Lh*rs p2p : une nouvelle structure de données distribuée et scalable pour les environnements Pair à Pair

Hanafi Yakouben

École Doctorale EDDIMO
LAMSADE

THÈSE

LH $*_{RS}^{P2P}$: Une Nouvelle Structure de Données Distribuée et Scalable pour les Environnements Pair à Pair

Pour l'obtention du titre de

Docteur en Informatique

(Arrêté du 25 avril 2002)

Présentée et soutenue le 14 Mai 2013

YAKOUBEN Hanafi

Composition du Jury

Directeur de Thèse :	**LITWIN Witold** Professeur à l'université Paris-Dauphine
Présidente du Jury :	**GRIGORI Daniela** Professeur à l'Université Paris-Dauphine
Rapporteurs :	**LONG Darrell Don Earl** Professeur à l'Université de Californie à Santa Cruz, USA
	DU MOUZA Cédric Maître de Conférences Habilité au Conservatoire National des Arts et Métiers
Examinateurs :	**PÂRIS Jehan-François** Professeur à l'Université de Houston, USA
	RIGAUX Philippe Professeur au Conservatoire National des Arts et Métiers

Dédicaces

A la mémoire de ma mère.

Remerciements

Mes premiers remerciements vont à mon directeur de thèse, Monsieur Litwin, professeur à l'Université Paris-Dauphine. Sa rigueur et la pertinence de ses jugements ont été très constructives et m'ont permis d'achever cette thèse. Merci pour tous vos conseils et de m'avoir donné la possibilité d'préparer ma thèse. Je tiens à vous exprimer toute ma profonde gratitude.

Je remercie chaleureusement Monsieur Long, Professeur à l'Université de Californie à Santa Cruz et Monsieur Pâris, Professeur à l'Université de Houston pour le temps qu'ils m'ont consacré durant leur séjour à Paris, pour tous leurs conseils, corrections et commentaires. Un très grand merci de m'avoir fait l'honneur d'accepter de participer au jury.

Je remercie vivement Monsieur Rigaux, Professeur au CNAM et Monsieur Du Mouza, Maître de Conférences Habilité au CNAM de m'avoir fait l'honneur de juger mon travail.

Je remercie vivement Madame Grigori, Professeur à l'Université Paris-Dauphine d'avoir acceptée de présidé ce jury.

Je tiens à adresser mes remerciements à Monsieur Thomas Schwartz, Professeur à l'Université Santa Clara, pour ses conseils pertinents durant ces années de préparation de la thèse.

J'adresse aussi mes vifs remerciements à Madame Bazgan, Professeur à l'Université Paris-Dauphine et Directrice de l'École Doctorale, pour son soutien et d'avoir appuyé ma demande de réinscription au conseil doctoral connaissant ma situation professionnel.

J'adresse mes vifs remerciements à Madame Moussa, Maître de Conférences à l'Université de Carthage, Tunisie, pour son aide précieuse sur le prototype de base et les algorithmes qu'elle a implémentés.

Je remercie particulièrement Madame Sahri, Maître de conférences à l'Université Paris Descartes, pour tous ses conseils, ses encouragements et son amitié.

Merci à Wassila, Maître de conférences à Centrale Paris et Samir pour leurs encouragements.

Je ne saurais remercier assez ma très chère épouse Émilie. Merci de m'avoir supporté tout au long de ces longues années d'études pour les weekends et soirée passés à rédiger ou à lire des papiers. Heureusement que tu étais à mes côtés.

Résumé

LH $*_{RS}^{P2P}$: Une nouvelle Structure de Données Distribuée et Scalable pour les Environnements Pair à Pair

Cette Thèse propose une nouvelle structure de données distribuée et scalable (SDDS) appelée LH $*_{RS}^{P2P}$, dédiée à l'environnement pair à pair (P2P). Un fichier LH $*_{RS}^{P2P}$ est un ensemble d'enregistrements contenant les données de l'application, chacun étant identifié par une clé. Ces enregistrements sont distribués dans des cases mémoire sur des pairs selon le principe du hachage linéaire (LH*). Tout fichier LH $*_{RS}^{P2P}$ peut s'étndre dynamiquement par des éclatements de cases, en s'étalant potentiellement sur un nombre quelconque de pairs.

De tels fichiers répondent à trois exigences majeures. La première est l'efficacité d'une requête *à clé*. Une telle requête peut être une recherche, une insertion, une mise à jour ou une suppression de l'enregistrement identifié par la clé spécifiée. Une requête à clé s'adresse à une case. La seconde exigence est l'efficacité d'un *scan*, c.-à-d. d'une requête s'adressant à toutes les cases. Les scans sont aussi connus sous le nom de 'Map/Reduce'. Enfin, la troisième exigence est la tolérance à l'indisponibilité de données d'un ou de plusieurs pairs. Le problème peut venir d'une panne d'un pair ou du réseau. Il peut aussi être dû à l'arrêt volontaire du service par le pair. Ce dernier peut d'ailleurs tenter de revenir au service plus tard, en offrant involontairement des données périmées, données qui auraient déjà été reconstruites ailleurs et mises à jour depuis. L'ensemble du problème est connu sous le nom de *Churn*.

LH $*_{RS}^{P2P}$ répond à ces exigences comme suit. En premier lieu, une requête à clé subit au maximum un seul renvoi entre les pairs. Ensuite, un scan nécessite au maximum deux rounds de messages. Par ailleurs, pour pallier le Churn, LH $*_{RS}^{P2P}$ utilise le calcul de parité hérité de la SDDS LH*$_{RS}$. Le fichier supporte ainsi l'indisponibilité de n'importe quels k pairs, où $k > 0$. La valeur de k est initialisée dès la création du fichier. Pour pallier l'accroissement de la probabilité d'indisponibilité de k pairs quand le fichier grandit, cette valeur peut également croitre dynamiquement, quel que soit k. Enfin, la structure offre un nouveau type de requêtes dites *sûres*. Celles-ci protègent contre toute utilisation de données périmées d'un pair 'revenant'.

L'ensemble de ces propriétés fait de LH $*_{RS}^{P2P}$ une SDDS parmi les plus efficaces. À notre connaissance, la seule SDDS caractérisée par un seul renvoi au maximum est *One Hope Lookups*, [GLR03]. Comme nous le discutons plus tard, elle est néanmoins bien plus couteuse en nombre de messages de notification pour maintenir à jour les nœuds du réseau. Les autres variantes de LH* connues, notamment LH*$_{RS}$, peuvent nécessiter deux renvois pour une requête à clé et O(log N) rounds pour un scan, pour un fichier de N cases. D'autres SDDSs nécessitent davantage de messages. Notamment, celle dite Chord, particulièrement populaire malgré son coût de O(log N) d'une requête à clé, [SMK+01]. Enfin, nous ne connaissons actuellement aucune SDDS offrant les requêtes sûres.

La thèse est organisée en six chapitres. Nous commençons, dans le chapitre 1, par une introduction générale à notre travail. Le second chapitre contient l'état de l'art. Nous y présentons les principes fondamentaux des SDDSs et les différentes familles de celles-ci. Nous introduisons aussi les systèmes P2P ainsi que des SDDS dédiées. Nous nous limitons aux SDDSs ayant un intérêt pour notre travail. Le chapitre 3 présente la conception de LH $*_{RS}^{P2P}$, ses principaux algorithmes et propriétés que nous démontrons. Nous y définissons aussi une variante particulièrement importante de LH $*_{RS}^{P2P}$. Dans le chapitre 4, nous présentons notre prototype. Nous y définissons son architecture fonctionnelle globale et celles des pairs serveurs de données et des serveurs de parité. Nous donnons aussi la structure des divers protocoles spécifiques de LH $*_{RS}^{P2P}$ que nous avons introduit et de tout message échangé. Le chapitre 5 présente les mesures de performances validant notre conception et implémentation. Nous terminons par le chapitre 6, où nous présentons la synthèse de notre contribution et les perspectives des travaux futurs.

Notre travail a fait l'objet de quatre publications [YLS07, LYS08, Y08, YS10]. En particulier, l'article [Y08] publié dans 'International Conference for Internet Technology and Secured Transactions (ICTST)', a été jugé parmi les dix meilleures de cette conférence. Ces articles ont été sélectionnés pour être publié dans une version étendue dans 'International Journal of Internet Technology and Secured Transactions' (IJITST), [YS10].

Mots clés : Structure de données distribuée et scalable, SDDS, système pair à pair, P2P, hachage linaire distribué, LH*, haute disponibilité, Churn.

Abstract

LH $*_{RS}^{P2P}$: A new Scalable and Distributed Data Structure for Peer to Peer System.

This Thesis presents a new scalable and distributed data structure (SDDS) for peer to peer environment (P2P) termed LH $*_{RS}^{P2P}$. A LH $*_{RS}^{P2P}$ file is a set of records that contains data. Each record is identified by its primary key. A SDDS distributes its data records in memory buckets spread over the nodes of the P2P network. This distribution follows the linear hashing (LH*) principles. Each LH $*_{RS}^{P2P}$ may scale dynamically with a split process and my spread over an unlimited number of nodes.

A SDDS file must deal with three major issues in P2P system. The first one is the efficiency of a key-based query. This may be an insert, search, update or delete of data record, each record has its own primary key. Each key-based query is sent to an identified data bucket. The second one is the efficiency of a *scan*. This type of query is sent to all nodes and known as 'Map/Reduce' query. The last one is fault-tolerance. The failure of node may happen because of network failure or failure of node. It may also happen by a voluntary of departure of the peer. This one may decide to back and give an out dated data. Thus, the data may be recovered and updated in meantime. This problem is known as a *Churn*.

LH $*_{RS}^{P2P}$ insures that a key-based query needs one forward at maximum when an addressing error occurs. The scan query needs at maximum two rounds. LH $*_{RS}^{P2P}$ copes with the Churn through the LH*$_{RS}$ management of unavailable data and parity buckets. As the result, the file transparently supports unavailability or withdrawal of up to any $k \geq 1$ peers, where k is a parameter that can scale dynamically with the file. LH $*_{RS}^{P2P}$ gives a new query termed *sure query*. This protects against using outdated data.

LH $*_{RS}^{P2P}$ properties make our structure one of the most efficient SDDS. Only *One Hop Lookups* gives one forward of the key-based query [GLR03]. This structure needs a large number of notification massages and implies a larger delay in propagating the notification. LH* and its variants like LH*$_{RS}$ may need two forward for a key-based query and O(log N) for a scan, N is a number of nodes. Others structures may need more forward and messages like Chord. This is one of the most popular P2P structure, it needs O(log N) [SMK+01]. There is no known structure that gives sure queries.

The Thesis is structured in six chapters. We start by an introduction of our work. In the second chapter we present a state of the art that contains a presentation of the SDDS principals. Also, we introduce the P2P systems and well known P2P structures. In the third chapter, we present the LH $*_{RS}^{P2P}$ algorithms, properties and a proof of each property. Also we give a new declustering schema of LH $*_{RS}^{P2P}$. In the fourth chapter, we present the functional architecture of our structure. The fifth chapter gives the performance measurements of LH $*_{RS}^{P2P}$ prototype. Finally we conclude the Thesis and give some perspectives of research.

We have published four papers [YLS07, LYS08, Y08, YS10]. The [Y08] paper is published in international conference ‘International Conference for Internet Technology and Secured Transactions (ICTST)’. The paper is selected in the top ten best papers to be published in the International Journal of Internet Technology and Secured Transactions (IJITST) [YS10].

Key words: Scalable and distributed data structure, P2P system, distributed linear hashing (LH*), high availability, Churn.

Tables des Matières

Tables des Matières

Tables des Matières

Table des Figures

Table des Figures

Chapitre I

Introduction

1. Motivation

La décennie 2000 a été dominée par le développement et la vulgarisation du concept de multi ordinateur qui s'est décliné en plusieurs architectures : grille (Anglais : Grid), pair à pair et récemment nuage (Anglais : Cloud). Les politiques de gestion des systèmes d'informations des organisations se sont mises à intégrer ces architectures à tous les niveaux, que ce soit dans l'administration, le commercial et bien d'autres domaines organisationnels.

Les besoins de ces organisations sont devenus de plus en plus demandeurs d'espace de stockage. Le stockage physique de données n'était pas un problème et ne l'est toujours pas de nos jours. La loi de Moore [M65] [M03] annonçait en effet :

- Chaque nouveau circuit contient deux fois plus de transistors que sa version précédente. La densité de transistors pour une surface donnée croit d'un facteur 2 tous les 12 mois
- Une nouvelle génération de microprocesseurs est lancée en moyenne tous les dix-huit mois.

Cette loi a toujours été vérifiée et continue de l'être. De plus un multi-ordinateur constitue un espace de stockage très étendu.

La loi de Gilder annonçait [G97] également que pour les vingt-cinq années à venir la bande passante des réseaux triplerait, à prix égal tous les ans, et que le débit de transmission quadruplerait tous les trois ans.

La Figure 1 ci-après montre les prévisions des lois de Moore et de Glider. Celles-ci ont toujours été vérifiées au fil des années.

Les multi-ordinateurs, toutes architectures confondues, exigent de nouvelles organisations de données, qui doivent répondre à de nombreux impératifs, tels que le stockage de grands volumes, l'adressage décentralisé, la scalabilité, la haute disponibilité et la sécurité accrue. Pour répondre à ces exigences, Litwin et al. ont proposé et développé une nouvelle classe de structures de données dite *Structures de Données Distribuées et Scalables* (SDDSs) [LSN93].

Les recherches sur les SDDSs ont concerné divers types de projets informatiques tels que les prototypes de systèmes de fichiers distribués (SFD) ainsi que les systèmes de gestion de bases de données (SGBD).

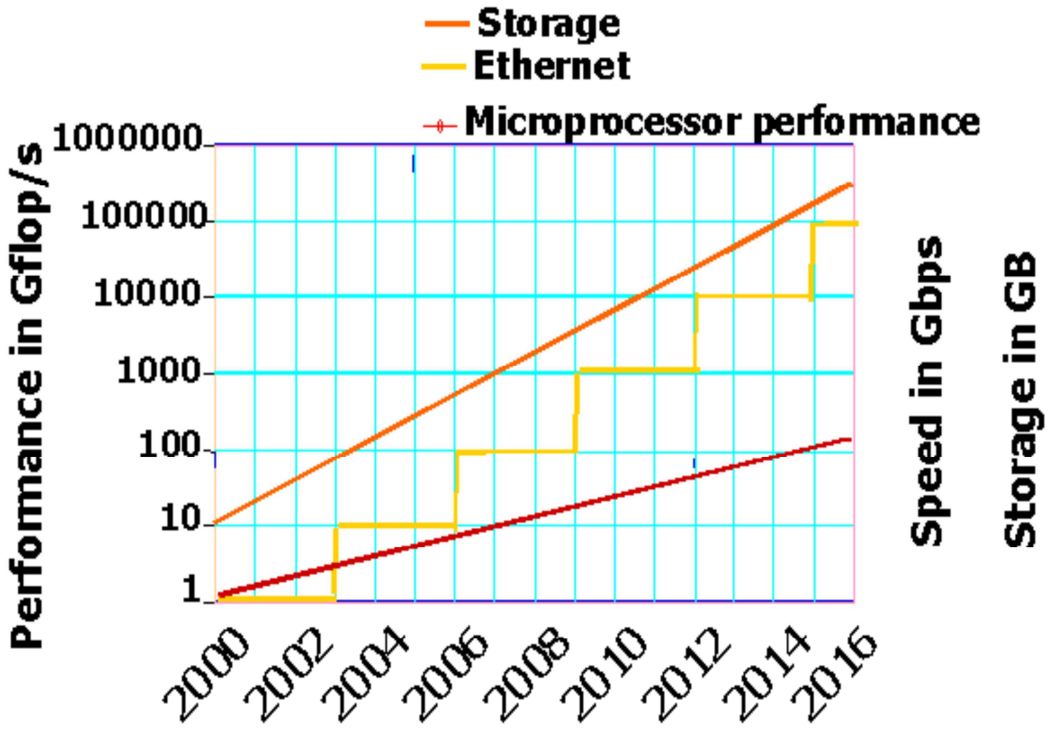


Figure 1. Prévision des lois de Moore et Gilder, [A01]

Les données d'une SDDS et notamment d'un fichier SDDS, sont supposées être réparties sur les sites de stockage du multi-ordinateur que sont les serveurs. Pour leur traitement les données sont rangées dans les mémoires vives de ces serveurs. Elles sont alors dix à cent fois plus rapidement accessibles qu'à partir de disques. Ces données peuvent être manipulées par différentes applications à partir de sites appelés clients. Ces clients gèrent ainsi l'accès mais ne stockent pas de données de la SDDS. Certains sites assurent tout de même les deux fonctions à la fois, ils sont dits *sites pairs*. Les SDDS doivent pouvoir s'adapter dynamiquement à l'accroissement du volume des données. Le fichier devra donc s'étendre de manière progressive d'un seul site serveur ou pair à, théoriquement, un nombre illimité de tels sites. Le placement de données d'une SDDS et son évolution doivent rester transparents pour les applications. Les utilisateurs peuvent ainsi manipuler les données d'une SDDS, via une interface offerte par un client. Comme pour un fichier centralisé les opérations de manipulation sont ; la création, la recherche, l'insertion, la mise à jour et la suppression.

Parmi les SDDS les plus connues, citons LH* [LSN93] et RP $*$ [LSN94] qui ont été parmi les premières à être déployées sur les architectures Client/Serveur. LH*$_{RS}$ [LMS02] est quant à elle connue pour la haute disponibilité des données. Elle est parmi les implémentations les plus récentes. Chord [SMK+01], BATON [JOV05], VBI-Tree [JOV06] et OHL pour 'One Hop Lookups' [GLR03] pour les architectures pair à pair.

Une propriété générale de toutes les SDDS est que l'adressage à partir d'une clé peut nécessiter des messages de renvois entre les serveurs/pairs. LH* garantit un coût de deux renvois au maximum, quel que soit le nombre de serveurs de la SDDS. Aucun autre système n'a, jusqu'ici, réalisé un coût plus bas.

Cependant, le nombre de ces renvois est de l'ordre O(Log *N*) pour les autres SDDS pair à pair citées, *N* étant le nombre de nœuds du réseau, à l'exception de la SDDS OHL [GLR03] qui est de O(1).

2. Contribution

Dans cette thèse, nous nous intéressons plus particulièrement aux problèmes qui guettent les architectures pair-à-pair, 'Grid' et 'Cloud'. La recherche de données devient de plus en plus rapide vu le haut débit des réseaux sur Internet. Néanmoins, un problème majeur continue d'affecter les recherches à clé, celui du nombre de renvois générés suite à une erreur d'adressage d'une requête à clé. Nous basons notre solution sur la structure LH*$_{RS}$ que nous adaptons aux environnements pair à pair. Ce nouvel algorithme, appelé LH $*_{RS}^{P2P}$, garantit un seul renvoi pour rester plus simple que l'algorithme OHL [GLR03]. Ce résultat est impossible à améliorer dans la mesure où la borne de 'zéro renvoi' conduirait nécessairement à une architecture centralisée. Rigaux et al. ont dressé un constat sur la gestion des données dans le web [AMR+11]. Les auteurs n'ont pas manqué de citer les SDDS les plus connues dont celle sur lequel notre travail est basé, LH*.

Notre démarche est simple. Nous utilisons l'expansion du fichier SDDS et les erreurs d'adressage pour mettre à jour l'image d'adressage des pairs impliqués dans le processus. À chaque fois qu'un pair, avec une case de données d'une capacité '*b*' atteint sa limite de stockage il éclate et transfert la moitié de ses données vers un nouveau pair. Lors de cet éclatement, nous mettons à jour son image d'adressage ainsi que l'image des pairs qui sont à sa charge, appelés *pupilles*, s'ils existent.

Un second problème découle du *Churn*, qui concerne toutes les architectures pair à pair, compte tenu leur propriété dynamique. Un pair quelconque peut rejoindre un réseau pair à pair et le quitter sans prévenir à la suite d'une panne ou déconnexion volontaire. Nous proposons ici le concept de requêtes sûres pour pallier ce problème. Ces requêtes sûres peuvent être des recherches, des insertions, des mises à jour ou des suppressions d'enregistrement de données. Elles permettent d'identifier si un pair portant une case de données a été reconstruit suite à son indisponibilité. Si tel est le cas, le pair qui était indisponible obtient la nouvelle mise à jour corrective. Grâce à cela, les requêtes sûres peuvent ainsi garantir l'exécution correcte de toutes les requêtes en toutes circonstances.

3. Plan de la Thèse

La suite de cette thèse est organisée en six chapitres. Après avoir introduit notre travail, nous faisons l'état de l'art dans chapitre 2. Nous y présentons les principes fondamentaux des SDDSs et différentes familles de celles-ci. Nous présentons aussi les systèmes P2P ainsi que des SDDS dédiées, les plus connues ou étant d'intérêt pour notre travail. Le Chapitre 3 présente la conception de LH $*_{RS}^{P2P}$, les principaux algorithmes et ses propriétés. Dans ce chapitre nous démontrons chaque propriété. Nous y définissons aussi une variante particulièrement importante de LH $*_{RS}^{P2P}$. Dans le quatrième chapitre, nous présentons notre prototype. Nous y définissons son architecture fonctionnelle globale et celles de pairs serveurs de données et de serveurs de parité. Nous donnons aussi la structure des divers protocoles

spécifiques de LH $*_{RS}^{P2P}$ que nous avons introduit et de tout message échangé notamment. Le Chapitre 5 présente les mesures de performances validant notre conception et implémentation. Nous terminons par le Chapitre 6 où nous présentons la synthèse de notre contribution et ses perspectives.

Chapitre II

Les Structures de Données Distribuées et Scalable (SDDS)

1. Introduction

La performance des réseaux a connu des améliorations importantes ces dernières années avec des débits de plus en plus élevés, conséquence de l'augmentation de la vitesse des CPU, de la taille des RAM et la performance des réseaux de télécommunication. Cependant, les disques durs, limités par des contraintes mécaniques n'ont pas atteint le même succès du point de vue temps d'accès (lecture/écriture) aux données stockées. Dans le but d'améliorer les temps d'accès et assurer la disponibilité des données en cas de panne plusieurs projets informatiques ont été lancés. De là, vient l'idée d'exploiter les performances des multi-ordinateurs et en particulier les structures de données distribuées pour l'organisation des données stockées.

En effet, les multi-ordinateurs constituent la solution idéale pour garantir le partage des données et leurs disponibilités en cas de panne. Cependant le problème d'accès aux données stockées sur disque et leurs disponibilités se pose toujours. Les Structures de Données Scalables et Distribuées ou SDDS (Anglais : ***S**calable and **D**istributed **D**ata **S**tructures*) ont été introduites comme solution pour organiser ces données et garantir leurs disponibilités en cas de panne d'un ou plusieurs nœuds. Les SDDSs utilisent la RAM de chaque nœud constituant le multi-ordinateur pour le stockage des données. Les premières SDDSs ont été présenté dans [LNS93a, LNS93b].

Les SDDS assurent des temps d'accès aux données beaucoup plus courts que les disques durs du fait du stockage sur la mémoire vive, RAM. De plus ces nouvelles structures disposent :

- De plus du traitement parallèle, car les requêtes peuvent s'exécuter sur chaque *nœud* du multi-ordinateur sans avoir un goulot d'étranglement.
- D'une capacité de stockage potentiellement illimitée car il suffit de connecter de nouveaux ordinateurs au réseau.

Ces caractéristiques assurent aux SDDS des performances de traitement largement supérieures à celles des structures de données traditionnelles.

Nous présentons ci-après la typologie générale des réseaux puis des multi-ordinateurs. Ensuite, nous décrivons les SDDS et leurs caractéristiques. Nous nous intéresserons particulièrement à une SDDS appelée LH*$_{RS}$ [LMS04], sur laquelle repose en partie notre travail.

2. Architecture des Systèmes Informatiques

Avant de présenter les principes généraux des structures de données et distribuées nous allons faire la présentation des différentes typologies des réseaux information. Ces architectures principales sont utilisées dans les systèmes de gestion de base de données [GG96] [G97].

2.1. Architecture Client/serveur

Cette architecture se présente en un modèle qui met en jeu une répartition de rôles entre serveurs et clients. Les clients constituent un support pour les applications et les interfaces utilisateurs. Ils envoient leurs requêtes aux serveurs. Les serveurs gèrent les données, gèrent les transactions, l'optimisation et sécurité des données. Ils effectuent le traitement des requêtes émises par les clients et retournent les réponses. La Figure 2 ci-après présente l'exemple d'une telle architecture.

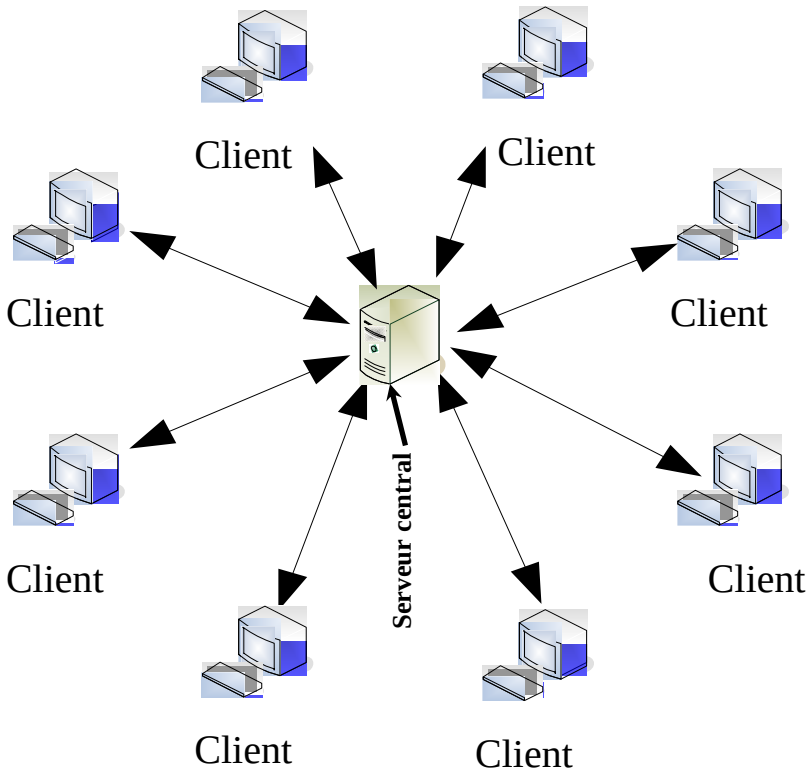


Figure 2. Architecture Client/Serveur

Plusieurs variantes de cette architecture peuvent être mises en évidence :

- L'architecture mono-tâche : Un serveur exécute les requêtes une par une. L'inconvénient majeur de cette architecture ne diffère pas trop de celui des bases de

données centralisées puisque la base est stockée sur seulement une machine (le serveur).

- L'architecture multitâche. Un serveur est capable de traiter plusieurs requêtes clients en parallèle. Une telle architecture permet de meilleures performances en présence d'un nombre important d'utilisateurs. Cependant, elle peut conduire à un client lourd si chaque client gère ses propres connexions à son serveur plus les traitements des données ou à un client léger si les traitements de gestion sont concentrés sur les serveurs.

Ces architectures ont été les premières à être utilisées pour la généralisation et la distribution des structures de données classiques dont Internet était le précurseur.

En effet, Internet a permis le développement des applications client/serveur. Toutefois, ces dernières années, Nous avons vu la démocratisation du concept du *Cloud Computing* (Nuage). Il s'agit d'un concept de déportation, sur des serveurs distants, des traitements et services informatiques usuellement localiser sur le poste de l'utilisateur. L'utilisateur peut être une personne physique ou une entreprise qui utilisera ces traitements et services en ligne via Internet de manière évolutive et sans s'occuper de la gestion de l'infrastructure qui représente des coûts financiers souvent importants.

Les données et les applications dont l'utilisateur se sert ne sont plus sur sa machine locale mais sur des serveurs distants interconnectés via une excellente bande passante. On dit que les données et applications sont sur un Cloud. Plusieurs acteurs et promoteurs de ce concept se sont déjà lancés dans cette nouvelle technologie de traitement de données, tel que Google, Amazon, Microsoft et tant d'autres. Exemple en Figure 3, ci-après.

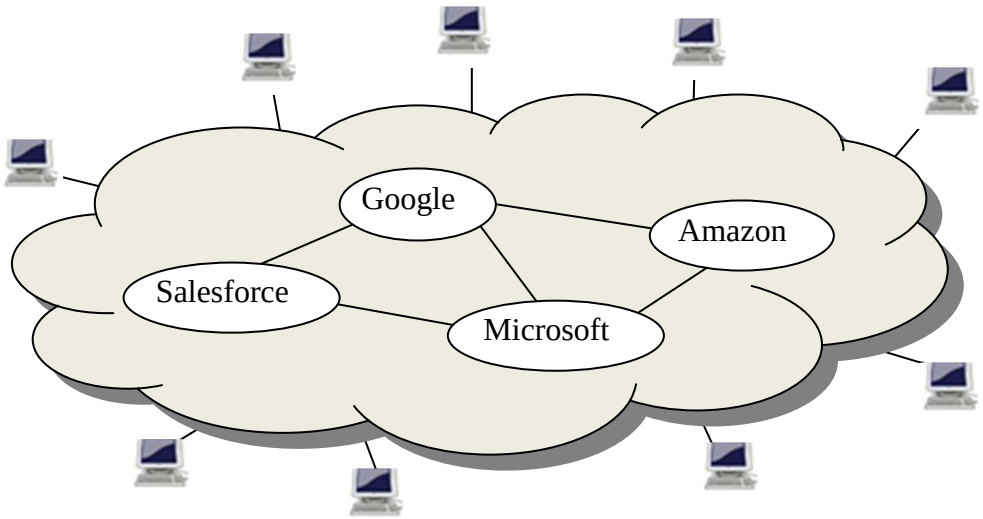


Figure 3. Vu d'un *Cloud Computing*

Le *Cloud Computing* peut également libérer l'utilisateur des contraintes de capacité de mémoire vives. C'est le cas du RAM-Clouds [OAE+10] proposé par Ousterhout et son équipe de l'université Stanford. Leur solution permet d'étendre au *Cloud* toutes les techniques de mémoire distribuée telle que LH* [LNS93], RP* [LNS94], CTH* [Z04] dont nous reparlerons plus tard.

2.2. Architecture Pair-à-Pair

Le Pair-à-Pair [S01] (Anglais : Peer to Peer noté P2P) est un réseau de nœuds, où chaque nœud joue le rôle d'un Client et d'un Serveur en même temps, voir Figure 4. Cette architecture P2P de réseau s'est imposée comme technologie de référence pour le partage de ressources multimédia. Nous pouvons classifier les réseaux P2P en 3 types :

- L'architecture non structurée : tous les nœuds sont au même niveau sur le réseau logique. Il n'y a pas de hiérarchie. La recherche de données s'effectue par diffusion aux nœuds voisins de la requête (Anglais : *Flooding*). Le temps de recherche est difficile à évaluer.
- L'architecture structurée : les nœuds sont organisés suivant une structure de données telle que le Table de Hachage Distribuée, de Hachage Linaire, des arbres de recherches binaires ou tant d'autres que nous citerons par la suite. Les données sont stockées sur les nœuds. Le temps de recherche est estimé à O(log *N*), où *N* est le nombre de nœuds du réseau. Nous verrons que la structure de données que nous traitons dans cette thèse permet de garantir un temps de recherche de O(1).
- L'architecture hybride : certains nœuds sont dits *Super-Pair*. Ils gèrent les index des données des autres nœuds et le routage du réseau. Le temps de recherche est partiellement garanti.

Pourquoi l'utilisation des systèmes P2P? Pour plusieurs raison les réseaux P2P se sont généralisés et ce malgré leurs inconvénients :

- Avantages :
  - Alternative du Client/Serveur
  - Autonomie des nœuds
  - Rapidité de mise en place à moindre coût (finance)
  - Échange de données rapide en profitant de bande passante
  - Équilibrage de charge
- Inconvénients :
  - Sécurité des accès, que nous ne traiterons pas dans notre travail.
  - Churn, un phénomène non déterministe d'arrivée et de départ de pairs du réseau. Nous traitons ce problème dans le chapitre suivant.
  - Absence d'applications pour la gestion de données dans un tel environnement.

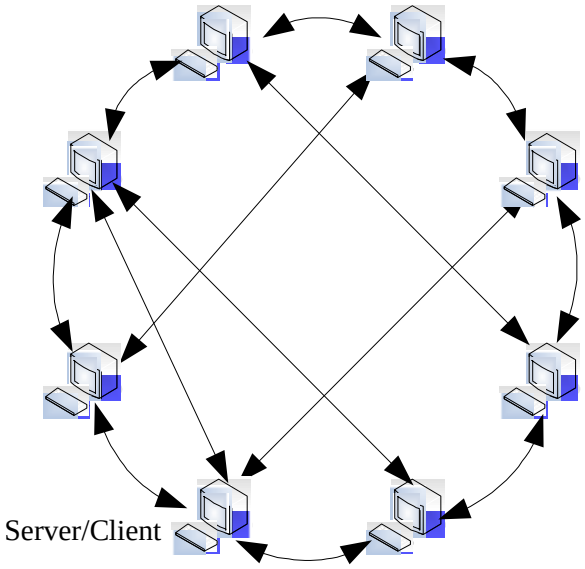


Figure 4. Architecture Générale d'un Système P2P

La suite de notre développement de la thèse s'intéresse plus particulièrement à l'architecture P2P structurée. Avant cela nous rappelons le concept des SDDS qui constitue le fondement de notre travail.

2.2.1.1. Le Churn

L'une des caractéristiques d'un environnement pair à pair est la dynamique de l'adjonction et départ de pairs. Ce phénomène est connu en anglais comme le *Churn*. Farsite [DH06] [BDH07] offre un bon exemple de Churn dans le contexte d'un système pair à pair de fichiers distribués. Farsite range tous ses fichiers et répertoires sur des pairs, qui peuvent donc se déconnecter à tout moment du réseau. Ces déconnexions apparaissent à Farsite comme des pannes et doivent être traitées comme telles. Toute application pair à pair réelle (ex. Gnutella [KM02], FastTrack [KAZ03], BitTorrent [B02]), (indépendamment de la conception théorique de l'application) est confrontée au Churn. Des études de mesures du Churn ont été réalisées pour mesurer la durée de vie d'un pair sur le réseau [RGRK04] [SR06].

Par définition le temps de session est le temps qui s'écoule entre le moment où un pair se connecte au réseau et le temps de sa déconnexion. La durée de vie d'un pair est l'intervalle de temps entre sa première connexion au réseau et l'instant où le pair quitte définitivement le réseau. Idéalement, un pair devrait rester toujours connecté durant sa durée de vie. Ce n'est pas le cas toujours le cas pour des raisons de panne du pair, d'indisponibilités du réseau ou de déconnexions volontaires. La disponibilité d'un pair est donc la somme des temps de sessions divisée par la durée de vie du pair [RGRK04]. La Table 1, ci-après, présente le temps de session des pairs dans différents systèmes pair-à-pair, qui sont tous des systèmes de distribution de fichier où chaque fichier réside sur un seul pair. Comme nous le constatons, Table 1, ces temps de session varient de système en système. Gnutella et Napster offrent un

temps de disponibilité inférieur à 60 minutes pour 50% des pairs initialement connectés au réseau. FastTrack offre un temps de session inférieur ou égal à une minute pour 50% de ses pairs, ce qui est très court. Kazaa [KAZ03], lui aussi offre des temps de session très courts parce qu'il utilise le même protocole de partage que FastTrack. Ainsi, le temps de session est variable selon le profil de l'utilisateur utilisant son pair pour la recherche de fichier ou son téléchargement.

Auteur	Système observé	Temps de session
Saroiu [SGG02]	Gnutella, Napster	50% ≤ 60 min
Chu [CLL02]	Gnutella, Napster	31% ≤ 10 min
Sen [SW02]	FastTrack	50 % ≤ 1 min
Bhagwan [BSV03]	Overnet	50 % ≤ 60 min
Gummadi [GDS+03]	Kazaa	50 % ≤ 2,4 min

Table 1. Temps de Session par système Pair-à-Pair, [RGRK04]

Une étude plus poussée du Churn est présenté par Godfrey, Shenker et Stoica [GSS06] comme le montre la Table 2. Cette étude se base sur l'analyse des *traces d'exécution* (Anglais : Traces) de cinq systèmes. La colonne durée représente le temps de mesure pendant lequel la disponibilité de chaque machine est testée par l'envoi d'un 'Ping'. La seconde colonne représente le nombre moyen de nœuds en marche. La dernière représente le temps moyen d'une session par nœud. Ainsi, pour PlanetLab [BBC+04], environ 50% des nœuds présentent un temps moyen d'indisponibilité de 3,9 jours. Plus de détails sont présentés dans [GSS06] quant à la simulation faite avec les traces de disponibilité (Anglais : Availability Traces).

L'efficacité d'une application pair à pair dépend de la manière dont celle-ci gère le Churn pour garantir la disponibilité des données après le départ ou panne d'un pair.

Trace	Durée (jours)	Moyenne de nœuds en marche 'modes up'	Moyenne temps de session par nœud
PlanetLab	527	303	3,9 jours
Web sites	210	113	29 heures
Microsoft PCs	35	41970	5,8 jours
Skype	25	710	11,5 heures
Gnutella	2,5	1846	1,8 heure

Table 2. Temps de disponibilité [GSS06]

3. Les Structure de Données Distribuées et Scalables

Une SDDS se base sur une structure de données classique. C'est la généralisation de celle-ci sur un multi-ordinateur qui en fait d'elle une structure scalable et distribué. Cette section présente la définition d'un multi-ordinateur, les principes généraux et caractéristique d'une SDDS, enfin la classification des SDDS.

