B.4, and B.6. □

ANNEX E

Data Currency in Replicated DHTs

R. Akbarinia, E. Pacitti, P. Valduriez

***ACM SIGMOD Int. Conf. on Management of Data (SIGMOD),* Pékin, Chine**

Data Currency in Replicated DHTs[1]

Reza Akbarinia Esther Pacitti Patrick Valduriez
Atlas team, INRIA and LINA
University of Nantes, France
{FirstName.LastName@univ-nantes.fr, Patrick.Valduriez@inria.fr}

ABSTRACT

Distributed Hash Tables (DHTs) provide a scalable solution for data sharing in P2P systems. To ensure high data availability, DHTs typically rely on data replication, yet without data currency guarantees. Supporting data currency in replicated DHTs is difficult as it requires the ability to return a current replica despite peers leaving the network or concurrent updates. In this paper, we give a complete solution to this problem. We propose an Update Management Service (UMS) to deal with data availability and efficient retrieval of current replicas based on timestamping. For generating timestamps, we propose a Key-based Timestamping Service (KTS) which performs distributed timestamp generation using local counters. Through probabilistic analysis, we compute the expected number of replicas which UMS must retrieve for finding a current replica. Except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of available replicas are current then the expected number of retrieved replicas is less than 3. We validated our solution through implementation and experimentation over a 64-node cluster and evaluated its scalability through simulation up to 10,000 peers using SimJava. The results show the effectiveness of our solution. They also show that our algorithm used in UMS achieves major performance gains, in terms of response time and communication cost, compared with a baseline algorithm.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *distributed databases, concurrency, query processing.*

General Terms

Algorithms, performance, reliability.

Keywords

Peer-to-Peer, distributed hash table (DHT), data availability, data currency, data replication

1. INTRODUCTION

Peer-to-peer (P2P) systems adopt a completely decentralized approach to data sharing and thus can scale to very large amounts of data and users. Popular examples of P2P systems such as Gnutella [9] and KaaZa [12] have millions of users sharing petabytes of data over the Internet. Initial research on P2P systems has focused on improving the performance of query routing in unstructured systems, such as Gnutella and KaaZa, which rely on flooding. This work led to structured solutions based on distributed hash tables (DHT), *e.g.* CAN [19], Chord [29], and Pastry [23]. While there are significant implementation differences between DHTs, they all map a given key k onto a peer p using a hash function and can lookup p efficiently, usually in O(*log n*) routing hops where n is the number of peers [5]. DHTs typically provide two basic operations [5]: *put(k, data)* stores a key k and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with k in the DHT.

One of the main characteristics of P2P systems is the dynamic behavior of peers which can join and leave the system frequently, at anytime. When a peer gets offline, its data becomes unavailable. To improve data availability, most DHTs rely on data replication by storing (*k, data*) pairs at several peers, *e.g.* using several hash functions [19]. If one peer is unavailable, its data can still be retrieved from the other peers that hold a replica. However, the mutual consistency of the replicas after updates can be compromised as a result of peers leaving the network or concurrent updates. Let us illustrate the problem with a simple update scenario in a typical DHT. Let us assume that the operation *put*(k, d_0) (issued by some peer) maps onto peers p_1 and p_2 which both get to store the data d_0. Now consider an update (from the same or another peer) with the operation *put*(k, d_1) which also maps onto peers p_1 and p_2. Assuming that p_2 cannot be reached, *e.g.* because it has left the network, then only p_1 gets updated to store d_1. When p_2 rejoins the network later on, the replicas are not consistent: p_1 holds the *current* state of the data associated with k while p_2 holds a *stale* state. Concurrent updates also cause inconsistency. Consider now two updates *put*(k, d_2) and *put*(k, d_3) (issued by two different peers) which are sent to p_1 and p_2 in reverse order, so that p_1's last state is d_2 while p_2's last state is d_3. Thus, a subsequent *get(k)* operation will return either stale or current data depending on which peer is looked up, and there is no way to tell whether it is current or not. For some applications (*e.g.* agenda management, bulletin boards, cooperative auction management, reservation management, etc.) which could take advantage of a DHT, the ability to get the current data is very important.

Many solutions have been proposed in the context of distributed database systems for managing replica consistency [17] but the

[1] Work partially funded by ARA "Massive Data" of the French ministry of research and the European Strep Grid4All project.

SIGMOD'07, June 11–14, 2007, Beijing, China.

high numbers and dynamic behavior of peers make them no longer applicable to P2P [6]. Supporting data currency in replicated DHTs requires the ability to return a current replica despite peers leaving the network or concurrent updates. The problem is partially addressed in [13] using data versioning. Each replica has a version number which is increased after each update. To return a current replica, all replicas need to be retrieved in order to select the latest version. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica.

In this paper, we give a complete solution to data availability and data currency in replicated DHTs. Our main contributions are the following:

- We propose a service called Update Management Service (UMS) which deals with improving data availability and efficient retrieval of current replicas based on timestamping. After retrieving a replica, UMS detects whether it is current or not, *i.e.* without having to compare with the other replicas, and returns it as output. Thus, in contrast to the solution in [13], UMS does not need to retrieve all replicas to find a current one. In addition, concurrent updates raise no problem for UMS.
- We give a probabilistic analysis of UMS's communication cost. We compute the expected number of replicas which UMS must retrieve for finding a current replica. We prove that it is less than the inverse of the probability of currency and availability, *i.e.* the probability that a replica is current and available. Thus, except for the cases where the availability of current replicas is very low, the expected number of replicas which UMS must retrieve is typically small.
- We propose a new Key-based Timestamping Service (KTS) which generates monotonically increasing timestamps, in a distributed fashion using local counters. KTS does distributed timestamp generation in a way that is similar to data storage in the DHT, *i.e.* using peers dynamically chosen by hash functions. To maintain timestamp monotonicity, we propose algorithms which take into account the cases where peers leave the system either normally or not (*e.g.* because they fail). To the best of our knowledge, this is the first paper that introduces the concept of key-based timestamping, and proposes efficient techniques for realizing this concept in DHTs. Furthermore, KTS is useful to solve other DHT problems which need a total order on operations performed on each data, *e.g.* read and write operations which are performed by concurrent transactions.
- We provide a comprehensive performance evaluation based on the implementation of UMS and KTS over a 64-node cluster. We also evaluated the scalability of our solution through simulation up to 10,000 peers using SimJava. The experimental and simulation results show the effectiveness of our solution.

The rest of this paper is organized as follows. In Section 2, we first propose a model for DHTs which will be useful to present our solution, and then we state the problem. Section 3 presents our update management service for DHTs. In Section 4, we propose a distributed timestamping service to support updates. Section 5 describes a performance evaluation of our solution through implementation and simulation. In Section 6, we discuss related work. Section 7 concludes.

2. DHT MODEL AND PROBLEM STATEMENT

In this section, we first present a model of DHTs which is needed for describing our solution and proving its properties. Then, we precisely state the problem.

2.1 DHT Model

A DHT maps a key k to a peer p using a hash function h. We call p the *responsible for k wrt h*. A peer may be responsible for k wrt a hash function h_1 but not responsible for k wrt another hash function h_2. The responsible for k wrt h may be different at different times, *i.e.* because of peers' joins and leaves. We can model the mapping mechanism of DHT as a function that determines at anytime the peer that is responsible for k wrt h; we call this function *DHT's mapping function*.

