Haute Disponibilité de la Réplication Optimiste dans les Systèmes DHT

Manal El Dick

9 juin 2006

encadrée par Esther Pacitti et Patrick Valduriez

MASTER RECHERCHE

Architectures Logicielles Distribuées

Ecole Polytechnique de l'Université de Nantes

Laboratoire d'informatique de Nantes Atlantique
2, rue de la Houssinière
B.P.92208 - F 44322 NANTES CEDEX 3

Rapport Bibliographique

Table des matières

I Introduction 4

II Réplication optimiste 6

1 Introduction 6

2 Concepts de base 6
2.1 Mono-maître x multi-maître 6
2.2 Réplication totale x partielle 7
2.3 Réplication synchrone x asynchrone 8

3 Caractéristiques de la réplication optimiste 12
3.1 Cohérence éventuelle 12
3.2 Journalisation 12
3.3 Réconciliation 13
3.3.1 Réconciliation sémantique 13
3.3.2 Estampillage 15
3.4 Diffusion des mises à jour 20
3.4.1 Topologie de communication imposée 20
3.4.2 Protocoles épidémiques 21
3.5 Comparaison entre diverses approches 22

III Réplication dans les systèmes P2P 24

4 Introduction 24

5 Architecture des systèmes P2P 24
5.1 P2P non structuré 24
5.2 Super-pair 25
5.3 P2P structuré 25

6 Solutions de réplication en P2P 28
6.1 Napster, Gnutella 28
6.2 Freenet 28
6.3 OceanStore 28
6.4 P-Grid 29
6.5 UPTReC 30
6.6 Comparaison entre diverses approches 32

IV Réplication optimiste dans APPA 33

7 Introduction 33

8 Architecture de APPA 33

9 Approche de réconciliation **34**
9.1 Algorithme DSR 34
9.2 DSR sur Chord 36
9.3 Résultats expérimentaux 37

V Conclusion et Perspective **39**

Bibliographie **42**

Première partie

Introduction

Le peer-to-peer (P2P), qu'on peut traduire par "égal à égal" ou pair à pair, est un résau permettant à des internautes de communiquer et partager toute sorte de données stockées et géographiquement dispersées. Les technologies P2P se sont d'ailleurs montrées si efficaces que le P2P est considéré par certains comme "l'étape ultime de la liberté et de la démocratie sur Internet". Sans aller jusque-là, on considère souvent que le P2P repose sur une philosophie de partage et un profond esprit communautaire. De plus, il a fait sauter une restriction importante des machines centralisées : avant lui, plus un contenu devenait populaire, moins le serveur en devenait accessible pour des raisons de saturation. Avec le P2P, l'augmentation du succès d'un contenu se traduit par une augmentation de facilité de son chargement.
L'ancêtre du P2P est Napster qui proposait un échange de fichiers musicaux entre internautes. Le réseau, qui utilisait un serveur central, a été démantelé par décision judiciaire. En 2005, toute personne dotée d'une connexion suffisante peut télécharger des médias depuis Internet. Les principales maisons de productions et les distributeurs ne s'étaient pas préparés à un accès si facile aux fichiers et aux contenus. Ils ont désormais du mal à faire valoir leurs droits et se plaignent d'un "piratage via Internet". Aujourd'hui, les principales limitations des réseaux P2P sont légales : ils subissent les pressions de l'industrie de la musique, qui lutte contre ces systèmes permettant de se soustraire aux droits d'auteurs. Depuis quelques années, une nouvelle génération de P2P suscite l'intérêt des internautes : les P2P cryptés comme Freenet qui garantissentt aux utilisateurs une confidentialité dans leurs échanges.

Néanmoins, les systèmes P2P offrent d'autres perspectives légales et avantageuses. Techniquement, ils sont des systèmes distribués qui s'auto-organisent et se développent d'autant plus que le nombre de leurs participants augmente : un site y contribue en disposant ses propres ressources (données, calcul, processeur, stockage...) au bénéfice des autres. Les participants collaborent ensemble en fournissant et recevant des données les uns des autres s'opposant ainsi aux architectures client-serveur qui impliquent des sites ayant des fonctionnalités déterminées. Les pairs sont dynamiques et autonomes pouvant se déconnecter et se reconnecter à leur guise et à n'importe quel moment.
Dernièrement, ces systèmes deviennent de plus en plus populaires puisqu'ils offrent un grand nombre de caractéristiques convoitées par les applications sur le web : passage à l'échelle, disponibilité, administration décentralisée, tolérance aux fautes, etc. Or, la disponibilité des processus et des machines participant à un système P2P est imprévisible. Les services P2P ne peuvent donc pas garantir l'accès à des ressources individuelles. Par contre, en exploitant la réplication de ces ressources, ils peuvent garantir l'accès à leurs copies et résister ainsi à leur volatilité.
L'idée générale consiste à répliquer les données au niveau de plusieurs pairs afin de permettre l'accès de différents utilisateurs en même temps. Cependant, il est nécessaire de pouvoir localiser une donnée sinon comment l'accéder. Donc, pour améliorer les performances des requêtes et permettre le passage à l'échelle, il faut

trouver une solution de réplication et de recherche d'information qui s'adapte à un système vaste, ouvert, donc vulnérable aux pannes et au dynamisme des pairs en prenant compte qu'aucune forme de connaissance globale et centralisée sur le système et ses ressources n'existe.
Cependant, la plupart des systèmes P2P existants supposent que les données sont statiques et les mises à jour rares et n'intègrent aucun mécanisme de gestion de données répliquées en présence de ces mutations. Mais, pour les applications au-delà du simple partage de fichiers exploitant des données riches en sémantiques, ces suppositions ne tiennent point, les mises à jour étant assez fréquentes. Plusieurs applications impliquent que des nouvelles données soient ajoutées, effacées, ou mises à jour fréquemment par plusieurs utilisateurs comme les forums de discussion, les calendriers partagés, les catalogues e-commerce. La recherche doit s'orienter vers une solution de réplication convenableà ces applications avancées des systèmes P2P, ce qui est d'ailleurs le but de l'état de l'art que nous dressons ci-dessous. On est particulièrement intéressé par la réplication asynchrone qui s'adapte aux environnements ponctuellement connéctés. Une approche asynchrone dite optimiste assure la grande disponibilité requise dans les systèmes P2P et l'étude se concentrera sur cette approche

Dans ce qui suit, nous commençons par présenter les concepts de base et les variantes de la réplication et par exposer diverses approches adoptées dans des applications collaboratives asynchrones présentant des similarités avec les systèmes P2P (partie 2). Ensuite, nous parlons des systèmes P2P, et du travail concernant la réplication effectué dans ce contexte (partie 3). Nous présentons le système APPA, un système de gestion de données P2P ayant une architecture indépendante du réseau, et nous exposons sa solution de réplication basée sur un algorithme appelé DSR (partie 4). Enfin, nous concluons et partant d'une solution adaptant le DSR à un réseau DHT, nous posons la possibilité d'extension à un réseau DHT location-aware (partie 5).

Deuxième partie
Réplication optimiste

1 Introduction

La réplication consiste à maintenir plusieurs copies de données,les réplicas, sur des machines séparées. Elle améliore la disponibilité en permettant l'accès aux données même si certains réplicas sont indisponibles. De plus, elle améliore la performance : en permettant aux utilisateurs d'accéder aux plus proches réplicas, la latence d'accès est réduite ; et en parallélisant et/ou répartissant la charge entre diverses machines, le débit est augmenté. Cependant, le prix à payer est d'assurer la cohérence entre les réplicas, en présence des mises à jour (dites aussi écritures) effectuées par les utilisateurs sur leurs réplicas locaux.
D'autre part, le développement des réseaux encourage diverses formes de collaboration entre les utilisateurs ce qui résulte en la formation de groupes de personnes partageant un intérêt et par suite des ressources communes. On appelle ces groupes les communautés (professionnelles ou non). Avec l'avènement d'Internet et l'émergence des communautés , tous les domaines applicatifs devront intégrer la dimension collaborativeà travers la réplication des données.
La réplication optimiste est largement utilisée dans les environnements collaboratifs ponctuellement connectés comme dans [1, 2, 3]. Elle permet l'accès concurrent aux données partagées tout en fournissant les garanties de cohérence adapté à l'application. Elle assure la mise à jour asynchrone des réplicas et par suite la progression des applications en dépit de quelques sites déconnectés.

Dans cette partie, nous introduisons les concepts de base de la réplication pour ensuite nous concentrer sur la réplication optimiste. Nous parlons de ses charactéristiques : la cohérence, la journalisation, la réconciliation et enfin la diffusion des mises à jour. Nous exposons aussi plusieurs solutions exploitant la réplication optimiste

2 Concepts de base

2.1 Mono-maître x multi-maître

Un *réplica* est une copie d'une collection d'objets (copie d'une table relationnelle, d'un document XML, etc.) stockée sur un site. Un élément de réplica est un objet appartenant à un réplica (un tuple d'une table relationnelle,un élément d'un document XML, etc.).
Les opérations pouvant etre effectuées sur un réplica sont les écritures et les lectures. Suivant le type des opérations permises sur une copie donnée, on distingue entre une *copie primaire* et une *copie secondaire* :

- **Copie primaire** : copie permettant les écritures et les lectures, stockée par un site *maître*. Elle est représentée par une lettre majuscule ;
- **Copie secondaire** :(dite *read-only copy*) copie permettant uniquement les lectures, stockée par un site *subordonné*. Elle est représentée par une lettre minuscule ;

Réplication mono-maître

Un seul site stocke la copie primaire alors que les autres, se contentent de copies secondaires (cf. figure 1). Dans ce cas, le site maître est le seul à pouvoir effectuer des mises à jour et à les propager ensuite aux subordonnés ; ce qui limite la disponibilité et favorise le blocage en cas de panne du site maître. Cette approche s'avère par contre simple à réaliser et à administrer.

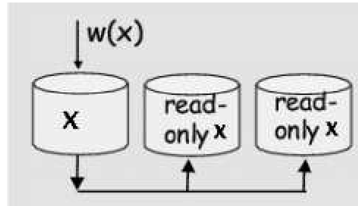

FIG. 1 – Un exemple de réplication mono-maître avec une copie primaire et deux secondaires

Réplication multi-maître

Plusieurs sites stockent des copies primaires d'un même objet (cf. figure 2). Dans ce cas, plusieurs mises à jour peuvent être soumises en même temps et indépendamment ; ce qui assure une plus grande disponibilté mais aussi une plus grande charge sur le réseau.

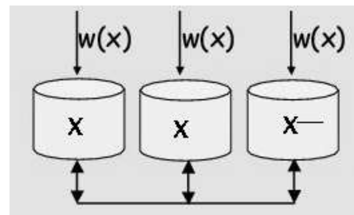

FIG. 2 – Un exemple de réplication multi-maître avec trois copies primaires

2.2 Réplication totale x partielle

Réplication totale

La réplication totale consiste à répliquer tous les objets partagés et à les stocker chez tous les sites en jeu (cf. figure 3). L'équilibrage de charge est simple puisque tous les sites ont les mêmes capacités et la disponibilité est maximale puisque tout site peut remplacer un autre en cas de panne.

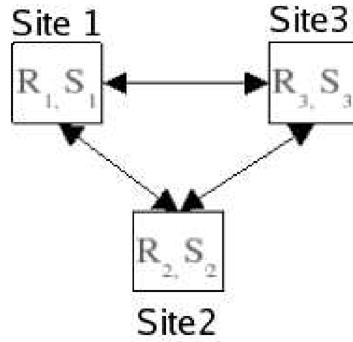

FIG. 3 – Un exemple de réplication totale avec deux objets R et S

Réplication partielle

La réplication partielle [4] consiste à répliquer pariellement les objets partagés : les sites peuvent ne pas stocker les mêmes réplicas (cf. figure 4). Son utilisation dans le cas de traffic volumineux d'information peut optimiser les échanges et réduire la consommation de bande passante et d'espace de stockage. Les mises à jour seront propagées uniquement aux sites affectés (stockant une copie primaire ou secondaire de l'objet en question). Le réseau ainsi que les sites seront moins chargés. Cependant, certaines données peuvent être liées ou se référencer ce qui peut créer des complications au niveau d'un site répliquant partiellement ces données. Les mécanismes de propagation doivent donc tenir compte du placement des données et/ou intégrer des systèmes de filtrage des mises à jour à propager.

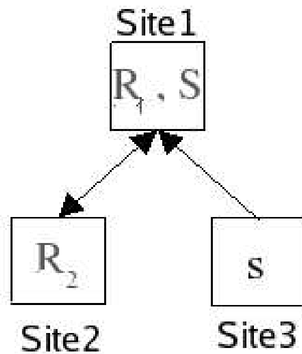

FIG. 4 – Un exemple de réplication partielle avec deux objets R et S

2.3 Réplication synchrone x asynchrone

Dans cette sous-section, nous mettons en évidence l'incompatibilité de la réplication synchrone avec les environnements collaboratifs asynchrones et par-

ticulièrement les P2P. Ensuite, nous présentons l'approche asynchrone.