3.1. Les Multi-Ordinateurs

Un *multi-ordinateur* est une collection d'ordinateurs sans mémoire partagée. Ces ordinateurs sont reliés par un réseau de type LAN ou WAN (Anglais : Local Area Network, Wide Area Network) ou encore par un bus [T85]. La puissance théorique cumulée des ressources en calcul et en mémoire d'un multi-ordinateur est impossible à atteindre par un ordinateur traditionnel. Cette perspective a permis de favoriser l'apparition de plusieurs projets de recherche autour de la conception et la mise en œuvre d'un multi-ordinateur. L'un des axes de recherche majeur a été la construction de systèmes de gestion de fichiers scalables et distribués dont le traitement est entièrement en RAM distribuée du multi-ordinateur. Grâce à l'amélioration de la vitesse des réseaux, les temps d'accès à une mémoire vive distante est aujourd'hui plus court que le temps d'accès à un disque local comme le montre la Table 3. Ainsi l'accès à un enregistrement sur RAM d'un ordinateur en local nécessite 100 nanosecondes contre 10 à 100 microsecondes pour un enregistrement stocké en RAM distante connectée en réseau et enfin de 10 millisecondes pour un enregistrement stocké sur le disque local d'un ordinateur.

Gray et Garcia-Molina ont présenté les avantages et l'intérêt de l'utilisation des RAM comme support de stockage par rapport aux disques dures, [G78] et [GLV84]. Aujourd'hui l'utilisation de nouveaux supports tels que les disques de type SDD (Solide-State-Drive) est chose faite offrant un temps d'accès entre 35 et 100 microsecondes contre 500 et 10000 microsecondes pour les disques durs classiques. Sachant que le temps de lecture est plus rapide que le temps d'écriture sur les deux supports et que ces temps dépendent de la technologie des constructeurs. La nouvelle génération de stockage de données sera sans doute les RAM qui ne cessent de se développer avec une capacité de stockage de plus en plus grande et une rapidité d'accès de plus en plus performante.

Toutefois, la conception d'un multi-ordinateur nécessite la réfection de logiciels système. Ils sont à faire ou à refaire, notamment les systèmes de gestion de fichiers, les systèmes de gestion de base de données et autres applications Web.

Ressource	**RAM locale**	**RAM distante** (réseau gigabit)	**RAM distante** (Ethernet)	**Disque local**
Temps d'accès	100 nsecs	1 µsecs	100 µsecs	10 msec

Table 3. Estimation des temps d'accès aux données.

3.2. Les Principes Architecturaux d'une SDDS

Les SDDS constituent une nouvelle famille de structures de données, basées sur le modèle client/serveur. Elles stockent les données dans des *cases* (Anglais : Bucket) sur des sites appelés *serveurs*, d'autres sites appelés *clients* y accèdent. Les sites clients gardent des paramètres pour le calcul de l'adresse des fichiers sur les sites serveurs. Ces paramètres constituent l'*image* du client sur le fichier SDDS. Le client fournit une application pour les utilisateurs pour se connecter au réseau, c'est l'interface Client/Utilisateurs. Ainsi ils pourront accéder aux données sauvegardées dans les cases de données (Anglais : Data Buckets) logées sur les serveurs. Il faut noter que l'accès aux données est fait de manière transparente pour l'utilisateur. La Figure 5 montre l'architecture d'un système à base d'une SDDS. Plusieurs schémas et d'architectures sont définis selon la structure de données utilisée pour gérer les données sur le réseau. Cependant le principe est le même, Client/Server. Dans ce qui suit, nous distinguerons l'évolution de cette architecture et la généralisation des structures de données aux systèmes Pair-à-Pair.

Un fichier SDDS contient des données typiquement sous la forme d'enregistrements ou bien d'articles identifiés par une *clé primaire* ou par plusieurs clés. Le stockage de ces données débute sur un serveur et peut être étendu par des insertions, théoriquement à un nombre quelconque de ceux-ci. Ce qui rend la capacité de stockage d'une SDDS potentiellement illimitée. Cette nouvelle structure dispose aussi du traitement parallèle, ce qui fait que l'augmentation de la taille des données ne détériore pas les performances d'accès.

Il est utile que les données d'une SDDS résident pour le traitement en mémoire vive distribuée du multi-ordinateur. Comme nous venons de le voir, le temps d'accès aux données en mémoire vive distribuée est bien plus court que celui des données stockées sur disque. Plusieurs SDDS ont été conçu sur ce principe, nous les présentons dans la Section 3.4.

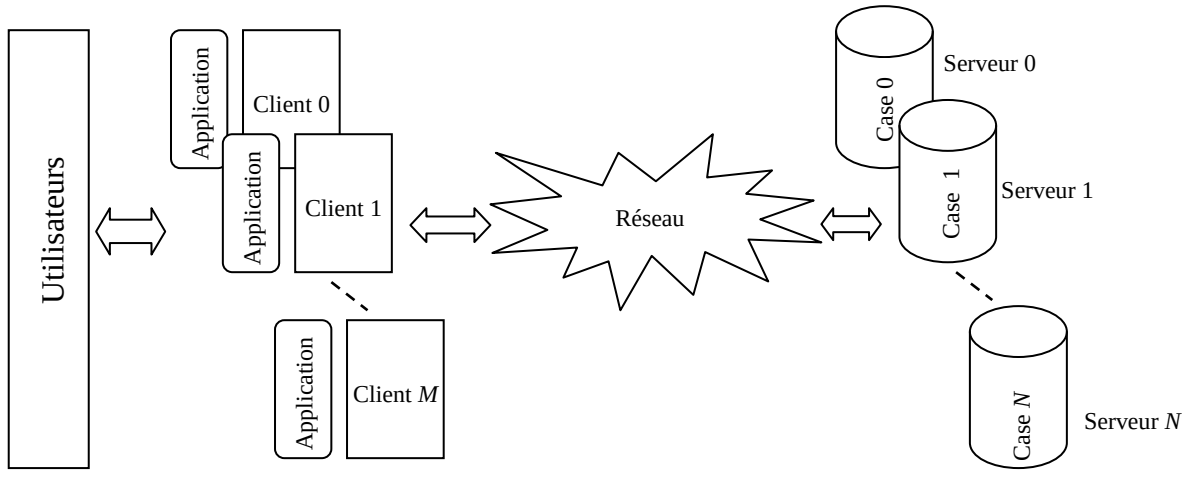


Figure 5. Architecture générale d'un système à base de SDDS

Plusieurs prototypes ont montré que toutes ces caractéristiques peuvent assurer à une SDDS des performances supérieures à celles des structures de données classiques. L'architecture de toute SDDS se base également sur les principes suivants, [LNS94, KLR94] :

- Un fichier SDDS n'a pas de répertoire central d'accès, ce qui évite d'avoir des points d'accumulation, qui engendreraient des goulots d'étranglement et dégraderaient les temps d'accès aux données du fichier.
- Les fichiers SDDS croissent et s'étendent suite à des insertions d'une manière incrémentale et transparente pour l'application (client). Il débute généralement sur un seul site (serveur) initial de stockage jusqu'à la surcharge de ce dernier. Le fichier est alors étendu par un éclatement qui transfert la moitié des données vers un autre site de stockage. Ainsi le fichier peut s'étendre progressivement à un nombre théoriquement illimité de sites (serveurs).
- Les serveurs constituant le fichier SDDS interagissent avec des applications autonomes appelées 'clients'. Chaque client supporte le logiciel propre aux SDDS et gère notamment sa propre image de la structure du fichier SDDS. Puisque les modifications du schéma de la structure du fichier SDDS, dues aux éclatements, ne sont pas diffusées d'une manière synchrone aux clients, cette image peut être inexacte. De ce fait, un client est susceptible de faire des erreurs d'adressage et d'envoyer une requête à un serveur incorrect.
- Chaque serveur est capable de détecter une erreur d'adressage le concernant. Lorsqu'une erreur est détectée alors la requête est redirigée vers un autre serveur. Si le nouveau serveur n'est toujours pas le bon alors d'autres redirections peuvent survenir. Toute bonne SDDS doit à la fois minimiser le nombre de redirections et corriger toute erreur de routage. Le bon serveur participant au processus de redirection doit, pour cela, envoyer un message correctif noté **IAM** (Anglais : **I**mage **A**djustment **M**essage) au client ayant fait l'erreur d'adressage. Lui permettant d'ajuster son image.

3.3. Caractéristiques d'une SDDS

Une SDDS peut posséder plusieurs caractéristiques qui découlent essentiellement des systèmes distribués [LNS96, L97]. Il s'agit principalement de *scalabilité* et la *disponibilité*.

3.3.1. La Scalabilité

Un système est dit scalable lorsqu'il s'adapte à la taille des fichiers qu'il gère de telle manière que ses temps de réponse puissent restés constants [G93]. Deux facteurs principaux caractérisent la scalabilité d'un système [G93] [G99]. Ce sont :

- Le *facteur d'échelle* (Anglais : Scale-up factor) est le ratio entre le temps de réponse d'une application et le nombre de serveurs alloués à cette application. Idéalement, ce facteur devrait être constant, ce qui est le cas lorsqu'il y a une relation linéaire entre le nombre de serveurs alloués à une application et son temps de réponse. Ce n'est pas toujours le cas comme le montre la Figure 6. Il y a des cas où une augmentation du nombre des serveurs proportionnelle à la charge ne suffit

pas à maintenir le temps de réponse. Nous nous trouvons alors en présence d'un système sous-linéaire.

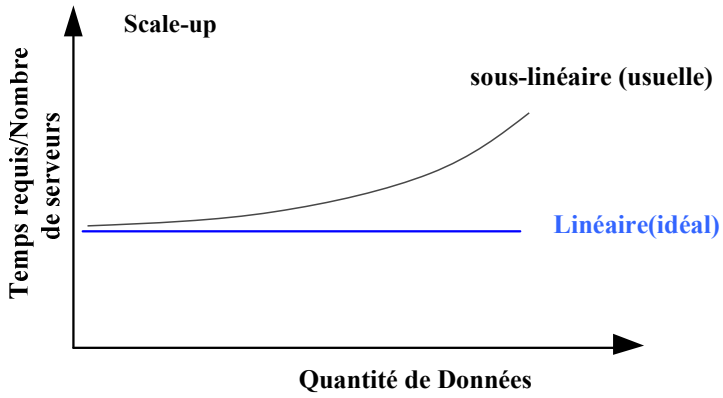


Figure 6. Facteur d'échelle

- Le *facteur de rapidité* (Anglais : Speed-up factor) : mesure la diminution possible du temps de réponse à charge constante lorsqu'on augmente le nombre de serveurs. Idéalement ce facteur de rapidité est linaire ; si nous multiplions par un facteur *n* le nombre des serveurs, le nombre total d'opérations par seconde exécutées par le système devrait être multiplié par le même facteur *n*. Comme la Figure 7 nous le montre, ce n'est pas toujours le cas. Nous nous trouvons alors en présence d'un facteur de rapidité sous linéaire.

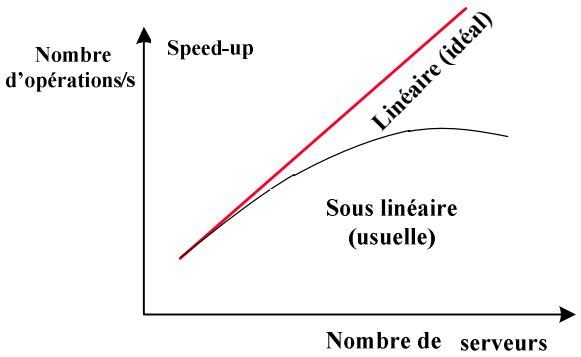


Figure 7. Facteur de rapidité

3.3.2. La Disponibilité

Lorsqu'une structure de données (ou un fichier) s'étend sur plusieurs serveurs, la probabilité qu'elle ne soit pas entièrement disponible à un moment donné augmente. Par exemple, si *n* est le nombre de serveurs, et *p* la probabilité pour qu'un serveur soit en service, alors la probabilité *p(n)* pour que la structure entièrement soit disponible est p^n. Si $p = 0.98$ et $n = 10$, alors $p(n) = 0.81$. Si $n = 100$ alors $p(n) = 0.00000000168$, environ zéro

[LRR97]. Un moyen de résoudre le problème de la disponibilité est d'ajouter des serveurs de parité à la structure distribuée [LS00]. Ce problème de disponibilité concerne aussi les architectures P2P.

3.4. Classification des SDDS

La plus part des travaux dans le domaine des SDDS consistent à partir d'une structure de données classique et de tenter de l'adapter aux environnements distribués. C'est le cas, pratiquement de toutes les SDDS proposées jusqu'à présent.

Selon l'organisation interne des données et le mécanisme qui permet d'y accéder, les SDDS peuvent être classées en plusieurs catégories, comme le montre la Figure 8 ci-après. Nous allons détailler ces catégories maintenant.

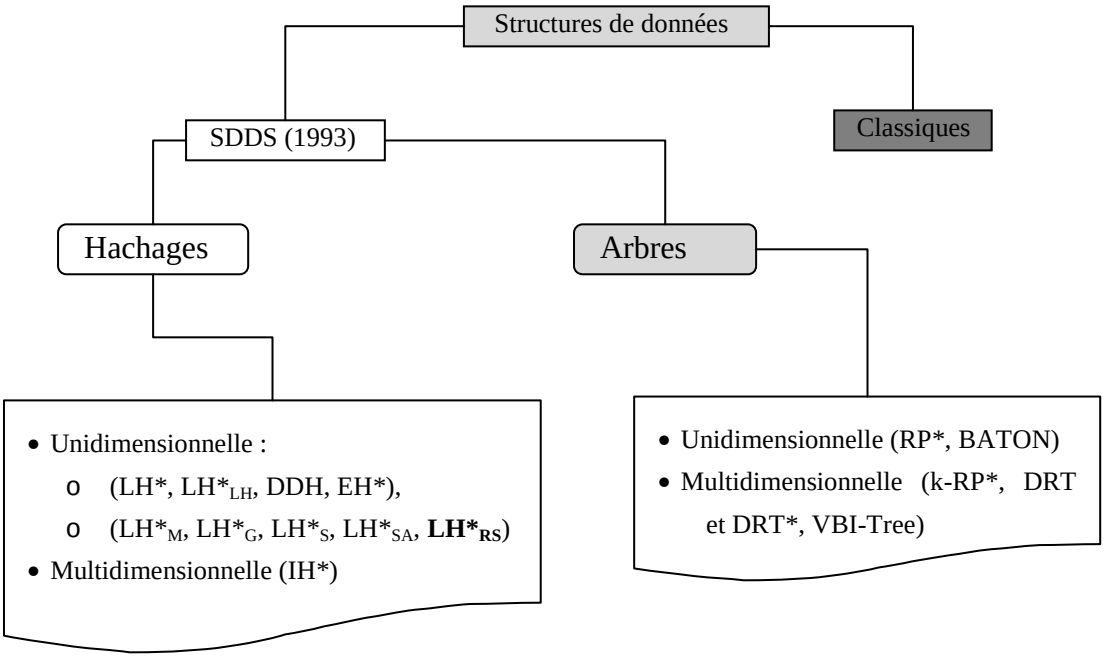


Figure 8. Classification des SDDS

3.4.1. SDDS Basées sur les Arbres

Les structures de données ordonnées telles que les B-arbres [B71] sont les plus utilisé lorsque le fichier doit supporter des requêtes par intervalle sur des données uni-dimensionnelles, des parcours transversaux des enregistrements ou des recherches d'enregistrements les plus proches (Anglais : Nearest, Neighbor search). Litwin, Neima et Schneider ont proposé une famille de SDDS pour les fichiers (mono-clés) ordonnés, appelée RP* pour 'Distributed and Scalable Range Partitioning' [LNS94]**,** dite aussi partitionnement par intervalle. Cette famille comprend plusieurs variantes [LNS94] telles que : $RP*_N$ (No index), $RP*_C$ (Client index) et $RP*_S$ (Server index). Le but majeur de RP* et ses variantes est

de construire des fichiers distribués qui préservent l'ordre des enregistrements. RP* partitionne ses fichiers en intervalles définis par les valeurs d'une clé et assigne à chaque serveur un intervalle [$Clé_{min}$, $Clé_{max}$] différent.

A) *RP*$*_N$ (No index)*

En l'absence d'index, tout client envoie ses requêtes par messages multicast car ils s'adressent à ensemble des serveurs qui appartiennent à un même groupe de réseau donné. Tout serveur ayant reçu le message vérifie si la clé de l'enregistrement demandé appartient à son intervalle. Si oui, l'enregistrement est retourné au client sinon la requête est négligée.

B) *RP*$*_C$ (Client index)*

Dans cette variante chaque client possède un index explicitant quel nœud a chaque intervalle de clés. Grâce à cet index, les clients peuvent envoyer leurs requêtes par messages unicast (message point à point destiné à un seul serveur). Un client peut faire une erreur d'adressage par suite d'une image inadéquate. Dans ce cas, le serveur recevant la requête la redirige par multicast. Celui qui exécute la requête du client lui répond et lui adresse un IAM.

C) *RP*$*_S$ (Server index)*

Cette dernière variante introduit une image du fichier SDDS au niveau même des serveurs. La redirection en cas d'erreur se fait par messages unicast. L'image sur les serveurs est construite sous la forme d'un index B-arbre distribué sur le réseau [AWD01]. L'index est construit sur la distribution des clés des enregistrements de données. Cet index est paginé et chaque page est stocké sur un serveur séparé. Nous distinguons, les serveurs d'index des serveurs de cases de données.

Une extension RP* aux fichiers multi-attributs a donné naissance à la famille k-RP*$_S$ qui est donc une version multidimensionnelle de RP* [LN95]. Dans cette SDDS, nous trouvons des serveurs pour le stockage de données (Anglais : Bucket Servers) et des serveurs d'index (Anglais : Index Servers).

Une autre SDDS pour les fichiers ordonnés, appelée DRT (Anglais : Distributed Random Search Tree), a été proposée par Kröll et Widmayer [KW94]. DRT utilise une version distribuée des arbres binaires de recherche (Anglais : Binary Search Tree) pour l'accès aux données à partir des clients et des serveurs. DRT* est une amélioration du DRT.

Litwin, Neimat, Schneider ont montré que DRT et RP* permettent toutes les deux de construire des fichiers plus larges et plus rapides par rapport aux structures de données traditionnelles [LNS96].

RP* est utilisé pour implémenter le système de fichiers distribués et scalables 'SDDS 2005' [LMS03]. Litwin et Sahri ont utilisé ces même principes pour l'implémentation du nouveau système de gestion de bases de données SD-SQL Server (Anglais : Scalable and Distributed SQL Server) [LS02]. Le partitionnement de la famille de RP* est utilisé par les chercheurs de Google dans leur système Google File System (GFS) [GGL03] et Bigtable [CDG+08].

3.4.2. SDDS Basées sur le Hachage :

Elles constituent une extension des schémas de hachage classiques sur les multi-ordinateurs. Nous avons :

- **LH*** (*Scalable and Distibuted Linear Hashing)* [LNS93] : premier schéma proposé en 1993. C'est une adaptation du hachage linéaire *(Linear Hashing)* [L80] aux environnements distribués. C'est la plus connue et qui inspira plusieurs travaux dont le nôtre.
- **LH*$_{LH}$** [KLR96] : est une variante de LH*. Ce schéma utilise un premier niveau d'index correspondant à LH*, permettant aux clients d'accéder aux serveurs. Le deuxième niveau d'index correspond à l'organisation interne des cases de données suivant l'algorithme LH.
- **DDH** (*Distributed Dynamic Hashing*) [D93] : c'est une version distribuée du hachage dynamique.DDH permet plus d'autonomie d'éclatement, par l'éclatement immédiat des cases en débordement donc nous n'avons pas besoin de site coordinateur, ce qui peut être un avantage. Le coordinateur est un serveur particulier chargé de gérer les éclatements et de garder une trace des paramètres réel du fichier SDDS. La DDH a introduit la notion populaire de table de hachage dynamique dite en anglais Distributed Hash Table (DHT).
- **EH*** (*Distributed Extensible hashing*) [HBC97] : c'est une version scalable et distribuée du hachage extensible. EH* permet une bonne utilisation de l'espace de stockage et offre un mécanisme de traitement de requêtes très efficace.
- **IH*** (*DistributedInterpolation-based hashing*) [BZ02] : Elle constitue, à la fois, une adaptation du hachage par interpolation proposé par W. Burkhard dans [B83] aux environnements distribués et une introduction de la notion d'ordre et de l'aspect multidimensionnel à la SDDS LH*.

D'autres variantes de LH* ont été proposées pour assurer la haute disponibilité. Celle-ci assure à une SDDS la continuité de fonctionnement de manière transparente, quand un ou plusieurs de ses sites serveurs tombent en panne.

Ces SDDS font généralement appel à certains principes de redondance et de récupération de données. Il s'agit de : *Miroring* pour LH*$_M$ [LN96], la fragmentation (Anglais : Striping) LH*$_S$ [LN96] et le groupement d'enregistrements (Anglais : Grouping) LH*$_G$ [L97] [LMR98], la *k*-disponibilité tels que LH*$_{RS}$ utilisant *Reed-Solomon Code* [LS99] et LH*$_{SA}$ (SA pour dire 'Scalable Availability') [LMR98]. RAID6 aussi utilise les mêmes principes que LH*$_{RS}$. Nous nous intéresserons particulièrement à LH*$_{RS}$ que nous allons décrire un peu plus dans ce qui suit.

3.5. Adaptation des Structures de Données aux Environnements Pair à Pair

Ces dernières années, nous assistons à un retour aux origines de l'Internet car cet Internet a été originalement conçu comme un système pair à pair. Il existe plusieurs applications de partage de fichiers sur Internet tels que Gnutella et Kazaa. Le mauvais fonctionnement de ces systèmes lorsque la taille des fichiers qu'ils gèrent augmente a également montré l'utilité de la

distribution de structures de données pour construire des fichiers scalables. Nous citons à titre d'exemple Chord [SMK+01], BATON [JOV05], OHL [GLR03].

A) *Chord*

Ce système est une adaptation des tables de hachages dynamiques, notées DHT (Anglais : Distributed Hash Table) [D93], aux environnements pair à pair. Comme le montre la Figure 9 Chord organise chaque système pair à pair en un anneau où chaque pair gère une partie de la DHT. Chaque pair tient une table de routage dite '*Finger Table*', voir Figure 10. La Figure 9 montre un exemple où les carrés représentent les clés, les petits disques sur l'anneau représentent les identifiants.

Chord est basé sur un espace de recherche de *m* bits, ce qui permet d'avoir jusqu'à 2^m clés. Il assigne à chaque pair ou nœud une clé et les organise en un anneau dans l'ordre de leurs clés. Les clés et les pairs sont hachés sur le même anneau. La '*Finger Table*' d'un nœud contient les adresses des nœuds situés à différents positions relatives dans l'anneau. Parmi ces nœuds se trouve le nœud à 180° dans l'anneau, les nœuds forment des ongles droits et ainsi de suite. Toutefois, chaque nœud connait les adresses correspondantes à toutes les clés des intervalles entre sa propre clé et la clé de son prédécesseur.

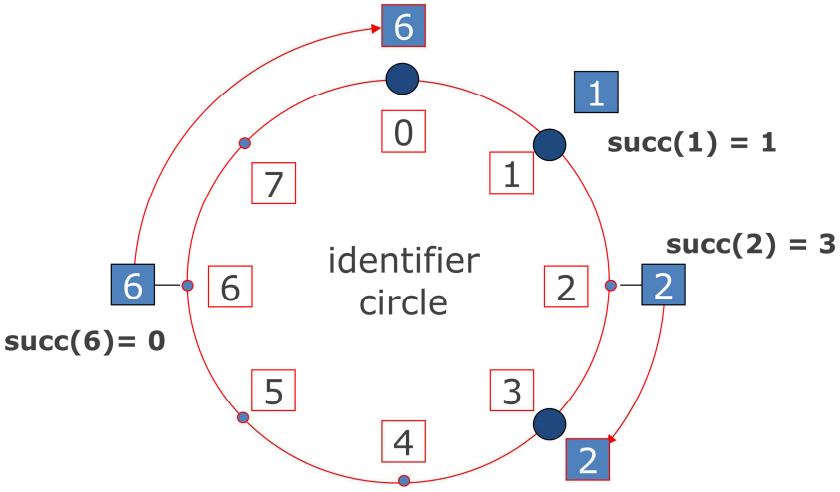


Figure 9. Architecture de Chord, [SMK+01]

La force de Chord provient de sa simplicité et de son efficacité. Ses principes fondamentaux sont :

- Adressage aléatoire : une requête de recherche est envoyée à un nœud quelconque. Le nœud ayant reçu la requête répond dans le cas où il détient la réponse, sinon il renvoie la requête à son nœud successeur et ainsi de suite.
- Afin d'éviter une recherche linéaire, Chord utilise la '*Finger Table*' de chaque nœud pour localiser le nœud dont la clé est immédiatement inférieure à celle de la clé recherchée.

- Si un nouveau nœud connait l'adresse associée à la clé recherchée, la recherche s'arrête avec succès. Dans le cas contraire, il consulte sa '*Finger Table*' et répète le processus.
- Cette organisation permet d'effectuer chaque recherche en O(log *N*) étapes, *N* étant le nombre de nœuds de l'anneau.

Afin de garantir l'efficacité de la recherche de clés dans le réseau, Chord contrôle l'adjonction et le départ de nœuds. En particulier il veille à ce que les tables de routages soient mises à jours périodiquement

1. Lors de l'adjonction d'un nœud *n* dans l'anneau Chord :
   - Le nouveau nœud *n* se voit affecter comme successeur *s* l'ancien successeur de son prédécesseur *p* dans l'ordre numérique de l'anneau.
   - Ce prédécesseur à pour nouveau successeur le nouveau nœud *n.*
   - Toutes les clés dans l'intervalle $]p,n]$ qui étaient jusqu'alors gérées par *s* se voient affectées au nouveau nœud *n*.

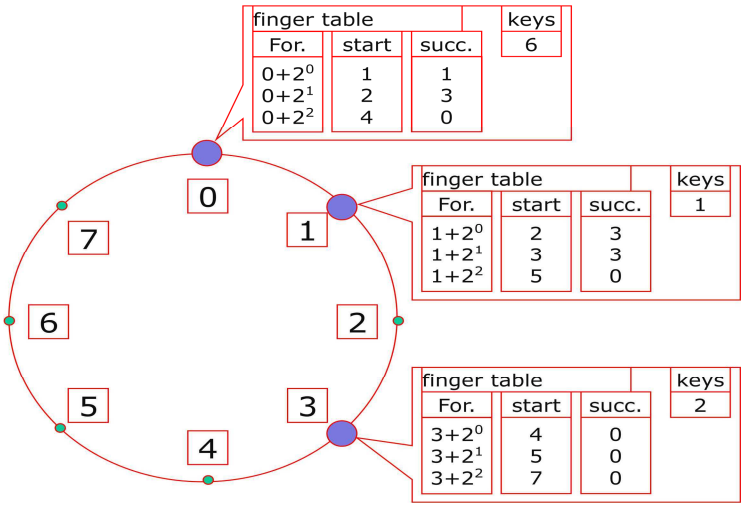


Figure 10. Tables de routages dans Chord, [SMK+01]

2. Lors du départ d'un nœud *n* de l'anneau Chord :
   - Toutes les clés du nœud *n* qui quitte le réseau sont déplacées sur le successeur de *n.*
   - Le pointeur du prédécesseur de *n* est mis à jour au successeur de *n*.
3. Lors d'une panne d'un nœud *n :*
   - Tous les nœuds de l'anneau ayant *n* dans leurs tables de routage '*Finger table*' doivent retrouver le successeur de *n* car le prédécesseur de *n* n'a plus de successeur valide (*n* en panne). Pour pallier ce problème une liste des *r* successeurs (*successor-list*) les plus proches est maintenue sur chaque nœud. Cette liste active est toujours maintenue à jour grâce à des mises à jour d'un nœud périodiques effectuées lors du départ volontaire.

- La liste des *r* successeurs permet de trouver la clé et l'adresse du successeur de nœud *n* défaillant.
- Si le nombre de successeurs de chaque nœud est O(log *N*) et la probabilité de panne d'un nœud est 1/2 , les auteurs de Chord ont montré qu'il y avait une forte probabilité que la recherche d'un nouveau successeur pour un nœud *n* après une panne renvoie le plus proche successeur disponible de *n*. De plus, l'exécution de cette recherche est d'ordre de O(log *N*)

Il faut noter que pour maintenir à jour les tables de routages, Chord nécessite O($\log^2 N$) messages, ce qui est plutôt couteux. Observons aussi que Chord ne préserve pas les données en cas de panne permanente ou de départ inopiné d'un nœud ; toutes les clés associées au nœud disparu sont perdues.

La question de la recherche à clé de données et du renvoi de messages en cas d'erreur constituent un domaine de recherche en base de données et systèmes répartis. Liskov et al du MIT ont présenté une nouvelle méthode d'organisation de l'anneau Chord pour garantir un seul renvoi en cas d'erreur d'adressage [GLR03].

B) ***Recherche en Un Saut 'One Hop Lookups'***

La recherche en un saut (OHL) [GLR03] est basée sur le système de référence Chord. Il s'agit d'une nouvelle organisation du l'anneau Chord maintient la '*Finger Table*' de Chord complète et à jour. Pour atteindre le but d'un seul renvoi en cas d'erreur d'adressage, OHL envoie à chaque nœud un nouveau message lors de chaque adjonction ou départ de nœud et garantit que ce message arrivera là tous les nœuds de l'anneau dans un délai raisonnable sans nécessiter une bande passante excessive (sans inonder le réseau).

Les principes généraux de la solution de OHL sont les suivants:

- L'anneau est réorganisé en une arborescence, ce qui permettra une recherche en un seul renvoi.
- Tous les nœuds de l'anneau ont des clés de 128 bits.
- Comme le montre la Figure 11, l'espace des 128 bits des identifiants de l'anneau est devisé en *k* intervalles réguliers et contigus appelés *Tranches* (Anglais : Slices). La $i^{ème}$ *Tranche* contient tous les nœuds dont l'identifiant est dans l'intervalle :

$$[i * \frac{2^{128}}{k}, (i+1) * \frac{2^{128}}{k}], \text{[GLR03]}$$

- Chaque *Tranche* a le même nombre de nœuds à tout moment, pour autant que la distribution des identifiants soit uniforme et aléatoire.
- Chaque *Tranche* a un *leader* qui est le successeur du nœud médian des identifiants de la *Tranche*. Tout Changement dans la composition de ces identifiants est susceptible de changer ce *leader*.
- Chaque *Tranche* est divisée à son tour en parties contiguës appelées *Units*, voir la Figure 12. Chacun de ces *Units* a également un *leader* choisi dynamiquement de la même manière que les leaders des *Tranches*.

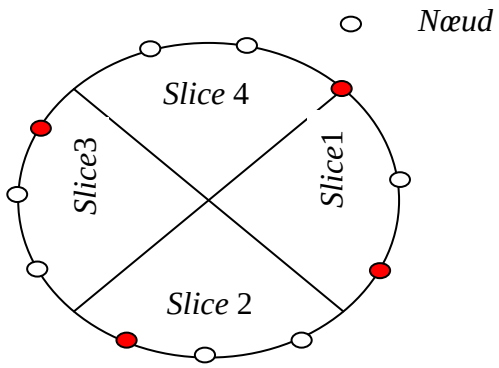


Figure 11. Découpage statique de l'anneau en *Tranches*, [GLR03]

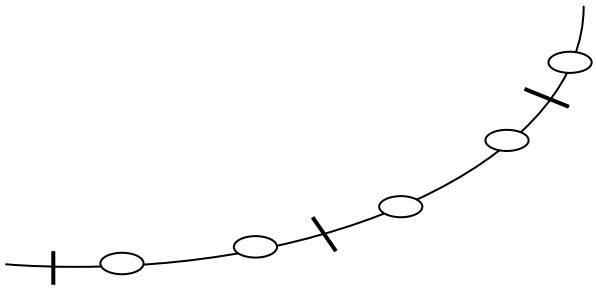


Figure 12. Découpage d'un *Tranche* en *Units*

1. Départ et Adjonction de Nœud

Avec cette nouvelle hiérarchisation de l'anneau Chord, chaque départ et adjonction de nœud est notifié par l'envoi d'un événement. Comme le montre la Figure 13, chaque nœud *n* ayant détecté un changement de son successeur (départ, panne ou arrivée d'un nouveau successeur), envoie un événement *leader* de sa *tranche*. Ce dernier collecte tous les événements de notifications d'adjonctions ou de départs de nœuds. Il prépare aussi un message de notification et l'envoie à tous les *leaders des tranches* de l'anneau. Chaque *leader de tranche* diffuse le message à ses *Units Leaders* et celui-ci le propage à son tour à ses nœuds prédécesseurs et successeurs. À la fin, tous les nœuds de l'anneau reçoivent le message de notification et mettent à jour leur *Finger Table*. Quand la composition d'une tranche évolue, il se peut que certains messages soient en doubles. Pour éviter les erreurs possibles, les nœuds devront rejeter tous les messages de notifications qui ne viennent pas du nœud voisin le plus proche.

Malheureusement, ce découpage en plusieurs *Tranches* et *Units* implique un allongement du temps de diffusion des messages de notification. En contrepartie la diffusion de ces messages est asynchrone pour mieux exploiter la bande passante du réseau

.

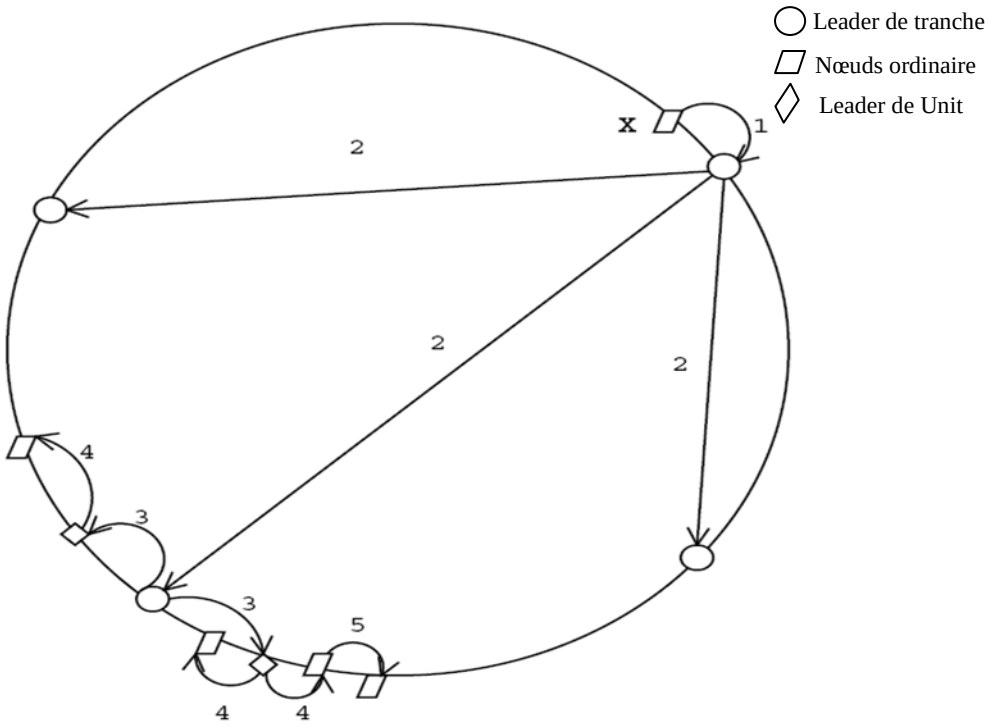


Figure 13. Diffusion d'un message de notification, [GLR03]

2. Panne d'un nœud

L'échec d'une requête ne renvoie pas d'erreur. Si un nœud n_1 envoie une requête de recherche au nœud cible n_2 et ce dernier ne répond pas alors le nœud n_1 transmet sa requête pour le successeur de n_2.

En cas de panne d'un nœud *leader* son successeur deviendra le nouveau *leader*. Dans ce cas le nouveau *leader* utilise des messages de diffusion dans sa *Tranche* ou son *Unit* pour obtenir les informations dont il a besoin.

Nous observons que OHL ne peut pas faire la reconstruction de données après une panne et se limite à la reconstruction de ses *Fingers Tables*.

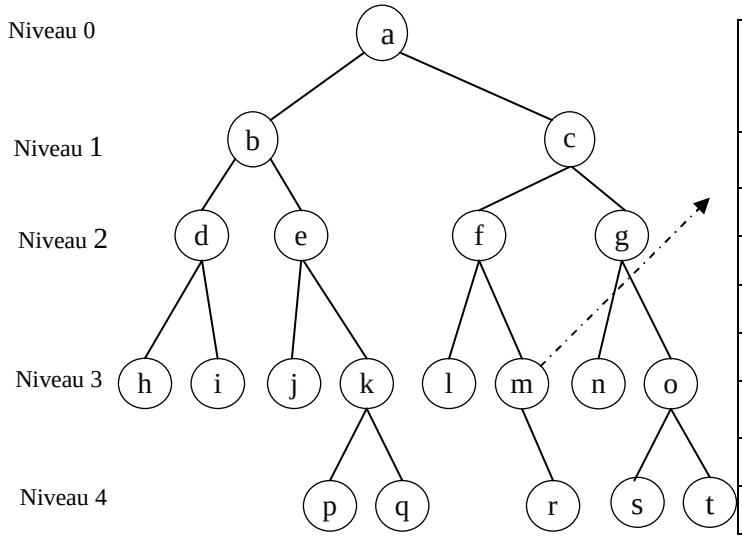
C) *BATON (A Balanced Tree Overlay Networks)*

BATON [JOV05] est la généralisation des arbres de recherches binaires équilibrés aux environnements pairs à pairs. Nous rappelons qu'un arbre binaire équilibré est un arbre binaire tel que les hauteurs des deux sous arbres de même niveau de tout l'arbre différent de 1 au plus, i.e. : | hauteur (arbre gauche) – hauteur (arbre droit) | ≤ 1.

