HAL
open science

Infrastructure P2P pour la Réplication et la Réconciliation des Données

Mounir Tlili

To cite this version:

Mounir Tlili. Infrastructure P2P pour la Réplication et la Réconciliation des Données. Base de données [cs.DB]. Université de Nantes, 2011. Français. NNT : . tel-00643789

HAL Id: tel-00643789

https://theses.hal.science/tel-00643789v1

Submitted on 22 Nov 2011

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

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

UNIVERSITE DE NANTES
FACULTE DES SCIENCES ET DES TECHNIQUES

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

Année 2011

N° attribué par la bibliothèque

Infrastructure P2P pour la Réplication et la Réconciliation des Données

THESE DE DOCTORAT
Discipline : Informatique
Spécialité : Bases de Données

Présentée
Et soutenue publiquement par

MOUNIR TLILI

Le 30 Juin 2011, devant le jury ci-dessous

Président	Pascal Molli, Professeur, Université de Nantes
Rapporteurs	Philippe Pucheral, Professeur, Université de Versailles St-Quentin
	Stéphane Gançarski, Maître de Conférence HDR, UPMC Paris 6
Examinateurs	Patrick Valduriez, Directeur de Recherche, INRIA
	Claudia Roncancio, Professeur, Institut Polytechnique de Grenoble
	Pascal Molli, Professeur, Université de Nantes
	Esther Pacitti, Professeur, Université de Montpellier
	Reza Akbarinia, Chargé de recherche, INRIA

Directeur de thèse : Esther Pacitti
Encadrant de thèse : Reza Akbarinia

ED 503-131

P2P Infrastructure for Data Replication and Reconciliation

Abstract. In this thesis, we address the problem of optimistic replication for collaborative text editing in Peer-to-Peer (P2P) systems. This problem is challenging because of concurrent updating at multiple peers and dynamic behavior of peers. Operational transformation (OT) is a typical approach used for handling optimistic replication in the context of distributed text editing. However, most of OT solutions are neither scalable nor suited for P2P networks due to the dynamic behavior of peers. In this thesis, we propose a scalable P2P reconciliation infrastructure for OT that assures eventual consistency and liveness despite dynamicity and failures. We propose a P2P logging and timestamping service, called P2P-LTR (P2P Logging and Timestamping for Reconciliation) that exploits a distributed hash table (DHT) for reconciliation. While updating replica copies at collaborating peer editors, updates are stored in a highly available P2P log. To enforce eventual consistency, these updates must be retrieved in a specific total order to be reconciled at the peer editors. P2P-LTR provides an efficient mechanism for determining the total order of updates. It also deals with the case of peers that may join and leave the system during update operations. We evaluated the performance of P2P-LTR through simulation; the results show the efficiency and the scalability of our solution.

Keywords: Optimistic replication, reconciliation, P2P systems, distributed hash tables (DHT), collaborative text editing

Infrastructure P2P pour la Réplication et la Réconciliation des Données

Résumé. Dans notre thèse, nous nous intéressons à la construction d'une infrastructure Pair-à-Pair (P2P) pour la réconciliation des données des applications d'édition de texte collaborative. Cependant, cette tâche est difficile à réaliser étant donné le comportement dynamique des pairs. Au regard de l'état de l'art, le modèle des transformées opérationnelles (OT) est une approche typiquement utilisée pour la gestion de la réplication optimiste dans le contexte d'édition de texte distribuée. Toutefois, la plupart des solutions d'OT ne passent pas à l'échelle et ne sont pas adaptées aux réseaux P2P. Pour répondre à ce problème, nous proposons une nouvelle approche appelée P2P-LTR (Estampillage et Journalisation P2P pour la Réconciliation) pour la réconciliation des données à base d'OT, qui assure la cohérence à terme malgré la dynamicité et les cas de pannes. P2P-LTR offre un service de journalisation P2P et un service d'estampillage fiable et réparti fonctionnant sur un modèle de réseau à base de DHT. Dans notre approche, les mises à jour sont estampillées et stockées en P2P dans des journaux à forte disponibilité. Lors de la réconciliation, ces mises à jour sont récupérées selon un ordre total continu afin d'assurer la cohérence à terme. En outre, P2P-LTR traite les cas où les pairs peuvent rejoindre ou quitter le système pendant les opérations de mise à jour. Nous avons évalué les performances de P2P-LTR par simulation. Les résultats montrent l'efficacité et le passage à l'échelle de notre solution.

Mots clés: Réplication Optimiste, Réconciliation, Système Pair-à-Pair, DHT, Edition Collaborative.

Discipline: Informatique N° : ED 503-131

Année 2011

Infrastructure P2P pour la Réplication et la Réconciliation de Données

THESE DE DOCTORAT
Discipline : Informatique
Spécialité : Bases de Données

Présentée
Et soutenue publiquement par

MOUNIR TLILI

REMERCIEMENTS

Au terme de ce travail, je tiens à assurer de ma profonde reconnaissance tous ceux dont j’ai eu à solliciter la compétence et l’expérience.

Je voudrais d’abord exprimer ma reconnaissance envers tous les membres du jury pour la grande attention qu’ils ont bien voulu porter à mon travail.

Je suis particulièrement reconnaissant à Patrick Valduriez qui m’a accueilli au sein de l’équipe ATLAS-GDD. Je le remercie pour le soutien qui m’a donné tout au long de cette thèse.

Je suis très reconnaissant à ma directrice de thèse, Esther Pacitti et mon encadrant Reza Akbarinia, qui m’ont encadré et dirigé dans mes recherches tout au long de ces années. Je leurs exprime ma gratitude pour l’aide compétente qu’ils m’ont apportée, pour leurs encouragements et pour la confiance qu’ils m’ont toujours témoignée.

Je tiens à remercier aussi tous les membres de l’équipe ATLAS-GDD, avec qui j’ai eu le plaisir de travailler, je pense particulièrement à Philippe, Pascal, Patricia, Sylvie, Gerson, William, Toufik, Thomas, Mohamed, Rabab, Manal, Eduardo, Wence, et Jorge.

Mes remerciements vont aussi à tous les personnels du LINA qui ont su être toujours présents au besoin, et pour la relation amicale qu’on a tenu.

A tous ceux que j'ai oublié de citer, qu'ils m'en excusent. Enfin, mes plus chaleureux remerciements vont également à mes parents, mes frères, mes sœurs et mes amis pour leur soutien et leurs encouragements continus tout au long de mes études.

TABLE DES MATIERES

TABLE DES FIGURES **10**

LISTE DES TABLEAUX **12**

I INTRODUCTION **13**

1 CONTEXTE 13
2 PROBLEMATIQUE 15
3 RESUME DES CONTRIBUTIONS 16
4 ORGANISATION DU DOCUMENT 19

II ETAT DE L'ART **20**

1 REPLICATION DES DONNEES 20
1.1 Les bases de données distribuées et répliquées *20*
1.2 Mécanismes de la réplication *21*
1.2.1 Réplication simple-maître (single master) 21
1.2.2 Réplication multi-maître (multi master) 22
1.2.3 Réplication Synchrone et Asynchrone 22
2 REPLICATION DANS LES SYSTEMES DISTRIBUES 23
2.1 Paxos : exemple d'un protocole de consensus *24*
2.2 Réalisation d'un consensus dans les systèmes asynchrones *24*
3 REPLICATION OPTIMISTE ET DIVERGENCE 25
3.1 Réconciliation *25*
3.1.1 CVS 25
3.1.2 IceCube 26
3.1.3 Bayou 27
3.1.4 Transformée Opérationnelle (OT) 28
3.1.5 So6 32
3.1.6 Google Wave 34
3.2 Synthèse Générale *34*
4 LES SYSTEMES P2P 35
4.1 Caractéristiques *35*
4.1.1 Le passage à l'échelle 36
4.1.2 La tolérance aux fautes 36
4.1.3 La dynamicité 36
4.1.4 L'auto-organisation 36
4.1.5 Un réseau virtuel 36
4.2 Types de réseaux P2P *37*
4.2.1 Les réseaux non-structurés 38
4.2.2 Les réseaux structurés 40
4.2.3 Réseau super-pair 45
4.2.4 Synthèse générale 46
4.3 Solutions de réplication dans les réseaux P2P *47*
4.3.1 Freenet 47
4.3.2 OceanStore 48
4.3.3 Ivy 50
4.3.4 P-Grid 51
4.3.5 WOOT 53
4.3.6 Logoot 54
4.3.7 DSR-P2P 55
4.3.8 Telex 57
4.3.9 Scalaris 59
4.3.10 MOT2 59
4.3.11 KTS 60

4.3.12 Synthèse générale 60
5 CONCLUSION 61

III P2P-LTR: MODELE ET ALGORITHMES 63

1 DEFINITION DU PROBLEME ET SOLUTION GENERALE 64
1.1 Définition du problème 64
1.2 Solution générale 65
2 MODELE DU P2P-LTR 66
2.1 Description 66
2.2 Opérations 68
2.3 Algorithmes 69
2.4 Cohérence à terme 72
2.5 Exemple 73
3 GESTION DU COMPORTEMENT DYNAMIQUE DES PAIRS 75
3.1 Arrivée et départ du Master-key 75
3.1.1 Arrivée d'un nouveau Master-key 76
3.1.2 Départ du Master-key 78
3.2 Tolérance aux fautes 80
3.2.1 Gestion de la défaillance du Master-key 81
3.2.2 Gestion de la défaillance simultanée du Master-key et Master-key-Succ 83
3.3 Propriété de vivacité 84
4 ANALYSE DES COUTS 85
4.1 Complexité de la communication pour publier un patch 86
4.2 Complexité de la communication pour la convergence 87
4.3 Latence pour publier un patch 88
4.4 Latence pour la convergence 88
4.5 Synthèse 89
5 ANALYSE PROBABILISTE 89
6 COMPARAISON AVEC LES APPROCHES EXISTANTES 91
7 CONCLUSION 93

IV EVALUATION DE PERFORMANCES 94

1 ENVIRONNEMENT DE SIMULATION 94
2 RESULTATS 95
2.1 Passage à l'échelle 95
2.2 L'effet du nombre de répliques sur le temps de réponse 97
2.3 L'effet de la fréquence des mise à jour sur le temps de réponse 98
2.4 L'effet des pannes sur la continuité des estampilles 99
3 CONCLUSION 100

V MISE EN ŒUVRE ET VALIDATION 102

1 XWIKI: UN WIKI FONCTIONNANT SUR UN RESEAU P2P 102
2 ARCHITECTURE XWIKI P2P 103
2.1 Description des composants 104
2.1.1 PatchManager 104
2.1.2 ReconcileManager 105
2.1.3 P2P-LTR 105
2.2 Interface des composants 106
2.2.1 IPatchManager 106
2.2.2 IReconcileManager 107
2.2.3 IP2P-LTR 108
2.3 Implémentation des composants 109
2.3.1 Implémentation de PatchManager 109
2.3.2 Implémentation de ReconcileManager 110
2.3.3 Implémentation de P2P-LTR 111
3 DEPLOIEMENT 115
3.1 Déploiement de la DHT + P2P-LTR 116
3.2 Déploiement des serveurs XWiki avec PatchManager 116

3.3 Déploiement des composants ReconcileManager *117*
3.4 Connexion de ReconcileManager avec un composant PatchManager *117*
3.5 Connexion des ReconcileManager avec les composants P2P-LTR *117*
3.6 Démarrage des composants ReconcileManager et P2P-LTR *118*
4 CONCLUSION 118

VI BILAN ET PERSPECTIVES 119

1 CONTRIBUTIONS 120
1.1 P2P-LTR *120*
1.2 XWiki: un Wiki P2P *121*
2 TRAVAUX FUTURS 121
2.1 Authentification et contrôle d'accès aux ressources *121*
2.2 Un Wiki sémantique sur un réseau P2P *122*
2.3 Amélioration de XWiki *122*
2.4 Le Cloud : compromis entre cohérence et disponibilité *122*

ANNEXES 124

BIBLIOGRAPHY 131

TABLE DES FIGURES

Figure 1. Réplication simple-maître 21
Figure 2. Réplication multi-maître 21
Figure 3. Principe de la propagation synchrone 22
Figure 4. Principe de la propagation asynchrone 23
Figure 5. Processus de réconciliation dans IceCube. 27
Figure 6. Divergence des copies due à la violation de la condition C1 29
Figure 7. Fonction de transformations naïve 29
Figure 8. Fonction de transformation vérifiant la condition C1 30
Figure 9. Fonction de transformation vérifiant la condition C1 31
Figure 10. Insuffisance de la condition C1 31
Figure 11. Architecture Générale de So6 32
Figure 12. Intégration d'opération dans So6 en utilisant l'algorithme SOCT4 33
Figure 13. P2P vs Client/Serveur 35
Figure 14. P2P overlay au dessus d'une infrastructure d'Internet 37
Figure 15. Classifications des réseaux P2P 38
Figure 16. Les échanges dans Napster 39
Figure 17. Extrait de [SMKK+01]. (a) Acheminement d'une requête par parcours de l'anneau. (b) Définition des fingers d'un pair. (c) Acheminement d'une requête en utilisant les fingers. 41
Figure 18. Extrait de [RD01]. Table de routage d'un pair d'identifiant 10233102 avec b=2 44
Figure 19. Exemple d'espace à deux dimensions avec 5 pairs (Extrait de [FHKS+01]) 44
Figure 20. Topologie P2P construite selon le modèle super-pair 45
Figure 21. Le parcours d'une mise à jour dans OceanStore 49
Figure 22. Un exemple de vue dans Ivy 50
Figure 23. Equilibrage dans un arbre équilibré 52
Figure 24. Equilibrage dans un arbre non équilibré 53
Figure 25. Exemple de génération d'un ordre partiel dans WOOT 54
Figure 26. Les étapes de DSR-P2P 56
Figure 27. Stockage de document dans Telex 58
Figure 28. Architecture de Telex 59
Figure 29. Le modèle de P2P-LTR 67
Figure 30. P2P-LTR: les opérations 69
Figure 31. P2P-LTR: Algorithme de réplication de patches 70
Figure 32. P2P-LTR: Algorithme de réplication de patches 71
Figure 33. Pseudo code de la fonction chargée du transfert des clés: cas de join 77
Figure 34. P2P-LTR: configuration initiale 78
Figure 35. Configuration après le join de P0 78
Figure 36. Fonction pour le maintien de la liste de clés en cas de départ 79
Figure 37. P2P-LTR face au départ d'un Master-key 80
Figure 38. Automate à états finis décrit le comportement d'un Master-key 83
Figure 39. Gestion de pannes du Master-key et son successeur 84
Figure 40. Temps de réponse/Nombre d'utilisateurs par document 95
Figure 41. Temps de réponse/Nombre de pairs 96
Figure 42. Nombre total de messages/Nombre d'utilisateurs par document 97
Figure 43. Temps de réponse/Nombre de répliques 98
Figure 44. Temps de publication de patches/Fréquence des mises à jour 99
Figure 45. Continuité des estampilles/Taux d'échec 100
Figure 46. Architecture XWiki P2P 103
Figure 47. Interfaces XWiki-PatchManger/ ReconcileManager/P2P-LTR 106
Figure 48. Interface PatchManager 106
Figure 49. Interface ReconcileManager 107

Figure 50. Déconnexion du réseau P2P-LTR.....107
Figure 51. Déconnexion du fournisseur de contenu XWiki.....108
Figure 52. Interface P2P-LTR.....109
Figure 53. Exemple simple d'un patch généré sur un document XWiki.....110
Figure 54. Architecture interne du composant ReconcileManager.....111
Figure 55. Architecture interne du composant P2P-LTR.....112
Figure 56. Interface principale du prototype P2P-LTR.....114
Figure 57. Schéma de déploiement.....115
Figure 58. Exemple de fonction de transformation dans So6.....128
Figure 59. Exemple de fonction de transformation dans P2P-LTR.....130

LISTE DES TABLEAUX

Tableau 1. Comparaison des solutions de réplication optimiste *34*
Tableau 2. Comparaison des systèmes P2P *46*
Tableau 3. Comparaison des solutions de réplication dans le système P2P *61*
Tableau 4. Description et rôle des nœuds de modèle P2P-LTR *68*
Tableau 5. Scénario de collaboration de deux pairs *75*
Tableau 6. Les paramètres de simulation *95*
Tableau 7. Composant PatchManager *104*
Tableau 8. Composant ReconcileManager *105*
Tableau 9. Composant P2P-LTR *105*

CHAPITRE 1

I Introduction

Dans ce chapitre, nous présentons le contexte et la problématique de notre travail de thèse. D'abord, nous commençons par donner le contexte et les enjeux qui ont motivé notre travail. Ensuite, nous discutons de la problématique de notre sujet de thèse, puis nous présentons un résumé de nos contributions. Enfin, nous exposons le plan du rapport.

1 Contexte

Les systèmes pair-à-pair (P2P) sont en plein essor depuis plusieurs années. Ce paradigme permet de concevoir des systèmes de très grande taille à forte disponibilité et à faible coût sans recourir à des serveurs centraux [VP04]. En effet, les réseaux P2P sont des réseaux overlay (c.-à-.d, des réseaux virtuels construits au dessus des réseaux physiques) [DM03], décentralisés, où tous les nœuds, appelés pairs, jouent à la fois le rôle de serveur et de client. L'organisation dans un tel réseau repose sur l'ensemble des pairs : idéalement, il n'y a aucune entité chargée d'administrer le réseau. Par conséquent, les réseaux P2P garantissent le passage à l'échelle. Il y a trois grandes classes des réseaux P2P [MPV06, HSG08] : les réseaux non-structurés, les réseaux structurés et les réseaux super-pairs. Dans les réseaux non-structurés chaque pair est complètement autonome et la propagation des requêtes se fait généralement par inondation. Les mécanismes sont simples et flexibles mais posent des problèmes importants de performances et de passage à l'échelle. Les réseaux structurés organisent les pairs, ainsi que la répartition des objets sur les pairs, selon une structure stricte et efficace, notamment des tables de hachage distribuées (Distributed Hash Tables ou DHT) [SMKK+01]. Ces réseaux gagnent à la fois en efficacité et donnent des garanties de recherche. Le prix à payer est la perte d'autonomie de placement des données et le coût de maintenance élevé. Les réseaux super-pairs sont des réseaux hybrides entre les réseaux non structurés et le client-serveur. Comme le réseau client-serveur, certains pairs, appelés super-pairs, jouent le rôle de serveur pour un ensemble de pairs et effectuent des fonctions complexes comme le traitement des requêtes, le contrôle d'accès et la gestion des méta-données. Dans ce type de réseau, les super-pairs sont organisés en mode P2P, choisis automatiquement en fonction de leurs caractéristiques (bande passante, vitesse de traitement, capacité mémoire, etc.) et remplacés en cas de présence d'une panne.

Une classe d'applications importante dans les systèmes P2P concerne les applications collaboratives où un grand nombre d'utilisateurs doivent pouvoir travailler sur les mêmes objets (*ex.* documents, tables, etc.) en parallèle depuis leurs postes de travail. Un exemple d'une telle application est le Wiki, un outil d'édition de documents

multimédias en ligne avec lequel les utilisateurs peuvent facilement tisser des liens hypertextes entre les documents. Un Wiki permet la création et la modification de documents à travers une interface Web. Lors de la navigation dans le contenu d'un Wiki, un utilisateur peut à tout moment entrer en mode édition pour modifier le contenu de la page. Lorsqu'il termine, il peut sauvegarder le nouveau contenu qui vient alors remplacer l'ancienne valeur de la page. Cependant, l'architecture actuelle des Wikis est client-serveur : un serveur détient les données et les utilisateurs consultent et éditent les pages à travers un navigateur Web qui interagit avec le serveur. Cette architecture, malgré sa simplicité et sa popularité, a des limites. Par exemple, l'utilisation d'un serveur central est un point de faiblesse qui rend le système vulnérable aux pannes. L'exploitation d'une architecture P2P pour un Wiki permet de dépasser ces limites afin de passer à l'échelle en nombre d'utilisateurs, supporter une grande mobilité des clients et assurer l'élaboration et l'entretien des documents partagés de façon collaborative et asynchrone.

Cependant, une application Wiki P2P demande des capacités générales de réplication avec différents niveaux de granularité (ligne, mot, objet multimédia, etc.) et un mode multi-maître où plusieurs répliques d'une même donnée peuvent être mises à jour par plusieurs pairs en parallèle. La réplication multi-maître offre de bons avantages de performance et de disponibilité. Cependant, les mises à jour du même document par différents pairs peuvent créer des divergences des copies, comme l'introduction de nouvelles lignes ou d'images différentes par différents pairs. La solution de réplication optimiste [PD02] adresse ce problème en permettant aux répliques d'être mises à jour indépendamment et de rester divergentes jusqu'à une étape de réconciliation. Toutefois, cette étape de réconciliation devient critique car elle doit supporter l'autonomie des pairs et la très grande échelle du réseau P2P.

De nombreuses solutions, basées sur un mécanisme de réplication optimiste, ont été proposées afin de gérer la cohérence des données partagées. Cependant, les solutions optimistes existantes sont peu applicables aux réseaux P2P car elles sont centralisées ou ne tiennent pas compte des comportements dynamiques des pairs.

Le modèle des transformées opérationnelles (OT) [MOSM+03] est un modèle populaire pour mettre en place un mécanisme de réplication optimiste dans un contexte distribué. En effet, le modèle OT, n'exige pas de site central. L'architecture générale de ce modèle distingue deux composants, à savoir un algorithme d'intégration, responsable de la réception, de la diffusion et de l'exécution des opérations et un ensemble de fonctions de transformation en charge de fusionner les mises à jour en sérialisant les opérations concurrentes. Un aspect important d'OT est d'être indépendant du type de données manipulées (chaîne de caractères, document XML, système de fichiers, etc.). Cependant, afin d'assurer la cohérence des données, les opérations doivent respecter un critère bien défini : toutes les opérations doivent être estampillées selon un ordre total.

De nombreuses solutions, basées sur OT ont été proposées pour maintenir la cohérence des données. Certaines [MOSM+03, OMSI04] utilisent un estampilleur central pour maintenir un ordre total et assurer la convergence. Mais l'utilisation d'un estampilleur central limite le passage à l'échelle et peut bloquer le système en cas de

panne. En outre, la taille du journal d'opérations utilisé par OT au niveau du serveur d'estampilles a tendance à être importante et peut être non maîtrisable dans un seul nœud. D'autres solutions décentralisées [LL04a, LL04b] ont été proposées pour OT. Cependant, elles reposent sur l'utilisation de vecteurs d'horloge [Mat89] pour assurer la convergence des répliques. Cela suppose que chaque nœud du réseau connaît la taille du réseau, ce qui n'est évidemment pas le cas dans un réseau P2P. De plus, le "*churn"* ou l'agitation dans les réseaux P2P rend l'utilisation de ces techniques inadéquates. MOT2 [CF07] est la seule solution OT qui ne nécessite ni estampilleur central ni vecteur d'horloge afin d'assurer un ordre total sur les opérations. Cependant, lors de chaque synchronisation, il est nécessaire de transmettre toutes les opérations, et pas seulement les opérations manquantes. Par conséquent, la propagation des opérations entre les pairs devient lente et coûteuse.

D'autres solutions [WUM07, OUMS+05, WUM09] sont basées sur le modèle d'édition collaborative CRDT [SP07]. Dans cette approche, les modifications produites localement sont ré-exécutées sur des répliques distantes. Il n'y a pas d'ordre total sur les opérations, par conséquent, des opérations peuvent être exécutées dans différents ordres. Ces solutions ne nécessitent donc ni site central, ni ordre global et ni vecteur d'horloges. Par conséquent, elles devraient pouvoir être déployées sur un réseau P2P de grande taille. Cependant, ces solutions garantissent la convergence des répliques pour des données structurées linéaires et s'appuient sur un modèle spécifique des données. Par exemple dans [OUMS+05], un objet supprimé est marqué comme invisible au lieu d'être retiré du modèle du document. Ces objets marqués ne peuvent pas être supprimés sans compromettre la cohérence du document. Par conséquent, le surcoût nécessaire pour gérer le document grossit indéfiniment, ce qui limite le passage à l'échelle en nombre de modifications.

D'autres approches de réplication optimiste [BBMS+08, MAPV06] proposent un algorithme de réconciliation sémantique réparti pour maintenir la cohérence des données. Ces algorithmes utilisent le cadre *action-contrainte* pour supporter la sémantique de l'application et résoudre des conflits de mise à jour. Ces approches sont plus adaptées pour la réconciliation des données structurées.

Nous verrons, dans la suite de ce rapport l'approche employée pour dépasser ces limites.

2 Problématique

Le contexte de notre travail de thèse concerne les applications collaboratives P2P, notamment l'édition collaborative de texte dans un Wiki P2P. Dans notre travail, nous considérons les réseaux P2P structurés comme Chord [SMKK+01] ou Pastry [RD01]. Les pairs sont dynamiques, pouvant entrer ou quitter le réseau à tout moment.

Dans ce contexte, les utilisateurs partagent des documents de tout type (chaîne de caractères, XML, etc…). Beaucoup d'utilisateurs ont fréquemment besoin d'accéder et de mettre à jour des informations, même s'ils sont déconnectés du réseau, par exemple dans un avion, un train, ou un autre environnement qui ne fournit pas de communication

réseau appropriée. Ceci exige que les utilisateurs tiennent des répliques locales des documents partagés. Ainsi, une application de collaboration répartie a besoin de la réplication multi-maître pour assurer la disponibilité de données à n'importe quel moment. Dans l'approche multi-maître, les mises à jour faites en parallèle sur différentes répliques du même objet peuvent causer des divergences et des conflits parmi les répliques, qui doivent alors être réconciliés.

Le défi de notre travail de thèse est de proposer une infrastructure de réconciliation et de réplication multi-maître supportant la modification simultanée de différentes copies d'un même objet avec l'intégration des mises à jour parallèles. Une telle infrastructure permet d'améliorer la disponibilité de données.

Afin de garantir la cohérence à terme *(Eventual Consistency)* les états des répliques doivent converger de telle manière que si les utilisateurs cessent de soumettre des mises à jour, toutes les répliques obtiennent le même état final. Une telle solution doit être capable d'assurer la propagation et l'intégration des mises à jour de données sur les différentes répliques d'un même objet. Cette intégration doit se faire de manière totalement décentralisée, sans faire l'hypothèse de l'existence d'une copie de référence.

Motivée par ce besoin, cette thèse a pour objectif de concevoir, implémenter et évaluer une solution de réplication optimiste avec l'intégration de mises à jour adaptées aux besoins d'un éditeur collaboratif de type Wiki, fonctionnant sur un réseau P2P. Pour ce faire, nous proposons une infrastructure de réplication et de réconciliation qui assure la cohérence à terme parmi des répliques, en prenant en compte les comportements dynamiques des pairs et les cas de pannes.

3 Résumé des contributions

Ce travail a été effectué dans le contexte du projet ANR XWiki Concerto [XWIKI07], dont le principal objectif était de concevoir une architecture XWiki P2P pouvant passer à l'échelle en nombre d'utilisateurs, tout en supportant la mobilité des clients. Dans le contexte de ce projet, l'objectif de cette thèse est de fournir une nouvelle approche de réplication optimiste adaptée aux besoins d'un Wiki P2P. Nos contributions principales sont les suivantes.

D'abord, nous avons étudié de façon approfondie les techniques proposées pour la réplication et la réconciliation de données dans les systèmes P2P. Nous avons dégagé les insuffisances de ces techniques par rapport à notre problématique.

La seconde contribution est une solution efficace pour le problème de gestion des mises à jour concurrentes. Nous proposons une nouvelle approche, appelée P2P-LTR (Estampillage et Journalisation P2P pour la Réconciliation) [TDPV+08, TDPA+08] pour la réconciliation des données à base des transformées opérationnelles (OT) [MOSM+03], qui assure la cohérence à terme [SS05] malgré la dynamicité et les pannes des pairs. P2P-LTR offre un service de journalisation P2P et un service d'estampillage fiable et réparti, fonctionnant sur un modèle de réseau à base de DHT (Table de hachage distribuée) [SMKK+01]. Dans notre approche, les mises à jour sont estampillées et

stockées dans le réseau P2P au format des journaux à haute disponibilité. Lors de la réconciliation, les mises à jour sont récupérées selon un ordre total afin d'assurer la cohérence à terme. P2P-LTR traite les cas où les pairs peuvent rejoindre ou quitter le système pendant les opérations de mise à jour.

La troisième contribution est la validation de notre approche P2P-LTR par l'implémentation d'un prototype [TDPA+08] basé sur les DHT OpenChord [SMKK+01, KL07] et FreePastry [RD01,]. Notre approche est validée aussi par la réalisation d'un prototype de Wiki P2P comme XWiki, un Wiki en logiciel libre pour les entreprises. P2P-LTR est un logiciel libre, disponible à l'adresse suivante: http://p2pltr.gforce.inria.fr/.

Dans la quatrième contribution, nous prouvons théoriquement les propriétés de correction et de vivacité de P2P-LTR en présence de pannes [TAPV10]. Nous avons évalué les performances de notre solution à l'aide de la simulation, en utilisant PeerSim [Pee10]. Les résultats montrent l'efficacité et le passage à l'échelle de notre solution.

Ci-dessous, la liste des publications que nous avons publiées dans le cadre de cette thèse.

Revues internationales avec comite de lecture

1. R. Akbarinia, M. Tlili, E. Pacitti, P. Valduriez, A. A.B. Lima. Replication in DHTs using Dynamic Groups. *Journal of Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)*, Vol. 3, Springer, LNCS 6790, 1-19, 2011. Version étendue de [3].

Conférences internationales avec comite de lecture

2. M. Tlili, R. Akbarinia, E. Pacitti, P. Valduriez. Scalable P2P Reconciliation Infrastructure for Collaborative Text Editing. *Int. Conf. on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, IEEE, 155-164, 2010.

3. R. Akbarinia, M. Tlili, E. Pacitti, P. Valduriez, A. Lima. Continuous Timestamping for Efficient Replication Management in DHTs. *Int. Conf. on Data Management in Grid and P2P Systems (Globe)*, 38-49, 2010. Sélectionné parmi les deux meilleurs articles de Globe2010, pour une version étendue pour TLDKS [1].

4. G. Canals, P. Molli, J. Maire, S. Laurière, E. Pacitti, M. Tlili: XWiki Concerto: A P2P Wiki System Supporting Disconnected Work. *Int. Conf. on Cooperative Design, Visualization, and Engineering (CDVE)*, 98-106, 2008.

Conférences nationales avec comité de lecture

5. M. Tlili, R. Akbarinia, E. Pacitti, P. Valduriez. A Fault-Tolerant Infrastructure for P2P Collaborative Text Edition. *Journées Bases de Données Avancées (BDA)*, 2010.

Papiers Demo dans des conférences nationales et internationales

6. M. Tlili, W. K. Dedzoe, E. Pacitti, R. Akbarinia, P. Valduriez, et al. P2P Logging and Timestamping for Reconciliation. *Int. Conf. on Very Large Databases (VLDB)*, 1420-1423, 2008.

7. M. Tlili, W. K. Dedzoe, E. Pacitti, P. Valduriez, R. Akbarinia, et al. Estampillage et Journalisation P2P pour XWiki. *Int. Conf. on New Technologies of Distributed Systems (NOTERE)*, ACM, 197-200, 2008.

8. M. Tlili, W. K. Dedzoe, E. Pacitti, P. Valduriez, R. Akbarinia et al. Data Reconciliation in P2P Collaborative Applications. *24ièmes journées Bases de Données Avancées (BDA)*, 2008.

9. G. Canals, P. Molli, J. Maire, S. Laurière, E.Pacitti, M. Tlili, P. Valduriez. XWiki Concerto : un Wiki sur réseau P2P supportant le Nomadisme. *4èmes journées Francophones Mobilité et Ubiquité (Ubimob)*, 83-85, 2008.

Posters

10. M. Tlili, R. Akbarinia, E. Pacitti, P. Valduriez. Optimistic Replication for P2P Collaborative Text Editing. *Journées des doctorants (JDOC)*. 2010.

11. M. Tlili, W. K. Dedzoe, E. Pacitti, P. Valduriez, R. Akbarinia et al. XWiki concerto: a P2P wiki for nomadic workers. *Int. Symposium on Wikis*, *ACM*, 2008.

Rapports de recherche

12. M. Tlili, W. K. Dedzoe, E. Pacitti, R. Akbarinia, P. Valduriez. P2P Logging and Timestamping for Reconciliation. *Rapport de recherche INRIA N° 6497*, 2008.

4 Organisation du document

Le reste de notre document est organisé comme suit.

Dans le chapitre 2, nous présentons l'état de l'art. D'abord, nous nous intéressons aux différentes approches permettant de répliquer et réconcilier des copies divergentes, notamment dans les réseaux P2P. Ensuite, nous dégageons les insuffisances des approches étudiées par rapport à notre problématique.

Dans le chapitre 3, nous commençons par présenter notre nouvelle approche de réplication et de réconciliation de données dans un réseau P2P. Ensuite, nous proposons une solution complète pour supporter les comportements dynamiques des pairs. Puis, nous évaluons analytiquement le coût de notre approche en termes de nombre de messages nécessaires pour publier les mises à jours et converger vers un état final, et puis nous présentons une étude probabiliste de notre algorithme de réplication. Enfin, nous dressons un bilan de comparaison entre les approches retenues au chapitre 2 et notre solution P2P-LTR.

Dans le chapitre 4, nous évaluons les performances de notre solution à travers la simulation.

Nous présentons dans le chapitre 5 la mise en œuvre et la validation de notre solution par l'implémentation d'un prototype et l'intégration de notre solution dans un Wiki P2P. Le dernier chapitre décrit le bilan de notre travail et présente les perspectives envisageables.

CHAPITRE 2

II Etat de l'Art

Dans ce chapitre, nous nous intéressons aux solutions existantes pour la réplication optimiste et les stratégies de réplication dans les réseaux P2P. Nous analysons leurs avantages et inconvénients. Cette analyse nous a permis d'évaluer ces solutions par rapport à notre problématique.

Le plan de notre chapitre est le suivant. Dans les deux premières sections, nous donnons quelques concepts et notions sur la réplication et sur les bases de données répliquées. La section 3 présente les principales solutions existantes pour la réplication optimiste et leurs différentes stratégies pour la réconciliation de données. Dans la section 4, nous présentons les principaux concepts et caractéristiques des systèmes P2P, et les différentes solutions de réplication optimistes dans ces réseaux. Enfin, dans la dernière section, nous présentons un résumé de ces solutions tout en dégageant leurs insuffisances par rapport à notre problématique.

1 Réplication des données

La réplication des données est reconnue comme un moyen efficace pour augmenter la disponibilité et la fiabilité des bases de données. La réplication offre aux utilisateurs de meilleures performances et une plus grande disponibilité des données par le maintien de plusieurs copies d'une donnée, appelées répliques, sur des sites séparés.

1.1 Les bases de données distribuées et répliquées

Une base de données distribuée (répartie) est une collection de sites connectés par un réseau de communication. Chaque site est une base de données qui stocke une portion de la base de données. Chaque donnée est stockée exactement sur un seul site [BHG87, Sphd10]. La gestion d'une base de données répartie est gérée de manière transparente par un système de gestion de données (SGBD) réparti. Pour améliorer les performances et la disponibilité des données, les données peuvent être répliquées sur plusieurs sites. Cependant, la réplication introduit un nouveau problème car si une réplique est mise à jour, toutes les autres répliques doivent l'être afin de garantir la *cohérence mutuelle*, ce qui signifie que toutes les copies de la base sont identiques.

Définition 2.1 ***(cohérence mutuelle).*** *À un instant donné, toutes les copies d'une donnée sont mutuellement cohérentes si elles ont la même valeur.*

1.2 Mécanismes de la réplication

La réplication a fait l'objet de nombreux travaux dans le contexte des systèmes distribués et des bases de données [Sphd10]. Dans le contexte des bases de données, on parle de cohérence transactionnelle. Une transaction, par définition, est une séquence d'actions qui transforment la base de données d'un état cohérent vers un autre état cohérent. Pour garantir la cohérence de la base de données, les transactions doivent vérifier les propriétés d'Atomicité, de Cohérence, d'Isolation et de Durabilité, résumées sous l'acronyme *ACID*.

- **Atomicité :** une transaction doit intégrer dans la base de données toutes ses mises à jour ou aucune. C'est tout ou rien.
- **Cohérence :** la transaction doit faire passer la base de données d'un état cohérent à un autre état cohérent.
- **Isolation :** chaque transaction apparait comme si elle était la seule à modifier les données, même si d'autres transactions s'exécutent au même moment.
- **Durabilité :** dès qu'une transaction est validée, le système doit garantir que ses modifications seront conservées quoiqu'il arrive (quelques soient les pannes).

De façon générale, la gestion de la réplication est classifiée selon les concepts suivants: réplication simple maître (*single master*), réplication multi-maître (*multi master*), réplication *synchrone* et *asynchrone*.

1.2.1 Réplication simple-maître (single master)

Dans cette approche [MPV06], il y a une seule copie primaire pour chaque donnée répliquée. Dans ce cas, chaque mise à jour est d'abord appliquée à la copie primaire du site maître, puis se propage vers les autres copies secondaires détenues par les sites esclaves. La centralisation des mises à jour sur un seul exemplaire permet d'éviter les mises à jour simultanées sur différents sites, ce qui simplifie le contrôle de concurrence et facilite la gestion de la cohérence globale du système (cohérence mutuelle). Toutefois, cette centralisation peut introduire un goulot d'étranglement potentiel, et un seul point de défaillance pour le système. Par conséquent, la panne du site maître bloque les opérations de mise à jour, et limite donc la disponibilité des données. La Figure 1 montre un exemple de réplication simple-maître avec une copie primaire et deux copies secondaires.

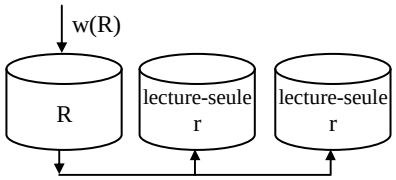

Figure 1. Réplication simple-maître
R est une copie primaire et r est une copie secondaire

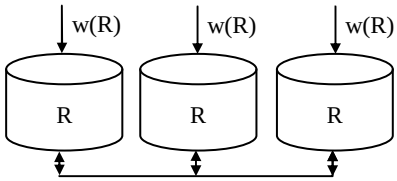

Figure 2. Réplication multi-maître
R est une copie primaire

1.2.2 Réplication multi-maître (multi master)

Dans cette approche, plusieurs sites tiennent des copies primaires d'une même donnée. Toutes ces copies peuvent être mises à jour par plusieurs sites en parallèle. La distribution des mises à jour sur les différents sites permet d'éviter les goulots d'étranglement et le point unique de défaillance, ce qui améliore la disponibilité des données et offre de meilleurs avantages de performance. Cependant, les mises à jour de la même donnée par différents sites peuvent créer des divergences de copies. Pour cela, un algorithme de réconciliation doit être appliqué pour résoudre les divergences. La Figure 2 montre un exemple de la réplication multi-maître.

1.2.3 Réplication Synchrone et Asynchrone

Dans les systèmes de bases de données répartis, l'accès aux données se fait via des transactions. Une transaction, prend un état d'une base de données, effectue une ou des actions et génère un autre état de la base. Ces actions sont essentiellement des opérations de lecture ou d'écriture de données suivies par une validation (*commit*). Si une transaction ne se termine pas avec succès, on dit qu'elle échoue. Une transaction qui met à jour une donnée doit être propagée à tous les sites qui contiennent des répliques de cette donnée afin de garder ses répliques cohérentes. Nous distinguons deux techniques de réplication de mise à jour: *réplication synchrone* et *réplication asynchrone*. Dans ce qui suit, nous discutons ces deux techniques.

La réplication synchrone (*eager*) : c'est un mode de distribution dans lequel toutes les sous-opérations locales effectuées suite à une mise à jour globale sont accomplies pour le compte de la même transaction. Cette technique est très utile lorsque les mises à jour effectuées sur un site doivent être prises en compte immédiatement dans les autres sites, comme le montre la Figure 3. L'avantage incontestable de la mise à jour synchrone est de fournir la dernière version des données (cohérence mutuelle) quelle que soit la copie accédée. Cela demande néanmoins, une gestion des transactions multi-site coûteuse en ressources et des algorithmes de contrôle de concurrence complexes. Pour cette raison, la mise à jour asynchrone est le plus souvent préférée.

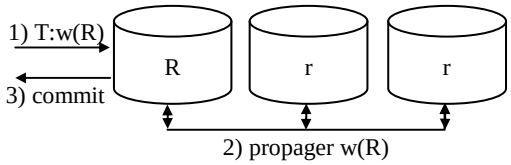

Figure 3. Principe de la propagation synchrone

La réplication asynchrone (lazy) : dans cette approche, les mises à jour sont propagées après la fin de la transaction (après le commit). En effet, la première transaction valide (commits) le plus tôt possible, et ensuite les mises à jour sont propagées à l'ensemble des répliques, comme le montre la Figure 4. Les solutions de réplication asynchrone peuvent être classées comme *optimiste* ou *non-optimiste* [PD02] selon leurs hypothèses concernant les mises à jour conflictuelles. En général, la

réplication *asynchrone optimisme* repose sur l'hypothèse optimiste que les mises à jour des conflits se produisent que rarement, voire pas du tout. Les mises à jour sont donc propagées à l'arrière-plan, et les conflits sont réconciliés après avoir eu lieu. En revanche, la réplication *asynchrone non-optimiste* suppose que les mises à jour conflictuelles sont susceptibles de se produire et met en œuvre des mécanismes de propagation qui empêchent les mises à jour conflictuelles.