Réplication synchrone

Dans la réplication synchrone, dite aussi *eager replication*, lorsqu'une mise à jour est soumise à un réplica donné, elle est propagée à tous les autres au sein d'une transaction atomique avant de valider la requête initiale et de retourner la réponse à l'utilisateur (cf. figure 5). Cette approche assure la cohérence mutuelle et son implémentation est réalisée par une phase de verrouillage des données (2PL, RAWA ou ROWA) et une autre de validation (2PC) [5].
Chaque transaction verrouille tout objet utilisé et ne relâche les verrous qu'à la

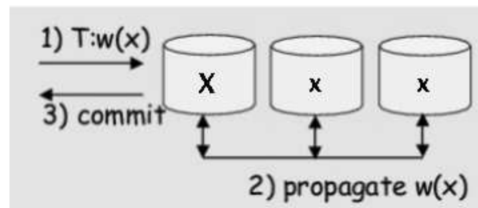

FIG. 5 – Principe général d'une écriture avec la réplication synchrone

validation. Le protocole RAWA, *read-all-write-all*, exige un verrou exclusif sur un objet en lecture ou en écriture : si une copie d'un objet est mise à jour ou simplement lue, aucune autre copie ne peut être accédée. Le protocole ROWA, *read-one-write-all*, une optimisation du précédent, exige un verrou exclusif sur un objet en écriture et un verrou partagé sur un objet en lecture permettant ainsi plusieurs copies d'un même objet d'être lues en parallèle. Cela exige des utilisateurs d'identifier à l'avance toutes les données qu'ils pourront utiliser éventuellement, afin de réserver leurs verrous [1].

La validation en deux étapes (2PC) implique un coordinateur ou gérant global de la transaction qui reçoit la requête et des participants ou gérants locaux. A la fin de chaque transaction, un coordinateur prépare les participants en vue d'une validation atomique et veille à l'application de la décision globale (validation ou annulation) par tous. Ces protocoles assurent une forte cohérence à tout moment et sur n'importe quel réplica et nécessite beaucoup d'échanges de messages pour maintenir chaque donnée à la même valeur sur tous les réplicas. L'approche synchrone est adaptée aux réseaux fiables où les latences sont minimes et les pannes rares ce qui n'est ni le cas de l'internet ni celui des environnements ponctuellement connectés puisqu'un site tentant de se synchroniser avec un autre non disponible restera bloqué indéfiniment et pourra bloquer les autres. Elle ne passe pas à l'échelle puisque les performances (temps de réponse et coût de communication) se dégradent avec l'augmentation du nombre des réplicas et elle ne supporte ni l'hétérogénéité ni l'autonomie puisqu'elle impose l'intégration de ses protocoles aux sites collaborants [4].

Réplication asynchrone

La réplication asynchrone, dite aussi *lazy replication*, privilégie l'efficience sur la cohérence et peut ne pas garantir une forte cohérence mais par contre est

facilement déployée puisqu'elle évite ou affaiblit les contraintes de la réplication synchrone. Lorsqu'une mise à jour est soumise à un réplica donné, elle est validée localement puis elle est propagée aux autres dans une transaction séparée (cf. figure 6). Cette approche peut passer à l'échelle et elle est beaucoup plus performante en terme de réduction de messages échangés. Elle ne bloque pas en cas de pannes et offre en plus une flexibilité au système qui peut choisir le moment convenable pour propager les mises à jour.

On peut définir un schéma de réplication caractérisé par quatre paramètres

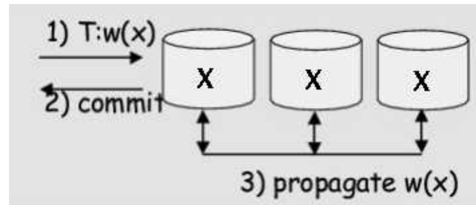

FIG. 6 – Principe général d'une écriture avec la réplication asynchrone

[6] : propriété, propagation, rafraîchissement et configuration. Le paramètre *propriété* définit la permsission de mettre à jour les copies primaire ou secondaire). Le paramètre *propagation* définit quand les mises à jour sont propagées aux réplicas concernés : en considérant une transaction comme une suite d'écritures ou de mises à jour, chacune de ces mises à jour ne peut être propagée qu'après la validation totale de la transaction (*deferred propagation*), ou directement et indépendamment des autres (*immediate propagation*). Le paramètre *rafraîssement* définit la gestion des mises à jour, une fois propagées et reçues par les sites. Le paramètre *configuration* définit les sites composants (multi-maître ou mono-maître).

On distingue deux approches asynchrones : **pessimiste** et **optimiste**. Des deux, c'est la réplication pessimiste qui asure la plus forte cohérence en évitant les conflits : résolution a priori donc plus besoin de détection a postériori. Plusieurs solutions ont exploité l'approche pessimiste. L'algorithme proposé en [7] fournit une forte cohérence pour une réplication partielle multi-maître. Cependant, il exige que les transactions mettent à jour une copie primaire fixe : chaque type de transaction est associé à un site et lui seul peut exécuter une transaction de ce type et ensuite diffuser uniquement les résultats aux sites concernés. Cela réduit l'overhead d'exécution mais pose un problème de disponibilité aux applications présentant des mises à jour intensives.
Au lieu d'utiliser la diffusion atomique, [4] a recours au service de réseau fiable FIFO (multicast). Chaque transaction est soumise au site approprié selon un équilibrage de charge. Une valeur chronologique lui est associée et puis elle est envoyée en mode multicast aux sites imliqués où un temps de délai est introduit avant son exécution pour garantir la réception et l'exécution de toute transaction ayant été soumise avant elle. Ainsi, est assuré l'ordonnancement chronologique des mises à jour qui attendent leur tour dans des queues au niveau de chaque site. Des optimisations réalisées permettent l'exécution de mises à jour concurrentes mais non conflictuelles et améliorent ainsi la performance en terme de débit. Néanmoins, cette approche requiert un support réseau garantissant un temps maximal pour la réception des messages comme c'est le cas des clusters.

De plus, certaines applications collaboratives donnent priorité à l'accessibilité en écriture au risque de déclencher des conflits ; ce qui contradit les principes de l'approche pessimiste.

D'autre part, la réplication optimiste [8] accepte a priori toutes les écritures en repoussant le contrôle de concurrence a posteriori grâce à la supposition "optimiste" qu'on rencontrera rarement des conflits. Les écritures sont intialement instables puisqu'elles ne sont pas encore validées : elles sont alors considérées comme *tentatives* et peuvent être enregistrées dans les journaux locaux des sites qui les performent. Les utilisateurs ont droit à exécuter des actions tentatives dont deux ou plusieurs peuvent entrer en conflit lorsqu'elles mettent à jour le même objet en parallèle. La figure 7 illustre un conflit entre deux actions soumises à deux sites A et B et mettant à jour le même tuple d'une table relationnelle R.
Des mécanismes de détection et de résolution de conflits doivent être intégrés au système et certaines actions tentatives peuvent être annulées (*undo*) ou refaites (*redo*) en se basant sur le journal. Les utilisateurs peuvent alors lire et écrire des données incohérentes en attendant que les conflits soient résolus. Cette approche permet la progression des applications au sein d'un réseau non fiable et des sites souvent inaccessibles. Un utilisateur peut tranquillement travailler sur son réplica local en déconnecté, en attendant pour se synchroniser qu'un autre réplica se connecte ou que des liens de communication se rétablissent. Les sites et les utilisateurs préservent leur autonomie sans se soucier de coordination explicite avec les autres sites. Les propagations des mises à jour s'effectuent dans l'arrière plan et s'adaptent à la connexion existante. Cependant, la solution optimiste n'est applicable que pour les applications tolérant des conflits occasionels et des données divergentes.

La suite de l'étude se concentre sur la réplication optimiste.

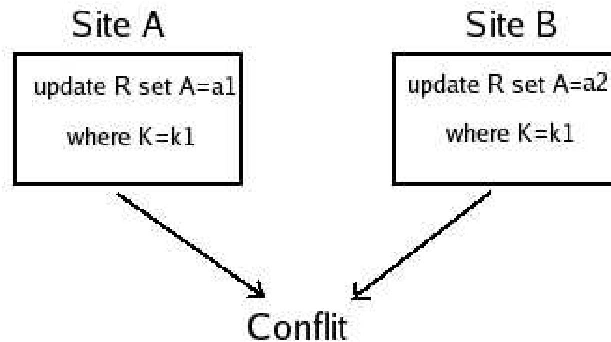

FIG. 7 – Un exemple de conflit

3 Caractéristiques de la réplication optimiste

3.1 Cohérence éventuelle

La réplication optimiste implique des divergences entre les réplicas. Le protocole de réplication doit néanmoins offrir une forme de cohérence.
La cohérence éventuelle se contente de garantir que les état des différents réplicas d'un objet vont éventuellement converger. En attendant, l'application peut observer des états incohérents. Quand les utilisateurs cesseront de soumettre des actions, les réplicas pourront atteindre une valeur finale identique. Ce cas est rarement atteint et le système est en perpétuelle instabilité. C'est d'ailleurs le prix à payer si l'on veut autoriser des écritures concurrentes en déconnecté.
Assurer cette cohérence nécessite plusieurs traitements : propager toutes les mises à jour à tous les réplicas, établir un modéle d'ordonnancement global des actions de tous les sites, identifier et résoudre les conflits, et enfin valider ces actions.
En [9], les auteurs proposent un formalisme pour la cohérence et identifient des propriétés que tout algorithme doit satisfaire pour assurer la cohérence :

- **Propagation éventuelle** : toute action soumise est éventuellemnt connue par tous les sites. Cela s'applique aussi aux conditions associées aux actions et appelées *contraintes* ;
- **Décision éventuelle** : une décision est prise pour toute action soumise. Elle est soit validée soit annulée et son effet ne peut plus être modifié. Elle devient alors *stable* ;
- **Mergeability** : c'est possible de trouver une résolution pour n'importe quel ensemble d'actions, à partir des journaux de différents sites, en respectant les contraintes. Cette propriété suggère par besoin de sûreté que tous les sites adoptent une stratégie détérministe de décision ; Par exemple un simple protocole basé sur l'estampillage peut garantir cette propriété en assurant que tous les sites ordonnent uniformément leurs actions suivant une estampille globale ;
- **Décisions sûres** : les sites ne prennent pas des décisions conflictuelles ;

Un *schedule* est défini comme étant une séquence ordonnée d'actions. Chaque site peut produire un schedule à partir des actions soumises et reçues. La propriété de cohérence éventuelle assure qu'éventuellemnet les sites appliqueront le même schedule ou des schedules *équivalents* : ils exécutent les mêmes actions et les paires non commutatives sont exécutées dans le même ordre. Partant d'un même état initial, les schedules équivalents produisent un même état final. A chaque moment, pour chaque réplica, on a un préfixe d'un schedule équivalent à un préfixe du schedule de tout autre réplica. Ce préfixe est formé d'actions stables et il augmente monotoniquement avec le temps.

3.2 Journalisation

Lors des échanges entre les sites, on doit pouvoir déterminer les nouveautés pa rapport à chacun ; les données crées, modifiées et effacées. Les écritures parallèles ou concurrentes sur les réplicas peuvent aussi engendrer des conflits, comme quand deux utilisateurs modifient en même temps le même tuple d'une table. Pour détecter les divergences, l'état courant des données ne suffit pas : par exemple si on ne retrouve pas une donnée déterminée sur un site, on ne pourrait

pas savoir s'il ne l'a jamais reçue ou s'il l'a déjà effacée. Il faut donc retenir les écritures soumises à chaque site c.à.d les écrire dans un journal propre au site pour une gestion efficace des échanges.

La journalisation décrit l'histoire des données ou leurs versions successives. Elle représente les mutations des données par des opérations (ex. create, update, delete) identifiées (par identifiant unique) et associées aux objets les subissant. Elle les enregistre suivant un ordre monotone croissant conforme à celui de leur exécution.

Deux protocoles *undo* et *redo* manipulent les actions instables du journal dont les changements n'ont pas encore été réflétés sur les données persistantes stockées localement sur chaque site. On peut appeller ces actions tentatives puisqu'elles pourront être refaites et défaites tant qu'elles ne sont pas validées et stables. *Undo* défait certaines actions pour restaurer l'ancienne image ou valeur d'un objet, tandis que *redo* refait les actions pour s'assurer qu'elles sont performées et génère ainsi la nouvelle image.