Pour un tel arbre, le coût de l'opération de mise à jour lorsqu'un nœud quitte le réseau est de O(log *N*) messages. Le coût d'une opération d'insertion, de recherche, et de suppression d'article est de O(log *N*) messages dans le pire des cas.

Comme le montre la Figure 14, l'idée de BATON est de structurer le réseau en un arbre de recherche binaire équilibré. Chaque nœud de l'arbre correspond à un et un seul nœud du réseau. Chaque nœud a des liens vers son parent (exemple : parent=f) ses enfants (leftchild, rightchild), nœuds adjacents (leftajacent, rightajacent) et ses voisins. De cette manière, chaque nœud maintient deux tables de routage, à savoir une table de routage gauche et droite.

L'avantage de cette solution est d'accélérer les recherches dans les tables de routage car chaque acheminement consulte une seule des deux tables.



Node m : level = 3, Number = 6, Parent = f, Lefchild = null, rightchild = r, leftadjacent = f, rightadjacent = r

Left routing table

	Node	Left Child	Right child	Lower bound	Upper Bound
0	l	Null	Null	l_{lower}	l_{upper}
1	k	p	q	k_{lower}	k_{upper}
2	i	Null	Null	i_{lower}	i_{upper}

Right routing table

	Node	Left child	Right child	Lower bound	Upper Bound
0	n	Null	Null	n_{lower}	n_{upper}
1	o	s	t	o_{lower}	o_{upper}

Figure 14. Structure de BATON, [JOV05]

3.5.1.2. Adjonction d'un nœud

Un nœud *n* qui veut rejoindre le réseau s'adresse à un nœud quelconque. Celui-ci vérifie en utilisant sa table de routage s'il peut l'avoir comme enfant. Deux cas sont possibles : (i) Si c'est possible, le nœud *m* insère *n* comme enfant, met à jour sa table de routage et donne éventuellement au nouveau nœud *n* une partie de ses données. (ii) Si ce n'est pas possible, le nœud *m* diffuse la requête du nœud *n* de manière à trouver une bonne place d'insertion.

Si l'insertion affecte l'équilibre de l'arbre, BATON procède à une restructuration de l'arbre. La complexité de cette opération est de l'ordre de O(log *N*), *N* étant le nombre total de nœuds.

3.5.1.3. Départ d'un nœud

Un nœud peut quitter le réseau sans problème si son départ n'affecte pas l'équilibre du réseau. Si ce n'est pas le cas alors le nœud doit trouver un nœud qui peut prendre sa place en consultant sa table de routage. Ainsi il procède au transfert de son contenu, puis envoie un message à tous ses voisins et parents les informant qu'il quitte réseau. La complexité de cette opération est O(log *N*) car elle requière la mise à jour des tables de routage de chaque nœud voisin ou parent.

3.5.1.4. Restructuration du réseau

L'adjonction et départ de nœud peut compromettre l'équilibre de la structure arborescente du réseau. Une opération de restructuration aura alors pour but de retrouver cet équilibre. Ce processus est classique comme pour la structure d'index de l'arbre binaire. BATON utilise les rotations de gauche à droite et de droite à gauche. La Figure 15 montre le départ du nœud *g* qui cause un déséquilibre de l'arbre (à gauche de la Figure 15). Par 'rotation droite' le, nœud *k* prend la place de *a* et *a* la place de *f*, *f* la place de *c* et *c* remplace *g* qui est parti.

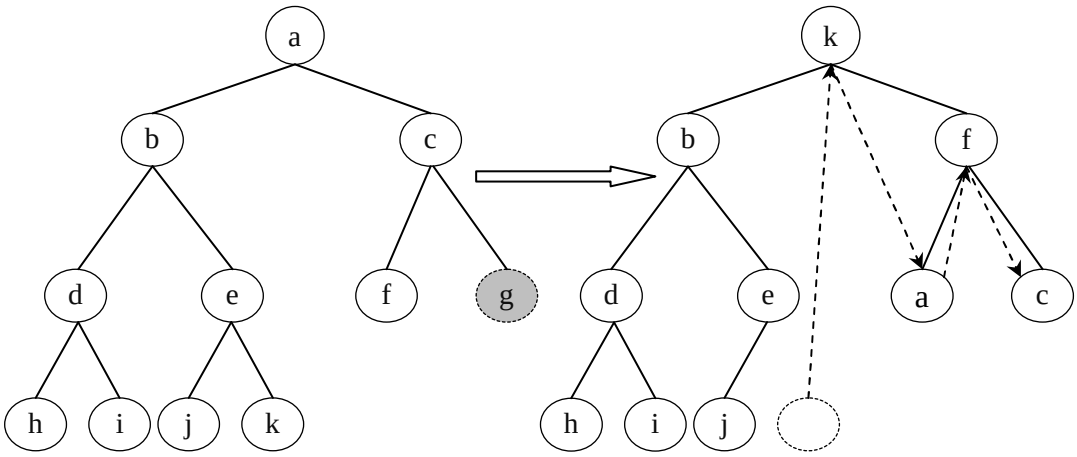


Figure 15. Restructuration suite à un départ d'un nœud, [JOV05]

Les opérations de bases telles que la recherche, l'insertion et la suppression ont un coût de O(log *N*) messages au maximum en cas d'erreur. Quant à la requête à intervalle qui consiste à rechercher les données qui sont dans un intervalle I = [clé_min, clé_max] a besoin d'un coût de O(log *N*+*X*), tel que *N* est le nombre de nœuds total du réseau et X est le nombre de nœuds impliqués dans la recherche.

3.6. RAM-Clouds

RAM-Clouds offre une alternative pour le stockage et la gestion de données en RAM autre que le système pair à pair que nous proposons. Cependant, du fait de la virtualisation des machines, de plus en plus d'applications sont disposées dans une organisation répartie de type machine virtuelle. Nous pouvons donc nous poser la question de savoir quelle serait la meilleure organisation de données pour le Cloud ?

Ousterhout et son équipe présente un système nommé RAMClouds [OAE+10]. Leur idée clé est d'utiliser les mémoires vives du Cloud pour stocker les données. Cette idée est loin d'être nouvelle. Elle a déjà été traitée de façon très détaillée depuis les années 90.

Ils présentent différentes approches pour l'organisation et le stockage de données sans toutefois spécifier une solution performante pour résoudre les problèmes de scalabilité et de haute disponibilité des données. Ainsi pour la disponibilité, ils proposent de répliquer les données, une technique à la fois couteuse et bien connue depuis les années 80. Les auteurs proposent sans toutefois implémenter d'utiliser une technique de codage tel que le découpage des données en blocs de parité (Anglais : Parity Striping) [PGK88] pour réduire le nombre de répliques.

Finalement, ils mentionnent une solution hybride appelée *Buffered logging*. Elle consiste à garder une seule copie des données en mémoire vive. Cette copie sera la copie primaire alors que les autres copies secondaires seront, elles, stockées sur les disques durs d'autres serveurs (serveurs de reprise). Notons que la mise à jour des copies sur disque n'est pas faite lors des opérations d'écriture. Dans ce cas, une information dite *log* (Anglais : Log Entries) est envoyée à chaque mise à jour de la copie primaire. Le *log* est sauvegardé dans un premier temps dans la DRAM du serveur de reprise puis transférée de manière asynchrone au disque utilisant un batch (un Buffer) comme l'indique la Figure 16. Si un serveur tombe en panne, ses données en DRAM peuvent être récupérées depuis un des serveurs de reprises en utilisant ses *logs*. Cette technique est coûteuse car elle requière de nombreux accès aux disques durs des serveurs de reprise. Elle nécessite de plus des serveurs de reprise à grande capacité de mémoire. Ce problème n'a pas échappé à l'attention des auteurs qui ont proposé des solutions de rechange sans les avoir implémentées ou analysées.

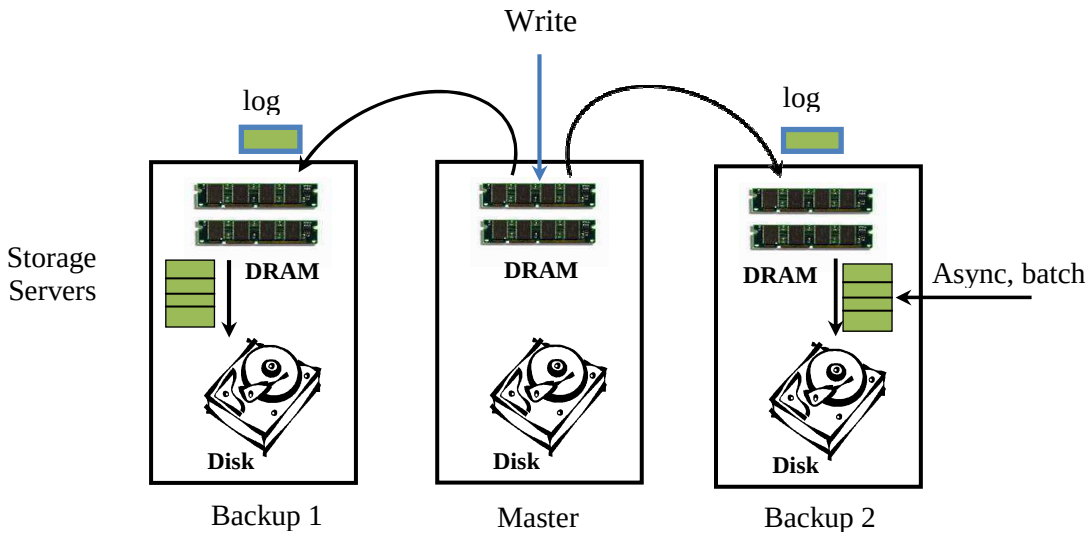


Figure 16. Serveurs de stockage et de reprise, [OAE+10]

4. Résumé

Nous avons tout au long de ce chapitre introduit le concept de SDDS. Ce concept datant des années 90 vient de refaire surface dans la communauté informatique. Cette renaissance des SDDS est propulsée par le progrès technologique dont le matériel de stockage et l'interconnexion des réseaux de type pair à pair, plus récemment Cloud. Nous avons présenté les SDDS les plus pertinents en mentionnant leurs avantages et inconvénients quant à leurs applications réelles. Dans ce qui suit nous présenterons une nouvelle SDDS à base de hachage linaire spécifiquement adaptée aux architectures pair à pair et Cloud.

Chapitre III

LH $*_{RS}^{P2P}$: Hachage Linéaire Scalable et Distribué Pair à Pair

1. Introduction

Dans ce chapitre, nous présentons une adaptation du hachage linéaire distribué LH* aux environnements pair à pair (P2P). Nous rappelons qu'en général, un nœud d'un fichier LH* est soit un client, soit un serveur soit un pair qui est le client et le serveur à la fois. Pour une utilisation P2P, nous considérons que les nœuds avec l'interface client ne sont que des pairs. Nous désignons la SDDS qui en résulte comme LH $*_{RS}^{P2P}$. Le sigle indique que notre travail réutilise en fait la variante de LH* à haute disponibilité scalable dite LH*$_{RS}$ [LMS05]. La haute disponibilité est nécessaire au traitement de Churn caractéristique des applications P2P. L'architecture fonctionnelle et notre implémentation de LH $*_{RS}^{P2P}$ réutilisent le prototype de LH*$_{RS}$ [M04]. Ce prototype a des propriétés spécifiques que nous avons dû prendre en compte pour notre propre prototype que nous décrirons dans ce qui suit.

Nous présentons d'abord les caractéristiques spécifiques à LH $*_{RS}^{P2P}$, par rapport à LH*$_{RS}$. Il s'agit surtout de l'architecture d'un pair LH $*_{RS}^{P2P}$. Le gain est la réduction du nombre maximal de renvois d'une requête à clé d'un pair. Ce nombre est d'un seul renvoi, alors que dans un fichier LH*$_{RS}$ et LH* en général, il est de deux messages. Pour rappel, la performance d'un algorithme d'adressage P2P, par exemple Chord utilisant une DHT typique sur *N* nœuds est O(log *N*). La vitesse de LH $*_{RS}^{P2P}$ en fait avec l'algorithme dans [GLR03], présenté dans le chapitre de l'état de l'art, l'algorithme *le plus rapide en vitesse de recherche par clé connus à l'heure actuelle*. Comme nous l'avions dit, l'algorithme de [GLR03] est néanmoins bien plus compliqué et couteux en messages spécifiques de maintenance de l'anneau Chord sous-jacent et ne protège pas contre le Churn.

À noter enfin, que l'adressage de LH $*_{RS}^{P2P}$ est d'ailleurs optimal dans le cadre des principes d'une SDDS et du P2P en général. En effet, dans ce cadre un nœud client ou serveur de données peut ne pas connaître l'état exact du fichier. Pour le rappeler, celui-ci peut différer de l'image du client utilisé par ce dernier pour l'adressage. Une erreur d'adressage à partir de certaines clés est alors inévitable.

Dans ce chapitre nous présentons en premier notre terminologie de base. Celle-ci comporte des différences par rapport à celle LH*$_{RS}$ et celle de LH* aussi. Ensuite nous présentons l'architecture générale d'un fichier LH $*_{RS}^{P2P}$

Puis nous discutons les algorithmes d'adressages et nous prouvons qu'ils sont corrects. Nous finissons par la présentation d'une variante de LH $*_{RS}^{P2P}$ avec une distribution d'enregistrements de parité différente de celle basique de notre schéma, hérité du prototype LH*$_{RS}$.

2. Terminologie de base de LH $*_{RS}^{P2P}$

La Figure 17 ci-après illustre cette terminologie. D'abord, tout fichier LH $*_{RS}^{P2P}$est supposé installé sur un réseau de machines, identifiées chacune par une adresse unique, celle de l'IP typiquement. Le réseau local sur lequel nous déployons notre solution est constitué de machines réelle.

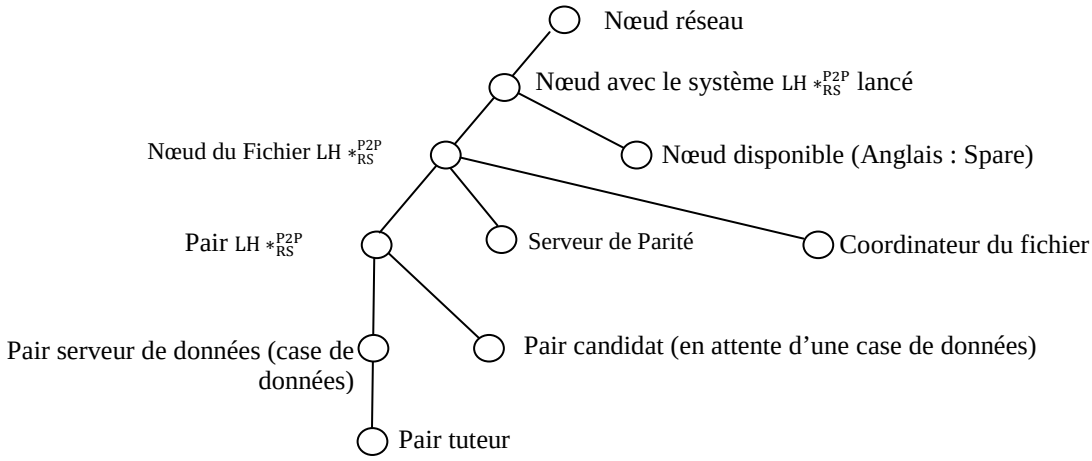


Figure 17. Terminologie de base de LH $*_{RS}^{P2P}$

Les deux composants logiciels, client et serveur peuvent néanmoins être installés sur des machines virtuelles. Ce qui est le cas en général d'un Cloud, Microsoft Azur [HLM+10] en est un exemple.

Nous appelons *nœud réseau* une machine d'un tel réseau. Un nœud réseau devient un *nœud* LH $*_{RS}^{P2P}$dès qu'une instance du logiciel de LH $*_{RS}^{P2P}$ y est lancée. Un nœud LH $*_{RS}^{P2P}$peut être alors un nœud de *fichier* ou un nœud *disponible* (Anglais : Spare). Le premier participe à un fichier LH $*_{RS}^{P2P}$, le deuxième est prêt à être utilisé ainsi et attend un message demandant un tel service. Un nœud de fichier peut être un *coordinateur*, un *serveur de parité* ou un *pair* LH $*_{RS}^{P2P}$. Un coordinateur et un serveur de parité assurent, à des détails spécifiques près, les mêmes fonctions que les ceux d'un fichier LH*$_{RS}$. Le pair LH $*_{RS}^{P2P}$est un concept spécifique à notre algorithme, comme son nom l'indique. Il comporte toujours le logiciel client interfaçant l'application et ajustable par les IAMs. Ce logiciel est analogue à des fonctions spécifiques près, à celui de LH*$_{RS}$. En même temps, le pair comporte le logiciel serveur en charge, des données de l'application stockées dans une *case de données* du fichier,

sur le pair. En général, le pair comporte aussi une case, mais pas nécessairement. Dans le premier cas nous le désignons comme pair serveur (de données d'application). La propriété caractéristique d'un pair serveur est qu'un éclatement de la case du pair déclenche un ajustement synchronisé de l'image client du pair par un IAM interne. Cette propriété est inconnue de toute autre variante de LH* et LH*$_{RS}$ notamment.

La Figure 18 montre plus en détails l'architecture d'un pair LH $*_{RS}^{P2P}$et la correspondance de celle-ci avec la terminologie de base. Un pair qui n'a pas de case de données est un pair *candidat*, prêt à accepter les données résultant d'un éclatement. Tout candidat est originellement un nœud disponible, que le coordinateur vient d'enregistrer comme pair. Un fichier LH $*_{RS}^{P2P}$ est supposé comporter à tout moment au moins un pair candidat, afin de pouvoir s'étendre efficacement. Tout pair *candidat* et seulement un tel pair, échange des messages spécifiques avec un certain pair serveur dit *tuteur* de ce candidat. Le candidat est alors un *pupille* du tuteur. Un pair tuteur a au moins un pupille et peut avoir plusieurs pupilles. Le tuteur notifie immédiatement à chacun de ses pupilles par un IAM tout éclatement auquel il vient de procéder. Tout pupille a un et un seul tuteur. Tout tuteur est choisi par le coordinateur au moment de l'enregistrement du candidat. Le pupille ayant reçu une case de la part du coordinateur, à travers le message dit ANDB (Anglais : Add New Data Bucket), cesse d'être un pupille en avertissant immédiatement son tuteur. Le pupille peut aussi se déconnecter et avertir son tuteur. Un tuteur n'ayant plus de pupilles redevient seulement un pair serveur. La Figure 19 illustre le diagramme de différents états d'un pair LH $* _{RS}^{P2P}$.

Nous détaillons cette terminologie dans ce qui suit. Néanmoins, nous insistons sur le fait que le pair LH $*_{RS}^{P2P}$ n'est pas celui de LH*$_{RS}$ hérité de LH*. Ces derniers sont simplement des nœuds, soit client ou serveur de données du fichier. Il n'y a pas d'IAMs internes, ni de concept de pair candidat ni de tuteur avec ses pupilles.

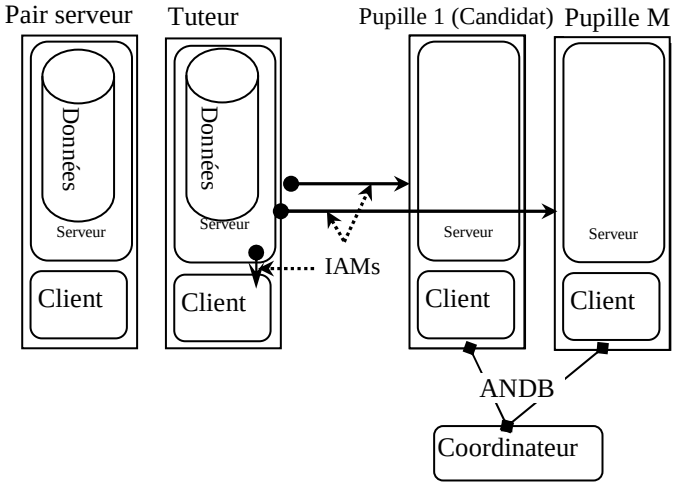


Figure 18. Architecture d'un pair

Nous supposons dans ce qui suit que tout pair serveur de données ne gère qu'une seule case. La possibilité d'avoir plusieurs cases correspond à l'équilibrage de charge analysée en premier par Vingralek dans [VBW94]. À titre d'exemple cette analyse est présente aujourd'hui dans Windows Azure [HLM+10]. Dans notre cas, nous laissons cette variante pour l'analyse future.

Dans notre terminologie nous ne considérons pas de *fusions*. Pour rappel, une fusion de cases de données dans un fichier LH* a lieu quand il y a un taux inférieur au taux de remplissage[1] minimum d'un fichier de plusieurs cases. Si les suppressions baissent le taux sous ce seuil, le coordinateur fusionne la dernière case du fichier, donc la case *N*-1, avec sa case *mère*. Nous déplaçons alors les données de la case fille et nous supprimons celle-ci. Les fusions améliorent ainsi le taux de remplissage. Dans le cas de LH $*_{RS}^{P2P}$, un pair serveur redeviendrait à la suite d'une fusion un pair candidat. Nous ne traiterons pas les fusions car elles sont rares en pratique.

En principe, tout coordinateur LH $*_{RS}^{P2P}$ n'est que sur un nœud. Néanmoins, il peut aussi être répliqué comme en général pour LH*. Enfin, comme en général pour LH*, nous pouvons probablement concevoir une variante de LH $*_{RS}^{P2P}$ sans coordinateur. Nous laissons cette question intéressante pour l'analyse future.

Enfin, nous rappelons que par définition d'un pair. Un pair LH $*_{RS}^{P2P}$ quand il a une case de données agit comme un serveur quand il répond aux requêtes de manipulations de données produites par l'application. Il agit comme un client quand il envoie des requêtes. Nous détaillerons progressivement ces deux fonctions.

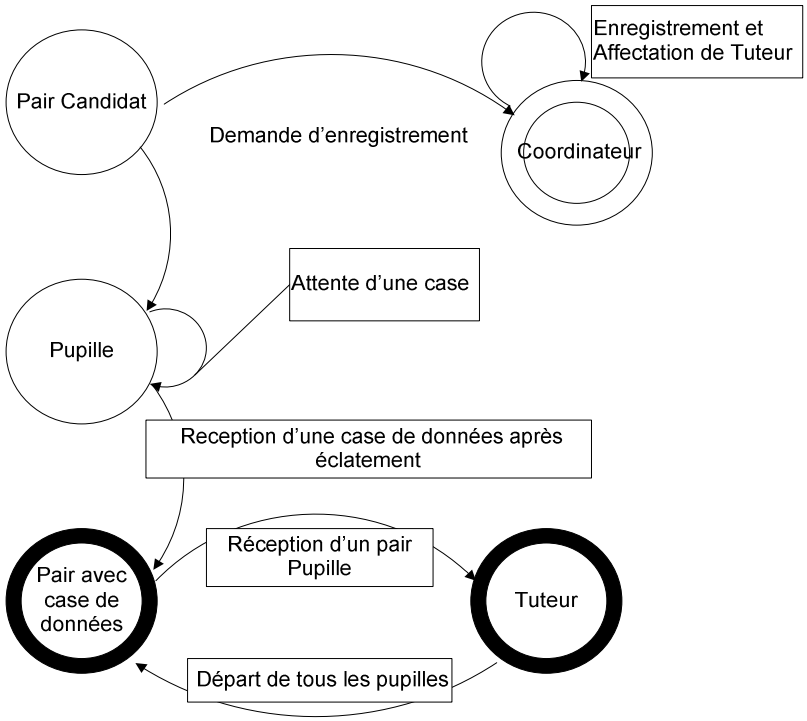


Figure 19. Différents états d'un pair LH $*_{RS}^{P2P}$

[1]Taux de remplissage : calculé par la formule, **f** = **x** / (**n*****b**) où **x** est le nombre d'articles insérés, **n** est le nombre de cases du fichier et **b** la capacité de la case.

3. Structure du Fichier LH $*_{RS}^{P2P}$

Les figures ci-après, Figure 20, Figure 21, Figure 22, présentent la structure de tout fichier LH $*_{RS}^{P2P}$. Un fichier LH $*_{RS}^{P2P}$ contient d'abord des enregistrements *de données*, identifiés chacun par sa clé primaire et comportant aussi chacun une zone non-clé (couple clé – donnée) comme le montre la Figure 22 (a). La clé et les données non-clés sont définies par l'application. Tout enregistrement de données est stocké dans une case *de données* sur un pair serveur, dite case *primaire* pour la clé, ou dans une zone *de débordement* de la case, sur le même pair. Toute case primaire a une *capacité* de stockage de *b* enregistrements, les enregistrements en surnombre sont mis dans la zone de débordement de la case. Les cases sont numérotées 0, 1, 2… Chaque case réside sur un pair (serveur) différent. Le nombre courant de cases noté *N* est dit *étendue* ou *taille* (en nombre de cases) du fichier.

À chaque enregistrement de données est associé $k \geq 1$ enregistrements de *parité*, Figure 21, Figure 22. Tout enregistrement de parité est dans une case de parité. Toute case de parité est sur un serveur différent. La valeur de *k* est initialement $k = K$ où *K* est un paramètre du fichier dit *niveau de (haute) disponibilité* de celui-ci. Ce niveau est fixé à la création du fichier qui est dit alors *K-disponible*. La valeur de *k* peut ensuite augmenter progressivement avec l'étendu (nombre de cases de données) du fichier suite aux éclatements des cases de données. Dans ce cas, nous parlons alors de (haute) disponibilité *scalable*, [M05]. Dans ce cas, nous pouvons avoir $k = K$ ou $k = K + 1$ selon les cases de données [YS10].

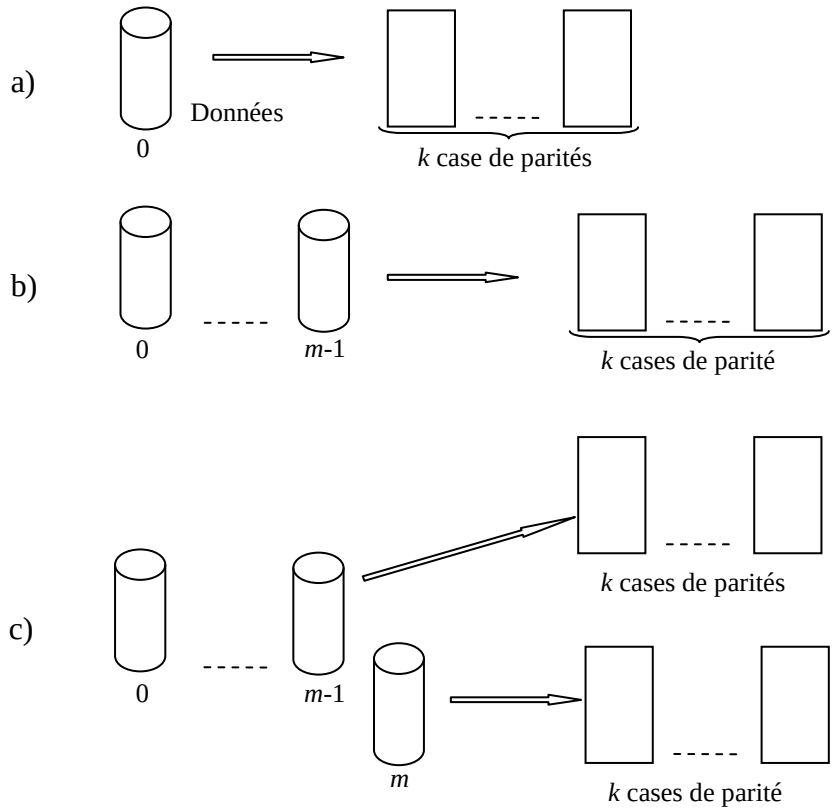


Figure 20. Structure d'un Fichier LH $*_{RS}^{P2P}$ en Groupe de Parité

L'association d'enregistrements de données avec ceux de parité est liée au concept du groupe de parité, illustré par la Figure 20, Un tel groupe consiste en *m* = 2,3… cases de données successives associées à *k* cases de parité, [LMS04]. Ces cases de parité ne sont associées qu'à leurs cases de données. La Figure 20 montre un fichier structuré ainsi en groupes de parité de *m* cases de données avec *k* cases de parités. Les deux paramètres sont définis à la creation du fichier, avec *k* = *K* comme nous l'avons présenté plus haut. Les premières *k* cases de parité sont allouées à la création de la case 0, Figure 20 a). Un groupe avec moins de *m* cases de données est alors *incomplet*. Cette situation persiste jusqu'à *N* = *m*. Le 1er groupe de parité devient alors *complet*. Ensuite, à la création de la case de données *m*, le coordinateur commence un nouvel groupe de parité, en lui allouant son propre ensemble de *K* cases de parité. Ainsi, en créant le 2ème groupe incomplet jusqu'à l'allocation de la case de données $2m - 1$, Figure 20 b). L'*étendu* (taille) *m* du groupe en principe ne change plus pendant l'expansion du fichier. Par contre, la valeur de k peut augmenter comme nous le presentons dans la Section 9, ici-bas, avec le nouveau schéma de disponibilité scalable.

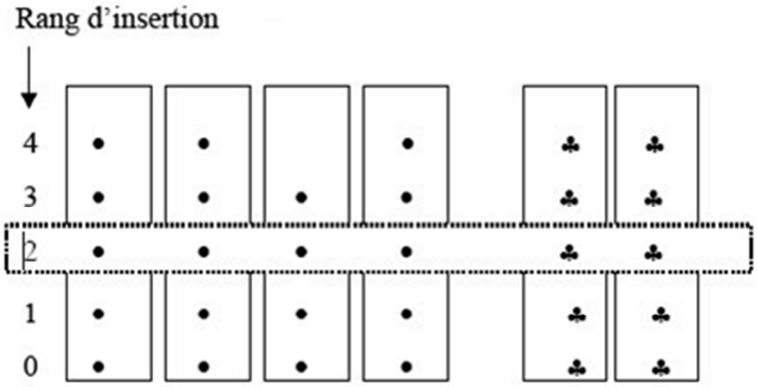


Figure 21. Un groupe de parité avec m = 4 et k = 2

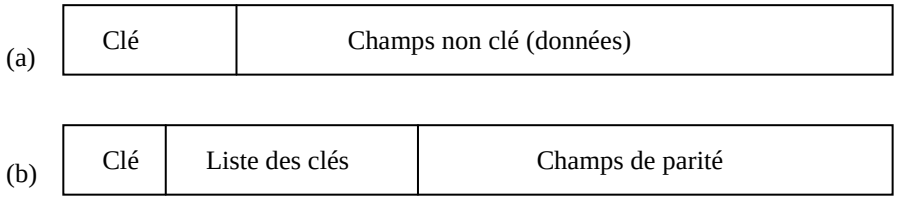


Figure 22. Enregistrement du fichier LH $*_{RS}^{P2P}$

Les enregistrements de données dans les cases de données et ceux de parité d'un groupe sont liés comme suit, Figure 21. D'abord, le serveur attribue à tout enregistrement de données un rang séquentiel noté r où $r = 1,2, \ldots$, Figure 21. Celui-ci définit un numéro d'ordre (position) unique dans la case de données, $(r = 1, 2, \ldots)$. Le rang est d'abord attribué au moment de l'insertion de l'enregistrement. Un rang est aussi attribué à chaque enregistrement de parité, également au moment de sa création. Les rangs sont recalculés au moment de l'éclatement selon la nouvelle distribution des enregistrements. Ainsi les rangs sont incrémentés de 1. Ensuite, un segment de parité de rang *r* contient tous les enregistrements de

données ayant le même rang dans un groupe de parité. Il contient aussi *k* enregistrements de parité du même rang donc nous détaillons la structure ci-après. Ainsi, la Figure 21 montre tout particulièrement le segment avec le rang $r = 2$, dans un groupe de parité avec $m = 4$ et $k = 2$.

La Figure 22, montre la structure d'un enregistrement de parité, [LS00]. Tout enregistrement de parité a le rang *r* comme clé. Ensuite un champ dit liste des clés contient toutes les clés des enregistrements de données composant le segment *r*. Par exemple le segment $r = 2$ sur la Figure 21, contient quatre enregistrements de données donc ce champ contiendra les quatre clés correspondantes. Ceci n'est pas le cas du segment *r = 4* sur la figure, composé de trois enregistrements de données seulement.

L'enregistrement de parité contient enfin un 3ème champ dit *de parité*. La valeur de celui-ci est calculée en utilisant les codes Reed Salomon, à partir des champs non-clé des *m* enregistrements de données du segment. Les valeurs du champs clé et de la liste des clés sont identiques pour tous les *k* enregistrements du segment. Les champs de parité sont différents. Le résultat global est que jusqu'à *k* enregistrements du segment, ceux de données ou ceux de parité, peuvent devenir indisponibles sans perte de données du segment. De même, il peut s'agir de *k* cases d'un groupe de parité. D'où le concept de *k*-disponibilité d'un groupe de parité. Les détails du calcul de parité sont illustrés par Litwin et Schwarz dans [LS00]. La récupération des données suite à une indisponilité est garantie par les enregistrements de parités.

4. Adressage dans un Fichier LH $*_{RS}^{P2P}$

Cette section présente l'adressage de base des enregistrements dans un fichier LH $*_{RS}^{P2P}$. Cet adressage dans LH $*_{RS}^{P2P}$ est basé sur celui de LH*$_{RS}$ que l'on suppose en principe connu du lecteur. Il y a néanmoins des aspects spécifiques que nous présenterons. Nous rappelons que chaque requête initiée par l'utilisateur depuis l'application installée sur le client d'un pair porte un identifiant unique. Ainsi, les composants clients et serveurs identifies l'adressage adéquat à exécuter. Il faut noter que dans ce qui suit nous ne traitons pas du cacul de parité.

4.1. Expansion d'un Fichier

L'expansion du fichier détermine l'emplacement de chaque enregistrement de données ainsi que de ceux de parité dans le fichier. Ceci-ci est assurée par des éclatements de cases de données. Dès qu'un nombre d'enregistrements à insérer dans une case de données dépasse *b*, le serveur signale le débordement au nœud coordinateur. Celui-ci envoie le message d'éclatement au pair avec la case désignée par le pointeur d'éclatement. La valeur du pointeur est en général notée *n* pour un fichier LH*. La case *n* qui éclate est en général différente de celle qui déborde. Le coordinateur utilise aussi un autre paramètre dit niveau du fichier et noté d'habitude *i*. Le couple (n, i) constitue l'état du fichier. L'état est stocké sur le coordinateur et géré par celui-ci exclusivement. Il résulte des principes de base de LH*. Ainsi, initialement le fichier n'a que la case 0 et nous avons $n = i = 0$. Ensuite, tout éclatement utilise pour tout enregistrement la case *n* qui éclate, quel que soit *n*. La fonction de hachage spécifique utilisée est dite de hachage linéaire et notée en général $h_{i+1}(C) = C \ mod \ 2^{i+1}$, où *C* désigne une clé

d'un enregistrement de données. Le résultat de ce calcul est qu'en moyenne un enregistrement sur deux de la case *n*, qui éclate, reste dans celle-ci. Un enregistrement sur deux reçoit la nouvelle adresse qui est celle de la case $n + 2^i$. C'est en fait l'adresse de la nouvelle case allouée au fichier. Il s'agit toujours de la case *N* où *N* est l'*étendue* du fichier, avant l'éclatement. Tout enregistrement ayant une nouvelle adresse est déplacé dans la case *N* en conséquence. Après l'éclatement, le coordinateur met à jour l'état du fichier par la formule caractéristique de LH* :

$$n := n + 1 ; si\ n = 2^i\ alors\ n = 0\ et\ i = i + 1$$

Nous pouvons aisément voir que le fichier s'étend progressivement sur les cases successives. Ainsi, il contient successivement les cases de données : 0 ; 0,1 ; 0, 1, 2 ; 0…N – 1 Figure 20 c).

À des fins d'adressage correct des requêtes à clé décrites plus loin dans la Section 4.4 tout pair maintient sur son composant serveur un paramètre dit *niveau* et noté *j*. Pour une case ayant subi un éclatement, *j* est égal à l'indice (niveau) de la fonction de hachage linéaire utilisée lors du dernier éclatement. Donc $j = i + 1$. La case qui vient d'être créée par un éclatement reçoit ce *j* aussi comme valeur initiale. Enfin, pour la case 0, on a initialement $j = 0$. Plus précisément, pour toute case *n*, chaque éclatement commence en fait par $j = j + 1$.

La migration d'enregistrements de la case *n* durant son éclatement conduit à la réorganisation des segments de parité associés, [LMS04]. Tout enregistrement, restant ou partant, en général change de rang donc de segment. Les enregistrements qui migrent, peuvent changer en plus de groupe de parité. Comme présenté aussi ci-dessus, toute création d'une case de données $m, 2m, ...$ s'accompagne aussi de celle de *k* nouveaux serveurs de parité (*k* cases de parité) pour le fichier.

Puis, dans un fichier LH $*_{RS}^{P2P}$ spécifique, le composant serveur du pair serveur de la case *n*, envoie après l'éclatement de sa case l'IAM dont nous avons parlé plus haut, à son composant client. Cet IAM contient le couple (i, n) connu du serveur. Après l'éclatement de toute case *n* et avant l'éclatement suivant, cette valeur est celle de l'état du fichier sur le coordinateur. Le serveur de toute case *n* connait donc pendant tout éclatement de celle-ci cet état exactement et uniquement en ces moments. L'image client de l'état du fichier, maintenue par son composant client peut alors devenir égale à l'état du fichier à coup sûr. Toute requête à clé émise par ce client est alors adressée sans erreur donc sans renvoi possible par le serveur la recevant. Ceci jusqu'au prochain éclatement subi par le fichier. Le moment d'éclatement de celui-ci est toutefois inconnu du pair discuté. Il ne sait pas aussi à quel moment son image client cesse d'être correcte. Ces propriétés sont cruciales pour notre schéma.