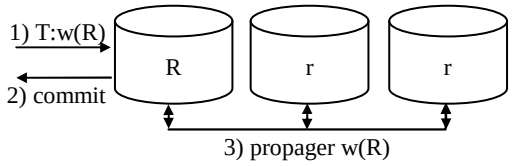

Figure 4. Principe de la propagation asynchrone

Un avantage de l'approche *asynchrone* est que la mise à jour ne bloque pas en raison d'indisponibilité des répliques, ce qui améliore la disponibilité des données. En outre, la communication ne nécessite pas de coordonner les mises à jour simultanées, ce qui réduit les temps de réponse des transactions et améliore le passage à l'échelle du système. En particulier, la réplication optimiste asynchrone est plus flexible que d'autres approches, puisque le système peut choisir le moment approprié pour propager les mises à jour et l'application progresse dans un environnement dynamique dans lequel les nœuds peuvent se connecter et se déconnecter à tout moment. Son principal inconvénient est que les répliques peuvent diverger. La réplication asynchrone non-optimiste n'est pas aussi souple que l'approche optimiste, mais elle fournit des valeurs à jour à haute probabilité pour les opérations locales de lectures.

2 Réplication dans les systèmes distribués

Dans les systèmes distribués, la répartition des données sur plusieurs sites est assurée par la réplication. Cependant, la réplication de données nécessite de maintenir à jour les différentes répliques et donc de gérer les accès simultanés et les modifications concurrentes de données.

Le contrôle de concurrence entre les différentes répliques bute sur le problème de consensus. Le consensus est un des problèmes majeurs des systèmes répartis, en particulier dans les systèmes *asynchrones*. Il consiste à mettre d'accord tous les processus sur une valeur (oui ou non par exemple). Le problème du consensus peut être défini de la façon suivante :

Soit un ensemble de *n* processus, chaque processus *pi* propose une valeur initiale *vi*. Ces *n* processus doivent se mettre d'accord sur une valeur commune *v* qui est la valeur initiale d'un des processus. Les conditions suivantes doivent être respectées : tout processus *correct* (i.e. s'il n'est pas en panne) doit finir par décider (*Terminaison*), si un processus décide la valeur *v*, alors *v* est une valeur initiale d'un des processus (*Validity*), deux processus corrects ne peuvent pas décider différemment (*Agreement*).

Le consensus doit être initialisé, c'est-à-dire tous les processus doivent savoir quand démarrer l'algorithme. Cela peut se faire, par exemple : à l'initiative d'un processus qui diffuse aux autres un message pour lancer le consensus ou à une heure fixée à l'avance.

2.1 Paxos : exemple d'un protocole de consensus

Dans cette section, nous introduisons rapidement le protocole Paxos [Lamport89] comme l'un des principaux protocoles de consensus. Auparavant, nous présentons d'abord certaines définitions qui seront utiles pour la suite de cette section.

Définition 2.2 ***(Système synchrone).*** *Un système distribué synchrone est un système dans lequel il existe une limite connue du délai de transmission d'un message entre deux processus (i.e. les contraintes temporelles sur les messages sont bornées).*

Définition 2.3 ***(Système asynchrone).*** *Un système distribué asynchrone est un système qui n'offre aucune borne temporelle (i.e. pas de délai limite sur la transmission des messages). C'est le cas de la plupart des systèmes réels utilisés actuellement.*

Paxos [Lamport89] est un protocole pour le consensus dans un système *asynchrone*. De façon informelle, le principe de Paxos est le suivant: il utilise un processus coordinateur (ou *primaire*) qui gère un ensemble de processus (dits *agents*) et tente d'obtenir une valeur de décision par vote majoritaire. L'algorithme doit résister à la défaillance du primaire. Plusieurs processus peuvent jouer le rôle du primaire, successivement ou même simultanément. L'algorithme se termine s'il existe un primaire unique pendant une période suffisamment longue pour que le primaire ait deux tours d'échanges avec une majorité d'agents. Cette condition peut être assurée à l'aide de délais de garde. Cette méthode peut échouer, car la communication étant asynchrone, et dans ce cas l'algorithme ne se termine pas. Paxos ne passe pas à l'échelle pour un nombre important de répliques.

2.2 Réalisation d'un consensus dans les systèmes asynchrones

Le consensus est facile à mettre en œuvre dans un système synchrone. Toutefois Fischer, Lynch et Paterson [FLP85] ont montré qu'il est sans solution déterministe dans les systèmes asynchrones où les canaux de communication sont non fiables et asynchrones et où des défaillances de nœuds sont possibles.

Dans un algorithme de consensus, tel que par un échange des valeurs calculées *Vi*, tous les processus corrects décident de la même valeur *V*. Ceci engendre un nombre important de messages et par conséquent les algorithmes de consensus passent difficilement à l'échelle.

Dans les systèmes répartis, le partage de données est assuré par la réplication. Le maintien de la cohérence entre les différentes répliques bute sur plusieurs problèmes, en particulier l'impossibilité du consensus. Ces difficultés sont contournées par la cohérence optimiste, qui laisse diverger les répliques pour les réconcilier a posteriori. Dans la section suivante, nous présentons les principales solutions existantes pour la réplication optimiste et leurs différentes stratégies pour la réconciliation de données.

3 Réplication Optimiste et Divergence

Un système de réplication optimiste est constitué d'un ensemble de sites interconnectés par un réseau. Chaque site possède une copie des objets partagés comme des documents textuels ou des images. Sur un site, une réplique peut être modifiée au moyen d'opérations. Quand une réplique est modifiée sur un site, l'opération correspondante est immédiatement exécutée sur ce site, puis propagée aux autres sites pour y être ré-exécutée. Lorsque deux répliques de deux sites différents sont modifiées en parallèle, elles *divergent*. Il est donc possible d'observer au même moment une valeur sur un site et une autre valeur sur un autre site. Les algorithmes de réplication optimiste doivent assurer la convergence des répliques. Le système doit être convergeant quand il est au repos, c'est-à-dire quand toutes les opérations ont été propagées à tous les sites.

3.1 Réconciliation

Les mises à jour parallèles d'un système de réplication optimiste peuvent provoquer des conflits et la divergence des répliques. La réconciliation est l'activité qui assure la cohérence mutuelle entre les différentes répliques. Un critère important pour assurer la cohérence de données consiste à garantir un *ordre total* sur les mises à jours exécutés sur les différentes répliques. Nous présentons dans la suite quelques systèmes de réplication optimiste utilisant des stratégies différentes pour la réconciliation des données et le maintien d'un ordre total.

Définition 2.4 ***(Ordre total).*** *Un ordre total sur les opérations de mises à jour permet d'assurer que toutes les répliques de la même donnée reçoivent les opérations dans le même ordre.*

3.1.1 CVS

CVS [Ber90, Ophd05] est un gestionnaire de configuration qui permet à des utilisateurs de partager des documents et de les éditer de manière collaborative. CVS intègre un système de gestion de versions qui repose sur un protocole optimiste de gestion des accès concurrents: le paradigme du *Copier-Modifier-Fusionner*. Ce modèle est un modèle centralisé. On distingue deux types d'espaces :

- Espace référentiel. Cet espace maintient un ensemble multi-versionné des objets partagés. C'est à dire que pour chaque objet partagé, il conserve l'ensemble des versions de cet objet dont la publication par les utilisateurs a réussi ;
- Des espaces de travail. Chaque utilisateur possède son propre espace de travail. Les modifications apportées dans cet espace restent confidentielles jusqu'à leur publication.

Pour modifier un objet, l'utilisateur en crée une réplique dans son espace de travail. Il peut ensuite librement la modifier. Une fois sa tâche terminée, il publie une nouvelle version dans le référentiel. Deux utilisateurs peuvent modifier en parallèle deux

répliques du même objet. Dans ce cas, le premier qui a fini peut publier sans problème. Le second est forcé d'intégrer les modifications du premier avant de publier à son tour.

Convergence : Le paradigme du Copier-Modifier-Fusionner assure la convergence des répliques. Cette convergence a cependant un coût élevé. En effet, dans un système où n répliques sont modifiées en parallèle, il faut $(2 * n - 1)$ étapes pour atteindre la convergence.

Passage à l'échelle : Le paradigme Copier-Modifier-Fusionner associe à des outils de fusion déterministes un ordre total. Ainsi, chaque opération reçoit un numéro d'intégration avant d'être publiée. Une opération ne peut être publiée que si toutes les opérations précédentes ont été reçues. Cependant, maintenir un ordre total a un coût élevé. Soit cet ordre est maintenu par un site central avec les risques intrinsèques de pannes, soit il est maintenu avec des algorithmes de consensus distribués asynchrones. Dans ce cas, le système peut être tolérant aux pannes mais passe difficilement à l'échelle.

3.1.2 IceCube

IceCube [KRSD01, SPR04, PSM03, Ophd05] est un système de réconciliation générique. Ce système traite la réconciliation comme un problème d'optimisation : celui d'exécuter une combinaison optimale de mises à jour concurrentes. A partir des différents journaux présents sur les différents sites, IceCube calcule un journal optimal unique contenant le nombre maximum de mises à jour non conflictuelles. Pour cela, IceCube utilise la sémantique des mises à jour exprimée sous forme de contraintes.

Dans IceCube, les mises à jour sont modélisées par des actions. Le système permet de définir des dépendances sémantiques entre les actions sous forme de contraintes. Deux types de contraintes sont disponibles : les contraintes statiques et les contraintes dynamiques. Les contraintes statiques sont évaluées sans utiliser l'état des objets. Les contraintes dynamiques peuvent utiliser l'état des objets. La réconciliation se déroule en trois phases (Voir Figure 5) :

- Ordonnancement : durant cette phase, le système construit toutes les exécutions possibles en combinant les différentes actions issues des journaux des différents sites. L'espace de recherche est limité par les contraintes statiques.
- Simulation : on exécute les différents ordonnancements trouvés dans la phase précédente. Si une contrainte dynamique est violée alors l'ordonnancement en question est abandonné et rejeté. Après cette phase, il ne reste plus que les ordonnancements respectant les contraintes statiques et dynamiques.
- Sélection : la phase de sélection classe les ordonnancements restants et doit en choisir un. Le critère de sélection est choisi par un administrateur.

Cette description est une vision simplifiée du fonctionnement d'IceCube. Dans IceCube, les trois phases décrites précédemment ne sont pas exécutées de manière séquentielle, mais en parallèle. Nous évaluons maintenant IceCube selon les critères suivants :

Convergence : IceCube assure la convergence. Pour faire converger *n* sites, IceCube désigne un site comme responsable de la réconciliation. Tous les sites doivent envoyer les opérations effectuées depuis la dernière réconciliation vers ce site. L'algorithme principal d'IceCube peut alors calculer un journal réconcilié. Ce journal est ensuite renvoyé à tous les sites. Chaque site défait ses modifications locales avant réconciliation et rejoue le journal réconcilié. Dans le cadre d'un Wiki P2P, cela signifie que régulièrement :

- Le système doit être gelé,
- Un site central élu, et toutes les modifications concurrentes sont envoyées sur ce site,
- Le site central calcule le journal réconcilié et assure un ordre total,
- Les sites réintègrent le journal réconcilié,
- Le système peut continuer,
- Rien n'est dit si une panne d'un site survient pendant ce cycle.

Il est clair que ce type de fonctionnement n'est pas compatible avec les contraintes d'un réseau P2P.

Passage à l'échelle : IceCube utilise un site central pour assurer la convergence. Donc, il a du mal de passer à l'échelle.

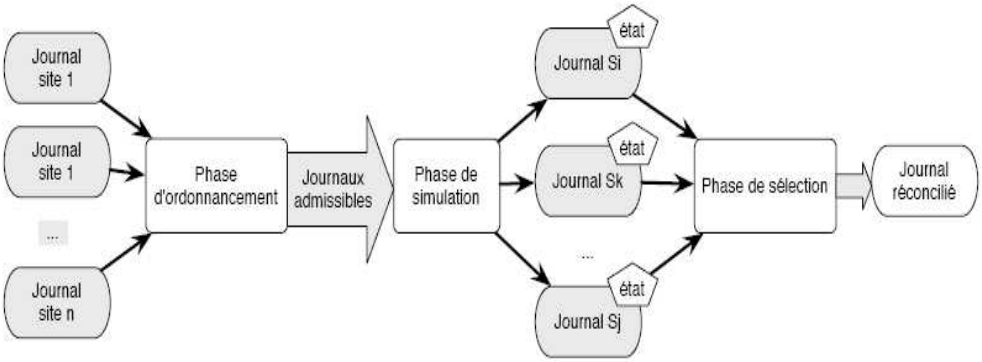

Figure 5. Processus de réconciliation dans IceCube.

3.1.3 Bayou

Bayou [PSTT+97, TTPD+95, Ophd05] est un système de gestion de données pour des applications collaboratives dans un environnement mobile. Les données sont répliquées entre différents serveurs. Dès le moment où un serveur reçoit une opération, il tente de l'exécuter. Deux sites peuvent donc recevoir et exécuter les mêmes opérations dans un ordre différent. Pour assurer la convergence, les serveurs doivent exécuter toutes les opérations dans le même ordre. Dans Bayou, les serveurs vont continuellement défaire et rejouer les opérations au fur et à mesure qu'ils prennent connaissance de l'ordre final. Cet ordre final est décidé par un serveur principal désigné au lancement du système.

Ainsi, chaque serveur maintient un journal des opérations exécutées. Ce journal est scindé en deux parties. Un préfixe *p* qui contient les opérations validées par le serveur principal. Ces opérations sont ordonnées définitivement selon un ordre total introduit lors de la validation par le serveur principal. Le reste du journal, qualifié de "provisoire", est ordonné selon un ordre total qui peut être remis en cause au fur et à mesure que de nouvelles opérations sont reçues. Bayou utilise donc un site primaire qui assure un ordre global continu sur un préfixe de l'historique des modifications. Ce préfixe est ensuite distribué aux autres sites. Ce site primaire représente un point de congestion.

Malgré que Bayou assure la convergence, La mise en place d'un site primaire représente un point de congestion et limite donc le passage à l'échelle de Bayou.

3.1.4 Transformée Opérationnelle (OT)

Le modèle des transformées opérationnelles (OT) [MOSM+03, Ophd05] a été développé par la communauté des éditeurs collaboratifs synchrones. La préservation de l'intention est au centre de la définition du modèle de cohérence de cette approche. L'architecture générale du modèle des transformées opérationnelle distingue deux composants :

- Un algorithme d'intégration. Il est responsable de la réception, de la diffusion et de l'exécution des opérations. Si nécessaire, il fait appel aux fonctions de transformation. Cet algorithme est indépendant du type des données manipulées (chaîne de caractères, document XML, système de fichiers).
- Un ensemble de fonctions de transformation. Ces fonctions ont la charge de "fusionner" les mises à jour en sérialisant deux opérations concurrentes. Ces fonctions sont spécifiques à un type de données particulier.

Le modèle OT considère *n* sites. Chaque site possède une copie des objets partagés. Quand un objet est modifié sur un site, l'opération correspondante est exécutée immédiatement sur ce site, puis diffusée aux autres sites pour y être également exécutée. Autrement dit, une opération est traitée en quatre étapes:

- Génération sur un site.
- Diffusion aux autres sites.
- Réception par les autres sites.
- Exécution sur ces autres sites.

Lorsqu'une opération est reçue, son contexte d'exécution peut être différent de celui dans lequel elle a été générée. Dans ce cas, l'intégration de cette opération sur les autres sites peut conduire à des incohérences entre les différentes copies. Dans OT, les opérations reçues doivent être transformées par rapport aux opérations locales concurrentes avant d'être exécutées. Cette transformation est effectuée en utilisant des fonctions de transformation.

L'algorithme d'intégration est défini comme correct s'il assure la Causalité, la Convergence et la préservation des Intentions (CCI) [SCF97, SZJY97, SE98] :

- *Respect de la causalité* : Pour certaines opérations, il existe une relation de précédence causale qui doit être maintenue. On dit que l'opération *op1* précède l'opération *op2* (noté *op1* -> *op2*) si et seulement si op2 a été générée et exécutée sur une réplique après l'exécution de op1 sur cette même réplique. Puisque *op2* a été générée après *op1*, elle tient compte implicitement de ces effets. Le respect de la causalité garantit que pour les opérations pour lesquelles il existe une relation de causalité, celles-ci seront exécutées dans le même ordre sur toutes les répliques.
- *Préservation de l'intention* : Deux opérations *op1* et *op2* qui ne sont pas liées par une relation de précédence sont concurrentes. De ce fait, ces deux opérations peuvent être exécutées sur les différentes répliques dans un ordre quelconque. Cependant, si on exécute l'opération *op1* avant l'opération *op2* alors il faudra lors de l'exécution d'*op2* tenir compte des effets d'*op1* de façon à respecter les effets d'*op2*.
- *Convergence des répliques* : Lorsque tous les sites ont exécuté les mêmes opérations, les répliques doivent être identiques. Cependant, le fait de respecter les relations de causalité entre les opérations et de préserver les effets des opérations ne suffit pas pour garantir la convergence des répliques. La Figure 6, montre un exemple de scénario où les deux critères sont respectés sans les copies convergent.

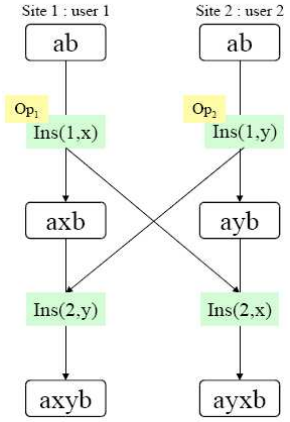

Figure 6. Divergence des copies due à la violation de la condition C1

En effet, pour intégrer les opérations sur les différentes répliques, la fonction de transformation illustrée dans la Figure 7 est utilisée.

```
T(Ins(p1,c1),Ins(p2,c2)):-
  if (p1 < p2) then
    return Ins(p1, c1)
  else
    return Ins(p1 + 1, c1)
  endif
```

Figure 7. Fonction de transformations naïve

Il a été montré [EG89] que pour garantir la convergence des copies, les fonctions de transformation doivent vérifier la condition C1 suivante.

Définition 2.5 ***(Condition C1).*** *Soient op1 et op2 deux opérations concurrentes définies sur le même état. T remplit la condition C1 si et seulement si :*

$$op1 \circ T(op2, op1) \equiv op2 \circ T(op1, op2)$$

Où ○ dénote un opérateur concaténant deux opérations pour former une séquence d'opérations, et où ≡ signifie que les deux séquences *op1*○ *T(op2, op1) et op2* ○ *T(op1, op2)* produisent le même état.

Dans l'exemple précédent (Figure 6), les opérations *op1* et *op2* sont concurrentes et définies sur le même état *ab*. Pourtant, l'exécution de la séquence d'opération *op1* ○*T(op2, op1)* sur cet état donne l'état *"axyb"* tandis que l'exécution de la séquence *op2*○*T(op1, op2)* donne l'état *"ayxb"*. Les deux états ne sont pas égaux ce qui prouve la violation de la condition C1. Pour satisfaire cette condition, il faut que la fonction de transformation prenne en compte le cas particulier de deux insertions à la même position dans la chaîne de caractères. Etant donné que rien ne peut permettre de décider lequel des deux caractères doit être placé devant l'autre, il faut donc prendre un choix arbitraire. Et, qu'il faut garantir que ce choix sera le même sur toutes les répliques.

La Figure 8 présente un exemple de fonction de transformation vérifiant la condition C1. Pour distinguer les deux opérations, lors de l'insertion à la même position, il faut comparer les caractères. L'insertion dont le caractère est le plus petit selon l'ordre lexicographique s'effectue devant l'autre caractère.

```
T(Ins(p1,c1),Ins(p2,c2)):-
  if (p1 < p2) then
    return Ins(p1, c1)
  else if (p1 > p2) then
    return Ins(p1 + 1, c1)
  else if (p1 = p2) then
      if (c1 < c2) then
         return Ins(p1, c1)
      else
         return Ins(p1 + 1, c1)
      endif
  endif
```

Figure 8. Fonction de transformation vérifiant la condition C1

La condition C1 n'est pas suffisante pour garantir la convergence des copies en présence de plus de deux opérations concurrentes. La Figure 10 illustre une telle situation. Dans ce scénario, la fonction de transformation satisfaisant la condition C1 présentée précédemment est utilisée. La fonction de transformation donnée par la Figure 9 est utilisée également.

```
T(Ins(p1,c1),Del(p2)):-
  if (p1 ≤ p2) then
    return Ins(p1, c1)
  else
    return Ins(p1 − 1, c1)
  endif
```

Figure 9. Fonction de transformation vérifiant la condition C1

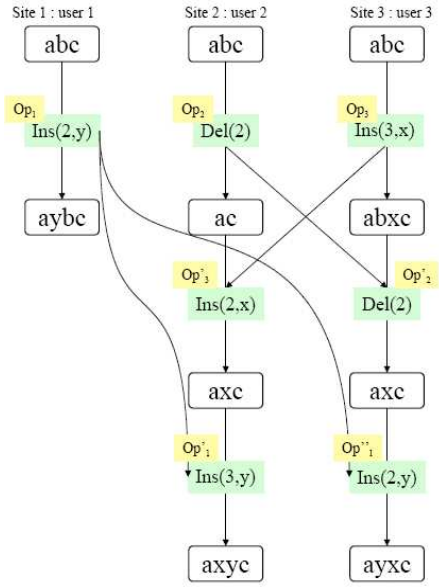

Figure 10. Insuffisance de la condition C1

Le respect de la condition C1 garantit la convergence des copies après l'intégration d'*op2* sur le site 2 et d'*op3* sur le site 3. Cependant, un problème survient lors de l'intégration d'*op1*. Sur le site 2, on obtient l'opération *op'1 = Ins(3, y)* tandis que sur le site 3 on obtient l'opération *op"1 = Ins(2, y)*. Ces deux opérations étant différentes lorsqu'elles sont exécutées sur le même état donnent forcément deux états différents. Les copies ne convergent donc pas.

Pour éviter cette situation un certain nombre d'algorithmes d'intégration, tels que GOT, SOCT3, SOCT4 [VCFS00], SOCT5 [Vid02], imposent un ordre de sérialisation unique. Ainsi dans notre exemple, sur le site 3, *op2* est toujours transformée par rapport à *op3* et sur le site 2 *op3* est toujours transformée par rapport à *op2*. Par contre, l'intégration sur ces deux sites de l'opération *op1* est réalisée en transformant par rapport à la même séquence *[op2;T(op3, op2)] = [op2; op'3]* et non plus par rapport à deux séquences équivalentes. Au final, l'équivalence entre les deux séquences exécutées sur les sites 2 et 3 est bien assurée. En effet, sur ces deux sites nous avons exécuté, tout d'abord, une des deux séquences équivalentes *[op2; op'3]* ou *[op3; op'2]*,

puis la même opération *op'1*; cette opération résultant de la transformation de *op1* par rapport à la séquence *[op2; op03]*.

La construction d'un ordre de sérialisation unique dans un contexte réparti est coûteuse et passe difficilement à l'échelle. C'est pourquoi, d'autres algorithmes d'intégration, tels que adOPTed [RNRG96], GOTO [SE98] SOCT2 [SCF98], ne requièrent pas d'ordre total, mais en contrepartie imposent aux fonctions de transformation de satisfaire une condition supplémentaire [RNRG96], la condition C2.

Définition 2.6 ***(Condition C2).*** *Soient trois opérations op1, op2 et op3 concurrentes définies sur le même état, la fonction de transformation vérifie la condition C2.*

$$T(op3, op1 \circ T(op2, op1)) = T(op3, op2 \circ T(op1, op2))$$

Contrairement à la condition C1, il s'agit bien ici d'une égalité sur les opérations obtenues après transformation. La convergence est assurée dans le cas général si les fonctions de transformation vérifient C1 et C2. Avec des fonctions de transformation vérifiant C1 et C2, aucun ordre total, ni site central sont requis. Par conséquent, l'approche OT doit passer à l'échelle.

3.1.5 So6

So6 [MOSM+03, OMSI04, Ophd05] est un gestionnaire de configuration reposant sur les modèles des transformées opérationnels. So6 montre qu'il est possible d'adapter les algorithmes OT, développés pour des éditeurs collaboratifs temps réels, afin de réaliser un outil comparable à CVS ou Subversion. So6 met en avant l'aspect générique de l'approche OT. En séparant l'algorithme d'intégration des fonctions de transformation, il est possible de construire de façon simple un gestionnaire de configuration. En démontrant que les fonctions de transformation vérifient la propriété C1, on obtient immédiatement un outil assurant la convergence des copies dans tous les cas. Cet outil reste extensible à de nouveaux types de données répliquées en ajoutant les fonctions de transformation correspondantes. L'architecture générale de So6 est donnée Figure 11.

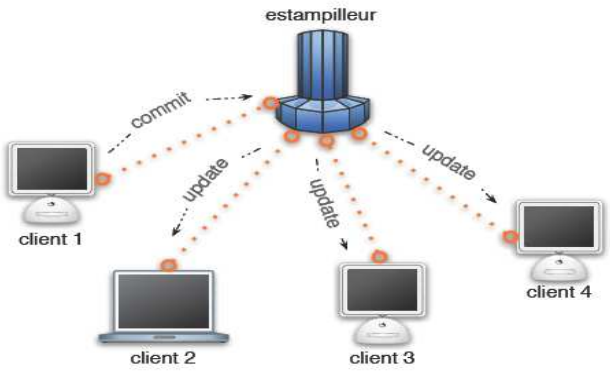

Figure 11. Architecture Générale de So6

So6 est basé sur l'algorithme SOCT4 [VCFS00]. Cet algorithme utilise un estampilleur pour ordonner totalement les opérations. Il se démarque des autres algorithmes d'intégration OT du même type en utilisant un mécanisme de diffusion différée (voir Figure 12). Pour envoyer une opération, il faut d'abord avoir reçu toutes les opérations estampillées et transformé les opérations locales en conséquence. Il est ensuite possible d'estampiller ses opérations locales et de les envoyer. Le fonctionnement de cet algorithme s'apparente au paradigme "*copier-modifier-fusionner*" utilisé dans CVS où un utilisateur ne peut publier ses modifications que s'il a pris en compte toutes les modifications déjà publiées.

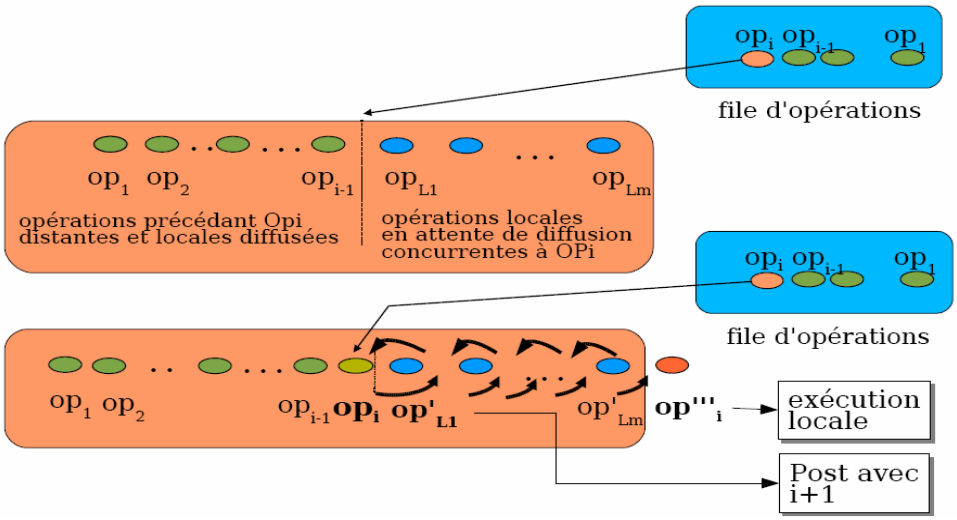

Figure 12. Intégration d'opération dans So6 en utilisant l'algorithme SOCT4

L'architecture générale du système est centralisée autour d'un estampilleur qui délivre une estampille à chaque opération. Une application cliente s'exécute sur chaque machine des utilisateurs. Les applications clientes ne s'échangent pas directement les opérations, elles doivent les faire transiter par l'estampilleur.

So6 met en œuvre l'estampilleur sous forme d'une file d'opérations. Cette file maintient, de manière persistante, les opérations publiées par les différents sites dans l'ordre des estampilles. So6 assure la convergence des données. L'état de convergence est déterminé en fonction du code des fonctions de transformation. So6 est générique et permet l'intégration de nouveaux types de données.

Cependant, So6 requiert un estampilleur central pour fonctionner. Si cet estampilleur est en panne, le système est bloqué. En outre, la taille du journal utilisé par OT au niveau du serveur d'estampille a tendance à être important et peut non maîtrisable en un seul nœud. En conséquence, So6 ne passe pas à l'échelle.

3.1.6 Google Wave

Google Wave [GW09] est un projet de Google. Il s'agit d'un système d'édition collaborative innovant dans lequel les utilisateurs peuvent collaborer en temps réel sur un document hiérarchique représenté par un document XML. Le concept de ce système est de mélanger les notions de services de courriel, de messagerie instantanée, de Wiki et de réseau social, le tout associé à un correcteur orthographique et un traducteur instantané. Il faut bien noter que Wave se base sur les aspects fondamentaux de HTML 5 qui n'est pas encore adopté partout.

Le système de Google Wave est basé sur l'approche des transformées opérationnelle (OT) et utilise l'algorithme d'intégration Jupiter [NCDL95]. Jupiter est dérivé d'un algorithme optimiste développé par Ellis et Gibbs [EG89]. L'architecture de Jupiter est centralisée, le serveur ayant la charge de réaliser toutes les transformations des opérations concurrentes avant de les intégrer. Par conséquent, la collaboration et les performances globales du système reposent uniquement sur le serveur. Afin d'augmenter le passage à l'échelle, il serait intéressant de répartir la charge sur l'ensemble des pairs. Plusieurs approches décentralisées [LL04a, LL04b] ont été proposées dans les modèles des transformées opérationnels, cependant, elles reposent sur l'utilisation de vecteurs d'horloge, qui ne sont pas compatibles avec les systèmes très dynamiques, par ex. P2P.

3.2 Synthèse Générale

Nous avons évalué les systèmes potentiellement intéressants pour le domaine de la réconciliation et la réplication optimiste selon trois critères: architecture, convergence et passage à l'échelle. Le tableau ci-dessous résume nos conclusions :

Tableau 1. Comparaison des solutions de réplication optimiste

	Objet	**Architecture**	**Convergence**	**Passage à l'échelle**
CVS	fichier	centralisé	Oui	Non
IceCube	quelconque	centralisé	Oui	Non
Bayou	base de données	primaire	Oui	Non
OT (C1 + C2)	quelconque	décentralisé	Oui	Oui
So6	quelconque	centralisé	Oui	Non
Google Wave	quelconque	centralisé	Oui	Non

Seule l'approche OT est indépendante du type des données manipulées, garantit la convergence et passe à l'échelle avec des fonctions de transformation vérifiant les conditions C1 et C2. Cependant, l'écriture de telles fonctions est un problème réputé difficile. Dans [IMOR02, IMOR03, IROM05], les auteurs proposent un démonstrateur de théorème pour vérifier la correction des fonctions de transformation existantes et pour valider rapidement leurs propositions de nouvelles fonctions de transformations.

Dans l'approche OT, et afin d'assurer la cohérence à terme des données, les opérations doivent respecter un critère bien défini. En effet, toutes les opérations doivent être estampillées selon un ordre total. So6 qui est une solution basée sur les transformées opérationnelles garantit la convergence avec des fonctions de transformations vérifiant seulement la condition C1 mais nécessite un ordre total sur les opérations. Cependant, So6 ne passe pas à l'échelle parce qu'il requiert un estampilleur central afin de garantir l'ordre total.

Plusieurs approches décentralisées [LL04a, LL04b] ont été proposées dans les modèles des transformées opérationnels, cependant, elles reposent sur l'utilisation de vecteurs d'horloge, qui ne sont pas compatibles avec les caractéristiques des réseaux P2P.

4 Les Systèmes P2P

Dans cette section, nous présentons tout d'abord les principaux concepts et les caractéristiques des systèmes pair-à-pair. Puis, nous proposons une classification qui distingue trois types de réseaux P2P, et pour chacun de ces réseaux, nous présentons les principales applications. Enfin, nous présentons une évaluation de différents types de réseaux P2P.

4.1 Caractéristiques

Les systèmes P2P sont en plein essor depuis maintenant plusieurs années. Ce paradigme permet de concevoir des systèmes de très grande taille à forte disponibilité et à faible coût sans recourir à des serveurs centraux [VP04]. En effet, les réseaux P2P sont des réseaux overlay, décentralisés, où tous les nœuds, appelés pairs, jouent à la fois le rôle de serveur et de client (voir Figure 13). L'organisation dans un tel réseau repose sur l'ensemble des pairs, donc il n'y a pas d'entité chargée d'administrer le réseau. En distribuant les données et les traitements sur tous les pairs du réseau, les systèmes P2P peuvent passer à très grande échelle sans recourir à des serveurs très puissants. Toutefois, ils induisent certains problèmes, liés par exemple à la sécurité des ressources. Dans la suite de cette section, nous présentons l'ensemble des caractéristiques que présente le réseau P2P [Dphd05].

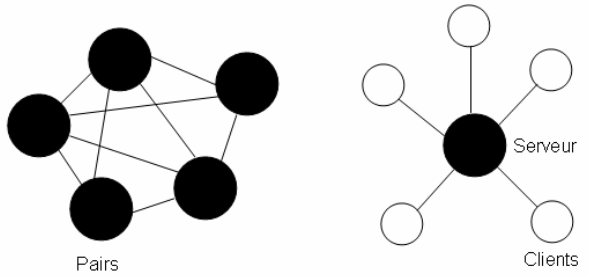

Figure 13. P2P vs Client/Serveur

4.1.1 Le passage à l'échelle

L'absence de coordination globale entre les pairs se répercute directement sur le passage à l'échelle des applications P2P. Les architectures P2P se présentent actuellement comme une solution viable pour permettre le passage à l'échelle de ses applications. Par exemple, les applications P2P de partage de fichiers comptent un très grand nombre de participants et fonctionnent sans problème. D'après une étude menée en 2001 [SGG02], OceanStore [KBCC+00] qui est une application P2P de stockage de fichiers, permet de gérer 10^{10} utilisateurs stockant au total plus de 10^{14} fichiers.

4.1.2 La tolérance aux fautes

Dans l'architecture client-serveur la disponibilité d'un service repose intégralement sur celle du serveur. Si celui-ci s'écoule, le service qu'il fournit devient indisponible. Dans un contexte P2P, il n'existe potentiellement aucun point central de faute. Si un pair disparaît, le service continuera d'être fourni par ceux qui restent. En d'autres termes, la disponibilité d'un service n'est plus liée aux pairs mais à la communauté de pairs qui le fournissent.

4.1.3 La dynamicité

L'une des caractéristiques les plus importantes des réseaux P2P c'est le comportement dynamique des pairs. Dans un réseau P2P, les pairs peuvent se connecter ou se déconnecter du système de manière libre et à n'importe quel moment.

4.1.4 L'auto-organisation

Une autre caractéristique des systèmes P2P, c'est la possibilité d'auto-organisation. Cependant, l'absence d'élément central dans les applications P2P nécessite la mise en place de mécanismes d'auto-organisation qui permet à un ensemble de pairs de réaliser une fonction globale de manière décentralisée quels que soient les allers et retours des pairs et la disponibilité des ressources dans le réseau.

4.1.5 Un réseau virtuel

Les pairs participants à un système P2P forment souvent un réseau virtuel, appelé overlay [DM03] construit au dessus de réseau physique. Un exemple d'overlay est représenté sur la Figure 14. Généralement, un tel réseau virtuel présente une propriété de transparence qui s'applique à plusieurs points. Tout d'abord, il permet de faire abstraction des différences de nature des pairs. Un réseau peut être composée de pairs avec des caractéristiques diffèrent sur les plans (1) matériel, avec par exemple des stations de travail, des téléphones mobiles ou un cluster de machines, (2) logiciel, au niveau des systèmes d'exploitations et langages de programmation, et (3) de la communication, avec l'utilisation de technologies différentes. La transparence s'applique aussi sur le routage effectué au niveau sous-jacent : deux voisins de la topologie virtuelle ne le sont pas forcément physiquement : ils peuvent être situés dans des espaces physiques et sur des réseaux différents (exemple, les pairs A et B présentés dans la Figure 14). L'overlay rend ainsi transparent le routage effectué au niveau physique. Enfin, concernant les ressources, un overlay rend transparent leur accès, leur

localisation et leur réplication. Lorsqu'un pair accède à une ressource, il ne sait pas que cette ressource est locale ou distante. Dans le cas où la ressource est distante, il n'a pas de connaissance de l'hôte qui l'héberge, et si elle est répliquée, il ne sait pas à quelle réplique il accède.

L'utilisation d'un overlay nécessite toutefois la mise en œuvre des mécanismes de nommage et routages dédiés. Les pairs ne sont donc plus représentés par leur adresse de niveau transport ou adresse physique mais par un identifiant définit dans le cadre d'overlay. En outre, pour pouvoir découvrir et accéder à des ressources, des algorithmes de routage et d'accès sont mis en place. Dans la section 4.2.2, nous nous intéressons à cet aspect du système P2P.

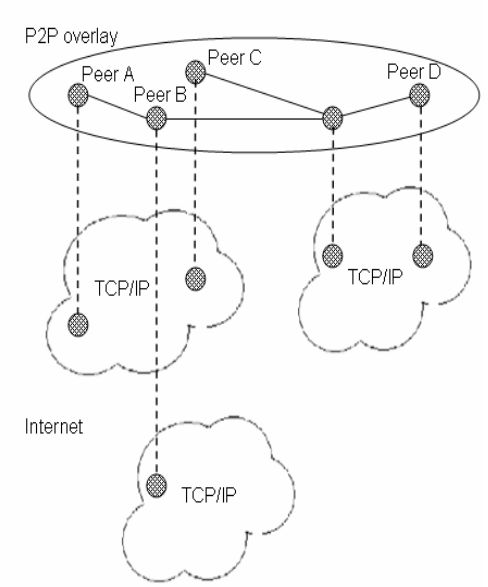

Figure 14. P2P overlay au dessus d'une infrastructure d'Internet

4.2 Types de réseaux P2P

Les approches dans le domaine P2P reposent sur la construction d'un réseau virtuel sur le réseau physique (typiquement Internet). Il existe trois classes principales de réseaux virtuels: les non-structurés, les structurés et les super-pair [Dphd05, MPV06, HSG08] (voir Figure 15).

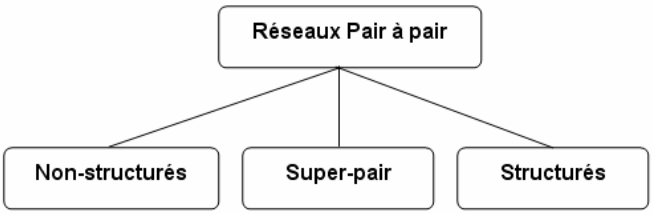

Figure 15. Classifications des réseaux P2P

4.2.1 Les réseaux non-structurés

Dans les approches non structurées, le réseau P2P est créé de manière non-déterministe. Dans ce type d'approches, le placement des données dans le réseau est indépendant du réseau virtuel lui-même. Par conséquent, un nœud peut connaître ses voisins dans le réseau mais il ne connaît pas les données qu'ils stockent. Le placement de données dans ce type de réseau est effectué en général sur le nœud demandeur. Par ailleurs, la recherche d'une donnée sur ce type de réseau est réalisée par inondation, la localisation de la donnée n'étant à priori pas connue. La recherche peut concerner une donnée particulière ou un ensemble. Les approches dans ce domaine donnent lieu à des systèmes tolérants aux fautes et où les nœuds sont très autonomes. Par contre, la recherche d'une donnée dans un réseau avec un grand nombre de nœuds peut être longue et ne donne que des résultats partiels.

Différents niveaux de décentralisation scindent les réseaux non-structurés en deux modèles: le modèle centralisé et le modèle purement décentralisé.

4.2.1.1 Le modèle centralisé

Le modèle centralisé est à la limite du modèle P2P car il repose sur un serveur dédié qui centralise et maintient l'ensemble des connaissances de la communauté, les ressources étant toujours hébergées sur les pairs. Ce modèle est utilisé dans des applications telle que Napster [S01]. Dans un modèle centralisé chaque pair ne possède au minimum qu'une connaissance du serveur central et pas des autres pairs, bien qu'une fois les opérations de découverte et localisation effectuées, il puisse interagir directement avec eux. Ainsi, cette interaction possible entre les pairs différencie le modèle P2P centralisé du modèle client-serveur. Nous présentons dans cette section l'application Napster comme exemple du modèle P2P centralisé.

Napster [S01] est une application de partage de fichiers musicaux *mp3*. Le service est fondé par Shawn Fanning et Sean Parker. Napster utilise un modèle P2P à répertoire centralisé pour les données administratives et les index des fichiers *mp3* téléchargeables par les pairs. Un pair se connecte directement sur le serveur. Le protocole basique est simple, avec des primitives de service : *login request*, *login OK*, *login failed*, *user unknown*, *user not accepted*, …

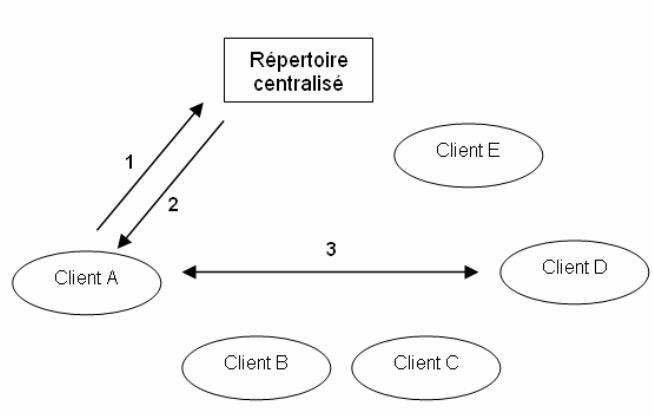

Figure 16. Les échanges dans Napster

Le pair communique au serveur central la liste des fichiers qu'il partage, ainsi qu'un numéro de port TCP où il pourra être contacté pour un téléchargement. Les pairs sont donc les fournisseurs de fichiers, et les échanges vont se faire directement entre eux. La recherche et le téléchargement vont se passer de la manière représentée sur la Figure 16: un pair recherchant un fichier va envoyer sa requête au serveur central (1), qui va lui répondre par une liste triée de pairs qui hébergent le fichier recherché (2). Le pair va alors choisir parmi les réponses celle qui lui convient le mieux (pertinence de la réponse, bande passante, taille du fichier, …) et contacte directement le pair choisi. Le téléchargement s'effectue alors directement entre les deux pairs (3).