On [10] peut associer à chaque opération une opération de compensation qui l'annule au cas où on veut restituer un état des réplicas correspondant à un instant donné. Cette exécution peut se faire par deux deux méthodes : soit par gestion d'un pointeur sur la dernière opération supprimée, soit par écriture explicite de l'opération de compensation. Le premier modèle modifie uniquement l'état courant des données et non le journal tandis que le second est plus riche et performant mais grossit le journal.

Certains systèmes comme [1] intègrent des mécanismes de groupement des opérations produites simultanément (séquence liée) afin de préserver l'atomicité du groupe lors de la synchronisation et de la compensation.

Donc la journalisation fournit l'information nécessaire au rétablissement du système en présence de pannes et de conflits. Néanmoins elle peut engendrer un espace de stockage énorme affectant coût et performance. Un mécanisme de nettoyage doit alors s'associer au jouranl afin de réduire la cosommation d'espace disque tout en veillant à ne pas l'appauvrir en information.

3.3 Réconciliation

Les utilisateurs peuvent effectuer sur les réplicas des écritures conflictuelles. Cela concerne généralement des opérations comme *delete*, *update* ou *insert* portant sur un même objet et se produisant presqu'en même temps . Or afin d'établir une cohérence éventuelle entre tous les réplicas, il faut pèriodiquement fusionner toutes les actions tentatives. Cela introduit le problème de réconciliation qui consiste à résoudre les conflits. Nous présentons deux approches de réconciliation : la première est sémantique et la deuxième repose sur l'estampillage.

3.3.1 Réconciliation sémantique

La réconciliation sémantique est une approche flexible permettant à l'utilisateur de spécifier certains critères et exploitant des propriétés sémantiques telle que la commutativté et l'idempotence afin de réduire les conflits. Deux actions concurrentes mais commutatives ne sont pas conflictuelles. La solution suivante utilise la réconciliation sémantique.

IceCube IceCube [11] propose une méthode générale de réconciliation sémantique basée sur des contraintes réelles entre les actions. Une *contrainte* représente une précondition pouvant être fournie par différentes sources : l'utilisateur, l'application, un type de données, etc.
Pour cela, Ice Cube assure des abstractions permettant d'exprimer les sémantiques d'une contrainte d'une façon concise, comme la dépendance (*predSucc(a,b)* : b s'éxecute seulement après l'exécution réussie de a), l'implication (*parcel(a,b)* : tout ou rien), l'exclusivité (*mutuallyExclusive(a,b)* : a ou b s'éxecute et jamais les deux ensemble), le choix (*alternative(a,b)*), etc.
Les contraintes citées ci-dessus sont *statiques* et relient deux actions indéfiniement. Il existe des contraintes *dynamiques* qui concernent le succés ou l'échec d'une seule action en fonction de l'état courant (ex. une action de dépense échoue si les fonds du compte bancaire sont insuffisants). Aucun schedule ordonnant les actions ne viole une contrainte statique alors que la violation d'une dynamique cause des repositionnements arrière dans les jounaux qu'on appelle *rollbacks* (par undo).

Les réplicas étant dans un état initialement identique, les utilisateurs travaillent en déconnecté (lecture et écriture de leur réplica local) et maintiennent des journaux de leurs actions tentatives. Pèriodiquement, les sites se reconnectent et un site réconciliateur collecte les journaux. Il regroupe les actions liées par des contraintes dans des *clusters* mutuellement indépendants : c.à.d les actions de l'un commutent avec ceux de l'autre puisqu'elles ne sont pas liées par des contraintes. Il ordonne chaque cluster de la manière suivante : il sélectionne l'action ayant le plus grand mérite (estimation de son bénéfice) et la met dans un schedule puis l'élimine du cluster et rejette ses actions conflictuelles. L'itération continue jusqu'à ce que le cluster se vide. On aura alors des clusters ordonnés. Leur combinaison engendre un schedule global que le réconciliateur propage à tous les sites. Ces derniers appliquent le schedule sur leurs réplicas, résultant de nouveau à un état commun à tous. Ainsi est assurée une cohérence éventuelle.
Considérons l'exemple de la figure 8 qui représente 4 actions conflictuelles produites par les sites s_1, s_2 et s_3 ; a_n^i est une action effectuée sur le site n et ayant comme identifiant i ; textitparcel est une contrainte définie par un utilisateur et liant les deux actions sémantiquement ; T est une table relationnelle ayant comme attributs A et B et comme clé primaire K. Le réconciliateur identifie une dépendence entre a_1^1 et a_2^1 et l'application génère *mutuallyExclusive*(a_1^1, a_2^1).

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

FIG. 8 – Un exemple d'actions conflictuelles dans IceCube

Or plusieurs alternatives d'ordonnancement existent. IceCube traite cela en tant que problème d'optimisation visant à trouver le meilleur schedule compatible avec les contraintes, l'évaluation d'un schedule étant en fonction des exigences de l'application (ex. on définit le meilleur schedule comme celui qui limite

le plus les conflits et les rejets d'actions). Pour cela, IceCube explore l'espace de recherche d'une facon heuristique en le subdivisant en sous-poblèmes disjoints (les clusters). Bien que c'est un problème np-complexe, les expériences montrent qu'une solution quasi-optimale est trouvée au bout d'un temps raisonnable. De plus, ce partitionnement restreint les rollbacks à peu d'actions.

Néanmoins, l'approche centralisée de IceCube limite le passage à l'échelle : son site réconciliateur peut devenir un goulot d'étranglement dans les applications distribuées et étendues, et s'il tombe en panne, tout le système se bloque.

3.3.2 Estampillage

Les nouvelles actions soumises et reçues par un site doivent être intégrées à son journal. D'où la nécessité d'ordonnancement des actions. Puisque des actions parallèles faites sur différents réplicas peuvent entrer en conflit, il faut en plus être capable de les détecter et de les résoudre. On ne peut pas se contenter de disposer les actions à la fin du journal puisque certaines actions concurrentes non commutatives exigent un ordre global. L'ordonnancement ne peut pas reposer sur les horloges physiques des machines. En effet ces horloges doivent être synchronisées pour assurer un ordre global, ce qui est quasi-impossible dans le cas de sites ponctuellement connectés ou en grand nombre.
Dans ce qui suit, nous parlons de techniques reposant sur l'estampillage pour exprimer les relations d'ordre entre les actions et nous présentons deux solutions [12] et [1] exploitant ces techniques.

Ordre causal L'ordre causal, appelé aussi *happens-before* et défini par Lamport [13], capture les relations entre des évènements distribués (dans notre cas deux actions exécutées chacunes sur un réplica). Deux évènements sont causalement liés, si le second dépend du premier. En considérant deux actions a et b, on dit que *a précède directement b* ssi l'une des conditions suivantes est vraie :

- a et b se produisent sur le même site et a est soumise au site avant b ;
- a et b se produisent chacune sur un site (respectivement site A et site B), mais b est soumise au site B qui a déja reçu et executé a ;

Donc dans les deux cas, a est connue au moment de l'execution de b et peut par suite affecter son exécution. S'il n'y a pas de dépendance entre a et b ni dans un sens ni dans l'autre, les deux actions sont alors concurrentes.

Cet ordre partiel est implémenté en estampillant chaque action avec l'état d'une horloge logique, un simple compteur maintenu sur chaque site et incrémenté à chaque action soumise. La nouvelle valeur appelée ***estampille*** (notée h) est aussi associée à l'action correspondante. A la réception d'une action d'estampille h_j, un site i règle son horloge h_i de manière qu'elle prenne la valeur maximale entre sa valeur courante incrémentée et l'estampille de l'action reçue ($h'_i = max(h_i + 1, h_j)$). Ainsi, si *a préc ède directement b*, l'estampille de a est inférieure à celle de b et a est classée avant b dans le journal. L'exemple de la figure 9 illustre une interaction entre deux sites A et B maintenant chacun une horloge logique. x_i est une action effectuée par le site X et ayant pour estampille i. Le site B reçoit de A ses deux actions, les int ègre dans son journal suivant leurs estampilles et r ègle son horloge logique. Pour ordonner globalement les actions de tous les sites, un ordre total arbitraire peut être établi suivant des paires formées par les identifiants des sites et les estampilles des actions.

Cependant, l'ordre causal considère uniquement les dépendances entre les actions et ne permet pas de classer les actions parallèles suivant un ordre dans le journal. Si un site progresse lentement et n'entre pas fréquemment en contact avec les autres, ses actions seront alors classées avant les leurs et des rollbacks importants de leurs jounaux seront impliqués chaque fois qu'il entre en contact avec eux.

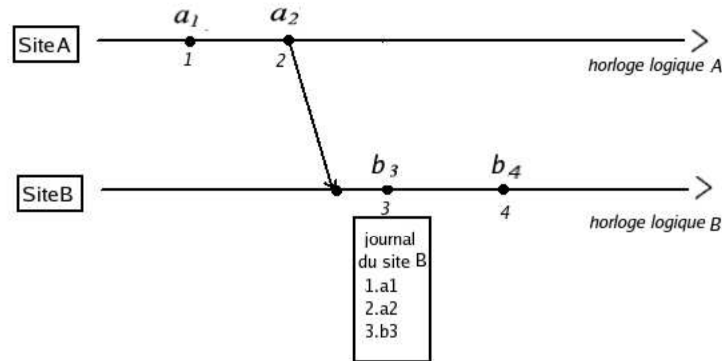

FIG. 9 – Un exemple d'horloge logique avec deux sites A et B

Vecteurs d'estampilles Les vecteurs de versions [8] dressent une vue globale de l'état d'avancement de tous les sites tout en capturant l'ordre causal ; ce qui permet contrairement aux horloges logiques de déterminer durant les échanges quelles actions manquent à un site donné. Chaque site i gère un vecteur V_i par réplica. Dans la suite, on suppose que tout site stocke un seul réplica par besoin de simplicité. En pratique, le vecteur ressemble à une table qui fait correspondre l'identité de tout site y compris lui-même à son estampille. Pour un site i, $V_i[i]$ représente la valeur de son horloge logique qu'il incrémente à chaque action soumise ou reçue (cf. paragraphe précédent), et $V_i[j]$ représente l'estampille la plus récente de j que i a reçue.
Les vecteurs de version garantissent la propriété préfixe : si $V_i[j]$ est égal à t, cela implique que le site i a déjà reçu toute action du site j ayant une estampille infèrieure ou égale à t. Ainsi, durant les échanges, le récepteur compare son vecteur à celui de l'émetteur et ne sélectionne que les actions ayant une estampille supérieure aux composantes de son vecteur. Ces composantes sont ensuite positionnées à la plus grande estampille reçue. Dans l'exemple de la figure 10, le site B reçoit de A une action ; le site C reçoit de B deux actions, celle de B ainsi que celle de A. Si deux actions comme a_1 et b_1 dont aucune des estampilles ne domine l'autre mettent à jour un même tuple d'une table relationnelle, elles sont considérées en conflit ; en effet (cf. paragraphe précedent) ce conflit est justifié puisqu'aucune des deux actions n'est executée en tenant compte de l'autre (aucune ne *précède directement* l'autre).
Donc, la détection de conflits par cette technique opère de la manière suivante : si un réplica est plus récent qu'un autre, son vecteur domine celui de l'autre ; les données et le vecteur de la version dominante remplacent ceux de la version

dominée. Si deux vecteurs ne sont pas égaux et aucun ne domine l'autre, un confit est détecté. Des mécanismes de résolution des conflits doivent alors être mis en place.

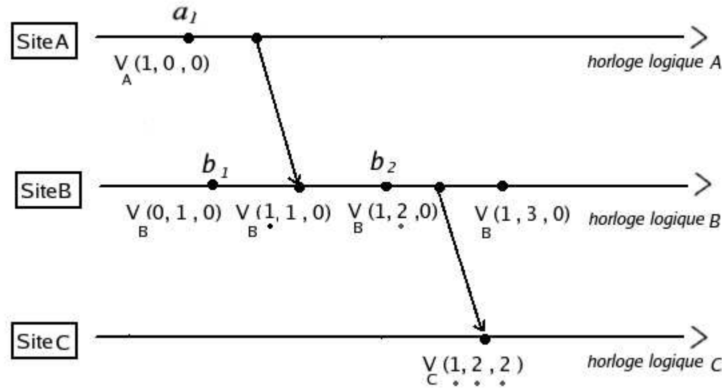

FIG. 10 – Un exemple de trois vecteurs de versions $V_{site}(V_{site}[A], V_{site}[B], V_{site}[C])$ maintenus sur trois sites

Bayou Bayou [12, 14] est une infrastructure de stockage gérant les conflits d'une activité concurrente tout en dépendant uniquement sur la faible connectivité disponible dans les applications mobiles. Bayou supporte une architecture client-serveur, ce qui suppose des sites fiables ayant des fonctionnalités spécifiques.