Si le pair *n* est un tuteur, il envoie cet IAM aussi à chacun de ses pupilles. Enfin, l'éclatement peut également transmettre cette fonction au nouveau pair. Le pair *n* cesse alors d'être le tuteur et le nouveau hérite de tous ces pupilles. Nous détaillons la gestion des éclatements tout le long de ce chapitre.

4.2. Adressage Global d'un Enregistrement à Clé

Soit *C* la clé d'un enregistrement d'un fichier LH $*_{RS}^{P2P}$. Notons $a(C); a = 0,1, ...$ le numéro de la case dans laquelle l'enregistrement devrait être stocké. Alors $a(C)$ est l'*adresse* (*logique*) *correcte* de *C* dans le fichier. Elle résulte du hachage linaire de *C* en fonction de l'état en cours du fichier (i, n). C'est Algorithme A dans l'encadré ci-dessous qui définit $a(C)$ pour tout *C* et tout état d'un fichier LH $*_{RS}^{P2P}$. Nous disons que Algorithme A définit *l'adressage global* des clés dans un fichier LH $*_{RS}^{P2P}$. Il est en fait le même que pour un fichier LH* basique (générique). Comme nous l'avons présenté, la fonction de hachage h_i la plus usitée est le hachage par division classique, soit $h_i\ (C) = \ C\ mod\ 2^i$.

Algorithme A – Adressage global dans un fichier LH $*_{RS}^{P2P}$
$\boldsymbol{a \leftarrow\ h_i(C)}$;
Si $\boldsymbol{a\ <\ n}$ ***Alors***
    $\boldsymbol{a\ \leftarrow\ h_{i+1}(C)}$ ***;***
Fin Si

4.3. Adressage sur le Composant Client

Comme pour les fichiers LH $*_{RS}^{P2P}$et LH*, seul le coordinateur d'un fichier LH $*_{RS}^{P2P}$connait en permanence (i, n). Dans le cas d'un fichier LH $*_{RS}^{P2P}$ nous distinguons deux types de requêtes. Une requête de premier type est dite *simple*. Cette section décrit de telles requêtes. Une requête de second type est dite *sûre*. Une requête sûre protège davantage contre le Churn au prix d'un temps de traitement plus long. Nous aborderons les requêtes sures dans la Section 6. Toute requête simple ou sûre peut être de recherche, d'insertion, de mise à jour ou de suppression d'un enregistrement étant donnée sa clé. Il peut s'agir aussi de scans, présentés dans la Section 5.

Soit *Q* la requête à clé simple. Comme le montre la Figure 23, le composant client de tout pair maintient dès lors une donnée dite *image client*. L'image est un couple noté(i', n') où $i' = 0,1 \ldots, n' = 0,1, \ldots$. Nous avons $(i', n') = (0,0)$ initialement, ces valeurs peuvent être mises à jour lors de la réception d'un IAM par le client. Nous montrerons en détail dans ce qui suit ce processus. Le client du pair qui envoie *Q* dit alors pair client ou le client simplement, l'accompagne de son image. Le message portant *Q* contient aussi un booléen *R* dit renvoi que le client initialise à 'False'. Ce qui veut dire pour le pair recevant *Q* qu'il s'agit de l'envoie par le client, pas d'un renvoi de *Q* par un pair. Le client envoie le tout au serveur du pair noté $a'(C)$.Cette valeur est l'adresse logique donnée par Algorithme A1 ci-dessous, calculée par le client envoyant *Q*. Nous constatons qu'il s'agit en fait d'Algorithme A, appliqué toutefois à (i', n') au lieu de (i, n). L'adresse $a'(C)$ est ainsi une approximation par le client de l'adresse correcte de *Q*. Comme nous le savons, il se peut que $(i', n') \neq (i, n)$, cette diférence conduisant alors à $a'(C) \neq a(C)$.

Algorithme A1 – Adressage côté pair client dans un fichier LH $*_{RS}^{P2P}$

$a' \leftarrow h_{i'}(C)$;
Si $a' < n'$ ***Alors***
    $a' \leftarrow h_{i'+1}(C)$;
Fin Si

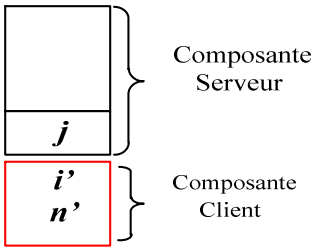


Figure 23. Paramètre d'Adressage d'un pair LH $*_{RS}^{P2P}$

Exemple 1

Supposons que le fichier est constitué d'une seule case de données logée sur le pair P_0. Donc l'état du fichier (sur le coordinateur) est $(i, n), = (0,0)$, l'image du fichier au niveau du composant client de P_0 est $(i', n') = (0,0)$ aussi. L'insertion de la clé 8 doit être calculée suivant A1 qui donnera l'adresse de la case de données *a'*, telle que $a' = 8 \, mod \, 2^0 = 0 \; et \; a' = n'$, donc la clé 8 sera envoyée à la case 0 pour insertion.

4.4. Adressage sur le Composant Serveur

L'image (i', n') d'un client peut être *incorrecte* au sens d'être différente de l'état du fichier. Ceci arrive à la suite de l'expansion du fichier, inconnue du client. Le résultat peut être $a'(C) < a(C)$, [Y08, YS10]. Nous disons alors *a* est une adresse correcte tel que connu par le coordinateur et que *a'* est une adresse *incorrecte* pour *Q*. Le pair *a'* étant incorrect pour *Q* naturellement aussi. Le pair *a'* commence par vérifier s'il est correct donc si $a' = a$, ou n'en est pas. Le pair utilise à cette fin Algorithme A2 ci-dessous.

Algorithme A2 vérifie d'abord si *R =True*. Alors le pair est nécessairement le pair correct pour *C*, c.-à-d. le pair $a(C)$. Donc le pair traite *Q*. Autrement, le pair calcule *l'image* du niveau de la case *a'* résultant de l'image client ayant envoyé *Q*. Nous notons *j'* cette image. Si nous avons $j' = j$, alors nous avons aussi $a' = a$. L'adresse *a'* est correcte pour *C* donc pour *Q*. Sinon l'algorithme vérifie si $j - j' = 1$. Sinon, il y a une erreur d'image du client que le pair *a'* ne peut pas toujours corriger. Il avertit alors le client de celle-ci et ne traite pas *Q*. Si oui, alors le pair calcule l'adresse $h_j(C)$. Celle-ci est nécessairement l'adresse correcte $a(C)$, comme nous le démontrons bientôt dans ce chapitre. Nous la notons *a* dans A2. Il se peut que $a = a'$. Dans ce cas le pair exécute *Q*. Il se peut aussi que $a > a'$. Pair *a'* met alors *R* à *True* et renvoie *Q* avec, au pair *a*.

Le pair recevant *Q* renvoyé exécute A2 de nouveau. La valeur de R sera alors *True* et le pair traitera *Q*.

Algorithme A2 et spécifique à LH $*_{RS}^{P2P}$. L'algorithme général de calcul d'adresse sur le serveur *LH* $*$ s'applique néanmoins aussi. Il est cependant moins rapide. La raison en est le calcul systématique de la fonction de hachage linéaire. Celui-ci est plus complexe que les simples comparaisons dans A2, suffisantes en général. Nous discuterons cet aspect plus en détail plus dans la Section 8.

Le pair *a* recevant *Q* renvoyée, envoie après le traitement de *Q*, un IAM au pair client. L'IAM est envoyé avec la réponse éventuelle à *Q* au composant client. L'IAM contient le niveau de la fonction de hachage *j*. Le pair joint à l'IAM son adresse IP qui peut ne pas être encore connu du pair client.

L'envoi de l'IP en plus de l'IAM n'existe pas sous LH*$_{RS}$ ni LH*. Il oblige tout pair recevant un IAM de rafraichir sa liste des adresses physiques des bonnes cases de données en faisant une demande au coordinateur, action faite par l'utilisateur via l'application client. Toutefois, un pair a toujours besoin de rafraichir sa liste des adresses physiques dans le cas où il n'est pas au courant des éclatements.

Algorithme A2 – Vérification d'adressage de Q et Renvoi éventuel
Si R =True Alors ***/* R est un booléen initialisé à False*/***
    Exécuter Q ;
Sinon
    Si $a' \geq n'$ ***Alors***
        $j' = i'$***;***
    Sinon
        $j' = i' + 1$***;***
    Fin Si
    Si $j = j'$ ***Alors***
        Exécuter Q;
    Sinon Si $j - j' = 1$ ***Alors***
        $a \leftarrow h_j(C)$***;***
    Fin Si
    Si $a \geq a'$ ***Alors***
        R = False ;
        Renvoi de Q vers le pair a; Exit;
    Sinon
        Envoi d'un message d'Erreur d'Image ;
    Fin Si
Fin Si

Exemple 2

On suppose le même cas cité dans l'exemple 1. Le pair P_0 qui reçoit la demande d'insertion de la clé 8 exécute l'algorithme A2. Ainsi, la variable booléenne est initialement à *False*. Donc $a' = n' = 0$, $j' = i' = 0$. En suite P_0 vérifie si son niveau de fonction de

hachage *j* est égal à celui sur le pair client ayant envoyé *Q*. Ainsi $j = j' = 0$ alors P_0 exécute *Q* et insère la clé 8 dans sa case de données 0.

4.5. Ajustement d'Image du Client

Le client du pair recevant l'IAM met à jour son image (i', n')en exécutant l'algorithme A3 ci-après. Cet algorithme est utilisé par ailleurs pour mettre à jour l'image lors d'un éclatement de la case du pair. Si le pair est aussi un tuteur, ses pairs pupilles exécutent A3 aussi.

Algorithme A3 – Ajustement d'image client
$\boldsymbol{i' = j - 1;}$
$\boldsymbol{n' = a' + 1\ ;}$
Si $\boldsymbol{a' = 2^{i'}}$ ***Alors***
    $\boldsymbol{i' = i' + 1\ ;}$
    $\boldsymbol{n' = 0\ ;}$
Fin Si

5. La Recherche Scan

La recherche scan est un type de recherche différent de la recherche à clé d'un d'enregistrement qui utiliser l'algorithme A2 pour l'adressage. La première a pour but la rechercher sur tout le fichier et la seconde a pour but la recherche ciblée. Néanmoins la recherche scan utilise l'algorithme A2 pour que le pair serveur détermine si la requête de type scan lui est adressée. Quant à la recherche sûre, elle permet de chercher un enregistrement sur tout le réseau sachant l'ensemble des pairs depuis son image. La requête scan se propage ainsi des pairs connus aux pairs inconnus comme nous le présentons dans cette section.

Pour effectuer un scan, soit *S*, nous supposons que le (pair) client utilise en général des messages unicast. Dans des configurations spécifiques, par exemple de pairs sur un même segment d'un réseau local, un multicast peut servir néanmoins. Le client envoie un message unicast à tout (pair) serveur dans son image, c.-à-d. à tout serveur d'une case *a*. D'éventuel pairs hors image, c.-à-d. tout serveur de case $a' = N' \ldots N - 1$, ne reçoivent alors pas ces messages. Les serveurs de *S* parmi ceux dans l'image assurent alors des renvois de *S* vers ces pairs. À cette fin, d'abord chaque message du client contient le niveau présumé *j'* de la case. Chaque serveur *a* de *S* effectue le scan demandé. En même temps, il vérifie si *S* doit être renvoyée aussi. Si c'est le cas il renvoie *S* vers un et un seul autre pair *a'*. L'algorithme A4 exécuté sur chaque serveur *a* définit tout ce traitement et détermine notamment pour tout serveur *a* l'adresse *a*'. Comme nous le démontrons plus loin, le résultat en est : quels que soient l'image et l'état du fichier, tout serveur dans le fichier reçoit *S* et ne la reçoit qu'une fois.

Algorithme A4 – Recherche de type Scan
Si $\boldsymbol{j = j' + 1}$ ***Alors***
    $\boldsymbol{a' = a + 2^j - 1;}$
    Exécuter S ;

Transférer S au pair a';
Exit ;
Fin Si
Si j = j' Alors
Exécuter S;
Exit;
Sinon
Envoi d'un message d'erreur d'image au pair qui a envoyé S ;
Fin Si

Le pair client termine *S* par le *protocole de terminaison*. Tout protocole de terminaison de LH* s'applique à notre solution. Pour rappel deux protocoles principaux sont définis pour LH* :

1. Protocole probabiliste : un timeout *T* est réinitialisé après chaque réponse reçue. Le client considère *S* terminé quand *T* expire.
2. Protocole déterministe : le pair qui envoie *S* attend jusqu'à ce que tous les pairs aient répondu à *S*. Nous ne décrivant pas ce protocole de plus amples détails sont présentés dans [LNS96]. Ce protocole est valable pour toutes les variantes de LH*.

Algorithme A4 est spécifique à LH $*_{RS}^{P2P}$. Notamment il implique un seul renvoi de tout message. Rappelons qu'un scan dans un fichier LH* ou LH*$_{RS}$ peut conduire à plusieurs renvois par contre. Le scan dans le fichier est dans ce sens plus rapide. Algorithme A4 est aussi plus rapide en exécution. La différence devrait être néanmoins d'une importance marginale en pratique. Cette affirmation reste néanmoins à être appuyer par une expérimentation que nous laissons comme perspective d'un travail futur.

Le Churn peut conduire un scan à trouver une indisponibilité d'un ou de plusieurs serveurs. Le client peut détecter indisponibilité à l'envoi d'un message ou à la terminaison déterministe. La formulation d'Algorithme A4 ne tient pas compte de Churn. Dans tous les cas, le client traite l'indisponibilité comme pour toute autre requête.

6. La Requête Sûre

Une requête sûre protège davantage contre le Churn que celle simple. En effet, le Churn conduit à une probabilité relativement élevée d'une inaccessibilité temporaire d'un pair serveur, disons le pair *A* ayant pour adresse physique *A*, avec la case de données notée *a*. Néanmoins, cette inaccessibilité peut être détectée par une requête. La case *a* est alors considérée comme étant en panne et est reconstruite sur un autre pair, disons le pair *A'*. Le calcul de parité durant la reconstruction suit les principes de LH*$_{RS}$ [LMS05]. Ceci, y compris toute (méta) donnée dans la case particulière à LH $*_{RS}^{P2P}$, l'enregistrement de tutorat contenant la table des pupilles d'un pair tuteur par exemple, voir Section 7.2. Un pair client apprend la nouvelle adresse *A'* de la case en différé, comme pour LH*. Ceci, par un message IAM ou par l'interrogation du coordinateur. Avant toute mise à jour l'adresse de la case *a* chez le client est toujours *A*.

Le pair *A* peut-être temporairement inaccessible pour plusieurs raisons. Il peut s'agir d'une déconnexion et une reconnexion volontaire. Il peut s'agir d'une coupure réseau dont le serveur n'était même pas au courant. Le client *A* peut alors redevenir accessible ayant alors une copie de la case *a*, sans le savoir. Celle-ci peut ne pas être à jour à l'évidence. Un client avec l'adresse *A* peut néanmoins diriger sa requête sur le pair *A* ayant des données périmées. Une requête simple serait alors traitée et une réponse avec une erreur transparente pour le client et le serveur pourrait survenir.

Ce cas ne peut pas se produire si la requête est une requête sûre. Elle comporte alors un traitement complémentaire que nous décrirons dans ce qui suit. Nous présentons d'abords l'algorithmique correspondante. Nous prouvons ensuite qu'elle atteint son objectif. Notons que notre prototype est le premier à offrir ce type de requêtes et la protection correspondante, à notre meilleure connaissance de la littérature.

Comme nous l'avons présenté au début de ce chapitre, toute manipulation d'un fichier LH $*_{RS}^{P2P}$ peut donner lieu à une requête sûre. Nous restreignons la présentation aux recherches à clé sûres. Les autres types de requêtes sûres suivent les principes similaires.

Pour traiter les requêtes sûres, tout pair serveur de données réserve le rang $r = 0$ à un enregistrement système, *S* particulier. Le coordinateur crée *S* avec la case. Tout *S* contient alors dans son champ non-clé une table vide. Celle-ci est assimilée pour le calcul de parité, assurant la *k*-disponibilité, à la valeur zéro. *S* n'a pas de clé, c.-à-d. formellement sa clé est *nulle*. Une telle clé rend *S* inaccessible aux requêtes des pairs clients ou des serveurs de données. En tant que valeur nulle, elle n'est pas non plus re-hachée durant un éclatement de la case. Ainsi, *S* reste toujours dans sa case initiale. Le choix de la valeur *nulle* est un choix technique qui dépend de l'implémentation du prototype. La création de chaque *S* conduit à la création ou la mise à jour usuelle d'un enregistrement avec le rang $r = 0$ aussi sur chaque serveur de parité.

Supposons maintenant qu'un pair découvre une indisponibilité d'un pair serveur, soit le pair *A* avec la case *a*. Notons le *serveur G*, le serveur de parité qui est le gestionnaire du groupe contenant la case *a*. Soit *a'* le numéro de la première case du groupe. Appelons la table dans *S* de cette case *RecoveredPeers* (RP). Le serveur *G* insère l'adresse *A* dans RP. Il génère à cette fin une requête d'insertion d'un enregistrement adéquat, identifié par la clé *nulle*, au pair, soit *A'*, connu de *G* comme ayant la case *a* reconstituée. Comme nous le montrons plus loin, le calcul de parité habituel sur serveur *A'* crée alors une copie de RP dans la case de parité sur *G*. Ceci est fort utile, car *G* peut accéder alors à la table RP localement, évitant donc des messages avec la case *a'*. Le traitement usuel de parité met aussi à jour les enregistrements *S* dans les autres k-1 cases de parité du groupe. En même temps, le gestionnaire récupère comme d'habitude la case *a* sur un autre serveur, soit le serveur *A'*. L'adresse *A'* est alors transmise au client.

Supposons maintenant qu'un pair client envoie une requête sûre *Q*, exécute A5, la requête est envoyée selon l'algorithme A1, fonction *RequeteSure*(*Q, CodeErreur*). Cette fonction envoie *Q* et reçoit la réponse avec le code erreur. Si CodeErreur <> 0 alors il doit mettre à

jour la liste de ces adresses physique au près du coordinateur. Notons que cette demande de mise à jour est une contrainte hérité du prototype de base LH*$_{RS}$.

Tout serveur ayant pour adresse physique *A* recevant *Q* procède à l'identification de *Q* et si *Q* lui est adressé selon le principe de l'algorithme A2 et exécute A6. S'agissant d'une requête sûre, le serveur envoie une demande au manager (gestionnaire) du groupe de parité *G*, la fonction utilisée est *CaseOperationnelle*(*G,A',CodeErreur*). Ce dernier vérifie si le pair *A* n'a pas été reconstruit suite à une panne détectée. Le manager consulte RP pour vérifier si le pair *A* a été reconstruit ou pas. G renvoie un code d'erreur, soit *CodeErreur* égal à 1 si le pair serveur a été reconstruit plus *A'*, l'adresse qui contient la case reconstruite sinon 0. Si le manager ne répond pas à la demande du pair serveur *A* la fonction renvoie un code d'erreur *CodeErreur* = 2 et dans ce cas il déclare l'indisponibilité de *G* au coordinateur utilisant la fonction *ManagerIndisponible*(*G, CRD*), *CRD* étant l'adresse physique du coordinateur.

Algorithme A5/ Requête sûre. Exécuté sur le pair Client
***RequeteSure*(*Q,CodeErreur*)**
Si CodeErreur*<>*0 Alors
    MAJAdresse(CDR)
Sinon
    Exit ;
Fin Si

Algorithme A6/ Requête sûre. Exécuté sur le pair serveur qui reçoit *Q*
***CaseOperetionnelle*(*G,A',CodeErreur*)**
Si CodeErreur* = 0 *Alors
    Execute(Q) ;
Fin Si
Si CodeErreur* =1 *Alors
    ErreurAdresseCase(Q,A')
Fin Si
Si CodeErreur* =2 *Alors
    ManagerIndisponible(G,CDR)
Fin Si

7. Expansion d'un fichier LH $*_{RS}^{P2P}$

Comme tout fichier SDDS le fichier LH $*_{RS}^{P2P}$ évolue grâce aux éclatements des cases de données. Nous présentons ci-après cette expansion.

7.1. Adjonction d'un pair

Un nœud LH $*_{RS}^{P2P}$ *P* qui veut utiliser un fichier *F*, contacte le coordinateur de *F*. Celui-ci l'enregistre dans sa table des pairs. Le pair a le choix : de rester comme un client, ou à intégrer le fichier SDDS. Dans le premier cas, le pair se comportera comme un client simple idem que dans. Le coordinateur désigne aussi le tuteur de *P*. À cette fin, le coordinateur hache

l'adresse IP de *P* utilisant l'algorithme A1. Le pair à l'adresse résultante devient le tuteur *T*. Le coordinateur le contacte alors en lui assignant *P* comme pupille. Le tuteur enregistre le pupille dans ses méta-tables, discutées pour le prototype dans le Chapitre IV. Puis, il envoie à *P* son image *j'* et les adresses IP des pairs dans son image. Le pupille *P* enregistre ces données et initialise son image (i', n') exécutant l'algorithme A3 avec *j'*. Il peut dès lors manipuler le fichier en tant que client. En même temps, en tant que pair candidat, *P* se met en attente d'une demande d'une case de la part du coordinateur. Enfin, *P* est à l'écoute de messages IAM venant du tuteur, issues à chaque éclatement de la case du tuteur.

L'attribution de tuteur aux pairs candidats est faite par le hachage linéaire, selon l'algorithme (A1), pour assurer l'efficacité de la distribution de la charge de tutorat. Ainsi, quelle que soit l'étendue *N* du fichier tout pair serveur dans fichier peut potentiellement être un tuteur. L'emploi de (A1) assure ensuite la rapidité du calcul correspondant. Puis, les propriétés connues du hachage linéaire assurent une bonne uniformité de la distribution de hachage pour tout *N*. Enfin, les modifications de la fonction de hachage à la suite des éclatements assurent une adaptation continuelle aux changements de *N* correspondants. En effet, tout éclatement de la case d'un tuteur faisant que (A1) hache dès lors l'adresse d'un pupille *X* vers la nouvelle case, déplace conformément la charge de tutorat de *X* vers le nouveau pair. Dans l'ensemble, nous évitons ainsi des tuteurs surchargés inutilement.

Quand un éclatement se produit le coordinateur désigne un pupille pour la nouvelle case. Ce dernier contacte son tuteur pour l'informer de son départ en tant que pupille. Le tuteur le supprime alors de sa liste des pupilles appelée Table d'Adressage Pupille (TAP) discutée dans le chapitre IV, Section 2.5.

7.2. Éclatement d'une case LH $*_{RS}^{P2P}$

Quand un pair P_a éclate sa case de données, il met à jour l'image de son image locale utilisant son niveau de la fonction de hachage après l'éclatement *j*. Si le pair a des pupilles, il envoie un message point à point (unicast) pour les informer de l'éclatement. Le pair et les pupilles mettent à jour leurs images utilisant l'algorithme A3.

Lors de l'éclatement, le coordinateur envoi au pair P_a l'adresse physique de la nouvelle case et les adresses physiques des cases créées depuis le dernier éclatement de pair. Ces cases sont $a + 2^{j-1} \ldots a + 2^j - 1$ où j est la valeur après éclatement. Le pair P_a fait suivre les adresses physiques à ses pupilles. Si l'un des pairs pupilles avait reçu un IAM, il se peut qu'il ait déjà quelques adresses des cases. Aussi, il se peut qu'une case ait deux adresses physiques. Ce cas se produit lorsque la case de données est reconstruite sur un autre pair à un instant donné. Cette case sera adressable par l'adresse physique du pair la portant dont le coordinateur connaît l'IP.

Le tuteur maintiendra son image jusqu'au prochain éclatement, ou jusqu'à ce qu'il reçoive un IAM. Et si le pair qui éclate était un tuteur ayant des pupilles alors, il partagera en plus des données la moitié de ses pupilles. Chacun de ces pupilles recevra un message de son nouveau tuteur. Ce schéma assure la distribution uniforme des pairs pupilles entre les tuteurs, un schéma qui évitera la surcharge du tuteur et permettra l'adjonction davantage de pupilles.

Pour assurer la mise à jour des images des pairs pupilles et assurer un seul renvoi d'une requête à clé, nous devons veiller à ce que lors d'une panne ou d'indisponibilité du tuteur la récupération de la liste des pupilles. Pour cela nous devons considérer la liste des pairs pupilles comme un enregistrement de données. Ainsi en cas de panne, la liste comme les données seront récupérés grâce au système de récupération de LH*$_{RS}$.

Cependant, le tuteur enregistre les pairs pupilles dans un enregistrement dit *enregistrement de tutorat*. Cet enregistrement a toujours pour clé le numéro de la case de donnée du tuteur. Nous avons choisi de mettre comme clé le numéro de la case de données pour l'enregistrement du tutorat pour que ce dernier reste toujours dans la même case et donc sur le même tuteur en cas d'un éclatement. Car la case qui éclate transfère la moitié de ses enregistrements à la nouvelle case créée, par hachage des clés des enregistrements.

Le problème de collision de clé ne se pose vraiment pas car il faut prévoir la génération de clé sur un autre domaine que celui des numéros de cases de données. Ainsi nous n'aurons pas de problème d'insertion des enregistrements de données.

Exemple 3

On suppose que notre fichier est 1-disponible (k=1) et la taille du groupe de parité est de m=4. Supposons que le fichier contienne 3 cases distribuées sur 3 pairs, comme le montre la Figure 24. Nous ne tenons pas compte du calcul de parité et à l'allocation des cases de parité.

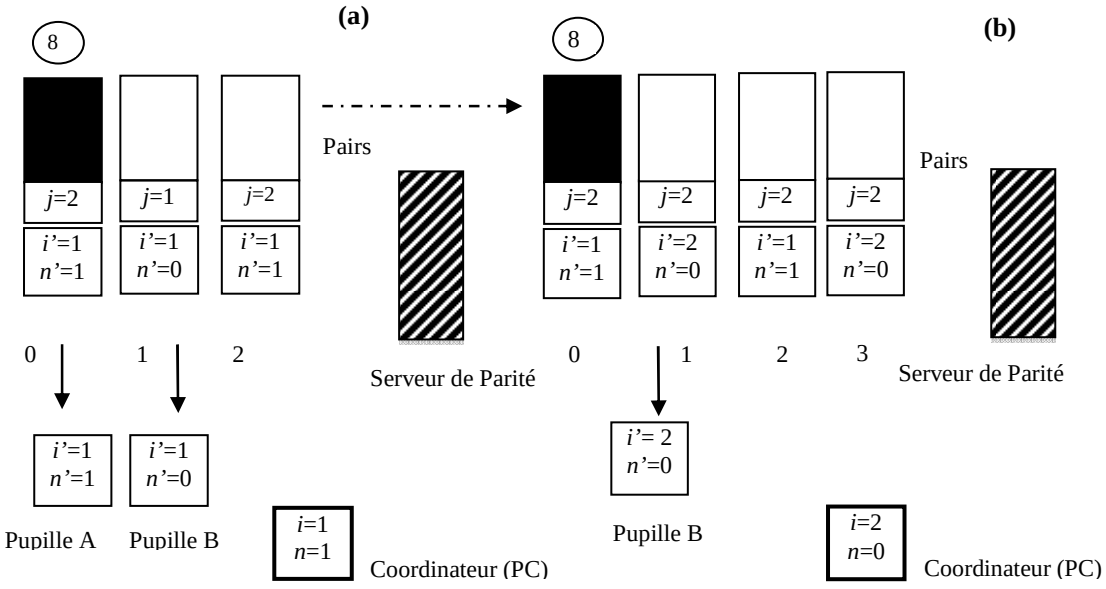


Figure 24. Éclatement dans LH $*_{RS}^{P2P}$ $\{m = 4\ et\ k = 1\}$, (a) avant (b) après

Les adresses logique des pairs sont numérotées 0, 1, 2 et l'état réel du fichier est (i=1, n=1) image sur le coordinateur, Figure 24 (a). Les images sur les pairs sont celles à la création ou celles issues du dernier éclatement de la case 0. Les pairs 0 et 1 sont tuteur d'un pair chacun, initialisé avec leurs images respectives.

L'image du pair 1 n'est plus à jour. Pour notre exemple, nous supposons que le pair 0 a atteint sa limite de stockage. Le pair 1 insère l'enregistrement ayant pour clé 8, donc exécute l'algorithme A1, $a = 8\ mod\ 2^1$, l'enregistrement est envoyé vers le pair 0. Ce dernier exécute

l'algorithme A2, et insère l'enregistrement. Un débordement est constaté alors le pair 0 contacte le coordinateur qui indiquera le pair portant la case à éclater. L'image du coordinateur est (*i*=1, *n*=1). Le pair 1 éclate donnant ainsi la nouvelle case qui sera sur le pair 3, comme le montre la Figure 24 (b), le pair 1 envoie un message IAM de mise à jour plus son niveau de fonction de hachage après éclatement *j* = 2 pour son client local et ses pairs pupilles qui exécutent l'algorithme A3 pour mettre à jour leurs images.

$a' = 1; i' = j = 1;\ n' = a' + 1 = 2\ et\ n' = 2^{i'} = 2\ Alors\ i' = 2; n' = 0.$

La sélection du pair pupille pour recevoir la nouvelle case peut se faire de différente manière. Soit par ordre d'enregistrement auprès du coordinateur et dans ce cas c'est au coordinateur d'indique lequel sera choisi. Soit par diffusion de message d'éclatement et dans ce cas c'est le premier pair pupille ayant répondu qui prend la case. Nous donnerons plus de détails quant à la solution que nous avion pris dans le chapitre de l'architecture fonctionnelle car étroitement liée à celle de LH*$_{RS}$.

Exemple 4

Nous illustrons par cet exemple l'expansion d'un fichier LH $*_{RS}^{P2P}$. Comme tout fichier SDDS, il peut s'étendre théoriquement à un nombre infini de cases, donc de nœuds sur le réseau. Supposons :

- Que la taille du groupe de parité est de trois et que le fichier est 1-disponible ($m = 3\ et\ k = 1$).
- Que la capacité d'une case de données est de 4 enregistrements.
- Qu'il y ait toujours des pairs candidats qui se présentent pour recevoir une case de données et rejoindre fichier.
- Qu'il y ait des serveurs disponibles pour accueillir les cases de parités.

Pour les besoin de cet exemple, nous ne tenons pas compte du calcul de parité et de l'allocation des cases de parité.

Comme le montre la Figure 25 ci-après, initialement le fichier LH $*_{RS}^{P2P}$ est constitué d'un seul pair, noté P_0, sur lequel est logée la première case de données et d'une case de parité logée sur un serveur dit de parité, noté SP_0. Le tout est dans le même groupe de parité G_0. Les paramètres du pair sont ; *j*, *i'*, *n'*. Pour rappel, *j* est le niveau de la fonction de hachage à la création de la case. Le couple (i', n') est l'image du fichier sur le client local du pair *a*, tel que *a* est l'adresse logique du pair. Les paramètres du coordinateur est le couple (n, i) tel que *i* est l'état réel de la fonction de hachage du fichier et *n* est le pointeur d'éclatement de la prochaine case. Tous les paramètres sont à zéro à la première case du fichier comme le montre la Figure 25.

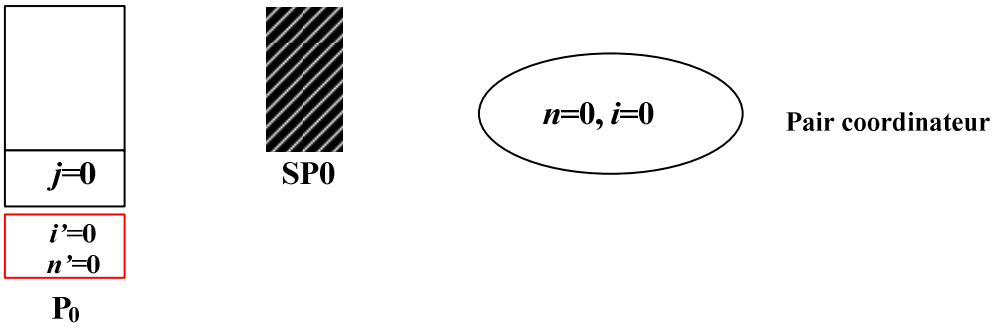


Figure 25. État initial du fichier

Insertion des clés 1, 3, 4, 6 dans P_0. Supposons qu'un nouveau pair P_1 candidat se présente et devient pupille de P_0 après son enregistrement auprès du coordinateur. P_1 insère la clé 7 et provoque un débordement. Le coordinateur indique la case du pair qui éclate tel que $n = 0$. L'adresse logique du pair qui éclate sa case est $a' = 0$, P_0 met à jour l'image de son client local avec l'algorithme A3 utilisant le j après éclatement donc $(i' = j = 1 - 1;\ n' = a' + 1;)$ et $n' = 2^{i'=0}$ donc l'image du client sera $(i' = i' + 1 = 1; n' = 0)$

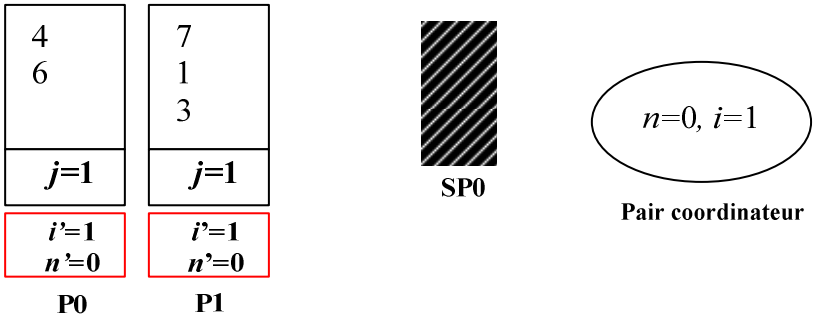


Figure 26. Fichier à deux cases de données

Insertion des clés 2, 8 par P_1 vont directement vers P_0. L'insertion de 10 par P_1 provoque le débordement de la case sur P_0 d'où son éclatement, le pointeur d'éclatement $n = 0$. L'adresse du pair est $a' = 0$. Mise à jour de l'image du client local de P_0 avec l'algorithme A3, $(i' = j = 2 - 1; n' = a' + 1 = 1)$ tel que $n' \neq 2^{i'=1}$, Figure 27.

Insertion de la clé 5 va dans P_1, la clé 11 provoque un débordement de la case de P_1. Alors le pointeur d'éclatement $n = 1$ indique l'éclatement de P_1, ayant pour adresse $a' = 1$. Idem l'image du client local de P_1 est mise à jour par l'algorithme A3 d'où $(i' = 2, n' = 0)$, comme le montre la Figure 28.

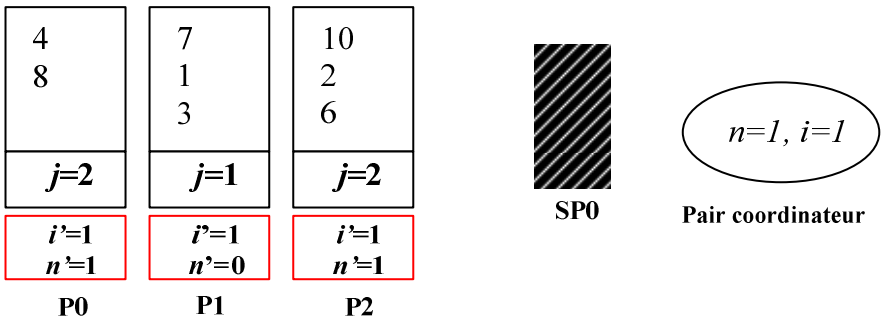


Figure 27. Fichier à trois cases de données

Cet éclatement crée une case de données tel que $m > 3$ d'où la création d'une nouveau groupe de parité G_1 de taille égale à G_0 tel que $m = 3$ avec sa case de parité, SP_1comme le montre la Figure 28 ci-après.

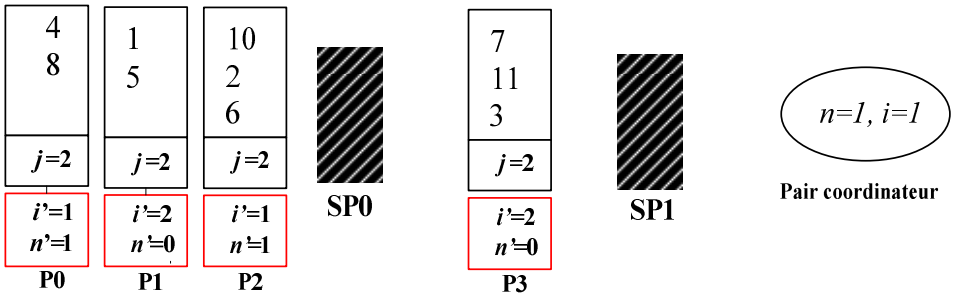


Figure 28. Fichier à quatre cases de données

8. Propriétés de LH $*_{RS}^{P2P}$

Nous allons maintenant présenter trois propriétés fondamentales de LH $*_{RS}^{P2P}$. Les deux premières déterminent le nombre maximum de messages de renvoi, respectivement pour une requête à clé et un scan. La troisième propriété montre qu'en ce qui concerne le nombre maximum de messages de renvoi pour une requête à clé, aucune SDDS ne peut faire mieux que notre schéma. Nous prouvons les trois propriétés annoncées.

A) *Propriété 1 :*

Le nombre maximum de messages de renvoi pour une requête à clé est de 1.

B) *Propriété 2 :*

Le nombre de rounds (Anglais : rounds) de messages unicast pour envoyer un *Scan* est au maximum 2.

C) *Propriété 3 :*

Le résultat de la propriété 1 est le meilleur possible pour une SDDS.