Definition 1: DHT's mapping function. *Let K be the set of all keys accepted by the DHT, P the set of peers, H the set of all pairwise independent hash functions which can be used by the DHT for mapping, and T the set of all numbers accepted as time. We define the DHT's mapping function as $m: K \times H \times T \rightarrow P$ such that $m(k,h,t)$ determines the peer $p \in P$ which is responsible for $k \in K$ wrt $h \in H$ at time $t \in T$.*

Let us make precise the terminology involving peers' responsibility for a key. Let $k \in K$, $h \in H$ and $p \in P$, and let $[t_0..t_1)$ be a time interval such that $t_1 > t_0$. We say that p is *continuously responsible* for k wrt h in $[t_0..t_1)$ if it is responsible for k wrt h at anytime in $[t_0..t_1)$. In other words, $(\forall t \in T,\ t_0 \leq t < t_1\ ) \Rightarrow (\ p = m(k,h,t))$. If p obtains and loses the responsibility for k wrt h respectively at t_0 and t_1, and is continuously responsible for k wrt h in $[t_0..t_1)$, then we say that $[t_0..t_1)$ is a *p's period of responsibility* for k wrt h. The peer that is responsible for k wrt h at current time is denoted by $rsp(k,h)$. We also denote by $prsp(k,h)$ the peer that was responsible for k wrt h just before $rsp(k,h)$. The peer that will become responsible for k wrt h just after $rsp(k,h)$ is denoted by $nrsp(k,h)$.

Example 1. Figure 1 shows the peers responsible for $k \in K$ wrt $h \in H$ since t_0. The peer that is currently responsible for k wrt h is p_1, thus $p_1 = rsp(k,h)$ and $p_3 = prsp(k,h)$. In the time interval $[t_1..t_2)$, p_2 is continuously responsible for k wrt h. It has obtained and lost its responsibility respectively at t_1 and t_2, thus $[t_1..t_2)$ is p_2's period of responsibility for k wrt h. Also $[t_0..t_1)$ and $[t_2..t_3)$ are respectively p_4's and p_3's periods of responsibility for k wrt h.

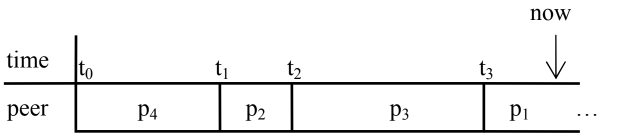


Figure 1. Example of peers' responsibilities

In the DHT, there is a *lookup service* that can locate $rsp(k,h)$ efficiently. The lookup service can return the address of $rsp(k,h)$ usually in $O(Log\ |P|)$ routing hops, where $|P|$ is the number of peers in the system.

2.2 Problem Statement

To improve data availability we replicate the pairs *(k, data)* at several peers using several hash functions. We assume that there is an operation that stores a pair *(k, data)* at *rsp(k,h)* which we denote by $put_h(k, data)$. This operation can be issued concurrently by several peers. There is another operation, denoted by $get_h(k)$, that retrieves the data associated with *k* which is stored at *rsp(k,h)*.

Over time, some of the replicas stored with *k* at some peers may get stale. Our objective is to provide a mechanism which returns efficiently a current replica in response to a query requesting the data associated with a key.

Formally, the problem can be defined as follows. Given a key $k \in K$, let R_k be the set of replicas such that for each $r \in R_k$, the pair *(k, r)* is stored at one of the peers of the DHT. Our goal is to return efficiently an $r \in R_k$ which is current, *i.e.* reflects the latest update.

3. UPDATE MANAGEMENT SERVICE

To deal with data currency in DHTs, we propose an *Update Management Service (UMS)* which provides high data availability through replication and efficient retrieval of current replicas. UMS only requires the DHT's lookup service with put_h and get_h operations. To return current replicas, it uses timestamps attached to the pairs *(k, data)*. In this section, we give an overview of our timestamping solution and present in more details UMS' update operations. We also analyze UMS's communication cost.

3.1 Timestamping

To provide high data availability, we replicate the data in the DHT using a set of pairwise independent hash functions $H_r \subset H$ which we call *replication hash functions*. To be able to retrieve a current replica we "stamp" each pair *(k, data)* with a logical timestamp, and for each $h \in H_r$ we replicate the pair *(k, newData)* at *rsp(k,h)* where *newData={data, timestamp}*, i.e. newData is a data composed of the initial data and the timestamp. Upon a request for the data associated with a key, we can thus return one of the replicas which are stamped with the latest timestamp. The number of replication hash functions, *i.e.* $|H_r|$, can be different for different DHTs. For instance, if in a DHT the availability of peers is low, for increasing data availability a high value of $|H_r|$ (*e.g.* 30) is used. Constructing H_r, which is a set of pairwise independent hash functions, can be done easily, *e.g.* by using the methods presented in [14].

To generate timestamps, we propose a distributed service called *Key-based Timestamping Service (KTS)*. The main operation of KTS is *gen_ts(k)* which given a key *k* generates a real number as a *timestamp for k*. The timestamps generated by KTS have the *monotonicity* property, *i.e.* two timestamps generated for the same key are monotonically increasing. This property permits us to order the timestamps generated for the same key according to the time at which they have been generated.

Definition 2: Timestamp monotonicity. *For any two timestamps ts_1 and ts_2 generated for a key k respectively at times t_1 and t_2, if $t_1 < t_2$ then we have $ts_1 < ts_2$.*

At anytime, KTS generates at most one timestamp for a key (see Section 4 for the details). Thus, regarding to the monotonicity property, there is a total order on the set of timestamps generated for the same key. However, there is no total order on the timestamps generated for different keys.

```
insert(k, data)
begin
   ts := KTS.gen_ts (k);
   for  each  $h \in H_r$  do
      newData := {data, ts};
      DHT.$put_h$(k, newData);
end;

retrieve(k)
begin
   $ts_1$ := KTS.last_ts(k);
   $data_{mr}$ := null;
   $ts_{mr}$ := - ∞;
   for  each  $h \in H_r$  do begin
       newData := DHT.$get_h$(k);
      data := newData.data;
      ts := newData.ts;
      if ($ts_1$ = ts) then begin
         return data; // one current
                      // replica is found
         exit;
      end
      else if  (ts > $ts_{mr}$) then  begin
         $data_{mr}$ := data;//keep the most
         $ts_{mr}$ := ts;//recent replica and
                    //its timestamp
      end;
   end;
    return $data_{mr}$
end;
```

Figure 2. UMS update operations

KTS has another operation denoted by *last_ts(k)* which given a key *k* returns the last timestamp generated for *k* by KTS.

3.2 Update Operations

To describe UMS, we use the *KTS.gen_ts* and *KTS.last_ts* operations discussed above. The implementation of these operations is detailed in Section 4. UMS provides *insert* and *retrieve* operations (see Figure 2).

Insert(k, data): inserts a pair *(k, data)* in the DHT as follows. First, it uses KTS to generate a timestamp for *k*, *e.g. ts*. Then, for each $h \in H_r$ it sends the pair *(k, {data, ts})* to the peer that is *rsp(k,h)*. When a peer *p*, which is responsible for *k* wrt one of the hash functions involved in H_r, receives the pair *(k, {data, ts})*, it compares *ts* with the timestamp, say ts_0, of its data (if any) associated with *k*. If $ts > ts_0$, *p* overwrites its data and timestamp with the new ones. Recall that, at anytime, *KTS.gen_ ts (k)* generates at most one timestamp for *k*, and different timestamps for *k* have the monotonicity property. Thus, in the case of concurrent calls to *insert(k, data)*, *i.e.* from different peers, only the one that obtains the latest timestamp will succeed to store its data in the DHT.

Retrieve(k): retrieves the most recent replica associated with *k* in the DHT as follows. First, it uses KTS to determine the latest timestamp generated for *k*, *e.g.* ts_1. Then, for each hash function $h \in H_r$, it uses the DHT operation $get_h(k)$ to retrieve the pair *{data, timestamp}* stored along with *k* at *rsp(k,h)*. If *timestamp* is equal to ts_1, then the data is a current replica which is returned as output and the operation ends. Otherwise the retrieval process continues while saving in $data_{mr}$ the most recent replica. If no replica with a timestamp equal to ts_1 is found (*i.e.* no current replica is found) then the operation returns the most recent replica which is available, *i.e.* $data_{mr}$.