Un client réplique une base de données sur un ordinateur portable, la modifie en déconnecté en appliquant des actions tentatives et se met à jour avec n'importe quel serveur accessible. Il maintient un journal des actions ordonnées dont le contenu est échangé incrémentalement, se basant sur les vecteurs de versions. A chaque action, il associe une précondition spécifique de l'application (*dependency check*) et une méthode de réconciliation (*merge procedure*) qui sont propagées avec elle et qui permettent respectivement de détecter un conflit et de le résoudre. Chaque fois qu'une action est ajoutée à un schedule ou que l'ordonnancement est modifié, le système vérifie s'il y a des dépendances avec d'autres actions déjà inclues au schedule et si c'est le cas, il lance la procédure de fusion qui tente d'appliquer ses alternatives. Au cas où cette résolution automatique échoue, le conflit est enregistré pour une résolution manuelle éventuelle.

Bayou adopte un protocole dit *primary-commit protocol*. Il partitionne les données de façon à avoir des partitions indépendentes et désigne par partition un site *primaire* pour la validation des actions relatives à ces données. Un site primaire ordonne les actions au fur et à mesure qu'il les reçoit, produit la décision finale et stable concernant l'ordre et la résolution des conflits et l'envoie aux autres sites. Ces derniers ayant chacun produit un ordonnancement différent, doivent refaire et défaire des parties de leurs journaux tant qu'elles ne sont pas stables et donc modifiables. Dans l'exemple de la figure 11, on considère un seul site primaire (pour la simplicité) ; les site A et B diffusent leurs actions tentatives respectives c et d en attendant qu'elles soient ordonnées par le site primaire qui

a déjà ordonné a et b. La gestion des conflits n'est pas transparente : un client peut observer des données incohérentes ou tentatives et doit être conscient que certaines de ses écritures et lectures ne sont pas encore confirmées et stables. Cependant, Bayou offre des garanties de sessions pour limiter les divergences observées par des clients désirant accéder plusieurs serveurs :

- Ecritures monotones (MW) : l'écriture d'un utilisateur n'est acceptée qu'après que toutes ses écritures soient intégrées à ce même réplica ;
- Lectures monotones (MR) : les lectures successives d'un utilisateur retournent des valeurs équivalentes ou de plus en plus récentes ;
- Lecture des ses écritures (RYW) : le contenu d'un réplica lu par un utilisateur réflète ses écritures précédentes ;
- Ecritures succédant aux lectures (WFR) : l'écriture d'un utilisateur n'est acceptée qu'après que toutes les écritures lues par lui soient intégrées à ce même réplica ;

Un utilisateur se déplaçant d'un réplica à un autre percevra des vues cohérentes. Il peut par exemple changer son mot de passe sur un site et puis accéder à son compte d'un autre site grâce à RYW.

De plus, la détection d'un conflit et les procédures de fusion sont déterministes afin que les conflits soient gérés uniformément par tous les serveurs.

Bayou intègre un réel support paramétrable de détection des conflits et supporte ainsi une variété d'applications collaboratives. Nénamoins, la mécanique de résolution est déchargée sur l'utilisateur : il doit prévoir les éventuels conflits et trouver des alternatives. Bayou repose sur des sites fiables avec des fonctionalités spéciales (les serveurs), ce qui ne convient pas à un système P2P.

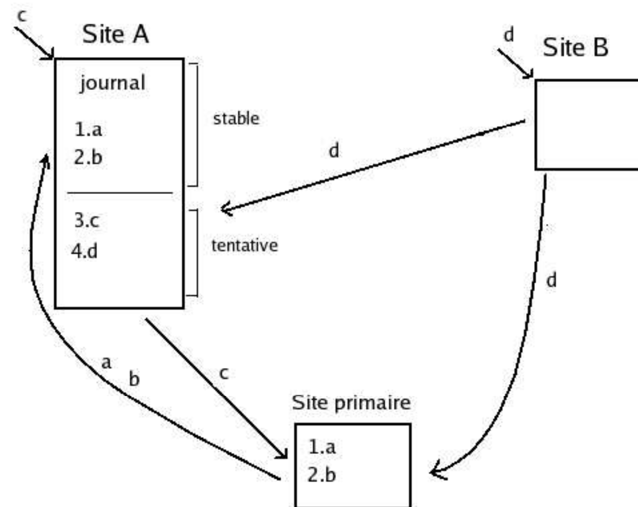

FIG. 11 – Un exemple de Bayou

Approche pour les applications collaboratives asynchrones En [1], l'auteur vise la collaboration de réplicas autonomes dans des environnements ponctuellement connectés, favorisés par l'essor de l'informatique nomade (ex.

PDA, téléphone portable...). L'utilisateur ne peut pas rester connecté en permanence (connexion et communication coûteuses...), il peut accéder aux données par plusieurs points d'accès et doit donc trouver une copie disponible en écriture et en lecture chaque fois qu'il se connecte. Le protocole de réplication adopté dans ces applications doit s'adapter à cet environnement de faible connectivité.

Une *synchronisation* est définie par le fait que deux réplicas de deux sites établissent contact pour échanger leurs opérations (ou actions) et résolver leurs divergences. Lors d'une synchronisation monodirectionnelle c.à.d. impliquant un émetteur et un récepteur, on part d'un préfixe commun aux deux journaux, délimité par la plus grande estampille commune pour déterminer les opérations inconnues du récepteur. Un exemple mettant en jeu deux sites R et E est donné dans la figure 12 : en considérant le journal de R, $1.A$ indique que R a reçu une opération d'estampille 1 produite par le site A. Le récepteur mémorise son suffixe divergent et restaure le dernier état commun avec l'émetteur par compensation (cf. 3.2).
Tout réplica maintient, en conjugué avec son vecteur de version, une table associant à chaque partenaire de synchronisation le plus grand indice d'arrivée déjà collecté, qu'il envoie à ce partenaire pour qu'il filtre les opérations à envoyer. L'emploi conjugué de vecteurs de versions et de table d'arrivées prend compte des synchronisations avec divers partenaires et évite de collecter des opérations déjà reçues ainsi que la collecte complète du journal lors d'une première synchronisation avec un nouveau partenaire. La figure 13 représente le journal du récepteur de la figure 12 après l'intégration des nouvelles opérations.

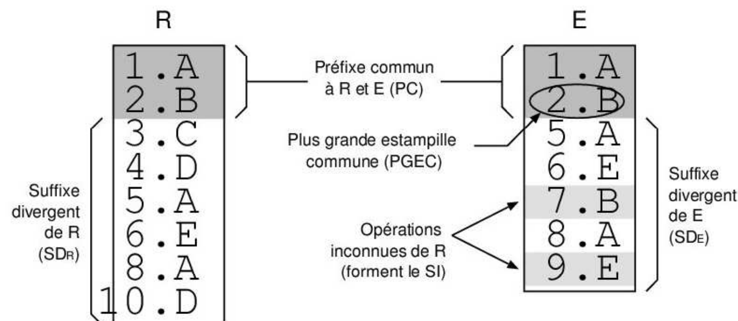

FIG. 12 – Des parties des journaux du récepteur R et de l'émetteur E entrant en jeu lors d'une synchronisation

Le récepteur doit alors détecter les conflits entre les opérations reçues at celles de son suffixe divergent. Ensuite, il doit les résoudre et intégrer les opérations produites à son journal pour les éxecuter localement. Les opérations rejettées sont traitées comme des opérations ordinaires en tant qu'opérations de rejet enregistrées dans le journal afin d'assurer la suppression cohérente de l'ensemble des réplicas.

Le système considère des conflits de mutation (c.à.d. des mises à jour divergentes d'un même objet) et des conflits inter-objets (des objets liés par ré-

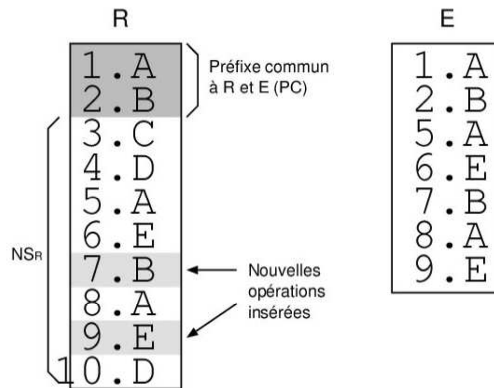

FIG. 13 – Intégration des mises à jour dans le journal du récepteur R

férencement et des doublons) qu'il peut détecter automatiquement. Un modèle d'estampillage logique à incrément temporel assure un ordonnancement global cohérent avec le temps physique et résoud "naturellement" les conflits de mutation : par exemple, si on a un conflit entre deux mises à jour, les deux opérations sont ordonnées et intégrées. S'il s'agit du même attribut d'un tuple, le résultat réflètera la mise à jour ayant la plus grande estampille.

Des extensions permettent d'adapter le système à des cas particuliers. Pour les réplicas obsolètes, on impose des barrières de synchronisation qui assurent la stabilité d'une partie régulièrement croissante des journaux en refusant d'intégrer de trop vieilles opérations. Pour la réplication partielle, on intègre un filtre lors d'une synchronisation avec changement d'une à l'autre.

3.4 Diffusion des mises à jour

Une solution naïve pour diffuser les actions entre les sites serait d'envoyer pèriodiquement le journal tout entier de chaque site vers un autre. Toutes les actions seront éventuellement propagées à tous les sites, en dépit des pannes et déconnexions temporaires. Cela résulte en une consommation énorme de temps et de ressources (bande passante, stockage, etc.). Diverses approches ont été exploitées dans le but de réduire la quantité de données échangées entre les sites et d'assurer une cohérence éventuelle.

3.4.1 Topologie de communication imposée

Il existe des protocoles de synchronisation qui définissent des contraintes sur le choix de patenaires et la fréquence de synchronisation. Ces protocoles veulent contrôler la topologie de communication et le flux de données dans le but d'améliore la vitesse de propagation et de réduire les conflits [8].

Le taux de conflits peut être réduit en connectant les réplicas suivant des manières spécifiques : arbre (*tree*), étoile (*star*), anneau (*ring*), etc.

Une organisation des sites en étoile implique la centralisation de la communication autour d'un site unique par lequel doit passer toute mise à jour avant

d'être propagée. L'architecture deux-tiers ou client-serveur (*two-tier*) est une généralisation de cette topologie : elle distingue entre les sites les plus connectés, les *core sites*, et ceux faiblemet connectés, les *mobile sites*. Les *core sites* communiquent entre eux alors que les *mobile sites* communiquent seulement par l'intermédiaire des premiers. Coda [15, 16, 17], un système de fichiers répliqués, supporte cette architecture deux-tier et assure un modèle *read-one, write-all*. Un client travaille en déconnecté sur les fichiers de son cache et se reconnecte à un groupe de serveurs disponibles pour propager ses modifications.
Néanmoins, on limite la disponibilité et l'équilibrage de charge puisque les sites centraux peuvent se surcharger ou tomber en panne, bloquant ainsi les autres.

3.4.2 Protocoles épidémiques

Les protocoles épidémiques [18] se basent sur la théorie des épidémies puisqu'ils comptent sur la diffusion d'un message comme une maladie contagieuse dans une population. La transmission transitive de mise à jour d'un site à l'autre vise à infecter un maximum de réplicas dans un minimum de temps et de messages.

Un protocole anti-entropique se base sur une communication de paires choisies aléatoirement : des paires arbitraires établissent contact pèriodiquement et résolvent les divergences entre leurs données. Les réplicas diffusent de proche en proche leurs nouvelles actions mais aussi celles qui sont reçues des autres. Une même action peut donc être collectée sur plusieurs réplicas. Pour cela, il est important de savoir l'état d'avancement du récepteur vis-à-vis d'autres réplicas. Les journaux et les mécanismes de vecteurs associés à ce protocole assurent des échanges incrémentaux et réduisent ainsi le volume de données échangé. Bayou [14] utilise un protocole anti-entropique pour la propagation des mises à jour entre ses réplicas, tout en se basant sur les vecteurs de versions.

Les opérations de suppression (delete) sont traitées et propagées comme les mises à jour en utilisant des certificats de décés. Pour empêcher l'accumulation dans la mémoire, on leur associe des dates d'expiration au bout desquelles les certificats sont effacés. On peut les garder en permanence sur un site ou plus pour garantir la supression par tous les sites, y compris ceux qui tardent à se synchroniser.

La technique pull repose sur l'initiative du récepteur : un site demande des actions à un autre quand il en a besoin. Elle convient aux sites qui restent longtemps déconnectés et qui contactent les autres pour se mettre à jour une fois connectés.