- **Preuve de la propriété 1**

Nous considérons trois situations qui sont les seules possibles pour une requête à clé. Les Figure 29, Figure 30, Figure 31 illustrent ces situations. Soit a l'adresse (logique) du pair lançant la requête. Nous notons comme d'habitude son image par (i', n'). Nous notons par j le niveau de la case du pair, également comme d'habitude. Les trois situations possibles pour le pair a sont comme suit :

1. Un pair a' désigné par le pointeur d'éclatement vient de subir un éclatement, mais pas le pair suivant le pair a'. Donc avant l'éclatement nous avons $n = a'$. Après celui-ci, le pointeur n pointe typiquement vers le pair $n = a' + 1$. Rarement, il pointe vers le pair $n = 0$. La Figure 29 illustre le cas typique. Supposons que l'on a $a = a'$. Nous pourrions alternativement avoir $a = a' + 2^i$. En d'autres termes, la requête à clé pourrait émaner du pair qui vient d'éclater, comme sur la Figure 29 ou du pair serveur de la nouvelle case. Enfin, la requête pourrait être émise par tout pupille éventuel de ces pairs serveurs. Nous démontrons qu'aucun renvoi n'est possible pour une telle requête.

2. Le fichier avec $n = a' + 1$ à un moment, a subi davantage d'éclatements. Cependant, pas assez pour que le pair 0 éclate à son tour aussi. Donc, pas suffisamment pour que n soit $n = 0$, après avoir été $n > a'$ pour $a' < 2^i - 1$. La Figure 30 illustre une telle situation. Nous démontrons aussi qu'un et un seul renvoi est alors possible pour tout pair considéré plus haut.

3. Le fichier de la Figure 29 et de la Figure 30 auparavant a grandi encore davantage. Au point que l'on a $n \leq a$. C'est le cas illustré par la Figure 15. Nous montrons que de nouveau, un et un seul renvoi est alors possible pour tout pair considéré plus haut.

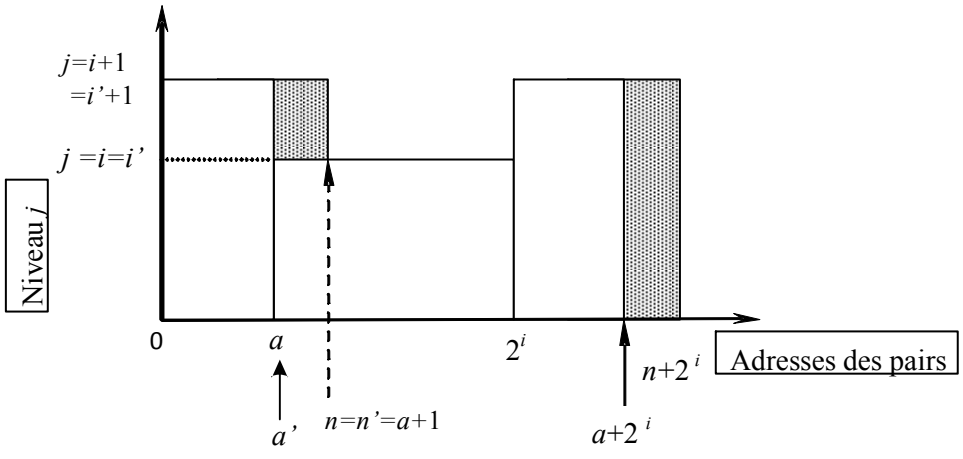


Figure 29. Régions d'adressages sans renvois

Situation 1 : Le pair a vient de procéder à l'éclatement. Pour cela, il a utilisé la fonction h_{j+1}. Sur la Figure 29, nous avons en conséquence $n = a + 1$. Après l'éclatement, le pair a mis à jour la valeur de son j à $j = j + 1$. Cette valeur j est aussi celle initiale de la

nouvelle case créée par le pair, la case $a + 2^i$ donc. Nous avons dès lors pour le pair a, $i' = i = j - 1$ et $n' = n$. Autrement dit, par le principe même de notre algorithme, l'image client sur le pair a est égale à l'état (i, n) du fichier. Toute adresse calculée par le pair, (en utilisant l'Algorithme A2), ne peut être que correcte. Pour chaque requête à clé issue par le pair il ne peut pas donc y avoir de renvoi.

La Figure 29 montre en particulier que les cases (pairs) du fichier constituent trois régions d'adressage. Celles-ci sont caractérisées par leur valeur de j par rapport à i. Apres l'éclatement de la case a discuté, la première région qui est celle de $0, \dots, a'$, est caractérisée par $j = i + 1$. La deuxième région comporte les cases $a' + 1 \dots 2^i - 1$, avec $j = i$. Enfin la troisième région comporte les cases $2^i \dots a' + 2^i$ avec de nouveau $j = i + 1$. Si l'on avait $a' = 2^i - 1$, alors, après l'éclatement, toutes les régions de la Figure 29 auraient une même valeur de j qui serait $j = i$.

Il est aisé de voir que dans ce dernier cas, pour les mêmes raisons que celles discutées plus haut, qu'aucune requête à clé émise par le pair a après l'éclatement ne peut créer un renvoi. Il en est de même pour toute requête à clé émise par tout pupille éventuel du pair a, après l'IAM lui notifiant l'éclatement de la case de son tuteur. Enfin, il en est de même pour le pair $a' + 2^i$ avec la nouvelle case qui vient d'être créée sur la Figure 29.

Le pair a dispose naturellement de l'image des régions. Dans la situation (1) discutée, cette image concorde avec celle réelle sur la Figure 29. Les Figure 30 et Figure 31 montrent l'évolution des régions. Cette évolution peut être au moins partiellement inconnue au pair a ou à son pupille. Le niveau j d'une partie dans une région dans l'image du pair a peut alors être plus petit que le niveau réel. C'est la raison fondamentale pour des renvois éventuels signalés pour les Situations 2 et 3.

Dans la situation 2 citée ci-dessus le pair qui éclate est $n = a' + 1$ utilise la fonction de hachage h_{j+1} et met à jour son niveau *j* tel que $j = i + 1$. Dans ce cas le pair *a* qui lancer une requête à clé va dans les pair se situant dans l'intervalle $[2^i - 1, a + 2^i]$ ce qui équivalent à la situation 1 décrite. La zone où y a possibilité de renvoi se situe dans l'intervalle $]a + 1, 2^i - 1]$, dans ce cas $a' + 1 < n < 2^i - 1$ et le niveau de la fonction de hachage des pairs appartenant à cet intervalle ont $j = i$. Toute requête à clé issue de ces pairs peut être erronée car le fichier a évolué entre temps comme le montre la Figure 30. La requête sera envoyée soit à la zone $[0, a + 1]$ soit à la zone $[2^i, a + 2^i]$ dans les deux cas il n'y aura qu'un seul renvoi car il ne peut pas y avoir de pair qui éclate deux fois sans que le pointeur d'éclatement revienne à zéro et que le niveau de la fonction de hachage du fichier soit incrémenté.

Dans la situation 3, les régions d'adressage possibles est comme le montre la Figure 31. Dans ce cas, le pointeur d'éclatement est revenu à zéro ($n = 0$) et quelques cases ont éclaté utilisant la fonction de hachage de niveau h_{i+1}. Cependant, la case du pair *a* n'a pas encore éclaté car $n < a$. Soit le pair *a* lance une requête à clé *C* et utilisant l'algorithme A1, l'envoi au pair ayant pour adresse a'. Dans ce cas, il pourrait y avoir un renvoi vers la case du pair $a' + 2^i$ et pas au-delà car cela supposerait que le pair a' a éclater plus d'une fois depuis le dernier éclatement du pair a. Or ce cas est impossible car le pair a aurait éclaté et aurait son $j = i + 1$ et le pair a' aurait $j = i$. Le résultat aura toujours un seul renvoi.

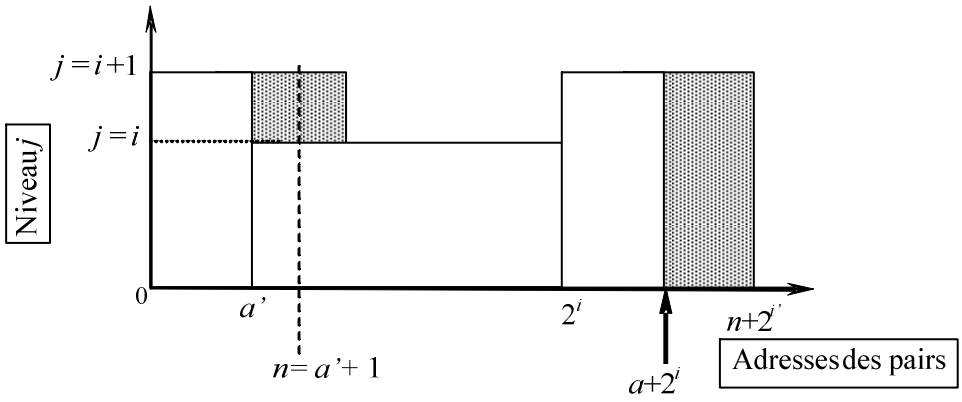


Figure 30. Régions d'adressages avec possibilités de renvois

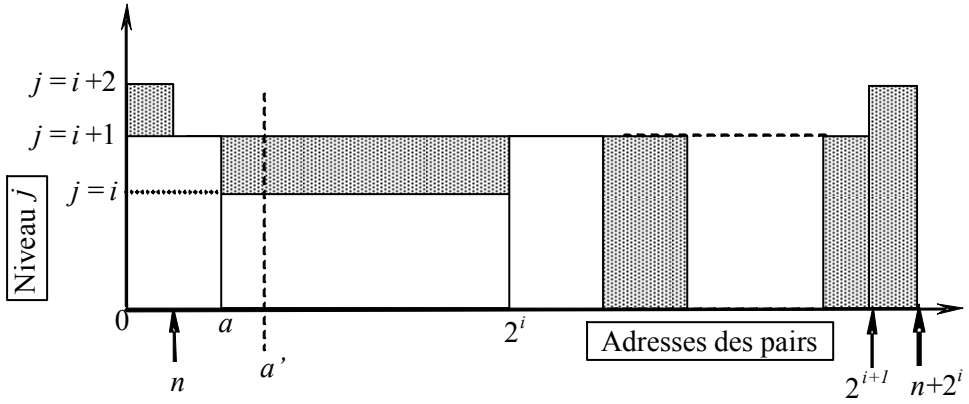


Figure 31. Régions d'adressage avec la possibilité de renvoi

Preuve de la propriété 2

En rappelle de la Section 5, un scan envoyé par des messages unicast devrait être délivré *correctement*. Un message unicast de scan doit être délivré ainsi à tout pair serveur du fichier. Puis, tout pair serveur ne devrait recevoir qu'un seul de tels messages. La requête de scan peut être délivrée à un pair serveur par un message unicast venant du pair qui a initié le scan. Alternativement, elle peut être délivrée par un message (unicast) de renvoi produit par l'Algorithme A4. Le pair initiant un scan peut être un pair serveur ou un pupille d'un tel pair.

Supposons ainsi qu'un pair a qui peut être ici un pair serveur avec la case a ou son pupille éventuel, demande un scan. Il envoie le message avec le scan à tout pair a' dans l'image. Tout pair a' exécute alors l'Algorithme A4. Le niveau j du pair a' ne peut être alors que $j = j'$ où $j = j' + 1$. Ce dernier cas signifie que le pair a avait subi l'éclatement inconnu du pair a, utilisant alors la fonction $h_{j'+1}$. L'éclatement a créé le pair serveur descendant, ayant l'adresse $a' + 2^{j-1}$. Celui-ci est nécessairement en dehors de l'image du pair a. Ce qui est une condition nécessaire et suffisante pour $j = j' + 1$. Tout pair serveur d'un fichier LH $*_{RS}^{P2P}$ ne peut être que dans l'image du pair a ou en dehors de cette image, étant un

descendant d'un pair dans cette l'image. Ce que nous voyons aisément sur les régions des Figure 29, Figure 30 et Figure 31. Il n'y a aucun autre pair serveur dans le fichier car aucun descendant n'aurait pu subir encore d'éclatement. D'autre part, aucun pair dont l'image n'aurait pu subir d'éclatement plus qu'une fois depuis le dernier éclatement du pair a. Donc il ne peut avoir qu'un seul descendant.

Selon l'Algorithme A4, tout pair a' ayant un descendant lui renvoie le scan. Comme évoqué en Section 5 celui-ci ne fait alors qu'exécuter le scan reçu. Ainsi tout pair serveur d'un fichier LH $*_{RS}^{P2P}$ reçoit et exécute le scan, soit à la suite du message du premier round, envoyé par le pair a ou à la suite du message du second round, messages envoyés par tout pair a'. Aucun pair ne reçoit plus d'un message, que ça soit d'un même round ou du second round. Tout scan s'exécute donc correctement et en deux rounds de messages au maximum.

Preuve de la propriété 3

Par définition, tout SDDS peut subir des éclatements et tout éclatement dans une SDDS n'est pas connu en synchrone d'aucun client ou serveur d'une case de données autre que celui de la case qui éclate et le serveur de la nouvelle case. Il peut y avoir alors un client ou un serveur de données ou un pair donc, dont l'image n'inclut pas la nouvelle case et qui envoie une requête à clé, adressant une clé qui vient de migrer dans la nouvelle case. Le client l'adressera au mieux au serveur de la case qui vient d'éclater. La requête va générer donc au moins un renvoi. La borne de zéro renvoi au maximum est donc impossible pour une SDDS. Le résultat de la Propriété 1 est le meilleur possible pour une telle structure de données.

9. Variante de LH $*_{RS}^{P2P}$

Plusieurs variantes peuvent se décliner du schéma de base de LH $*_{RS}^{P2P}$. Nous nous limitons à en décrire une, publiée dans [YS10]. Cette variante vise une nouvelle distribution des enregistrements de parités. De plus ces enregistrements se trouvent sur des pairs serveurs ou candidats. Le schéma est nouveau par rapport à LH*$_{RS}$ et les autres variantes LH* à haute disponibilité connues.

Les pairs sont, comme d'habitude, numérotés (0,1,…). Ils sont organisés en groupes de parité, comme dans LH $*_{RS}^{P2P}$ de base. Tout pair n appartient au groupe $g = n/m$, où m est la taille d'un groupe et le caractère '/' désigne la division entière. Nous distribuons les enregistrements de parités de tout groupe $g = 0, 2$ ... sur tous les pairs du groupe $g' = g + 1$. Ceux du groupe g' sont distribués sur le groupe g. La distribution de ces enregistrements est uniforme, d'abord dans le sens que les $k < m$ enregistrements de parité d'un groupe de m enregistrements de données, donc d'un même rang r dans un groupe de parité, sont mappés dans des cases différentes. Puis, les k enregistrements de parité d'un groupe d'enregistrements de données du rang $r + 1$ sont décalés d'une case à droite, modulo m. Plus précisément, les k enregistrements de parités d'un groupe d'enregistrements de données de rang $r = 1$ sont mappés sur le 1er, 2ème,…$k^{ème}$ pair du groupe g' ou g. Ceux de rang $r = 2$ sont mappés sur le 2ème, puis 3ème pair.

Le calcul d'adresse se fait $modulo\ m$, pour que tous ces enregistrements restent dans le même groupe de parité. Ainsi, d'une manière générale, si $P_0, P_1 \ldots P_{m-1}$ sont des pairs successifs dans un groupe, alors les k enregistrements de parité du groupe de parité du même rang r tel que $r = 1, 2, \ldots$; vont successivement dans les pairs numérotés comme suit $P_{m+(r-1\ mod\ m)}, P_{m+(r-2\ mod\ m)}, \cdots, P_{m+(r-k\ mod\ m)}$.

La Figure 32 montre un exemple de cette distribution. La taille m du groupe de parité pour ce fichier est $m = 4$. Nous supposons que $k = 2$ enregistrements de parité par groupe d'enregistrements de données donc par groupe de quatre enregistrements d'un même rang et dans un même groupe de parité. Initialement, le fichier est constitué d'un seul pair P_0 avec sa case de données, appartenant au groupe de parité 0. Le fichier contient aussi les m pairs (candidats car sans cases de données encore) du groupe 1, portant chacun une case de parité. À noter que dans cette variante, chaque pair candidat a une adresse logique, donc la case de données qu'il aura à gérer est définie par avance. Le candidat attend l'éclatement correspondant. Il est, en attendant, sous tutorat, comme pour la version de base de LH $*_{RS}^{P2P}$.

Chaque case de parité du groupe 1 reçoit les enregistrements de parités du groupe 0. Selon le schéma présenté plus haut, les k enregistrements de parité, symbolisés par α sur la figure et correspondant aux $m = 4$ enregistrements de données de rang $r = 1$, notés aussi α sur la figure, sont successivement dans les cases de parité du pair 4, puis 5. Idem, pour les enregistrements de données et de parité symbolisés par β, correspondant au rang r = 2. Ceux de parité sont dans les cases 5 et 6. Suivis par ceux symbolisés par μ pour $r = 3$ avec ceux de parité dans les cases 6 et 7. Les enregistrements de parité pour $r = 4$ seraient alors dans les cases 7 et 4 de nouveau, et ainsi de suite. Ce schéma garanti que pour tout r dans le groupe 0, les k enregistrements de parité sont toujours dans les cases de parité différentes. Cette condition est nécessaire à l'évidence pour la *k*-disponibilité.

Les éclatements de cases du groupe 0 font ensuite élargir le groupe avec les cases de données $1, \ldots m - 1$, comme illustré dans la Figure 32(B). Quand le pair m reçoit une case de données, alors le groupe 1 peut commencer à recevoir des enregistrements de données. Dans ce cas les k enregistrements de parité vont dans le groupe 0. Figure 32(C) illustre cette évolution. Chaque groupe d'enregistrements de données et ses enregistrements de parité sont symbolisés par une même lettre grecque.

Par la suite, la création de la case de données sur le pair 8 déclenche la création du groupe 3 pour les cases de parité correspondantes, Figure 32(D). Puis, quand la 1ère case de données du groupe 4 sera créée, c'est le groupe 3 qui recevra les enregistrements de parité du groupe 4.

Ce schéma d'abord ne permet pas d'avoir les enregistrements de données et de parités du même groupe sur les cases du même groupe de parité. Ceci garanti qu'une indisponibilité d'une case avec un enregistrement de données ne peut pas affecter les cases de parité correspondantes et vice versa. Ensuite, le placement discuté garantit que chaque enregistrement de parité dans chaque ensemble de k enregistrements est dans une case différente. Ainsi l'indisponibilité de l cases n'affecte au maximum que l enregistrements parmi les k. Le tout, garantit la *k*-disponibilité, comme nous l'avons détaillé dans [YS10].

Enfin le schéma équilibre la charge de stockage et de calcul de parité entre les nœuds du fichier mieux que schéma basique de LH $*_{RS}^{P2P}$.

Surtout, le schéma permet potentiellement une reconstruction des cases indisponibles m fois plus rapidement. La reconstruction pourrait avoir besoin d'un parallélisme du processus avec un temps de $1/m$ pour une case à reconstruire. Si nous avons 1GB par case de données et la taille du groupe $m = 4$ dans notre exemple, nous aurons un temps T inférieur à 43 seconds par reconstruction d'une case de données. Un temps inférieur à 70 seconds et 6 minutes pour 2GB et 3GB respectivement. Voir les détails de ce calcul dans [YS10].

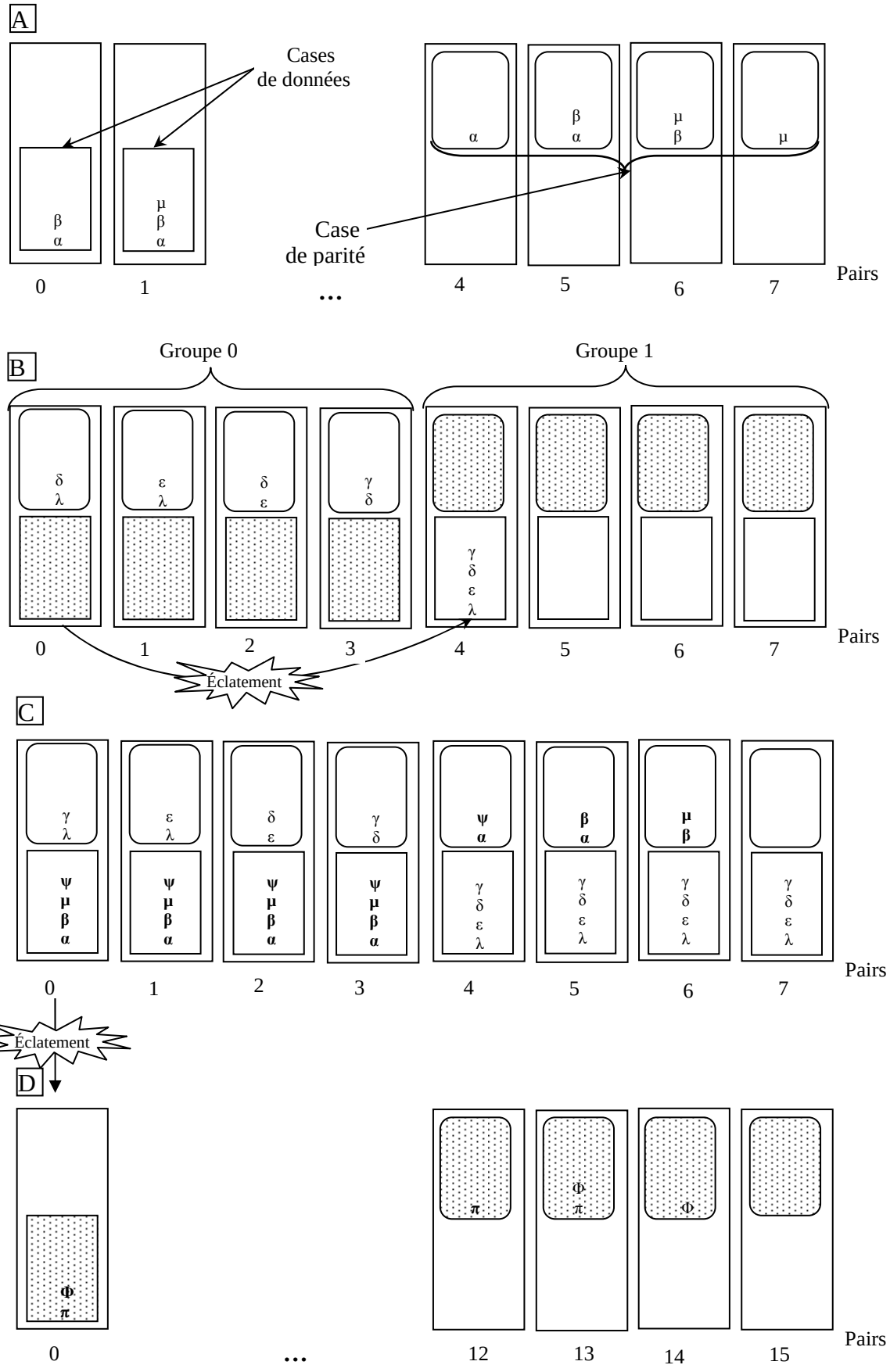


Figure 32. Schéma de distribution de parités (*m*=4, *k*=2)

10. Résumé

Dans ce chapitre nous avons présenté l'architecture et l'algorithmique de notre système. Nous avons montré que le tout garantit un seul renvoi en cas d'erreur d'adressage d'une requête à clé, ainsi que les performances présentées d'un scan. Nous pouvons conclure que les propriétés de notre SDDS font d'elle une structure de données rapide et efficace. Enfin, en ce qui concerne le nombre maximal de renvois d'une requête à clé, notre structure est optimale dans sa classe.

Chapitre IV

Architecture Fonctionnelle de LH $*_{RS}^{P2P}$

1. Introduction

Nous présentons maintenant l'architecture fonctionnelle de notre prototype LH $*_{RS}^{P2P}$. Celle-ci assure l'exécution répartie de toute requête et la communication entre les nœuds coopérants à cette fin. Notre système d'exploitation sur chaque nœud était Windows Server 2000. L'exécution répartie utilisait sur chaque nœud des *threads* qui sont les processus légers. Les threads s'échangent les données à travers des files d'attente. La communication entre les nœuds passe par des messages. Ceux-ci utilisent des structures d'adressage spécifiques. Ainsi, chaque nœud possède en premier une adresse logique unique dans le fichier dite numéro d'entité. Chaque pair serveur de données possède aussi l'adresse logique qui est le numéro de sa case. A ces adresses correspondent enfin pour chaque nœud une adresse IP et un numéro de port, car nos protocoles de communication sont UDP et TCP/IP. Ce dernier est utilisé en unicast et multicast. Des (métas) tables d'adressage sur chaque nœud contiennent toutes ces adresses de ses correspondants. Les numéros de ports sont calculés dynamiquement.

Notre architecture se base largement sur celle du prototype LH*$_{RS}$ [LMS04] [M04] [LMS05]. Elle nécessitait néanmoins la révision et l'extension aussi bien de la conception générale que de certaines fonctions existantes. Nous décrirons surtout cet aspect de notre travail, c.-à-d. la conception et l'implémentation de nouvelles fonctions spécifiques à LH $*_{RS}^{P2P}$. Les fonctions communes seront indiquées, mais beaucoup de leur détails tels que les paramètres et les valeurs de retour ne seront pas répétés. Le lecteur intéressé est invité à consulter les références indiquées.

Les différences entre les deux architectures résultent de l'implémentation des prototypes LH $*_{RS}^{P2P}$ et LH*$_{RS}$. L'image du client dans LH*$_{RS}$ n'est ajustée que par des IAMs asynchrones lors des éclatements. Dans LH $*_{RS}^{P2P}$, cette image est ajustée aussi par les IAMs synchrones. Ceux-ci sont émis par le serveur local du pair ou par le tuteur du candidat. Les nouvelles fonctions assurent surtout le fonctionnement de ces nouveaux IAMs. En interne sur le pair d'une part, en dialogue entre le pair tuteur et un pupille d'autre part.

Dans ce qui suit nous détaillons la description de notre architecture. D'abord nous présentons l'organisation générale de chaque type de nœud et son architecture interne. Ensuite, pour chaque type, nous décrivons les tables d'adressages et la structure des messages

émis et reçus. Nous présenterons ensuite les scénarios qui régissent l'exécution répartie des requêtes (Insertion, recherche simple, recherche sûre et recherche scan). Chaque scénario est résumé par un diagramme UML. Nous illustrons la description aussi par des exemples. Nous concluons enfin notre architecture.

2. Architecture Fonctionnelle d'un Nœud LH $*_{\mathbf{RS}}^{\mathbf{P2P}}$

On rappelle qu'un nœud LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$ peut être un pair, pair serveur, comportant alors une case de données, ou pair candidat, en attente d'une case de données. L'architecture d'un pair serveur et d'un pair candidat est la même. Nous la désignons simplement comme celle d'un pair LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$. Alternativement, un nœud peut être (juste) un serveur de parité, avec une case de parité ou le nœud coordinateur.

L'architecture fonctionnelle représente la description des composants logiciels d'un nœud LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$, indépendamment de l'organisation et de l'interconnexion de diverses machines constituant un réseau. Toute machine d'un réseau est caractérisée par une adresse IP. Sur cette machine nous pouvons installer une ou plusieurs instances du logiciel LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$. Ainsi une machine peut accueillir un ou plusieurs nœuds candidats, un ou plusieurs serveurs de parité. Toutefois, une machine ne peut avoir qu'un seul coordinateur lancé car au lancement du coordinateur un fichier mappé est créé en mémoire. Il ne peut y avoir plus d'un. Ainsi une machine ne peut supporter qu'un seul fichier SDDS à la fois. C'est une des limites héritée du prototype LH*$_{\mathrm{RS}}$.

Un nœud coordinateur a un numéro d'entité égal à zéro par défaut. Ce nœud peut-être sur une machine dédiée, ou peut partager celle comportant un pair ou un serveur de parité. Le coordinateur d'un fichier *k*-disponible est répliqué sur *k* nœuds du fichier. Au lancement du coordinateur, il initialise ses threads d'écoute et d'envoi UDP avec le numéro de port 6000 et 6001 respectivement. Le thread TCP quant à lui est initialisé avec le numéro de port 6002. Seul le coordinateur est initialisé avec ces numéros. Les autres pairs auront des numéros de ports calculés selon leur numéro d'entité attribué par le coordinateur.

2.1. Pair LH $*_{\mathbf{RS}}^{\mathbf{P2P}}$

L'architecture d'un pair de données LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$ se présente comme la jonction de deux architectures : (i) d'un nœud 'client' et (ii) de celui d'un 'serveur' LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$, comme le montre la Figure 33. Les deux nœuds correspondent dans notre architecture à deux composants logiciels, appelés respectivement 'client' et 'serveur'. Notons que le client et le serveur sont des exécutables générés à partir du projet source C/C++. Chaque client ou serveur d'un pair se connecte au réseau via la même adresse IP du pair, ayant chacune un numéro de port distinct. Ce dernier est calculé suivant la formule donnée dans la Section 3. Le pair offre une interface utilisateur dite 'Application' décrite ci-après.

La gestion des deux composants est simple et avantageuse. Simple car les deux composants peuvent communiquer juste en cas de besoin et apporte un avantage ; en cas de bug sur le 'serveur' ou le 'client'. Le pair/nœud ne cessera pas de fonctionner et offrira les services du composant n'ayant pas de bug. La seule panne qui peut interrompre la disponibilité des deux composants est la panne matérielle de la machine où sont installés les

deux composants logiciels ou une panne du réseau. La manipulation du fichier SDDS est assurée par l'application interfaçant le logiciel client.

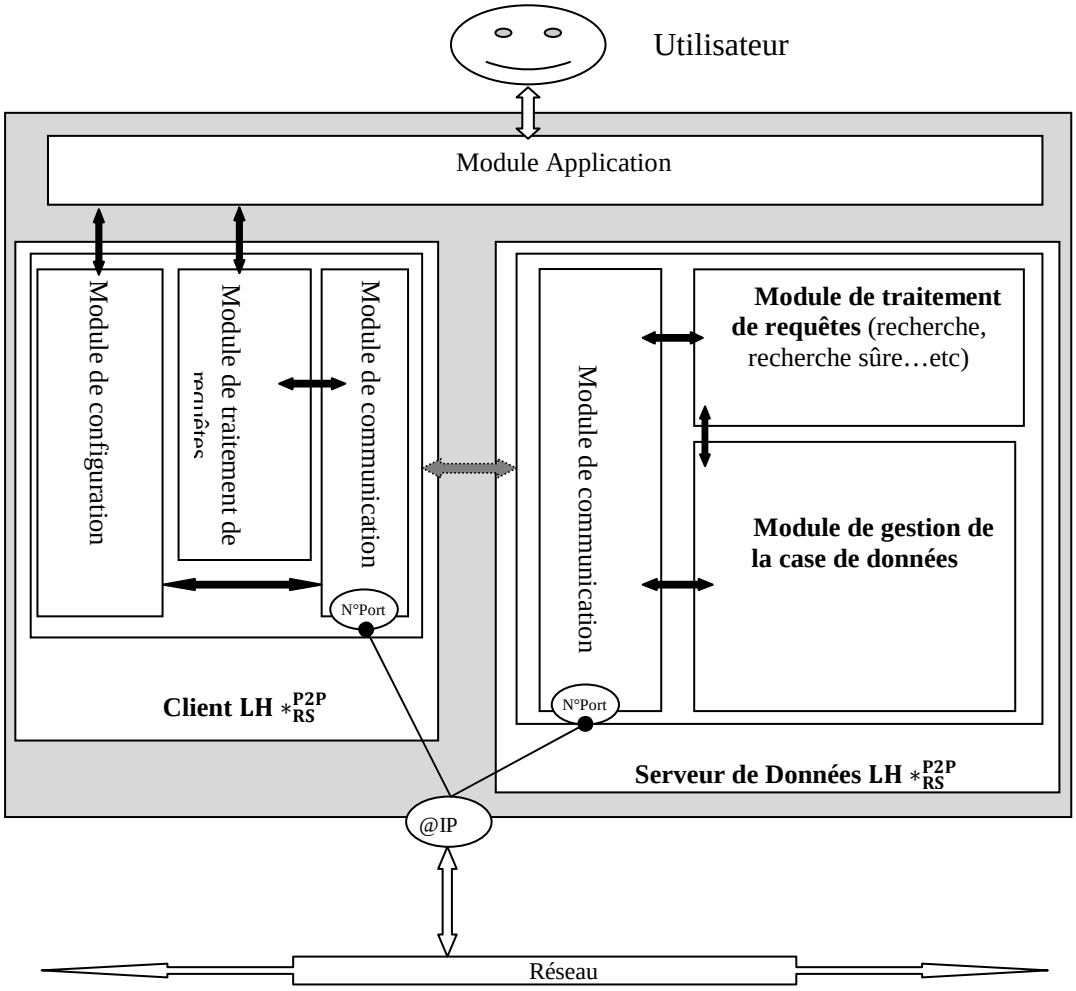


Figure 33. Architecture fonctionnelle d'un pair de données LH $*_{RS}^{P2P}$

L'interface du logiciel 'serveur' sert d'écran d'affichage d'informations et de suivi d'opérations diverses (recherche, insertion, mise à jour, suppression). Nous avons aussi tiré profit de l'architecture existante, pour réutiliser le plus possible les fonctionnalités offertes par le prototype LH*$_{RS}$. Sur la Figure 33 les flèches bidirectionnelles pleines en noire (⟷) désignent des méthodes, écrites en langage C/C++, internes qui assurent la communication entre les différents modules. La flèche bidirectionnelle en gris rayée (⟺) reliant les deux composants logiciels 'client et serveur' au milieu, est un protocole de communication basé sur un appel LPC (Anglais : Local Call Procedure). Celui-ci assure l'échange de messages en cas de mise à jour d'image synchrone avec l'éclatement de la case de données du pair. Les deux autres flèches (⟺) sont les protocoles de communication réseau assurant l'interconnexion des nœuds du fichier.

2.1.1. Application

Le module application constitue l'interface entre l'utilisateur et le client LH $*_{RS}^{P2P}$. Il est lancé au lancement du composant client sauf que c'est un module externe au client.

L'application offre un menu d'opérations pour la manipulation du fichier SDDS ainsi que des options de configuration du système. Chaque opération de manipulation concerne un enregistrement à la fois, identifié dans la requête par sa clé primaire.

Il s'agit ainsi :

- De la recherche simple par clé,
- De la recherche sûre que nous avons rajoutée et présentée dans Section 5.5,
- De recherche simple et sûre en flux (Anglais : Bulk) d'enregistrement que nous avons rajoutée. le mot 'massive' correspond à une génération automatique de clés que le système recherche que nous utilisons dans l'insertion d'enregistrement.
- De l'insertion individuelle et en flux d'enregistrements,
- De suppression et de mise à jour.
- Des options de configuration qui sont : l'enregistrement du pair auprès du coordinateur que nous avons rajouté pour la gestion de l'affectation des pupilles, la gestion de la *k*-disponibilité, l'ajout d'une case de parité et l'arrêt du fonctionnement en tant qu'un nœud du fichier.

Le module application fait appel au client pour tous ces traitements. En fait, le client ventile ces appels vers le module de configuration pour les opérations de configuration et celui de traitement des requêtes pour les opérations de manipulation du fichier.

L'application se charge de traduire les demandes de l'utilisateur choisies via le menu en requêtes prêtes à être envoyer au client LH $*_{RS}^{P2P}$. Une présentation du menu de l'application est donnée dans l'annexe avec des exemples d'exécution.

2.1.2. Client LH $*_{RS}^{P2P}$

Le composants Client LH $*_{RS}^{P2P}$est constitué de trois principaux modules : configuration, traitement de requêtes et communication. Le module configuration et traitement de requêtes font appel au module de communication pour permettre au nœud pair d'interagir avec les autres pairs du fichier SDDS ou de permettre un échange de messages internes au pair. Ainsi, les composants client/serveur communiquent entre eux de manière synchrone et asynchrone contrairement aux autres algorithmes basés sur LH*, proposés dans le passé. Une seule instance du composant logiciel du client LH $*_{RS}^{P2P}$ avec l'application peut être installée sur une machine donnée du réseau. Le client LH $*_{RS}^{P2P}$ utilise le numéro de port 6060 au lancement et avant l'enregistrement auprès du pair coordinateur. À l'enregistrement de ce dernier auprès du coordinateur. Il reçoit un numéro d'entité qui sera utilisé pour lancer ses threads d'écoute et d'envoi UDP dont nous détaillons l'utilisation dans la Section 2.4.1 et Figure 37. La formule de calcul des numéros de ports est donnée à la Section 3.

Notons que l'enregistrement du pair est la première opération que fait l'utilisateur pour avoir la possibilité de manipuler le fichier SDDS. L'utilisateur saisi via l'application l'adresse IP du pair coordinateur et valide sa saisie et la requête est envoyée au pair coordinateur.

A) ***Module de Configuration***

Le module de configuration regroupe toutes les méthodes nécessaires à la configuration du pair, notamment celle assurant l'enregistrement du pair auprès du coordinateur et l'initialisation du niveau (k = 0, 1, 2…) de la haute disponibilité du fichier LH $*_{RS}^{P2P}$ créé. Nous donnerons plus de détails dans ce qui suit quant à l'exécution de différentes opérations sur notre prototype LH $*_{RS}^{P2P}$.

B) ***Module de Traitement de Requêtes***

Le module de traitement de requêtes regroupe quant à lui l'ensemble des méthodes interfaçant les requêtes de l'application. Celles-ci sont identifiées et traitées, en vue de l'appel au module de communication. Avant d'être envoyées, les requêtes sont analysées. Nous vérifions le contenu des requêtes et nous rajoutons quelques données complémentaires nécessaires à l'exécution de la requête tels que l'identifiant de la requête, l'adresse IP et le numéro du port d'envoi de l'émetteur. La structure des messages est donnée dans la Section 4.

C) ***Module de Communication***

Celui-ci est en charge de l'envoi de requêtes sous-jacentes aux pairs pertinents. Le module de communication a aussi pour rôle de réceptionner les réponses venant des autres pairs et du composant interne. Enfin, le module de communication regroupe les fonctions et protocoles de connexion, de transmission et réception de requêtes.