4.2.1.2 Le modèle purement décentralisé

Dans un modèle purement décentralisé, chaque pair joue à la fois le rôle d'un client et d'un serveur. Chaque pair est connecté à un ensemble de voisins. Pour lancer une recherche, un pair interroge tous ses voisins en leur envoyant un message de recherche. Ses voisins font de même avec leurs propres voisins. Un champ TTL (Time To Live) est associé au message de recherche pour comptabiliser le nombre de retransmissions restantes. Quand celle-ci est nulle, le message n'est plus renvoyé. Cette méthode de propagation est appelée inondation. Les pairs ayant des fichiers qui répondent à la requête renvoient leur réponse (nom du fichier + leur adresse IP) au voisin qui leur a retransmis la requête. La réponse remonte ainsi de proche en proche jusqu'au pair qui a initié la requête. Le pair initiateur de la requête va ensuite choisir les fichiers à télécharger en envoyant directement une requête de téléchargement au pair qui possède le fichier. Cependant cette inondation est coûteuse en bande passante et les recherches sont plus lentes que dans les réseaux centralisés (par exemple Napster). Nous présentons dans cette section l'application Gnutella [MPV06] comme exemple du modèle P2P purement décentralisé.

Gnutella [MPV06] est un protocole de fichiers partagés. La première version (version 0.4) date de mars 2000. Cette application a montré qu'il est possible de proposer aux usagers un service qui ne repose sur aucune infrastructure physique concrète mais sur

un simple logiciel distribué au sein d'une communauté d'utilisateurs. Dans Gnutella, chaque pair dispose de 6 types de messages principaux:

- *Ping* : Utilisé pour découvrir les autres pairs sur le réseau. Un pair qui reçoit un *Ping* doit répondre avec un ou plusieurs *Pong*.
- *Pong* : C'est la réponse à un *Ping*; ce message contient l'adresse IP et le port du pair qui répond, ainsi qu'une information sur les fichiers et leurs tailles partagés par celui-ci.
- *Query* : Message pour la recherche de fichier sur le réseau. Un pair qui reçoit un *Query* répondra par un *Query Hit* s'il trouve une correspondance avec un ou plusieurs de ses fichiers partagés et les critères de recherche.
- *Query Hit* : C'est la réponse à un *Query*.
- *Get*: téléchargement des données.
- *Push* : Permet de télécharger des données depuis un pair qui se trouve derrière un Firewall.

Bien que le réseau supporte des milliers d'usagers, son mécanisme de *flooding* amène des limites, et ne semble pas supporter le passage à l'échelle (Réseau rapidement inondé par des Ping et des Pong). Son mécanisme de recherche présente une limite, fixée par la valeur initiale du TTL. Ainsi une requête peut être stoppée par une expiration de TTL sans avoir parcouru l'intégralité du réseau et retourner une réponse négative.

4.2.2 Les réseaux structurés

Les réseaux structurés pallient les limites des réseaux non-structurés en utilisant un protocole de contrôle de la topologie du réseau virtuel. En effet, chaque donnée est stockée dans un nœud tel que la relation entre la donnée et sa localisation soit déterminée par une table de hachage distribuée (DHT - Distributed Hash Table). La première proposition de DHT est apparue en 2001 et a été proposée par *Ratanasamy et al*. Elle porte le nom de CAN [FHKS+01]: *A Content Adressable Network* (un réseau au contenu adressable). Ce nom est maintenant utilisé de manière générique pour désigner l'ensemble des infrastructures de type DHT [FG03]. Dans une table, un tuple *(DataID, NodeID)* qui indique qu'une donnée *DataID* est stockée sur le nœud *NodeID*. La recherche sur les réseaux structurés est plus rapide que sur les non-structurés car les nœuds connaissent les données stockées par leurs voisins.

Au niveau des implémentations il existe plusieurs grandes classes de DHTs qui se différencient principalement par le modèle topologique sur lequel elles reposent. En outre au sein de chaque classe, il existe de nombreuses propositions de DHTs. Dans la suite de cette section, nous allons présenter une DHT majeure pour chaque classe. Nous allons détaillons en particulier Chord [MPV06, SMKK+01] et Pastry [RD01] qui sont des infrastructures sur lesquelles nous avons travaillé.

4.2.2.1 Propositions à topologie annulaire

Organiser les pairs selon un anneau est envisagé tout d'abord dans Chord [SMKK+01] qui propose d'organiser les pairs selon un anneau simple. De nombreuses DHTs [FHKS+01] utilisent de manière plus au moins directe l'organisation des pairs selon un anneau. C'est par exemple le cas de Pastry [RD01] qui utilise un anneau pour finaliser son routage. Néanmoins, nous avons choisi de ne pas détailler ces infrastructures dans cette section mais plutôt dans celle qui correspond au mieux à leur modèle topologique générale. Nous présentons ici la DHT Chord [SMKK+01].

Le Protocole Chord. Chord [SMKK+01] est un protocole de recherche P2P pour les applications Internet : pour une clé donnée, Chord y associe un pair. Chord est déployé dans plusieurs applications: tout d'abord, dans CFS (*Collaborative File System*) [DKKM+01] qui est un système de fichiers distribué à l'échelle de l'Internet, ensuite dans ConChord [ACMR02] qui utilise CFS afin de fournir une infrastructure distribuée pour la délivrance de certificats de sécurité et enfin dans DDNS [CMM02] qui propose une implantation P2P du DNS (*Domain name Service*).

La DHT Chord [SMKK+01] repose sur une topologie en anneau et utilise donc une notion de successeurs et de prédécesseurs dont elle garde trace dans sa table de routage à *m* entrées. Donc tout utilisateur peut contacter directement *m* nœuds du réseau pour transmettre ses requêtes. Une fonction de hachage régulière génère un identifiant pour chaque pair à partir de son adresse IP. Ensuite, chaque pair est placé dans l'anneau de manière à ordonner les identifiants par ordre croissant. Ainsi, chaque pair d'identifiant *n* est responsable de l'intervalle de clés *[precedent(n), n]*.

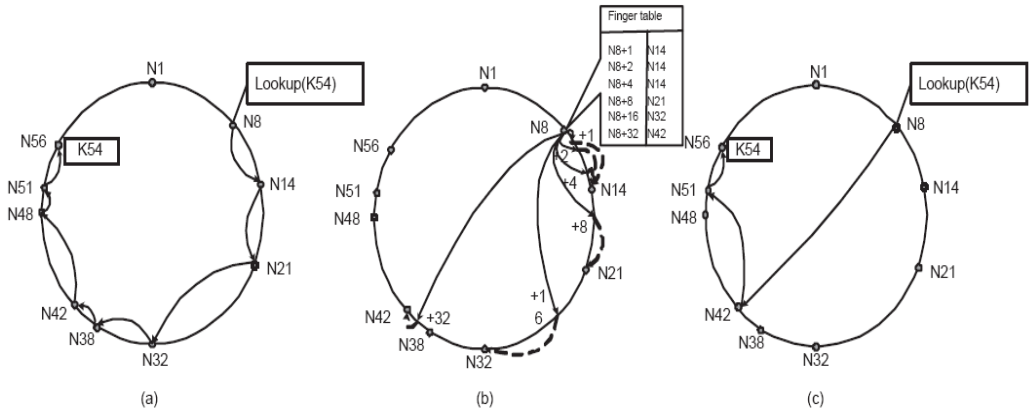

Figure 17. Extrait de [SMKK+01]. (a) Acheminement d'une requête par parcours de l'anneau. (b) Définition des fingers d'un pair. (c) Acheminement d'une requête en utilisant les fingers.

Pour un pair donné, la simple connaissance de son prédécesseur et de son successeur n'est pas suffisante pour garantir une bonne performance de l'anneau, notamment en termes de nombre de sauts par requête. La Figure 17.a illustre ce type de problème avec

un anneau contenant 10 pairs avec une plage d'adressage comprise dans l'intervalle [0, 64[. On peut constater que le routage d'une requête d'un pair de clé N8 à destination de la clé *K54* nécessitera 8 sauts. Afin de pallier ce problème, pour un espace de clés compris dans l'intervalle $[0, 2^m[$, chaque pair n se voit doté d'une entrée vers les pairs, appelés fingers, de clé suivante $(n + 2^{i-1})$ avec $1 \leq i \leq m$. Ainsi, le nombre maximum de pairs parcourus pour acheminer une requête s'exprime en termes de $O(log(N))$. La Figure 17.b présente la table de routage du pair N8 muni de fingers et la Figure 17.c montre que la même requête de clé *K54* est cette fois acheminée en 3 sauts. La table de routage d'un pair Chord compte ainsi O(log N) entrées.

Insertion, départ et maintenance. L'insertion d'un nouveau nœud dans une communauté se fait par la simple recherche de son suivant dans l'anneau. C'est ensuite le processus de maintenance qui, exécuté régulièrement, va prendre en charge le maintien de la cohérence des tables de routage des pairs concernés par l'arrivée du nouvel élément. En effet, à intervalles réguliers chaque nœud vérifie la validité des informations détenues dans sa table de routage à savoir: son suivant et les *fingers*.

La vérification de son suivant s'effectue en lui demandant son précédent. Si la réponse est soi-même, le suivant est correct. Si le résultat est l'identifiant d'un nœud situé entre soi et son suivant, il devient alors le nouveau suivant, et est notifié que le nœud courant est le nouveau précédent.

La vérification des *fingers* est similaire à la construction de la table et consiste à rechercher pour chaque entrée le nœud correspondant d'identifiant *suivant* $(n + 2^{i-1})$, avec $1 \leq i \leq m$. La table est mise à jour si jamais un pair différent des pairs actuels est trouvé. Par le biais de ces vérifications régulières, Chord garantit le maintien de la cohérence de l'anneau.

Dans le cas d'un départ de nœud, c'est le processus de maintenance qui permet la mise à jour des informations maintenues par les pairs. En outre, afin de limiter les échecs de requêtes survenant dans l'intervalle de temps situé entre la disparition d'un nœud et l'exécution de ce processus, chaque nœud maintient une liste de suivants, qui peuvent être utilisés alternativement au suivant défaillant.

4.2.2.2 Propositions fondées sur l'algorithme de Plaxton

Une seconde famille de DHTs repose sur l'algorithme de Plaxton [PRR97]. Parmi les DHTs de cette famille, nous considérons les trois suivantes. Pastry [RD01] est une DHT qui est utilisée dans PAST, une application de stockage de fichiers. Tapestry [ZHSR+04, ZJK01] est une autre proposition qui est déployée dans OceanStore [KBCC+00], une application de stockage de fichier à grande échelle. Enfin, P-Grid [DHA03] est une proposition préliminaire à celles des DHTs actuelle qui n'effectue pas d'association directe entre l'identifiant des nœuds et celui des ressources. Dans cette section, nous présentons la DHT Pastry [RD01].

Le protocole Pastry. Pastry [RD01] est un protocole de routage et de localisation P2P: pour une clé donnée, Pastry associe un pair dont l'identifiant est le plus proche

numériquement de la clé. Son architecture repose sur un modèle P2P décentralisé et utilise une approche hybride pour le routage: un routage est effectué grâce à l'algorithme de Plaxon [PRR97], mais les dernières étapes utilisent la notion de voisins virtuels organisés selon une topologie annulaire [GGGR+03] similaire à Chord [SMKK+01]. Pastry est actuellement utilisé dans deux applications P2P qui sont : PAST [DR01], une application distribuée de stockage de fichiers et Scribe [RKCD01] une application P2P de diffusion d'évènements qui peuvent être reçus en souscrivant à un thème particulier.

Chaque pair Pastry est muni d'un identifiant de 128 bits qui est le résultat d'une fonction de hachage appliquée à l'adresse IP ou la clé publique du pair. Dans un réseau contenant N pairs Pastry, pour une clé donnée, le protocole est capable d'associer un pair dont l'identifiant est le plus proche numériquement en $log_2^b N$ sauts (b est un paramètre de configuration typiquement égal à 4), en moyenne.

Une table de routage Pastry contient trois catégories d'entrées. La première contient l'ensemble des voisins virtuels du pair considéré en termes de distance numérique de l'identifiant de pair. La seconde catégorie est la table de routage à proprement parler du pair. Chaque ligne l contient $2^b - 1$ entrées dont les l premiers chiffres sont communs à ceux du pair considéré. On peut noter que chaque entrée de la table est celle dont la distance réelle est la plus petite entre le pair et l'entrée considérée. Enfin, la dernière catégorie de la table contient l'ensemble des voisins réels du nœud. La métrique choisie pour évaluer la distance réelle entre pairs est le nombre de sauts IP. Ainsi, une table de routage Pastry contient $(2^b - 1) * (log_2^b N) + 2l$ entrées. La Figure 18 présente un exemple simplifié de table de routage Pastry. Dans cet exemple, b est fixé à 2 et la longueur des identifiants est de 8 digits. On y voit par exemple que la deuxième colonne de la cinquième ligne (soit $l = 4$) possède une entrée pour un pair possédant un préfixe commun de 4 digits, et un cinquième digit égal à 1.

Le principe de routage de Pastry est le suivant : lorsqu'un message muni d'une clé K arrive dans un pair n, celui-ci vérifie tout d'abord si la clé correspond à un pair d'identifiant voisin virtuellement. Dans ce cas, le message est routé directement vers le pair correspondant. Sinon, le pair n détermine m, le nombre de chiffres communs à son identifiant et à K, consulte sa table de routage pour trouver un pair qui présente $m+1$ chiffres en commun et lui fait suivre le message. Si aucun pair ne correspond, le pair n cherche un autre pair ayant m chiffres en commun et dont l'identifiant est le plus proche de K. Le message est arrivé à destination lorsqu' aucun voisin virtuel ni entrée de la table de routage ne permet de se rapprocher plus de la clé K.

Pair **10233102**			
Voisins virtuels			
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232
Table de routage			
-0-2212102	**1**	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	**2**	10-3-23302
102-0-0230	102-1-1302	102-2-2302	**3**
1023-0-322	1023-1-000	1023-2-121	**3**
10233-0-01	**1**	10233-2-32	
0		102331-2-0	
		2	
Voisins réels			
13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figure 18. Extrait de [RD01]. Table de routage d'un pair d'identifiant 10233102 avec b=2

4.2.2.3 Propositions fondées sur un hypercube

La topologie de Chord repose sur un espace cartésien circulaire à une seule dimension. CAN [FHKS+01] étend cette topologie à un espace multidimensionnel et repose sur une topologie torique de dimension d. On parle aussi de topologie en hypercube, car CAN peut être vu comme un cube de dimension d, dont les extrémités de chaque dimension se rejoignent.

Le protocole CAN. Dans une communauté CAN, l'ensemble de l'espace est partitionné parmi l'ensemble des pairs inscrits de sorte que chaque pair soit responsable d'un sous-ensemble unique. La Figure 19 présente un exemple de partitionnement dans un espace à deux dimensions, mis à plat, munis de coordonnées comprises entre 0 et 1 sur chaque axe. Elle permet notamment de voir que l'ensemble de l'espace est constamment alloué.

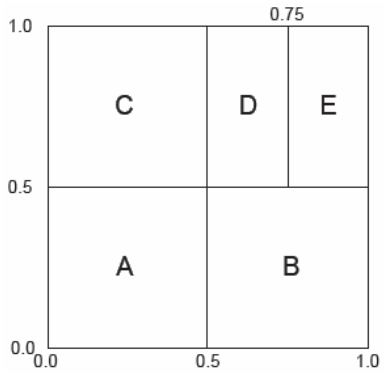

Figure 19. Exemple d'espace à deux dimensions avec 5 pairs (Extrait de [FHKS+01])

Le principe de stockage d'un couple (*clé, valeur*) est le suivant: une fonction de hachage unique, appliquée à la clé, permet de déterminer un point de coordonnées P appartenant à l'espace. Le pair responsable du sous-ensemble dans lequel P est inscrit est alors responsable du stockage du couple (*clé, valeur*). Ensuite, pour récupérer une valeur stockée, un pair applique à une clé la fonction de hachage, en récupère une coordonnée de point P' et utilise alors le protocole de routage de CAN pour contacter le pair responsable du point P' considéré.

4.2.3 Réseau super-pair

Les réseaux P2P structurés sont considérés comme purs car tous les nœuds jouent le même rôle et possèdent les mêmes fonctionnalités. Cependant, le réseau super-pair est un hybride entre les réseaux non structurés et le client-serveur (voir Figure 20). Comme le réseau client-serveur, certains pairs, les super-pairs, jouent le rôle du serveur pour un ensemble de pairs et effectuent des fonctions complexes comme le traitement des requêtes, le contrôle d'accès et la gestion des méta-données. Comme le réseau pair-à-pair pur, les super-pairs sont organisés en mode pair-à-pair, choisis automatiquement en fonction de leurs caractéristiques (bande passante, vitesse de traitement, capacité mémoire) et remplacés en cas de présence d'une faute.

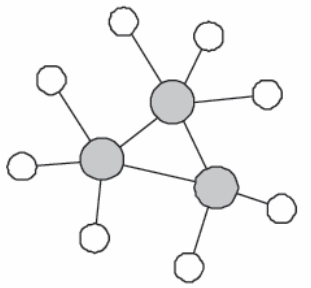

Figure 20. Topologie P2P construite selon le modèle super-pair

La recherche d'une donnée sur ce type de réseau est simplement effectuée par l'envoi de la requête au super-pair correspondant. Celui-ci cherche ensuite le pair qui contient la donnée et transmet son adresse au nœud demandeur. Dans ce type de réseau, le placement de données se fait sur le nœud demandeur ou sur un nœud calculé automatiquement.

Les avantages principaux de ce type de réseaux sont l'efficacité et la complétude du résultat d'une recherche. Le temps nécessaire pour trouver une donnée par interrogation du super-pair est plus faible que par inondation. De plus, le super-pair exploite bien la différence de capacité des pairs en termes de vitesse de traitement, capacité mémoire ou bande passante. *Kazaa* [KAZA10] et *JXTA* [Jxta10] sont des exemples de ce type de réseaux.

Remarques. Certaines classifications des architectures P2P considèrent les modèles hybrides (super-pair) et centralisés comme identiques. Le modèle centralisé étant alors un cas particulier du modèle hybride contenant un seul super-pair. Toutefois, nous

préférons conserver la distinction entre ces deux modèles, car le modèle hybride peut être vu comme la composition des modèles purement distribués et centralisés, en ce sens que, chaque super-pair agit comme une entité centrale pour les pairs qui lui sont connectés, mais que l'organisation des super-pairs suit le modèle purement décentralisé.

4.2.4 Synthèse générale

Le réseau P2P est un réseau distribué où les entités appelées pairs jouent le rôle de client et serveur. Les approches dans le domaine P2P reposent sur la construction d'un réseau virtuel sur le réseau physique. Il existe trois classes principales de réseaux virtuels: les non-structurés, les structurés et les super-pair (hybride). Le réseau P2P non-structuré est créé de manière non-déterministe. Dans ce type de système, le placement des données dans le réseau est indépendant du réseau virtuel lui-même. Par conséquent, la recherche d'une donnée sur ce type de réseau est réalisée par inondation (par exemple Gnutella, version 0.4). Le réseau P2P structuré est constitué de pairs strictement équivalents, il pallie les limites des réseaux non-structurés en utilisant un protocole de contrôle de la topologie du réseau virtuel. Ce réseau garantit une localisation efficace grâce à l'utilisation d'un index distribué, notamment les réseaux à base de DHT. Enfin le réseau P2P hybride utilise des super-pairs qui présentent des fonctions avancées par rapport aux autres pairs du réseau : indexation et recherche des fichiers.

Les caractéristiques du système P2P sont nombreuses. La principale est la décentralisation qui confère aux applications P2P et, comparativement au système client-serveur, un bon équilibre de la charge, un meilleur passage à l'échelle, une répartition des coûts de mise en œuvre et maintenance du service offert et une bonne tolérance aux fautes. Les autres caractéristiques de ce système sont: (a) l'auto-organisation qui permet aux communautés de pairs de délivrer leur service de manière autonome, (b) la connectivité ad-hoc des pairs qui induit une dynamicité des données et des ressources, (c) l'utilisation d'un overlay qui abstrait les caractéristiques physiques des éléments.

Nous évaluons les différents types de systèmes P2P, décrit précédemment, selon les critères suivants [DGY03] : autonomie, efficacité, expressivité de la requête, tolérance aux pannes, sécurité et qualité de service. Le tableau ci-dessous résume nos conclusions:

Tableau 2. Comparaison des systèmes P2P

	Non structuré	**Hybrides (super-pairs)**	**Structuré**
Autonomie	Forte	Moyenne	Faible
Efficacité	Faible	Moyenne	Forte
Expressivité de la requête	Forte	Forte	Faible
Tolérance aux pannes	Forte	Faible	Forte
sécurité	Faible	Forte	Faible
QoS	Faible	Forte	Forte

Conclusion. Il n'y a pas de solution pleinement satisfaisante. En effet, un seul type de système (non-structuré, structuré ou hybride) ne peut pas bien supporter tous les besoins des applications et satisfaire tous les critères cités dans le tableau ci dessus. Nous pensons donc que différents systèmes P2P (non-structurés, structurés, hybrides) seront mieux adaptés que d'autres selon l'application. Par exemple, un système non-structuré peut offrir une plus grande tolérance aux fautes car aucun pair n'est un point unique de panne. Cependant, les systèmes non-structurés mettent en œuvre des techniques simples pour la localisation des données (par ex. routage des messages par inondation du réseau) ce qui conduit à des problèmes de performances.

En revanche, un système hybride permet de supporter efficacement des recherches complexes et un système structuré peut donner de meilleures garanties de performances en proposant des techniques de routage efficace pour la localisation des ressources dans un réseau P2P. Enfin, on peut imaginer la combinaison de différents systèmes pour une même application, par exemple La DHT pour la recherche par mot-clé et l'hybride pour la recherche plus complexe.

4.3 Solutions de réplication dans les réseaux P2P

La réplication de données dans les systèmes P2P devient un enjeu majeur pour les applications collaboratives, comme les forums de discussion, les calendriers partagés, ou les catalogues e-commerce. En effet, les données partagées doivent pouvoir être mises à jour en parallèle par différents pairs. Les premiers systèmes P2P existants supposent que les données sont statiques et n'intègrent aucun mécanisme de gestion des mises à jour et de réplication. Une mise à jour d'une donnée par le pair qui la possède implique une nouvelle version non propagée à ceux qui répliquent 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 répliques n'est alors garantie. Plus récemment, des solutions de gestion de mise à jour et de réplication ont été proposées pour des systèmes P2P, comme *Freenet* [CSWH00, CMHS+02], *OceanStore* [KBCC+00], *Ivy* [MMGC02], *P-Grid* [DHA03], *WOOT* [OUMS+05, Ophd05], *Logoot* [WUM09], *P2P-DSR* [MAPV06, MPV06a, MP06], *Telex* [BBMS+08] et *Scalaris* [SRHS10].

4.3.1 Freenet

Le système Freenet [CSWH00, CMHS+02] est un réseau P2P fonctionnant au-dessus d'Internet. Contrairement aux réseaux P2P habituels, Freenet assure l'anonymat des échanges et assure une répartition intelligente des données sur le réseau. Le système P2P Freenet utilise une stratégie pour router les mises à jour vers les différentes répliques, mais il ne garantit pas la cohérence des données. En effet, dans le cas d'une mise à jour, la requête est acheminée vers les pairs ayant une réplique. Toutefois, il n'existe aucune garantie que tous les pairs reçoivent la mise à jour, en particulier ceux qui sont absents au moment de mise à jour.

Le principe de Freenet est différent de celui des DHTs notamment sur un point majeur : les pairs ne sont pas identifiés par le hash d'un attribut particulier comme

l'adresse IP ou le nom de la machine. L'utilisation de l'adresse de niveau transport est suffisante. Par contre, les identifiants de fichiers sont le résultat d'une fonction de hachage appliquée à leur description. Dans ce contexte, il n'est fait aucun rapprochement entre les clés des ressources et les pairs. Pour ajouter un nouveau fichier, l'utilisateur envoie au réseau un message d'insertion contenant le fichier, un identifiant global unique (la clé) généré indépendamment de son emplacement géographique et une valeur de TTL. Ensuite, ce message est envoyé aux voisins de l'utilisateur. Le fichier sera enfin propagé le long du chemin établit par la requête d'insertion. Grâce à un mécanisme de routage spécifique à Freenet, le fichier est ensuite stocké sur plusieurs nœuds possédant des clés similaires, ce qui renforce la disponibilité des données. Pendant sa durée de vie, un fichier peut migrer d'un nœud à un autre ou être répliqué sur plusieurs nœuds.

La découverte d'une ressource est effectuée comme suit: chaque pair possède des voisins auxquels il publie la liste des identifiants de fichiers qu'il héberge. Lorsqu'un nœud désire accéder à un fichier, il consulte tout d'abord dans son cache local pour vérifier qu'il ne possède pas le fichier. Sinon, il envoie la requête à un de ses voisins qui possède un fichier dont l'identifiant se rapproche de celui du fichier requis. Ce routage s'effectue de part en part jusqu'à l'atteinte de la ressource requise. Si la ressource n'est pas disponible, la requête sera stoppée du fait de l'expiration de TTL ou par l'absence de nouveaux pairs à contacter. L'identification des messages permet en effet d'éviter les boucles. L'accès à un fichier se fait en parcourant en sens inverse le chemin emprunté par la requête de découverte. Au sein de chaque pair parcouru, le fichier est copié. Par ce biais, Freenet rend très difficile la suppression d'un fichier stocké et permet d'améliorer la qualité du routage car, si une requête pour le même fichier se reproduit, les nœuds intermédiaires peuvent y répondre sans avoir à router la requête jusqu'au pair qui stocke initialement le fichier requis. De plus, la topologie du réseau tend à s'organiser car, au fil des requêtes, les pairs se spécialisent sur des fichiers de clés proches.

L'anonymat est garanti par deux principes fondamentaux: dans le processus de routage, chaque pair ne connaît ni la source ni la destination d'une requête, et chaque fichier est crypté à l'aide d'une clé publique qui est la description non hachée du fichier. Ainsi, un pair ne peut pas savoir quel fichier il héberge.

4.3.2 OceanStore

OceanStore [KBCC+00] fournit un support de stockage persistant de fichiers mutables pour les applications P2P. Toute mise à jour d'un objet génère une nouvelle version non mutable et conservée pour toujours. OceanStore repose sur des liens fiables et rapides entre les pairs et ne permet qu'un nombre restreint de copies.

OceanStore permet la gestion de données modifiables et intègrent des protocoles de cohérence spécifiques. Le modèle de cohérence appliqué dans OceanStore compte sur la non concurrence des écritures, aucune synchronisation n'étant offerte. Ainsi, lorsqu'une collision est détectée, la mise à jour peut tout simplement être annulée avant son application. Certains pairs du système peuvent utiliser un modèle de cohérence encore

plus optimiste en appliquant sans vérification sur un éventuel ordre causal les mises à jour sur leur copie locale. Un mécanisme ordonne les mises à jour sur les données via un algorithme de consensus (évidemment asynchrone dans ce cadre) auprès d'un nombre restreint de pairs sur les copies primaires. Ce consensus est coûteux en communication, il faut préférablement que les copies primaires se trouvent sur des pairs localisés dans une section rapide du réseau.

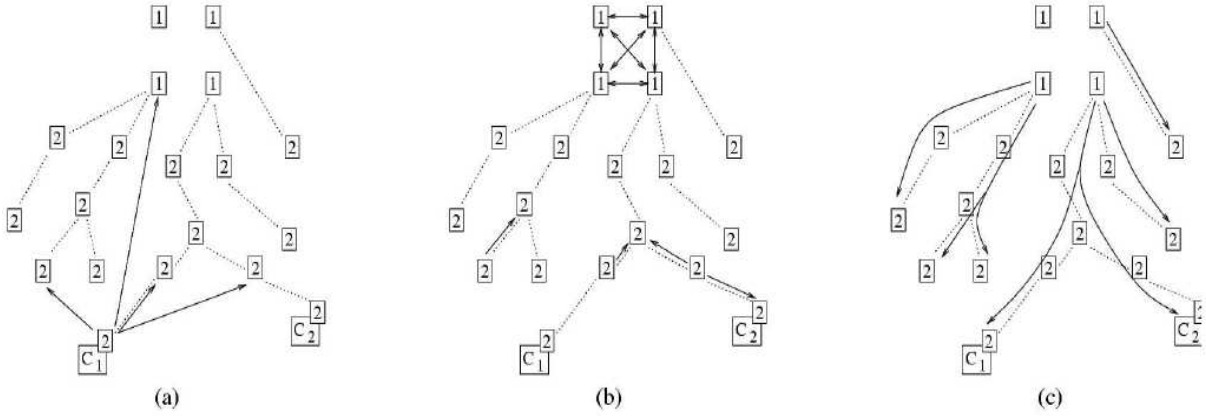

Figure 21. Le parcours d'une mise à jour dans OceanStore
La Figure (a) montre qu'après avoir créé un fichier de mise à jour, le client C2 l'envoie vers l'une des copies primaires ainsi que vers les possesseurs de copies secondaires connues par le site. Illustré par la figure (b), pendant qu'un consensus sur l'ordre des applications des mises-à-jour est réalisé par les pairs primaires, les pairs disposant des copies secondaires propagent la mise à jour et peuvent décider de l'appliquer en négligeant tout ordre. En (c), le consensus une fois réalisé par les pairs primaires, la mise à jour est propagée avec une estampille définissant l'ordre des mises à jour.

Les pairs des copies secondaires sont les sites qui ne participent pas au consensus. Ils reçoivent les mises-à-jour estampillées d'un ordre total de la part des pairs primaires pour cette donnée. Ils peuvent donc appliquer en toute sécurité la mise à jour. Le protocole de consensus d'une mise à jour peut être long et, comme le montre la Figure 21, les pairs secondaires peuvent décider d'appliquer les mises à jour au plus tôt, ce qui est le comportement par défaut dans OceanStore. Cependant, le pair peut recevoir une mise à jour estampillée montrant que sa copie de la donnée est corrompue. Il faut alors reconstruire une donnée cohérente. Ce modèle fonctionne bien lorsqu'il n'y a pas trop d'écritures concurrentes sur une donnée. Les pairs désirant une meilleure garantie de la cohérence de leurs données doivent attendre que les *primary replicas* leur aient diffusé les mises à jour validées.

4.3.3 Ivy

Ivy [MMGC02], comme OceanStore, est un système P2P de gestion de données qui propose le partage de données modifiables pour un nombre restreint d'écrivains. Ivy permet la gestion de données modifiables en intégrant des protocoles de cohérence spécifiques.

Ivy est un système reprenant le comportement d'un système de fichier sans état, tel NFS, en offrant un modèle de cohérence du type *close-to-open*. Toute lecture sur un premier site verra les modifications apportées par une écriture sur un second site, si la fermeture du fichier par le second site a précédé son ouverture par le premier site. Dans Ivy, tout le système de fichiers est contenu dans des journaux, chacun provenant d'un utilisateur du système de fichier. Les journaux sont des listes d'enregistrements qui décrivent chacun une modification du système de fichiers. Ivy fonctionne selon un concept de vue. Un utilisateur ne voit que les modifications des utilisateurs dont les journaux sont référencés dans son bloc de vue ou *block view*.

Chaque fois qu'un utilisateur modifie le système de fichier, un ou plusieurs enregistrements sont rajoutés dans son journal. Pour déclarer ce nouvel enregistrement et le rajouter au journal, il suffit de modifier le pointeur d'enregistrement de l'utilisateur, ou *log-head*, qui est un bloc modifiable seulement par l'utilisateur local. La Figure 22 représente ces structures de données qui définissent le système de fichier vu par un utilisateur. Pour chaque opération sur le système de fichier, par exemple une lecture, l'utilisateur doit parcourir tous les journaux pour avoir le contenu le plus récent de la donnée.

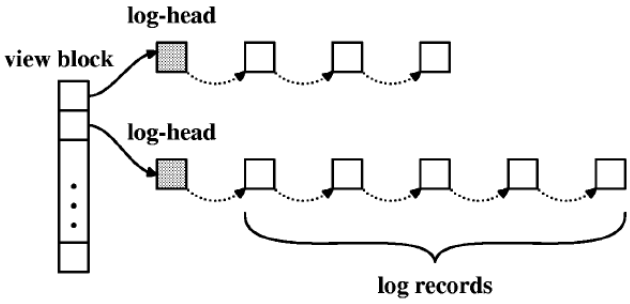

Figure 22. Un exemple de vue dans Ivy

Un exemple de vue dans Ivy avec la liste des enregistrements provenant de chaque site appartenant à la vue. Les cases blanches représentent les journaux qui sont chaînés entre eux ; ce sont des blocs en lecture seule dans Ivy. Les cases grisées sont des blocs modifiables qui contiennent les pointeurs vers le journal le plus récent.

Chaque enregistrement est identifié par un numéro de version incrémenté par le site émetteur et un vecteur référençant les dernières versions connues par le site lors de la création de l'enregistrement. Cela crée un *ordre causal* permettant d'ordonner les enregistrements. Cependant, deux enregistrements peuvent être indépendants et être en conflit sur une écriture. Ivy propose une réparation manuelle de ce genre de conflit via des outils spécifiques au type de la donnée endommagée. L'un des avantages liés à

l'utilisation des journaux est qu'en cas de conflit, on peut toujours revenir à un état antérieur cohérent, au prix du stockage de tous les enregistrements.

4.3.4 P-Grid

P-Grid [DHA03] supporte la réplication mono-maître avec propagation des mises à jour par inondation vers les pairs connectés qui possèdent une copie secondaire. Les pairs déconnectés ne voient pas les mises à jour de leurs copies. P-Grid n'offre donc que des garanties probabilistes de cohérence.

Dans les DHT traditionnels, le hachage est uniforme et conduit à une répartition de la charge de stockage elle aussi uniforme. Les nœuds n'ont aucune stratégie individuelle, et ne peuvent adapter la charge à leur capacité. Au contraire, P-Grid se propose d'avoir des clés de longueur variable. Chaque nœud peut ensuite choisir les clefs dont il est responsable. Les clés considérées sont binaires, soit dans *{0,1}**. A chaque nœud *n* est associée une clé *chemin(n)*, appelée chemin du nœud. Celui-ci détermine quelles clés sont sous la responsabilité du nœud : si une clé à pour préfixe *chemin(n)*, alors *n* est responsable du stockage de la paire contenant la clé. Un nœud peut avoir pour chemin un préfixe du chemin d'un autre nœud, même si cela n'est que transitoire dans le système proposé qui produit un arbre 'libre de préfixe'. Même si les clés ne sont pas uniformément réparties, la recherche reste logarithmique. Un nœud est une réplique d'un autre si les deux partagent le même chemin. Ils doivent donc contenir les mêmes données.

Nous pouvons identifier deux sous problèmes pour équilibrer la charge dans P-Grid:

- équilibrage de la charge de stockage : la charge de stockage est le nombre de clés sous la responsabilité d'un nœud. L'équilibrer est un problème local, puisqu'un nœud peut de lui-même savoir s'il est sur ou sous chargé.
- réplication uniforme : nous définissons le facteur de réplication comme le nombre de pairs ayant le même chemin. Equilibrer ce facteur pour tous les chemins différents est important pour assurer que toutes les données soient également répliquées. Ce problème est global, puisqu'un nœud ne peut pas déterminer seul ce facteur de réplication.

Algorithme d'équilibrage de la charge de stockage. Tout algorithme doit être local à chaque nœud. Initié par un processus complexe, il se résume à une interaction entre deux nœuds. L'objectif de l'algorithme est de répartir la charge d'un nœud entre C_{min} et C_{max} (on peut supposer que les capacités des nœuds sont égales). Les opérations possibles sont alors :

- fusion de nœuds, fabrication de répliques : si deux nœuds ont le même chemin et si la taille totale des données est inférieure à C_{max}, alors les nœuds peuvent fusionner.
- expansion de chemins : deux nœuds ayant le même chemin avec une charge trop importante peuvent augmenter la taille du chemin d'un bit, chaque pair prenant en charge les données d'une des branches.

- rétractation de chemins : si deux nœuds ont le même chemin sauf le dernier bit, et si la taille totale des données est inférieure à C_{max}, alors les nœuds peuvent fusionner.

Réplication uniforme: du local au global. L'algorithme exact étant compliqué, nous donnons ici uniquement un aperçu pour expliquer son fonctionnement. L'idée est qu'un nœud, en connaissant son facteur de réplication et celui dans d'autres branches, doit prendre la décision de changer de branche ou non. Considérons un arbre très simple avec uniquement 2 feuilles (2 chemins). Le chemin 0* a un facteur de réplication N1, N2 pour le chemin 1*. On suppose N1>N2. (Voir Figure 23)

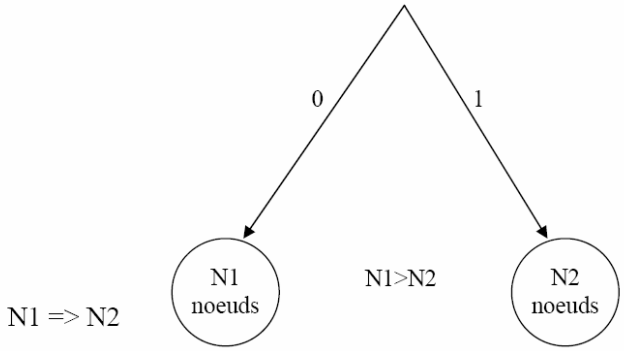

Figure 23. Equilibrage dans un arbre équilibré

Pour équilibrer les facteurs de réplication, il faut que des nœuds de 0* migrent vers 1*, soit plus précisément (N1-N2)/2 nœuds. Or, les nœuds sont indépendants, nous voulons garder un algorithme local. Au lieu d'introduire un système complexe pour désigner ces nœuds, nous implémentons un algorithme probabiliste simple : chaque nœud à une probabilité *p* de changer de branche, avec $p = (N1-N2)/(2*N1)$.

Pour connaître le facteur de réplication d'autres voisins, chaque nœud doit faire un sondage. Pour cela, il peut envoyer un petit nombre de demandes, et compter le nombre de réponses. Il applique ensuite cet algorithme sur ces données. La situation est plus compliquée pour des arbres non équilibrés, puisqu'il faut pondérer le nombre de requêtes. L'algorithme équilibre alors les facteurs de réplication par profondeur.

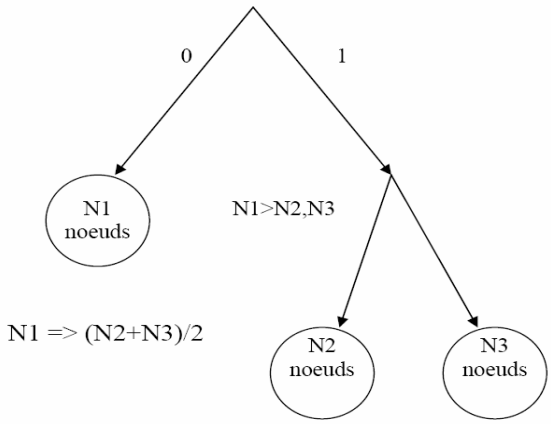

Figure 24. Equilibrage dans un arbre non équilibré

4.3.5 WOOT

WOOT [OUMS+05, Ophd05] est un algorithme proposé pour assurer la cohérence dans le cadre de l'édition collaborative répartie sur le réseau P2P. Cet algorithme de réplication optimiste assure la convergence, préserve les intentions et passe à l'échelle pour les structures linéaires, c'est à dire de structures munies d'un ordre total. WOOT ne repose pas sur le modèle de transformées opérationnelles (OT). L'idée de cette nouvelle approche est simple. Elle consiste à diffuser les relations avec les opérations au lieu de recalculer celle-ci lors de l'intégration. En effet, lorsqu'une opération est générée, les relations sont connues, alors qu'à la réception les algorithmes OT doivent les recalculer. WOOT repose sur une fonction de linéarisation monotone des ordres partiels formés par les relations entre les différents éléments de la structure. Il ne nécessite ni serveur central, ni ordre global, ni vecteur d'horloges.

Quand un utilisateur observe la chaîne de caractères "ABC" et qu'il insère "2" à la position 2, en réalité, il insère "2" entre A et B. Respecter l'intention des effets de cette opération consiste à préserver la relation 'A' < "12" < 'B' sur tous les états futurs de la chaîne. Dans le modèle OT, l'opération insert(2, "12") est exécutée et diffusée. Dans WOOT, l'opération insert(2, "12") est exécutée, mais l'opération insert('A' < "12" < 'B') qui sera diffusée. Il faut donc changer le profil des opérations.

Le problème qui se pose est comment exécuter l'opération insert ('A' < "12" < 'B') si localement le caractère 'A' est supprimé ? La solution est simple: le caractère 'A' n'est pas détruit mais marqué comme invisibles seulement. Ainsi 'A' existera toujours. Bien sûr, si les caractères ne sont pas détruits, WOOT consomme plus de mémoire et génère des fichiers plus volumineux.