La technique push repose sur l'initiative de l'émetteur : un site ayant de nouvelles actions, effectue une livraison proactive aux autres (sans leur demande). Elle ne nécessite aucune information de la part du récepteur et convient aux rumeurs récentes c.à.d les mises à jour neuves qui ne sont pas encore trop connues. Des variantes heuristiques de cette technique sont : le *blind flooding* et le *gossip* [8].
Dans le *blind flooding*, un site ayant une nouvelle action la diffuse aveuglément à tous ses voisins qui à leur tour la font suivre aux autres.
Le protocole *gossip* ou diffusion de rumeurs vise à minimiser les messages dupliqués reçus. En commençant comme le *blind flooding*, chaque site compte le nombre de duplicas qu'il reçoit pour chaque action et arrête de faire suivre une action dès que ce nombre dépase une limite.

Le *directionnal gossip* [19] est une adaptation du *gossip* à un réseau étendu WAN interconnectant plusieurs réseaux locaux LAN. Un site compte le nombre de chemins distincts traversés par une action. Un lien inter-site non partagé par plusieurs chemins est par suite considéré d'une grande importance puisqu'il peut être le seul lien connectant certains sites, auquels on attribue alors des poids faibles . Le site propage les opérations plus fréquemment à travers ces liens critiques. Pour ceux partagés par plusieurs chemins, le site le fait moins fréquemment. On dit que ce processus "gossip" pour les voisins de poids lourds et "flood" pour les voisins de poids faibles. Ce protocole est plus fiable qu'un *gossip* traditionnel tout en impliquant moins d'overhead que le *flooding*. Il est robuste, résistant contre les pannes usuelles des liens et des pocesseurs. Les échanges peuvent se faire suivant pull, push ou push-pull pour des échanges bidirectionnels.
Ces potocoles épidémiques assurent une garantie probabiliste de cohérence éventuelle : la probabilité d'avoir des réplicas qui n'ont pas encore convergés décroît exponentiellement avec le temps. Ils requièrent très peu de garanties du réseau et s'accomodent avec les réplicas diponibles et les liens non coupés.

3.5 Comparaison entre diverses approches

Les tableaux suivants résument les différents aspects des protocoles adoptés par les approches que nous avons présentées. Ficus [20, 21] et Coda [15, 16, 17] n'ont pas été pleinement abordés dans notre étude puisqu'ils sont fortement liés au domaine des fichiers. Cependant, nous dressons un aperçu de leurs aspects qui présentent des points communs avec ceux étudiés.

Approche	Objet répliqué	Topologie	Cohérence
IceCube	généraliste	non fixée	éventuelle
Bayou	généraliste	non fixée	éventuelle
Ficus	fichier	non fixée	éventuelle
Coda	fichier	star	éventuelle
Approche collaborative	généraliste	non fixée	éventuelle

TAB. 1 – Comparaison(1) entre plusieurs systèmes de réplication optimiste

Approche	Détection de conflits	Résolution de conflits
IceCube	sémantique(contraintes)	définie par utilisateur(alternatives)
Bayou	préconditions	définie par utilisateur(merge proc.)/manuelle
Ficus	VV et arborescence	resolvers/manuelle
Coda	VV et journal	resolvers/manuelle
Approche collaborative	automatique	ordonnancement/spécifique de app.

TAB. 2 – Comparaison(2) entre plusieurs systèmes de réplication optimiste

Approche	Ordonnancement final des op.	Diffusion m.à.j.
IceCube	optimisation	par le reconciler
Bayou	ordre de réception au primaire	épidémique avec VV
Ficus	VV	pull
Coda	ordre de réception au site primaire	push
Approche collaborative	estampillage logique à incrément temporel	VV et TA

TAB. 3 – Comparaison(3) entre plusieurs systèmes de réplication optimiste

Troisième partie
Réplication dans les systèmes P2P

4 Introduction

L'essor de l'internet et les plusieurs millions de machines et d'utilisateurs voulant partagé des resssources dispersées, ont favorisé le développement des systèmes P2P. Ces systèmes en constante évolution et expansion, acceptent tout utilisateur qui contribue par des données et des ressources informatiques tout en lui assurant anonymat et autonomie. Tous les pairs ont des capacités et fonctionnalités identiques en dépit des différences au niveau des ressources.

Les systèmes P2P visent à fournir des services complètement décentralisés, en équilibrant dynamiquement les charges de stockage et de traitement entre tous les participants, en présence de pairs joignant et quittant le réseau à tout moment. Leur nature décentralisée implique des interactions entre pairs basés uniquement sur l'information locale de chacun. La disponibilité des processus et des machines participant à un système P2P est imprévisible. Les services P2P ne peuvent donc pas garantir l'accès à des ressources individuelles. Par contre, en exploitant la réplication de ces ressources, ils peuvent garantir l'accès à leurs copies et résister ainsi à leur volatilité et leur fiabilité déconcertante. La réplication optimiste est largement utilisée dans les applications P2P pour assurer la performance et la disponibilité maximales.
Or pour accéder à une donnée, il faut pouvoir la localiser surtout que les pairs n'ont pas une adresse IP permanente : les services P2P doivent gérer le placement des données, afin de pouvoir les stocker et les retirer au besoin, et ils doivent intégrer des mécanismes de cryptage et d'authentification pour préserver leur sécurité dans un environnement ouvert [22].

Cette partie présente les diverses caractéristiques des P2P et les défis techniques auxquels ils sont confrontés en tentant d'assurer la disponibilité requise. Des solutions adoptées dans le domaine de réplication des données sont exposées en soulignant leurs contributions et leurs limites.

5 Architecture des systèmes P2P

Les systèmes P2P peuvent être classés suivant leur architecture [23]. On distingue entre trois classes :

5.1 P2P non structuré

Un P2P non structuré ne contrôle ni la topologie du système ni le placement de ses données. Chaque pair connaît uniquement ses voisins sans toutefois connaître leurs ressources. Les pairs sont énormément autonomes et des réplicas peuvent facilement se remplacer en cas de pannes. Pour localiser un fichier, un mécanisme de recherche basé sur le *flooding* est employé : une requête est propagée à tous les voisins compris dans un rayon donné. Cette méthode supporte une grande expressivité des requêtes mais consomme trop de bande passante ;

elle ne passe pas à l'échelle puisqu'elle génère des charges énormes sur les participants. De plus, elle n'est pas toujours efficace et ne retourne pas des résultats exacts et rapides.

5.2 Super-pair

Des pairs spécifiques, les super-pairs, sont chargés de veiller à la sécurité et la qualité de service. Chaque super-pair supervise un nombre de pairs et assure certaines fonctions complexes : indexer les ressources de ses pairs, contrôler leurs accès, gérer les méta-données et la communication avec les autres pairs, etc. Néanmoins, cela consomme moins de bande passante mais plus de puissance de processeur. Les super-pairs peuvent aussi présenter des points fragiles menaçant le système en cas de pannes (*single-point-of-failure*).

5.3 P2P structuré

Un P2P stucturé contrôle la topologie du système et le placement des données sans toutefois gérer un index centralisé. Les données ne sont pas placées arbitrairement mais à des locations spécifiques et par suite les requêtes sont plus faciles à satisfaire. Certains systèmes intègrent un mécanisme d'indexage décentralisé, par l'intermédiaire d'une table de hachage qui fait correspondre des clés à des valeurs : les clés (appelées aussi *GUIDs*) sont générées par hachage des valeurs ou des contenus des objets stockés, sont uniques et utilisées par les techniques de routage pour retrouver ou stocker les objets [24]. Un modèle *DHT, distributed hash table,* fournit l'interface *put(GUID, data)* pour stocker la donnée et l'interface *get(GUID)* pour l'accéder. Une donnée est répliquée et stockée au niveau des sites ayant les GUIDs (générés par hachage de leurs adresses) les plus proches numériquemnt de sa propre clé. Chaque pair connaît un nombre détérminé de pairs et maintient une table de routage associant leurs GUIDs à leurs addresses. Le routage exploite ces tables pour restreindre et cerner la recherche lorsqu'il passe d'un pair à l'autre. Quand un nouveau pair rejoint le réseau ou qu'un ancien le quitte, les pairs coopèrent pour refléter ces changements sur leurs tables. Cette approche assure qu'une recherche exacte soit efficace, et elle permet le passage à l'échelle. Cependant, un système ainsi structuré limite l'autonomie des pairs puisque chacun est responsable d'un ensemble de clés. Il peut devenir déséquilibré lorsque les données diffèrent par leur popularité et leur nombre de réplicas : les sites populaires deviennent surchargés contrairement aux autres. Face à ces deux problèmes, des techniques de mise en cache ou de réplication des clés par plusieurs pairs peuvent augmenter la disponibilité des données.
Dans le contexte du travail de recherche en cours, nous nous intéressons à quelques unes des approches DHT décrites ci-dessous.

CAN CAN [25], un exemple de DHT, se base sur un espace virtuel de coordonnées cartésiennes à d dimensions. A chaque instant, cet espace est dynamiquement partitionné entre les nœuds du système, de façon que chaque nœud possède une zone distincte qui représente un morceau de la table de hachage globale. La figure 14 montre un espace de coordonnées [0, 1] x [0, 1] partitionné

entre 5 nœuds ($C(0.0-0.5, 0.5-1.0)$ signifie que la zone du nœud C est comprise entre 0 et 0.5 sur l'axe des x et entre 0.5 et 1.0 sur l'axe des y.
La localisation d'une valeur est déterminée par ses coordonnées générées à partir de sa clé par une fonction de hachage détérministe. Un nœud maintient des informations sur un nombre fixe de zones adjacentes, indépendant du nombre total de nœuds : sa table de routage associe l'adresse IP de chaque voisin aux coordonnées de sa zone. Dans la figure 15, on veut accéder une valeur. On applique des fonctions de hachage sur sa clé et on obtient ses coordonnées. Pour le routage, chaque nœud dirige la requête à un voisin.
A l'arrivée ou le départ d'un nœud, une réallocation d'espace (division d'une

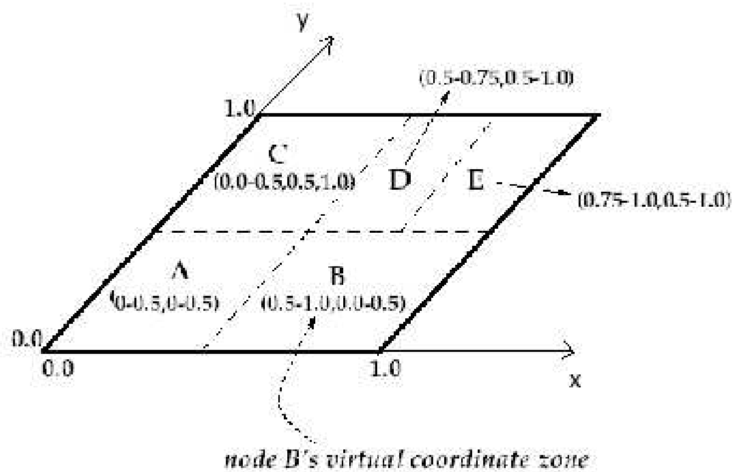

FIG. 14 – Un exemple d'espace de coordonées 2-d avec 5 nœuds dans CAN

zone ou fusion de deux respectivement) et une mise à jour des tables des voisins seulement, sont appliquées. Le routage contourne les zones en pannes, puisqu'il y a plusieurs chemins entre deux points de l'espace de CAN et des optimisations peuvent être appliquées. L'équilibrage de charge est réalisé par cache ou réplication des clés populaires.

Chord Dans Chord [26], l'espace de clés est ramené à une séquence de locations ordonnée et disposée en cercle. Une clé est assignée à un nœud dont l'ID est supérieur ou égal à celui de la clé et qui sera désigné comme son *successeur*. Une requête est transmise à partir d'un nœud vers un de ses successeurs jusqu'à trouver la clé souhaitée : il détermine le plus grand prédécesseur de la clé dans sa table de routage et lui renvoie la requête (voir figure 16). Par une méthode pèriodique de stabilisation, chaque nœud détecte les pannes parmi ses successeurs et prédecesseurs ; le nœud en panne est remplacé par son successeur.