Le module de communication a été proposé dans sa version de base SDDS2000 par Diène dans [D01]. Il se base sur la technique d'utilisation d'une fenêtre coulissante d'émission des messages. Il s'inspire de l'algorithme proposé par Van Jacobson pour le contrôle de congestion pour TCP/IP [J88]. Cette architecture a été améliorée par Rim Moussa dans [M04], en réduisant le nombre de threads utilisé, afin de répondre au besoin du prototype de LH*$_{RS}$. Dans les deux versions, l'envoi de requête se fait par UDP. Ce protocole n'est pas fiable, comme nous savons bien. De plus, dans le cas d'insertions en flux, il pouvait facilement causer un goulot d'étranglement au niveau du nœud en cours d'éclatement. Pour LH*$_{RS}$, un contrôleur de flux à fenêtre glissante d'une largeur donnée, soit W, a été rajouté. Ce contrôleur se charge d'empiler les requêtes dans une file d'attente. À chaque réception d'accusé de réception (AR) d'une requête envoyée, celle-ci est supprimée de la file d'attente. Les AR reviennent groupés, plusieurs dans un même message, pour l'efficacité de la transmission. La requête dont l'AR manque après un certain délai est renvoyée une fois de plus. Une description plus détaillée de ce contrôleur est faite dans la Section 2.4.

Exemple 1 : l'utilisateur choisit de faire une insertion d'un enregistrement. La demande est transmise par l'application. Le module de traitement de requête récupère le niveau de la fonction de hachage au niveau du client et hache la clé de l'enregistrement pour déterminer à

quelle case de données il faut envoyer la requête d'insertion. Ainsi la requête est reconstruite et prête à être envoyer au destinataire via le module de communication.

2.1.3. Serveur de Données LH $*_{RS}^{P2P}$

Le Serveur de Données LH $*_{RS}^{P2P}$ de la Figure 33, est constituée de trois principaux modules : (i) module de communication, (ii) celui de traitement des requêtes et (iii) de gestion de la case de données.

A) ***Module de communication***

Le module de communication assure l'interconnexion avec les autres nœuds et avec le composant client du pair. Le dialogue peut concerner le client d'un pair ou son serveur. Il peut s'agir alternativement d'un serveur de parité du groupe du pair. Enfin, il peut s'agir du coordinateur, surtout au moment de l'éclatement. Ce module gère la communication multicast au lancement du composant serveur. Ce dernier se met à l'écoute sur l'adresse IP 233.1.1.1 et sur le port d'écoute 10000.

B) ***Module de traitement de requêtes***

Le module de traitement de requêtes est constitué de méthodes d'analyse et de traitements spécifiques aux diverses requêtes. Ainsi, chaque requête est identifiée grâce à son numéro d'identification. Elle est alors traitée si le pair répond, et une réponse est envoyée via ce même module. La description de messages constituant les requêtes réceptionnées est faite dans la Section 4 ci-dessous.

C) ***Module de gestion d'une case de données***

Le module de gestion d'une case de données est repris du prototype LH*$_{RS}$. De même, ce dernier a repris ce module de l'implémentation d'origine. Celle-ci a fait l'objet d'une Thèse à lui seule. La Thèse avait pour but une variante de LH* dite LH*$_{LH}$ [KLR96]. La structure interne d'une case LH*$_{LH}$ accélère notablement le processus d'éclatement, par rapport à celui générique de LH*.

La Figure 34 illustre cette structure. Toute case LH*$_{LH}$ donc aussi toute case de LH $*_{RS}^{P2P}$ est organisée en pages de capacité de *b*'>> 1 articles par page. L'adresse de page, 0,1… d'un article à clé *C* dans la case résulte de LH(*C*), hachage de la clé *C*. Les paramètres de ce calcul sont liés d'une manière spécifique à celui de LH* et à l'état courant du fichier.

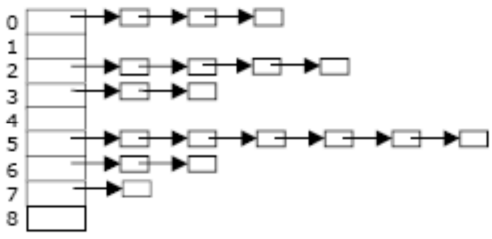


Figure 34. Organisation d'une case de données LH $*_{LH}$ versus LH $*_{RS}^{P2P}$

L'éclatement LH*$_{LH}$ ne déplace alors dans la nouvelle case que des pages entières. Il n'est pas nécessaire d'examiner la clé de chaque article, ce que suppose LH*. En conséquence, une accélération notable de la vitesse de traitement en résulte [KLR96].

Dans LH*$_{RS,}$ ce module contient en plus de LH*$_{LH}$ les méthodes nécessaires pour la mise à jour des données de parités. Pour chaque mise à jour, suppression, insertion d'enregistrement de données et recherche sûre, ce module envoi via le module de communication un message de mise à jour ou d'insertion aux pairs de parités.

2.2. Serveur de Parité LH $*_{RS}^{P2P}$

La Figure 35 présente l'architecture de notre serveur parité. Il est constitué à la base par les modules hérités du serveur de parité LH*$_{RS}$. À ceux-ci nous avons rajouté la gestion de la recherche sûre.

A) *Module de communication*

Le module de communication assure la connexion au réseau et le dialogue avec d'autres nœuds sur réseau. Le module de communication assure la communication entre les pairs serveurs de parité et serveurs de données dans le cas des mises à jour ou insertion d'enregistrement sur la case de données. Ce module empile les messages reçus dans une file d'attente utilisant un thread d'écoute UDP tel que présenté sur la Figure 38 de la Section 2.4.2. Aussi, il accuse réception des requêtes de mises à jour et insertion envoyées par le pair serveur de données. Ce module gère aussi la communication multicast au lancement du serveur de parité. Il se met en écoute sur l'adresse IP 233.1.1.2 et sur le port d'écoute 10001.

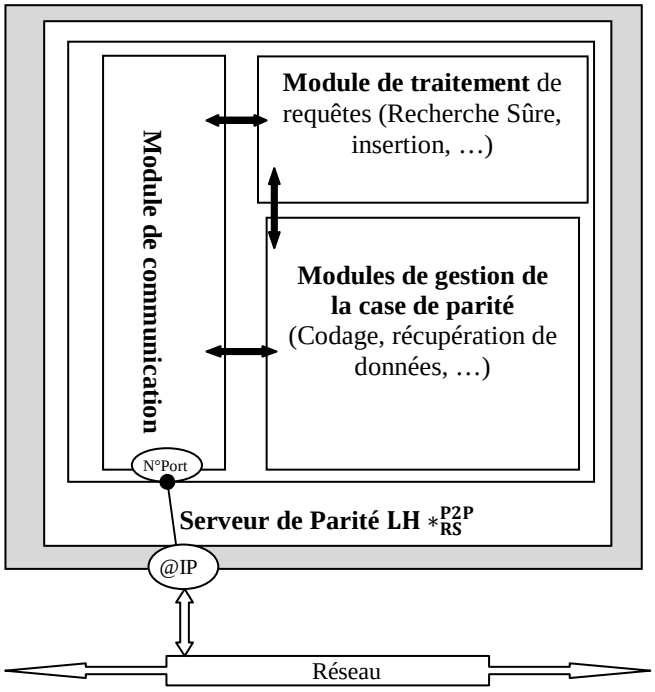


Figure 35. Architecture fonctionnelle d'un pair de parité LH $*_{RS}^{P2P}$

B) ***Module de Traitement des Requêtes***

À la réception des requêtes, le module de traitement dépile les requêtes selon la politique FIFO puis procède à l'identification de celles-ci et le lancement du traitement adéquat pour chacune d'elle. Une fois que le traitement de la requête est terminé il prépare (formatage) la réponse et l'envoie à travers le module de communication. Il appelle enfin le module de gestion de la case de parité pour la mise à jour de données de parité suite à une insertion, une mise à jour d'enregistrement de données ou en cas de reconstruction d'une case de données indisponible.

C) ***Module de Gestion de Parité***

Celui-ci gère la structure de la case de parité et le calcul de parité. Il a aussi la charge de récupérer des données en cas de panne d'une case de données. Les détails correspondant à la récupération de données sont dans [LMS04] [M04]. De plus il a pour charge la gestion de la table *RecoveredPairs* décrite ci-après, Section 3.3. Nous avons rajouté cette table pour gérer la recherche sûre.

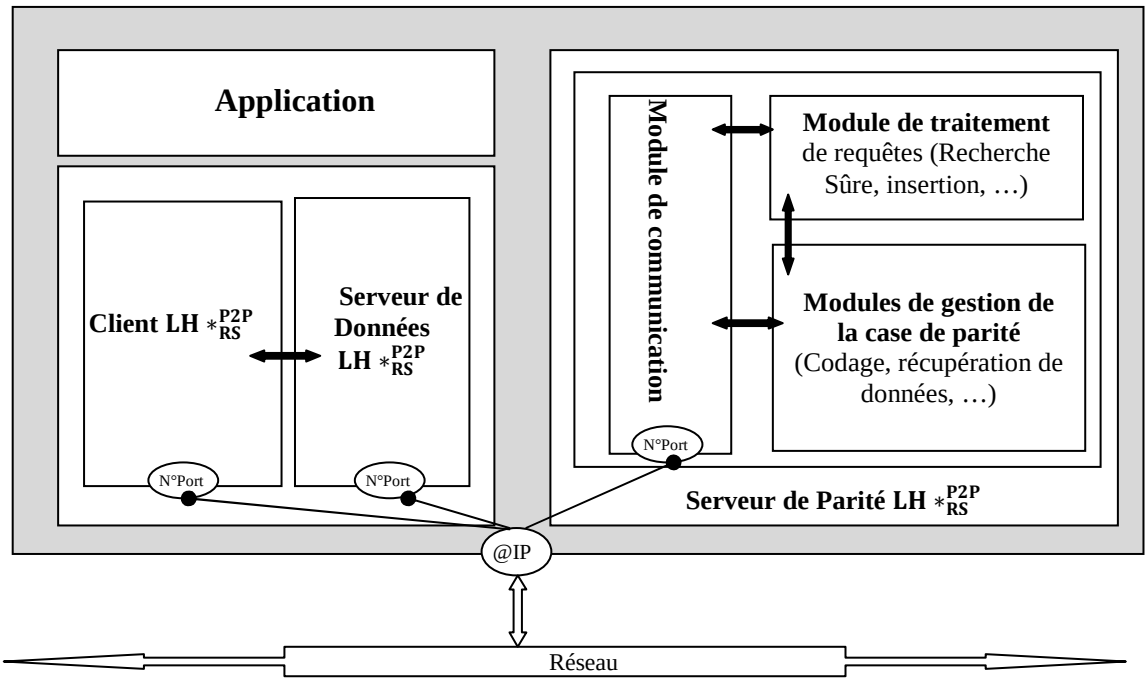


Figure 36. Architecture d'un pair LH $*_{RS}^{P2P}$ pour la variante de distribution de parité

La Figure 33 et Figure 35 correspondent à l'architecture générale sur la Figure 35 du Chapitre III. Tous les *m* pairs constituant un groupe de parité partagent le même serveur de parité. Quand le fichier subit une série d'insertions, la charge de la case de parité peut atteindre *m* fois celle de chaque pair du groupe. Une solution contre cette surcharge est en anglais le *Declustering* des enregistrements de parité décrit dans [YS08]. Chaque pair LH $*_{RS}^{P2P}$ comporte alors aussi bien la case de données que celle de parité. L'unique case de

parité du groupe de la Figure 35 est remplacée par *m* cases. Les enregistrements de parité sont distribués uniformément sur ces cases par un calcul d'adresse spécifique [YS08].

La Figure 36, présente cette dernière architecture. Chaque nœud contient le pair avec ses composants 'client' et 'serveur'. Il contient aussi le serveur de parité. Nous considérons que l'architecture d'un pair LH $*_{RS}^{P2P}$ dans sa version *déclusterisée* est celle du nœud ci-dessus. Ce pair peut être notamment un candidat. Les trois composants sont connectés au réseau via une même adresse IP. Ils écoutent par contre sur des ports différents où chaque numéro de port est calculé selon la formule donné dans la Section 3. Cette architecture cependant n'a pas été implémentée dans notre prototype. A elle-même, elle peut faire l'objet d'une Thèse dans un travail futur. Nos mesures de performance du chapitre V sont faites sur l'architecture illustrée par la Figure 33 et Figure 35.

2.3. Communication avec les Pairs Candidats

Tout pair serveur d'une case de données de LH $*_{RS}^{P2P}$ est d'abord un pair candidat. En rappel de la Section 6.2 du Chapitre III, lorsqu'un éclatement a lieu le coordinateur du fichier demande un *volontaire* parmi les candidats pour stocker la nouvelle case de données. Le coordinateur sélectionne le serveur de parité d'un nouveau groupe parmi les nœuds en disponibilité (Anaglais : *Spares*). Un serveur de parité peut ne pas être un pair.

Les modalités de cette demande sont, dans notre prototype, les mêmes que pour celui de LH*$_{RS}$. La communication se fait ainsi par multicast. Le coordinateur ne connait pas par avance tous les nœuds disponibles. Ils forment ainsi deux groupes de multicast. Le premier est le groupe des pairs candidats et le deuxième est le groupe de serveurs de parité en disponibilité. Par ailleurs, chaque groupe est caractérisé par une adresse multicast et un port d'écoute multicast. Le groupe multicast des cases de données a pour adresse IP choisie 233.1.1.1 et pour port d'écoute 10000 comme présenté en Section 2.1.3. Quant au groupe multicast de cases de parités disponibles a pour adresse multicast 233.1.1.2 et pour port d'écoute 10001 présenté dans la Section 2.2.

Pour le pair candidat, la demande (suite à éclatement d'une case de données) du coordinateur est traitée par le composant 'serveur' du pair via son module de communication. Dans les deux cas (serveur de données, serveur de parité) le premier serveur qui répond est pris. Dans ce cas il quitte son groupe multicast est se remet en écoute sur un autre port calculé selon la formule donnée dans la Section 3.

2.4. Architecture Interne d'un Pair LH $*_{RS}^{P2P}$

Le fonctionnement d'un pair LH $*_{RS}^{P2P}$ repose sur le mode de fonctionnement des ses composants logiciels. Grace à son architecture multithread il peut être en écoute, pour la réception de messages ou bien envoyer des messages avec des requêtes aux autres nœuds du système.

La Figure 37, montre l'architecture interne du composant 'client' du pair LH $*_{RS}^{P2P}$. La Figure 38 montre celle du composant serveur de données versus de serveur parité.

2.4.1. Client LH $*_{RS}^{P2P}$

Le client utilise trois threads, Figure 37, pour l'envoi ou la réception de message : (i) le Thread d'Application, (ii) le Thread de Traitement de Réponses et (iii) le Thread de Réémission de Messages.

Chaque thread a pour fonction, [M04] :

- Le Thread dit '*Thread Application*', interface le module Application (Section 2.1.1) où l'utilisateur lance sa requête via le menu offert par l'application. Le thread formate le message et l'insert dans la *Liste Messages Non Encore Acquittés*. Dès que le message est envoyé, le crédit d'envoi *Fenêtre W* est décrémenté de un. Quand le paramètre *Fenêtre* atteint 0, l'événement *ExisteZoneLibre*, devient à l'état non signalé, et ce pour arrêter les envois.
- Le Thread de Traitement de Réponse : À la réception d'un accusé, le Thread de Traitement de Réponses supprime le message correspondant à l'accusé reçu. Une cellule de *Liste Messages Non Encore Acquittés* est alors libérée, augmentant le crédit d'envoi de messages du Thread Application.
- Le Thread de Réémission de Messages vérifie les délais d'envoi des messages envoyés et non acquittés afin de les ré-envoyer.

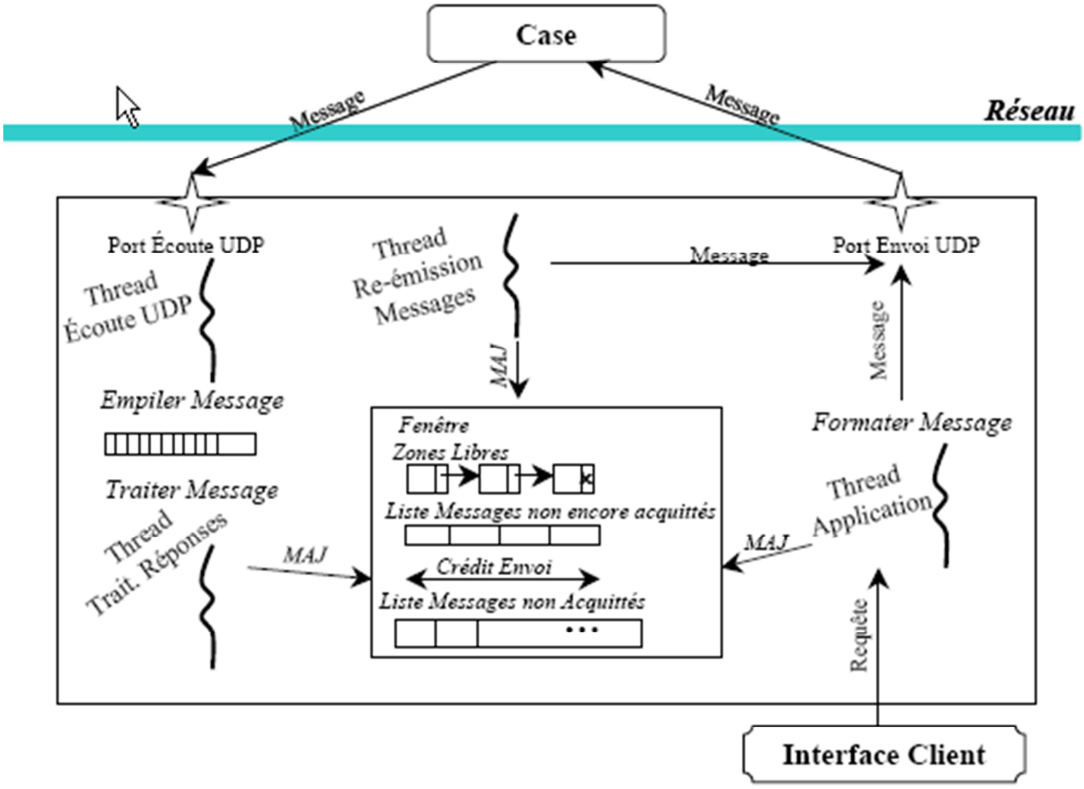


Figure 37. Architecture du client

2.4.2. Serveur de Données/Parité LH $*_{RS}^{P2P}$

La connexion entre deux pairs est assurée par le thread d'écoute TCP, Figure 38 . Une fois qu'un pair est lancé, il reste à l'affut des demandes de connexion. Pour une gestion plus performante des connexions TCP /IP l'architecture du serveur de données et de parité se base sur le concept de connexion TCP passives.

Définition d'une connexion TCP passives :

Une ouverture passive signifie que le processus de connexion se met en attente d'une demande de connexion plutôt que de l'initier lui-même. Dans la plupart des cas, ce mode est utilisé lorsque l'application est prête à répondre à tout appel. Cependant, le 'socket' distant spécifié n'est composé que de zéros (socket indéfini). Le 'socket' indéfini ne peut être passé à TCP que dans le cas d'une connexion passive, [ISI81].

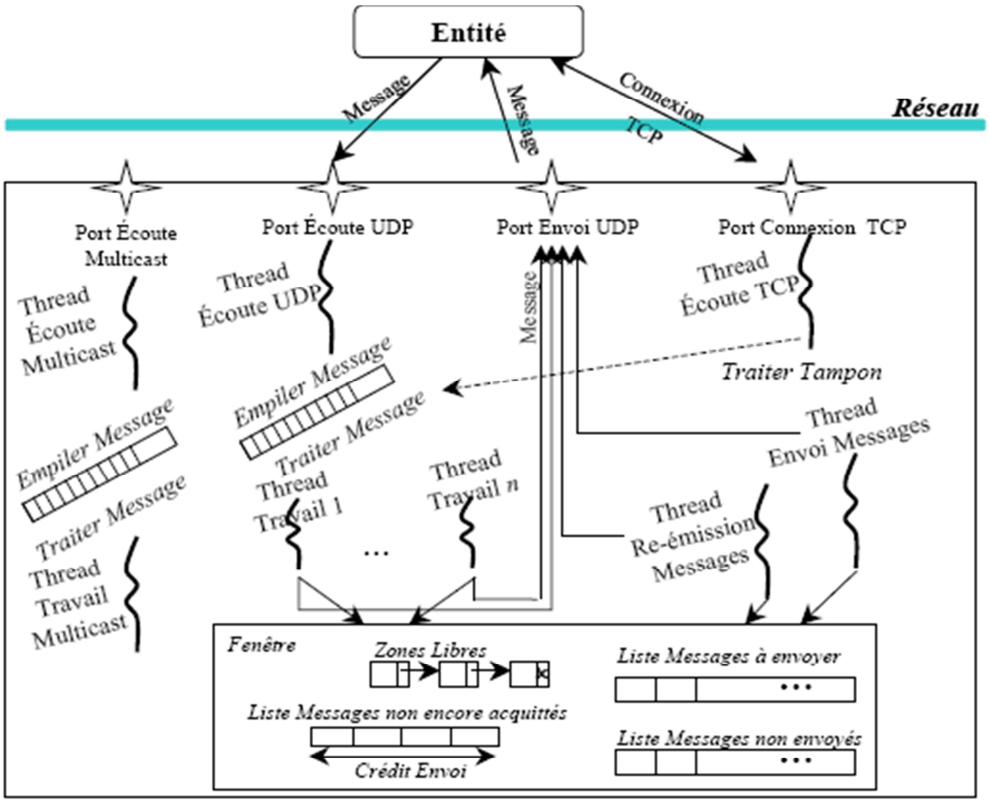


Figure 38. Architecture du serveur de données versus serveur de parité [M04]

L'architecture logicielle 'serveur' utilise différents threads :

i. Thread d'écoute multicast : celui-ci est toujours à l'écoute des messages entrant sur son port d'écoute (10000 pour les cases de données et 100001 pour les cases de parités). Chaque message reçu est empilé dans une pile de requêtes multicast. Il signale la présence de message dans la pile par envoi d'un événement au deuxième thread dit *thread de traitement* multicast. Ce dernier reste en attente d'un événement pour dépiler les messages

multicast et les traités. Pour plus de détails quant au fonctionnement des threads et les détails architecturaux voir [M04]. Ce thread est utilisé au lancement d'un pair en disponibilité.

ii. Thread d'écoute UDP : celui-ci écoute sur son port d'écoute calculé après qu'il est reçu une case de donnée ou de parité. Empile les messages reçus dans une pile, qui seront traité par des threads travails. Un thread de travail exécute la requête contenue dans le message empilé.

iii. Thread d'envoi : celui-ci se charge d'envoyer la réponse aux requêtes reçus, (résultat de recherche simple, recherche sûre…)

iv. Thread d'écoute TCP : celui-ci est utilisé lors de transfert de données d'éclatement (enregistrements) dans le cas d'un serveur de données et des mise à jours de données de parités dans le cas d'un serveur de parité.

Voir [M04] pour plus de détails.

3. Tables d'Adressage

Nous détaillons ci-après les différentes tables d'adressage utilisées dans notre prototype. Elles servent à la gestion de divers paramètres que nous présentons aussi.

3.1. Adressage d'un Nœud LH $*_{\mathbf{RS}}^{\mathbf{P2P}}$

Nous allons présenter dans cette section les nouvelles tables, introduites pour stocker et récupérer les paramètres d'adressage.

Nous avons réutilisé l'adressage du prototype LH*$_{RS}$. Il s'agissait notamment du calcul automatique du numéro de port pour l'envoi ou la réception de messages. Ce calcul utilisait l'identificateur de chaque nœud LH*$_{RS}$, dit *numéro d'entité*. Le coordinateur d'un fichier LH*$_{RS}$ attribut celui-ci au moment où un nœud LH*$_{RS}$ disponible devient un nœud du fichier. Au moment de l'enregistrement, tout nœud avec le numéro d'entité *x* a les numéros de port suivants :

- Pour le *port d'écoute UDP* : 6000 + (3**x*),
- Pour le *port d'envoi UDP* : 6000 + (3**x*) + 1
- Enfin, pour le *port de connexion TCP/IP* : 6000 + (3**x*) + 2.

Dans un fichier LH*$_{RS}$, tout nœud client et tout nœud serveur avaient un numéro d'entité différent. Dans le cas de LH $*_{RS}^{P2P}$, un numéro d'entité caractérise chaque pair, étant donc le même pour ses composants 'client' et 'serveur'. Le coordinateur donne ce numéro au moment de l'enregistrement du pair candidat. Enfin le coordinateur procède de manière similaire pour un serveur de parité.

Comme dans LH*$_{RS}$ le coordinateur LH $*_{RS}^{P2P}$ attribue aussi un numéro logique à chaque nouvelle case de données. Il l'utilise ensuite pour l'adressage les éclatements et la gestion des groupes de parité, comme pour LH*$_{RS}$. Néanmoins, dans LH*$_{RS}$ l'attribution de deux numéros au serveur était synchrone. Le coordinateur LH $*_{RS}^{P2P}$ attribue par contre ce numéro, plus tard au moment où un éclatement se produit.

3.2. Table d'Adressage du Coordinateur

Pour gérer les numéros d'entités, le coordinateur utilise la table dite '*Adresse des entités*' noté 'TAE' en abrégé. TAE est indexée par le numéro d'entité, Figure 39. Chaque élément de la 'TAE' possède deux champs : (Type Entité, Adresse IP). Type Entité est égale à :

- 0 si le nœud est un pair serveur donc porte une case de données.
- 1 si le nœud est un serveur de parité.
- 2 si c'est un pair candidat.

Le deuxième champ indique l'adresse IP du nœud.

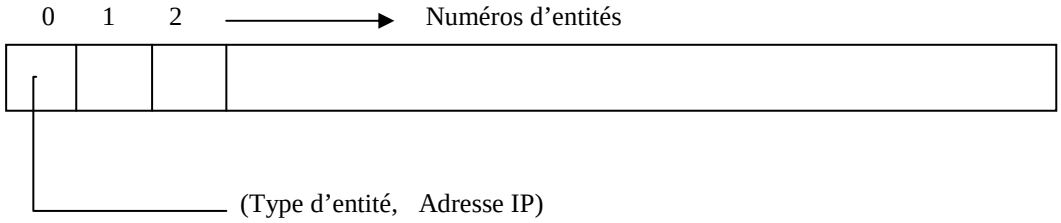


Figure 39. Table des Adresses des Entités (TAE)

Pour gérer les cases de données le coordinateur possède une deuxième table dite '*Table des Cases de Données*' et notée 'TCD' en abrégé. Comme le montre la Figure 40, cette table est indexée par les numéros logiques. Un élément *i*, *i*=1,2… de TCD contient le numéro d'entité du pair avec sa case de donnée *i*. Le couple TAE et TCD établit la correspondance entre chaque case de données et l'adresse IP de son pair.

Le coordinateur dispose enfin de la table dite '*Table des Tuteurs et Pupilles*' noté 'TTP' et illustré par la Figure 41. TTP est indexée par les numéros logiques des tuteurs *j* = 0,1,…. Le coordinateur utilise TTP pour gérer l'attribution de tuteurs aux pairs candidats. Chaque élément de TTP[*j*], contient une table de structure d'enregistrement dite '*Table Pupille*' et noté TP car un tuteur peut avoir zéro ou plusieurs pupilles. Chaque élément de TP décrit un pupille par deux champs :

NumeroEntite : c'est le numéro d'entité du pupille, pour le calcul de son port.

AdresseIP : est l'adresse IP du pupille.

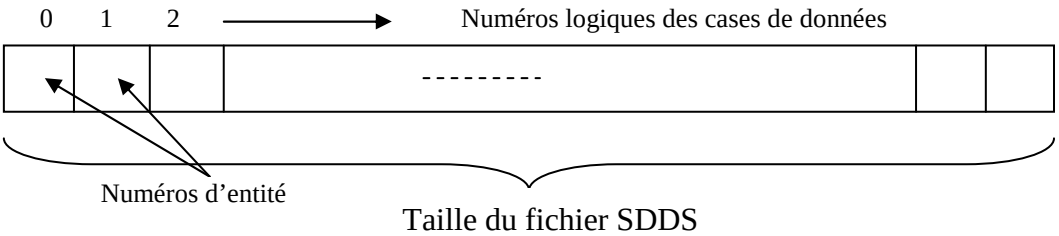


Figure 40. Table des Cases de Données (TCD)

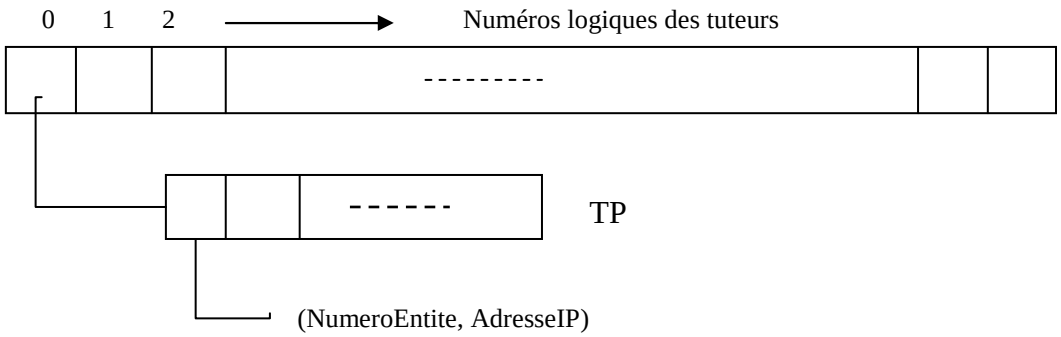


Figure 41. Table des Tuteurs et Pupilles (TTP)

3.3. Table d'Adressage de la Recherche Sûre du Serveur de Parité

Comme nous l'avions déjà indiqué dans le chapitre précédent, la recherche sûre nous permet de palier au problème du Churn. Pour ce faire, nous avons implémenté une table dite *RecoveredPeers* (noté RP) sur le pair de parité. RP[*k*] tel que k= 0,1,… contient le couple B (*numéro de la case, adresse IP pair*), comme c'est illustré par la Figure 42. Le *numéro de la case* correspond au numéro de la case de données indisponible suite à une panne ou autre raison. L'*adresse IP pair* dans le couple est l'adresse IP du nouveau pair portant la case reconstruite. La gestion de cette table est faite sur le manager du groupe de pairs de parité, soit le premier pair de parité.

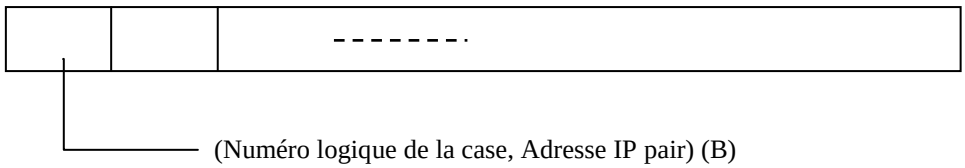


Figure 42. Table des Cases Reconstruites (RP)

3.4. Table d'Adressage du Pair Pupille

Le pair candidat après son enregistrement auprès du coordinateur reçoit son numéro d'entité. Le pair est toujours candidat s'il n'a pas de case de données à recevoir immédiatement. Auquel cas, le coordinateur choisit alors un pair serveur qui devient le tuteur du candidat. Le candidat lui-même alors devient un pupille du tuteur. Pour échanger des messages avec son tuteur il a une structure de table dite '*Table d'Adressage du Pupille*' noté TAP comme nous le montre la Figure 43. La TAP permet d'identifier aisément son tuteur, lors de son enregistrement ou lorsqu'il quitte son tuteur. De plus il utilise cette table pour adresser les autres pairs du réseau.

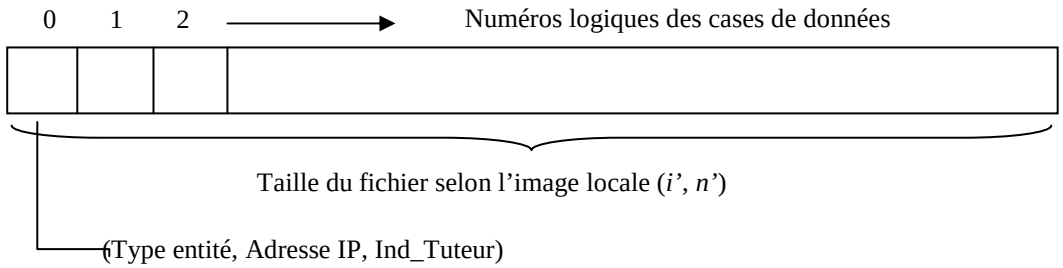


Figure 43. Table d'Adressage Pupille (TAP)

La TAP est créée quand le pair candidat reçoit son numéro d'entité. Elle est mise à jour lorsque que le pair tuteur contacte le pupille. Ce dernier enregistre toujours le numéro d'entité et l'adresse IP de son tuteur dans TAP[0] avec l'indice Ind_Tuteur à 1. Les autres cases TAP[*h*] tel que *h* = 1,2,… contiennent les numéros des entités et adresses IP des autres nœuds du fichier SDDS avec Ind_Tuteur à 0. La TAP est mise à jour ;

- Lors d'une erreur d'adresse et réception d'IAM,
- Lorsque le pupille change de tuteur
- Lorsque le pupille reçoit une case de données et quitte son tuteur.
- Enfin, lorsque le Pupille reste sans activité et entre temps le fichier a évolué alors il fait une demande de mise à jour de TAP auprès du coordinateur. Ce dernier envoie la liste complète des pairs du fichier.

Comme le pair pupille n'a pas de case de données à gérer alors il n'a pas besoin de connaitre les adresses IP et numéros d'entités des pairs de parités. La table TAP est utilisée pour utiliser le fichier SDDS jusqu'à la déconnexion du pair. Cette table n'est pas répliquée ni reconstruite sur un autre pair donc en cas de panne autre que la déconnexion, elle est perdue. Dans ce cas une demande de rafraichissement auprès du coordinateur est nécessaire à chaque remise en connexion du pair après une déconnexion inopinée ou reconnexion après une panne.

3.5. Table d'Adressage du Pair de données

Un pair LH $*_{RS}^{P2P}$ serveur de données, ayant reçu une case de données continue à utiliser la table TAP. La seule différence est que le champ Ind_Tuteur de la TAP est à 0 car il n'a plus de tuteur.

Ce pair peut être néanmoins un tuteur à son tour. Dans ce cas il enregistre la liste de ses pupilles. Dans un premier temps nous avons utilisé une table T ayant la même structure que TP de la Figure 41. Chaque tuteur *t* enregistrait ses pupilles dans T. Nous avons réalisé que T présentait un problème de disponibilité pour notre prototype. Dans celui-ci, comme pour LH*$_{RS}$, en cas de panne d'un pair serveur, la case était reconstruite sur un autre pair. Mais la protection par les enregistrements de parité ne concernait que les enregistrements de données

dans la case. En conséquence l'indisponibilité du tuteur implique la perte de la liste des pupilles enregistrés dans la table T car T est en dehors de la case de données.

Nous avons alors élaboré une autre solution comme suit. Nous avons structuré la liste des pupilles en un enregistrement de données du fichier ayant la liste comme valeur non clé le numéro d'entité du pair tuteur *t* comme clé. Cet enregistrement particulier restera toujours sur le tuteur. Nous l'insérions comme un enregistrement, dit *tutor record* noté *TR*, quelconque sauf que cet enregistrement a pour clé le numéro logique de la case de données. Aucun éclatement ne pouvait le migrer par la suite. Nous décrivons dans la Section 5.2 et 5.1 le protocole et la mise en œuvre de cette solution. Le résultat a été que la liste est devenue aussi hautement disponible que toutes les autres données du fichier. C.-à-d. la liste est devenue également *k*-disponible.

4. Structure de Messages

Nous décrivons maintenant la structure des nouveaux messages que nous avons créé pour la communication entre les nœuds de LH $*_{RS}^{P2P}$. Nous discutons seulement les champs essentiels de chaque message. Notons que ces message son envoyés en mode UDP. Il faut noter dans ce qui suit que les IDMessage sont définit en amont dans un fichier '*header, .h*' et chaque IDMessage est unique et caractérise un type de message.

4.1. Déclaration de Candidature

C'est le message de déclaration de la candidature d'un nouveau pair. Le pair qui veut rejoindre le fichier adresse sa requête par l'envoi d'un message noté *PairCandidat* au pair coordinateur. Sa structure est la suivante :

PairCandidat(IDMessage, AdresseIP)

IDMessage : identifiant du message. Tout IDMessage est un numéro séquentiel unique sur deux octets.

AddresseIP : c'est l'adresse IP du nœud. Elle est sur quatre octets (comme toutes les adresses IP).

La longueur du message est fixe et de six octets au total.

4.2. Message du Coordinateur au Tuteur

Dès que le coordinateur reçoit une demande d'enregistrement d'un pair candidat, il hache son adresse IP. Le résultat est le numéro logique du futur tuteur du candidat. Le coordinateur envoie à celui-ci le message dit *NouveauPupille*. Le message est d'une longueur fixe et sa structure est comme suit :

NouveauPupille(IDMessage, NumeroEntite, AdresseIP),

où :

IDMessage : identifiant du message.

NumeroEntite : le numéro d'entité attribué par le coordinateur au candidat selon le processus présenté en Chapitre III Section 7. Sa taille est de deux octets.

AdresseIP : adresse IP du pair candidat.

Le coordinateur attend l'acquittement du message. La stratégie d'acquittement est la même que pour LH*$_{RS}$. Voir la structure du message d'acquittement dans [M04].

4.3. Message du Tuteur au Pair Candidat

Une fois que le coordinateur a désigné un tuteur, ce dernier envoie le message dit *PriseEnCharge*, au pair candidat. Ce message indique à son nouveau pupille le numéro d'entité et l'adresse IP du tuteur. Ce message est aussi de taille fixe (neuf octets), sa structure est la suivante :

PriseEnCharge(IDMessage, NiveauJ, NumeroLogique, NuméroEntité, AdresseIPTuteur), où :

IDMessage : identifiant du message.