Le principe de fonctionnement de l'algorithme de WOOT est le suivant: quand un utilisateur demande à modifier une réplique, l'opération correspondante est générée. Cette opération est immédiatement intégrée en local, puis diffusée aux autres sites. Elle est ensuite reçue par les autres sites, pour y être enfin intégrées. L'opération doit être

intégrée dans la copie locale afin que le caractère inséré soit positionné par rapport aux caractères invisibles, qui ont été détruits, comme il le sera sur les autres sites.

L'algorithme ne fait donc aucune hypothèse sur l'ordre d'arrivée des messages de diffusion des opérations. Il impose moins de contraintes sur l'algorithme de diffusion des patches. Des algorithmes classiques de diffusion sur réseau P2P peuvent donc être utilisés. La Figure 25 montre un exemple de génération d'un ordre partiel dans WOOT.

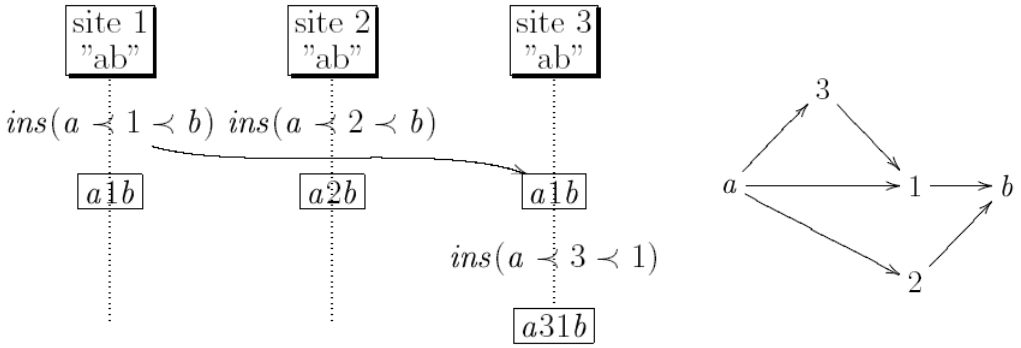

Figure 25. Exemple de génération d'un ordre partiel dans WOOT

Commentaires. L'algorithme WOOT a été conçu pour l'édition collaborative sur réseau P2P. En particulier, il est capable de passer à l'échelle puisqu'il n'utilise *aucun ordonnancement* sur la diffusion des messages (pas de vecteur d'horloge notamment). WOOT se limite à la réconciliation de structure linéaire. Une chaîne de caractères, comme le contenu textuel d'une page Wiki, est une structure linéaire typique. WOOT peut ainsi manipuler une page Wiki comme une séquence de caractères, comme une séquence de mots, ou séquence de lignes. Cependant, WOOT est difficilement extensible à d'autre type de structures de données. Pour assurer la convergence des répliques, tous les sites doivent trouver la même extension linéaire. L'objectif de cette approche est donc de garantir la convergence en utilisant une fonction de linéarisation. WOOT utilise un modèle de données particulier et un profil d'opération spécifique. Pour appliquer l'algorithme au cas d'un Wiki, il est donc nécessaire de stocker les données du wiki (les pages) dans ce modèle.

4.3.6 Logoot

Logoot [Wphd10, WUM09] est un algorithme de réplication optimiste qui assure la cohérence de données pour des structures linéaires. Logoot peut être utilisé sur des réseaux P2P structurés et non-structurés. Contrairement à WOOT, Logoot ne nécessite pas de garder un objet supprimé comme invisible au lieu d'être retiré du modèle du document. Par conséquent, le surcoût reste linéaire en fonction de la taille de document au cours de la durée d'édition et aucune procédure de purge n'est nécessaire. Donc, Logoot est capable de supporter un passage à l'échelle en terme de nombre d'utilisateurs ou en terme de nombre d'édition.

Logoot est basée sur le modèle d'édition collaborative CRDT [SP07]. Dans cette approche, les modifications produites localement sont ré-exécutés sur des répliques distantes. Il n'y a pas d'ordre total sur les opérations, par conséquent, des opérations peuvent être exécutées dans des différents ordres. L'idée principale est d'utiliser un type de données où toutes les opérations commutent. Combiné avec le respect de la relation de causalité entre les opérations, cette commutation garantit les critères de convergence.

Afin d'éviter l'utilisation d'horloge vectorielle pour obtenir la causalité, Logoot utilise un identifiant de position sur la base d'une liste de nombres entiers pour chaque ligne de document. Avec une telle identification, une ligne peut être supprimée du document, sans affecter l'ordre des lignes restantes.

Modèle de donnée de Logoot. Logoot utilise un modèle de donnée spécifique. En effet, un document Logoot est composé de lignes définies par : <*pid*, *content*> où *content* est le contenu de la ligne et *pid* un unique identifiant de position. Deux lignes virtuelles l_B et l_E représentent le début et la fin du document.

L'idée principale pour insérer une ligne est de générer une nouvelle position A telle que $P < A < N$ où P est la position de la linge précédente et N la position de la ligne suivante. Pour rendre les opérations commutatives, les identifiants de positions doivent être uniques. Aussi, puisqu'un utilisateur peut toujours insérer une ligne, Logoot doit être capable de produire une position A telle que $P < A < N$ pour tout P et N (tels que $P < N$)

Modification d'un document Logoot. Un document Logoot étant plutôt destiné à l'édition asynchrone, un utilisateur peut insérer plusieurs lignes en une fois. Par conséquent, les auteurs de Logoot proposent un algorithme capable d'insérer N lignes entre deux positions, ex. p et q. Pour insérer N lignes entre une ligne à la position p et une ligne à la position q, l'algorithme génère une liste de position liste telle que $p < list[1] < \ldots < list\ [N] < q$. Pour cela, l'algorithme calcule une liste contenant les N positions les plus petites possibles comprises entre p et q sur une réplique hébergée sur un site identifié par s.

Intégration des modifications distantes. Les opérations d'insertion et de suppression peuvent être intégrées dans des documents situés sur un réseau P2P. En effet, Logoot utilise un algorithme de recherche dichotomique pour trouver la position dans le document correspondant à l'identifiant de position. En outre, l'intégration d'une opération de suppression peut retirer la ligne du modèle du document, étant donné que l'ordre total des lignes restantes n'est pas affecté. Ainsi, cette suppression permet de libérer un identifiant de position qui pourra être réutilisé. Ce mécanisme permet donc de réduire la croissance des identifiants de positions.

4.3.7 DSR-P2P

DSR-P2P (*Distributed Semantic Reconciliation*) [MAPV06, MPV06a, MP06] est un algorithme distribué de réconciliation conçu pour les réseaux P2P. Il s'agit d'une version distribuée d'IceCube. Pour permettre la collaboration au sein d'un réseau P2P, DSR-P2P supporte des données riches en sémantique et propose une solution de

réplication multi maître optimiste assurant la haute disponibilité. Cette solution se base sur une réconciliation sémantique en vue de réduire les conflits. Elle exploite la distribution et le parallélisme pour le passage à l'échelle et la réduction des latences, et prend en compte le dynamisme des pairs.

DSR-P2P utilise la table de hash distribuée (DHT) comme modèle de réseaux P2P, la réplication de données dans DSR-P2P se fait en trois phases :

1) *Production local d'action*: un nœud exécute localement des actions pour mettre à jour une réplique en respectant les contraintes définies par l'utilisateur de l'application "*user-defined constraints*",
2) *Stocker l'action dans le réseau P2P*: les actions avec les contraintes associées sont stockées dans la DHT,
3) *Réconciliation distribuée des actions*: les nœuds appelés " *réconciliateur* ", récupèrent les actions et les contraintes du réseau P2P et produisent un ordre global de ses actions avec une résolution distribuée des conflits en se basant sur les sémantiques de l'application.

Dans DSR-P2P, nous distinguons les nœuds suivants :

- *nœuds de réplique*: qui tiennent une réplique locale,
- *nœud réconciliateur*: qui est un nœud de réplique qui participe à la réconciliation repartie,
- *nœud fournisseur*: un nœud dans la DHT qui stocke des données consommées ou produites par les nœuds réconciliateurs.

L'algorithme DSR-P2P exécute la réconciliation répartie en 6 étapes comme le montre la Figure suivante :

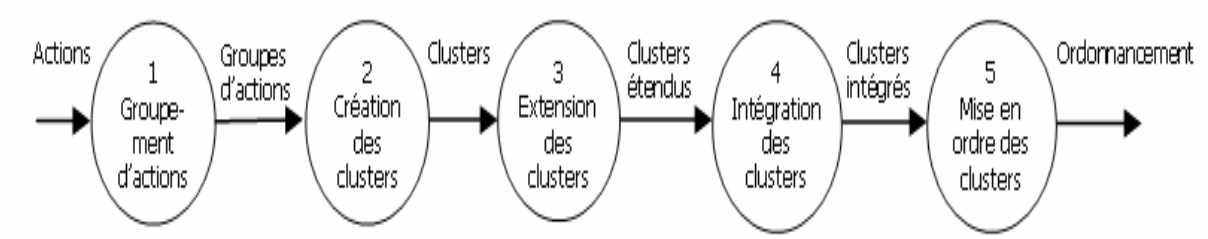

Figure 26. Les étapes de DSR-P2P

1) *Allocation des nœuds* : choisir un sous ensemble de "*nœud de réplique"* pour participer à la réconciliation. N'importe quel nœud connecté peut commencer la réconciliation en invitant d'autres nœuds.
2) *Groupement d'actions*: les réconciliateurs prennent des actions du journal d'actions et mettent les actions qui essayent de mettre à jour les mêmes items d'objet dans le même groupe.

3) *Création des clusters* : les réconciliateurs prennent les groupes d'actions du journal d'actions et le divisent dans des clusters d'actions en conflit et sémantiquement dépendantes.
4) *Extension des clusters*: des contraintes définies par l'utilisateur ne sont pas prises en compte dans la création des clusters. Donc, dans cette étape, les réconciliateurs étendent les clusters en ajoutant de nouvelles actions en conflit, selon les contraintes définies par l'utilisateur.
5) *Intégration des clusters* : l'extension des clusters mène à la superposition des clusters (une superposition a lieu quand l'intersection de deux clusters produit un ensemble non nul d'actions). Dans cette étape, les réconciliateurs mélangent les clusters superposés.
6) *Mise en ordre des clusters* : dans cette étape, les réconciliateurs prennent des clusters de l'ensemble de clusters et mettent en ordre les actions des clusters. Les actions ordonnées associées à chaque cluster sont stockées dans l'objet de réconciliation *ordonnancement* (*S*) ; la concaténation de toutes les actions ordonnées des clusters compose l'*ordonnancement global* qui est exécuté par tous les nœuds de répliques.

4.3.8 Telex

Dans le cadre du projet européen Grid4All l'équipe Regal de l'INRIA a développé la plate-forme Telex [BBMS+08] pour le support des applications collaboratives. Telex prend en charge les aspects complexes et communs à toutes les applications : la réplication des données, la détection des conflits de mise-à-jour, et leur résolution cohérente sur l'ensemble des sites. Telex sépare clairement la logique applicative des fonctions système. L'application communique à Telex les opérations sur les données et les relations sémantiques entre opérations. Telex permettra ainsi le développement d'applications collaboratives pair à pair innovantes.

Telex propose un modèle d'exécution optimiste: l'application accède à une copie locale des données partagées, sans synchronisation préalable avec les autres sites. Elle progresse en exécutant des opérations de façon locale. Telex exécute en arrière-plan un protocole de convergence qui garantit à terme la cohérence des différentes copies des données en utilisant un graphe des conflits entre opérations concurrentes (*ACG : Action-Constraint Graph*) [SBK04]. Le modèle d'exécution optimiste de Telex permet aux utilisateurs de ne pas subir les latences réseau et éventuellement de travailler hors ligne. Un certain nombre d'applications coopératives ont été développées au-dessus de Telex, par exemple un calendrier coopératif ou un gestionnaire de bibliothèque de document.

Telex utilise un système de fichier pour stocker les documents. Comme objet de réplication, Telex utilise une structure de donnée appelé *multilog* : consiste à un graphe dont les nœuds présentent les actions de mise à jour et les arrêtes présentent les contraintes entre ces actions. En effet, les opérations de chaque utilisateur sont journalisées dans un log. Dans Telex, chaque log est répliqué aux autres sites en utilisant un système P2P de fichier appelé VOFS [CTVK+07] (voir Figure 27).

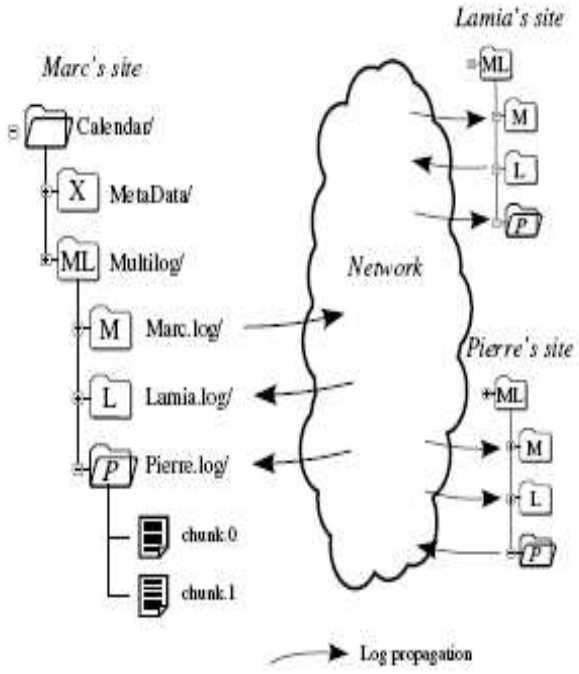

Figure 27. Stockage de document dans Telex

Architecture. L'architecture détaillée de Telex est présentée par la Figure 28. Le cycle d'interaction entre une application et Telex est comme suit. Les requêtes d'utilisateurs de l'application sont traduites d'abord sous forme d'actions et contraintes avant d'être envoyées ensuite à Telex. Comme retour, Telex calcule l'ordre possible des opérations en se basant sur l'ensemble d'actions et contraintes stockées localement. L'application exécute l'ordonnancement fournit par Telex et présente le résultat à l'utilisateur. Si certaines actions produisent des conflits, alors plusieurs ordres existent, chacun correspondant à une solution possible au conflit. L'application présente les ordres à l'utilisateur de manière qu'il puisse choisir la solution qu'il préfère. Telex échange les actions et les contraintes avec les autres sites en utilisant la structure *multilog*.

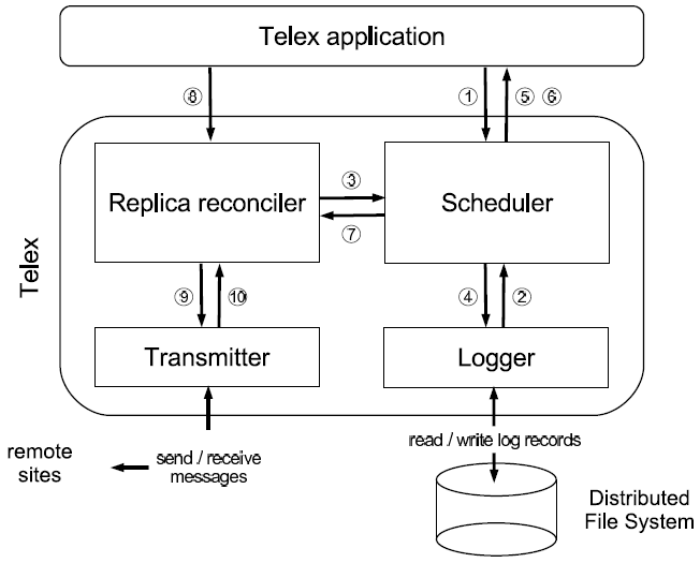

Figure 28. Architecture de Telex

4.3.9 Scalaris

Scalaris [SRHS10] permet la mise en œuvre d'une infrastructure pour les systèmes collaboratifs qui exigent un niveau de cohérence trop élevé (*strong consistency*). Dans Scalaris, le partage et la disponibilité de données est assuré par la réplication. Le maintien de la cohérence entre les différentes répliques est assuré par un gestionnaire de transaction basé sur un algorithme de consensus (appelé *Paxos* [Lamport89]).

L'implémentation de Scalaris est basée sur une DHT. Cependant, il passe difficilement à l'échelle puisqu'il utilise un consensus, qui lui, exige que tous les sites prennent une décision afin de valider (commit) une modification. Autrement dit, si le nombre de réplique augmente, le coût de la communication devient inacceptable.

En outre, d'après *Fisher, Lynch et Patterson*, le consensus n'est pas possible dans un système distribué asynchrone en présence de pannes. Ces difficultés sont contournées par la cohérence optimiste, qui laisse diverger les répliques pour les réconcilier après. Donc, avec une approche de réplication optimiste, on peut assurer un niveau plus faible de cohérence: cohérence à terme (*eventual consistency*). Ce niveau de cohérence, assuré par exemple par les OT, est plus approprié pour les applications collaboratives, notamment les éditeurs de texte collaboratives.

4.3.10 MOT2

MOT2 [CF07] est une solution de réconciliation P2P reposant sur les modèles des transformées opérationnels. Contrairement aux autres solution d'OT [MOSM+03] [LL04a, LL04b], MOT2 ne nécessite ni site central ni vecteur d'horloge pour ordonner totalement les opérations. Dans MOT2, la réconciliation des répliques se fait deux à deux. Lors de chaque synchronisation, un pair envoie tout l'historique (la liste de toutes

les opérations produites sur une réplique des données) à l'autre pair qui peut alors déterminer quelles opérations sont concurrentes. Par conséquent, l'envoi de la totalité des opérations pose des problèmes de charge surtout lorsque l'édition devient massive. D'autre part, cet algorithme suppose l'existence des fonctions de transformation répondant aux contraintes C1 et C2 [EG89, RNRG96]. L'écriture de telles fonctions est un problème réputé difficile et requiert des contraintes fortes telles que la vérification des contraintes C1 et C2.

4.3.11 KTS

Key-based Timestamp Service (KTS) [APV07] a été proposé pour supporter la *currency* des répliques de données, c'est à dire la capacité de retrouver une réplique courante (à jour) malgré la dynamicité du réseau et les mises à jour concurrentes. La validation analytique et expérimentale [APV07] a démontré que KTS conduit à un surcoût très faible en terme de communications.

KTS offre un service distribué pour gérer (créer et mettre à jour) des estampilles dans des systèmes P2P structurés reposant sur une table de hachage distribuée (DHT). En effet, l'opération principale de KTS est de produire des estampilles monotonement croissantes pour les clefs. Les solutions classiques à la génération d'estampilles (par ex. horloges synchronisées) supposent un réseau stable ce qui n'est pas le cas d'une DHT dans un environnement P2P. La solution KTS exploite donc le stockage des données dans les DHT qui est basée sur le fait d'avoir un pair responsable pour stocker chaque donnée et de déterminer le pair en utilisant une fonction de hachage. Dans KTS, pour chaque clef, il y a un pair responsable de l'estampillage qui est choisi dynamiquement en utilisant une fonction de hachage. Cependant, la disponibilité des noeuds stockant les données n'est pas étudiée, c'est ainsi qu'une incohérence peut se produire si un noeud stockant la dernière version d'une donnée quitte le système avant d'avoir propagé sa donnée. En outre, et en produisant des estampilles non continues, la cohérence à terme des répliques après les mises à jour n'est pas garantie en raison des pairs qui quittent le réseau (par exemple dans le cas des pannes) ou les mises à jour concurrentes.

4.3.12 Synthèse générale

Nous évaluons les différentes solutions de réplication dans le réseau P2P, décrites précédemment, selon les critères suivants: type de réseau, type de données, autonomie, cohérence de données et les hypothèses sur le réseau. Le tableau ci-dessous résume nos conclusions :

Tableau 3. Comparaison des solutions de réplication dans le système P2P

	Type de réseau	**Type de données**	**Autonomie**	**Cohérence**	**Hypothèses sur le réseau**
Freenet	Structuré	Fichier	Moyenne	Pas de garantie	Faible
OceanStore	Structuré (DHT)	Quelconque	Forte	Eventuel (efficace pour un nombre restreint des copies)	Forte
Ivy	Structuré	Fichier	Forte	Eventuel (efficace pour un nombre restreint des copies)	Faible
P-Grid	Structuré	Fichier	Forte	Probabiliste	Faible
WOOT	Non-structuré	Structure linéaire	Forte	Oui	Faible
DSR-P2P	Structuré (DHT)	Quelconque	Forte	Oui	Faible
Telex	VOFS	Quelconque	Forte	Oui	Faible
Logoot	Structuré ou non structuré	Structure linéaire	Forte	Oui	Faible
Scalaris	Structuré (DHT)	Quelconque	Forte	Trop élevé (*strong consistency*)	Forte
MOT2	Non structuré	Quelconque	Forte	Oui	Faible
KTS	Structuré (DHT)	Quelconque	Forte	Pas de garantie	Faible

Seules les approches suivantes : WOOT, Logoot, DSR-P2P, Telex et MOT2 ; assurent la *cohérence à terme* des données avec faibles contraintes sur le réseau P2P. Dans le chapitre suivant, nous comparons les approches retenues avec notre solution P2P-LTR.

5 Conclusion

Un des problèmes majeurs posé par la dynamicité du modèle P2P concerne la disponibilité des données. Dans un système P2P, les pairs peuvent joindre ou quitter le

réseau à tout moment et parfois de manière spontanée et donc imprévisible. Cette présence dynamique des pairs se répercute directement sur la disponibilité des données. Pour garantir une bonne disponibilité des données offertes à une communauté, les infrastructures P2P doivent ainsi mettre en place des mécanismes de réplication qui puissent pallier ce problème.

Dans un réseau P2P, chaque objet peut être répliqué en un nombre limité ou illimité de fois. Dans tous les cas, la réplication nécessite de définir et de maintenir la cohérence des copies. Certains systèmes de réplication P2P [KBCC+00] sont basés sur des algorithmes de consensus pour maintenir une forte cohérence de données. Cependant, si le nombre de répliques augmente, le coût de la communication devient inacceptable.

De nombreux algorithmes ont été proposés pour le maintien de la cohérence des données. Certaines approches [SCF98] ne sont pas compatibles avec les contraintes P2P telles que le comportement dynamique des pairs qui composent le réseau P2P. D'autres [WUM07, OUMS+05] s'appuient sur un modèle spécifique des données. Dans ces approches, un objet supprimé est marqué comme invisibles au lieu d'être retiré du modèle du document. Ces objets marqués ne peuvent pas être supprimées sans compromettre la cohérence du document. Par conséquent, le surcoût nécessaire pour gérer le document grossit indéfiniment.

D'autres approches de réplication optimiste [MAPV06, BBMS+08] proposent un algorithme de réconciliation sémantique répartie. Ces algorithmes utilisent le cadre *action-contrainte* pour capter la sémantique de l'application et résoudre des conflits de mise à jour. Ces approches semblent plus adaptées pour la réconciliation des données structurées.

Dans le chapitre suivant, nous proposons une nouvelle approche de réplication et de réconciliation de données partagées. Notre solution doit assurer la cohérence à terme parmi les répliques et tenir en compte le comportement dynamique des réseaux P2P. Cette approche est fondée sur un service de journalisation P2P et un service d'estampillage fiable et répartie fonctionnant sur un modèle de réseau structuré à base de DHT.

CHAPITRE 3

III P2P-LTR: Modèle et Algorithmes

Après avoir présenté un état de l'art sur la réplication optimiste, les systèmes P2P et les approches développées pour assurer la réplication et la réconciliation de données dans un réseau P2P, nous nous focalisons dans ce chapitre sur les fondements de notre approche et plus précisément, nous présentons notre contribution concernant la conception d'une nouvelle solution de réplication optimiste sur un réseau P2P.

Notre travail a été effectué dans le contexte du projet XWiki Concerto [XWIKI07]. L'objectif principal de ce projet est de concevoir une architecture Wiki P2P qui dépasse les limites d'un Wiki client-serveur, et cela afin de supporter un grand nombre d'utilisateurs (passage à l'échelle), de supporter la dynamicité des pairs, d'assurer la disponibilité des données et l'élaboration des documents partagés de façon collaborative et asynchrone. Dans ce contexte, nous avons proposé une nouvelle approche de réplication optimiste, appelée P2P-LTR (en anglais, *P2P Logging and Timestamping for Reconciliation*), adaptée aux besoins d'un Wiki fonctionnant sur un réseau P2P.

P2P-LTR propose un mécanisme d'estampillage fiable, réparti et fonctionnant sur un réseau P2P utilisant une DHT. Il sert non seulement à estampiller les opérations reçues mais aussi à les stocker dans la DHT en garantissant un ordre total continu. Afin de traiter de manière correcte les modifications émises en concurrence, P2P-LTR dispose d'un algorithme d'intégration des modifications concurrentes. L'intégration est réalisée en utilisant un algorithme de réconciliation de données basé sur les transformées opérationnelles (OT). P2P-LTR montre qu'il est possible d'adapter les algorithmes OT afin de réaliser un outil d'édition collaborative à large échelle, tout en supportant le comportement dynamique des pairs.

La suite de ce chapitre est structurée de la manière suivante : la section 1 résume la problématique et les défis de notre travail. Dans la section 2, nous présentons une vue globale de notre modèle P2P-LTR et l'algorithme principal pour la réplication et la réconciliation de données. Dans la section 3, nous proposons une solution complète pour supporter les comportements dynamiques des pairs. Dans la section 4, nous évaluons analytiquement le coût de P2P-LTR en termes de nombre de messages et de nombre d'aller-retours nécessaires pour publier les patches et converger vers un état final. Dans la section 5, nous présentons une étude probabiliste de notre algorithme de réplication. Dans la section 6, nous comparons notre solution par rapport aux approches retenues au chapitre précédent. Enfin, dans la dernière section, nous présentons un résumé de notre contribution.

1 Définition du problème et solution générale

Dans cette section, nous présentons d'abord la problématique de notre travail. Nous présentons par la suite la solution générale et les défis à relever.

1.1 Définition du problème

Le contexte général de notre travail porte sur les applications collaboratives P2P, et plus particulièrement sur l'édition collaborative de texte dans un Wiki P2P. Une application Wiki P2P permet aux utilisateurs de travailler sur les mêmes documents d'une façon collaborative et asynchrone. Pour cela, et afin d'améliorer la disponibilité des données dans les systèmes Wiki P2P qui sont très dynamiques, une application Wiki P2P requière des capacités de réplication avec un mode multi-maître. Les répliques d'une même donnée peuvent être modifiées par plusieurs pairs en parallèle. C'est ainsi qu'en cas d'indisponibilité d'un pair, ses données peuvent être récupérées à partir d'un autre pair détenant la réplique.

Cependant, la *cohérence à terme* des répliques n'est pas forcement garantie après leurs mises à jour. En effet, les modifications concurrentes effectuées sur les différentes répliques peuvent créer des divergences de copies. D'autre part, quand certains pairs rejoignent le réseau, après l'avoir quitté, se pose le problème de la mise à jour de leur réplique. Cette non garantie de la cohérence à terme des données provoque par conséquent un état non cohérent du système.

Par ailleurs, nous avons présenté au chapitre précédent le modèle OT comme étant la solution la plus répandue pour mettre en place un mécanisme de réplication optimiste dans un contexte distribué. Nous avons aussi vu que pour assurer la cohérence à terme des données, les opérations de mise-à-jour doivent être estampillées selon un ordre total.

La réalisation d'un ordre total restait problématique dans un système distribué. En effet, pour réaliser cet ordre, certaines solutions, basées sur OT utilisent un serveur central d'estampille [MOSM+03, OMSI04], notamment la solution So6 (voir chapitre 2, section 3.1.5). L'utilisation d'un estampilleur central limite le passage à l'échelle et peut bloquer le système en cas de panne du serveur. En outre, la taille du journal des opérations utilisé par OT au niveau du serveur d'estampilles a tendance à être important et peut être non maîtrisable dans un seul nœud. D'autres solutions décentralisées qui s'appuient sur des vecteurs d'horloges [LL04a, LL04b] ont été proposées pour le modèle OT. Cependant, ces solutions ne sont pas adaptées aux caractéristiques des réseaux P2P. MOT2 [CF07] est la seule solution dans OT qui ne nécessite ni estampilleur central ni vecteur d'horloge afin d'assurer un ordre total sur les opérations. Cependant, lors de chaque synchronisation, il est nécessaire de transmettre toutes les opérations, et pas seulement les opérations manquantes. Par conséquent, la propagation des opérations entre les pairs devient lente et coûteuse.

1.2 Solution générale

Dans le reste de ce chapitre, nous proposons une solution originale aux problèmes rencontrés dans les solutions précédentes, tout en donnant une nouvelle technique de propagation et d'intégration des mises à jour de données dans un réseau P2P. Notre solution consiste à fournir une disponibilité de données en les répliquant sur plusieurs pairs du réseau. En outre, elle adresse efficacement le problème de cohérence à terme des répliques par l'utilisation des transformées opérationnelles (OT). Le principe ressemble à celui de So6 (voir chapitre 2, section 3.1.5). Cependant, à la différence de So6 qui utilise un estampilleur central pour maintenir et ordonner totalement les opérations, notre solution distribue ce rôle dans le système P2P. En effet, notre solution construit un mécanisme d'estampillage fiable, réparti et fonctionnant sur un réseau P2P utilisant une DHT. Ce mécanisme sert à estampiller les opérations puis à les stocker dans le réseau selon un *ordre total continu*. L'autre différence entre notre solution et So6 réside dans la gestion des pannes et des comportements dynamiques des pairs.

Définition 3.1 ***(Ordre total continu).*** *Pour la gestion de la réconciliation, notre solution estampille les opérations selon un ordre total continu. En effet, pour deux estampilles quelconques ts_1 et ts_2, produite pour un document respectivement aux temps t_1 et t_2, si $t_1 < t_2$ alors nous avons $ts_1 < ts_2$ et pour deux valeurs consécutives d'estampilles ts_1 et ts_2 nous avons toujours $ts_2 = ts_1+1$. Si $ts_i < ts_j$ alors nous pouvons dire que les opérations de mise à jour estampillées par ts_i précèdent celles estampillées par ts_j.*

Notre solution gère les opérations de mises à jour sous forme de *patches*. Un patch est l'unité de réconciliation de notre système. Il correspond à une séquence d'opérations (*insertion*, *suppression*, *modification*) ordonnées chronologiquement. Une fois ces opérations ordonnées selon un ordre total continu, elles sont utilisées pour la réconciliation grâce à un algorithme d'intégration d'OT [VCFS00]. Cet algorithme est basé sur des fonctions de transformations proposées dans [MOSM+03] pour la réconciliation de documents textuels. La solution proposée doit relever les défis suivants :

- **Décentralisation** : le mécanisme de réplication doit fonctionner de manière décentralisée. En particulier, il ne doit pas utiliser de serveur centralisé pour générer les estampilles, stocker ou diffuser les patches dans le réseau.

- **Passage à l'échelle, dynamicité, tolérance aux pannes** : dans un système collaboratif P2P, le nombre de pairs introduisant des changements est potentiellement grand, et les pairs sont relativement dynamiques. Le mécanisme de réplication proposé doit donc être capable de gérer ce passage à l'échelle (afin de supporter une réplication multi-maître massive) et de supporter les pannes et le comportement dynamique des pairs.

- **Disponibilité des patches** : la solution proposée doit fournir une journalisation fiable et distribuée des mises à jour (patches). Ces mises à jour sont utilisées ensuite pour la réconciliation.

- **Cohérence à terme** : lorsque le système est au repos, c'est-à-dire lorsque plus aucune modification n'est apportée, toutes les répliques doivent convergées vers un état identique en un temps fini. Notre solution doit assurer la cohérence à terme des répliques malgré la dynamicité et les cas de pannes.

- **Vivacité (liveness)** : dans notre cas, la propriété de vivacité est exprimée par la capacité du système à continuer à fonctionner correctement (*i.e.* la continuité des estampilles est assurée) malgré la dynamicité et les cas de pannes. Cette propriété doit assurer la publication à terme des patches en cas de pannes (*i.e.* dans certains cas l'utilisateur est obligé de recommencer sa demande de publication des patches, notamment dans le cas des mises à jour concurrentes).

2 Modèle du P2P-LTR

Dans cette section, nous donnons d'abord une vue d'ensemble de notre modèle, puis nous décrivons brièvement les opérations de base nécessaires à l'implémentation de P2P-LTR. Ensuite, nous présentons notre algorithme utilisé pour la validation des patches. Puis, nous décrivons la manière dont P2P-LTR assure la cohérence à terme de différentes répliques. Enfin, nous présentons un exemple de fonctionnement de P2P-LTR.

2.1 Description

Pour fournir un support distribué, dynamique et fiable à l'implantation d'un mécanisme de réplication optimiste, nous proposons le modèle illustré par la Figure 29. Ce modèle est composé de quatre nœuds dont chacun représente un rôle bien défini : le *User Peer*, le *Master-key Peer*, le *Master-key-Succ* et le *Log-Peer*. Notre modèle est facile à contrôler étant donné la répartition des rôles et des fonctionnalités entre les différents nœuds du système. De plus, pour faire face aux problèmes de surcharge dans le cas des documents lourdement mis à jour, cette répartition permettra de partager la charge entre les différents nœuds du système au lieu de surcharger un seul nœud responsable de toutes les mises à jour.

User Peer. Ce nœud présente l'application cliente (dénoté par *u*). Chaque application dispose d'une copie primaire d'objet partagé (dans notre cas, un document). Une fois qu'un patch est généré sur un document *D*, il est capturé par *u* et estampillé ensuite selon un ordre continu. Pour ce faire, les utilisateurs doivent contacter le nœud responsable de la production d'estampille pour *D*, appelé le *Master-key* (voir label 1,

Figure 29). En prenant en compte les mises à jour simultanées sur *D*, un nœud utilisateur a besoin de récupérer tous les patches qui ont déjà été publiés et qui sont disponibles sur la DHT (voir label 2', Figure 29).

Master-key Peer. Dans P2P-LTR, pour chaque document identifié par une clé *key*, nous avons un nœud responsable de l'estampillage choisi dynamiquement en utilisant une fonction de hachage spécifique (dénoté par h_t). Le Master-key Peer permet la gestion distribuée des estampilles de différentes clés et la réplication des patches estampillés dans la DHT (voir label 3,
Figure 29). Le Master-key doit disposer au préalable d'un ensemble de fonctions de hachage, dénoté par $H_r = \{h_1, h_2..., h_n\}$. H_r présente l'ensemble de fonctions de réplication (en anglais, *replication hash functions*). Ces fonctions sont utilisées pour la réplication des patches dans la DHT.

Master-key-Succ. Ce nœud remplace le Master-key en cas de panne et prend la responsabilité des répliques pour la dernière valeur d'estampille générée *last_ts*.

Log-Peers. Ce sont les nœuds responsables du stockage des patches estampillés des différents documents.

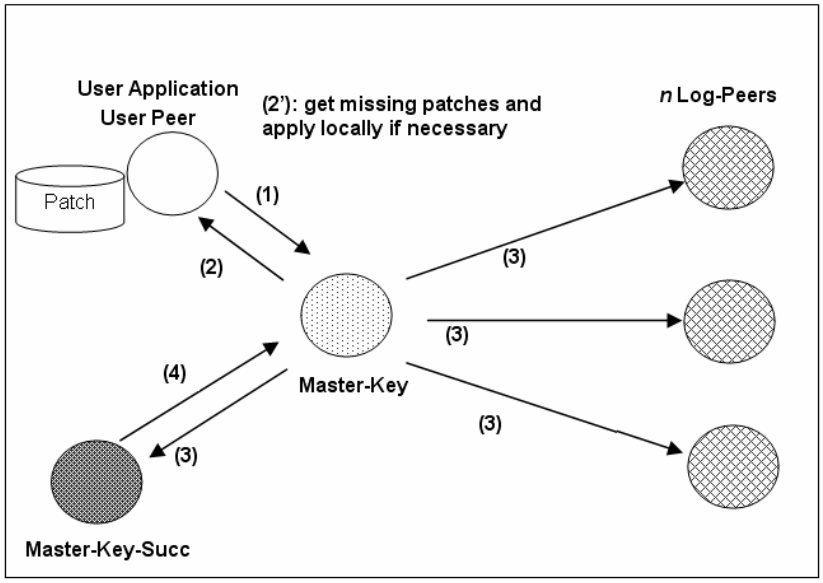

Figure 29. Le modèle de P2P-LTR

Le tableau suivant montre un récapitulatif de la répartition des rôles entre les différents nœuds du modèle.

Tableau 4. Description et rôle des nœuds de modèle P2P-LTR

Nœud	Rôle
User Peer	- Mise à jour d'un document. - Publication d'un patch. - Récupération d'un patch publié sur la DHT.
Master-key Peer	- Estampillage des patches reçus en garantissant. un ordre total continu sur les patches d'un même document. - Réplication des patches estampillés sur la DHT.
Master-key-Succ	- Remplacement de Master-key en cas de panne.
Log-Peer	- Stockage des patches estampillés des différents documents.

Notre modèle de réseau est *semi-synchrone*, une combinaison entre le modèle synchrone et asynchrone, similaire à celles proposées dans [APV07] [MP06].

2.2 Opérations

Après avoir donné un aperçu global du modèle P2P-LTR, nous présentons dans cette partie les principales opérations nécessaires à l'implémentation de P2P-LTR. Ces opérations sont utilisées par les différents nœuds de la DHT.

Pour la publication des patches, un utilisateur doit disposer des opérations suivantes :

- La méthode ***publishPatch(key, patch, ts)*** : permet à un utilisateur de publier un patch et son estampille *ts* dans la DHT comme suit : $Put(h_{ts}(key), (Patch+ts))$, avec h_{ts} est la fonction de hachage utilisée pour localiser le Master-key Peer (voir label 1, Figure 30).
- La méthode ***get(h_i(key,ts))*** : permet à un utilisateur de récupérer, selon un ordre continu, les patches manquants (les opérations qui sont déjà publiées) de la DHT à partir de sa clé *key* et son estampille *ts*, avec $h_i \in H_r$ qui est l'une des fonctions de hachage utilisées pour la réplication (voir label 2', Figure 30).

Pour la gestion des patches, un Master-key doit disposer des opérations suivantes :

- La méthode ***gen-ts(key)*** : c'est l'opération principale de Master-key qui, en prenant une clé *key*, produit un nombre entier correspondant à l'estampillage de *key*. Les estampilles produites ont les propriétés suivantes : la *monotonicité* et la *continuité*, *i.e.* deux estampilles produites pour la même clé sont monotonement croissantes et la différence entre deux estampilles de deux mises à jour consécutives est égale à un.

- La méthode ***last-ts(key)*** : permet à un *Master-key* de récupérer la dernière estampille générée pour la clé *key*. La méthode *last-ts* peut être implémentée tout comme la méthode *gen-ts* sauf que *last-ts* est plus simple car elle retourne la valeur d'estampille sans avoir besoin d'incrémenter sa valeur.
- La méthode ***sendToPublish(key, last_ts, patch)*** : permet de répliquer un patch dans la DHT comme suit : $Put(h_1(key,ts),Patch)$, $Put(h_2(key,ts),Patch)$,… $Put(h_n(key,ts),Patch)$, avec $h_i \in H_r$. De plus, cette méthode permet de répliquer la dernière estampille délivrée par le Master-key (*last_ts)* dans le pair Master-key-Succ (voir label 3, Figure 30).

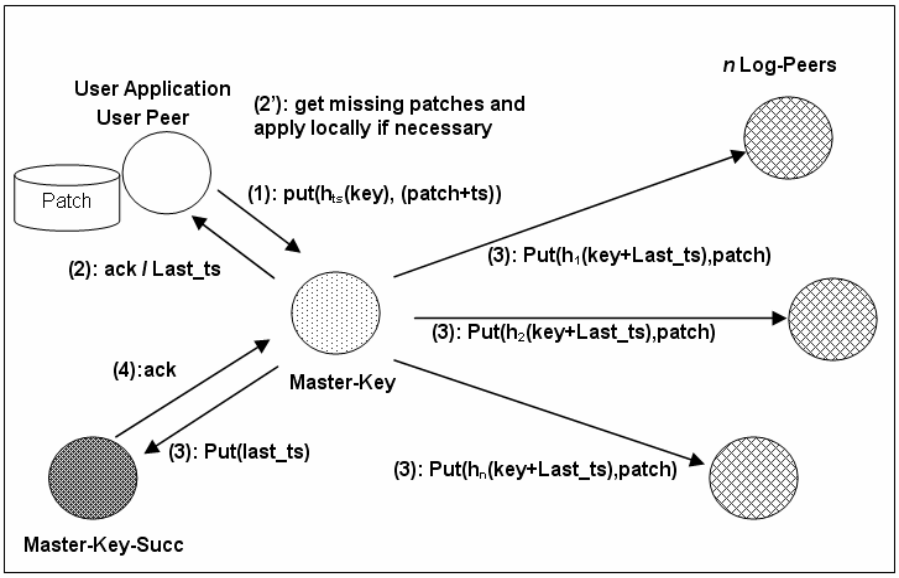

Figure 30. P2P-LTR: les opérations

2.3 Algorithmes

Nous décrivons dans cette partie le principe de fonctionnement de notre algorithme de réplication proposé pour garantir la disponibilité de données dans les systèmes P2P. Notre algorithme traite également la mise à jour des données répliquées et la récupération efficace des mises-à-jour de système P2P afin d'assurer la cohérence à terme des différentes répliques.