3.3 Cost Analysis

In this section, we give a probabilistic analysis of the communication cost of UMS in terms of number of messages to retrieve a data item. For a non replicated DHT, this cost, which we denote by c_{ret}, is *O(log n)* messages where *n* is the number of peers. The communication cost of retrieving a current replica by UMS is $c_{ums} = c_{kts} + n_{ums} * c_{ret}$, where c_{kts} is the cost of returning the last generated timestamp by KTS and n_{ums} is the number of replicas that UMS retrieves, *i.e.* the number of times that the operation $get_h(k)$ is called. As we will see in the next section, c_{kts} is usually equal to c_{ret}, *i.e.* the cost of contacting the responsible of a key and getting the last timestamp from it. Thus, we have $c_{ums} = (1 + n_{ums}) * c_{ret}$.

The The number of replicas which UMS retrieves, *i.e.* n_{ums}, depends on the probability of currency and availability of replicas. The higher this probability, the lower n_{ums} is. Let H_r be the set of replication hash functions, *t* be the retrieval time, and p_t be the probability that, at time *t,* a current replica is available at a peer that is responsible for *k* wrt some $h \in H_r$. In other words, p_t is the ratio of current replicas, which are available at *t* over the peers responsible for *k* wrt replication hash functions, to the total number of replicas, *i.e.* $/H_r/$. We call p_t the *probability of currency and availability* at retrieval time. We give a formula for computing the expected value of the number of replicas, which UMS retrieves, in terms of p_t and $/H_r/$. Let *X* be a random variable which represents the number of replicas that UMS retrieves. We have $Prob(X=i) = p_t * (1- p_t)^{i-1}$, *i.e.* the probability of having *X=i* is equal to the probability that *i-1* first retrieved replicas are not current and the *i*th replica is current. The expected value of *X* is computed as follows:

$$E(X) = \sum_{i=0}^{|H_r|} i * \Pr ob(X = i)$$

$$E(X) = p_t * (\sum_{i=0}^{|H_r|} i * (1 - p_t)^{i-1}) \qquad (1)$$

Equation 1 expresses the expected value of the number of retrieved replicas in terms of p_t and $/H_r/$. Thus, we have the following upper bound for *E(X)* which is solely in terms of p_t:

$$E(X) < p_t * (\sum_{i=0}^{\infty} i * (1 - p_t)^{i-1}) \qquad (2)$$

From the theory of series [2], we use the following equation for $0 \le z < 1$:

$$\sum_{i=0}^{\infty} i * z^{i-1} = \frac{1}{(1-z)^2}$$

Since $0 \le (1- p_t) < 1$, we have:

$$\sum_{i=0}^{\infty} i * (1 - p_t)^{i-1} = (\frac{1}{(1-(1-p_t))^2} \qquad (3)$$

Using Equations 3 and 2, we obtain:

$$E(X) < \frac{1}{p_t} \qquad (4)$$

Theorem 1: *The expected value of the number of replicas which UMS retrieves is less than the inverse of the probability of currency and availability at retrieval time.*

Proof: Implied by the above discussion.□

Example. Assume that at retrieval time *35%* of replicas are current and available, *i.e.* $p_t=0.35$. Then the expected value of the number of replicas which UMS retrieves is less than 3.

Intuitively, the number of retrieved replicas cannot be more than $/H_r/$. Thus, for *E(X)* we have:

$$E(X) \le \min(\frac{1}{p_t}, |H_r|) \qquad (5)$$

4. KEY-BASED TIMESTAMP SERVICE

The main operation of KTS is *gen_ts* which generates monotonically increasing timestamps for keys. A centralized solution for generating timestamps is obviously not possible in a P2P system since the central peer would be a bottleneck and single point of failure. Distributed solutions using synchronized clocks no longer apply in a P2P system. One popular method for distributed clock synchronization is Network Time Protocol (NTP) which was originally intended to synchronize computers linked via Internet networks [16]. NTP and its extensions (*e.g.* [8] and [18]) guarantee good synchronization precision only if computers have been linked together long enough and communicate frequently [18]. However, in a P2P system in which peers can leave the system at any time, these solutions cannot provide good synchronization precision.

In this section, we propose a distributed technique for generating timestamps in DHTs. First, we present a technique based on local counters for generating the timestamps. Then we present a direct algorithm and an indirect algorithm for initializing the counters, which is very important for guaranteeing the monotonicity of timestamps. We also apply the direct algorithm to CAN and Chord. Finally, we discuss a method for maintaining the validity of counters.

4.1 Timestamp Generation

Our idea for timestamping in DHTs is like the idea of data storage in these networks which is based on having a peer responsible for storing each data and determining the peer dynamically using a hash function. In KTS, for each key we have a peer responsible for timestamping which is chosen dynamically using a hash function. Below, we discuss the details of timestamp responsibility and timestamp generation.

4.1.1 Timestamping Responsibility

Timestamp generation is performed by KTS as follows. Let $k \in K$ be a key, the *responsible of timestamping for k* is the peer that is responsible for *k* wrt h_{ts}, *i.e.* $rsp(k, h_{ts})$, where h_{ts} is a hash function accepted by the DHT, i.e. $h_{ts} \in H$. Each peer *q* that needs a timestamp for *k,* called *timestamp requester*, uses the DHT's lookup service to obtain the address of $rsp(k, h_{ts})$ to which it sends a *timestamp request (TSR)*. When $rsp(k, h_{ts})$ receives the request of *q*, generates a timestamp for *k* and returns it to *q*. Figure 3 illustrates the generation of a timestamp for *k* initiated by peer *q*.

If the peer that is *rsp(k, h_{ts})* leaves the system or fails, the DHT detects the absence of that peer, *e.g.* by frequently sending "ping" messages from each peer to its neighbors [19], and another peer becomes responsible for *k* wrt h_{ts}. Therefore, if the responsible of timestamping for *k* leaves the system or fails, another peer automatically becomes responsible of timestamping for *k*, *i.e.* the peer that becomes responsible for *k* wrt h_{ts}. Thus, the dynamic behavior of peers causes no problem for timestamping responsibility.

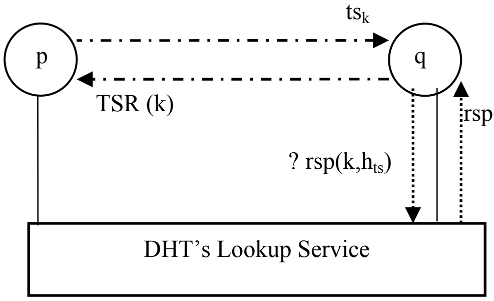


Figure 3. Example of timestamp generation

4.1.2 Guaranteeing Monotonicity

Let us now discuss what a responsible of timestamping should do to maintain the monotonicity property. Let *k* be a key, *p* the peer that is responsible of timestamping for *k*, and ts_k a timestamp for *k* which is generated by *p*. To provide the monotonicity property, we must guarantee two constraints: (1) ts_k is greater than all timestamps for *k* which have been previously generated by *p* itself; (2) ts_k is greater than any timestamp for *k* generated by any other peer that was responsible of timestamping for *k* in the past.

To enforce the first constraint, for generating timestamps for each key *k*, we use a local *counter of k at p* which we denote as $c_{p,k}$. When *p* receives a timestamp request for *k*, it increments the value of $c_{p,k}$ by one and returns it as the timestamp for *k* to the timestamp requester.

To enforce the second constraint, *p* should *initialize* $c_{p,k}$ so that it is greater than or equal to any timestamp for *k* previously generated by other peers that were responsible of timestamping for *k* in the past. For this, *p* initializes $c_{p,k}$ to the last value of $c_{q,k}$ where *q* is the last peer that has generated a timestamp for *k*. In Section 4.2, we discuss how *p* can acquire $c_{q,k}$. The following lemma shows that the initialization of $c_{p,k}$ as above enforces the second constraint.