NiveauJ : niveau ' *j*' du tuteur, sur un octet.

NumeroLogique : numéro de la case du tuteur, nécessaire pour l'exécution de l'algorithme A3 sur le candidat.

NumeroEntite : numéro d'entité du tuteur.

AdresseIPTuteur : est l'adresse IP du tuteur.

4.4. Message de Notification d'Éclatement

Le tuteur envoie cette notification à chacun de ces pupilles lors de l'éclatement de sa case. Ce message est acquitté à réception. Le pupille met à jour alors son image client. Sa structure est comme suit :

MiseAJourTuteur(IDMessage, NF_j, NumeroLogique, AdresseIPTuteur,NumeroEntité)

IDMessage : identifiant du message.

NF_j : niveau '*j*' de la case qui éclate.

NumeroLogique : numéro de la case qui a éclaté, nécessaire pour l'exécution de A3.

AdresseIPTuteur : adresse IP du tuteur. nous rappelons qu'un éclatement peut faire migrer le tutorat vers le nouveau pair

NumeroEntite : numéro d'entité du tuteur (qui va servir au calcul du port d'écoute sur le pupille).

4.5. Message de Notification de Fin de Tutorat

Un pupille envoie cette notification à son tuteur quand il devient le serveur d'une case, comme décrit dans la Section 7 du Chapitre III. Ce message est acquitté à la réception. Le tuteur supprime alors le pupille de son enregistrement de tutorat *TR*.

La structure du message est la suivante :

FinTutorat(IDMessage, AdresseIP, NumeroEntite) où :

IDMessage : identifiant du message.

AdresseIP : adresse IP pupille.

NumeroEntite : numéro d'entité du pupille.

4.6. Message de la Recherche Sûre

La structure du message de la recherche sûre (requête Q1 de la Section 5.2 du Chapitre III) se présente comme ci-après. Ce message est envoyé par un pair quelconque via son composant 'client'. Il se présente comme suit :

RechercheSure(IDMessage, Key) où :

IdMessage : identifiant du message.

Key : la clé de l'article recherché.

Après réception et identification du message par le pair portant une case de données, ce dernier envoie à son tour un message pour le gestionnaire du groupe de parité. Le message correspond à la demande D1 décrite dans la section 4.2 et utilisée dans l'algorithme A5 du chapitre III. Sa structure est la suivante :

AmIOK(IDMessage,NumeroLogique) où :

IdMessage : Identifiant du message.

NumeroLogique : numéro logique de la case de données.

Le gestionnaire du groupe de parité vérifie le message ci-dessous, analyse de l'idMessage pour savoir si c'est une recherche sûre. Il lance alors une recherche sur sa table RP décrite en haut. Le résultat de cette recherche est envoyé dans le message ci-après :

RpAmIOK(IDMessage, Bool, AdresseIP) où

IdMessage : Identifiant du message.

Bool ; est variable booléen qui est à 'Faux' si la case de donnée du pair émetteur a été reconstruite sinon à 'Vrai'

AdresseIP : l'adresse IP du pair ayant la case reconstruite si Bool est à 'Faux' et égale à chaine de caractère vide sinon.

Ce message est acquitté à la réception par le gestionnaire et l'envoi ce fait par UDP.

4.7. Message d'Insertion d'Enregistrement de Tutorat

On distingue trois types d'insertions selon le type de données, tutorat et de parité, l'insertion de données et parité est la même que pour LH*$_{RS}$. Nous avons rajouté l'insertion d'enregistrement de tutorat pour palier au problème de disponibilité de la liste des pupilles pour un tuteur évoqué dans la Section 3.5.

Dès que le tuteur reçoit un pupille, il lance une insertion interne dans sa case de l'enregistrement tutorat. La clé de l'enregistrement est le numéro logique de la case du tuteur. Le scénario de l'enregistrement est présenté dans la Section 5.1. Le message d'insertion d'enregistrement de tutorat est comme suit :

InsererPupille(IDMessage, NumeroLogique, EnregistementTutorat) où

IdMessage : identifiant du message.

NumeroLogique : le numéro logique de la case recevant le pupille.

EnregistementTutorat : est constitué à son tour d'un couple (Clé, données) où le premier champ désigne la clé de l'enregistrement et le second le champ non clé qui désigne les données, c'est la même structure que celle présentée dans la Section 5.2 du Chapitre III.

4.8. Message Recherche Scan

Après qu'un pair ait récupéré la liste des cases concernées par la recherche scan, il leurs envoie un message point à point, unicast. Chaque pair recevant ce message exécute l'algorithme A4 décrit précédemment. Le message se présente comme suit ;

RechercheScan(IdMessage, NiveauJ, Destinataire) où :

IdMessage : identifiant du message

NiveauJ : est le niveau de la fonction de hachage *j* de case sur le pair émetteur du message.

Destinataire : est une structure contenant l'adresse IP et le numéro du port du destinataire.

Notons que cette la recherche Scan n'a pas été implémentée, mais nous avons mis en place l'algorithme A4 (Section 4, Chapitre III) et sa structure de message.

5. Protocoles d'Échange de Messages de LH $*_{\mathbf{RS}}^{\mathbf{P2P}}$

Nous allons présenter les protocoles d'échange de messages que nous avons rajouté pour assurer les nouvelles fonctions implémentées. Ces protocoles sont décrits par des scénarios présentés ci-dessous. Nous utilisons les diagrammes de séquence pour illustrer l'échange de messages entre les nœuds dans le temps.

5.1. Enregistrement de Pairs Candidats

Un nouveau nœud peut se déclarer comme candidat pour être le pair LH $*_{\mathrm{RS}}^{\mathrm{P2P}}$, ou seulement le 'serveur' ou, enfin, seulement le 'client'. Dans le cadre de nos travaux, nous nous intéressons au premier cas seulement. Les deux autres sont traités de la même manière que LH*$_{RS}$, la Figure 44, présente le traitement fonctionnel d'une telle requête. Dans notre système, un pair candidat adresse sa demande, par un message noté *PairCandidat*, au coordinateur. Cette approche est la plus simple par sa compatibilité avec le protocole d'adjonction d'un 'client' implémentée sous LH*$_{RS}$. D'autres stratégies sont également possibles, où le candidat solliciterait un pair ou un serveur de sa connaissance. Le coordinateur commence par hacher l'adresse IP du candidat comme s'il s'agissait d'une clé d'un article. Le résultat sera l'adresse logique $1, 2 ... N$ du pair ou serveur tuteur. Le coordinateur envoie ensuite un message, noté *NouveauPupille*, au tuteur. Il y inclut les coordonnées du candidat. Le tuteur accuse la réception au coordinateur. Puis, il prend contact avec le pupille à son tour. Il lui envoie le message, noté *PriseEnCharge*. Le pupille accuse enfin la réception de ce message au tuteur.

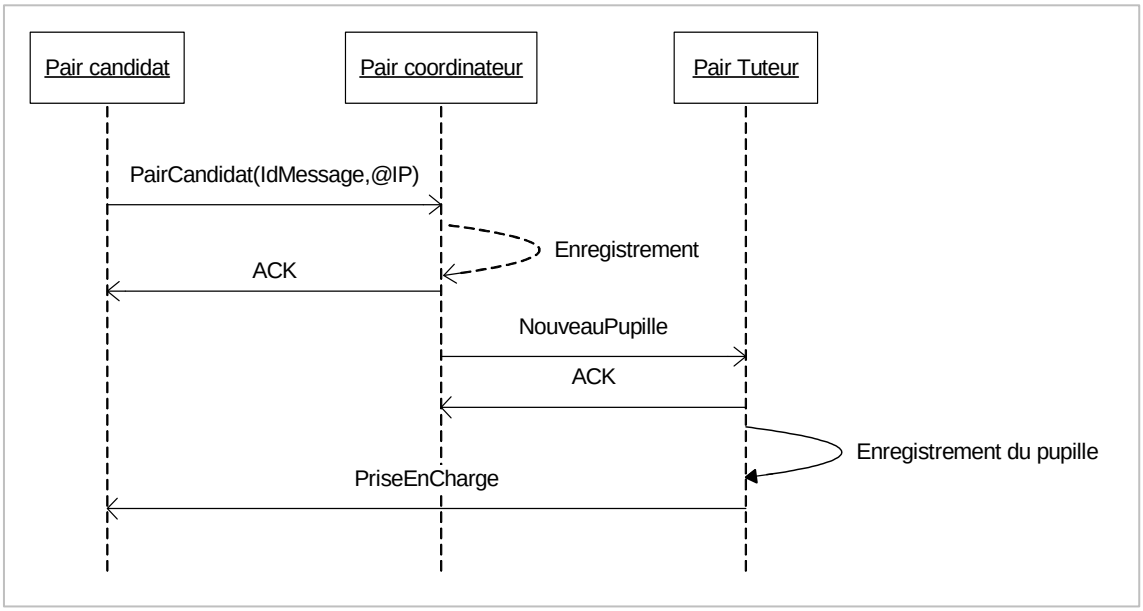


Figure 44. Protocole d'adjonction d'un pair candidat

Pour pallier le problème de la récupération de la liste des pupilles lors d'une panne du tuteur. Nous avons proposé une autre solution plus efficace qui nous permet de reconstruire cette liste. Nous avons dû transformer la table P de la Figure 41, en un enregistrement de données, mais ayant pour données la liste des pupilles. Cette liste est enregistré dans le champ non clé de l'enregistrement de données. La clé d'un tel enregistrement n'est que le numéro logique de la case. En effet ayant pour clé le numéro logique de la case, cela permet à l'enregistrement de rester sur la case malgré les éclatements qui par hachage distribuent la charge entre la case qui éclate et la nouvelle case. Dans le cas où un pair tuteur reçoit un IAM suite à une erreur d'adressage ou un éclatement de sa case, il consulte son enregistrement de pupilles en utilisant le numéro logique de sa case qu'il connait. Alors il extrait la liste des pupilles auxquels il envoie la mise à jour. Cette nouvelle solution permet à coup sûr de reconstruite et les données et la liste des pupilles.

Cependant le pair tuteur, en plus de gérer les enregistrements de données, a la charge la gestion de sa liste de pupilles. Les pupilles sont ajoutés dans la liste au fur et à mesure de leurs arrivées avec une simple concaténation et un caractère séparateur. Nous avons pris comme caractère '|'.

Exemple :

Un enregistrement ayant pour clé 1 de la case numéro 1 avec l'adresse IP 192.168.0.1. Soient les pupilles 1, 2, 3, avec leurs couples (numéro du port, adresse IP), (6200,192.168.0.2), (6201,192.168.0.3), (6203,192.168.0.4).

La liste des pairs pupilles sera sauvegarder comme suit dans l'enregistrement des pupilles au niveau du tuteur avec un séparateur '|' ;

{1,(6200,192.168.0.2| 6201,192.168.0.3|6203,192.168.0.4)}

À partir de ce moment le pair candidat peut commencer à travailler en tant que 'client'. Nous rappelons que sous le prototype LH*$_{RS}$ existant, il doit néanmoins, demander encore au coordinateur les adresses IP des serveurs des cases courants. Le candidat se met aussi en disponibilité d'allocation de la (nouvelle) case, à l'un des prochains éclatements. Sous le prototype LH*$_{RS}$, cette attente se traduit par l'écoute sur le port UDP d'un message d'appel général de la part du coordinateur. La case est allouée au premier répondant, les autres restent en disponibilité

5.2. Insertion d'Enregistrement de Tutorat

Le protocole d'insertion, insère soit un enregistrement de données, comme pour LH*$_{RS}$, soit celui de tutorat. Pour rappelle (Section 3.4 et 5.2), ce dernier contient la liste des pupilles affectés au pair tuteur.

La Figure 45, rappelle le protocole d'insertion d'un enregistrement de données. L'insertion dans la case correcte se propage sur les *k* cases de parités du groupe. À la réception de l'enregistrement de données par le pair *d* un rang *r* lui est affecté. Puis un message de mise à jour des enregistrements de parité ayant pour clé *r* est envoyé aux pairs de parité. Nous n'aborderons pas davantage de détails présentés dans [M04].

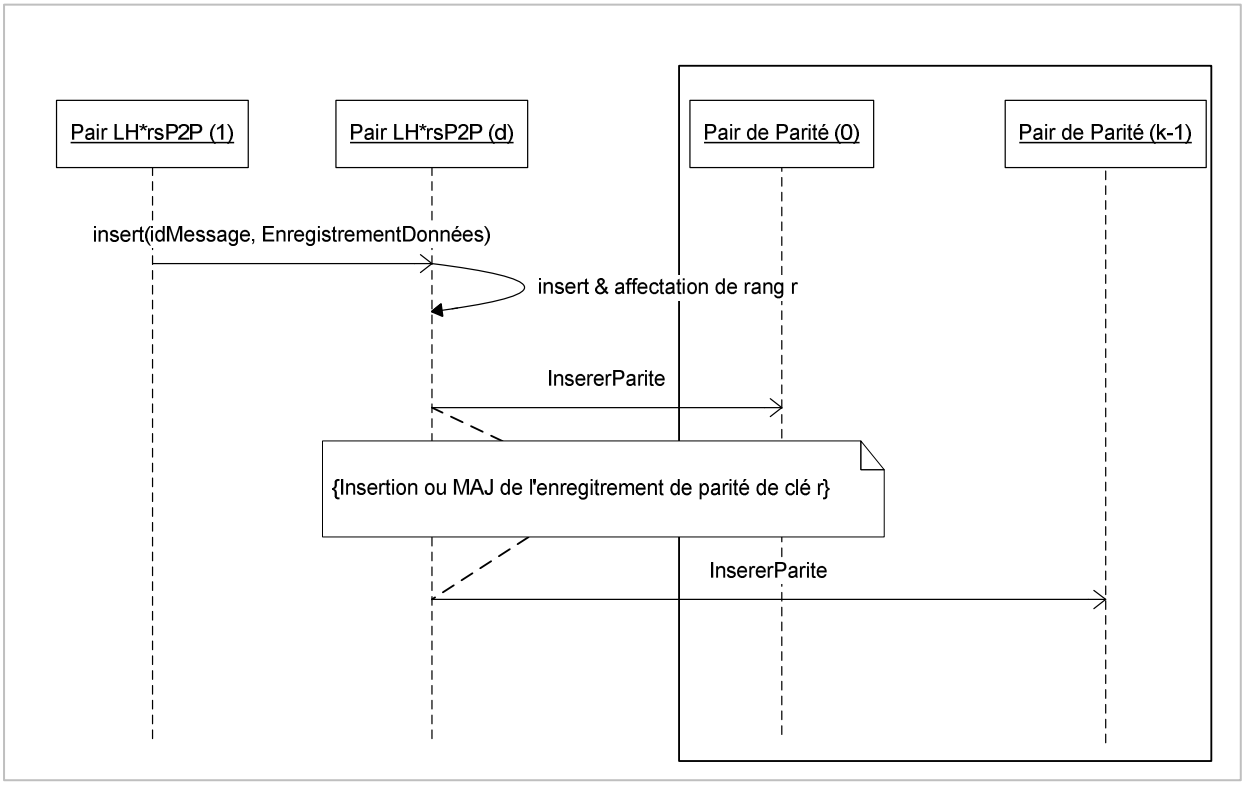


Figure 45. Insertion d'un enregistrement de données

Nous avons distingué deux stratégies pour la gestion de l'insertion de l'enregistrement de tutorat. La première est sa création au moment de l'affectation d'une case de données au pair candidat. Autrement dit, quand un éclatement arrive, la case est initialisée avec l'enregistrement de tutorat vide ayant pour clé le numéro logique de la case de données. Cette

solution n'est pas optimale car nous ne pouvons pas avoir de pairs pupilles durant toute l'activité du pair. Dans ce cas l'enregistrement de tutorat sera vide. Ce qui obligerait à le consulter lors du prochain éclatement de la case ou lors de la réception d'un IAM. Il s'agit d'une étape supplémentaire, inutile et qui compliquerait davantage la gestion du code source.

La seconde stratégie, que nous avons adoptée et implémentée, consiste à créer l'enregistrement lors de la réception du premier pupille et de le supprimer après le départ de tous les pupilles. Cette solution présente l'avantage d'éviter toute étape inutile dans l'algorithme.

Comme le montre la Figure 46, dès que le pair tuteur reçoit un nouveau pupille du coordinateur, *NouveauPupille*, il lance une insertion de ce pupille dans sa liste. À la différence d'un enregistrement de données, l'enregistrement de tutorat est inséré par le tuteur lui-même. Il le crée s'il n'existe pas en lui donnant comme clé le numéro de sa case de données sinon il le met à jour. Cette mise à jour consiste à insérer dans l'enregistrement de tutorat les coordonnées du nouveau pupille comme présenté en Section 4.2. Cette insertion se répercute aussi sur toutes les cases de parité comme pour l'insertion de données.

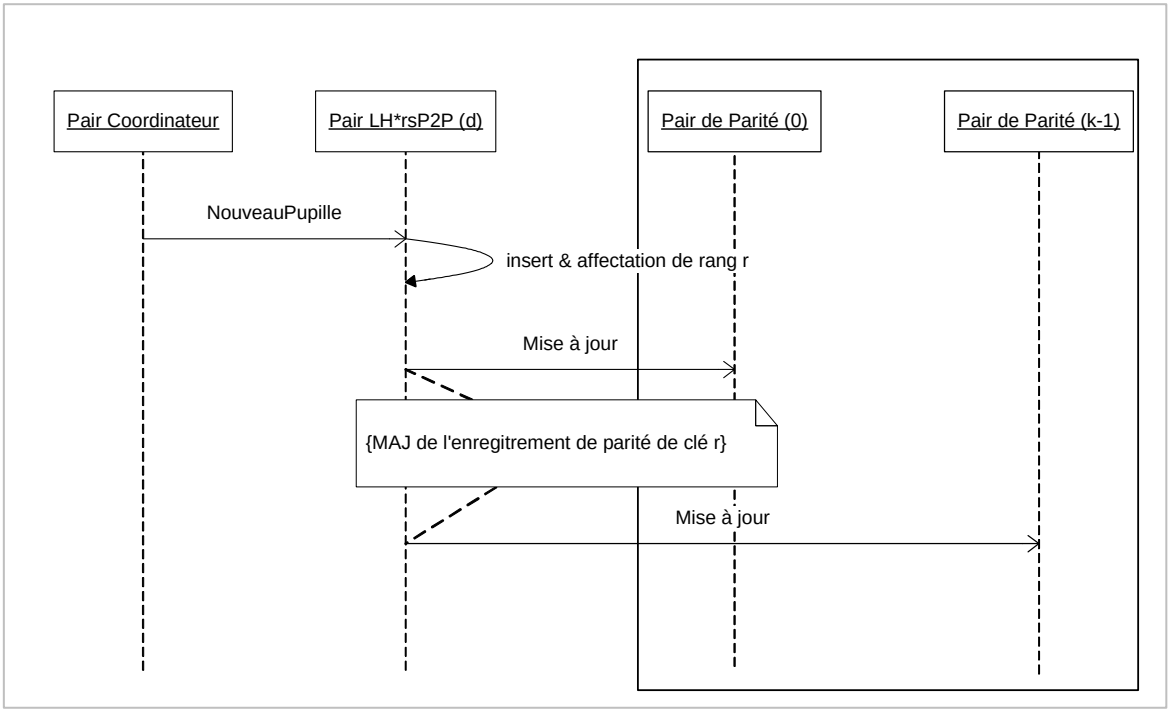


Figure 46. Insertion d'un enregistrement de tutorat

5.3. Recherche Scan

Un pair LH $*_{RS}^{P2P}$ qui veut lancer une recherche scan doit récupérer la liste des pairs portant les cases numéroté $a = 1, 2, \dots, n' + 2^i - 1$. Cette liste peut ne pas être à jour car cette celle-ci est mise à jour manuellement par demande auprès du pair coordinateur via l'application. Seul le coordinateur connait les adresses de tous les nœuds constituant le fichier SDDS. Une contrainte héritée du prototype LH*$_{RS}$. Si la liste n'est pas mise à jour au moment de l'envoi

d'une recherche scan il y aura un message d'erreur indiquant que le/les pair(s) n'existe/n'existent pas. Dans la description du scénario de recherche scan par le diagramme de séquence de la Figure 47, nous négligeons la présence des pairs de parité. Un pair quelconque peut lancer la recherche scan. Il envoie son message à tous les pairs qu'il connait. Chaque pair recevant ce message doit accuser réception par envoi d'un ACK, sinon il est déclaré comme indisponible et dans ce cas sa reconstruction est imminente. Le destinataire, après avoir accusé réception il exécute l'algorithme A4 de la recherche scan.

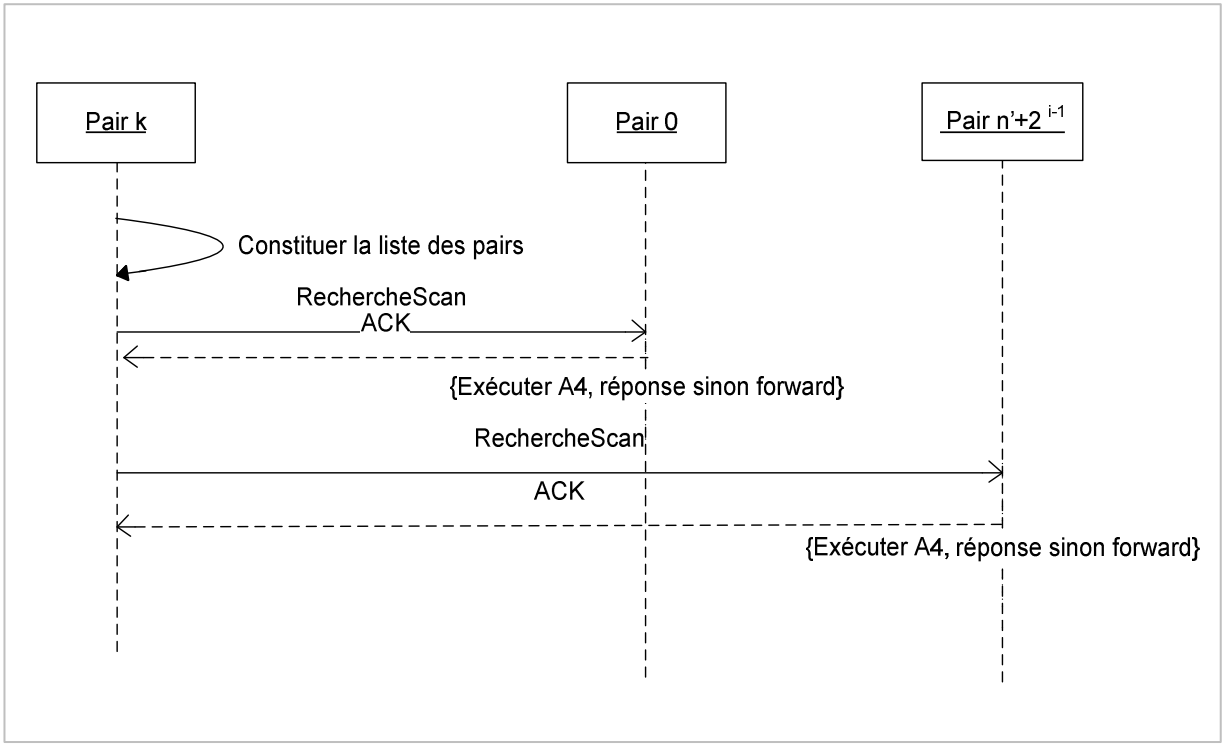


Figure 47. Recherche scan

5.4. Éclatement de Pair

Les opérations de recherche, d'insertion, de mise à jour et de suppression d'article de LH $*_{RS}^{P2P}$ sont les mêmes pour LH$*_{RS}$, détails dans [M04]. Il n'est pas tout à fait de même pour l'opération d'éclatement qui peut suivre celle d'une insertion. LH $*_{RS}^{P2P}$ comporte en effet une phase supplémentaire, Figure 48. Durant cette phase, il transmet au nœud avec la nouvelle case les adresses IP de pupilles qui seront désormais à sa charge. Celles-ci sont, pour rappel, toutes les adresses IP de candidats hachés vers la nouvelle case après l'éclatement. Le nouveau nœud prend désormais à sa charge le tutorat. Il envoie alors un message, *MiseAJourTuteur*, unicast à chacun de ces pupilles. Chaque pupille accuse la réception et met à jour son image.

Le nœud de l'éclatement rafraîchit par ailleurs chaque pupille qu'il avait sous son tutorat avant l'éclatement. Comme le montre la Figure 48, il utilise le message noté *MiseAJourTuteur*. En plus, il rafraîchit encore l'image interne de son 'client' par le même message.

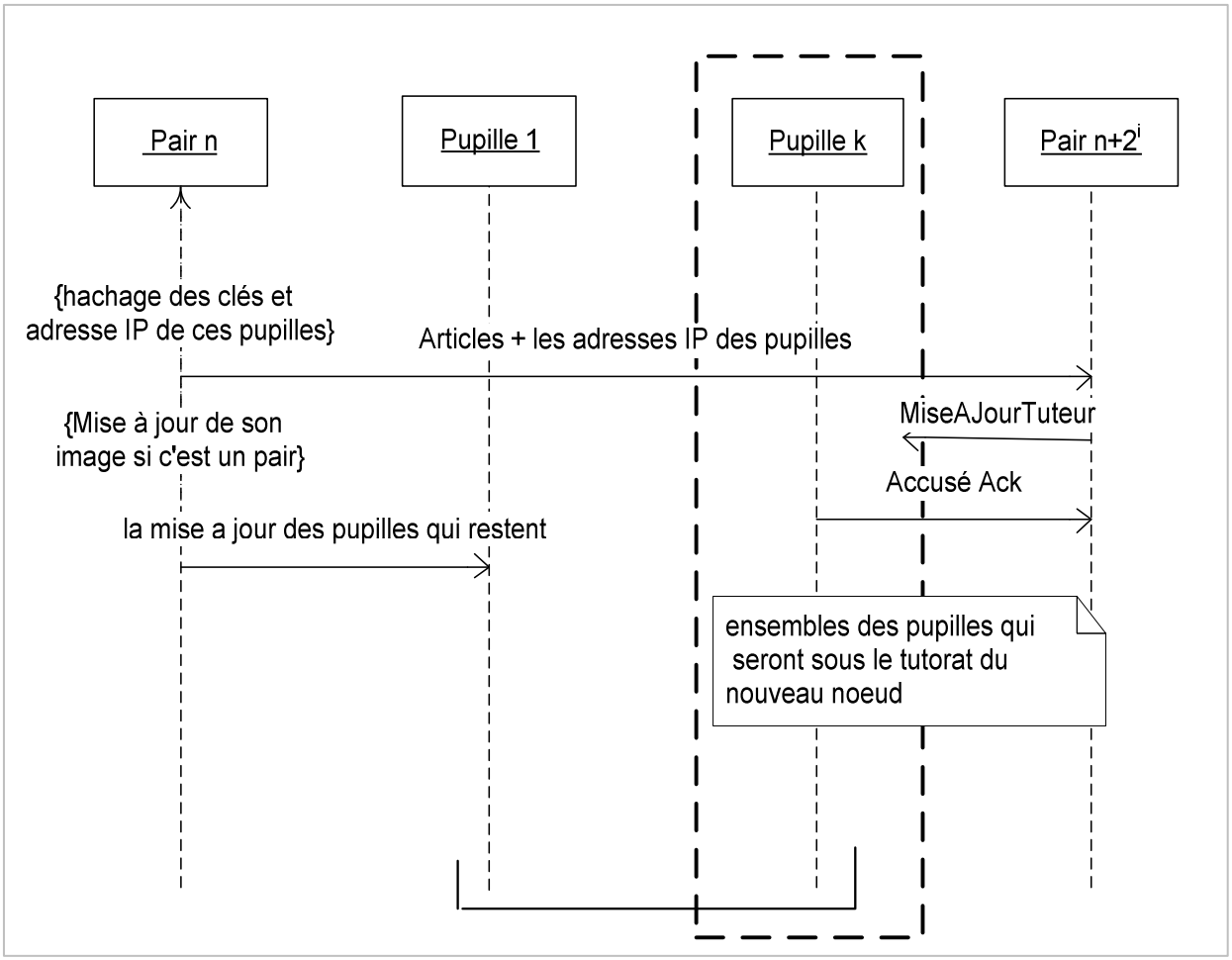


Figure 48. Éclatement dans LH $*_{RS}^{P2P}$

Les pupilles rafraîchissent leurs images. Les pupilles vérifient si leur tuteur n'a pas changé. Tout pupille peut le faire à l'évidence à partir de l'image. Dans la version actuelle du prototype, chaque pupille ou 'client' local du pair LH $*_{RS}^{P2P}$ demande ensuite au coordinateur l'actualisation des adresses physiques (IP) des nœuds du fichier. Chaque pupille dispose le cas échéant dans cette liste reçue l'adresse IP du nouveau tuteur.

Il distingue deux stratégies d'éclatements. La première est l'éclatement forcé, qui consiste à éclater la case d'un pair pointé par le pointeur d'éclatement du pair coordinateur à chaque arrivé d'un nouveau pair candidat. Alors, une case de données est attribué aux pairs au fur et à mesure de leurs arrivés. Cette stratégie a pour avantage de ne pas garder de pair candidat et de ne pas avoir à gérer une liste de pupilles potentiels. Mais elle a pour inconvénient de laisser des cases de données vides et une gestion de ces cases en plus.

La deuxième qui consiste à éclater la case d'un pair à la suite d'un débordement est la plus classique. Ce cas est comme celui du schéma de LH*, qui garantit un taux de remplissage des cases de près de 70%. Pas de perte d'espace avec des cases vide. Mais comme l'autre stratégie, elle nécessite une gestion des pairs pupilles. Cette gestion ne surcharge pas trop le tuteur. Nous avons également opté pour cette solution, vu que nous restons sur le schéma de LH* et LH*$_{RS}$. Y a aussi les contraintes du prototype actuel qui nous laissent sur cette stratégie pour exploiter les avantages du prototype LH*$_{RS}$.

5.5. Rechercher sûre

Pour illustre la recherche sûre, nous supposons qu'il n'y a pas de renvoi de requête et que le pair *k* a le bon niveau de fichier SDDS. Nous négligeons aussi les accusés de réception de chaque nœud à une requête. Nous considérons que nous avons un groupe de parité constitué de *m* cases de données distribué sur *m* pairs et de *k* cases de parité sur *k* pairs tel que $k < m$.

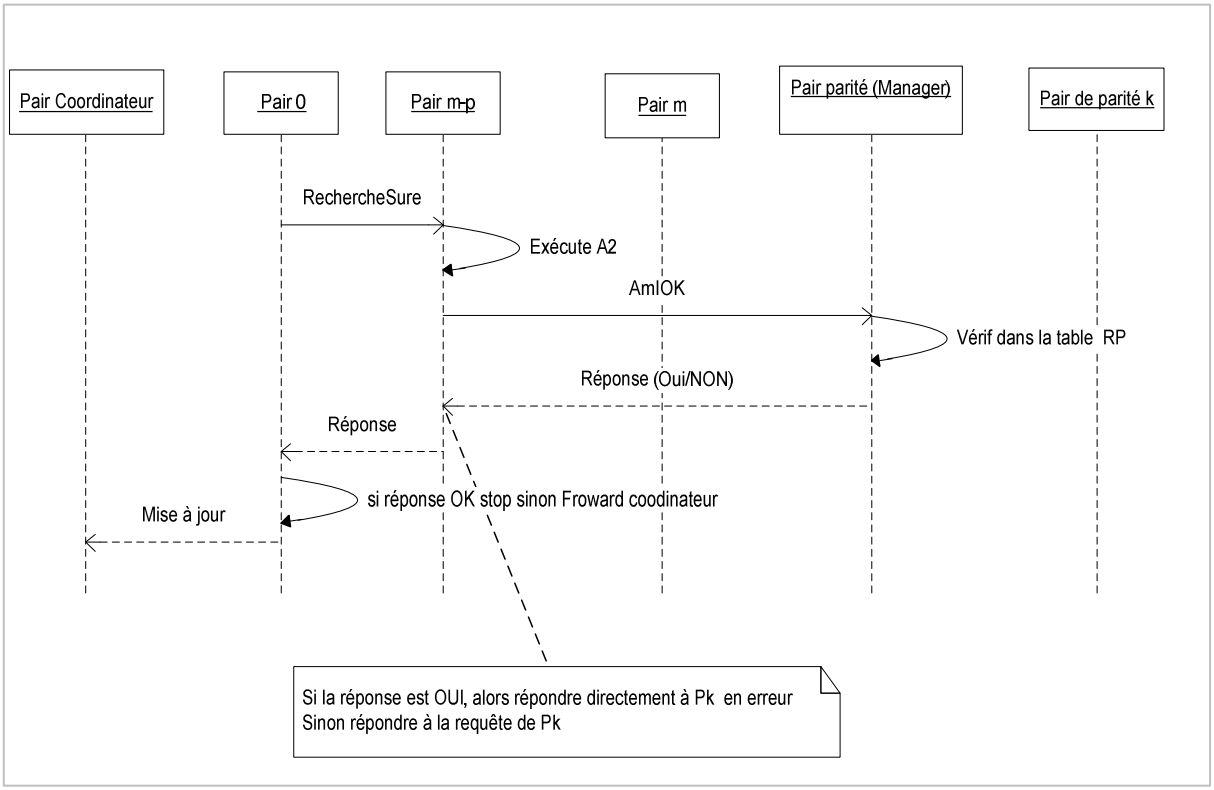


Figure 49. Recherche Sûre

Comme le montre la Figure 49, un pair P_0 envoi sa requête de recherche sûre au pair P_{m-p} tel que $p < m$. Ce dernier analyse la requête et exécute l'algorithme A5. Si y a renvoi alors il renvoie la requête et c'est au récepteur final de poursuivre l'exécution de la recherche sûre. Le pair P_{m-p} envoie un message au manager du groupe de parité pour savoir s'il a été reconstruit ou pas. Le manager vérifie si le pair P_{m-p} est dans sa table de *RecoveredPeers*. Si oui alors le P_{m-p} est reconstruit et considéré comme étant pas à jour, le manager envoie l'adresse du

nouveau pair où la case est reconstruite. Le pair $P_{m\text{-}p}$ informe le pair P_0 qu'il n'est pas à jour dans ce cas il P_0 met à jours ces adresses de Pairs en contactant le pair coordinateur. Si la réponse du manager est non, alors le $P_{m\text{-}p}$ est considéré comme à jour, alors il exécute la requête de la recherche sûre et répond à P_0 et le processus s'achève ainsi.

6. Résumé

Dans ce chapitre nous avons présenté l'architecture fonctionnelle des pairs du prototype LH $*_{RS}^{P2P}$ et l'architecture interne d'une case de données versus de parité. La mise en place d'un contrôleur de flux pénalise nos mesures et consomme de la ressource. Pour des besoins de mesures de performances nous avons modifié le protocole d'envoi de requête spécifiquement pour la recherche sûre. Le contrôleur de flux est sûr mais pas nécessaire vue la définition même d'un système pair à pair.

Le Churn pèse encore sur cette architecture pair à pair car la détection de panne doit passer en premier et le plus rapidement possible. Le fait de faire des renvois et de mettre les requêtes dans une file d'attente ralentie la détection du Churn.

Nous avons pu mettre en cohésion la structure de nos messages avec les différents messages et méthodes du prototype LH*$_{RS}$. Nous avons aussi donné les scénarios d'exécutions de chaque opération offerte par notre prototype. Nous n'avons présenté que les nouveaux scénarios car les autres sont détaillés dans [M04].

Chapitre V

Mesures de Performances de LH $*_{RS}^{P2P}$

1. Introduction

Après avoir présenté l'architecture fonctionnelle implémentée dans notre prototype, nous présentons les mesures de performances de ce dernier. Ces mesures sont faites pour valider le fonctionnement de notre prototype et donner la preuve de la faisabilité des opérations de base pour la manipulation du fichier LH $*_{RS}^{P2P}$.

Nous avons repris quelques mesures de performances des opérations de bases, telles que la création du fichier et la recherche simple du prototype de base LH*$_{RS.}$ En effet, les modules LH $*_{RS}^{P2P}$ que nous avons implémentés représentent une couche indépendante qui ne dégrade pas les performances du prototype LH*$_{RS}$ de base. Ces coûts peuvent correspondre au temps d'accès aux enregistrements et aux messages envoyés entre différents nœuds du réseau.

Étant sur le même environnement d'expérimentation que celui de LH*$_{RS}$, les mesures de performances des opérations de base n'ont pas changé. Alors il n'est pas nécessaire de refaire des mesures de performances déjà existantes. Nous les discuterons dans le cadre de notre travail. Nous donnerons aussi les mesures de nouvelles opérations pour le prototype LH $*_{RS}^{P2P}$, la recherche sûre en flux et la recherche simple.

Nous commencerons par la présentation de configuration matérielle de notre réseau local utilisée puis la manière de lancer le prototype. Par la suite, nous présenterons les mesures de performances des opérations de base du prototype LH $*_{RS}^{P2P}$.

2. Configuration de l'Environnement Expérimental

Nos expérimentations ont été menées sur l'environnement matériel et logiciel de base suivant :

- Machines : de type PC, en nombre de 4 dont un serveur. Processeur Pentium IV 1.8 Ghz. RAM 512 Mo sauf une a 1Go.
- Système d'exploitation : trois machines sont sur Windows 2000 Server. Le serveur est sous Windows 2003 Serveur avec 1Go de RAM.
- Réseau cadencé à 1Gbps

Notons que pour nos expérimentations, nous avons installé notre logiciel uniquement que sur les machines sous Windows 2000 Server. Le serveur Windows 2003 Server est utilisé pour le paramétrage du réseau local sur lequel est installé le DHCP (Dynamic Host Configuration Protocol). Le serveur attribue des adresses IP pour les trois autres machines qui se connectent au réseau.