Dans notre architecture, chaque site disposant d'une copie primaire des données partagées génère des opérations et les intègre localement avant de les publier dans la DHT sous forme de patch. Pour envoyer un patch, il faut d'abord recevoir tous les patches estampillés et les intégrer localement. Il faut ensuite estampiller les opérations locales et de les envoyer. Le fonctionnement principal de l'algorithme est basé sur les

Master-key Peers qui délivrent de façon distribuée les estampilles à chaque patch. Les patches sont donc estampillés selon un ordre total continu.

Supposons qu'une application cliente s'exécute sur chaque pair. Chaque utilisateur *u* possède localement une copie primaire d'un document *D*. Une fois le document modifié, l'application cliente produit des patches. Les applications clientes ne s'échangent pas directement les patches produits. En effet, elles doivent les faire transiter par les estampilleurs (*Master-key Peers*) qui, à leur tour, les font valider puis stocker dans les *Log-Peers* en utilisant un ensemble de fonctions de hachage, noté par $H_r = \{h_1, h_2..., h_n\}$. Ces *Log-Peers* maintiennent, de manière persistante, les patches publiés par les différents sites selon l'ordre des estampilles.

```
Algorithm 1: User-Peer.publishPatch_ht(key, patch, ts)


1: begin
2: //Tentative de publication de patch dans la DHT
3: Master-key-Peer.sendToPublish(key, patch, ts);
4: //Attendre un message d'Ack de Master-key-Peer
5: If receive (ack, last_ts) then
6:    Notify User-Peer of successful publish patch in DHT;
7: // Attendre une période de temps δ avant la synchronisation:
8: // vérifie périodiquement l'existence des patches manquants dans la
9: // DHT
10:   Synchronization();
11: Else receive(fail, last_ts)
12: //Récupérer les patches manquants dans la DHT;
13:   // h_i ∈ H_r= {h_1, h_2…, h_n}
14:   For each h_i ∈ H_r do
15:         Patches = get(h_i(key,last_ts));
16:   End do,
17: //Transformer et appliquer localement ses patches en utilisant OT
18:   Apply Patches;
19: //Envoyer les patches validés dans la DHT
20:   Send patches;
21: End if;
22: End;
```

Figure 31. P2P-LTR: Algorithme de réplication de patches

Afin de réaliser la validation des patches, chaque document a une valeur locale d'estampille associée (dénotée par *ts*). Le *Master-key* tient la dernière estampille (dénotée par *last_ts*) produite pour le même document. Ainsi, pour un document donné *D*, l'utilisateur u_1 contacte le *Master-key* correspondant, auquel il envoie sa demande de publication de patch avec la valeur *ts* (Lignes 2-4, Algorithme 1). Le *Master-key* est

choisi dynamiquement en utilisant une fonction de hachage, dénotée par h_t. Supposons que *key* est une clé, alors le responsable d'estampillage pour *key* est le pair qui est responsable de *key* par rapport à h_t *(resp (key,* h_t *))*. L'utilisateur u_1, qui a besoin d'une estampille valide pour la clé *key*, utilise le service de *lookup* de la DHT pour obtenir l'adresse de *Master-key (i.e. resp (key,* h_t *))* auquel il envoie une demande de publication de patch.

Quand le *Master-key* reçoit la demande de u_1, il compare d'abord les deux valeurs d'estampilles *last_ts* et *ts*. Si la valeur locale d'estampille (*last_ts)* de *Master-key* est égale à *ts*, alors le *Master-key* incrémente la valeur de *last_ts* et confirme à l'utilisateur u_1 le déclenchement de la procédure de réplication du patch. Ensuite, le *Master-key* réplique les patches estampillés dans les *Log-Peers* et la dernière valeur d'estampille dans son successeur. Enfin, le Master-key envoie un message de confirmation (*Ack*) à u_1 contenant la dernière valeur d'estampille (Lignes 2-9, Algorithme 2). Afin d'éviter une grande différence entre les répliques sur lesquelles les utilisateurs travaillent, les utilisateurs vérifient périodiquement s'il y a de nouveaux patches publiés par d'autres utilisateurs, par exemple à chaque unité de temps δ. Si des patches existent, ils doivent être récupérés et intégrés localement (Lignes 7-10, Algorithme 1).

Maintenant, si la valeur locale d'estampille *(last_ts)* de Master-key est supérieure à *ts*, cela signifie qu'il existe des patches validés dans le système, qui sont générés par d'autres utilisateurs et doivent être intégrés dans le document *D* de l'utilisateur u_1. Pour ce faire, u_1 doit effectuer la récupération de tous les patches manquants dans un ordre continu jusqu'à ce que la valeur de *last_ts* soit égale à la valeur de *ts* (Lignes 11-21, Algorithme 1).

```
Algorithm 2: Master-key-Peer.sendToPublish(key, patch, ts)

1: when receive(patch, ts)
2:    If ts == last_ts then
3:          Last_ts = last_ts + 1;
4:          For each h_i ∈ H_r do in parallel
5:                put(h_i(key, last_ts), patch);
6:          //Réplication d'estampille dans le successeur de Master
7:          Replicate current-ts at Master-key-Succ;
8:          //Envoyer un message de notification à l'utilisateur
9:          Send Ack message, contain last_ts to User-Peer;
10:   Else
11:         // Indiquer la récupération des patches manquants
12:         Send fail message, contain last_ts to User-Peer;
13:   End if;
14: End;
```

Figure 32. P2P-LTR: Algorithme de réplication de patches

Pour gérer la validation simultanée des patches estampillés sur le même document, le *Master-key* sert chaque utilisateur de façon séquentielle. Par conséquent, une nouvelle valeur d'estampille pour un même document *D* est reportée jusqu'à la réplication du précédent patch de document *D* estampillé par la valeur *ts-1*.

Propriété 1 ***(Ordre total continu).*** *P2P-LTR garantit que les patches générés sur les différentes répliques sont estampillés et stockés dans la DHT selon un ordre total continu.*

2.4 Cohérence à terme

La réplication multi-maître offre plusieurs avantages en termes de performance et de disponibilité de données. Cependant, les mises à jour de la même donnée par différents pairs peuvent créer des divergences de copies et donc un état instable du système de réplication. Le défi est donc d'assurer la cohérence à terme des données en raison de mises à jour concurrentes.

Dans la réplication optimiste, on dit qu'un système converge lorsque toutes les opérations sont propagées sur tous les sites et que les répliques sont identiques. On dit alors que le système est dans un état cohérent.

Dans la suite de cette partie, nous présentons la manière dont P2P-LTR garantit la cohérence à terme *(CT)* des répliques. Pour cela, nous montrons que si tous les utilisateurs arrêtent de publier des patches (par exemple après un temps *t*), toutes les répliques convergent vers le même état. Par conséquent, tout nouvel utilisateur qui se connecte au système verra le même état du document reflétant toutes les modifications publiées avant *t*.

Supposons que nous avons *k* utilisateurs disposant d'une copie (*réplique*) d'un même objet partagé (ex. un document *D)*, et peuvent le modifier librement et à tout moment. Dans notre système, la collaboration ne doit pas être contrainte par une topologie rigide. Par conséquent, les participants à l'édition collaborative du document peuvent se connecter ou se déconnecter à tout moment du système. En mode déconnecté, un utilisateur peut continuer à introduire des modifications dans le document. Dans ce cas, une fois que l'utilisateur rejoint le système, il peut publier ses modifications et récupérer les modifications des autres utilisateurs.

Soit u_1 un utilisateur qui vient de faire une modification sur le document *D* à temps *t*. Nous montrons que s'il n'y a pas de nouvelle mise à jour alors toutes les répliques convergentes vers le même état que celui de u_1. Considérons maintenant un utilisateur u_2 qui maintient une réplique du document *D*. u_2 peut avoir l'une des situations suivantes : 1) après le temps *t*, il s'est déconnecté au moins une fois du système, 2) après le temps *t*, il est toujours connecté au système. Pour chacun de ces cas, nous montrons que le document de u_2 converge finalement vers celui de u_1.

Dans le premier cas, lorsque u_2 se reconnecte au système, il appelle l'opération *last-ts(key)* de Master-key. Ensuite, il compare la dernière estampille *last_ts* générée par le Master-key avec son estampille locale *ts*. Si $last_ts \neq ts$, u_2 récupère d'abord tous les

patches manquants dans un ordre continu, il les intègre ensuite localement en utilisant les transformées opérationnelles. Ainsi, l'état de son document devient égal à celui de u_1.

Pour le deuxième cas (*i.e.* celui où u_2 est toujours connecté au système), sachant que chaque utilisateur vérifie périodiquement l'existence de patches manquants dans la DHT, autrement dit, à chaque unité de temps δ, si des patches existent, ils sont récupérés et intégrés localement au plus tard après le temps $t' \leq \delta + t$. Ainsi le document de u_2 converge vers le même état que celui du document de u_1.

Propriété 2 ***(récupération des patches manquants).*** *Tous les nœuds, qu'ils soient connectés ou temporairement déconnectés du système, finissent par appliquer localement tous les patches publiés dans la DHT.*

Dans ce qui suit, nous donnons la spécification formelle de la satisfaction de la propriété de *CT* par notre système P2P-LTR. Pour cela, considérons d'abord les notations suivantes :

Notation	Description
U	Représente l'ensemble des utilisateurs travaillant sur le même document D et qui ne quittent pas le système définitivement.
α, β	Représentent les deux patches générés respectivement par les utilisateurs u et v.
$E_{(u,D)}(t)$	Indique l'état du document D, sur la machine de l'utilisateur u, à l'instant t.
$Stop(u,D,t)$	Indique qu'avant le moment t, l'utilisateur u a arrêté définitivement de modifier le document D.

Nous pouvons formuler la satisfaction de la propriété de *CT* par notre système comme suit :

$$Si\ (\exists t) \wedge (\forall u \in U,\ stop(u,D,t)) \Rightarrow \exists t'' \geq t\ \forall u,v \in U\ E_{(u,D)}(t'') \equiv E_{(v,D)}(t'')\ (3.1)$$

Lemme 1 ***(Cohérence à terme).*** *Supposons que tous les nœuds gardant des répliques sont connectés ou temporairement déconnectés du système. Lorsque le système est au repos, c'est-à-dire lorsque plus aucune modification n'est introduite dans le réseau, P2P-LTR assure la cohérence à terme des répliques des mêmes données.*

Preuve. *Induite par la discussion ci-dessus.*

2.5 Exemple

Nous illustrons maintenant le fonctionnement de notre protocole sur un simple exemple (voir Tableau 5). Nous considérons un ensemble de pairs interconnectés par un réseau DHT selon le modèle P2P-LTR décrit dans la Figure 30. Les pairs p_1 et p_2

possèdent une copie d'un objet partagé, par exemple un document textuel *D*. Le pair p_3 est le Master-key du document *D*.

Au début, chacun des pairs p_1 et p_2 possède le même état initial d'objet, *i.e.* le même contenu du document *D*. Supposons que *t*, *t'* et *ts* sont respectivement les estampilles locales de p_1, p_2 et p_3. Initialement, nous avons $t=0$, $t'=0$ et $ts=0$. Suite à une modification du document *D*, le pair p_1 produit le patch p_{11} contenant deux opérations de mise à jour op_1 et op_2. De même, le pair p_2 produit le patch p_{21} contenant les deux opérations concurrentes op_3 et op_4.

1. Au point (1) : le pair p_1 essaie de publier le patch p_{11}. Pour cela, p_1 contacte p_3, auquel il envoie sa demande de publication de patch estampillé avec la valeur $t = 0$. Quand p_3 reçoit la demande de publication de p_1, il compare les deux valeurs d'estampille *t* et *ts*. Les deux valeurs d'estampilles sont égales, alors p_3 incrémente d'abord la valeur de l'estampille *ts* par un. Ensuite, p_3 réplique le patch p_{11} dans les Log-Peers et la dernière valeur d'estampille dans son successeur. Enfin, p_3 confirme à l'utilisateur p_1 la réussite de la procédure de réplication de patch en envoyant un message de confirmation (Ack) à p_1 contenant la dernière valeur d'estampille ($ts=1$). À la réception du message d'*Ack*, p_1 incrémente sa valeur locale d'estampille par 1. Par conséquent, nous avons les valeurs suivantes de différentes estampilles: $t = 1$, $t'=0$ et $ts=1$.

2. Au point (2) : le pair p_2 tente de publier son patch p_{21}. Pour cela, il contacte le pair p_3. Le système détecte qu'il y a des patches concurrents dans la DHT : la valeur d'estampille de p_3 est supérieure à p_2. Par conséquent, p_3 envoie un message à p_2 contenant la valeur d'estampille *ts*, afin d'indiquer la récupération des patches manquants, *i.e.* p_{11}. p_2 doit donc récupérer d'abord le patch généré par l'utilisateur p_1. Ensuite, il l'intègre localement en utilisant OT. Après avoir intégré le patch p_{11}, p_2 essaie de publier de nouveau son patch estampillé avec la valeur $t' = 1$. Nous avons $t' = ts = 1$, donc p_3 incrémente la valeur de l'estampille *ts* par un et déclenche la procédure de réplication de patch (de même que (1)). Enfin, nous avons les valeurs suivantes de différentes estampilles : $t = 1$, $t'=2$, $ts=2$.

3. Au point (3) : Après une période de temps δ, p_1 essaie de synchroniser avec le système, *i.e.* vérifier si des patches existent dans la DHT afin de les récupérer et les intégrer localement. Par conséquent, p_1 récupère le patch p_{21} et nous avons donc les valeurs suivantes des estampilles : $t = 2$, $t'=2$, $ts=2$.

Tableau 5. Scénario de collaboration de deux pairs
(*)P_i : les Log-Peers responsables du stockage des patches estampillés.

	p_1	p_2	p_3	P_i(*)
(0)	$t = 0$ $p_{11}(op_1, op_2)$	$t' = 0$ $p_{21}(op_3, op_4)$	$t_s = 0$	
(1)	$t = 1$	$t' = 0$	$t_s = 1$	$(p_{11},1)$
(2)	$t = 1$	$t' = 2$ *synchronize*	$t_s = 2$	$(p_{11},1)$ $(p_{21},2)$
(3)	$t = 2$ *synchronize*	$t' = 2$	$t_s = 2$	$(p_{11},1)$ $(p_{21},2)$

3 Gestion du comportement dynamique des pairs

L'une des caractéristiques les plus importantes des réseaux P2P est le comportement dynamique des pairs : c'est-à-dire que les pairs peuvent se connecter ou se déconnecter du système à tout moment. Dans cette partie, nous nous intéressons à la prise en compte de ce comportement dans la modélisation de P2P-LTR. D'abord, nous considérons le cas où un nouveau *Master-key Peer* rejoint le système ou le quitte normalement, c'est-à-dire qu'il ne tombe pas en panne, et nous proposons des stratégies avec lesquelles P2P-LTR peut faire face à ces cas. Ensuite, nous abordons les situations où le Master-key ou son successeur quitte le système sans notification (par exemple en cas de panne). Enfin, nous décrivons la manière dont P2P-LTR assure la propriété de vivacité malgré les pannes.

Quand un nœud tombe en panne, ses traitements s'arrêtent anormalement, ce qui peut conduire à des incohérences. Dans notre travail, nous supposons qu'un nœud fonctionne correctement ou s'arrête complètement (*i.e.* il est en panne). En d'autres termes, nous ne prenons en compte que les pannes *franches* (en anglais, *Crash failures)* [XAP04] et non pas les pannes *byzantines*.

Définition 3.2 ***(panne franche).*** *Le système s'arrête prématurément. Cela signifie qu'il cesse d'exercer toute activité y compris l'envoi, la transmission, ou la réception de messages.*

Définition 3.3 ***(panne byzantine).*** *Le comportement du système est arbitraire vis-à-vis de la panne.*

3.1 Arrivée et départ du Master-key

Le Master-key d'une clé *k* peut changer dynamiquement suite à l'arrivée d'un nœud ou à son départ du système. P2P-LTR est une solution générique à toute DHT. Pour des raisons de simplicité et afin de maintenir la connectivité entre les différents nœuds du système, P2P-LTR utilise la topologie en anneau de la DHT Chord [SMKK+01]. Chord réalise la tolérance aux pannes et maintient la connectivité des pairs, en stockant une

liste de successeurs pour chaque pair. Ainsi, même si le successeur d'un pair *p* quitte le système, *p* peut utiliser sa liste de successeurs pour identifier un autre pair et se reconnecter sur l'anneau.

Pour maintenir à jour la liste de successeurs, Chord fournit une fonction de vérification appelée *stabilize* [SMKK+01]. Cette fonction permet à un nœud et à son successeur de vérifier qu'ils forment un couple correct, c.-à-.d. qu'il n'y a pas de nouveaux nœuds entre les deux. Donc, la fonction *stabilize* assure correctement la mise à jour des entrées de la table de routage de chaque pair afin qu'il puisse suivre les changements survenus dans le réseau. Grâce à cette fonction de maintenance des tables de routage, Chord garantit l'exactitude de sa fonction de recherche *lookup* et offre une forte disponibilité du système.

Dans la suite de cette partie, nous supposons l'existence d'une transmission fiable de messages échangés entre les différents nœuds du système.

3.1.1 Arrivée d'un nouveau Master-key

Dans ce scénario, nous nous concentrons sur le cas où un nouveau pair rejoint le système et devient Master-key pour certaines clés.

L'insertion d'un nouveau nœud dans le réseau se fait à travers la fonction *join* du protocole Chord. Cette fonction renvoie au nouveau Master-key son successeur dans le réseau. Cependant, l'insertion de ce pair provoque un défaut de successeur au niveau du nœud qui le précède dans l'anneau. Il est donc nécessaire de l'avertir de ce changement. Donc, le déclenchement de la fonction *stabilize* a pour but de garantir que le prédécesseur du nœud successeur sur lequel le nœud pointe est bien lui-même. Une fois le successeur du nœud mis à jour, il est nécessaire d'avertir le nouveau Master-key qu'il a un nouveau prédécesseur. Par conséquent, la fonction *stabilize* garantie qu'un nœud donné pointe toujours sur un successeur valide et la configuration globale du réseau n'est jamais brisée suite à un événement de *join*.

Afin d'être informé des modifications faites dans la topologie du réseau DHT, P2P-LTR doit nécessairement s'abonner à l'événement de stabilisation de la DHT. Par conséquent, le modèle de communication que nous adoptons entre la couche P2P-LTR et la couche DHT, ressemble au modèle *publish/subscribe*. Dans notre modèle, nous considérons la DHT comme un éditeur d'évènement et la couche P2P-LTR comme abonnée à ces évènements. L'abonnement de P2P-LTR aux évènements de la DHT se fait à travers un écouteur d'évènements (en anglais, *Event Listeners*). Les écouteurs (*EL*) permettent à la couche P2P-LTR d'être informée de l'évènement et de réagir en conséquence. Donc, notre système peut s'inscrire comme un écouteur pour l'événement de stabilisation de la DHT. Par conséquent, lorsque la procédure *stabilize* est invoquée, l'écouteur d'événements appelle la fonction de la couche P2P-LTR responsable de la détermination du nouveau Master-key et son successeur. Ainsi, notre système peut surveiller les évènements de *join* produits sur la couche DHT et donc maintenir automatiquement la liste des clés et estampilles sur la couche de P2P-LTR.

Dans le but de notifier l'écouteur LTR par les événements de la DHT, la couche DHT passe simplement l'événement spécifique à l'écouteur LTR. Un événement doit inclure toutes les informations nécessaires pour qu'un écouteur LTR puisse comprendre ce qui s'est passé exactement à la couche DHT. De cette façon, la couche P2P-LTR peut réagir d'une manière significative.

Une fois la stabilisation de la DHT terminée, P2P-LTR assure que l'ancien Master-key transfère la liste de ses estampilles au prochain responsable. Supposons que q est le responsable actuel d'estampillage, K est l'ensemble de clés pour lequel q est le Master-key et p est un nouveau pair qui rejoint le système et devient Master-key pour certaines clés $K' \subseteq K$. À la fin de sa responsabilité de certaines clés dans K' (c'est à dire quand p rejoint le système), q envoie tout d'abord à p toutes les estampilles qui ont été générées pour les clés impliquées dans K' et modifie alors son rôle envers ces clés de Master-key à Master-key-Succ.

Soit L un ensemble vide de paires *(clé, estampille)* et $Role_{a.k}$ le rôle du pair q vers la clé k. Notons par t_k la valeur de la dernière estampille produite pour une clé k, q exécute la fonction *LoseResponsability(K', p)* à la fin de sa responsabilité (voir Figure 33).

```
Algorithm 3: loseResponsability(K', p)

1: begin
2:    For each k ∈ K do
3:          if (k ∈ K') then
4:                L.add(k, t_k)
5:          //Modifier le rôle envers la clé k
6:                Role_{q,k} = Master-key-Succ;
7:          End if;
8:    End do;
9: //Envoyer à p toutes les estampilles générées pour les clés k ∈ K'
10:   send L to p;
11: End;
```

Figure 33. Pseudo code de la fonction chargée du transfert des clés: cas de join

Avant de recevoir les estampilles de q, le pair p ne génère pas d'estampille pour les clés impliquées dans K'. Donc, P2P-LTR ne produit des estampilles que si la DHT est stabilisée. Par conséquent, toute demande de publication des patches reçus au moment de la stabilisation est rejetée, tout en envoyant un message au demandeur de requête indiquant que le système exécute la procédure de stabilisation.

Avec un exemple, nous illustrons le cas où un responsable d'estampillage perd sa responsabilité suite à un *join* avec le scénario suivant. Nous considérons la configuration du système illustré à la Figure 34.

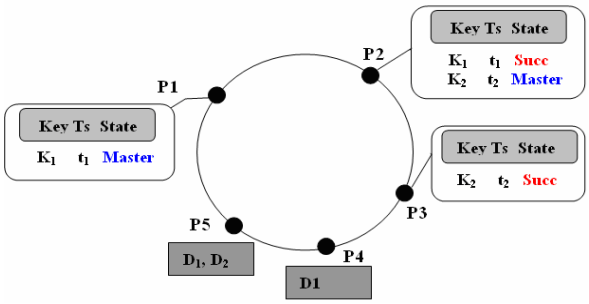

Figure 34. P2P-LTR: configuration initiale

Nous représentons le réseau sous la forme d'un anneau. Chaque nœud est marqué par la lettre *P* (pour pair) suivie de son identifiant. La liste des clés et estampilles stockées sur les différents nœuds est représentée par des cadres à l'intérieur desquels nous trouvons le rôle de chaque nœud par rapport à sa liste des clés, *i.e.* Master-key ou Master-key-Succ. Par exemple, ici, nous pouvons voir que le pair P_1 est le Master-key pour la clé k_1, dont la dernière valeur d'estampille produite est égale à t_1.

Supposons que P_0 est un nouveau pair qui rejoint le système et devient le nouveau Master-key pour la clé k_2. Le successeur de P_0 dans l'anneau, ici le pair P_2, lui transmet automatiquement la clé k_2, dont il aura la charge. La Figure 35 montre la nouvelle configuration du système après l'opération de *join*, où P_2 transfère (k_2, t_2) à P_0. En conséquence, la nouvelle configuration des pairs P_0 et P_2 devient comme suit:

- P_0 Master-key de la clé k_2 et Master-key-Succ de la clé k_1
- P_2 Master-key-Succ de la clé k_1

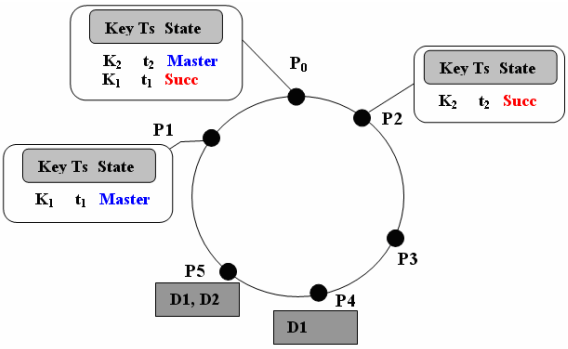

Figure 35. Configuration après le join de P0

3.1.2 Départ du Master-key

Quand un Master-key quitte le système normalement, il transfère ses clés et estampilles à son successeur, *i.e.* au *Master-key-Succ*. Comme pour l'évènement *join*, le départ d'un nœud déclenche la stabilisation de la DHT. En utilisant les écouteurs d'événements,

P2P-LTR détecte l'événement de départ. Une fois que la stabilisation est terminée, l'ensemble des clés et estampilles du précédent Master-key sont envoyés à son successeur et à un autre pair qui deviennent respectivement le nouveau Master-key et le Master-key-Succ pour la liste des clés. Soit *q* et *p* deux pairs et *K'* l'ensemble d'estampilles pour laquelle *q* est l'actuel Master-key et *p* le Master-key-Succ, c'est-à-dire le nouveau responsable des estampilles. Une fois *q* atteint la fin de sa responsabilité pour les clés de *K'*, c'est à dire une fois qu'il a quitté le système, il envoie à *p* toutes les estampilles qui ont été générées pour les clés impliquées dans *K'*. Par conséquent, *p* modifie son rôle vers les clés du Master-key-Succ au Master-key et envoie les clés avec ses estampilles à son successeur.

Soit $Role_{p,k}$ le rôle du pair *p* vers la clé *k*. Le pair *p* exécute la fonction suivante :

```
Algorithm 4: changeResponsability(K')

1: begin
2:    For each k ∈ K' do
3:    //Modifier le rôle envers la clé k
4:          Role_p,k = Master-key;
5:    //Envoyer la clé k et son estampille au successeur
6:          send (k, k.timestamp)to p.successor;
7:    End do;
8: End;
```

Figure 36. Fonction pour le maintien de la liste de clés en cas de départ

Avant de recevoir les estampilles du pair *q*, le pair *p* ne génère pas d'estampille pour les clés impliquées dans *K'*. Comme le cas de *join*, toute demande de publication de patches reçue au moment de la stabilisation est rejetée.

Considérons la configuration initiale du système P2P-LTR illustrée dans la Figure 34. Supposons que P_1 quitte le système normalement et P_2 devient le nouveau Master-key pour la clé k_1. La Figure 37 montre la nouvelle configuration du système après l'opération de départ, où le pair P_2 modifie son rôle envers la clé k_1 de Maître-key-Succ au Master-key et réplique (k_2, t_2) à P_3 (*i.e.* son successeur). La nouvelle configuration des pairs P_2 et P_3 devient comme suit :

- P_2 : Master-key de k_1 et k_2
- P_3 : Master-key-Succ de k_1 et k_2

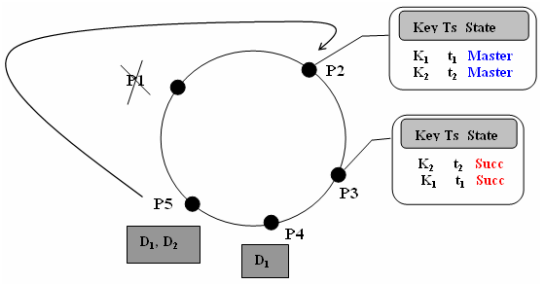

Figure 37. P2P-LTR face au départ d'un Master-key

3.2 Tolérance aux fautes

Dans cette section, nous étudions l'effet des pannes du Master-key et de son successeur sur le comportement de notre système et nous discutons la façon dont elles sont manipulées. Nous montrons qu'aucun cas de panne ne peut bloquer notre algorithme de publication de patches. Nous montrons également que même en présence de ces pannes, P2P-LTR garantit la continuité des estampilles. Pour cela, il suffit de montrer que chaque estampille produite est stockée avec son patch dans la DHT et cela avant de produire une nouvelle estampille.

Hypothèses

Nous allons d'abord décrire les hypothèses que nous considérons dans notre travail :

Hypothèse 1. (*Disponibilité des Log-Peers*) : soit $T = [t_1, t_2]$ un intervalle de temps, tel que t_1 et t_2 sont les moments où respectivement le système se lance et se termine. À n'importe quel instant $t \in T$, il doit exister au moins un pair dans le système responsable de stockage de patches. Formellement, soit D un document, et n_t le nombre de pairs qui sont disponibles à l'instant t et qui stockent des patches pour le document D. Donc nous supposons que :

$$\forall t \in T \Rightarrow (n_t \geq 1)$$

Hypothèse 2. (*Fiabilité des messages*) : nous supposons l'existence d'une transmission fiable des messages échangés entre les différents nœuds du système.

Hypothèse 3. (*Comportement correct du service lookup de la DHT*) : comme dans plusieurs protocoles conçus pour les réseaux DHTs (par exemple [CRRL+05]), nous supposons que l'opération de recherche de la DHT se comporte correctement. En effet, étant donné une clé k, la fonction *lookup* trouve correctement le responsable de k. Sinon, elle signale une erreur, par exemple dans le cas où un problème de partitionnement réseau empêche de joindre le responsable.

Hypothèse 4. (*Détection de pannes, pas forcément fiable*) : nous supposons que notre système est capable de signaler la présence d'une panne du Master-key. La signalisation

de pannes nécessite une méthode de détection ciblée qui permet à tous les pairs de surveiller le fonctionnement d'un Master-key. Cette méthode s'appuie généralement sur des envois de messages périodiques entre les pairs et le Master-key : chaque pair du système envoie périodiquement un message fiable au Master-key, et attend une réponse de ce dernier, pour vérifier sa présence. Si le Master-key ne répond pas au message (ou si sa réponse arrive en retard par rapport à un délai prévu *d*) on considère que le Master-key est en panne.

3.2.1 Gestion de la défaillance du Master-key

Pour rendre notre système P2P-LTR tolérant aux pannes de Master-Key, nous modifions notre algorithme de publication de patches proposé dans la section 2.3 comme suit. Après avoir reçu la demande de publication de patches envoyée par le pair *p* (User Peer) et vérifié la valeur d'estampille, le Master-key réplique les patches dans les Log-Peer et attend un message de confirmation de ses pairs. Chaque fois qu'un Log-Peer reçoit un patch à partir du Master-Key, il envoie de façon fiable un message au Master-key pour confirmer la réception de la réplique. Si au moins *N* messages de confirmation sont reçus par le Master-key (où $N < n_t$ est un paramètre du système qui est calculé selon la dynamicité des pairs), le Master-key envoie un message d'acquittement (*Ack)* au pair *p*, afin de valider l'opération de publication. Sinon, il envoie un message d'annulation à *p*, pour indiquer l'échec de l'opération de publication. Par conséquent, *p* essaie de nouveau l'opération de publication.

Le comportement du Master-key est modélisé par l'automate à états finis illustré par la Figure 38. Un automate [LM96] est un modèle utilisé pour la description du comportement des systèmes. Il est constitué d'états et de transitions. Un automate est dit fini s'il possède un nombre fini d'états distincts.

Nous expliquons ci-dessous comment P2P-LTR tolère les défaillances dans chaque état d'automate. Une défaillance d'un Master-key peut se produire dans l'un des états suivants :

- **Panne au moment de réception de patch (S1)** : dans ce cas, le Master-key tombe en panne juste après la réception de la demande de publication de patch. Le pair *p* détecte cette panne et annule l'opération de publication. De même pour le Master-key-Succ, en détectant la panne du Master-key, il devient le nouveau Master-key. Par conséquent, dans cet état, le protocole ne se bloque pas et la continuité d'estampille est garantie.

- **Panne au moment de la génération d'estampille (S2)** : dans cet état, comme dans le précédent, le pair *p* détecte la panne du Master-key et la publication de patch est donc abandonnée. De même, le Master-key-Succ détecte la panne du Master-key et devient le nouveau Master-key. Comme dans l'état précédent, la continuité d'estampille est assurée.

- **Panne au moment de la réplication de patch (S3)** : si le Master-key échoue dans cet état, un message d'échec est envoyé au pair *p* pour indiquer l'échec de la tentative de publication. Etant donné que notre système est capable de signaler la panne du Master-key, les Log-Peers détectent la panne du Master-key et les patches sont simplement rejetés. Après un certain temps, le successeur de l'ancien Master-key devient le nouveau Master-key. Dans cet état, comme dans le précédent, le protocole ne se bloque pas et la continuité des estampilles est assurée. Les estampilles générées par l'ancien Master-key sont abandonnées.

- **Panne au moment de la réplication de last_ts (S4)** : si le Master-key échoue dans cet état, un message d'échec est envoyé au pair *p* indiquant la panne du Master-key. Le successeur de l'ancien Master-key devient le nouveau Master-key du document. Grâce à l'ensemble des fonctions de hachage, le nouveau Master-key envoie un message au Log-Peer pour vérifier s'ils ont bien reçu le patch. Si au moins *N* réponses sont retournées, le Master-key envoie un message d'*Ack* à *p* pour indiquer le succès de l'opération de publication. Sinon, il envoie un message d'abandon à *p*. Dans cet état, comme dans l'état (S3), la continuité d'estampille est assurée.

Lemme 2. *Si pendant l'exécution du protocole le Master-key tombe en panne alors le protocole ne se bloque pas et la continuité des estampilles est garantie.*

Preuve. *Déduite par l'analyse ci-dessus.*

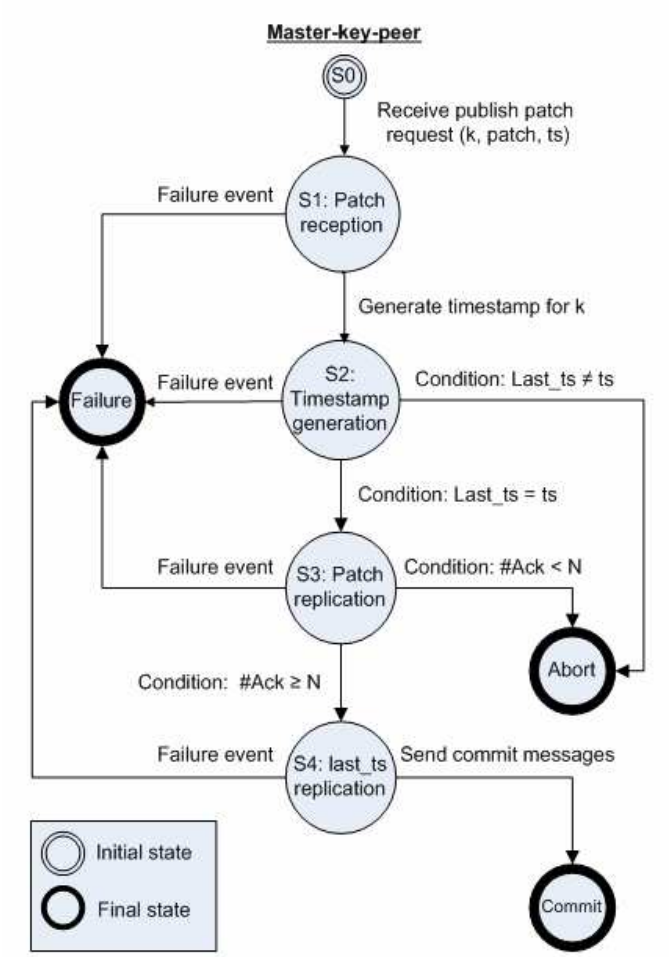

Figure 38. Automate à états finis décrit le comportement d'un Master-key

3.2.2 Gestion de la défaillance simultanée du Master-key et Master-key-Succ

Dans cette section, nous étudions un cas très pessimiste de panne. Il s'agit de la défaillance simultanée du Master-key et du Master-key-Succ. Dans ce cas, la valeur d'estampille n'est pas disponible, ni au Master-key ni au Master-key-Suc. La question est de savoir comment le prochain Master-key peut récupérer la valeur la plus récente d'estampille qui est nécessaire pour assurer l'ordre total des patches. Dans cette situation, nous profitons des estampilles qui sont stockées dans les Log-Peers.

Lorsque les pannes se produisent, les nœuds défaillants peuvent soit apparaître en tant que successeur direct d'un nœud, soit figurer dans la table de routage. L'algorithme de stabilisation de la DHT corrige les incohérences en mettant à jour les listes de successeurs et les tables de routage. De plus, cet algorithme prend en charge le déplacement de clés entre les nœuds concernés par cette panne. Par conséquent, un

nœud existant dans le réseau sera automatiquement responsable des clés de l'ancien Master-key. P2P-LTR attend alors la fin de la stabilisation de la DHT. Ainsi, toute demande de publication de patches reçue au moment de la stabilisation sera rejetée. Donc, une fois la stabilisation de la DHT terminée et en recevant la première demande de publication de patch correspondant à l'une des clés de l'ancien Master-key, le nouveau responsable commence par initialiser les valeurs d'estampille pour chaque clé (en récupérant, à partir des Log-Peers, pour chaque clé la valeur d'estampille la plus récente). Ensuite, il réplique l'ensemble des clés et estampilles sur son successeur.

Formellement, soit *p* un nouveau Master-key, et *K* un ensemble de clés dont *p* est la responsable. Pour chaque clé $k \in K$, le pair *p* récupère depuis les Log-Peers les dernières estampilles générées pour chaque clé *k*. Ensuite, *p* initialise les valeurs d'estampilles de chaque clé. Pour cela, P2P-LTR propose l'algorithme présenté Figure 39.

Soit H_r l'ensemble des fonctions de hachage utilisées pour la réplication des patches dans la DHT. Pour initialiser *k*, pour chaque $h \in H_r$, *p* récupère les patches (avec ses estampilles associées), qui sont stockés dans les Log-Peers spécifiques. Parmi les estampilles récupérées, *p* sélectionne la plus récente, notée par *last_ts*. Le pair *p* initialise ensuite la valeur de *ts* avec *last_ts*. Dans le cas où il n'y a pas de patches et d'estampilles stockés dans la DHT pour la clé *k*, *p* initialise *ts* à zéro.

```
Algorithm 5: recoveryMaster(List K)

1: begin
2:    List:= [null, 0];
3:    For each k ∈ K do
4:          ts:=0;
5:          For each hi ∈ Hr do
6:          //Obtenir l'estampille associé à la clé k
7:                Last_ts:= getTimestamp(k);
8:                if (Last_ts > ts)then
9:          //initialiser ts à Last_ts
10:               ts:= Last_ts;
11:               End if;
12:         End do;
13:   //La clé k et sa récente estampille
14:   List.add(k,ts);
15:   End do;
16: End;
```

Figure 39. Gestion de pannes du Master-key et son successeur

3.3 Propriété de vivacité

Afin de garantir la convergence des différentes répliques [SBK04], les systèmes de réplications ont besoin de satisfaire la propriété de *vivacité (liveness)* [G99]. Dans notre travail, la propriété de vivacité est exprimée par la capacité du système à continuer à fonctionner correctement (*i.e.* la continuité des estampilles est assurée) malgré la

dynamicité et les cas de pannes. Le fait de garantir la propriété de vivacité est un complément important à la propriété de cohérence à terme.

Pour assurer la propriété de vivacité, il faut observer correctement le comportement des Master-key Peers. Pour cela, nous considérons toujours les hypothèses de la section 3.2, notamment une transmission fiable des messages échangés entre les différents nœuds du système et une détection de pannes (pas forcément fiable) des Master-key Peers.

Notre protocole prend en compte le comportement dynamique des pairs (voir section 3). A chaque fois qu'un nouveau Master-key Peer rejoint ou quitte le système, P2P-LTR propose des stratégies pour faire face à ces cas. P2P-LTR assure ainsi un changement dynamique des Master-key Peers et un transfert correct des clés entre les différents nœuds du système. En outre, nous avons montré précédemment que dans les différents cas de panne du Master-key ou de son successeur, notre protocole ne se bloque pas et la continuité des estampilles est toujours assurée. Les comportements dynamiques des pairs et les cas des pannes n'ont par conséquent pas d'impact sur la garantie de la continuité des estampilles. Ceci induit que notre protocole préserve la propriété de vivacité.

Notons cependant qu'au moment de la stabilisation de la DHT (suite à un ajout ou un départ d'un pair du réseau), P2P-LTR ne produit pas d'estampilles. Ainsi, toute demande de publication des patches reçus au moment de la stabilisation est rejetée. Ce dernier cas oblige l'utilisateur à recommencer sa demande de publication. De plus, dans le cas des mises à jour concurrentes, un utilisateur peut être amené à récupérer les patches manquants (produits par d'autres utilisateurs) avant de ré-exécuter sa demande. De ce fait, la continuité des estampilles et la cohérence à terme des données sont toujours assurées, mais ceci étant, les cas évoqués ont toutefois un impact sur l'efficacité de notre protocole.

4 Analyse des coûts

Dans cette partie, nous évaluons analytiquement le coût de P2P-LTR en termes de nombre de messages et de nombre d'aller-retour (*tour*) nécessaires pour publier les patches et converger vers un état final. Plus précisément, les paramètres que nous mesurons sont les suivants :

- *La complexité de la communication pour publier un patch* : le nombre de messages échangés pour publier un patch.
- *La complexité de la communication pour la convergence* : le nombre total de messages échangés pour que tous les utilisateurs convergent vers un état final.
- *La latence pour publier un patch* : le nombre minimal de tour nécessaires pour publier un patch. Nous définissons un tour comme le nombre d'aller-retour entre deux pairs.

- *La latence pour la convergence* : le nombre de *tour*s nécessaires pour que tous les utilisateurs convergent vers un état final.

Dans notre analyse, nous supposons qu'il existe *k* utilisateurs travaillant sur le même document partagé, et après avoir fait leurs modifications locales, ils essaient en même temps de publier leurs patches. Soit *n* le nombre de Log-Peers qui maintiennent les répliques de chaque patch dans le DHT, et *N* le nombre total des pairs dans le système. Dans notre modèle, chaque utilisateur a besoin de *O (log N)* messages pour communiquer avec le Master-key (*i.e.* le coût nécessaire pour localiser le responsable d'estampillage). Le même nombre de messages est utilisé par le Master-key pour rechercher un Log-Peer et répliquer un patch sur ce pair. Pour simplifier, nous supposons que le coût d'une recherche est égal à *log (N)* messages.