Lemma 1: *If each peer p, during each of its periods of responsibility for k wrt h_{ts}, initializes $c_{p,k}$ before generating the first timestamp for k, then each generated timestamp for k is greater than any previously generated one.*

Proof: Follows from the fact that initializing $c_{p,k}$ makes it equal to the last timestamp generated for *k*, and the fact that timestamp generation is done by increasing the value of $c_{p,k}$ by one and returning its value as output. □

After $c_{p,k}$ has been initialized, it is a *valid counter*, *i.e. p* can use it for generating timestamps for *k*. If *p* loses the responsibility for *k* wrt h_{ts}, *e.g.* because of leaving the system, then $c_{p,k}$ becomes *invalid*. The peer *p* keeps its valid counters in a *Valid Counters Set* which we denote by VCS_p. In other words, for each $k \in K$, if $c_{p,k}$ is valid then $c_{p,k}$ is in VCS_p. Each peer $p \in P$ has its own VCS_p and respects the following rules for it:

1. When *p* joins the P2P system, it sets $VCS_p = \varnothing$.
2. $\forall k \in K$, when *p* initializes $c_{p,k}$, it adds $c_{p,k}$ to VCS_p.
3. $\forall k \in K$, when *p* loses the responsibility for *k* wrt h_{ts}, if $c_{p,k}$ is in VCS_p then *p* removes it from VCS_p.

```
gen-ts(k) // timestamp generation by KTS
begin
   p := DHT.lookup(k, h_ts);
   return gen-ts(p, k);
end;

gen-ts(p, k) //generating a timestamp
            // for a key k by peer p
            // that is rsp(k, h_ts)
begin
   c_{p,k} := search_counter(VCS_p, k);
   if (c_{p,k} is not in VCS_p) then
   begin
      new(c_{p,k});//allocate memory for c_{p,k}
      KTS.CounterInitialize(k, c_{p,k});
      VCS_p := VCS_p + {c_{p,k}};
   end;
   c_{p,k}.value := c_{p,k}.value + 1;
   return c_{p,k}.value;
end;
```

Figure 4. Timestamp generation

When *p* receives a timestamp request for a key *k*, it checks for the existence of $c_{p,k}$ in VCS_p. If $c_{p,k}$ is in VCS_p then *p* generates the timestamp for *k* using $c_{p,k}$. Otherwise *p* initializes $c_{p,k}$, appends it to VCS_p and then generates the timestamp using $c_{p,k}$ (see Figure 4).

The data structure used for VCS_p is such that given a key *k* seeking $c_{p,k}$ in VCS_p can be done rapidly, *e.g.* a binary search tree. Also, for minimizing the memory cost, when a counter gets out of VCS_p, *p* releases the memory occupied by the counter, *i.e.* only the counters involved in VCS_p occupy a memory location. To prevent the problem of overflow, we use a large integer, *e.g.* 128 bits, for the value of $c_{p,k}$.

The following theorem shows that using VCS_p and respecting its rules guarantees the monotonicity property.

Theorem 2: *If the peer p, which is responsible for k wrt h_{ts}, for generating timestamps for k uses $c_{p,k}$ that is in VCS_p, then each generated timestamp for k is greater than any previously generated one.*

Proof: Let $[t_0, t_1)$ be a *p*'s period of responsibility for *k* wrt h_{ts} and let us assume that *p* generates a timestamp for *k* in $[t_0, t_1)$. Rules 1 and 3 assure that at t_0, $c_{p,k}$ is not in VCS_p. Thus, for generating the first timestamp for *k* in $[t_0, t_1)$, *p* should initialize $c_{p,k}$ and insert it into VCS_p (Rule 2). Therefore, in each of its periods of responsibility for *k* wrt h_{ts}, *p* initializes $c_{p,k}$ before generating the first timestamp for *k*. Thus, each peer *p*, during each of its periods of responsibility for *k* wrt h_{ts}, initializes $c_{p,k}$ before generating the first timestamp for *k*, so by Lemma 1 the proof is complete. □

The other KTS operation *last_ts(k)*, which we used in Section 3, can be implemented like *gen_ts* except that *last_ts* is simpler: it only returns the value of $c_{p,k}$ and does not need to increase its value.

4.2 Counter Initialization

Initializing the counters is very important for maintaining the monotonicity property. Recall that for initializing $c_{p,k}$, the peer *p*, which is responsible of timestamping for *k*, assigns to $c_{p,k}$ the value of $c_{q,k}$ where *q* is the last peer that has generated a timestamp for *k*. But, the question is how *p* can acquire $c_{q,k}$. To answer this question, we propose two *initialization algorithms*: *direct* and *indirect*. The direct algorithm is based on transferring directly the counters from a responsible of timestamping to the next responsible. The indirect algorithm is based on retrieving the value of the last generated timestamp from the DHT.

4.2.1 Direct Algorithm for Initializing Counters

With the direct algorithm, the initialization is done by directly transferring the counters from a responsible of timestamping to the next one at the end of its responsibility. This algorithm is used in situations where the responsible of timestamping loses its responsibility in a normal way, *i.e.* it does not fail.

Let *q* and *p* be two peers, and $K' \subseteq K$ be the set of keys for which *q* is the current responsible of timestamping and *p* is the next responsible. The direct algorithm proceeds as follows. Once *q* reaches the end of its responsibility for the keys in *K'*, *e.g.* before leaving the system, it sends to *p* all its counters that have been initialized for the keys involved in *K'*. Let *C* be an empty set, *q* performs the following instructions at the end of its responsibility:

```
for each $c_{q,k} \in VCS_q$ do
    if ($k \in K'$) then
       C := C + {$c_{q,k}$};
Send C to p;
```

At the beginning of its responsibility for the keys in *K'*, *p* initializes its counters by performing the following instructions:

```
for each $c_{q,k} \in C$ do begin
   new($c_{p,k}$);
   $c_{p,k}$.value := $c_{q,k}$.value;
   $VCS_p$ := $VCS_p$ + {$c_{p,k}$};
end;
```

4.2.1.1 Application to CAN and Chord

The direct algorithm initializes the counters very efficiently, in O(1) messages, by sending the counters from the current responsible of timestamping to the next responsible at the end of its responsibility. But, how can the current responsible of timestamping find the address of the next responsible? The DHT's lookup service does not help here because it can only lookup the current responsible for *k*, *i.e.* *rsp(k,* h_{ts}*)*, and cannot return the address of the next responsible for *k*. To answer the question, we observe that, in DHTs, the next peer that obtains the responsibility for a key *k* is typically a neighbor of the current responsible for *k*, so the current responsible of timestamping has the address of the next one. We now illustrate this observation with CAN and Chord, two popular DHTs.

Let us assume that peer *q* is *rsp(k,h)* and peer *p* is *nrsp(k,h)* where $k \in K$ and $h \in H$. In CAN and Chord, there are only two ways by which *p* would obtain the responsibility for *k* wrt *h*. First, *q* leaves the P2P system or fails, so the responsibility of *k* wrt *h* is assigned to *p*. Second, *p* joins the P2P system which assigns it the responsibility for *k* wrt *h*, so *q* loses the responsibility for *k* wrt *h* despite its presence in the P2P system. We show that in both cases, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. In other words, we show that both CAN and Chord have the important property that *nrsp(k,h) is one of the neighbors of rsp(k,h) at the time when rsp(k,h) loses the responsibility for k wrt h*.