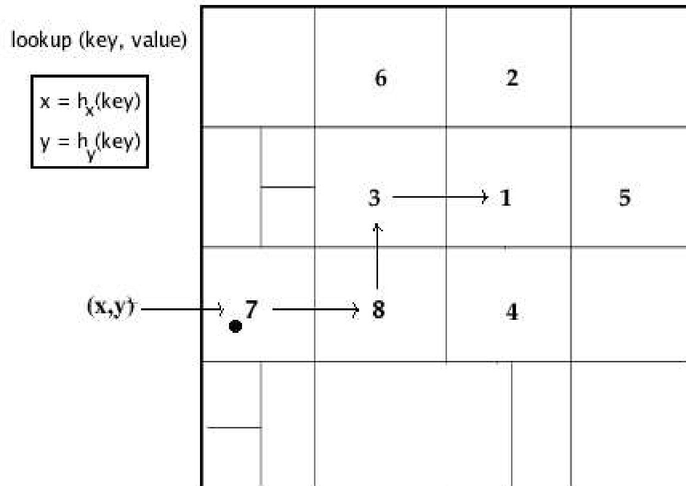

FIG. 15 – Un exemple de recherche d'une valeur *value* ayant une clé *key* dans CAN

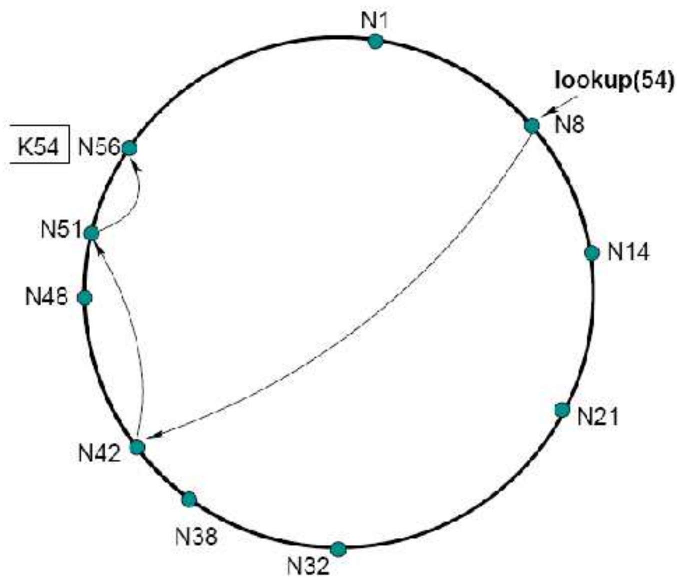

FIG. 16 – Un exemple de routage dans Chord

Pastry Dans Pastry [22], une infrastructure de routage en cercle (comme Chord), chaque nœud maintient une table de routage structurée en arbre, contenant les GUIDs (en valeur hexadécimale) et les adresses IP d'un ensemble de nœuds dispersés sur toute la plage de GUIDs possibles, avec une converture plus

dense des GUIDs numériquemnet proches de celui du nœud. A la construction de chaque table, les distances réseau sont utilisées pour sélectionner les voisins adéquats et minimiser les temps de réponse : on compare les candidats pour chaque position et on choisit le plus proche en distance disponible.

Tapestry Tapesry [22] emploie le même mécanisme de routage que Pastry mais il masque sa table de hachage par une interface *DOLR* plus flexible que celle de DHT : les objets peuvent etre stockés n'importe où et la couche DOLR est responsable de maintenir une correspondance entre les GUIDs des objets et les adresses des nœuds stockant des réplicas de ces objets. Chaque nœud publie les GUIDs des ressources qu'il possède par l'intermédiaire de *publish(GUID).* Les ressources répliquées sont publiées avec le même GUID et le routage décide du réplica disponible le plus proche à accéder.

6 Solutions de réplication en P2P

Dans cette section, nous abordons des solutions de réplication adoptées dans des environnements P2P.

6.1 Napster, Gnutella

La plupart des systèmes P2P exsistants considèrent que les données sont statiques (*read-only*) et n'abordent pas les mises à jour. Les systèmes centralisés comme Napster maitiennent des index centralisés des données disponibles sur les pairs connectés. Une mise à jour d'une donnée par le pair qui la possède implique une nouvelle version non propagée à ceux répliquant cette donnée. Cela résulte en 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 réplicas n'est alors garantie. Le même cas est observé chez des systèmes décentralisés comme Gnutella.

6.2 Freenet

Freenet [24], un système P2P de stockage et de recherche d'information structuré en DHT, aspire à la liberté de parole et l'anonymat sur l'internet en assurant une sécurité et un cryptage renforcés. Les émetteurs et les receveurs de l'information communiquent tout en restant anonymes. Le nombre et les locations des réplicas sont adaptés selon les demandes sur chaque fichier. Les mises à jour sont partiellement adressées : elles ne sont effectuées que par les propriétaires des données et propagées uniquement aux pairs connectés suivant une stratégie heuristique avec des garanties incertaines d'une éventuelle cohérence. Des clés obtenus par hachage de contenus sont utilisées pour distinguer entre les versions d'un fichier ainsi que pour le routage. Le système ne prend pas en considération le dynamisme des pairs.

6.3 OceanStore

OceanStore [22, 27] s'appuie sur Tapestry et fournit un support de stockage persistant de fichiers mutables pour les applications P2P (dynamiques et à large

échelle). Il se base sur un mécanisme de gestion de cohérence inspiré de Bayou et adaptable aux besoins de l'application : des prédicas ordonnés sont associés aux actions et évalués avant leur application mais les procédures de fusion n'existent pas par manque de confiance aux pairs. Chaque objet est représenté comme une séquence ordonnée de versions non mutables conservées pour toujours ; toute mise à jour de l'objet génère une nouvelle version. Certains blocs de données d'un objet restent inchangés d'une version à l'autre qui intègre alors une indirection vers ces blocs au lieu de les représenter ; ce qui réduit la consommation de ressource de stockage. Des identifiants uniques et permanents générés par hachage, les GUIDs, sont associés à chaque objet, version et bloc ; les pairs stockant des réplicas de ces éléments publient leurs GUIDs qui seront alors utilisés par les techniques de routage d'une requête. Chaque fois qu'une nouvelle version est créée, un certifica est généré associant les GUIDs de l'objet et de la version avec son estampille ; le GUID de cette version comme toute autre référence celui de la précédente, créant ainsi une chaîne de références qui permet à un utilisateur muni d'un certifica d'accéder à n'importe quelle version.
Pour faire face aux pannes et aux comportements malicieux de certains pairs non fiables, chaque mise à jour requiert le consentement d'un ensemble réduit de pairs stockant des copies primaires, appelé *inner ring* et séléctionné à la création d'un objet. L'*inner ring* est invoqué à chaque écriture concernant sa copie primaire ; il décide d'un ordre total sur une série de mises à jour qui seront alors appliquées atomiquement. Les actions validées correspondent à des certificas propagés aux pairs stockant une copie secondaire de l'objet concerné : le protocole de diffusion est épidémique basé sur un arbre de diffusion contenant les GUIDs des pairs en question. En attendant, ces pairs peuvent appliquer des actions tentatives et les ordonner suivant leurs estampilles : cet ordre pourrait donc correspondre à l'ordre final. La figure 17 montre le chemin complet d'une mise à jour a : (a) une action effectuée sur un réplica secondaire r est envoyée à un réplica primaire R du même objet ainsi qu'à d'autres réplicas arbitraires ; (b) pendant que l'*inner ring* performe le protocole de validation de a, les réplicas secondaires la propagent entre eux épidémiquement ; (c) le résultat de (a) est diffusé suivant l'arbre de diffusion.
Un service P2P basé sur OceanStore est une solution effective pour une distribution de fichiers qui ne sont pas modifiés fréquemment puisque le nombre de copies primaires est très restreint. OceanStore repose sur des liens fiables et rapides (*high speed links*) alors que cet aspect n'est point garanti dans un système P2P.

6.4 P-Grid

P-grid est un système P2P structuré qui se base sur une table virtuelle distribuée similaire à une DHT et qui s'auto-organise dynamiquement en vue d'une allocation de ressources [28] et d'un équilibrage de charge optimaux. Dans P-Grid, les mises à jour sont adressées pr un algorithme [29] *push-pull*, basé sur une réplication mono-maître, dans le but de réduire le nombre de messages impliqués par *flooding*, dans un contexte de réplicas qui se déconnectent pour de longues durées et fréquemment. Il assure des garanties probabilistes de cohérence et des latences réduites et s'applique dans des environnements décentralisés et non fiables.
Durant la phase *push* déclenchée par la copie primaire d'un objet, une mise à

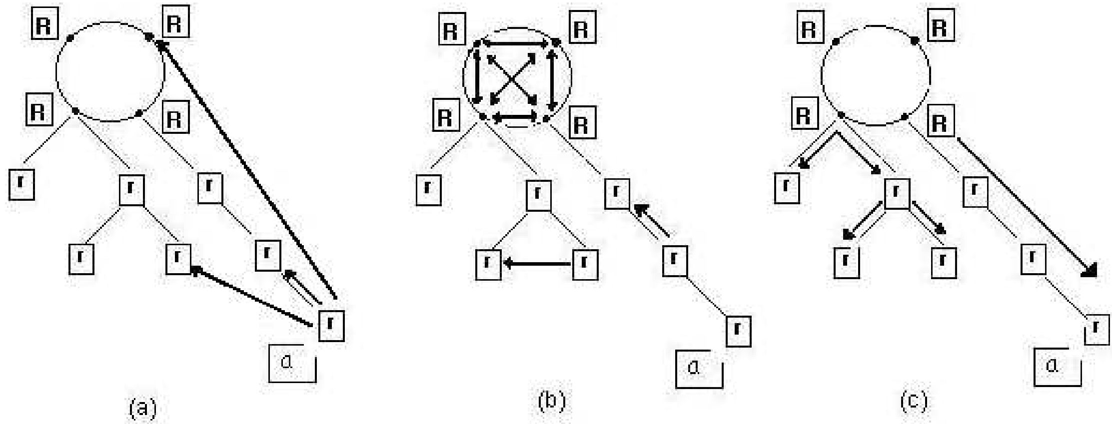

FIG. 17 – Le chemin d'une mise à jour dans OceanStore

jour est propagée par *flooding* parmi les pairs connectés et stockant une copie secondaire de cet objet ; les messages dupliqués sont évités par la propagation simultanée d'une liste des pairs ayant été informés et par diminution progressive de la probabilité qu'un pair diffuse. Pour localiser les pairs connectés, les pairs ayant reç une mise à jour peuvent le confirmer à leurs émetteurs par des messages (*ack*). Pour adapter dynamiquement les paramètres de l'algorithme (comme la probabilité de diffusion par un pair donné), on emploie le nombre de duplicas reçus par un site afin d'estimer l'expansion d'un message.
La phase *pull* porte sur les pairs déconnectés qui ont râté des mises à jour et qui cherchent dès qu'ils se reconnectent à se synchroniser avec le pair connecté le plus à jour.
Cet algorithme exploite la connaissance en connectivité logique indépendamment de la connectivité physique ou relative au réseau. Néanmoins, le maintien d'information sur la location des pairs concernés par une mise à jour n'est pas abordé surtout que c'est difficile avec le dynamisme des adresses IP. L'approche ne prend pas en compte des mises à jour conflictuelles puisqu'elle intègre un modèle mono-maître et considère que les conflits sont rares. On suppose que des garanties probabilistes au lieu d'une cohérence stricte, sont suffisantes. L'overhead dû aux messages est toujours considérable, ce qui résulte en une consommation de bande passante considérable en présence de mises à jour fréquentes.

6.5 UPTReC

UPTReC [30], *Update Propagation Through Replica Chain*, vise la cohérence dans un système P2P non structuré. Un processus *push-pull* se base sur une chaîne logique de réplicas par fichier. Les pairs d'une chaîne maitiennent chacun une connaissance partielle de cette chaîne bidirectionnelle (IP et ID des pairs les plus proches) et assurent la propagation progressive d'une mise à jour sur un fichier en se relayant. Dans la figure 18, un site i ayant produit une action l'envoie, par *push* suivant les deux sens de la chaîne, aux sites connectés dont il maintient l'information c.à.d $i-k$, ... ,$i+k-1$. Le site le plus loin, $i+k-1$,

est notifié pour prendre le relais et poursuivre la propagation de la mise à jour le long de la chaîne.
Pour construire la chaîne d'un fichier donné, chaque pair retient les pairs qui lui ont demandé et répliqué ce fichier. Un pair déconnecté qui se reconnecte diffuse son addresse IP à la chaîne par la même méthode push utilisée pour les mises à jour.
Cette approche multi-maître fournit des overheads réduits et des garanties probabilistes au lieu d'une cohérence stricte. De plus, elle ignore les conflits supposés assez rares.