4.1 Complexité de la communication pour publier un patch

Etant donné que le nombre de messages communiqués pour une opération de publication de patch dépend du nombre de patch à récupérer, nous analysons la complexité de la communication pour publier un patch selon les cas suivants : 1) le meilleur cas, c'est le cas où aucun patch ne doit être récupéré. Dans ce même cas, nous évaluons le nombre de messages requis pour qu'un utilisateur arrive à publier son patch, 2) le cas moyen, c'est le cas dans lequel nous évaluons le nombre moyen de messages échangés pour publier un patch, 3) le pire cas, est celui où (*k-1*) des patches sont nécessaires à récupérer avant la publication d'un patch, c'est-à-dire lorsqu'un utilisateur est le dernier à réussir à publier son patch.

Pour le meilleur cas, un utilisateur a besoin de *log(N)* messages pour rechercher et localiser un Master-key. Il faut alors *(n log (N))* messages pour que le Master-key localise les Log-Peers et réplique leur patch, où *n* est le nombre de Log-Peers. Deux messages sont également nécessaires pour copier l'estampille sur le Master-key-Succ et envoyer la réponse à l'utilisateur. Dans ce cas, le coût de communication est défini par l'équation suivante :

$$C_B = (n+1)\log(N) + 2 \qquad (3.2)$$

Pour le cas moyen, nous mesurons le nombre total de messages nécessaires pour la publication des patches des *k* utilisateurs, et divisons ensuite le total par *k*. La publication des *k* patches se fait sur *k* étapes. Dans la première étape, les *k* utilisateurs essayent de publier leurs patches et chacun a besoin de *log(N)* messages pour localiser le Master-key correspondant. Un seul utilisateur réussit, et les *k-1* autres reçoivent un message d'échec. Donc *(n log (N) +1)* messages sont utilisés par le Master-key pour répliquer le patch dans les Log-Peers et le Master-key-Succ. Les *k-1* utilisateurs n'arrivant pas à publier leurs patches devront récupérer le patch manquant. Pour cela il faudra échanger *2 (k-1)* messages entre ces utilisateurs et les Log-Peers.

À noter que les utilisateurs n'ont pas besoin d'utiliser l'opération de recherche pour contacter les Log-Peers puisqu'ils ont déjà obtenu leurs adresses à travers le Master-key.

Au début de la deuxième étape il reste *k-1* utilisateurs qui n'ont pas publié leurs patches. Après avoir récupéré et fusionné le patch manquant, c'est-à-dire celui du premier utilisateur, ces *k-1* utilisateurs essaient de publier leurs modifications.

Un utilisateur arrive à publier ses patches, tandis que les autres *k-2* échouent. Dans cette étape *2(k-1)* messages sont échangés entre le Master-key et les utilisateurs. Notons que dans cette étape, les utilisateurs n'ont pas besoin d'utiliser l'opération de recherche pour communiquer avec le Master car ils ont déjà obtenu son adresse. Le même processus se poursuit jusqu'à ce que le dernier utilisateur publie ses patches. Après chaque opération de publication de patch le Master-key a besoin de *(n +1)* messages pour publier le patch et la nouvelle valeur d'estampille dans les Log-Peers et le Master-key-Succ; sauf à la première étape où le Master-key a besoin de *(n log (n) +1)* messages pour localiser les Log-Peers. Ainsi, le nombre total de messages pour publier les *k* patches est : $[k\times log(N)+k+2(1+2+\ldots(k-1))+(nlog(N)+1)+(k-1)\times(n+1)+2(1+2+\ldots+(k-1))]$. Par conséquent, le nombre total de messages est défini par l'équation suivante :

$$C_{Total} = 2k^2+k[n+ log(N)]+n[log(N)-1] \quad (3.3)$$

Pour le coût moyen, on divise le coût total par *k*. En utilisant (3.3), le coût moyen de communication pour publier tous les patches est le suivant :

$$C_A(k)= (C_{Total} / k) = (2k^2+k[n+ log(N)]+n[log(N)-1])/k \quad (3.4)$$

Pour le pire cas, l'analyse des coûts peut se faire d'une manière similaire à celle du coût total, sauf que dans ce cas, le patch est envoyé une seule fois pour les Log-Peers. Ainsi, seulement *(n log (n) +1)* messages sont utilisés par le Master-key pour répliquer les patches et la dernière estampille. En utilisant (3.3), le coût de communication pour pouvoir publié un patch est le suivant :

$$C_W(k)= [C_{Total} – (k-1)*(n+1)] = 2k^2+k[log(N)-1]+(nlog(N)+1) \quad (3.5)$$

4.2 Complexité de la communication pour la convergence

Une fois que tous les utilisateurs publient leurs patches et cessent de faire des modifications sur le document, seul *(k-1)* utilisateurs doivent obtenir les patches manquants, à savoir ceux des autres utilisateurs, et ceci, afin de converger vers un état final. Chaque utilisateur a besoin de deux messages pour récupérer les patches manquants depuis les Log-Peers, soit *2 (k-1)* messages au total. En utilisant (3.3), le nombre total de messages pris pour publier les patches et assurer la convergence de toutes les copies est de $[C_{Total} + 2(k-1)]$ messages. Le coût de communication de convergence peut alors être décrit par l'équation suivante :

$$C_c(k)= 2k^2+k[log(N)+ n+2]+n[log(N)-1]-2 \quad (3.6)$$

4.3 Latence pour publier un patch

Comme dans la section 4.1, nous considérons trois cas de mesures de latence pour publier un patch : meilleur, moyen et pire cas. Dans le meilleur cas, l'utilisateur a besoin d'un aller-retour pour rejoindre le Master-key. De même un aller-retour est utilisé par le Master-key pour répliquer l'estampille et le patch sur les Log-Peers et le Master-key-Succ. Ainsi, la latence pour publier un patch est de deux allers-retours.

Dans le cas moyen, les *k* utilisateurs essayent de valider d'abord leurs mises à jour. Par conséquent, les *k* utilisateurs ont besoin d'un aller-retour pour communiquer avec le Master-key. Un utilisateur reçoit un message de validation, alors que les *k-1* reçoivent un message d'échec. Les *k-1* utilisateurs ont donc besoin d'un tour (en parallèle) pour récupérer les patches manquants. Dans l'étape suivante, ces *k-1* utilisateurs essaient de publier leurs modifications. Un utilisateur réussit, alors que les autres *k-2* échouent. Dans la deuxième étape, *k-2* utilisateurs ont besoin d'un aller-retour pour récupérer les patches manquants. Ce processus se poursuit jusqu'à ce que le dernier utilisateur réussisse à publier ses patches.

Pour chaque publication de patch, le Master-key a besoin d'un aller-retour pour publier les patches et la dernière estampille respectivement sur les Log-Peers et le Master-key-Succ. Au total *k* aller-retour sont nécessaires pour publier les patches et la dernière valeur d'estampille. La latence totale pour publier tous les patches est ainsi de *(3k-1)* tours. En divisant le nombre total d'aller-retours par le nombre d'utilisateurs, on obtient la moyenne de la latence :

$$L_A(k)= (3k-1)/k \quad (3.7)$$

Pour le pire cas, le dernier utilisateur a besoin de *(k-1)* tours pour récupérer les patches manquants des autres utilisateurs, et *k* aller-retour pour communiquer avec le Master-key. Le Master-key a besoin d'un aller-retour pour répliquer les patches et la dernière estampille sur les Log-Peers et le Master-key-Succ. Dans ce cas, la latence pour publier un patch est de *((k-1) + k +1)* tours. D'où l'équation suivante :

$$L_W(k)= 2k \quad (3.8)$$

4.4 Latence pour la convergence

Une fois qu'un utilisateur atteint l'état final, un seul aller-retour supplémentaire est nécessaire pour chacun des autres utilisateurs pour récupérer les patches manquants et atteindre le même état final. Ainsi, en considérant la latence totale de la section 4.3, *i.e.* (*3k-1)* tours, nous pouvons décrire la latence de convergence comme suit :

$$C_L(k) = (3k – 1) + (k-1) = 4k -2 \quad (3.9)$$

4.5 Synthèse

Notre analyse des coûts montre que le coût moyen de communication pour la publication du patch et le temps de latence moyen de convergence est linéaire par rapport au nombre d'utilisateurs. En outre, notre analyse montre que la latence moyenne pour publier un patch est également constante par rapport au nombre d'utilisateurs. Ces résultats montrent que P2P-LTR a une bonne capacité à passer à l'échelle dans les systèmes P2P avec un grand nombre d'utilisateurs travaillant sur les mêmes documents partagés.

5 Analyse probabiliste

Dans la section 3, nous avons présenté notre mécanisme pour la gestion des pannes. Nous avons montré qu'aucun cas de panne ne peut bloquer notre protocole de publication de patches. Cependant, la probabilité de publier un patch dès la première tentative par le Master-key Peer varie proportionnellement avec le nombre de pannes de Log-Peers et du Master-key Peer. Pour démontrer l'efficacité de notre protocole, nous montrons que la probabilité de publier un patch est souvent élevée et ça même si peu de Log-Peers sont présents dans le système.

Dans la suite de notre analyse nous supposons qu'il n'y a pas de patches à récupérer des Log-Peers. Nous notons par f la probabilité (*indépendante* et *équiprobable*) qu'un nœud soit en panne. Soit N le nombre total de Log-Peers dans le système et n le nombre de messages de confirmation reçus par le Master-key Peer afin de valider une demande de publication d'un patch.

Supposons que X est une variable aléatoire représentant le nombre de Log-Peers disponible dans le système. Donc, $P(X = i) = (1-f)^i * f^{(N-i)}$ est la probabilité que i Log-Peers soient disponible dans le système. Donc, la probabilité qu'il y ait au moins n Log-Peers disponibles dans le système est obtenue avec la formule suivante :

$$E(X) = \sum_{i \geq n}^{N} P(X = i) = \sum_{i \geq n}^{N} (1-f)^i * f^{(N-i)} \quad (3.10)$$

Nous notons par $P_r(n)$ la probabilité qu'une opération de publication de patch par Master-key Peer termine avec succès. Donc, $P_r(n)$ est défini comme suit :

*$P_r(n)$ = P(Master-key ne tombe pas en panne)*P(au moins n Log-Peers sont disponibles dans le système)*

La probabilité que le Master-key Peer soit disponible est $(1-f)$. Par conséquent, en utilisant (3.10), la probabilité de publier un patch est donnée par :

$$P_r(n) = (1-f) * \sum_{i \geq n}^{N} (1-f)^i * f^{(N-i)}$$

$$P_r(n) = (1-f) * f^N \sum_{i \geq n}^{N} (1/f-1)^i \quad (3.11)$$

Soit $S = \sum_{i \geq n}^{N} (1/f-1)^i$ la suite géométrique de raison $q = (1/f-1)$ et de premier terme $S_0 = (1/f-1)^n$.

La valeur de la somme des termes de la suite S est donnée par la formule suivante :

$$S = (1/f-1)^n * \frac{1-(1/f-1)^{(N-n+1)}}{1-(1/f-1)} \qquad (3.12)$$

En utilisant (3.12) et (3.13), nous avons la formule suivante pour P_r(n) :

$$P_r(n) = \frac{(1-f)^{(n+1)}}{(2f-1)} [\, f^{(N-n+1)} - (1-f)^{(N-n+1)} \,] \qquad (3.13)$$

$P_r(n)$ est minimum lorsque n se rapproche de N. Dans ce cas, le système est trop exigeant puisque la tolérance est nulle afin de publier un patch (*i.e.* nous exigeons N réponses de Log-Peers pour confirmer la publication de patch). Notons P_{min} la probabilité minimale pour réussir à publier un patch dans la DHT. Donc, la valeur de P_{min} est obtenue par la formule suivante :

$$P_{min} = \lim_{n \to N} P_r(n) = (1-f)^{(N+1)} \qquad (3.14)$$

L'efficacité de notre système est démontrée lorsque la valeur de la probabilité $P_r(n)$ est proche de la probabilité maximum (dénoté par P_{max}) pour une valeur faible de n. P_{max} est définie lorsque $n = 1$, c'est-à-dire il suffit d'avoir au moins un Log-Peer présent dans le système pour pouvoir publier un patch dans la DHT. Donc, P_{max} est obtenue par la formule suivante :

$$P_{max} = P_r(1) = \frac{(1-f)^2}{(2f-1)} [\, f^N - (1-f)^N \,] \qquad (3.15)$$

En utilisant les formules (3.13) et (3.15), nous pouvons estimer le nombre minimal de Log-Peers pour avoir une probabilité P_r proche de P_{max}. Par exemple, pour $N = 10$, $n = 2$ et une probabilité de panne $f = 2\%$ nous avons : $(P_r/P_{max}) = 0{,}97$, *i.e.* P_r représente 97% de la valeur du P_{max}. A partir de ce résultat, nous concluons que P_r est très proche de P_{max} pour une valeur faible de n.

6 Comparaison avec les approches existantes

Dans cette section, nous comparons les approches retenues au chapitre précédent avec notre solution P2P-LTR.

P2P-LTR versus WOOT

L'avantage de l'algorithme WOOT est qu'il est capable de passer à l'échelle puisqu'il n'utilise aucun ordonnancement sur la diffusion des messages (pas de vecteur d'horloge notamment). Malheureusement, WOOT se limite à la réconciliation de structures linéaires, et il est difficilement extensible à d'autres types de structures de données. WOOT utilise un modèle de données particulier et un profil d'opération spécifique. Pour appliquer l'algorithme au cas d'un Wiki, il est donc nécessaire de stocker les données du Wiki (les pages) en utilisant ce modèle. De plus, WOOT ne détruit pas les caractères supprimés par l'utilisateur de l'application ; il consomme donc plus de mémoire et génère des fichiers pouvant être très volumineux.

Wooki [WUM07] et XWOOT [CMML+08] sont deux systèmes Wiki P2P utilisant WOOT pour la fusion de données et un algorithme épidémique probabiliste (appelé LPBCAST) [GHKK03] pour la dissémination des modifications et la gestion du réseau. LPBCAST donne des garanties probabilistes que les modifications envoyées sont livrées à tous les nœuds connectés au réseau P2P.

En revanche, P2P-LTR utilise un modèle efficace de réseau P2P à base de DHT pour la diffusion des modifications. Dans P2P-LTR, seules les opérations de mises à jour sont stockées dans la DHT, ce qui ne nécessite pas de modification importante dans le code de l'application, comme dans le cas de WOOT. Pour la fusion de données, P2P-LTR utilise un algorithme de réconciliation basé sur l'approche de transformées opérationnelles (OT). Cet algorithme est indépendant du type des données manipulées. Par contre, l'architecture P2P-LTR est basée sur une DHT. Par conséquence, elle ne supporte pas les problèmes liés au partitionnement du réseau. C'est pour cela que nous avons fait une hypothèse sur la correction du service *lookup* de la DHT (voir Hypothèse 3, Section 3.2).

P2P-LTR versus Logoot

Logoot [WUM09] propose une solution importante de réplication optimiste applicable aux réseaux P2P structurés ou non structurés. L'algorithme de Logoot utilise un modèle de donnée spécifique. Cependant, et contrairement à WOOT, Logoot ne nécessite pas de garder un objet supprimé comme invisible au lieu d'être retiré du modèle du document.

Logoot assure la cohérence des données pour des structures linéaires. En revanche, P2P-LTR utilise un algorithme de réconciliation basé sur l'approche OT. Cet algorithme est indépendant du type de données manipulées.

P2P-LTR versus MOT2

MOT2 [CF07] propose une solution importante de réconciliation de données adaptée à un environnement P2P. L'algorithme d'intégration de MOT2 repose sur le modèle de

transformées opérationnelles. Cependant, et contrairement à toutes les solutions d'OT, MOT2 ne nécessite ni site central ni vecteur d'horloge afin d'assurer un ordre total sur les opérations. Dans MOT2, la réconciliation des répliques se fait deux à deux. Lors de chaque synchronisation, un pair envoie toutes les opérations produites sur sa réplique à l'autre pair qui peut alors déterminer quelles opérations sont concurrentes. Par conséquent, l'envoi de la totalité des opérations pose des problèmes de charge surtout lorsque l'édition devient massive. D'autre part, cet algorithme suppose l'existence de fonctions de transformation répondant aux contraintes C1 et C2 (voir chapitre 2, section 3.1.4). L'écriture de telles fonctions est un problème réputé difficile et requiert des contraintes fortes telles que la vérification des contraintes C1 et C2.

En revanche, P2P-LTR nécessite d'envoyer uniquement les opérations manquantes dans le réseau. En outre, afin de traiter de manière correcte les opérations concurrentes, P2P-LTR repose sur un algorithme d'intégration (SOCT4), dans lequel les fonctions de transformation doivent vérifier uniquement la condition C1. Elles sont donc plus simples que celles de MOT2.

P2P-LTR versus DSR-P2P

P2P-DSR [MAPV06, MPV06, MP06] propose une solution de réplication et de réconciliation importante et applicable aux réseaux P2P. P2P-DSR satisfait les principaux besoins d'une application collaborative à grande échelle : haut niveau d'autonomie, réplication multi-maître, détection et résolution de conflit, cohérence à terme parmi des répliques, hypothèses faibles concernant le réseau, et indépendance des types de données.

P2P-DSR propose un algorithme de réconciliation sémantique répartie appelé *P2P-reconciler*. Cet algorithme utilise le cadre *action-contrainte* proposé pour le système IceCube pour capturer la sémantique de l'application et résoudre des conflits de mise à jour. Cette approche paraît complexe et difficile à utiliser pour la réconciliation de simples données textuelles comme le cas des pages Wiki.

En revanche, P2P-LTR utilise l'approche OT pour la réconciliation de données. Cette approche est déjà testée et implémentée dans plusieurs applications d'édition collaborative. Elle est performante et efficace dans le cas des données textuelles.

P2P-LTR versus Telex

Comme le cas de l'approche DSR-P2P, Telex [BBMS+08] propose un algorithme de réconciliation sémantique en utilisant le cadre *action-contrainte*. De plus, un certain nombre d'applications coopératives ont été développées au dessus de Telex, par exemple un calendrier coopératif, un gestionnaire de bibliothèque de document.

Telex utilise un modèle de données particulier et un profil d'opération spécifique. En effet, il utilise un graphe des conflits entre opérations concurrentes (*ACG : Action-Constraint Graph*). L'implémentation actuelle de Telex souffre d'un excès de consommation de mémoire. L'ACG peut rapidement atteindre des tailles de plusieurs dizaines de milliers de nœuds.

7 Conclusion

Dans ce chapitre, nous avons proposé une approche originale de réplication optimiste pour un réseau P2P. Notre approche de réplication optimiste, appelée P2P-LTR, est chargée de diffuser les modifications apportées sur une réplique dans le réseau DHT, et d'intégrer les modifications reçues des nœuds distants sur la réplique locale. Ce mécanisme de réplication, permet d'assurer la *cohérence à terme* des copies répliquées et propose un mécanisme d'estampillage fiable et réparti fonctionnant sur une DHT. Ce mécanisme sert non seulement à estampiller les opérations reçues mais aussi à les stocker dans la DHT en garantissant un *ordre total continu*.

Nous avons proposé une solution complète aux problèmes qui peuvent se produire pendant l'exécution du protocole suite au comportement dynamique des pairs. En particulier, nous avons traité les problèmes suivants : le cas où un nouveau Master-key rejoint le système ou le quitte normalement; le cas où le Master-key ou son successeur quittent le système sans notification. Nous avons montré que si, pendant l'exécution de notre protocole, le Master-key ou le Master-key-Succ tombent en panne alors notre protocole ne se bloque pas et la continuité des estampilles est garantie.

Dans notre analyse des coûts, nous avons montré que le coût moyen de communication pour la publication du patch et le temps de latence moyen de convergence est linéaire par rapport aux nombres d'utilisateurs. En outre, notre analyse montre que la latence moyenne pour publier un patch est également constante par rapport au nombre d'utilisateurs. Ces résultats montrent que P2P-LTR a une bonne capacité de passer à l'échelle dans les systèmes P2P avec un grand nombre d'utilisateurs travaillant sur les mêmes documents.

Dans ce chapitre, nous avons aussi comparé notre solution avec les approches retenues dans le chapitre précédent, à savoir WOOT, Logoot, MOT2, Telex et DSR-P2P. Nous avons constaté que P2P-LTR est indépendant du type de données manipulées, tolérant aux pannes, passe à l'échelle et beaucoup plus simple à mettre en œuvre que les autres solutions. Cependant, nous pensons que ces différentes solutions de réplication P2P sont mieux adaptées que d'autres selon l'application. Par exemple, Telex et DSR-P2P peuvent offrir une plus grande performance dans le cas des applications richement sémantique (par exemple calendrier coopératif). P2P-LTR quant à lui, met en œuvre des techniques simples pour les applications d'édition de texte collaborative.

P2P-LTR est actuellement diffusé sous forme de logiciel libre [TDPA+08] et également intégré dans l'architecture XWiki [XWIKI07] P2P. Dans le chapitre suivant, nous étudions les performances de notre solution de réplication optimiste.

CHAPITRE 4

IV Evaluation de Performances

Dans ce chapitre, nous allons évaluer les performances de notre approche à travers la simulation. Nous décrivons tout d'abord l'outil utilisé à savoir le simulateur PeerSim. Ensuite, nous présentons et analysons les résultats de nos expérimentations. Enfin, à partir de ces expériences nous montrons l'efficacité de notre solution.

1 Environnement de simulation

Notre simulation est basée sur Chord [SMKK+01] comme une DHT. Le service *lookup* de la DHT Chord est robuste dans les cas de pannes et garantit une forte disponibilité du système. Nous nous sommes basés sur le simulateur PeerSim dans notre implémentation. PeerSim est un logiciel libre, écrit en java et destiné à simuler le mode de fonctionnement de réseau P2P. Il permet alors de concevoir et de tester toute sorte d'algorithme des réseaux P2P dans un environnement dynamique. Le code source peut être téléchargé à partir de [Pee10]. Ce logiciel est caractérisé essentiellement par :

- Passage à l'échelle (plus de un million de nœuds).
- Une très grande dynamicité (arrivée/départ) des nœuds.
- Une architecture ouverte et composée de modules.

Essentiellement, PeerSim supporte deux types de simulation qui sont :

- La simulation par cycle : la simulation s'effectue dans un ordre séquentiel. Dans chaque cycle, chaque protocole peut exécuter son comportement.
- La simulation par évènement : elle supporte la concurrence ; un ensemble d'évènements est planifié et les protocoles d'un nœud sont exécutés en fonction des évènements survenus.

Le Tableau 6 décrit les paramètres de notre simulation. Nous utilisons les valeurs des paramètres qui sont typiques aux systèmes P2P [SGG02]. Le temps de latence entre deux pairs est un nombre aléatoire normalement distribué avec une moyenne de 200 ms. Le simulateur nous permet de réaliser des tests jusqu'à 10.000 pairs. Dans nos expériences, chaque T = 30 secondes, chaque utilisateur sélectionne un document qu'il modifie et tente de publier ses patches sur la DHT. Le nombre par défaut de documents partagés pour chaque utilisateur est de 10. La valeur par défaut du nombre moyen d'utilisateurs qui travaillent sur le même document est de 5. La valeur par défaut du taux moyen pour publier un patch est $\alpha = 1$ *toutes les 2 mn*. Dans notre simulation, nous considérons que le nombre de répliques pour chaque patch est de 10, soit $|H_r|=10$.

Dans nos expériences, nous comparons les performances de P2P-LTR par rapport à So6*, une variante de So6 [MOSM+03, OMSI04] qui réplique les données partagées dans la DHT pour améliorer la disponibilité des données. So6* utilise un estampilleur central pour déterminer un ordre total continu sur les opérations. Lors de nos expériences, nous mesurons les performances de la publication et de la récupération des patches en termes de temps de réponse. Le temps de réponse correspond au temps nécessaire pour publier un patch généré après une opération de mise à jour.

Tableau 6. Les paramètres de simulation

Paramètres	Valeurs
Latence	Un nombre aléatoire normalement distribué avec une moyenne de 200 ms, Variance = 100
Taille du réseau	10 000 pairs
Nombre de Log Peers pour chaque donnée	10
Nombre de documents partagés pour chaque user	10
Taux moyen de mise à jour et de publication d'un patch généré sur chaque réplique	Programmer par un processus aléatoire uniforme, soit un patch toute les 2 minutes
Taux d'échec	2%

2 Résultats

Dans cette section, nous présentons les résultats expérimentaux obtenus.

2.1 Passage à l'échelle

Dans cette section, nous étudions le passage à l'échelle de P2P-LTR. Pour cela, nous étudions l'effet du nombre moyen d'utilisateurs par document sur les performances de P2P-LTR.

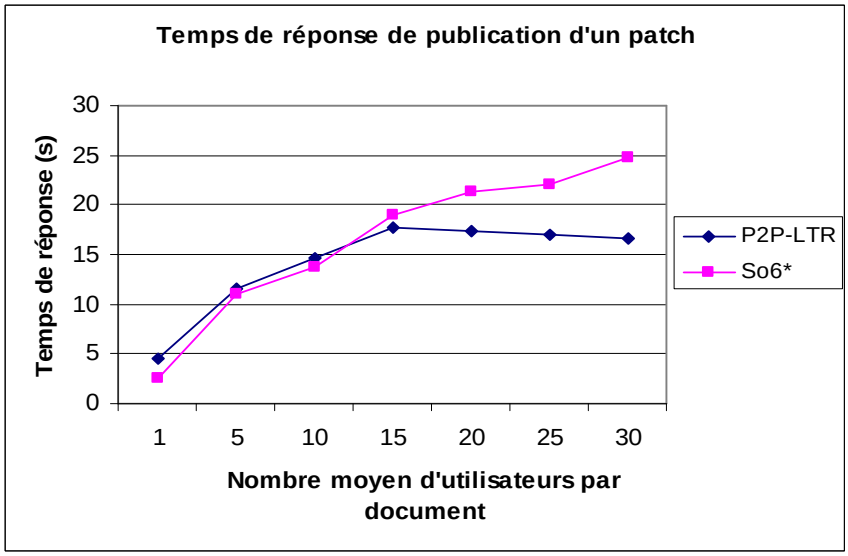

Figure 40. Temps de réponse/Nombre d'utilisateurs par document

La Figure 40 montre le temps de réponse de l'opération de publication d'un patch en variant le nombre moyen d'utilisateurs jusqu'à 30 utilisateurs par document tout en considérant les autres paramètres de simulation figurant dans le Tableau 6. Le nombre moyen d'utilisateurs qui travaillent sur le même document a un impact très faible sur le temps de réponse de P2P-LTR ce qui signifie un excellent passage à l'échelle par rapport aux nombre d'utilisateurs. Lorsque le nombre d'utilisateurs par document est plus de 15, le temps de réponse de P2P-LTR est meilleur que So6*. Plus le nombre d'utilisateurs par document est élevé, plus est le facteur par lequel P2P-LTR surpasse So6*. La raison est qu'avec So6*, la génération d'estampille et la publication de patch pour tous les documents sont exercées par un seul pair. En revanche, avec P2P-LTR la responsabilité de la production d'estampille et de stockage de patches sont entièrement répartis sur les différents pairs de la DHT.

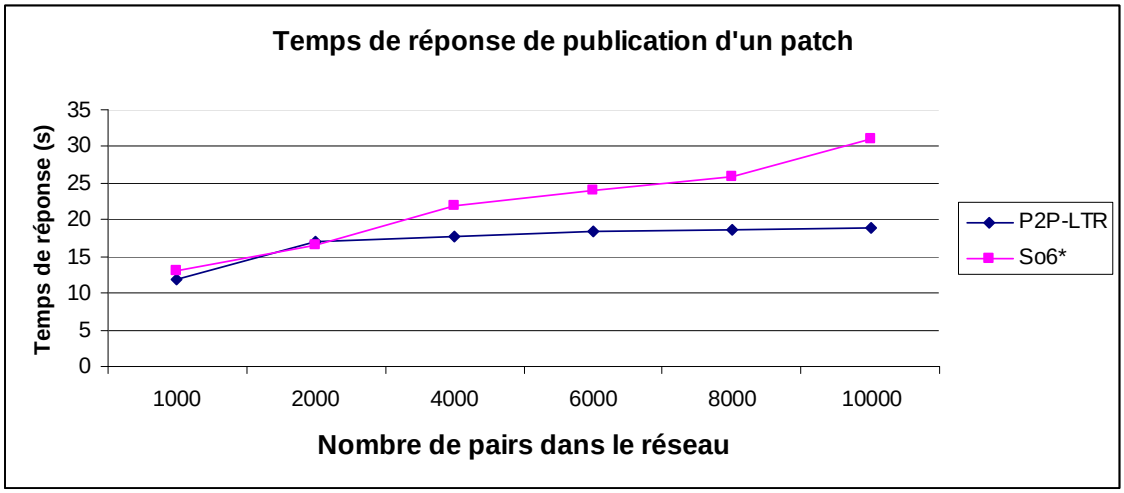

Figure 41. Temps de réponse/Nombre de pairs

En utilisant le simulateur, la Figure 41 montre le temps de réponse de publication d'un patch en variant le nombre de pairs jusqu'à 10 000 pairs tout en considérant les autres paramètres de simulation figurant dans le Tableau 6. Le nombre de pairs dans le réseau a un impact faible sur le temps de réponse de P2P-LTR ce qui signifie un excellent passage à l'échelle par rapport aux taille du réseau. Plus le nombre de pairs est élevé, plus est le temps de réponse de So6* surpasse P2P-LTR. La raison est qu'avec So6*, la génération d'estampille et la publication de patch pour tous les documents sont exercées par un seul pair. En revanche, avec P2P-LTR la responsabilité de la production d'estampille et de stockage de patches sont entièrement répartis sur les différents pairs de la DHT. Par conséquent, plus le nombre de pairs est élevé dans le réseau plus la responsabilité des estampilles est mieux réparties entre les pairs du réseau.

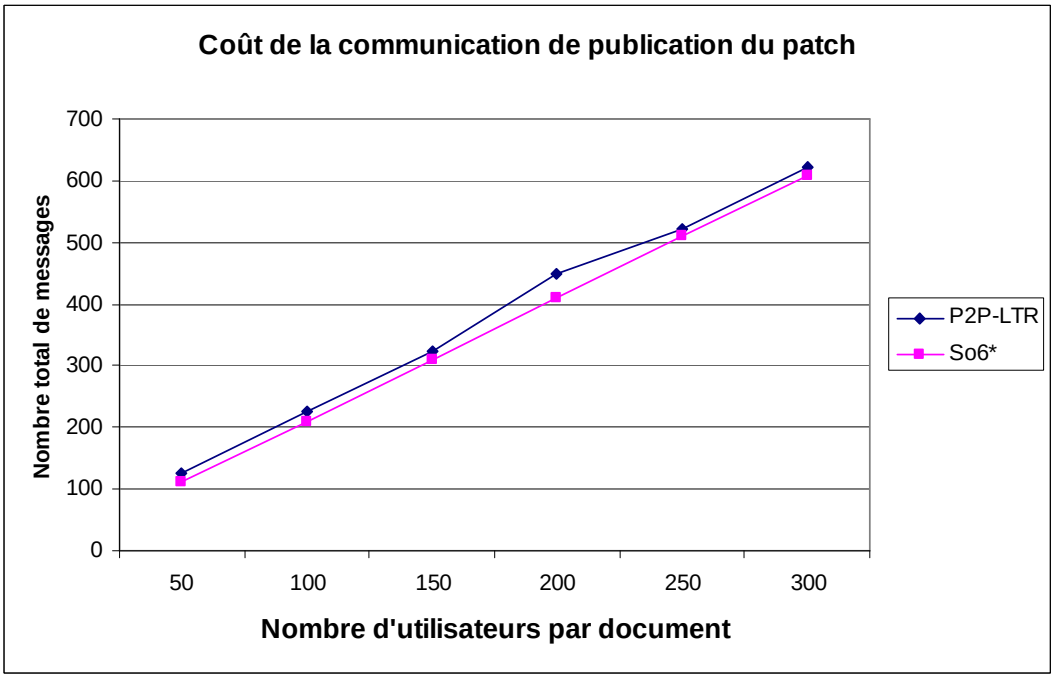

Figure 42. Nombre total de messages/Nombre d'utilisateurs par document

En utilisant le simulateur, la Figure 42 montre le nombre total de messages tout en augmentant le nombre total d'utilisateur par document jusqu'à 300, avec les autres paramètres de simulation présentés dans Tableau 6. Les résultats montrent que le coût de la communication pour P2P-LTR et So6* augmente linéairement avec le nombre d'utilisateur. Cependant, le coût de la communication de P2P-LTR est un peu supérieur à celui de So6*. La raison c'est que P2P-LTR effectue chaque fois un *lookup* dans la DHT pour trouver le Master-key de document. Cette légère augmentation du coût de la communication de P2P-LTR est le prix à payer pour garantir la continuité d'estampilles et la résistante aux pannes.

2.2 L'effet du nombre de répliques sur le temps de réponse

Dans les systèmes dynamiques, nous avons besoin de nombreuses répliques pour garantir une haute disponibilité des données. Nous allons étudier l'effet du nombre de répliques, reproduit pour chaque patch dans la DHT (dans les Log-Peers), sur les performances de P2P-LTR et So6 *.

En utilisant notre simulateur, nous avons étudié comment évolue le temps de réponse tout en augmentant le nombre de réplique. Nous considérons 15 utilisateurs par document. Les autres paramètres de simulation sont indiqués dans leTableau 6. Comme présentés dans la Figure 43, les résultats montrent qu'en augmentant le nombre de répliques de P2P-LTR ou So6*, le temps de réponse pour la publication du patch diminue. Ceci est dû au fait que l'augmentation du nombre de répliques diminue le temps moyen pour trouver les patches manquants. Ainsi, le temps de réponse de l'opération de publication d'un patch diminue. Toutefois, le temps de réponse de P2P-LTR vaut mieux que So6*. En effet, pour ce dernier la production et la publication d'un patch estampillé pour tous les documents sont réalisées par un seul pair.

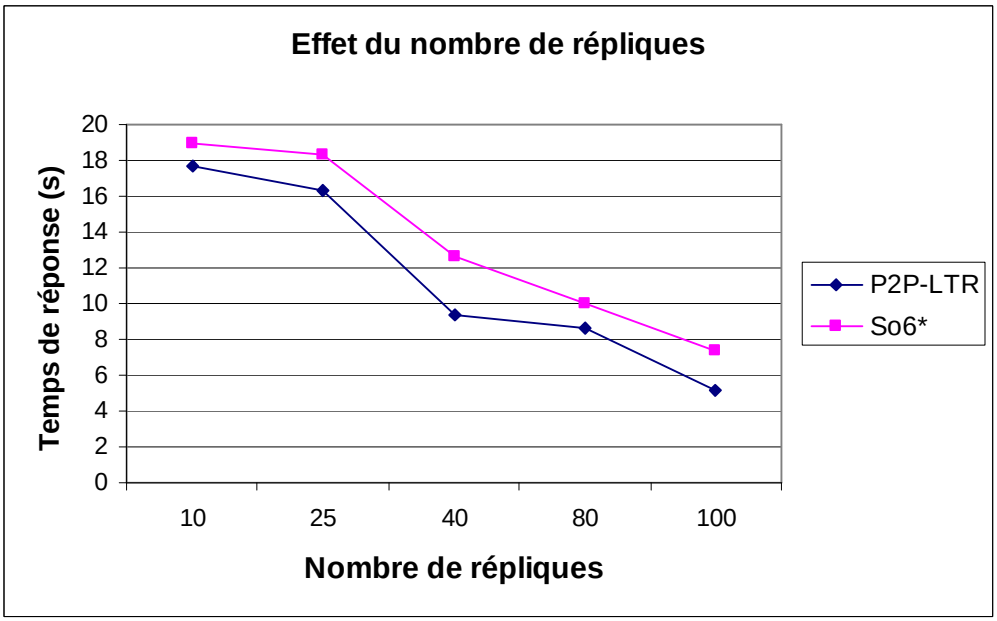

Figure 43. Temps de réponse/Nombre de répliques

2.3 L'effet de la fréquence des mise à jour sur le temps de réponse

Dans cette section, nous étudions l'effet de la fréquence des mises à jour sur les performances des P2PLTR et So6*. Dans les expériences précédentes, les mises à jour sur chaque ensemble de données ont été faites aléatoirement suivant un processus avec un taux moyen de 2 mises à jour par minute. Dans cette section, nous faisons varier le taux moyen (fréquence des mises à jour) et nous étudions ses effets sur les temps de réponse.

En utilisant notre simulateur, la Figure 44 montre l'évolution du temps de réponse en fonction de la fréquence des mises à jour. Considérant les paramètres de simulation présentés dans le Tableau 6, les résultats montrent que le temps de réponse de P2P-LTR diminue en augmentant la fréquence des mises à jour. La raison est que l'augmentation de la fréquence des mises à jour diminue la distance entre l'actuelle et la dernière mise à

jour, ce qui diminue le nombre de patches manquants. Par conséquent, le temps nécessaires pour récupérer les patches manquants diminue.

Le temps de réponse de P2P-LTR est nettement meilleur que So6*. La raison est qu'avec So6*, les opérations de production d'estampille et de stockage des patches estampillés pour tous les documents sont exercées par un seul pair. En effet, en augmentant la fréquence des mises à jour, l'estampilleur central prend plus de temps pour générer des estampilles et publier les patches dans la DHT.

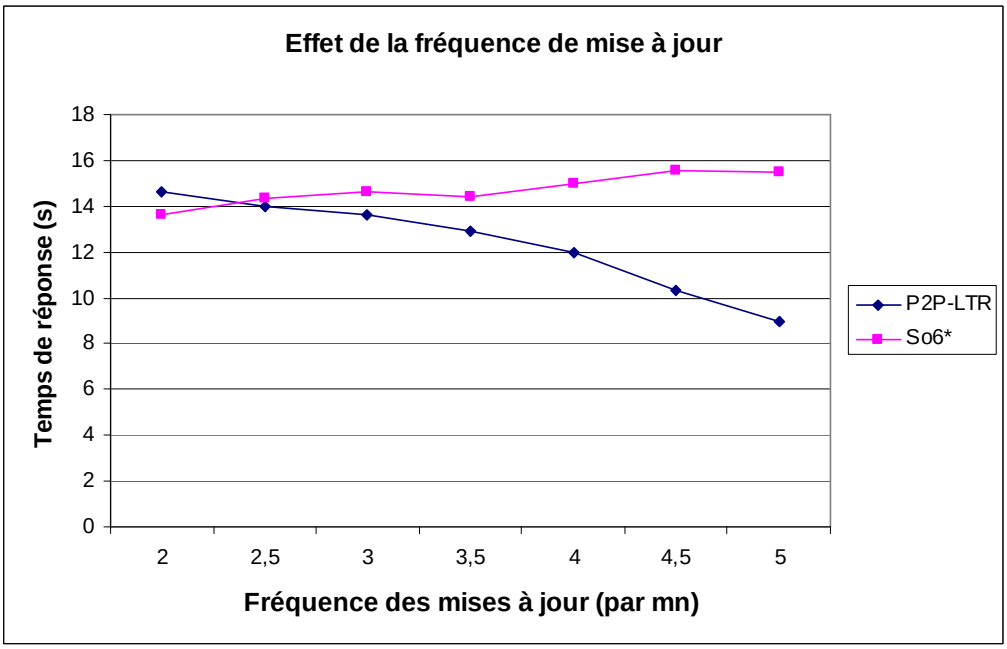

Figure 44. Temps de publication de patches/Fréquence des mises à jour

2.4 L'effet des pannes sur la continuité des estampilles

Etudions maintenant l'effet des pannes sur la continuité des estampilles utilisées pour les mises à jour des données. Dans nos expériences, nous mesurons le taux de continuité d'estampilles. Nous avons varié le taux d'échec, et nous observons son effet sur le taux de continuité d'estampilles.

La Figure 45 montre le taux de continuité d'estampilles pour P2P-LTR et So6* tout en augmentant le taux d'échec. Les échecs des pairs n'ont pas d'effet négatif sur la continuité des estampilles utilisées par P2P-LTR puisque notre protocole assure la continuité d'estampilles. Toutefois, dans le cas So6*, en présence de défaillances dans les pairs, il y a des trous dans les valeurs d'estampilles générées par So6*. Par conséquent, So6* n'est donc pas adapté aux applications qui nécessitent une continuité d'estampilles.

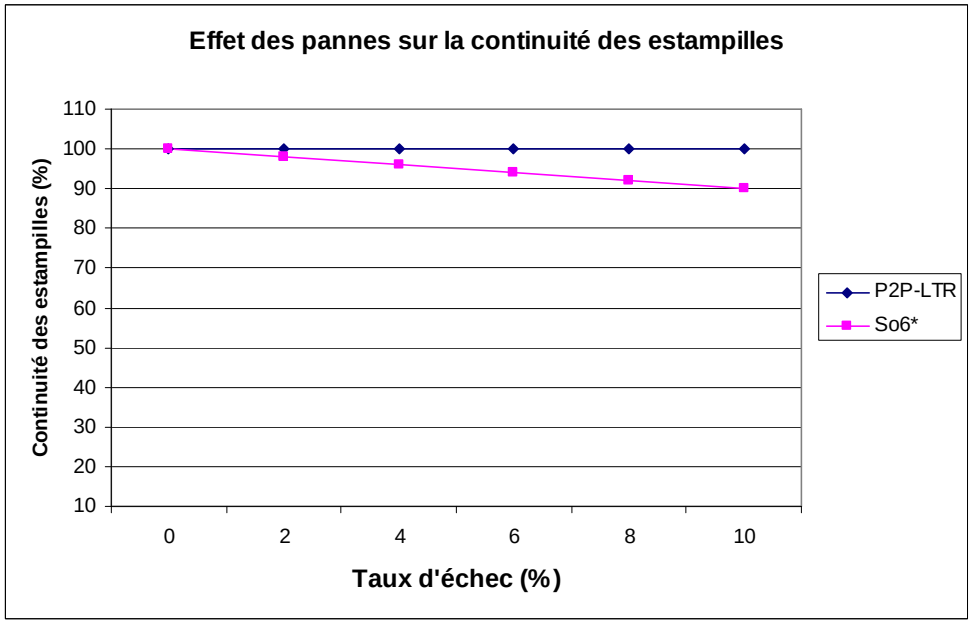

Figure 45. Continuité des estampilles/Taux d'échec

3 Conclusion

Pour valider notre contribution, nous avons implémenté P2P-LTR et nous avons simulé un environnement P2P en utilisant PeerSim comme simulateur. Nous avons évalué les performances de P2P-LTR selon les paramètres suivants : le nombre moyen d'utilisateurs travaillant sur le même document, le nombre de répliques dans le réseau, l'effet de la fréquence des mises à jour de et l'effet des pannes. Afin d'évaluer le temps de réponse de P2P-LTR (à savoir le temps nécessaire pour publier un patch), nous avons comparé les résultats de P2P-LTR avec les résultats de So6* (variante de la solution So6).