CAN. We show this property by giving a brief explanation of CAN's protocol for joining and leaving the system [19]. CAN maintains a virtual coordinate space partitioned among the peers. The partition which a peer owns is called its zone. According to CAN, a peer *p* is responsible for *k* wrt *h* if and only if *h(k)*, which is a point in the space, is in *p*'s zone. When a new peer, say *p*, wants to join CAN, it chooses a point *X* and sends a join request to the peer whose zone involves *X*. The current owner of the zone, say *q*, splits its zone in half and the new peer occupies one half, then *q* becomes one of *p*'s neighbors. Thus, in the case of join, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. Also, when a peer *p* leaves the system or fails, its zone will be occupied by one of its neighbors, *i.e.* the one that has the smallest zone. Thus, in the case of leave or fail, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)* , and that neighbor is known for *rsp(k,h)*.

Chord. In Chord [29], each peer has an m-bit identifier (ID). The peer IDs are ordered in a circle and the neighbors of a peer are the peers whose distance from *p* clockwise in the circle is 2^i for $0 \leq i \leq m$. The responsible for *k* wrt *h* is the first peer whose ID is equal or follows *h(k)*. Consider a new joining peer *p* with identifier ID_p. Suppose that the position of *p* in the circle is just between two peers q_1 and q_2 with identifiers ID_1 and ID_2, respectively. Without loss of generality, we assume that $ID_1 < ID_2$, thus we have $ID_1 < ID_p < ID_2$. Before the entrance of *p*, the peer q_2 was responsible for *k* wrt *h* if and only if $ID_1 < h(k) \leq ID_2$. When *p* joins Chord, it becomes responsible for *k* wrt *h* if and only if $ID_1 < h(k) \leq ID_p$. In other words, *p* becomes responsible for a part of the keys for which q_2 was responsible. Since the distance clockwise from *p* to q_2 is 2^0, q_2 is a neighbor of *p*. Thus, in the case of join, *nrsp(k,h)* is one of the neighbors of *rsp(k,h)*. When, a peer *p* leaves the system or fails, the next peer in the circle, say q_2, becomes responsible for its keys. Since the distance clockwise from *p* to q_2 is 2^0, q_2 is a neighbor of *p*.

Following the above discussion, when a peer *q* loses the responsibility for *k* wrt *h* in Chord or CAN, one of its neighbors, say *p*, is the next responsible for all keys for which *q* was responsible. Therefore, to apply the direct algorithm, it is sufficient that, before losing its responsibility, *q* sends to *p* its initialized counters, *i.e.* those involved in VCS_q.

4.2.2 Indirect Algorithm for Initializing Counters

With the direct algorithm, the initialization of counters can be done very efficiently. However, in some situations the direct algorithm cannot be used, *e.g.* when a responsible of timestamping fails. In those situations, we use the indirect algorithm. For initializing the counter of a key *k*, the indirect algorithm retrieves the most recent timestamp which is stored in the DHT along with the pairs (*k, data)*. As described in Section 3.2, peers store the timestamps, which are generated by KTS, along with their data in the DHT.

The indirect algorithm for initializing the counters proceeds as follows (see Figure 5). Let *k* be a key, *p* be the responsible of timestamping for *k*, and H_r be the set of replication hash functions

```
Indirect_Initialization(k, var $c_{p,k}$)
begin
   $ts_m$ := -1;
   for each  $h \in H_r$  do begin
      {data, ts} := DHT.$get_h$(k);
       if ($ts_m$ < ts) then
          $ts_m$ := ts;
   end;
   $c_{p,k}$.value := $ts_m$ + 1;
end;
```

Figure 5. Indirect algorithm for initializing counters

which are used for replicating the data in the DHT as described in Section 3.2. To initialize $c_{p,k}$, for each $h \in H_r$, p retrieves the replica (and its associated timestamp) which is stored at *rsp(k, h)*. Among the retrieved timestamps, p selects the most recent one, say ts_m, and initializes $c_{p,k}$ to $ts_m + 1$. If no replica and timestamp is stored in the DHT along with k, then p initializes $c_{p,k}$ to 0.

If p is at the beginning of its responsibility of timestamping for k, before using the indirect algorithm, it waits a while so that the possible timestamps, which are generated by the previous responsible of timestamping, be committed in the DHT by the peers that have requested them.

Let c_{ret} be the number of messages which should be sent over the network for retrieving a data from the DHT, the indirect algorithm is executed in $O(/H_r/*c_{ret})$ messages.

Let us now compute the probability that the indirect algorithm retrieves successfully the latest version of the timestamp from the DHT. We denote this probability as p_s. Let t be the time at which we execute the indirect algorithm, and p_t be the probability of currency and availability at t (see Section 3.3 for the definition of the probability of currency and availability). If at least one of the peers, which are responsible for k wrt replication hash functions, owns a current replica then the indirect algorithm works successfully. Thus, p_s can be computed as follows:

p_s *= 1 – (the probability that no current replica is available at peers which are responsible for k wrt replication hash functions)*

Thus, we have:

$$p_s = 1 - (1 - p_t)^{|H_r|}$$

In this equation, $|H_r|$ is the number of replication hash functions. By increasing the number of replication hash functions, we can obtain a good probability of success for the indirect algorithm. For instance, if the probability of currency and availability is about 30%, then by using 13 replication hash functions, p_s is more than 99%.

By adjusting the number of replication hash functions, the probability of success of the indirect algorithm is high but not 100%. Thus, there may be some situations where it cannot retrieve the latest version of timestamp, in which case the counter of the key is not initialized correctly. To deal with these situations in a correct way, we propose the following strategies:

- **Recovery**. After restarting, the failed responsible of timestamping contacts the new responsible of timestamping, say p, and sends it all its counters. Then, the new responsible of timestamping compares the received counters with those initialized by the indirect algorithm and corrects the counters which are initialized incorrectly (if any). In addition, if p has generated some timestamps with an incorrect counter, it retrieves the data which has been stored in the DHT with the latest value of the incorrect counter and reinserts the data into the DHT with the correct value of the counter.
- **Periodic inspection**. A responsible of timestamping which takes over a failed one, and which has not been contacted by it, periodically compares the value of its initialized counters with the timestamps which are stored in the DHT. If a counter is lower than the highest timestamp found, the responsible of timestamping corrects the counter. Furthermore, it reinserts the data which has been stored in the DHT with the latest value of the incorrect counter (if any).

4.3 Validity of Counters

In Section 4.1, the third rule for managing VCSs states that if a peer p loses the responsibility for a key k wrt h_{ts}, then p should remove $c_{p,k}$ from VCS_p (if it is there). We now discuss what p should do in order to respect the third rule for VCS_p. If the reason for losing responsibility is that p has left the P2P system or failed, then there is nothing to do, since when p rejoins the P2P system, it sets $VCS_p = \varnothing$. Therefore, we assume that p is present in the P2P system and loses the responsibility for k wrt h_{ts} because some other peer joins the P2P system and becomes responsible for k.

We can classify DHT protocols in two categories: *Responsibility Loss Aware (RLA)* and *Responsibility Loss Unaware (RLU)*. In an RLA DHT, a peer that loses responsibility for some key k wrt h and is still present in the P2P system detects its loss of responsibility. A DHT that is not RLA is RLU.