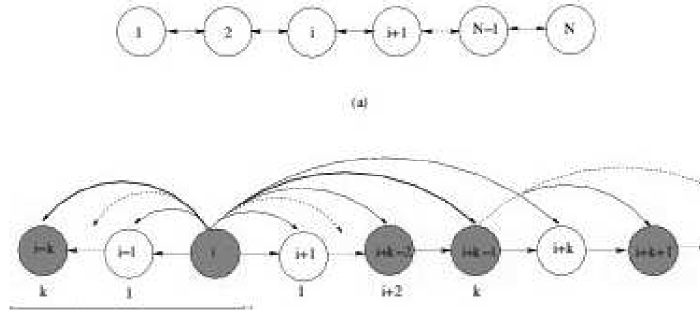

FIG. 18 – (a) Une chaîne logique de réplicas ; (b) Propagation d'une mise à jour d'un site i

Approche	Architecture	Type de réplication	Diffusion m.à.j.
Freenet	struturé(DHT)	mono-maître	heuristique basée sur clés
OceanStore	structuré(Tapestry)	inner ring et subordonnés	épidémique
P-Grid	structuré(DHT)	mono-maître	push-pull
UPTReC	non structuré	multi -maître	push-pull

TAB. 4 – Comparaison (1) entre des approches de réplication en P2P

Approche	Détection de conflits	Résolution de conflits	Cohérence
Freenet	nouvelle version	-	incertaine
OceanStore	prédicas(Bayou)	ordonnancement par inner ring	éventuelle
P-Grid	-	-	probabiliste
UPTReC	-	-	probabiliste

TAB. 5 – Comparaison (2) entre des approches de réplication en P2P

6.6 Comparaison entre diverses approches

Les tableaux 4 et 5 comparent les diverses solutions de réplication pour un système P2P que nous avons abordées ci-haut.

Quatrième partie

Réplication optimiste dans APPA

7 Introduction

APPA [31], (*Atlas Peer-to-Peer Architecture*), est un système de gestion de données dans les environnements P2P. APPA vise les applications collaboratives avancées dans le contexte de communautés virtuelles. APPA a une architecture indépendante du réseau en terme de services basiques et avancés pouvant être implémentés sur différents réseaux P2P (structurés, DHT, super-pair).
Pour permettre la collaboration au sein d'un réseau P2P, APPA supporte des données riches en sémantiques et propose une solution de réplication optimiste [32] assurant une haute disponibilité : cette solution se base sur une réconciliation sémantique en vue de réduire les conflits ; elle exploite la distribution et le parallélisme pour un passage à l'échelle et une réduction des latences ; elle prend en compte le dynamisme des pairs .
Dans cette partie, nous présentons l'architecture de APPA, puis un algorithme distribué pour la réconciliation sémantique des mises à jour conflictuelles, le *DSR (distributed semantic reconciliation).* Ensuite, nous addressons un protocole de réconciliation, pour les larges collaborations P2P dans un réseau DHT Chord.

8 Architecture de APPA

L'architecture de APPA [31] est organisée suivant trois couches de services comme le montre la figure 19. La couche *advanced services* fournit des services avancés pour le partage de données riches en sémantiques. La couche *basic services* fournit des services élémentaires qui supportent les services avancés, en utilisant la couche *P2P network.* Cette couche assure l'indépendance réseau par l'implémentation de services communs à tous les réseaux P2P ; ainsi, divers réseaux P2P peuvent être combinés en vue de profiter des avantages de chacun.
Le service de réplication [32] est un exemple de service avancé de APPA assurant une réplication multi-maître dans un environnement P2P. Les données manipulées par ce service sont gérées par le service basique *P2P data management service* (P2PDM) et par le service *Key-based storage and retreival* (KSR) de la couche P2P network. KSR s'occupe du stockage et de l'accès aux données en se basant sur leurs clés ; il implémente pour cela des opérations élémentaires comme read, write, et delete dont la composition forme des procédures ou fonctions implémentées par le P2PDM. Ces opérations et fonctions manipulent les clés.
Ainsi, les deux services P2PDM et KSR fournissent ensemble un espace de stockage commun aux pairs et utilisé par le service de réplication, le *common storage.* Ce common storage permet d'accéder toute donnée produite par un pair même si ce dernier est déconnecté. On définit *P2P data* comme étant les données produites et exploitées par les services de APPA et on suppose que chaque pair a les capacités nécessaires pour la gestion de ses données locales.

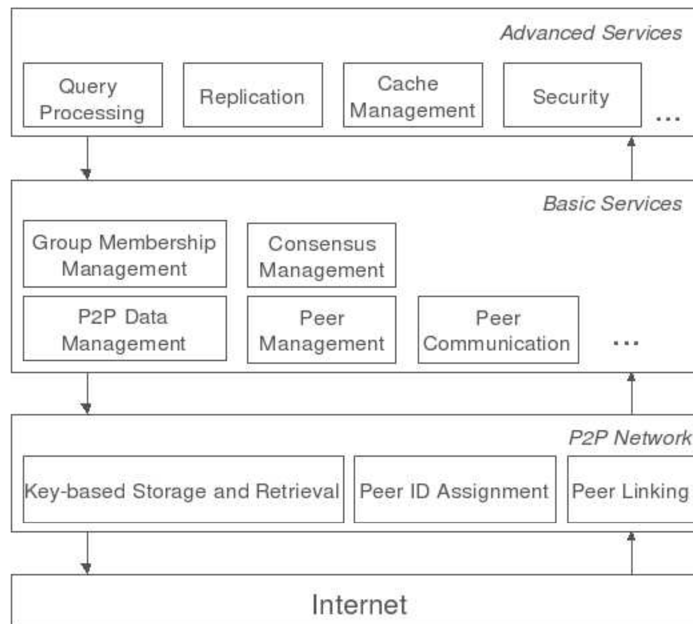

FIG. 19 – architecture d'APPA

9 Approche de réconciliation

L'approche proposée par APPA porte sur une réconciliation sémantique inspirée de IceCube. Cependant, elle se base sur un mécanisme totalement distribué évitant la réconciliation centralisée de IceCube et s'adaptant ainsi aux environnements P2P : cette approche garantit la vivacité face au dynamisme des pairs. Dans la suite, nous commençons par présenter les principes de base de l'algorithme DSR [33] sur lequel se base la réconciliation dans APPA [32], pour ensuite nous concentrer sur l'évaluation du DSR sur Chord [34].

9.1 Algorithme DSR

On distingue trois types de nœuds : les nœuds qui stockent des réplicas et effectuent des mises à jour et que l'on appelle *replica nodes* ; les nœuds qui participent à la réconciliation et que l'on appelle *reconciler nodes* ; les nœuds qui stockent et gèrent les P2P data générés ou utilisés par les reconcilers et que l'on appelle *provider nodes*.
Le processus basique peut être décrit de la façon suivante : dans une première phase, les *replica nodes* exécutent des actions locales tout en respectant les contraintes définies par l'utilisateur. Ces actions sont stockées avec leurs contraintes, en se basant sur l'identifiant du réplica concerné, dans le *common storage* implémenté par une DHT. Dans une deuxième phase, les *reconciler nodes* retirent les actions et leurs contraintes et performent une réconciliation

distribuée en se basant sur les sémantiques de l'application . Ils produisent un schedule global qui est localement exécuté sur chaque nœud assurant ainsi une éventuelle cohérence.

Considérons de nouveau l'exemple de la figure 20 qui représente 4 actions

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

FIG. 20 – Un exemple d'actions conflictuelles dans IceCube

conflictuelles produites par les nœuds n_1, n_2 et n_3 (avec a_n^i une action effectuée par le nœud n et ayant comme identifiant i). Nous allons utiliser cet exemple pour illustrer les explications qui suivent.

Les données manipulées durant la réconciliation sont portées par les *objets de réconciliation* qui sont stockés suivant leurs identifiants uniques dans leurs *provider nodes* correpondants [33]. On distingue les objets de réconciliation suivants :

- **Action Log R (L_R) :** cet objet contient toutes les actions tentatives qui mettent à jour le réplica R (en considérant une table relationnelle T, toute mise à jour d'un tuple de T effectuée par tout nœud est stockée dans L_T). Une action est stockée tout d'abord localement dans son *replica node* et puis dans le *provider node* portant l'*action log* correspondant ;
- **Cluster set (CS) :** conformément à IceCube,un cluster C_i contient un ensemble d'actions liées par des contraintes et peut être ordonné indépendamment des autres clusters. Tous les clusters produits durant la réconciliation sont contenus dans cet objet CS ;
- **Action summary (AS) :** cet objet capture les dépendances sémantiques entre les actions définies par les contraintes et décrit les relations entre les actions et les clusters (c.à.d les appartenances des actions aux clusters (a_n^i,C_j)) ;
- **Schedule (S) :** un schedule S est une liste d'actions ordonnées résultant de la concaténation S= $S_1 + S_2 + ... + S_n$ avec S_i la sous-liste des actions ordonnées provenant du cluster C_i ;

Tout nœud connecté peut déclencher la réconciliation en invitant d'autres nœuds disponibles à s'y engager. L'algorithme de réconciliation [33, 34] peut être divisée en 6 étapes distribuées (cf. figure 21) :

1. **allocation des nœuds :** un ensemble de *replica nodes* connectés est sélectionné pour jouer le rôle de reconcilers, de sorte que chaque étape soit exécutée par plusieurs reconcilers en parallèle. Des détails de cette étape sont présentés dans la section suivante ;
2. **regroupement des actions :** des reconcilers prennent les actions de l'action log et regroupent ceux qui mettent à jour le même élément d'un réplica (par exemple un tuple donné). Des groupes sont ainsi construits et stockés selon l'identifiant de leur élément de réplica d'une manière collaborative et parallèle par les noœuds participants. En prenant l'exemple de la figure 4.2, les groupes produits sont : :G_1= $\{a_1^1,a_2^1,a_3^1\}$, G_2= $\{a_3^2\}$;

3. **création des clusters :** l'application peut déterminer qu'une paire d'actions conflictuelles est sémantiquement indépendante comme c'est le cas de a_1^1 et a_3^1 qui modifient des attributs distincts et indépendants. Des reconcilers divisent chaque groupe en clusters d'actions sémantiquement dépendantes et les contraintes comme *mutuallyExclusive*(a_1^1,a_2^1) sont générées pour refléter les dépendances détectée et sont stockées dans l'AS avec les appartenances aux clusters. Les clusters créés et relatifs à l'exemple sont alors :C_1= $\{a_1^1, a_2^1\}$, C_2= $\{a_3^1\}$ et C_3= $\{a_3^2\}$;
4. **extension des clusters :** des reconcilers étendent les clusters déjà créés en leur ajoutant des actions conflictuelles selon les contraintes définies par l'utilisateur : dans notre exemple,en prenant compte de *parcel*(a_3^1,a_3^2), les clusters étendus sont $C_2 = C_2 \bigcup \{a_3^2\}$ et $C_3 = C_3 \bigcup \{a_3^1\}$.
5. **intégration des clusters :** des reconcilers unissent les clusters étendus ayant des actions en commun et produisent des clusters intégrés mutuellement indépendants. Comme résultat, on obtient $C_4 = C_2 \bigcup C_3$. Evidemment, l'AS est modifié selon les modifications opérées et les nouveaux clusters sont stockés dans le CS à la fin de chaque étape afin d'approvisionner les reconcilers de l'étape suivante ;
6. **ordonnancement des clusters :** des reconcilers prennent les clusters du CS et ordonnent les actions de chaque cluster indépendamment des autres en se basant sur les principes d'IceCube. Dans notre exemple, C_1 donne $S_1 = [a_1^1]$ puisque a_1^1 et a_2^1 sont mutuellement exclusifs alors que C_4 donne $S_2 = [a_3^1, a_3^2]$ puisque ses actions sont impliqués dans une contrainte parcel. Donc le schedule global exécuté par tous les replica nodes est $S = S_1 + S_2$;

Tout *reconciler node* est capable d'accéder à tout objet de réconciliation stocké dans son *provider node* correspondant au moyen de l'identifiant de l'objet. A chaque étape, quand un ensemble de reconcilers demandent des données à un *provider node*, celui-là partionne les données qu'il stocke selon le nombre de reconcilers. Ainsi, plusieurs nœuds performent simultanément des activités indépendantes en manipulant des sous-ensembles distincts de données.
Pour assurer une éventuelle cohérence en dépit des déconnexions dynamiques, chaque nœud stocke localement l'identifiant du dernier schedule qu'il a exécuté, noté S_{last}. En plus, on a recours à un nouvel objet de réconciliation *schedule history*, noté H, qui stocke une séquence chronologique des identifiants des schedules produits clôturée par S_{idn} . Quand un nœud déconnecté se reconnecte, il vérifie si son S_{last} est égal à S_{idn} ; si non, il applique localement tous les schedules suivant S_{last} dans H ; les actions qu'il a produites en déconnecté sont stockées dans l'*action log* pour une réconciliation ultèrieure.