Les résultats montrent un bon passage à l'échelle de P2P-LTR. Ils montrent que P2P-LTR surpasse la solution So6* par un facteur de 1,5 lorsque 30 utilisateurs travaillent simultanément sur le même document (voir Figure 40). Les résultats montrent également que le coût de communication pour publier des patches de P2P-LTR est linéaire par rapport au nombre d'utilisateurs qui travaillent sur le même document (voir Figure 42).

En outre, les résultats montrent que les performances de P2P-LTR sont très bonnes lorsque le nombre de Log-Peers atteint le 80 (voir la Figure 43). Toutefois, le nombre de Log-Peer ne peut pas augmenter indéfiniment en raison de l'importance du coût de la communication. Nous avons étudié aussi l'effet de la fréquence des mises à jour sur les performances de P2P-LTR (voir Figure 44). Les résultats montrent que, contrairement à

So6*, le temps de réponse de P2P-LTR diminue en augmentant la fréquence des mises à jour. Nos résultats montrent également que P2P-LTR fonctionne correctement, même dans les cas des pannes et garantit la propriété de continuité d'estampille.

En conclusion, les résultats de notre évaluation de performances montrent l'efficacité et le passage à l'échelle de notre solution.

CHAPITRE 5

V Mise en œuvre et Validation

Notre approche de réplication est validée par son utilisation dans le logiciel XWiki P2P [XWIKI07], un Wiki en logiciel libre. De plus notre service P2P-LTR a été implémenté comme un prototype [TDPA+08]. Dans ce chapitre, nous décrivons l'architecture de XWiki P2P, l'implémentation de nos algorithmes dans XWiki P2P, et le prototype de P2P-LTR. Nous présentons également les environnements choisis, les technologies et les plateformes utilisées.

Le plan de notre chapitre est le suivant. La première partie présente les motivations et les besoins pour mettre en œuvre une architecture XWiki P2P. La seconde partie décrit l'architecture d'un point de vue global et introduit les spécifications et les implémentations de différents composants. Enfin, et avant de conclure, la troisième partie décrit les scénarios de déploiement de notre architecture.

1 XWiki: un Wiki fonctionnant sur un réseau P2P

XWiki [XWIKI07] est un Wiki de seconde génération capable de gérer des données structurées et ainsi que non structurées. Un Wiki est un outil d'édition de documents en ligne dans lequel les utilisateurs peuvent facilement tisser des liens entre les documents. Il permet la création et la modification de documents à travers une interface Web. Lors de la navigation dans le contenu d'un Wiki, un utilisateur peut à tout moment entrer en mode édition pour modifier le contenu de la page. Lorsqu'il a terminé, il peut sauvegarder le nouveau contenu qui vient alors remplacer l'ancien contenu de la page. L'architecture actuelle de XWiki est basée sur une architecture classique client-serveur : un serveur détient les données et les utilisateurs consultent et éditent les pages à travers un navigateur Web qui interagit avec le serveur. Cette architecture, malgré sa simplicité et sa popularité, a des limites : par exemple, l'utilisation d'un serveur central est un point de faiblesse qui rend le système vulnérable aux pannes. De plus, le système impose d'être connecté au serveur pour réaliser des opérations d'édition.

L'objectif du projet ANR XWiki Concerto [XWIKI07] est de concevoir une architecture XWiki P2P qui dépasse ces limites afin de passer à l'échelle en nombre d'utilisateurs et de supporter la mobilité des clients. Une telle application demande des capacités générales de réplication avec différents niveaux de granularité (ligne, mot) et un mode multi-maître où plusieurs répliques d'une même donnée peuvent être mises à jour par plusieurs pairs en parallèle.

D'un point de vue architectural, un Wiki P2P est un Wiki dans lequel aucun site ne joue seul le rôle de serveur : chaque site embarque les données et la logique applicative pour rendre le service. En d'autres termes, tous les sites sont des serveurs, et ces

serveurs se coordonnent entre eux pour assurer la cohérence globale. Plus concrètement, chaque participant au réseau possède sur sa machine :

- Un serveur Wiki étendu pour assurer sa coordination avec les autres sites.
- Un client web classique pour interagir avec ce serveur local.

La mise en œuvre de cette architecture suppose l'introduction d'un composant particulier chargé de propager les modifications apportées sur un serveur local à l'ensemble des serveurs Wiki participants au réseau. C'est ce composant qui assure la coordination avec les autres sites. Son objectif est de garantir la cohérence globale du service ; dans notre cas, cela se traduit par s'assurer que chaque serveur Wiki dispose des mêmes données (les mêmes pages avec le même contenu). Le rôle de ce composant est donc de répliquer les modifications apportées par un site sur sa copie locale des pages Wiki à l'ensemble des copies détenues par les autres serveurs participant au réseau.

2 Architecture XWiki P2P

Cette partie est consacrée à la description de l'architecture conçue pour la réplication des données d'un Wiki sur un réseau P2P, et à la spécification des différents composants de cette architecture. L'architecture proposée assure la réplication des modifications (patches) produites sur les pages Wiki en utilisant une approche à base de transformées opérationnelles pour la fusion des données, et une DHT pour réaliser un estampillage fiable et un stockage réparti des patches.

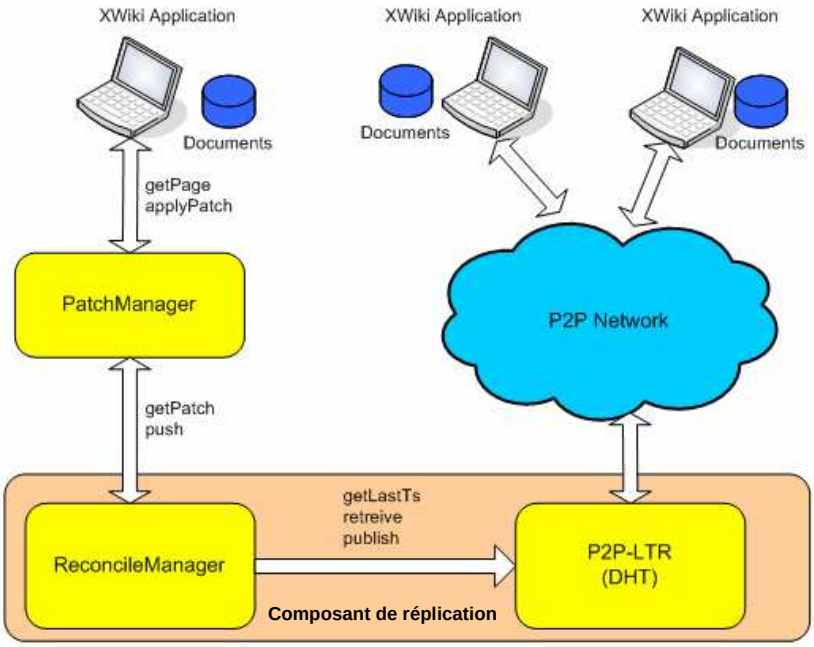

Figure 46. Architecture XWiki P2P

La Figure 46 illustre l'architecture générale de notre système. Cette architecture a été conçue avec l'objectif de limiter l'impact sur l'application XWiki (*i.e.* éviter la modification du code de XWiki). Ainsi, le composant de réplication incluant la fusion et la diffusion des données est un composant indépendant interagissant avec le serveur XWiki à travers une interface Web. Dans cette architecture, XWiki est vu comme un service Web capable de produire des modifications (patches) sur un ensemble de pages Wiki, et d'appliquer des modifications sur un ensemble de pages Wiki. Le service de réplication est donc en charge de répliquer une modification, produite par un service Web XWiki, dans la DHT. Une modification correspond aux changements produits par un utilisateur lorsqu'il sauvegarde une page. Lors de la réception d'un patch distant, le composant de réplication fusionne ce patch avec les modifications déjà réalisées localement (après éventuellement transformation) et génère un nouveau patch tenant compte de ces modifications. Il applique ensuite ce patch sur les pages Wiki locales en utilisant l'API web offerte par le serveur XWiki.

Dans la suite, nous détaillons les différents composants de notre architecture.

2.1 Description des composants

Dans notre architecture nous distinguons les composants suivants : *PatchManager*, *ReconcileManager* et *P2P-LTR*.

2.1.1 PatchManager

PatchManager est un composant permettant d'abstraire les interactions entre *ReconcileManager* et l'application XWiki. Ce composant assure la circulation de patches échangés entre l'application cliente XWiki et le composant de réplication. Ce composant génère, applique, et fournit des patches. Lorsque le bouton de sauvegarde XWiki est pressé, un ensemble de changements est effectué (ex. l'ajout et/ou la suppression des lignes à une ou plusieurs positions). Cet ensemble de changements est représenté sous la forme d'un patch, c.à-dire une séquence d'opérations. Dans notre architecture, le *PatchManager* est implémenté comme un service XWiki. Ce composant est notifié par le noyau d'XWiki à chaque changement effectué. Il est accessible via une API REST [FT02]. Il peut générer un patch à partir de deux versions d'un document XWiki. Tableau 7 résume le composant *PatchManager*.

Tableau 7. Composant PatchManager

Composant	
PatchManager: interface de patches pour XWiki	
Responsabilités	**Collaborateurs**
- Gère le journal des opérations d'un serveur XWiki. - Applique des patches sur un serveur XWiki.	- *ReconcileManager*

2.1.2 ReconcileManager

Tableau 8. Composant ReconcileManager

Composant	
ReconcileManager: composant de fusion de patches concurrents	
Responsabilités	**Collaborateurs**
- Fusion des patches concurrents.	- *PatchManager* - *P2P-LTR*

Ce composant (voir Tableau 8) a pour fonction d'intégrer les patches distants dans une copie locale en utilisant un ensemble de fonctions de transformations. Les fonctions de transformations ont donc pour rôle de gérer les problèmes de concurrence entre patches émis en parallèle. Lorsqu'un patch distant est récupéré depuis la DHT, ce patch est transformé par ces fonctions pour tenir compte des opérations locales non encore diffusées (et donc concurrentes). Dans le même temps, ces opérations locales sont également transformées pour tenir compte des opérations distantes reçues. Les fonctions de transformation sont donc les fonctions qui réalisent effectivement et concrètement la fusion des patches concurrents. Ces fonctions sont décrites en détail dans l'Annexe C. Pour que l'algorithme de transformation soit correct, les patches relatifs à une page doivent être *totalement ordonnés*. Cette garantie est offerte par le composant *P2P-LTR*. En effet, le *ReconcileManger* n'échange pas directement les patches avec les autres applications clientes, il doit d'abord les faire transiter par le composant *P2P-LTR*. Plus précisément, le *ReconcileManger* doit faire passer les patches par les estampilleurs (du composant *P2P-LTR*) qui, à leur tour, les font valider puis stocker dans la DHT de manière persistante (afin d'améliorer la disponibilité de données) et selon l'ordre des estampilles.

2.1.3 P2P-LTR

Tableau 9. Composant P2P-LTR

Composant	
P2P-LTR: estampillage et réplication des patches	
Responsabilités	**Collaborateurs**
- Diffuser les patches sur la DHT en garantissant un ordre total continu sur les patches d'une même page. - Récupération d'un patch en donnant son identifiant.	- *ReconcileManager*

Ce composant (voir Tableau 9) a pour fonction d'augmenter la disponibilité des données en répliquant les patches sur plusieurs nœuds de réseau. *P2P-LTR* sert également à estampiller les patches selon un ordre total continu avant de les stocker dans la DHT. Pour cela, *P2P-LTR* propose un mécanisme d'estampillage fiable, réparti et fonctionnant sur un réseau P2P utilisant une DHT. S'il y a des nouveaux patches dans la DHT, *P2P-LTR* doit d'abord recevoir tous les patches estampillés et les envoyer

ensuite à son moteur de réconciliation (*ReconcileManager*). Il est ensuite possible d'estampiller les opérations locales et de les stocker dans la DHT.

2.2 Interface des composants

Cette partie décrit les interfaces des différents composants de l'application. Trois interfaces sont présentes (voir Figure 47): l'interface de publication/application de patches (*IPatchManager*) fournie par l'application XWiki, l'interface d'intégration des patches distants dans une copie locale (*IReconcileManager*) et l'interface d'estampillage et de réplication de patches dans le réseau DHT (*IP2P-LTR*).

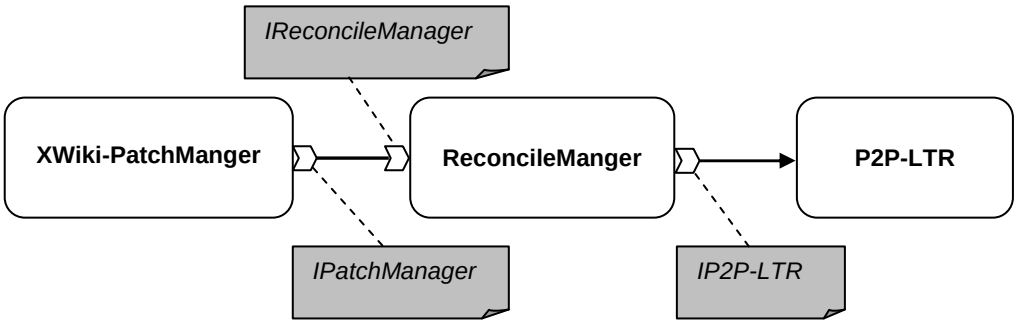

Figure 47. Interfaces XWiki-PatchManger/ ReconcileManager/P2P-LTR

2.2.1 IPatchManager

Cette interface est implantée par XWiki (ou un autre fournisseur de contenu wiki) et utilisée par le composant *ReconcileManger*; elle permet à XWiki de publier les patches correspondants à des modifications locales pour être diffusées sur le réseau, et à *ReconcileManger* d'appliquer sur les pages locales des patches distants après fusion et transformation.

La Figure 48 montre les méthodes nécessaires à la gestion du flux de patches transitant entre le composant *ReconcileManger* et le composant *PatchManager*.

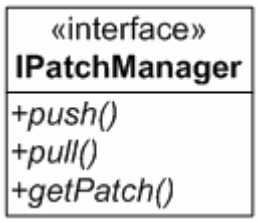

Figure 48. Interface PatchManager

- La méthode *push(p)* est utilisée pour appliquer un patch *p*. C'est à l'appelant de s'assurer que le patch n'a pas déjà été appliqué.

- La méthode *pull()* est utilisée pour chercher un nouveau patch produit localement. S'il existe un patch *p* disponible sur un serveur XWiki, la méthode *pull()* retourne *p*. Sinon, elle retourne *null* (cela signifie qu'il n'y a pas de nouveau patch disponible).
- La méthode *getPatch(key)* retourne le patch identifié par la clé *key* ou *null* si ce patch n'existe pas.

2.2.2 IReconcileManager

Cette interface est implantée par *ReconcileManger* et utilisée par XWiki pour piloter l'application de réplication. L'interface (voir Figure 49) contient les méthodes suivantes :

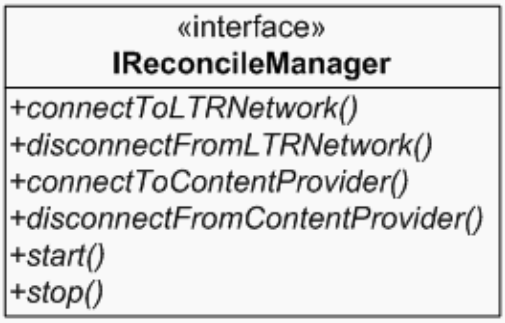

Figure 49. Interface ReconcileManager

- *connectToLTRNetwork(peer)*: cette méthode connecte le composant ReconcileManager au noeud *peer* du réseau P2P-LTR.
- *disconnectFromLTRNetwork() et reconnectToLTRNetwork()* : ces méthodes permettent de se déconnecter/reconnecter temporairement du réseau P2P (voir Figure 50). En mode déconnecté, les patches produits ne sont plus diffusés aux nœuds du réseau P2P-LTR et plus aucun patch n'est reçu en provenance de celui ci. De plus, si le fournisseur de contenu (i.e. le serveur XWiki) et le composant *ReconcileManager* sont encore connectés, ils ne le sont que "virtuellement" puisque le composant de fusion *ReconcileManager* (RM) arrête de consommer les patches tant qu'il n'a pas accès au réseau P2P-LTR.

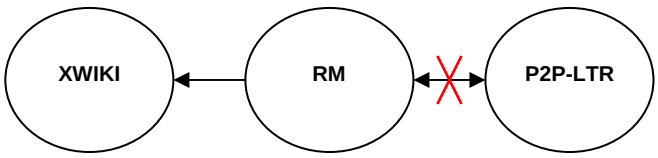

Figure 50. Déconnexion du réseau P2P-LTR

- *connectToContentProvider() et disconnectFromContentProvider()*: ces méthodes permettent de connecter/déconnecter (voir Figure 51) le composant *ReconcileManager* au fournisseur de contenu (i.e. le serveur XWiki dans le cas

présent). Une fois qu'un fournisseur de contenu se connecte à un composant de fusion, seul ce fournisseur devra être reconnecté à ce composant. De plus, si le composant *ReconcileManager* est encore connecté au réseau P2P, ils ne l'est que "virtuellement" puisque *ReconcileManager* arrête de consommer les patches tant qu'il n'a pas accès au fournisseur de contenu.

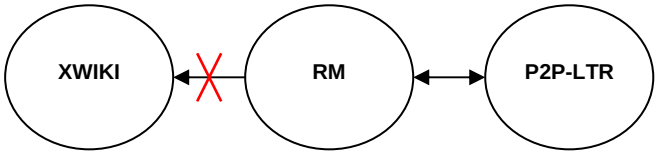

Figure 51. Déconnexion du fournisseur de contenu XWiki

- *start()*: cette méthode déclenche l'exécution de l'algorithme de réconciliation.
- *stop()*: cette méthode interrompt l'exécution de l'algorithme de réconciliation.

2.2.3 IP2P-LTR

Cette interface est implémentée par *P2P-LTR* et utilisée par *ReconcileManger* pour interagir avec l'estampilleur et la DHT. L'ensemble des méthodes définies dans cette interface est comme suit (voir Figure 52):

- *join():* cette méthode permet de créer un nouveau réseau DHT ou de joindre un réseau DHT existant.
- *getLastTimestamp(key)*: cette méthode retourne la dernière estampille générée pour la clé *key*. La clé *key* identifie une page Wiki. Le nombre retourné correspond à la dernière estampille produite pour cette page. Par exemple, si il y a eu 3 modifications sur la page, alors *lastTimestamp(key)* retourne 3.
- *retrieve(key, n)*: cette méthode permet de récupérer le patch *n* produit sur la page wiki identifiée par la clé *key*.
- *sendToPublish(key, patch, timestamp)*: cette méthode publie le *patch* produit sur la page *key* estampillé par l'estampille *timestamp*. Si l'opération réussit alors la méthode renvoie la valeur *timestamp*. Si elle échoue alors la méthode renvoie l'estampille *last_ts* disponible pour la page *key*. *last_ts* est strictement supérieur à *timestamp*.
- *leave()*: cette méthode permet a un nœud du réseau DHT de quitter la DHT.
- *getValidCounters()*: cette méthode permet de récupérer l'ensemble des couples *(key,(timestamp, status))* stockés sur un noeud donné de la DHT. Avec "*status"* nous représentons le rôle du nœud par rapport à la clé *key*, c'est-à-dire Master-key ou Master-key-Succ.
- *responsibleNodeSet()* : cette méthode retourne un tableau contenant dans l'ordre :
    - Le noeud responsable de la clé *key*.
    - Le futur nœud responsable de la clé *key*.
    - Le futur "futur" nœud responsable de la clé *key*.

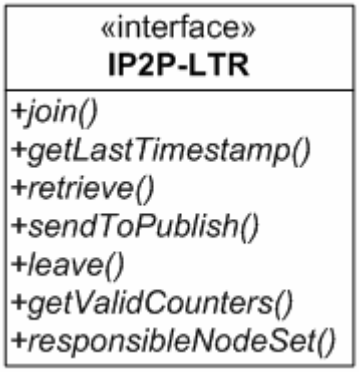

Figure 52. Interface P2P-LTR

2.3 Implémentation des composants

L'ensemble du code de cette réalisation repose sur le langage Java. Nous avons choisi d'utiliser ce langage pour plusieurs raisons. La première réside dans sa simplicité et sa portabilité. Etant donné la nature hétérogène des éléments constituant une application P2P, nous estimons qu'un langage portable est nécessaire pour garantir le fonctionnement de notre infrastructure quel que soit son contexte d'exécution. Ensuite, nous estimons que l'utilisation d'un seul langage de programmation pour la mise en œuvre des différents composants et de leur intégration est intéressante. Cette intégration permet d'éviter les traductions dues à des changements de technologies et simplifie le travail des développeurs de l'application XWiki P2P qui voudraient ajouter des fonctions à leurs composants.

La suite de cette partie décrit les implémentations des différents composants de l'application.

2.3.1 Implémentation de PatchManager

Le composant PatchManager gère le journal des opérations XWiki. Toute modification sur une page XWiki a pour effet de créer un patch. *PatchManager* peut générer un patch à partir de deux versions d'un document XWiki. Le composant *PatchManager* donne accès à ce journal et permet à une application distante d'exécuter un patch. Dans XWiki, un patch est un fichier XML décrivant les opérations atomiques effectuées sur chaque serveur XWiki. Le composant *PatchManager* les met à disposition par l'intermédiaire d'une API REST [FT02]. Les patches sont accompagnés d'une signature garantissant l'intégrité des données lors du transfert. Le Figure 53 montre un exemple de patch.

```
<patch id="IDXX"
Document id="myxwiki.mydocument.id"
Author="author id"
Date="DATE"
Signature="">
<operation type="insert" position="xxx">
<text> Bonjour ! </text>
</operation>
<operation type="delete" position="xxx">
<text> Paris </text>
</operation>
<operation type="set property">
<property name="author" value="John Doe" />
</operation>
</patch>
```

Figure 53. Exemple simple d'un patch généré sur un document XWiki

2.3.2 Implémentation de ReconcileManager

Le composant *ReconcileManager* assure l'intégration des patches reçus à travers les deux interfaces *PatchManager* et *P2P-LTR*. Il doit gérer les patches provenant de chaque interface. La Figure 54 décrit l'architecture interne du composant ReconcileManager en détaillant les interfaces.

ReconcileManager vérifie régulièrement la présence ou non de patches en attente de traitement. Lorsqu'un patch est disponible au traitement, il est récupéré (ce qui le supprime de la liste d'attente du composant *PatchManager*) et stocké dans le buffer. Une fois les transformations nécessaires exécutées, il est envoyé au composant *P2P-LTR* et supprimé du buffer. Il existe un buffer pour les patches en provenance de *PatchManager* et un autre pour les patches en provenance de *P2P-LTR*. L'utilisation des buffers est nécessaire pour permettre au composant *ReconcileManager* d'avoir accès à tous les patches en cours de traitement, durant l'application de l'algorithme de la réconciliation Soct4 [VCFS00] (Chapitre 2, Section 3.1.5). De plus, les deux buffers étant sérialisés, il est possible de récupérer les patches en cas de panne. En ce qui concerne le composant *P2P-LTR*, *ReconcileManager* maintient une estampille locale, représentant le dernier patch transféré au *PatchManager*. Cette estampille est gérée grâce au composant *ClockEngine*.

L'algorithme Soct4 employé est générique. L'algorithme est entièrement écrit sur la base d'une interface *TransformInterface*. Ce découpage logique permet d'injecter des fonctions de transformation spécifiques aux opérations XWiki définies dans les patches XWiki. Il est donc possible d'écrire les fonctions de transformation permettant de traiter les insertions et suppressions de blocs de texte dans les pages, mais aussi les fonctions de transformation permettant de traiter les méta-données des pages. Si de nouvelles

fonctionnalités sont ajoutées à XWiki, il sera possible d'ajouter les fonctions de transformation adéquates en conséquence.

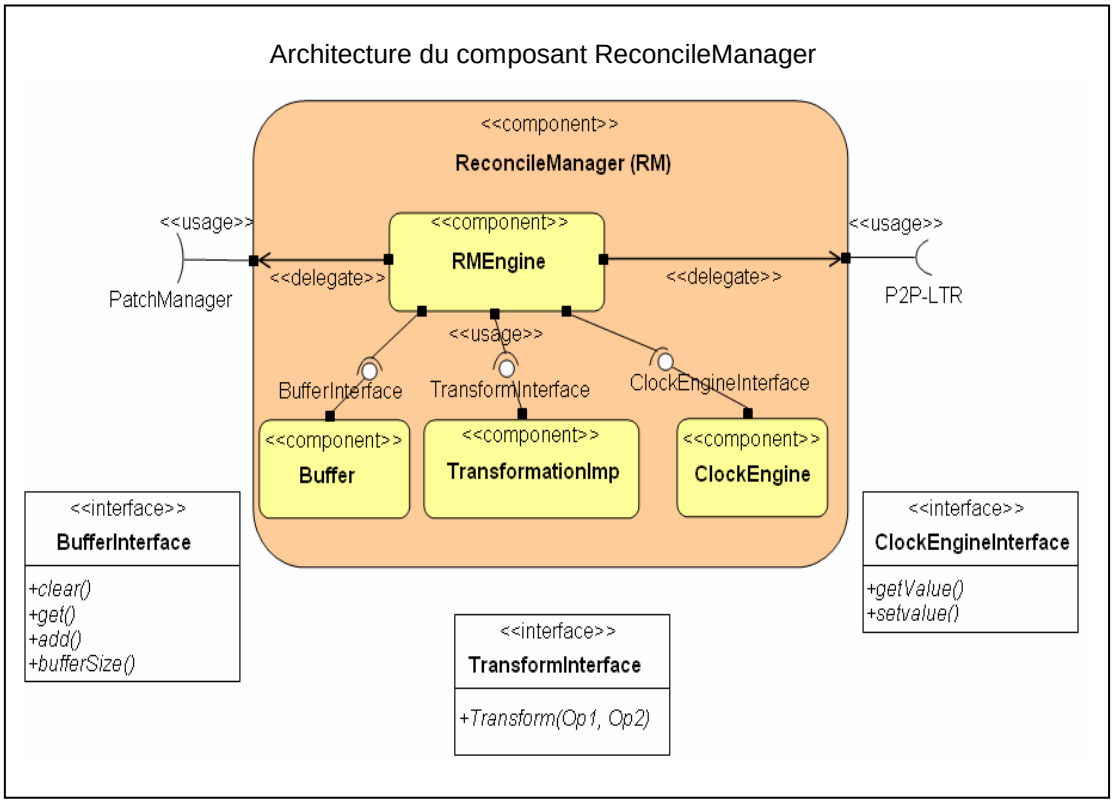

Figure 54. Architecture interne du composant ReconcileManager

2.3.3 Implémentation de P2P-LTR

Concernant l'implémentation du composant P2P-LTR, nous avons réalisé deux versions: une première version basée sur la DHT Chord [SMKK+01] et une deuxième version basée sur la DHT Pastry [RD01]. La Figure 55 décrit l'architecture interne du composant P2P-LTR.

P2P-LTR dispose d'un composant *Gestionnaire des évènements* (*GE*) qui s'abonne à l'événement stabilisation de la DHT à travers le composant *Ecouteur des évènements* (*EL : Event Listners*). Ainsi, à chaque fois qu'un événement de stabilisation est déclenché, *P2P-LTR* est notifié via le composant *EL* ce qui permet de maintenir automatiquement sa liste de clés dans la couche *P2P-LTR*. Le composant *GE* implémente les fonctions *LoseResponsability* et *changeResponsability* (voir chapitre 3, section 3.1) responsables de la détermination du nouveau Master-key et son successeur.

P2P-LTR dispose également de deux autres composants : *Gestionnaire des estampilles* et *Gestionnaire des patches* afin d'estampiller les patches selon un ordre total continu, avant de les stocker dans la DHT.

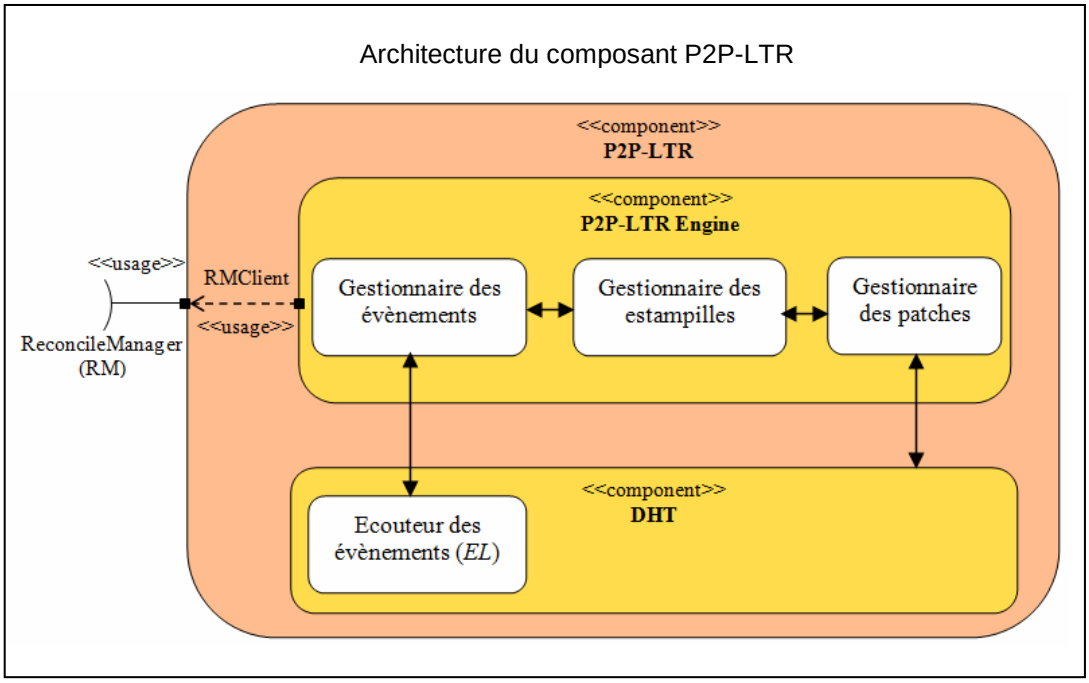

Figure 55. Architecture interne du composant P2P-LTR

2.3.3.1 Environnement de développement

L'ensemble du code de deux versions a été développé sous l'environnement de développement Eclipse, auquel peuvent être ajoutés les plugins suivants :

- Gestion de versions : SVN Subclipse 8
- Gestion de Tests : Junit 4.4
- Gestion de logs : la bibliothèque Log4J est utilisée pour logger le déroulement des opérations durant l'exécution.

Version basée sur Chord

L'implémentation de la première version de notre prototype est basée sur Open Chord [KL07] qui est une implémentation en Java de Chord [SMKK+01], proposée par des chercheurs de l'Université de Bamberg en Allemagne. Dans notre système, les pairs de DHT sont implémentés comme des objets Java. Chaque objet contient le code dont il a besoin afin d'implémenter le service P2P-LTR. La communication entre les objets P2P-

LTR est assurée via le protocole Java RMI [G01], permettant donc l'invocation des méthodes sur des objets distants.

Version basée sur Pastry

Ce qui change par rapport à la première version est l'utilisation d'une nouvelle implémentation de la DHT. Dans cette version de P2P-LTR, nous avons utilisé FreePastry [RD01] comme DHT à la place d'OpenChord. FreePastry est une implémentation en Java de Pastry, proposée par l'université Rice au Texas.

La topologie de Pastry est semblable à celle de Chord, donc en anneau. Les nœuds de Pastry forment un réseau de pairs décentralisé, auto-organisé et tolérant à la panne. Les identifiants des pairs sont choisis de telle façon que deux utilisateurs voisins au sens de la proximité de leurs *id* soient éloignés géographiquement. Une telle mesure a pour intérêt qu'une éventuelle panne du réseau dans une zone géographique entière ne pénalise pas Pastry, puisque la probabilité de perdre tous ses voisins pour un nœud dans ces circonstances se trouve être très faible.

Actuellement, FreePastry est de plus en plus utilisée pour le développement des applications à large échelle sur Internet. En effet, sa communauté est beaucoup plus active que celle d'Open Chord. Contrairement à Open Chord, une solution pour la traversée des NAT et de Firewall a été retenue à partir de la version 2.0 de FreePastry. Les développeurs ont inclus dans le protocole de connexion des méthodes pour la traversée de NAT et de Firewall. Ces méthodes se basent sur la technologie UPnP (Universal Plug and Play) [Upnp00].

Enfin l'un des points les plus positifs pour le développement d'une application basée sur FreePastry est la qualité de la documentation fournie par les développeurs, que ce soit au niveau des tutoriaux fournis sur le site de FreePastry [RD01] ou concernant la *javadoc* et les commentaires inclus dans le code source des classes. La prise en main de la librairie est donc rapide et sans grande difficulté en suivant les explications, ce qui évite de passer un long moment à chercher à comprendre comment fonctionne le code.

FreePastry semble donc plus approprié qu'Open Chord pour l'établissement de notre prototype P2P-LTR.

2.3.3.2 Description du prototype P2P-LTR

Notre approche de réplication optimiste présentée dans le chapitre précédent a été implémentée en premier lieu dans le prototype P2P-LTR, avant d'être intégré comme composant dans l'architecture XWiki P2P. Notre prototype a été principalement testé sur un réseau local en environnement Linux et Windows. P2P-LTR est distribué en licence libre (voir http://p2pltr.gforge.inria.fr/).

Dans la version actuelle de P2P-LTR, l'utilisateur est capable de manipuler le réseau DHT (exp. création de la DHT, ajout / suppression d'un nœud du réseau, …), insérer et retirer des données dans et depuis la DHT, afficher les données stockées dans chaque nœud ainsi que les listes des estampilles générées pour chaque clé (voir Figure 56). Dans la suite, nous allons présenter les fonctions du prototype P2P-LTR.

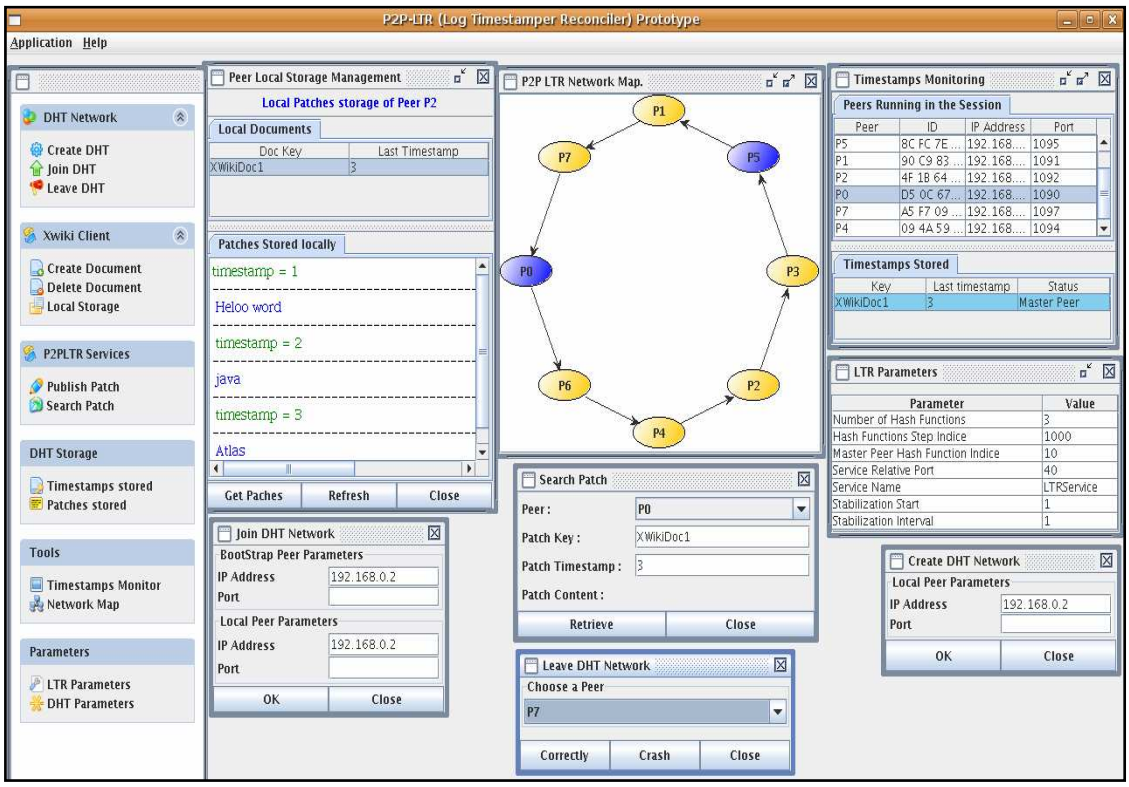

Figure 56. Interface principale du prototype P2P-LTR

Gestion de la DHT. La gestion de la DHT offre à l'utilisateur de l'application les fonctions suivantes :

- La création de la DHT: la construction de la DHT commence par la création du nœud "*Bootstrap*". Par la suite, tous les nouveaux arrivés contactent le nœud *Bootsrap* pour rejoindre le réseau.
- L'insertion d'un nouveau nœud dans le réseau DHT: pour qu'un nouveau nœud entre dans un réseau DHT il doit préalablement connaître un nœud existant dans ce réseau DHT. C'est ce nœud qui lui servira ainsi de "*Bootstrap*".
- La suppression d'un nœud existant de la DHT.

Gestion des données. La gestion des données offre à l'utilisateur de l'application les fonctions suivantes :

- La gestion des documents: création, modification et suppression des documents.
- La publication d'un patch généré sur un document.
- La récupération des patches manquants en cas des mises à jour concurrentes.

Consultation de l'état du système. La consultation des données doit permettre à l'utilisateur les fonctions suivantes :

- La consultation des informations liées à un document : les patches générés sur ce document, les patches appliqués dans ce document, etc.
- La consultation des informations liées au service d'estampillage : les responsables d'estampillage pour chaque document, liste des clés et estampilles stockées sur chaque responsable, etc.
- La consultation des informations liées aux Log-Peers : identification des nœuds responsable aux stockages des différentes patches et liste des patches stockés sur chaque nœuds, informations relatives à chaque patch, etc.

3 Déploiement

Le déploiement de notre architecture XWiki P2P peut être fait par les étapes suivantes (voir le schéma de déploiement dans Figure 57) :

1. Déployer une DHT+P2P-LTR.
2. Déployer des serveurs XWiki avec un service PatchManager (PM).
3. Déployer des composants ReconcileManager (RM).
4. Connecter les composants ReconcileManager avec les composants PatchManager.
5. Connecter les composants ReconcileManager avec les composants P2P-LTR.
6. Démarrer les composants ReconcileManager et P2P-LTR.

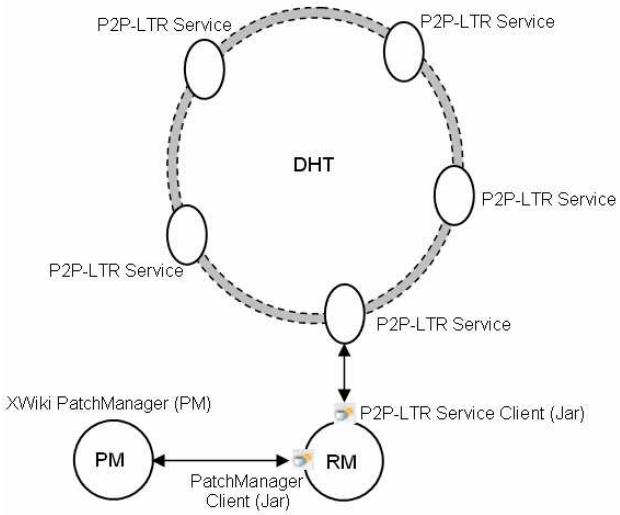

Figure 57. Schéma de déploiement

Les trois composants *PatchManager*, *ReconcileManager* et *P2P-LTR* peuvent être déployés sur des machines différentes.

3.1 Déploiement de la DHT + P2P-LTR

Le déploiement de la DHT doit être effectué par un utilisateur de la machine qui hébergera le nœud. A priori, le déploiement ne peut pas être piloté depuis l'application XWiki. En effet, il est nécessaire de pouvoir déployer et exécuter le code sur la machine, ce code ne s'exécutant pas forcément dans le serveur d'application hébergeant l'application XWiki.