Most DHTs are RLA, because usually when a new peer p becomes *rsp(k, h)*, it contacts *prsp(k,h)*, say q, and asks q to return the pairs *(k, data)* which are stored at q. Thus, q detects the loss of responsibility for k. Furthermore, in most of DHTs, p is a new neighbor of q (see Section 4.2.1), so when p arrives q detects that it has lost the responsibility for some keys. For the DHTs that are RLA, the third rule of VCS can be enforced as follows. When a peer p detects that it has lost the responsibility for some keys wrt h_{ts}, it performs the following instructions:

```
For each $c_{p,k} \in VCS_p$ do
      If $p \neq$ rsp(k,$h_{ts}$) then
        remove $c_{p,k}$ from $VCS_p$
```

If the DHT is RLU, then Rule 3 can be violated. Let us illustrate with the following scenario. Let k be a key and p the peer that is *rsp(k,h_{ts})* which generates some timestamp for k, *i.e.* $c_{p,k}$ is in VCS_p. Suppose another peer q joins the P2P system, becomes *rsp(k, h_{ts})* and generates some timestamps for k. Then q leaves the DHT, and p becomes again *rsp(k,h_{ts})*. In this case, if p generates a timestamp for k using $c_{p,k} \in VCS_p$, the generated timestamp may be equal or less than the last generated timestamp for k, thus violating the monotonicity property as a result of violating Rule 3. To avoid such problems in a DHT that is RLU, we impose that *rsp(k,h_{ts})* assumes that after generating each timestamp for k, it loses its responsibility for k wrt h_{ts}. Thus, after generating a timestamp for k, it removes $c_{p,k}$ from VCS_p. Therefore, Rule 3 is enforced. However, by this strategy, for generating each timestamp for k we need to initialize $c_{p,k}$, and this increases the cost of timestamp generation.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our Update Management Service (UMS) through implementation and simulation. The implementation over a 64-node cluster was useful to validate our algorithm and calibrate our simulator. The simulation allows us to study scale up to high numbers of peers (up to 10,000 peers).

The rest of this section is organized as follows. In Section 5.1, we describe our experimental and simulation setup, and the algorithms used for comparison. In Section 5.2, we first report experimental results using the implementation of UMS and KTS on a 64-node cluster, and then we present simulation results on performance by increasing the number of peers up to *10,000*. In Sections 5.3, we evaluate the effect of the number of replicas, which we replicate for each data in the DHT, on performance. In Section 5.4, we study the effect of peers' failures on performance. In Section 5.5, we study the effect of the frequency of updates on performance.

5.1 Experimental and Simulation Setup

Our implementation is based on Chord [29] which is a simple and efficient DHT. Chord's lookup mechanism is provably robust in the face of frequent node fails, and it can answer queries even if the system is continuously changing. We implemented UMS and KTS as a service on top of Chord which we also implemented. In our implementation, the keys do not depend on the data values, so changing the value of a data does not change its key.

We tested our algorithms over a cluster of 64 nodes connected by a 1-Gbps network. Each node has 2 Intel Xeon 2.4 GHz processors, and runs the Linux operating system. We make each node act as a peer in the DHT.

To study the scalability of our algorithms far beyond 64 peers, we implemented a simulator using SimJava [27]. To simulate a peer, we use a SimJava entity that performs all tasks that must be done by a peer for executing the services KTS and UMS. We assign a delay to communication ports to simulate the delay for sending a message between two peers in a real P2P system. Overall, the simulation and experimental results were qualitatively similar. Thus, due to space limitations, for most of our tests, we only report simulation results.

The simulation parameters are shown in Table 1. We use parameter values which are typical of P2P systems [25]. The latency between any two peers is a normally distributed random number with a mean of 200 ms. The bandwidth between peers is also a random number with normal distribution with a mean of 56 (kbps). The simulator allows us to perform tests up to *10,000* peers, after which simulation data no longer fit in RAM and makes our tests difficult. Therefore, the number of peers is set to be *10,000*, unless otherwise specified.

In each experiment, peer departures are timed by a random Poisson process (as in [21]). The average rate, *i.e.* λ, for events of the Poisson process is λ=1/second. At each event, we select a peer to depart uniformly at random. Each time a peer goes away, another joins, thus keeping the total number of peers constant (as in [21]).

Peer departures are of two types: normal leave or fail. Let *failure rate* be a parameter that denotes the percentage of departures which are of fail type. When a departure event occurs, our simulator must decide on the type of this departure. For this, it generates a random number which is uniformly distributed in [0..100]; if the number is greater than *failure rate* then the peer departure is considered as a normal leave, else as a fail. In our tests, the default setting for *fail rate* is 5%.

In our experiments, each replicated data is updated by update operations which are timed by a random Poisson process. The default average rate for events of this Poisson process is λ=1/hour.

In our tests, unless otherwise specified, the number of replicas of each data is 10, *i.e.* $|H_r|=10$.

Table 1. Simulation parameters

Simulation parameter	Values
Bandwidth	Normally distributed random number, Mean = 56 Kbps, Variance = 32
Latency	Normally distributed random number, Mean = 200 ms, Variance = 100
Number of peers	10,000 peers
$|H_r|$	10
Peers' joins and departures	Timed by a random Poisson process with λ=1/second
Updates on each data	Timed by a random Poisson process with λ=1/hour
Failure rate	5% of departures

Although it cannot provide the same functionality as UMS, the closest prior work to UMS is the BRICKS project [13]. To assess the performance of UMS, we compare our algorithm with the BRICKS algorithm, which we denote as BRK. We tested two versions of UMS. The first one, denoted by UMS-Direct, is a version of UMS in which the KTS service uses the direct algorithm for initializing the counters. The second version, denoted by UMS-Indirect, uses a KTS service that initializes the counters by the indirect algorithm.

In our tests, we compare the performance of UMS-Direct, UMS-Indirect and BRK in terms of response time and communication cost. By response time, we mean the time to return a current replica in response to a query *Q* requesting the data associated with a key. The communication cost is the total number of messages needed to return a current replica in response to *Q*. For each experiment, we perform 30 tests by issuing *Q* at 30 different times which are uniformly distributed over the total experimental time, *e.g.* 3 hours, and we report the average of their results.

5.2 Scale up

In this section, we investigate the scalability of UMS. We use both our implementation and our simulator to study the response time and communication cost of UMS while varying the number of peers.

Using our implementation over the cluster, we ran experiments to study how response time increases with the addition of peers. Figure 6 shows the response time with the addition of peers until 64. The response time of all three algorithms grows logarithmically with the number of peers. However, the response time of UMS-Direct and UMS-Indirect is significantly better than BRK. The reason is that, by using KTS and determining the last generated timestamp, UMS can distinguish the currency of replicas and return the first current replica which it finds while BRK needs to retrieve all available replicas, which hurts response time. The response time of UMS-Direct is better than UMS-Indirect because, for determining the last timestamp, UMS-Direct uses a version of KTS that initializes the counters by the direct algorithm which is more efficient than the indirect algorithm used by UMS-Indirect. Note that the reported results are the average of the results of several tests done at uniformly random times.