9.2 DSR sur Chord

Dans l'algorithme DSR décrit ci-desus, on néglige le côut de communication entre les nœuds ; ce qui est inopportun dans les systèmes P2P construits généralement au-dessus de l'Internet. Dans ce cas, les coûts de réseau varient considérablement d'un nœud à l'autre et affectent fortement la performance de la réconciliation. Pour cela, un modèle de coût basé sur un réseau DHT est proposé dans [34] : il s'adapte au DSR et permet d'estimer le nombre optimal de reconcilers par étape et d'allouer les meilleurs nœuds.
Le modèle de coût dans un réseau DHT repose sur trois métriques : le coût

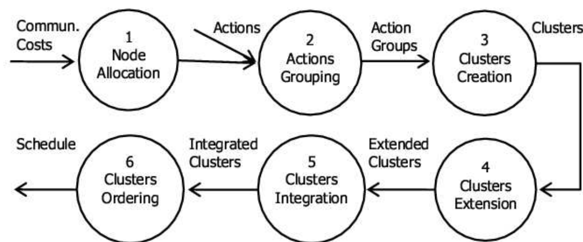

FIG. 21 – les étapes de la réconciliation sémantique distribuée

de recherche, le coût direct et le coût de transfert. Le *coût de recherche* est le temps de latence d'une recherche déclenchée par un nœud n_i pour trouver une donnée d identifiée par son ID et stockée dans un nœud n_j . Le *coût direct* est le temps de latence de la communication directe avec le nœud concerné n_j. Le *coût de transfert* est le temps de transfert de la donnée d de n_j vers n_i. Les deux premiers coûts changent dynamiquement tant que des nœuds se joignent ou quittent le réseau puisque le voisinage des nœuds ainsi que le routage changent ; ils doivent par suite être rafraîchis. Le coût de transfert change aussi puisqu'il varie avec la bande passante ou le débit entre les nœuds.
Cette approche suggère que chaque nœud calcule son coût de recherche incrémentalement en exploitant l'information de ses voisins relative à leur coût afin d'éviter de surcharger le réseau et d'encombrer certains nœuds. En imposant à chaque nœud de garder les coûts de 4 IDs (c.à.d. un pour chaque objet de réconciliation), cette approche est faisable puisque dans un réseau DHT, un nœud cherche un ID en communiquant avec ses voisins les plus proches de cet ID. Quant au coût direct, on peut soit le calculer exactement, soit l'estimer en prenant un ensemble de nœuds et en calculant la valeur moyenne correspondante.
L'étape d'allocation de la réconciliation introduit un compromis. L'augmentation du nombre de reconcilers réduit le temps de réponse mais augmente le nombre de messages échangés et la charge sur les *provider nodes*. Donc, au delà d'un certain nombre de reconcilers, le temps de réponse augmente. Pour cela, une méthode mathématique permet au nœud déclenchant la réconciliation de calculer le nombre maximal de reconcilers par étape [34].
On définit un nouvel objet de réconciliation, *communication costs* noté par CC, qui stocke pour chaque *replica node* l'estimation du total des coûts dûs à l'exécution de chaque étape. Cet objet est stocké par le *cost provider* auquel chaque *replica node* fournit son coût total. A la demande du nœud initiateur, le *cost provider* sélectionne alors les meilleurs nœuds pour chaque étape et le leur notifient.

9.3 Résultats expérimentaux

Nous présentons en résumé une évaluation de la performance du protocole de réconciliation basé sur le modèle de coût dréssé ci-haut.
La performance est mesurée en termes de temps de réconciliation dans plusieurs expériences [34] comparant des méthodes d'allocation ou de sélection de reconcilers. En variant la valeur moyenne des latences entre les nœuds ou en aug-

mentant le nombre d'actions, une approche basée sur le coût reste plus efficace qu'une approche arbitraire. De plus, elle apporte une amélioration considérable de la perforamance (par un facteur de 26) avec la variation du débit. En comparaison avec une approche centralisée assurant plus d'efficacité mais moins de disponibilité, ce protocole fournit une haute disponibilité et un excellent passage à l'échelle avec une performance acceptable et un overhead limité. Il assure une propriét'e importante, la vivacité, qui est fortement requise dans les systêmes distribués.

Cinquième partie

Conclusion et Perspective

Dans ce mémoire, nous avons dressé un état de l'art sur la réplication, que nous avons illustré par des travaux menés dans ce domaine.
Nous nous sommes particulièrement intéressés aux applications collaboratives asynchrones dans le contexte des environnements ponctuellement connectés. Nous avons vu que le mode de réplication adapté à ces applications est la réplication optimiste qui assure une haute disponibilté et une performance maximale en relâchant les contraintes sur les concurrences d'accès et en acceptant a priori toutes les écritures.
Des conflits peuvent résulter entre des actions tentatives parallèles provenant de différents sites, ce qui peut provoquer des divergences entre les données. Les protocoles de réplication doivent alors intègrer des mécanismes de réconciliation des donnés répliquées en vue d'assurer une cohérence éventuelle. Les approches abordées se basent soit sur les sémantiques de l'application soit sur l'estampillage ou l'ordonnancement des actions pour détecter et/ou résoudre les conflits.
La journalisation des actions permet de gérer les échanges entre les sites ainsi que de détecter les conflits en présence de mises à jour conflictuelles.
La diffusion des mises à jour vise à assurer la propagation éventuelle des actions entre les sites. Des échanges incrémentaux peuvent être fournis par l'utilisation des vecteurs de versions améliorant ainsi la performance. Dans ce contexte, les potocoles épidémiques sont souvent utilisés puisqu'ils requièrent peu de garanties du réseau tout en assurant une garantie probabiliste de cohérence éventuelle.
D'autre part, la réplication partielle s'avère être une solution intéressante pour optimiser les échanges et réduire la consommation de bande passante et d'espace de stockage mais elle est faiblement abordée par les approches présentées dans notre étude.

De nos jours, l'émergence des applications collaboratives au sein des réseaux P2P est en pleine croissance mettant en jeu un partage intensif de donnée riches en sémantiques. Vue la popularité croissante des systèmes P2P, des chercheurs s'acharnent à les doter de technologies plus performantes afin de pouvoir profiter de leurs propriétés convoitées et essentiellement de la disponibilité et du passage à l'échelle. Nous avons exposés les défis de construction d'un système P2P visant ces collaborations avancées. Le défi majeur est de garantir la vivacité face au dynamisme des pairs.
Nous avons présenté les différentes architectures de P2P avec les avantages et les limitations de chacune et nous nous sommes attardés sur l'architecture structurée en DHT qui gère le placement des données, puisque nous comptons l'exploiter dans notre travail futur.
La réplication optimiste satisfait nos ambitions concernant la disponibilité requise. Les approches centralisées sont inadéquates aux P2P, étant donné leur faible disponibilité ainsi que leur vulnérabilité aux pannes et aux partitions du réseau. La plupart des approches abordées ne supportent pas des données riches en sémantiques et se limitent aux fichiers.

Cet état de l'art est le premier pas vers notre solution qui a pour but de trouver une stratégie de réconciliation qui soit à la fois efficace, décentralisée et résistante aux pannes et qui convient aux systèmes P2P.

Notre objectif est de pouvoir établir une stratégie de réconciliation des données répliquées dans des environnements P2P tout en assurant une haute disponibilié, un passage à l'échelle et une vivacité face au dynamisme des pairs. De plus, nous cherchons à maximiser les performances de la réconciliation. Nous visons les applications collaboratives exploitant des données riches en sémantique dans le contexte de communautés virtuelles.
Le protocole de réconciliation basé sur un modéle de coût [34] satisfait nos exigences en termes de passage à l'échelle, de disponibilié et de vivacité. Cependant, il est évalué sur Chord qui ne considère pas les distances physiques entre les nœuds et choisit les voisins d'un nœud en se basant uniquement sur les clés stockées par les pairs. Or, une amélioration de la performance peut être réalisée si les communications se font entre des pairs physiquement proches : les coûts en termes de temps de latence et de transfert peuvent diminuer.
Nous comptons étudier le comportement du protocole sur un réseau DHT *location-aware* comme CAN. En effet, une optimisation de CAN [25], vise à construire sa topologie de façon qu'elle soit congruente à la topologie physique sous-jacente. On permet à plusieurs pairs de partager la même zone de l'espace en répliquant le contenu de la table de hachage attribuée à cette zone. Pour un nœud appartenant à une zone voisine, on choisit le pair le plus proche physiquement. Ainsi, les voisins dans l'espace de coordonnées peuvent bien être des voisins sur l'Internet.
Néanmoins, la proximité physique des pairs voisins doit s'intégrer dans les calculs des coûts et peut introduire des complications.

Références

[1] O. Dedieu. *Réplication optimiste pour les applications collaboratives asynchrones.* PhD thesis, Université de Marne la Vallée, 2000.

[2] E. Pacitti and O. Dedieu. Optimistic replication in pharos, a collaborative application on the web. *Journal of the Brazilian Computer Society*, 8(2) :7–14, 2002.

[3] L. Kawell, S. Beckhardt, T. Halvorsen, and R. Ozzie. Replicated document management in a group communication system. In *2nd Int. Conf. on Comp.-Supported Coop. Work*, pages 205–216, 1988.

[4] E. Pacitti, C. Coulon, P. Valduriez, and T. Ozsu. Preventive replication in a database cluster. In Distributed and Parallel Databases, to appear.

[5] T. Ozsu and P. Valduriez. *Principles of distributed database systems.* Prentice Hall, 2nd edition, 1999.

[6] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated database. *The VLDB Journal*, 8 :305–318, 2000.

[7] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters : Early results.

In *Proc. of the IEEE 22nd Int. Conf. on Dist. Computing Sys*, Vienna, Austria, 2002.

[8] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37 :42–81, 2005.

[9] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc.of the 8th Int. Conf. on Principles of Dist. Sys.*, Grenoble, France, 2004.

[10] A. Prakash and M. J. Knister. Undoing actions in collaborative work. In *Proc. of the ACM Conf. on Computer Supported Cooperative Work*, pages 273–280, Toronto, Ontario, Canada, 1992.

[11] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Proc. Conf. on Cooperative Information Systems*, Catalania, Italy, 2003.

[12] D. Terry, M. Theimer, K. Petersen, M. Spreitzer, A. Demers, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symp. on Op. Sys. Principles*, volume 29, 1995.

[13] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 :558–565, 1978.

[14] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of 16th ACM Symp. on Op. Sys. Principles*, volume 31, pages 288–301, 1997.

[15] M.Satyanarayanan. Coda : a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4) :447–459, 1990.

[16] P. Kumar and M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proc. of the 2nd Int. Conf. on Parallel and Distributed Info. Sys.*, pages 203–213, 1993.

[17] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proc. Usenix Winter Tech. Conf.*, pages 95–106, New Orleans, LA, USA, 1995.

[18] A. Demers, D. Green, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proc. of the 6th ACM Symp. on Princ. of Dist. Comp.*, Vancouver, British Colombia, Canada, 1987.

[19] M. Lin and K. Marzullo. Directional gossip : Gossip in a wide area network. In *Third European Dependable Computing Conf.*, 1999.

[20] P. Reither, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the ficus file system. In *Usenix Summer Tech. Conf.*, pages 183–195, 1994.

[21] T. Page, R. Guy, J. Heidemann, D. Ratner, P. Reither, A. Goel, G. Kuenning, and G. Popek. Perspectives on optimistically replicated p2p filing. *Software-Practice and Experience*, 11 :155–180, 1997.

[22] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems, concepts and design*, chapter 10. Addison Wesley, 4th edition, 2001.

[23] P. Valduriez and E. Pacitti. Data management in large-scale p2p systems. In *Int. Conf. on High Performance Computing for Computational Science*, pages 109–122, Springer, 2005.

[24] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with freenet. *IEEE Internet Computing*, 2002.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of SIGCOMM*, 2001.

[26] R. Morris, M. F. Kaashoek, D. Karger, H. Balakrishnan, I. Stoica, D. Liben-Nowell, and F. Dabek. Chord : a scalable peer-to-peer lookup protocol for internet applications. In *Proc. of the 2001 ACM SIGCOMM Conf.*, pages 149–160, 2001.

[27] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore : an architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, 2000.

[28] K. Aberer, P. Cudre-Mauroux, A. Datta, Z. Despovitch, M. Hauswirth, M. Punceva, and R. Schmidt. P-grid : a self-organizing structured p2p system. In *SIGMOD Record*, pages 29–33, 2003.

[29] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable replicated p2p system. In *Proc. of IEEE ICDCS*, pages 76–85, 2003.

[30] Z. Wang, S. K. Das, M. Kumar, and H. Shen. Update propagation through replica chain in decentralized and unstructured p2p systems. In *The 4th IEEE Int. Conf. on Peer-to-Peer Computing*, 2004.

[31] R. Akbarinia, V. Martins, E. Pacitti, and P. Valduriez. Replication and query processing in the appa data management system. In *Int. Workshop on Distributed Data and Structures, Lausanne*, 2004.

[32] V. Martins, E. Pacitti, and P. Valduriez. Reconciliation in the appa p2p system. In *Journées Masses de Données en Pair-à-Pair*, 2005.

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

[34] V. Martins, E. Pacitti, R. Jimenez-Peris, and P. Valduriez. Highly available semantic reconciliation for p2p networks. Submitted to the 32nd VLDB Conf.