Déploiement du premier nœud du réseau

1. Récupérer le code de P2P-LTR (*P2P-LTR-service.jar*)
2. Exécuter le code serveur : *java org.concerto.ltr.service.CreateNetwork Port.* Dans ce code *port* est celle qui est utilisée par le service *P2P-LTR*

Déploiement d'un nouveau nœud du réseau

1. Récupérer le code de P2P-LTR (*P2P-LTR-service.jar*)
2. Exécuter le code serveur :
   java org.concerto.ltr.service.JoinNetwork BootstrapIPAdress BootstrapPort Port
   - *BootstrapIPAdress:* adresse IP d'une machine hébergeant un nœud existant.
   - *BootstrapPort :* port utilisée par le service P2P-LTR du nœud existant.
   - *Port :* port local qui sera utilisé par le service P2P-LTR nouvellement démarré.

3.2 Déploiement des serveurs XWiki avec PatchManager

Le service PatchManager est intégré au code source XWiki en tant que "*plugin*". L'installation se déroule donc comme l'installation de tout *plugin* XWiki (cf. documentation d'installation d'un *plugin* XWiki).

Afin de se conformer à l'architecture à composants utilisée par XWiki, le composant *PatchManager* a été réorganisé en plusieurs sous composants:

- *patch-data*, consiste en un ensemble d'interfaces destinées à faciliter la manipulation des données. La nouvelle version de ces interfaces diffère de la précédente par la suppression de la méthode *apply(XWikiDocument doc)* et par l'ajout de nouvelles interfaces pour les différents types d'opérations. Deux implémentations de cette interface ont été développées :
  - *patch-internal*, utilisé par le moteur XWiki lors de la création des patches,
  - patch-pojo, destiné à la manipulation des patches par les clients. Dans les versions précédentes, la représentation XML du patch, utilisée pour la

communication, devait être traitée par les clients. Le composant patch-pojo dispense le client de cette phase de traitement en fournissant un accès direct aux données via les méthodes getDocument(), getOperations(), etc..

- *patchManager-client*, utilisé par les clients pour la communication avec XWiki. Il s'agit d'un nouveau composant destiné à faciliter la communication entre ReconcileManager et XWiki. Il utilise *patch-pojo* pour exposer des patchs à ReconcileManager.
- *patch-creator*, responsable de la création de patches lorsque des documents sont modifiés dans le wiki.
- *patchManager*, le plugin qui rassemble l'ensemble des composants.

Pour activer le composant PatchManager au sein d'un serveur XWiki, il faut :

- Placer tous les jars PatchManager dans le répertoire *WEB-INF/lib/*
- Déclarer le *servlet* PatchManager dans le fichier *WEB-INF/web.xml*

L'installeur XWiki Concerto 1.5 réalise automatiquement ce déploiement.

3.3 Déploiement des composants ReconcileManager

Le composant ReconcileManager doit être déployé sous la forme d'une application web (*webapp*) au sein d'un serveur d'application. Le déploiement d'un composant ReconcileManager se fait selon les étapes suivantes :

1. Récupérer le code de ReconcileManager (*ReconcileManager-service.war*)
2. Déployer l'application dans un serveur d'application (*e.x. tomcat*)
3. Configurer l'application à l'aide du fichier de propriétés *ReconcileManager.properties*

3.4 Connexion de ReconcileManager avec un composant PatchManager

L'application web de ReconcileManager contient une librairie (*PatchManager-client.jar*) permettant la communication entre le composant PatchManager et le composant ReconcileManager qui sont susceptibles de fonctionner sur deux serveurs différents. Pour réaliser la connexion, il suffit d'exécuter la méthode *connectToContentProvider(myXWiki).*

3.5 Connexion des ReconcileManager avec les composants P2P-LTR

L'application web de ReconcileManager contient une librairie (*P2P-LTRService-client.jar*) permettant la communication entre le composant P2P-LTR et le composant ReconcileManager qui sont susceptibles de fonctionner sur deux serveurs différents.
Pour réaliser la connexion, il suffit d'exécuter la méthode *connectToLTRNetwork(oneLTRPeer).*

3.6 Démarrage des composants ReconcileManager et P2P-LTR

Une fois, le composant ReconcileManager connecté à un composant PatchManager et à un composant P2P-LTR, l'algorithme de réconciliation peut s'exécuter. Pour démarrer celui-ci, il suffit d'exécuter la méthode *start()*.

4 Conclusion

Dans ce chapitre, nous avons présenté l'application XWiki P2P comme une solution de réplication optimiste qui consiste à répliquer les données d'un document XWiki dans un réseau P2P en utilisant le service P2P-LTR. Pour ordonner les patches publiés par les utilisateurs de système, P2P-LTR utilise un mécanisme d'estampillage fiable qui permet de garantir l'ordre total des estampilles.

CHAPITRE 6

VI Bilan et Perspectives

Ces dernières années ont connu un fort développement dans l'utilisation des applications collaboratives où un grand nombre d'utilisateurs doivent pouvoir travailler sur les mêmes données (documents, fichiers, etc.) en parallèle depuis leurs postes de travail. De nombreuses applications collaboratives sont maintenant très populaires comme les messageries instantanées, les Blogs, les gestionnaires de contenus, les calendriers partagés, etc.

L'édition collaborative s'est particulièrement illustrée par l'adoption par le grand public des éditeurs Wikis. Les Wikis ont rendu l'édition collaborative très populaire. Le succès par exemple de Wikipédia, l'encyclopédie libre et collaborative, est l'image la plus frappante. Ces outils sont basés sur l'architecture client/serveur traditionnelle du web. Cette architecture atteint aujourd'hui ses limites. Afin de permettre une croissance encore forte du point de vue taille et complexité des données gérées, ainsi que du nombre d'utilisateurs, l'idée de construire des Wikis reposant sur une architecture décentralisée de type Pair-à-Pair (P2P) est apparue.

Le terme P2P désigne une classe d'applications distribuées qui repose sur un modèle décentralisé où chaque élément présente simultanément le rôle de client et de serveur. Dans les réseaux P2P, les notions de serveur et de client disparaissent pour laisser place à celle de pair qui reflète particulièrement l'égalité entre les différents nœuds du réseau, tous les nœuds jouant le même rôle dans le réseau. Les réseaux de pairs sont des réseaux logiques et non physiques. Grâce à leur gestion décentralisée, à l'autonomie des nœuds et à la grande disponibilité des données qu'elles offrent, les architectures P2P sont bien adaptées à la gestion de très grandes masses de données, hétérogènes et réparties.

Cependant, une application Wiki P2P demande des capacités de réplication avec différents niveaux de granularité et un mode multi-maître où plusieurs répliques d'une même donnée peuvent être mises à jour par plusieurs pairs en parallèle. La réplication multi-maître offre de bons avantages de performance et de disponibilité. Cependant, les mises à jour du même document par différents pairs peuvent créer des divergences des copies.

De nombreuses solutions, basées sur un mécanisme de réplication optimiste, ont été proposées afin de gérer la cohérence des données partagées. Cependant, les solutions optimistes existantes sont peu applicables aux réseaux P2P puisqu'elles sont centralisées ou ne tiennent pas compte des comportements dynamiques des pairs.

C'est dans le cadre de cette problématique que notre travail de recherche s'est effectué. L'objectif principal de notre thèse est de concevoir, d'implanter et d'évaluer une solution de réplication optimiste avec fusion de données adaptée aux besoins d'un

éditeur collaboratif de type Wiki fonctionnant sur un réseau P2P et supportant des utilisateurs en situation de mobilité.

Dans la suite de ce chapitre, nous présentons d'abord un résumé de nos contributions. Ensuite, nous donnons un aperçu de nos futures directions de recherche pour améliorer ce travail de thèse.

1 Contributions

Nos contributions dans cette thèse peuvent être résumées en deux points décrits ci-après.

1.1 P2P-LTR

Le travail effectué dans cette thèse nous a permis de mettre en place une nouvelle approche appelée P2P-LTR (estampillage et journalisation P2P pour la réconciliation) [TDPV+08, TDPA+08] pour répondre aux challenges liés à la réplication et à la réconciliation de données dans un réseau P2P.

L'approche que nous avons proposée utilise une solution basée sur les transformées opérationnelles (OT) [MOSM+03], pour fusionner les données. Le choix d'un algorithme de fusion de données basé sur les transformées opérationnelles est motivé par la grande flexibilité pour réconcilier différents types de données, structurées, textuelles ou multimédias. Cependant, les algorithmes OT nécessitent un service de gestion d'estampilles continues. P2P-LTR offre un service de journalisation P2P et un service d'estampillage fiable et réparti fonctionnant sur un modèle de réseau à base de DHT (Table de hachage distribuée) [SMKK+01]. Dans notre approche, les mises à jour sont estampillées et stockées de façon P2P dans des Logs à forte disponibilité. Lors de la réconciliation, ces mises à jour sont récupérées selon un ordre total continu afin d'assurer la cohérence à terme. P2P-LTR traite les cas où les pairs peuvent rejoindre ou quitter le système pendant l'opération de mise à jour.

P2P-LTR rend donc possible l'utilisation d'un synchroniseur dans un contexte P2P pour gérer les estampilles et donc servir à la réconciliation des répliques divergentes. Nous avons validé notre solution par l'implémentation d'un prototype [TDPA+08] basé sur la DHT OpenChord [KL07] et FreePastry [RD01]. Ce prototype constitue une première validation de nos propositions, d'un point de vue faisabilité technique. Il permet la réalisation d'une validation expérimentale de notre approche, selon son utilisation et cela avant de l'intégrer comme un composant de réplication dans une application collaborative, notamment un Wiki. P2P-LTR est un logiciel libre, disponible à l'adresse suivante: http://p2pltr.gforce.inria.fr/.

En outre, nous avons prouvé théoriquement les propriétés de correction et de vivacité en présence de pannes [TAPV10]. Nous avons évalué les performances de notre solution à l'aide de la simulation en utilisant PeerSim [Pee10] comme simulateur. Les résultats montrent l'efficacité et le passage à l'échelle de notre solution.

En conclusion, P2P-LTR est un algorithme adapté à l'édition collaborative P2P.

1.2 XWiki: un Wiki P2P

Notre approche est validée aussi par la réalisation d'un prototype de Wiki P2P: XWiki [XWIKI07], un Wiki en logiciel libre pour les entreprises.

XWiki est un outil d'édition de texte en ligne. Plusieurs personnes peuvent éditer des données communes à travers une interface Web. Cependant, l'architecture actuelle de XWiki est basée sur une architecture classique client-serveur. Dans cette architecture, XWiki ne gère pas l'édition simultanée. Soit deux utilisateurs ne peuvent éditer la même donnée en même temps, soit une seule des éditions est prise en compte. Ce principe, qui est admissible pour un Wiki centralisé, devient inadapté dans un Wiki décentralisé avec des utilisateurs mobiles.

Cette thèse s'inscrit en partie dans le cadre du projet ANR XWiki Concerto [XWIKI07]. Le travail effectué dans ce projet nous a permis (en collaboration avec les différentes équipes du projet) de proposer une architecture XWiki P2P [XWIKI07] qui dépasse ces limites. L'architecture proposée se base essentiellement sur le composant P2P-LTR. Ce composant a pour fonction d'augmenter la disponibilité des données en répliquant les patches sur plusieurs nœuds du réseau. P2P-LTR sert également à estampiller les patches selon un ordre total continu avant de les stocker dans la DHT.

Notre architecture a été conçue en essayant de limiter l'impact sur l'application XWiki (*i.e.* éviter la modification du code de XWiki). Ainsi, le composant P2P-LTR est un composant indépendant interagissant avec le serveur XWiki à travers une interface Web.

2 Travaux futurs

Nos travaux futurs visent à enrichir notre solution P2P-LTR avec des fonctionnalités plus avancées. Nous présentons ci-après une liste des travaux que nous envisageons de réaliser.

2.1 Authentification et contrôle d'accès aux ressources

Dans un environnement P2P ouvert où les pairs sont considérés comme volatiles : ils rejoignent et quittent le réseau à tout moment, des pairs peuvent être malveillants et pouvant causer un routage incohérent, ou en tentant de corrompre le contenu partagé par des virus. Comme travail futur, il est nécessaire de penser à spécifier une API d'authentification et de contrôle d'accès aux ressources d'une application Wiki P2P. L'objectif est d'introduire au modèle P2P-LTR une couche permettant de gérer les politiques de contrôle d'accès indépendamment aux implémentations des applications collaboratives définies au dessus de notre modèle.

Le modèle OrBAC (Organization Based Access Control) [Orb10] peut être utilisé comme une solution intéressante aux problèmes d'authentification et de contrôle d'accès aux ressources dans les réseaux P2P. L'objectif d'OrBAC est d'introduire un niveau d'abstraction permettant d'exprimer la politique de contrôle d'accès indépendamment de son implémentation.

2.2 Un Wiki sémantique sur un réseau P2P

Les Wikis sémantiques [RSM09] sont une nouvelle génération d'outils d'édition collaborative. Ils permettent d'éditer des pages Wikis contenant des données sémantiques [MMDH+06]. En effet, un Wiki sémantique est un Wiki doté de fonctionnalités permettant de formaliser le sens des articles. Il permet de rajouter des informations sur les méta-données des articles et de caractériser leurs relations. Les utilisateurs peuvent alors collaborer non seulement pour écrire des pages Wikis mais aussi pour y mettre des connaissances sémantiques. L'aspect sémantique permet de rendre le contenu d'un Wiki "compréhensible" par une machine. Généralement, ceci est réalisé par l'annotation des liens entre les pages.

Nous avons l'intention de combiner les avantages des systèmes P2P et du Web sémantique afin de réaliser un Wiki sémantique sur un réseau P2P. L'intégration des technologies du Web sémantique dans un Wiki améliore la structuration des pages Wiki, la navigation entre ses pages et la recherche d'information. L'architecture P2P permet un passage à l'échelle, une amélioration des performances, une résistance aux pannes et un travail en mode connecté/déconnecté.

2.3 Amélioration de XWiki

La mise en œuvre du prototype d'XWiki Concerto [XWIKI07] a permis de valider fonctionnellement les cas d'utilisation de rédaction collaborative sur un réseau Pair-à-Pair. Les prochaines étapes consisteront à améliorer les performances du système, à étendre la rédaction collaborative aux méta-données liées aux pages XWiki, notamment les fichiers joints, et à sécuriser le système. En outre, il est important d'étendre le mécanisme de propagation de P2P-LTR aux objets et fichiers attachés aux documents du XWiki. Enfin, l'approche XWiki P2P proposée devra être comparée davantage avec d'autres solutions de Wiki P2P, notamment UniWiki [OMDM09], DistriWiki [M07], DTWiki [DB08] ou Piki [MLS08].

2.4 Le Cloud : compromis entre cohérence et disponibilité

Le principe du *cloud* [BYV08] consiste à l'externalisation, la virtualisation et la mutualisation des ressources qui favorisent la montée en charge et la haute disponibilité de données. Dans ce concept, il s'agit non seulement de virtualiser un logiciel, mais également d'externaliser le stockage, la CPU et la bande passante. Les logiciels et les données sont stockés dans des serveurs n'appartenant plus à l'éditeur, mais dans des vastes grilles de serveurs ou des *data centers* gigantesques tels que ceux d'Amazon ou de Google. L'accès à ces logiciels se fait la plupart du temps par un navigateur Web.

D'après Brewer [Bre00], les applications web ne peuvent pas à la fois assurer une cohérence forte des données (*strong consistency*), une accessibilité quasi immédiate et un déploiement à grande échelle. Parmi ces trois caractéristiques, le *cloud computing* met en avant l'accessibilité quasi immédiate et le déploiement à grande échelle. Par conséquent, il n'est pas possible de garantir une cohérence forte, d'où la mise en place de modèles de cohérence plus souples, notamment la *cohérence à terme*. Parmi les

solutions dédiées aux *clouds* et qui utilisent une cohérence à terme, nous pouvons citer Cassandra [LM09] et Amazon S3 [BFG+08].

Comme perspective, nous pensons étudier les solutions de cohérence à terme dédiées aux *clouds* et les comparer à notre solution. Nous pensons également tester et analyser le comportement de notre P2P-LTR dans un contexte de *cloud computing*.

Cependant, il est important de noter que le problème fondamental reste la confidentialité et la sécurisation de l'accès aux applications dans les *clouds*. En effet, dans le cas du *cloud*, une entreprise devrait connecter ses postes à Internet et ainsi les exposer aux risques d'attaques ou de pertes de données. Comme énoncé préalablement à la section 2.1, l'un des travaux futurs serait d'introduire à notre modèle P2P-LTR une couche permettant de gérer des politiques de contrôle d'accès. Ceci rentrera de ce fait dans les principaux challenges à relever, en vue d'une utilisation future de notre modèle P2P-LTR dans le *cloud*.

ANNEXES

A. *Tutorial du composant P2P-LTR*

A.1. Création d'un réseau DHT

```
import fr.inria.atlas.p2pltr.base.Peer;
import fr.inria.atlas.p2pltr.base.RemotePeer;
import fr.inria.atlas.p2pltr.service.P2pltrService;

public class LtrNewDHT {

	public static void main(String[] args) {
		P2pltrService ltr=null;
		try{
		 ltr=new P2pltrService();
		}
		catch (Exception e)
		{}
		InetAddress addr=null;
		try {
			addr = InetAddress.getLocalHost();

		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		long port=1090;
		Peer p=new RemotePeer(addr,port);
		 ltr.dhtCreate(p);
}
```

A.2. Entrée d'un nouveau Pair dans le réseau DHT

Pour qu'un nouveau pair entre dans un nœud DHT il doit préalablement connaître un pair existant dans ce réseau DHT. C'est ce pair qui lui servira ainsi de Bootstrap.

```
import fr.inria.atlas.p2pltr.base.Peer;
import fr.inria.atlas.p2pltr.base.RemotePeer;
import fr.inria.atlas.p2pltr.service.P2pltrService;
public class LtrPeer {

	public static void main(String[ ] args) {
		P2pltrService ltr=null;
		if (args.length <1){
			System.out.println("Please you don't give the argument!!!");
			System.exit(0);
		}
		try{
		 kts=new P2pltrService();
		}
		catch (Exception e)
		{}
		InetAddress addr=null;
		try {
			addr = InetAddress.getLocalHost();
			//System.out.print(addr);

		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		long port=Long.parseLong(args[0]);
		//System.out.print(port);
		Peer localPeer=new RemotePeer(addr,port);
		Peer bootstrapPeer=new RemotePeer(addr,1090);
		ltr.dhtJoin(localPeer, bootstrapPeer);
}
```

A.3. Publication de patches dans la DHT

La publication d'un patch dans le DHT se fait par la méthode *sendToPublish()* de l'objet P2pltrService. Cette méthode possède 3 paramètres : la clé du patch à publier qui est de type String, le patch à publier qui doit être de type Serializable et enfin le timestamp local associé a la clé de ce patch.

Cette méthode retourne 0 si la publication n'a pas eu lieu (a cause d'une publication concurrente), -1 s'il y a des patches manquantes a rapatrier ou un nombre >=1 qui est le timestamp délivré pour publier ce patch.

Récupération de patches dans la DHT

La récupération d'un patch dans le DHT se fait par la méthode *retrieve()* de l'objet *P2pltrService*. Cette méthode possède 2 paramètres : la clé du patch recherche et le timestamp associe à ce patch. Cette méthode retourne un objet de type Serializable qui est le patch recherché.

Exemple de publication et de Récupération dans un DHT.

```
import fr.inria.atlas.p2pltr.base.Peer;
import fr.inria.atlas.p2pltr.base.RemotePeer;
import fr.inria.atlas.p2pltr.service.P2pltrService;

public class LtrPeer {

	public static void main(String[] args) {
		P2pltrService ltr=null;
		if (args.length <1){
			System.out.println("Please you don't give the argument!!!");
			System.exit(0);
		}
		try{
		 kts=new P2pltrService();
		}
		catch (Exception e)
		{}
		InetAddress addr=null;
		try {
			addr = InetAddress.getLocalHost();
			//System.out.print(addr);

		} catch (UnknownHostException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		long port=Long.parseLong(args[0]);
		//System.out.print(port);
```

```
        Peer localPeer=new RemotePeer(addr,port);
        Peer bootstrapPeer=new RemotePeer(addr,1090);
        ltr.dhtJoin(localPeer, bootstrapPeer);
        String key='a'
        String value= "toto";
        ltr.sendToPublish(key,value,0);
        System.out.println(ltr.retrieve(key,1));
}
```

A.4. Les fichiers de configurations

Le composant LTR utilise principalement deux fichiers de configurations :

- *chord.properties* : fichier de configuration d'Open Chord.
- *p2pltr.properties* : fichier de configuration du composant LTR

B. So6: informations additionnelles

L'algorithme d'intégration utilisé dans So6 repose sur l'algorithme SOCT4. Les fonctions de transformation doivent donc satisfaire uniquement la condition C1. Toutes les fonctions de transformation de So6 sont présentées en détails dans [Ophd05]. Elles ont été conçues et validées selon l'approche décrite dans [IMOR02, IMOU03, IROM05].

B.1. Fonctions de transformations utilisées dans So6

Dans cette section, nous nous intéressons aux fonctions de transformation nécessaires pour réconcilier un document textuel. Dans So6 un document est modélisé sous la forme d'une liste de blocs de lignes de texte.

Trois opérations sont utilisées pour modifier le document :

- *AddBlock(p,v)* insère le bloc de lignes *v* à la ligne *p* du document ;
- *DelBlock(p,ov)* détruit le bloc de lignes *ov* débutant à la ligne *p* du document.
- *Id()* correspondant à la fonction identité. Cette opération n'a aucun effet sur l'´etat sur lequel elle est exécutée. Elle peut être obtenue par transformation de deux opérations identiques, comme par exemple deux insertions à la même ligne du même contenu.

Par exemple, dans le cas de deux insertions concurrentes la fonction de transformation correspondante est la suivante :

$$
\begin{array}{l}
T(AddBlock(p_1,v_1),\ AddBlock(p_2,v_2)) = \\
\quad \textbf{if}\ \ (p_1 < p_2)\ \textbf{then} \\
\qquad \textbf{return}\ AddBlock(p_1,v_1) \\
\quad \textbf{else if}\ \ (p_2 < p_1)\ \textbf{then} \\
\qquad \textbf{return}\ AddBlock(p_1 + sizeof(v_2),v_1) \\
\quad \textbf{else if}\ \ (v_1 = v_2)\ \textbf{then} \\
\qquad \textbf{return}\ Id() \\
\quad \textbf{else if}\ \ (code(v_1) < code(v_2))\ \textbf{then} \\
\qquad \textbf{return}\ AddBlock(p_1,v_1) \\
\quad \textbf{else} \\
\qquad \textbf{return}\ AddBlock(p_1 + sizeof(v_2),v_1) \\
\quad \textbf{endif};
\end{array}
$$

Figure 58. Exemple de fonction de transformation dans So6

Au cours de la transformation, trois cas peuvent se produire :

1. soit les deux insertions ne se font pas à la même ligne. Dans ce cas, l'une des deux insertions est décalée afin de prendre en compte l'autre insertion qui la précède selon l'ordre des numéros de ligne. L'utilisation de la fonction *sizeof (v)* a pour rôle de calculer le nombre de ligne du bloc *v* afin de s'en servir comme pas de décalage.
2. soit les deux insertions se font à la même ligne et insèrent le même bloc de lignes. Dans ce cas, une des deux opérations d'insertion est inhibée. Cette opération est transformée en une opération identité.
3. soit les deux insertions se font à la même ligne, mais les deux blocs sont différents. Dans ce cas, une fonction *code(v)* est utilisée pour calculer une valeur entière à partir du contenu du bloc *v*. Le bloc dont la valeur retournée par la fonction *code(v)* est la moins élevée est placé devant l'autre bloc.

Dans [Ophd05], l'auteur a décrit les fonctions de transformation permettant de réconcilier un système de fichiers partagé et les fonctions de transformation nécessaires à la réconciliation d'un arbre ordonné. Un tel objet peut être utilisé afin de modéliser un document XML.

C. Fonctions de transformations utilisées dans P2P-LTR

Cette partie décrit les fonctions de transformations utilisées par le composant de réconciliation (So6ClientService) dans le cas de P2P-LTR. Ces transformations sont appliquées aux opérations distantes et aux opérations locales non diffusées lors de l'intégration d'un patch distant dans une copie locale d'un document XWiki. Elle réalise donc concrètement la fusion d'opérations concurrentes.

Lorsqu'une opération distante est récupérée depuis la DHT par le service LTR, cette opération est transformée pour tenir compte des opérations locales non encore diffusées (et donc concurrentes).

C.1. Opérations modifiants le contenu d'un document XWiki

Un document XWiki a la structure d'un simple document Wiki. Le contenu d'une page Wiki est modélisé sous forme d'une chaîne de caractères classique. Comme So6 les opérations considérées sont l'insertion (*insert*) et la suppression (*delete*).

C.1.1. Les opérations

Afin d'être adapté à la structure d'un document XWiki, le bloc de lignes est remplacé par une chaîne de caractères et la position d'une ligne est remplacée par la position d'un caractère. Les opérations sont décrites comme suit :

- *insert(String text, int p)*: insère la chaîne de caractère *text* à la position *p* du document.
- *delete(String text, int position) :* détruit la chaîne de caractères *text* débutant à la position *p* du document.
- *nop :* correspond à la fonction identité.

C.1.2. Les Transforamtions

Les fonctions de transformations sont décrites comme suit :

- *T(insert(String text1, int position1), insert(String text2, int position2))* : cas de deux insertions concurrentes.
- *T(insert(String text1, int position1), delete(String text2, int position2)) :* cas d'une insertion et d'une suppression concurrente.
- *T(delete(String text1, int position1), insert(String text2, int position2))* : cas d'une suppression et d'une insertion concurrente.
- *T(delete(String text1, int position1), delete(String text2, int position2))* : cas de deux suppressions concurrente.

C.1.3. Exemple d'une fonction de transformation dans P2P-LTR

Par exemple, dans le cas de deux insertions concurrentes la fonction de transformation correspondante est décrite par la Figure 59.

Comme S06, la fonction *length( )* est utilisée pour calculer la taille de la chaîne de caractères afin de s'en servir comme pas de décalage. La fonction *hashCode( )*, c'est pour calculer une valeur à partir du contenu du texte. La chaîne dont la valeur retournée par la fonction *hashCode( )* est la moins élevée est placé devant l'autre chaîne

```
T(insert(String text1, int position1), insert(String text2, int
position2)) {
        if (position1 < position2) {
                return insert(text1, position1)
        } else if (position1 > position 2) {
                return insert(text1, position1+text2.length)
        } else {
        // position1 == position2
        if (text1.equals(text2)) {
                return nop
        } elseif (text1.hashCode() < text2.hashCode()) {
                return insert(text1, position1)
        } else {
                return insert(text1, position1+text2.length)
                }
        }
}
```

Figure 59. Exemple de fonction de transformation dans P2P-LTR

.

BIBLIOGRAPHIE

[ACMR02] S. Ajmani, D. E. Clarke, C. Moh, S. Richman. ConChord : Cooperative SDSI certificate storage and name resolution. *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 141–154, 2002.

[APV07] R. Akbarinia, E. Pacitti, and P. Valduriez. Data currency in replicated dhts. *In Proceedings of the ACM SIGMOD Int. Conf. on Management of Data*, 211-222, 2007.

[BBMS+08] L. Benmouffok, J. Busca, J. Marquès, M. Shapiro, P. Sutra, G. Tsoukalas. Telex: Principled System Support for Write-Sharing in Collaborative Applications. *Tech report INRIA N°RR6546*, 2008.

[Ber90] B. Berliner. CVS II: Parallelizing software development. *In Proceedings of the USENIX Winter Technical Conference*, 341–352, 1990.

[BFG+08] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. *In Proceedings of the ACM SIGMOD international conference on Management of data*, 251-264, 2008.

[BHG87] P. A. Bernstein, V. Hadzilacos, N. Goodman. Concurrency Control and Recovery in Database Systems. *Addison-Wesley*, ISBN 0-201-10715-5, 1987.

[Bre00] Brewer, Eric A. Towards robust distributed systems (abstract). *In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. 2000.

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

[CF07] M. Cart, J. Ferrie. Asynchronous reconciliation based on operational transformation for P2P collaborative environments. *In Proceedings of the Int. Conf. on Collaborative Computing: Networking, Applications and Worksharing*, 127-138, 2007.

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

[CMM02] R. Cox, A. Muthitacharoen, R. Morris. Serving DNS using Chord. *In Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, 2429 in LNCS, 155–165, 2002.

[CRRLS+05] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, J.M. Hellerstein. A case study in building layered DHT applications. *In Proceedings of the ACM SIGCOMM Int. Conf. on Management of Data*, 97-108, 2005.

[CSWH00] I. Clarke, O. Sandberg, B. Wiley, T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. *Workshop on Design Issues in Anonymity and Unobservability*, 46-66, 2000.

[CTVK+07] A. Chazapis, G. Tsoukalas, G. Verigakis, K. Kourtis, A. Sotiropoulos, N. Koziris. Global-scale peer-to-peer file services with dfs. *In Proceedings of the Int. Conf. on Grid Computing*, 251–258, 2007.

[DB08] B. Du, E. A. Brewer. Dtwiki: a disconnection and intermittency tolerant wiki. *In Proceeding of the 17th international conference on World Wide Web*, 945-952, 2008.

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

[DHA03] A. Datta, M. Hauswirth, K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. *IEEE Int. Conf. on Distributed Computing Systems*, 76-85, 2003.

[DKKM+01] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, I. Stoica. Wide-area cooperative storage with CFS. *In Proceedings of the 18th ACM Symposium on Operating Systems Principle*, 202–215, 2001.

[DM03] D. Doval and D. O'Mahony. Overlays networks: A scalable alternative for P2P. *IEEE Internet Computing*, 7(4): 77–82, 2003.

[Dphd05] G. Doyen. Supervision des réseaux et services pair-à-pair. *PhD thesis, Université Henri Poincaré - Nancy I*, 2005.

[EG89] C. Ellis, S. Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Int. Conf. on Management of Data*, 18, 399–407, 1989.

[FG03] P. Fraigniaud, P. Gauron. An overview of the content-addressable network D2B. *In Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, 151–151, 2003.

[FHKS+01] P. Francis, M. Handley, R. Karp, S. Shenker, S. Ratnasamy. A Scalable Content-Addressable Network. *ACM SIGCOMM Int. Conf. on Management of Data*, 161-172, 2001.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *In Journal of the Association for Computing Machinery (ACM)*, 32(2):374-382, 1985.

[FT02] R. T. Fielding, R. N. Taylor. Principled design of the modern web architecture. *ACM Transaction on Internet Technology*, 2(2):115-150, 2002.

[G01] W. Grosso. Java RMI. O'Reilly Media, pages 572, 2001.

[G99] F. Gârtner. Fundamentals of Fault-tolerant distributed Computing in Asynchronous environments. *ACM Computing Surveys*, 31(1),1999.

[GGGR+03] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, I. Stoica. The impact of DHT routing geometry on resilience and proximity. *In Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, 381–394, 2003.

[GHKK03] Th., Guerraoui, R., Handurukande, S.B., Kouznetsov, P., Kermarrec, A.M.: Lightweight probabilistic broadcast. *ACM Transaction on Computer System.* 21(4), 341–374, 2003.

[GW09] Google Wave. http://wave.google.com/, 2009.

[HSG08] T. HUU, M. SEGARRA, J. GILLIOT. Un système adaptatif de placement de données. *CFSE: conférence française sur les systèmes d'exploitation*, 2008.

[IMOR02] A. Imine, P. Molli, G. Oster, M. Rusinowitch. Development of transformation functions assisted by a theorem prover. *Fourth International Collaborative Editing Workshop*, 2002.

[IMOR03] A. Imine, P. Molli, G. Oster, M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. *Eighth European Conference of Computer-Supported Cooperative Work*, 277-293, 2003.

[IROM05] A. Imine, M. Rusinowitch, G. Oster, P. Molli. An algebraic framework for designing operational transformation algorithms. *Theoretical Computer Science: selected papers of the tenth International Conference on Algebraic Methodology of Software Technology*, 2005.

[Jxta10] JXTA. http://www.jxta.org/, 2010.

[KAZA09] KAZA. http://www.kazaa.com, 2009.

[KBCC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski et al. OceanStore: An Architecture for Global-Scale Persistent Storage. *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 190-201, 2000.

[KL07] S. Kaffile, K. Loesing. Open Chord User's Manual, 2007.

[KRSD01] A. Kermarrec, Antony I. T. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. *In Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, 210–218, 2001.

[Lamport89] L. Lamport, The part-time parliament, *Tech. Report 49, Systems Research Center, Digital Equipment Corp*, 1989.

[LL04a] D. Li, R. Li. Ensuring content and intention consistency in real-time group editors. *In Proceedings of the twenty fourth International Conference on Distributed Computing Systems*, 748–755, 2004.

[LL04b] D. Li, R. Li. Preserving operation effects relation in group editors. *In Proceedings of the ACM conference on Computer Supported Cooperative Work*, 457–466, 2004.

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

[LM96] D. Lee, M. Yannakakis. Principles and methods of testing finite state machines: A survey. *In Proceedings of the IEEE*, 84(8):1090-1126, 1996.

[M07] J. C. Morris. Distriwiki: A distributed peer-to-peer wiki network. *In Proceedings of the International Symposium on Wikis - WikiSym*, 69-74, 2007.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. *In Proceedings of the International Workshop on Parallel and Distributed Algorithms*, 215–226, 1989.

[MAPV06] V. Martins, R. Akbarinia, E. Pacitti, P. Valduriez. Reconciliation in the APPA P2P System. *IEEE Int. Conf. on Parallel and Distributed Systems*, 401-410, 2006.

[MLS08] P. Mukherjee, C. Leng, A. Schurr. Piki - a peer-to-peer based wiki engine. *In Proceedings of the IEEE International Conference on Peer-to-Peer Computing*, 185-186, 2008.

[MMDH+06] VÖLKEL M, KRÖTZSCH M, VRANDECIC D, HALLER H and STUDER R. Semantic Wikipedia. *In Proceedings of the 15th international conference on World Wide Web*, 585-594, 2006.

[MMGC02] A. Muthitacharoen, R. Morris, T. M. Gil, B. Chen. Ivy: A read/write peer-to-peer file system. *5th Symposium on Operating Systems Design and Implementation*, 36, 2002.

[MOSM+03] P. Molli and. Al. Using the transformational approach to build a safe and generic data synchronizer. *ACM Press, editor, ACM SIGGROUP Conference on Supporting Group Work*, 212–220, 2003.

[MP06] V. Martins and Esther Pacitti. Dynamic and Distributed Reconciliation in P2P-DHT Networks. *European Conf. on Parallel Computing (Euro-Par)*, 337-349, 2006.

[MPV06] V. Martins, E. Pacitti, and P. Valduriez. Survey of data replication in p2p systems. *Technical report, Institut National de Recherche en Informatique et en Automatique*, 2006.

[MPV06a] V. Martins, E. Pacitti, P. Valduriez. A Dynamic Distributed Algorithm for Semantic Reconciliation. *Distributed Data & Structures 6 (WDAS). Records of the 7th Int. Meeting*, 2006.

[NCDL95] D. Nichols, P. Curtis, M. Dixon, J. Lamping: High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System. *ACM Symposium on User Interface Software and Technology*, 111-120, 1995.

[OMDM09] G. Oster, P. Molli, S. Dumitriu, R. Mondéjar. UniWiki: A Reliable and Scalable Peer-to-Peer System for Distributing Wiki Applications. *RR6848*, 2009.

[OMSI04] G. Oster, P. Molli, H. Skaf-Molli and A. Imine. Un modèle sûr et générique pour la synchronisation de données divergentes. *Premières Journées Francophones : Mobilité et Ubiquité*, p9, 2004.

[Ophd05] G. Oster. Réplication Optimiste et Cohérence des Données dans les Environnements Collaboratifs Répartis. *PhD thesis, Université Henri Poincaré - Nancy I*, 2005.

[Orb10] OrBAC : http://orbac.org/, 2010

[OUMS+05] G. Oster, P. Urso, P. Molli, H. Skaf-Molli and A. Imine. Optimistic Replication for Massive Collaborative Editing. *Technical report RR-5719, INRIA*, 2005.

[PD02] E. Pacitti and O. Dedieu. Algorithms for optimistic replication on the web. *Journal of the Brazilian Computing Society*, 8(2):179–183, 2002.

[Pee10] Peersim : http ://peersim.sourceforge.net/, 2010.

[PRR97] C. G. Plaxton, R. Rajaraman, Richa A. W. Accessing nearby copies of replicated objects in a distributed environment. *In Proceedings the 9th annual ACM Symposium on Parallel Algorithms and Architectures*, 311–320, 1997.

[PSM03] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. *ACM Press, editor, On The Move to Meaningful Internet Systems*, LNCS 2888, 38–55. 2003.

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

[RD01] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *In Proceedings of the ACM Symp. On Operating Systems Principles*, 188-201, 2001.

[RKCD01] A. Rowstron, A. M. Kermarrec, M. Castro, P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. *In Proceedings of the 3rd international workshop on Networked Group Communication*, LNCS 2233, 30-43, 2001.

[RNRG96] M. Ressel, D. Nitsche-Ruhland, R. Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *In Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'96*, 288–297, 1996.

[RSM09] C. Rahhal, H. Skaf-Molli, P. Molli. SWooki, un wiki sémantique sur réseau pair-à-pair. *Ingénierie des Systèmes d'Information*, 14(1): 117-140, 2009.

[SBK04] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. *Int. Conf. on Principles of Dist. Sys. (OPODIS), number 3544 in Lecture Notes in Comp. Sc.*, 331-345, 2004.

[SCF97] M. Suleiman, M. Cart, J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. *In Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge*, 435–445, 1997.

[S01] C. Shirky. Peer-to-peer: Harnessing the Power of Disruptive Technologies. *Chapter Listening to Napster, O'Reilly & Associates, Inc*, 21–37, 2001.

[SCF98] M. Suleiman, M. Cart, J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. *In Proceedings of the fourteenth International Conference on Data Engineering*, 36–45, 1998.

[SE98] C. Sun, C. Ellis. Operational transformation in realtime group editors : Issues, algorithms, and achievements. *In Proceedings of the ACM Conference on Computer Supported Cooperative Work*, 59–68, 1998.

[SGG02] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. *In Proceedings of Multimedia Computing and Networking*, 156-170, 2002.

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

[SP07] M. Shapiro and N. Preguiça. Designing a commutative replicated data type. *Rapport de recherche INRIA RR-6320, INRIA*, 2007.

[Sphd10] I. SARR. Routage des Transactions dans les Bases de Données à Large Echelle. *PhD thesis, Université Pierre et Marie Curie (Paris VI)*, 2010.

[SPR04] M. Shapiro, N. Preguiça, and J. Rufis. Mobile data sharing using a generic constraint-oriented reconciler. *In Proceedings of 5th IEEE International Conference on Mobile Data Management*, 146–151, 2004.

[SRHS10] F. Schintke, A. Reinefeld, S. Haridi, T. Schütt. Enhanced Paxos Commit for Transactions on DHTs. *In Proceedings of the Int. Conf. on Cluster, Cloud and Grid Computing*, 448-454, 2010.

[SS05] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 42-81, 2005.

[SZJY97] C. Sun, Y. Zhang, X. Jia, Y. Yang. A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. *In Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, 425–434, 1997.

[TAPV10] M. Tlili, R. Akbarinia, E. Pacitti, P. Valduriez. Scalable P2P Reconciliation Infrastructure for Collaborative Text Editing. *In Proceedings of the Int. Conf. on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, IEEE, 155-164, 2010.

[TDPV+08] M. Tlili, W. K. Dedzoe, E. Pacitti, P. Valduriez, R. Akbarinia, L. Dubost, S. Dumitriu, S. Laurière, G. Canals, P. Molli, J. Maire. Estampillage et Journalisation P2P pour XWiki. *Int. Conf. on New Technologies of Distributed Systems (NOTERE), ACM*, 197-200, 2008. (Demo)

[TDPA+08] M. Tlili, W. K. Dedzoe, E. Pacitti, R. Akbarinia, P. Valduriez, P. Molli, G. Canals, S. Laurière. P2P Logging and Timestamping for Reconciliation. *Int. Conf. on Very Large Databases (VLDB)*, 1420-1423, 2008. (Demo)

[TTPD+95] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *In Proceedings of the fifteenth ACM symposium on Operating systems principles*, 172–182, 1995.

[Upnp00] UPnP Forum Technical Committee. Multicast and unicast UDP HTTP messages. *http://www.upnp.org/download/draft-goland-http-udp-04.txt*, 2000.

[VCFS00] N. Vidot, M. Cart, J. Ferrié, M. Suleiman. *Copies convergence in a distributed real-time collaborative environment*. *In Proceedings of the ACM conference on Computer supported cooperative work*, 171–180, 2000.

[Vid02] N. Vidot. Convergence des Copies dans un Environnements Collaboratifs Répartis. *Thèse de Doctorat, Université de Montpellier II*, 2002.

[VP04] P. Valduriez and E. Pacitti. Data management in large-scale p2p systems. *Springer Int. Conf. on High Performance Computing for Computational Science (VecPar)*, 109–122, 2004.

[Wphd10] S. Weiss. Edition collaborative massive sur réseaux Pair-à-Pair. *PhD thesis, Université Henri Poincaré - Nancy I*, 2010.

[WUM07] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. *Lecture Notes In Computer Science*, 4831(1005):503-512, 2007.

[WUM09] S. Weiss, P. Urso, and P. Molli. Logoot: a Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. *In Proceedings of Int. Conf. on Distributed Computing Systems*, 404-412, 2009.

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

[ZJK01] B.Y. Zhao, A.D. Joseph, J.D. Kubiatowicz. Tapestry: a infrastructure for fault-tolerant wide-area location and routing. *Technical report, University of California, Berkeley*, 2001.

[XAP04] D. Xavier, S. André, U. Péter. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4)372-421, 2004.

[XWIKI07] XWiki Concerto: http://concerto.xwiki.com, 2007