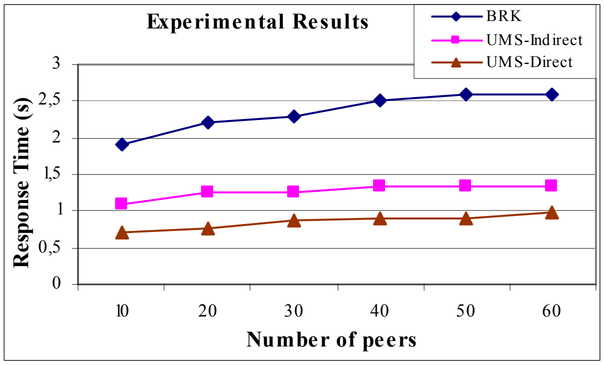


Figure 6. Response time vs. number of peers

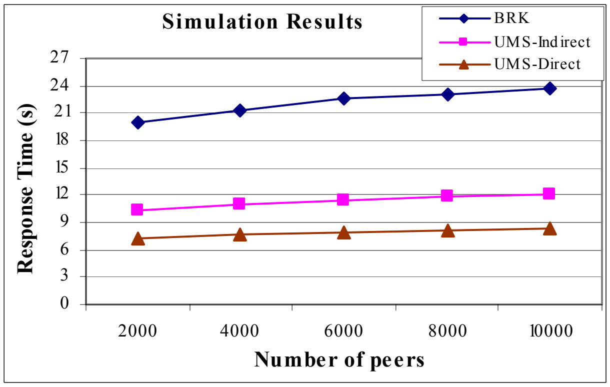


Figure 7. Response time vs. number of peers

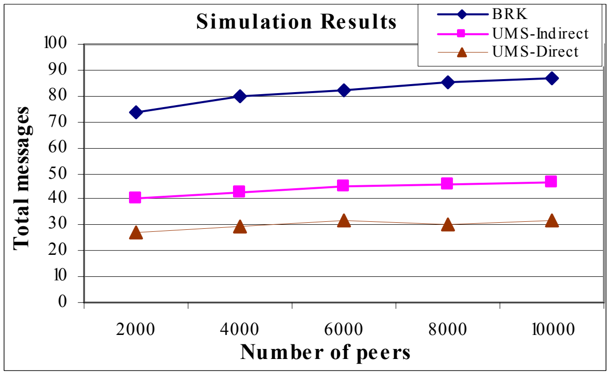


Figure 8. Communication cost vs. number of peers

Using simulation, Figure 7 shows the response time of the three algorithms with the number of peers increasing up to 10000 and the other simulation parameters set as in Table 1. Overall, the experimental results correspond qualitatively with the simulation results. However, we observed that the response time gained from our experiments over the cluster is slightly better than that of simulation for the same number of peers, simply because of faster communication in the cluster.

We also tested the communication cost of UMS. Using the simulator, Figure 8 depicts the total number of messages while increasing the number of peers up to 10,000 with the other simulation parameters set as in Table 1. The communication cost increases logarithmically with the number of peers.

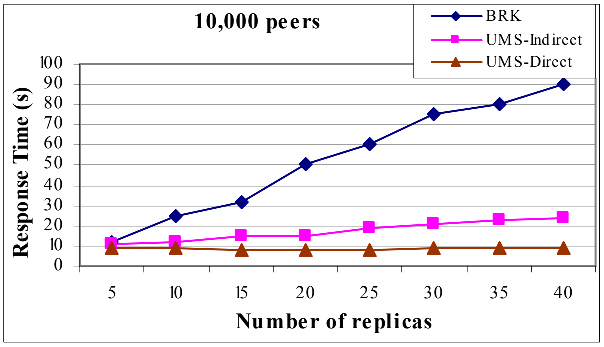


Figure 9. Response time vs. number of replicas

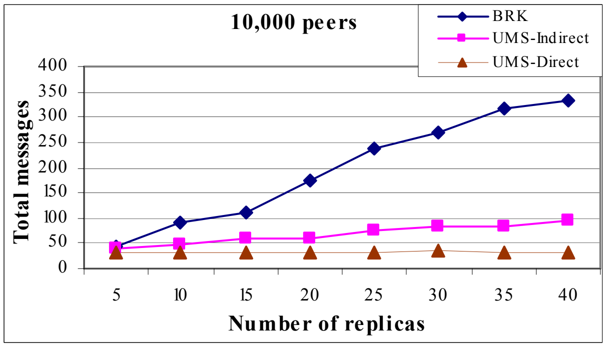


Figure 10. Communication cost vs. number of replicas

5.3 Effect of the Number of Replicas

In this section, we study the effect of the number of replicas, which we replicate for each data in the DHT, on the performance of MUS.

Using the simulator, Figures 9 and 10 show how respectively response time and communication cost evolve while increasing the number of replicas, with the other simulation parameters set as in Table 1. The number of replicas has a strong impact on the performance of BRK, but no impact on UMS-Direct. It has a little impact on the performance of UMS-Indirect because, in the cases where the counter of a key is not initialized, UMS-Indirect must retrieve all replicas from the DHT.

5.4 Effect of Failures

In this section, we investigate the effect of failures on the response time of UMS. In the previous tests, the value of failure rate was 5%. In this section, we vary the value of fail rate and investigate its effect on response time.

Figure 11 shows how response time evolves when increasing the fail rate, with the other parameters set as in Table 1. An increase in failure rate decreases the performance of Chord's lookup service, so the response time of all three algorithms increases. For the cases where the failure rate is high, *e.g.* more than 80%, the response time of UMS-Direct is almost the same as UMS-Indirect. The reason is that if a responsible of timestamping fails, both UMS-Direct and UMS-Indirect need to use the indirect algorithm for initializing the counters at the next responsible of timestamping, thus their response time is the same.

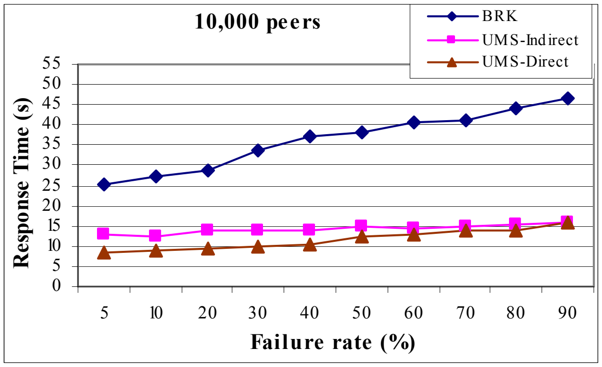


Figure 11. Response time vs. failure rate

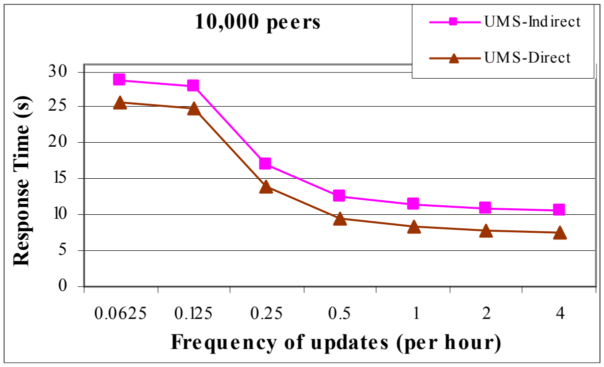


Figure 12. Response time vs. frequency of updates

5.5 Effect of Update Frequency

In this section, we study the effect of the frequency of updates on the performance of UMS. In the previous experiments, updates on each data were timed by a Poisson process with an average rate of 1/hour. In this section, we vary the average rate (*i.e.* frequency of updates) and investigate its effect on response time.

Using our simulator, Figures 12 shows how response time evolves while increasing the frequency of updates with the other simulation parameters set as in Table 1. The response time decreases by increasing the frequency of updates. The reason is that an increase in the frequency of updates decreases the distance between the time of the latest update and the retrieval time, and this increases the probability of currency and availability, so the number of replicas which UMS retrieves for finding a current replica decreases.

6. RELATED WORK

In the context of distributed systems, data replication has been widely studied to improve both performance and availability. Many solutions have been proposed in the context of distributed database systems for managing replica consistency [17], in particular, using eager or lazy (multi-master) replication techniques. However, these techniques either do not scale up to large numbers of peers or raise open problems, such as replica reconciliation, to deal with the open and dynamic nature of P2P systems.

Data currency in replicated databases has also been widely studied, *e.g.* [1], [10], [11], [14], [22] and [26]. However, the main objective is to trade currency and consistency for performance while controlling the level of currency or consistency desired by the user. Our objective in this paper is different, *i.e.* return the current (most recent) replica as a result of a get request.

Most existing P2P systems support data replication, but without consistency guarantees. For instance, Gnutella [9] and KaZaA [12], two of the most popular P2P file sharing systems allow files to be replicated. However, a file update is not propagated to the other replicas. As a result, multiple inconsistent replicas under the same identifier (filename) may co-exist and it depends on the peer that a user contacts whether a current replica is accessed.

PGrid is a structured P2P system that deals with the problem of updates based on a rumor-spreading algorithm [7]. It provides a fully decentralized update scheme, which offers probabilistic guaranties rather than ensuring strict consistency. However, replicas may get inconsistent, *e.g.* as a result of concurrent updates, and it is up to the users to cope with the problem.

The Freenet P2P system [3] uses a heuristic strategy to route updates to replicas, but does not guarantee data consistency. In Freenet, the query answers are replicated along the path between the peers owning the data and the query originator. In the case of an update (which can only be done by the data's owner), it is routed to the peers having a replica. However, there is no guarantee that all those peers receive the update, in particular those that are absent at update time.

Many of existing DHT applications such as CFS [4], Past [24] and OceanStore [20] exploit data replication for solving the problem of hot spots and also improving data availability. However, they generally avoid the consistency problem by restricting their focus on read-only (immutable) data.

The BRICKS project [13] deals somehow with data currency by considering the currency of replicas in the query results. For replicating a data, BRICKS stores the data in the DHT using multiple keys, which are correlated to the key *k* by which the user wants to store the data. There is a function that, given *k*, determines its correlated keys. To deal with the currency of replicas, BRICKS uses versioning. Each replica has a version number which is increased after each update. However, because of concurrent updates, it may happen that two different replicas have the same version number thus making it impossible to decide which one is the current replica. In addition, to return a current replica, all replicas need be retrieved in order to select the latest version. In our solution, concurrent updates raise no problem, *i.e.* this is a consequence of the monotonicity property of timestamps which are generated by KTS. In addition, our solution does not need to retrieve all replicas, and thus is much more efficient.

7. CONCLUSION

To ensure high data availability, DHTs typically rely on data replication, yet without currency guarantees for updateable data. In this paper, we proposed a complete solution to the problem of data availability and currency in replicated DHTs. Our main contributions are the following.

First, we proposed a new service called Update Management Service (UMS) which provides efficient retrieval of current replicas. For update operations, the algorithms of UMS rely on timestamping. UMS supports concurrent updates. Furthermore, it has the ability to determine whether a replica is current or not without comparing it with other replicas. Thus, unlike the solution in [13], our solution does not need to retrieve all replicas for finding a current replica, and is much more efficient.

Second, we gave a probabilistic analysis of UMS's communication cost by computing the expected number of replicas which UMS must retrieve. We proved that this number is less than the inverse of the probability of currency and availability. Thus, except for the cases where the availability of current replicas is very low, the expected number of retrieved replicas is typically small, *e.g.* if at least 35% of replicas are current and available then this number is less than 3.

Third, we proposed a Key-based Timestamping Service (KTS) which generates monotonically increasing timestamps in a completely distributed fashion, using local counters. The dynamic behavior of peers causes no problem for KTS. To preserve timestamp monotonicity, we proposed a direct and an indirect algorithm. The direct algorithm deals with the situations where peers leave the system normally, *i.e.* without failing. The indirect algorithm takes into account the situations where peers fail. Although the indirect algorithm has high probability of success in general, there are rare situations where it may not be successful at finding the current replica. We proposed two strategies to deal with these situations.

Fourth, we validated our solution through implementation and experimentation over a 64-node cluster and evaluated its scalability through simulation over 10,000 peers using SimJava. We compared the performance of UMS and BRK (from the BRICK project) which we used as baseline algorithm. The experimental and simulation results show that using KTS, UMS achieves major performance gains, in terms of response time and communication cost, compared with BRK. The response time and communication cost of UMS grow logarithmically with the number of peers of the DHT. Increasing the number of replicas, which we replicate for each data in the DHT, increases very slightly the response time and communication cost of our algorithm. In addition, even with a high number of peer fails, UMS still works well. In summary, this demonstrates that data currency, a very important requirement for many applications, can now be efficiently supported in replicated DHTs.

REFERENCES

[1] Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R., and Tamma, P. Relaxed-currency serializability for middle-tier caching and replication. *Proc. of SIGMOD*, 2006.

[2] Bromwich, T.J.I. *An Introduction to the Theory of Infinite Series*. 3rd edition, Chelsea Pub. Co., 1991.

[3] Clarke, I., Miller, S.G., Hong, T.W., Sandberg, O., and Wiley, B. Protecting Free Expression Online with Freenet. *IEEE Internet Computing 6(1)*, 2002.

[4] Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., and Stoica, I. Wide-Area Cooperative Storage with CFS. *Proc. of ACM Symp. on Operating Systems Principles*, 2001.

[5] Dabek, F., Zhao, B.Y., Druschel, P., Kubiatowicz, J., and Stoica, I. Towards a Common API for Structured Peer-to-Peer Overlays. *Proc. of Int. Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[6] Daswani, N., Garcia-Molina, H., and Yang, B. Open Problems in Data-Sharing Peer-to-Peer Systems. *Proc. of ICDT*, 2003.

[7] Datta A., Hauswirth M., and Aberer, K. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. *Proc. of IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, 2003.

[8] Elson, J., Girod, L., and Estrin, D. Fine-grained Network Time Synchronization Using Reference Broadcasts. *SIGOPS Operating Systems Review, 36(SI)*, 2002.

[9] Gnutella. http://www.gnutelliums.com/.

[10] Guo, H., Larson, P.Å., Ramakrishnan, R., and Goldstein, J. Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. *Proc of SIGMOD*, 2004.

[11] Guo, H., Larson, P.Å., and Ramakrishnan, R. Caching with 'Good Enough' Currency, Consistency, and Completeness. *Proc. of VLDB*, 2005.

[12] Kazaa. http://www.kazaa.com/.

[13] Knezevic, P., Wombacher, A., and Risse, T. Enabling High Data Availability in a DHT. *Proc. of Int. Workshop on Grid and P2P Computing Impacts on Large Scale Heterogeneous Distributed Database Systems (GLOBE)*, 2005.

[14] Lindstrom, G., and Hunt, F. Consistency and Currency in Functional Databases. *Proc. of INFOCOM*, 1983.

[15] Luby, M. *Pseudorandomness and Cryptographic Applications*. Princeton University Press, 1996.

[16] Mills., D.L. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications, 39(10)*, 1991.

[17] Özsu, T., and Valduriez, P. *Principles of Distributed Database Systems*. 2nd Edition, Prentice Hall, 1999.

[18] PalChaudhuri, S., Saha, A.K., and Johnson, D.B. Adaptive Clock Synchronization in Sensor Networks. *Proc. of Int. Symp. on Information Processing in Sensor Networks*, 2004.

[19] Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., and Shenker, S. A Scalable Content-Addressable Network. *Proc. of SIGCOMM*, 2001.

[20] Rhea, S.C., Eaton, P.R., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. Pond: the OceanStore Prototype. *Proc. of USENIX Conf. on File and Storage Technologies (FAST)*, 2003.

[21] Rhea, S.C., Geels, D., Roscoe, T., Kubiatowicz, J. Handling Churn in a DHT. *Proc. of USENIX Annual Technical Conf.*, 2004.

[22] Röhm, U., Böhm, K., Schek, H., and Schuldt,H. FAS: a Freshness- Sensitive Coordination Middleware for a Cluster of OLAP Components. *Proc. of VLDB*, 2002.

[23] Rowstron, A. I.T., and Druschel, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Proc. of Middleware*, 2001.

[24] Rowstron, A., and Druschel, P. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. *Proc. of ACM Symp. on Operating Systems Principles*, 2001.

[25] Saroiu, S., Gummadi, P.K., and Gribble, S.D. A Measurement Study of Peer-to-Peer File Sharing Systems. *Proc. of Multimedia Computing and Networking (MMCN)*, 2002.

[26] Segev, A., and Fang, W. Currency-based Updates To Distributed Materialized Views. *Proc. of ICDE*, 1990.

[27] Simjava. http://www.dcs.ed.ac.uk/home/hase/simjava/

[28] Spivak, M. *Calculus*. 3rd edition, Cambridge University Press, 1994.

[29] Stoica I., Morris, R., Karger, D.R., Kaashoek, M.F., and Balakrishnan, H. Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications. *Proc. of SIGCOMM*, 2001.