La Table 4 ci-après détermine la terminologie utilisée dans ce qui suit :

Paramètre	Description	Valeur
b	Capacité d'une case de données, exprimée en nombre d'enregistrements.	10 000, 50 000
b'	Capacité d'une case LH, exprimée en nombre d'enregistrements. La structure de la case $LH*_{LH}$ est présentée dans le Chapitre IV Section 2.1	100
N	Nombre d'enregistrements de données dans le fichier	2.5**b*
m	Taille d'un groupe de parité	4
k	Niveau de disponibilité d'un groupe g	0, 1, 2, 3
f	Nombre de cases de données en échec	0, 1, 2

Table 4. Terminologie

3. Lancement du prototype

L'installation des instances du prototype LH $*_{RS}^{P2P}$ a été faite sur chacune des trois machines connectées au réseau local. Le prototype se compose de 4 instances dont le coordinateur, le client qui porte l'application, la case de données et la case de parité. Nous avons lancé une instance du coordinateur et du client sur la même machine. Sur les deux autres machines nous avons lancé les instances de cases de données et de parité. Les instances de case de données

peuvent être lancées selon les besoins lors des éclatements. Rappelons que les trois machines sont sous Windows 2000 Server.

Au lancement du coordinateur LH $*_{RS}^{P2P}$, un fichier mappé est créé en mémoire avec une seule case de donnée vide. La première case de données est alors créée sur le pair coordinateur. Il faut noter que nous ne pouvons pas lancer deux instances de coordinateur sur la même machine car Windows n'accepte pas d'avoir deux fichiers mappés avec le même descripteur en mémoire. Cette contrainte est héritée du prototype LH*$_{RS}$ de base.

Comme nous l'avons présenté dans le Chapitre III l'architecture théorique prévoit la gestion de la première case de données par un pair autre que le coordinateur. La gestion de la case de données sur le coordinateur est faite indépendamment de la gestion de coordination. Le risque de surcharge du pair coordinateur est donc minime. La création de fichier sous notre prototype est identique à celle faite sous LH*$_{RS}$. La *k*-disponibilité est définit au lancement du prototype, ce qui permet d'enregistrer les serveurs de parité définissant cette *k*-disponibilités.

4. Création du Fichier par un Flux d'Insertions

Dans ce qui suit, nous rapportons les temps de création d'un fichier SDDS, k-disponible ($k \in \{0, 1, 2\}$) réalisé sur le prototype LH*$_{RS}$. Les temps sont calculés sur le client. L'expérimentation se déroule comme suit : un client procède à l'insertion de 25000 enregistrements (b = 10000, b' = 100), de 100 octets chacun. Les clés sont générées automatiquement par une fonction d'incrémentation. Cette fonction a comme paramètre une valeur saisie depuis l'application du client. Le fichier se crée sur quatre cases de données, avec un premier éclatement de la case de données 0 qui crée la case de données 1. S'ensuivent deux éclatements successifs de la case de données 0, qui crée la case de données 2. L'éclatement de la case de données 1 crée la case de données 3.

Notons que le client envoie une seule requête à la fois et attend l'accusé de réception correspondant. Lors de l'éclatement d'une case de données, celle-ci est verrouillée jusqu'à la fin du processus. Durant le temps d'éclatement de la case l'envoi des requêtes d'insertion est arrêté. Chaque case de données accuse réception des requêtes reçues. Cependant aucun mécanisme d'acquittement n'est implémenté entre les cases de données et les cases de parité, un mécanisme hérité du prototype LH*$_{RS}$. La dégradation des performances observée est due seulement à la propagation des mises à jour vers les cases de parité.

Ainsi, les requêtes de mise à jour envoyées aux cases de parité ne sont pas suivies d'acquittement. Dans ce qui suit, nous pouvons constater l'impact de l'attente des accusés de réception qui s'ajoute à celui des éclatements de cases de données. Cet impact se traduit par le ralentissement de la vitesse d'envoi des requêtes par les pairs.

La Figure 50 montre la création du fichier SDDS par insertion, par flux de 25000 clés. Seules la stratégie d'acquittement et de contrôle de flux entre [Pair client – Pair de données versus client-serveur de données], ont été implémentées et évaluées. La stratégie d'acquittement des messages de mises à jour par les cases de parité, n'est pas prise en compte. La Figure 50 montre les performances de création d'un fichier SDDS *k*-disponible, tel que k = 0,1, 2. Concernant l'envoi des requêtes d'insertion par le pair, le paramètre d'acquittement

de messages est initialisé à 5, ce qui correspond à un acquittement pour 5 messages envoyés. Notons que le schéma LH*$_{RS}$ 1-disponible (k = 1) dégrade les performances de création d'un fichier de 60% par rapport au schéma LH*$_{RS}$ et LH*$_{LH}$ [M04]. Le coût de mise à jour d'une case de parité en plus de la première case de parité est en moyenne de 11%.

Afin de calculer la perte de messages de mises à jour entre case de données et case de parité des compteurs de messages ont été mis en place des deux côtés des cases pour compter le nombre de messages envoyé par chaque case de données et réceptionnés par la case de parité. Le rapport des sommes des compteurs désigne le taux de perte moyen sur trois expérimentations est de 0,73% pour k = 1, et de 1,1% pour k = 2.

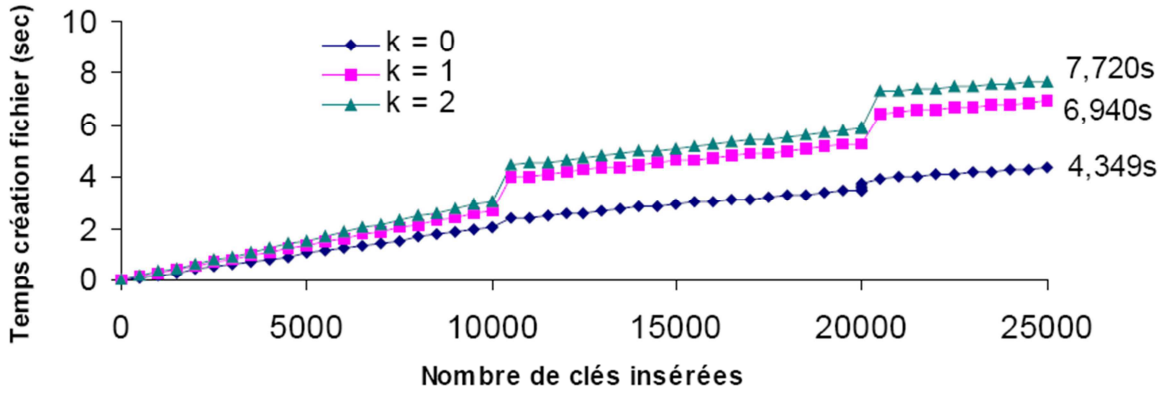


Figure 50. Création de fichier k-disponible (k=0, k=1, k=2), [M04]

5. Requête de Recherche

Le prototype de base LH*$_{RS}$ distingue deux modes de recherche par clé : recherche en flux et recherche synchrone. Ces deux recherches ont été reprises également sur notre prototype.

Pour les requêtes de recherche en flux, le pair client envoie une nouvelle requête sans attendre la réponse à la précédente. Pour éviter les pertes de messages, le flux peut être contrôlé par fenêtrage [M04], comme pour le TCP/IP. Les recherches sur les différents pairs peuvent être parallèles. La recherche en flux offre le meilleur temps de réponse possible par requête.

Pour une recherche synchrone, le pair envoie une nouvelle requête seulement après réception de la réponse à la précédente requête.

Les expériences mesurent les temps moyens de requêtes de recherche en flux et recherche synchrone. Pour les requêtes en flux, le débit d'envoi, le temps de réception des réponses ainsi que le taux de pertes sont mesurés sur le client. Le fichier contient 125 000 enregistrements et s'étend sur quatre cases de données. Les clés recherchées sont dans l'intervalle [1, x], *x* variant de 0 à 100 000. Ces clés sont générées séquentiellement. La taille d'une case LH* est de *b* = 50 000, celle d'une page LH interne à la case LH* est de *b'* = 100. Le temps de

réponse d'une requête synchrone comprend celui d'envoi sur le client plus le temps de traitement sur le serveur et de l'envoi de la réponse, soit approximativement le RTT dans le réseau (ang. Round Trip Time). Le composant 'client' d'un pair est doté, pour les réponses, d'un seul Thread de travail et également d'une file unique. Le composant 'serveur' de données est doté de quatre Threads de travail pour les requêtes reçues. Les temps sont mesurés entre le premier envoi et la dernière réponse.

Le Table 5 et la Figure 51 montrent ces résultats. Pour les requêtes en flux, les quatre pairs de données ont un débit de 17857 req/sec. Ainsi, un pair client envoie la clé *x* à un seul pair de données, telle que, x∈ {5000, 10000, 20000, 30000}, comme le montre la Table 5 ci-après.

Requête de recherche *x* clés	Débit d'envoi sur le pair (client)	Débit de réception sur le pair (client)	Taux de perte
5000	40000	10309	0,0036
10000	42553	10660	0,0045
20000	41322	10157	0,0078
30000	41724	9696	0,0127

Table 5. Débit pour les requêtes en flux (req/sec)

Nous remarquons que les débits sont presque constants, de l'ordre de 40K pour l'envoi et de 10K pour la réception. Les légères variations sont manifestement dues aux fluctuations pratiques du débit du réseau et du temps de traitement, et ce d'une expérience à l'autre. Par ailleurs, nous remarquons que le taux de pertes de messages (requêtes ou réponses) est négligeable, sous 1% en général. Ces résultats démontrent à l'évidence une efficacité espérée du prototype de base et donc de notre prototype, au niveau du traitement de messages. Néanmoins, le taux de perte augmente légèrement avec le nombre de requêtes envoyées.

Nous constatons sur la Figure 51 les temps moyen de recherches d'un enregistrement. Ils sont constants en montrant donc une très bonne scalabilité du fichier. Les valeurs sont de 0,24ms par requête synchrone, et de 0,056ms pour une requête en flux.

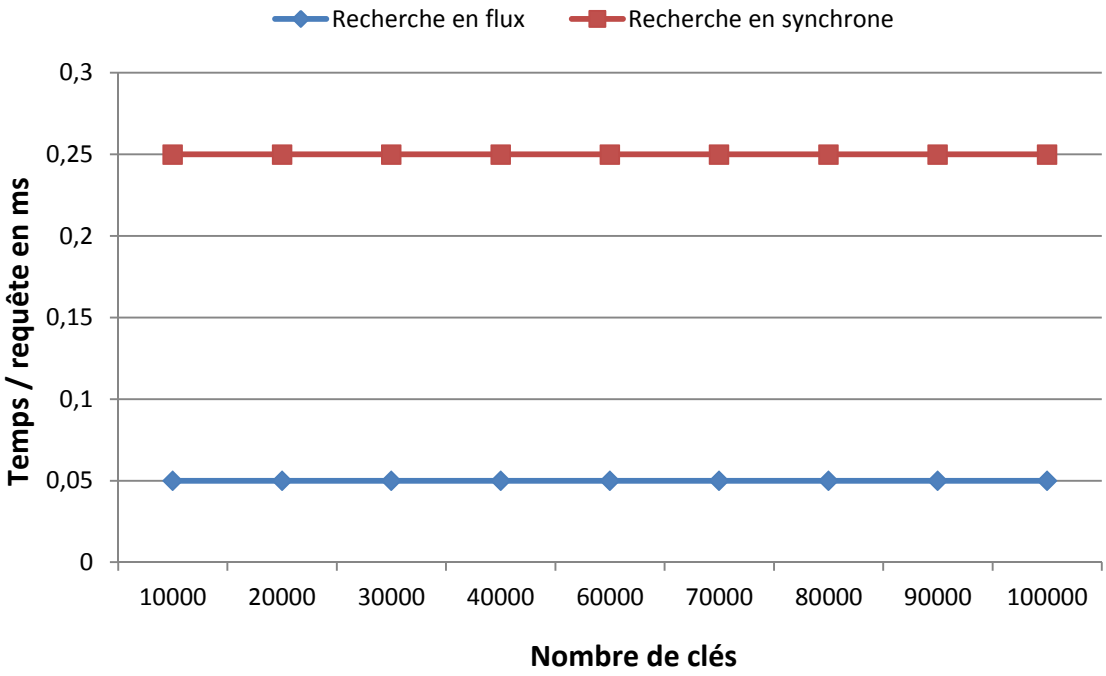


Figure 51. Recherche à clé

6. Requête de Recherche Sûre

Nous présentons maintenant les mesures de la recherche sûre. Le but de ces mesures est de valider l'intérêt et la faisabilité de la requête sûre pour un fichier LH $*_{RS}^{P2P}$ en se limitant à l'implémentation de la recherche sûre. Le résultat attendu de la recherche sûre est d'environ deux fois plus long que celui de la recherche simple.

Comme pour la recherche simple, nous avons expérimenté deux modes de recherches : la recherche en flux et la recherche synchrone. Dans les deux cas les requêtes recherchaient séquentiellement les clés dans l'intervalle [1, *x*].

Nous avons diminué la taille de la case de données $b = 100$, la taille du groupe de parité est de $k = 2$. Puis, nous inserons 1000 clés. Après insertion, nous avons lancé la recherche sûre et la simple recherche avec les deux types décrits ci-haut.

La Table 6 ainsi que la Figure 52 et la Figure 53, nous montrent les temps de recherche par clé, mesuré sur le pair 'client'. Le temps moyen d'une recherche simple en flux est de 0,078ms. Celui de la recherche sûre en flux est de 0,141ms. Soit presque le double du temps de la recherche simple. Ce temps s'explique par l'attente supplémentaire sur le pair serveur de la réponse à la demande envoyé au gestionnaire de parité du groupe. Cette demande vérifie si la case du pair serveur n'a pas été reconstruite ailleurs. Ce temps est en général le RTT d'un échange entre le pair serveur et le gestionnaire du groupe de parité. Il est de deux messages. Le doublement du RTT d'une requête sûre par rapport à celui d'une simple confirme notre théorie du Chapitre III.

Comme le montre la Table 6 et la Figure 52, montrent la différence en temps entre la recherche en flux et la recherche synchrone. Comme on peut le constater, il est le double. Il s'agit de 0,469ms pour la recherche sûre et 0,235ms pour la recherche simple. Les temps de la recherche simple sont presque ceux de LH*$_{RS}$ présentés dans la section précédente, comme ils devraient l'être.

	Temps Recherche sûre (millisecondes)		Temps Recherche simple (millisecondes)	
Nombre de clés	Flux	Synchrone	Flux	Synchrone
1000	0,141	0,469	0,078	0,235

Table 6. Temps de réponse pour la recherche sûre et simple

Nous constatons aussi une différence de taille entre les temps mesurés pour la recherche de type flux et synchrone. Comme le montre la Table 6, cette différence et d'environ 300%, le temps est de 0,141ms et 0,469ms pour la recherche sûre en flux et synchrone, respectivement. Idem pour la recherche simple en flux et synchrone. La recherche en flux est plus rapide que la recherche synchrone pour les deux types de rechercher car elle ne comporte pas d'attente d'accusé de réception.

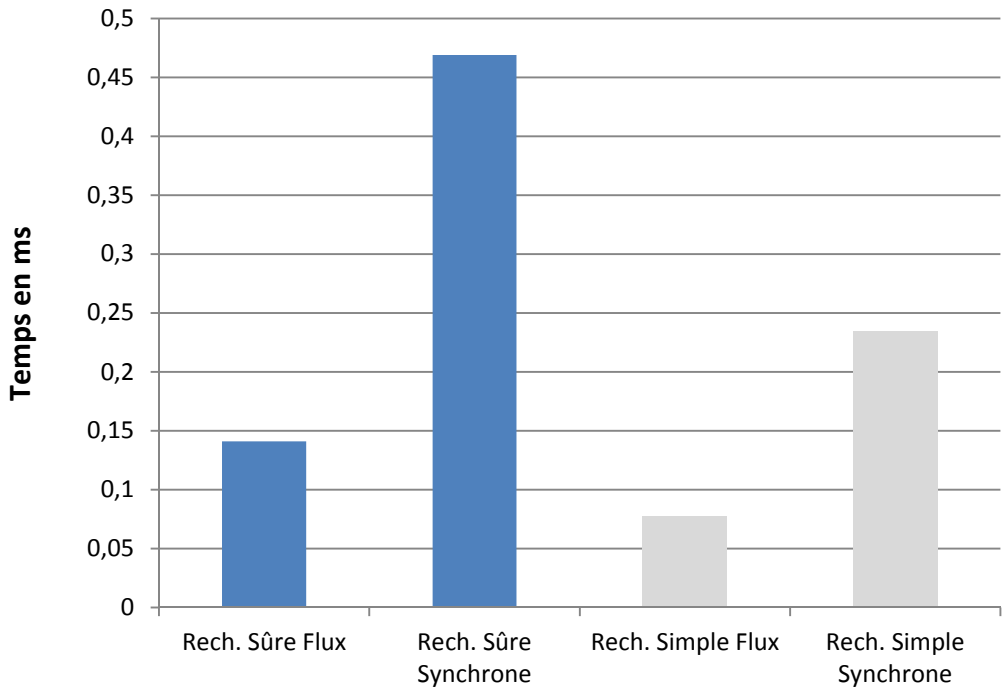


Figure 52. Temps total de réponse pour la recherche sûre et simple

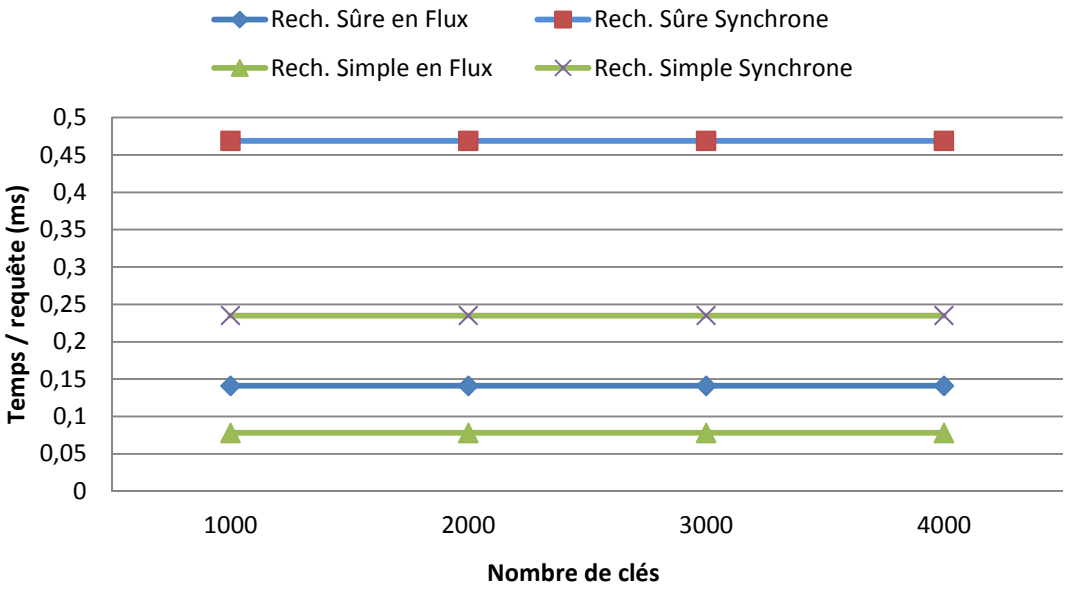


Figure 53. Temps de recherche à clé

7. Résumé

Nous avons présenté les principales mesures expérimentales de performances de notre prototype. Il s'agissait du temps moyens de création d'un fichier, d'une insertion, d'une recherche simple et d'un recherche sûre. Les temps ont été mesurés pour les deux modes habituels de fonctionnement, en flux et synchrone. Nos mesures concernaient les parties du logiciel de notre prototype qui ont été modifiées par rapport au logiciel repris du prototype LH*$_{RS}$. Nous n'avons pas mesuré les temps de récupération, car le logiciel correspondant est resté le même.

Les résultats obtenus confirment la validité (Anglais : Correctness) de notre conception générale de LH $*_{RS}^{P2P}$ et de notre implémentation. Ils confirment également l'analyse théorique de ces performances. Comme attendu, les temps des mêmes requêtes sont très proches de ceux mesurés sur le prototype LH*$_{RS}$. Ce qui prouve que nous avons su adapter ce prototype au nouveau mode de fonctionnement caractérisant LH $*_{RS}^{P2P}$, sans surcharge. Enfin, la conformité des mesures de la recherche sûre aux prévisions théoriques prouve aussi la validité de notre implémentation de cette nouvelle fonction.

A l'avenir nos mesures devraient être approfondies. L'expérimentation devrait se porter sur des gros fichiers qui s'étendent sur davantage de nœuds, expérimentation qui était impossible dans nos conditions actuelles.

Chapitre VI

Conclusion et Travaux Futurs

1. Conclusion

Nous avons proposé une nouvelle structure de données distribuées et scalable (SDDS), dite LH $*_{RS}^{P2P}$. Elle adapte aux réseaux pair à pair (P2P) les principes de la variante du hachage linéaire distribué, dite LH*$_{RS}$. Notre SDDS permet de construire de très grands fichiers distribués, pouvant s'étendre sur un nombre de pairs quelconque, ce nombre étant illimité théoriquement. Ces fichiers présentent des performances d'accès par clé et de scans les plus efficaces actuellement connues et résistent au Churn. Leurs performances d'accès sont dues à leurs caractéristiques principales, à savoir un renvoi au maximum d'une requête à clé entre les pairs serveurs de données du ficher dans le cas d'une erreur d'adressage. La seule structure connue caractérisée par le même nombre de renvois est celle du 'One hop Lookups' basée sur l'anneau Chord. Elle nécessite néanmoins une réorganisation de l'anneau Chord et une maintenance de chaque table de routage de chaque nœud, ce qui est très couteux en messages. Notre structure est dépourvue d'un tel inconvénient.

Quant à LH*, elle offre la performance de deux renvois au plus. D'autres organisations P2P connues comme Chord ou BATON, nécessitent O(log *N*) renvois.

LH $*_{RS}^{P2P}$ offre aussi un mécanisme de récupération de données offrant une protection contre le Churn. À cette fin, Nous avons adapté le mécanisme de haute disponibilité de LH*$_{RS}$. Nous protégeons le fichier contre l'indisponibilité de *k* pairs serveurs quelconques. La valeur de *k* est un paramètre d'utilisateur, défini à la création du fichier et scalable en fonction de l'étendu du fichier. Par ailleurs, nous avons introduit le concept, à notre connaissance original, de requêtes sûres. Celles-ci protègent chaque pair contre l'utilisation de données périmées à laquelle le Churn pourrait conduire.

Nous avons commencé nos travaux en définissant l'algorithmique de LH $*_{RS}^{P2P}$. Pour le valider, nous l'avons prototypé. Pour ce faire, nous avons procédé à une série de mesures de performances, prouvant son efficacité attendue.

2. Perspective

Notre contribution conduit à plusieurs développements dont les plus intéressants sont les suivants.

2.1. Vulnérabilité du Coordinateur

Dans notre étude nous avons supposé que le coordinateur est unique. Cette hypothèse était suffisante pour nos buts. L'indisponibilité du coordinateur compromettrait néanmoins l'utilisation du fichier. Les variantes ci-dessus allègent cet inconvénient.

2.1.1. Réplication du Coordinateur

Cette variante est la plus évidente. Le coordinateur pourrait être répliqué sur *k* nœuds, étant aussi *k* - disponible. Cette solution demande de résoudre certains problèmes quant à la consistance de données entre les répliques, comme pour toute réplication [WK02].

2.1.2. Schéma LH $*_{\mathbf{RS}}^{\mathbf{P2P}}$ sans coordinateur

Une telle variante pourrait se baser sur celle définie pour LH* dans [LN96]. Cette dernière utilise un jeton et des éclatements en cascade. La nouvelle variante nécessiterait une refonte correspondant à une partie de notre algorithmique, notamment, celle concernant la *k*-disponibilités.

2.2. Requêtes Sûres

Pour valider notre concept de requête sûre nous nous sommes limités à l'implémentation d'un seul type de telles requêtes qui est la recherche sûre. Nous devrions compléter l'étude d'autres types de requêtes sûres, les scans compris. Par ailleurs, nous avons choisi d'implémenter la table *RecoveredPeers* (RP) présentée dans le Chapitre III, Section 6 et dans le Chapitre IV en Section 2.3. RP était gérée en dehors du calcul de parité et sa perte n'était pas récupérée. Ce choix technique étant facile à implémenter, il nous a permis de valider notre algorithme et de faire les mesures correspondantes. C'était suffisant pour la preuve du concept, notre but dans cette Thèse.

Cependant, les futurs efforts doivent être concentrés sur une implémentation du schéma de la requête sûre permettant de garantir la haute disponibilité des métadonnées correspondantes. C'est le schéma dessiné également dans le Chapitre III en Section 6. Pour rappel, la table définie qui y est également un enregistrement de données. Elle est récupérée si besoin comme tout autre enregistrement de données. Les métadonnées qu'elle contient deviennent donc *k*-disponible.

2.3. Haute Disponibilité

L'étude du schéma de distribution de parité (Anglais : declustering) présenté dans chapitre III en Section 9 doit être poursuivie. Nous rappelons que dans ce schéma, chaque pair contient à la fois ses enregistrements de données et les enregistrements de parités d'une autre case. Le principe de base de ce schéma est que les données de parités d'un groupe pair sont sauvegardées dans le groupe impair et vice versa. Nous pouvons supposer la construction de

fichiers *k*-disponibles avec un meilleur équilibre des charges par rapport à notre schéma de base LH $*_{RS}^{P2P}$. Cette conjecture reste néanmoins à approfondir.

2.4. Implémentation Réelle

Notre travail se situe à la croisée des domaines de l'organisation de gros volumes de données, des architectures P2P et de celles de Clouds. Nous pensons avoir démontré que LH $*_{RS}^{P2P}$ constitue une solution désormais prête à son implémentation réelle. Celle-ci pourrait être particulièrement utile pour un RamClouds. Il nous semble aussi que le VMWare Gemstone RamCloud, [VMW], destiné aux fichiers distribués sur des Clouds privés, pourrait notamment en bénéficier. Néanmoins, nous sommes conscients du chemin restant à parcourir entre notre implémentation 'la preuve du concept' et l'implémentation réelle.

Bibliographie

[A01] **Aboba B.,** Pros and Cons of Upper Layer Network Access, March 2001

[AMR+11] **Abiteboul S., Manolescu I., Rigaux P., Rousset M., and Senellart P., 2011.** Web Data Management. Cambridge University Press, New York, NY, USA

[AWD01] **Ailamaki A., David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor:** Where Does Time Go?. In Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99), Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 266-277

[B02] **Bram C. (2001-07-02).** "BitTorrent — a new P2P app". Yahoo eGroups. Retrieved 2007-04-15.

[B71] **Bayer R. 1971.** Binary B-trees for virtual memory. In Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control (SIGFIDET '71). ACM, New York, NY, USA, 219-235. DOI=10.1145/1734714.1734731 http://doi.acm.org/10.1145/1734714.1734731

[B83] **Burkhard W.A. -** Interpolation-based Index Maintenance. Proc. of the 2nd Symp. On Principles of databases Systems. TODS 1983. pp.76-88.

[BBC+04] **Bavier A., Bowman M., Chun B., Culler D., Karlin S., Muir S., Peterson L., Roscoe T., Spalink T., and Awrzoniak M., 2004.** Operating system support for planetary-scale network services. In Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1 (NSDI'04), Vol. 1. USENIX Association, Berkeley, CA, USA, 19-19.

[BDH07] **Bolosky W. J., Douceur J. R., and Howell J., 2007.** The Farsite project: a retrospective. SIGOPS Oper. Syst. Rev. 41, 2 (April 2007), 17-26.

[BSV03] **Bhagwan R., Savage S., and Voelker G.** Understanding availability. In Proc. IPTPS, Feb. 2003

[BZ02] **Boukhlef D., Zegour D.E.** IH* : Hachage Linéaire Multidimentionnel Distribuée et Scalable (article soumis pour le CARRI2002) 04/01/2002

[CDG+08] **Chang F., Dean J., Ghemawat S., Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008**. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages

[CLL02] **Chu J., Labonte K., and Levine B. N.** Availability and locality measurements of peer-to-peer file systems. In Proc. of ITCom: Scalability and Traffic Control in IP Networks, July 2002.

[CPZ97] **Ciaccia P., Patella M., and Zezula P.. 1997.** M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97), Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan

Kaufmann Publishers Inc., San Francisco, CA, USA, 426-435.

[D01] **Diène A.W. Nov. 2001.**Contribution à la Gestion de Structures de Données Distribuées et Scalables, Thèse de doctorat Université Paris Dauphine. http://ceria.dauphine.fr/aly/aly.html.

[D93] **Devine R.** - Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. 4th Intel. Conf. on Foundations of Data Organizations and Algorithms(FODO-93), Chicago (Oct. 1993). Lecture Notes in Computer Science, Springer Verlag (publication). 1993.

[DH06] **Douceur J.R. and Howell J. 2006.** Distributed directory service in the Farsite file system. In Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06). USENIX Association, Berkeley, CA, USA, 321-334.

[DLR09] **Du Mouza C., Litwin W., and Rigaux P., 2009.** Large-scale indexing of spatial data in distributed repositories: the SD-Rtree. The VLDB Journal 18, 4 (August 2009), 933-958. DOI=10.1007/s00778-009-0135-4 http://dx.doi.org/10.1007/s00778-009-0135-4

[G78] **Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle (Eds.). 1978.** Operating Systems, an Advanced Course. Springer-Verlag, London, UK.

[G93] **Gray J., Super-Servers:** Commodity Computer Clusters Pose a Software Challenge. http://131.107.1.182:80/research/barc/gray/default.htm.

[G97] **Gilder G., Fiber keeps its Promise:** Get Ready Bandwidth will triple each year for the next 25. Forbes, 7 April 1997

[G97] **Ghernaouti-Hélie, S.** Client/Serveur les outils du Traitement Réparti Coopératif, Dunod (1 décembre 1997) ISBN-10: 2225845301

[G99] **Gray J.,** Turing Award Lecture: What Next? ACMComputer Conference, Atlanta, Georgia, 4 May 1999.

[GDS+03] **Gummadi K. P., Dunn R. J., Saroiu S., Gribble S. D., Levy H. M., and Zahorjan J.,** Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In Proc. ACM SOSP, Oct. 2003.

[GG96] **Gardarin, G. and Gardarin, O. L**e Client-Serveur, Editions Eyrolles 1996.

[GGL03] **Ghemawat S., Gobioff H., and Leung S.T., 2003.** The Google file system. SIGOPS Oper. Syst. Rev. 37, 5 (October 2003), 29-43. DOI=10.1145/1165389.945450 http://doi.acm.org/10.1145/1165389.945450

[GIA11] **Gilroy M, Irvine J., and Atkinson R., 2011.** RAID 6 Hardware Acceleration. ACM Trans. Embed. Comput. Syst. 10, 4, Article 43 (November 2011), 17 pages. DOI=10.1145/2043662.2043667 http://doi.acm.org/10.1145/2043662.2043667

[GLR03] **Gupta A., Liskov B., and Rodrigues R., 2003.** One hop lookups for peer-to-peer overlays. In Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03), Vol. 9. USENIX Association, Berkeley, CA, USA, 2-2.

[GLV84] **Garcia-Molina H., Lipton R. J., and Valdes J. 1984.** A Massive Memory Machine. IEEE Trans. Comput. 33, 5 (May 1984), 391-399. DOI=10.1109/TC.1984.1676454 http://dx.doi.org/10.1109/TC.1984.1676454

[GSS06] **Godfrey P. B., Shenker S., and Stoica I. 2006.** Minimizing churn in distributed systems. SIGCOMM Comput. Commun. Rev. 36, 4 (August 2006), 147-

158.DOI=10.1145/1151659.1159931 http://doi.acm.org/10.1145/1151659.1159931

[HBC97] **Hilford V, Bastani & Cukic B.** – EH* Extendible Hashing in a distributed Environment

[HLM+10] **Hill Z., Li J., Mao M., Ruiz-Alvarez A., and Humphrey M. 2010.** Early observations on the performance of Windows Azure. In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10). ACM, New York, NY, USA, 367-376. DOI=10.1145/1851476.1851532 http://doi.acm.org/10.1145/1851476.1851532

[ISI81] **Information Sciences Institute, RFC 793**: Transmission Control Protocol (TCP) – Spécification, Sept. 1981, http://www.faqs.org/rfcs/rfc793.html, traduit en Français http://www.abcdrfc.free.fr/rfc-vo/rfc093.txt

[J88] **Jacobson V. 1988. Congestion avoidance and control.** In Symposium proceedings on Communications architectures and protocols (SIGCOMM '88), Vinton Cerf (Ed.). ACM, New York, NY, USA, 314-329. DOI=10.1145/52324.52356 http://doi.acm.org/10.1145/52324.52356

[JOV05] **Jagadish H.V, Ooi B.C, Vu Q.H. 2005**. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. In proceedings of the 31st VLDB conference, 2005.

[JOV06] **Jagadish H.V, Ooi B.C, Vu Q.H**. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes.: 22nd IEEE International Conference on Data Engineering (ICDE), 2006 (to appear).

[KAZ03] **KaZaA.** Official web site, 2003, http://www.kazaa.com/. Standard FastTrack client.

[KLR96] **Karlsson, J. Litwin, W., Risch, T**. LH*lh: A Scalable High Performance Data Structure for Switched Multicomputers. Int. Conf. on Extending Database Technology, EDBT-96, Avignon, March 1996

[KM02] **Klinberg T. and Manfredi R. Gnutella Protocol Specification v0.6. http://rfc-gnutella.sourceforge.net/src/rfc-0 6-draft.html, June 2002.**

[KW94] **Kröll B. & Widmayer P. -** Distributing a Search Tree Among a Growing Number of Processors. ACM-SIGMOD Intel. Conf. On Management of Data, Minneapolis. p. 265-276, May, 1994

[L00] **Ljungstrom, M**.: Implementing LH*$_{RS}$: a Scalable Distributed Highly-Available Data Structure, Master Thesis, Feb. 2000, CS Dep. U. Linkoping, Sweden.

[L80] **Litwin, W. Linear Hashing:** A new tool for file and table addressing, In Proc. Of VLDB,Montreal, Canada, 1980. Reprinted in Readings in Database Systems, M. Stonebreaker ed., 2nd édition, Morgan Kaufmann, 1995.

[L97] **Lindberg., R. A** Java Implementation of a Highly Available Scalable and Distributed Data Structure LH*g. Master Th. LiTH-IDA-Ex-97/65. U. Linkoping, 1997, 62.

[LMR98] **Litwin, W., Menon J., Risch**, **T**. LH* with Scalable Availability. IBM Almaden Res. Rep. RJ 10121 (91937), (May 1998), (subm.).

[LMS03] **Litwin, W. Mokadem R s Schwarz.** Disk Backup Through Algebraic Signatures in Scalable and Distributed Data Structures", Proceedings of the Fifth Workshop on Distributed Data and Structures, Thessaloniki, June 2003 (WDAS 2003). http://ceria.dauphine.fr/Riad/Article_storage-wdas_wl.pdf

[LMS04] **Litwin, W., Moussa R., Schwarz T., 2004.** LH*RS: A Highly Available Distributed Data

Storage Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

[LMS05] **Litwin W., Moussa R., Schwarz T. 2005.** LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure. ACM-TODS, Sept. 2005.

[LMS06] **Litwin W., Mokadem R., Sahri R. 2006.** Virtual Repository for e-Gov life event Document. 2006.

[LN95] **Litwin, W., Neimat, M-A. 1995**. k-RP*S : A Scalable Distributed Data Structure for High-Performance Multi-Attribute Access. Res. Rep. GERM Paris 9& Distributed Inf. Techn. Dep. HPL Palo Alto, April 1995

[LN96] **Litwin, W. Neimat,** High-Performance Multi-Attribute Access. Res. Rep. GERM Paris 9& Distributed Inf. Techn. Dep. HPL Palo Alto, April 1995.

[LNS93a] **Litwin, W. Neimat, M-A., Schneider, D.** LH*: Linear Hashing for Distributed Files. ACM-SIGMOD Int. Conf. On Management of Data, 93.

[LNS93b] **Litwin, W., Neimat, M-A., Schneider, D.** LH*: A Scalable Distributed Data Structure. (Nov. 1993).

[LNS94] **Litwin, W., Neimat, M. et Schneider, D.** RP* : A family of order preserving scalable distributed data structures. VLDB, 1994.

[LNS96] **Litwin, W., Neimat, M-A., Schneider, D. LH*:** A Scalable Distributed Data Structure. ACM-TODS, (Dec., 1996).

[LRR97] **Litwin W. and Risch T. - LH*g:** a high-availability Scalable Distributed Data Structure through record grouping. U-Paris 9 Technical Report, May, 1997.

[LS00] **Litwin, W., J.E. Schwarz, T. LH*RS:** A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. ACM-SIGMOD-2000 Intl. Conf. On Management of Data.

[LS02] **Litwin, W., Sahri, S. 2004**. Implementing SD-SQL Server: a Scalable Distributed Database System. Intl. Workshop on Distributed Data and Structures, WDAS 2004, Lausanne, Carleton Scientific (publ.).

[LS99] **Litwin, W., Schwarz T. 1999.** LH*$_{RS}$: A High-Availability Scalable Distributed Data Structure using Reed Solomon Codes. CERIA Res. Rep. 99-2, U. Paris 9, 1999.

[LYS08] **Litwin W., Yakouben H., and Schwarz T. 2008.** LH*$_{RS}^{P2P}$: a scalable distributed data structure for P2P environment. In Proceedings of the 8th international conference on New technologies in distributed systems (NOTERE '08). ACM, New York, NY, USA, Article 1, 6 pages. DOI=10.1145/1416729.1416731 http://doi.acm.org/10.1145/1416729.1416731

[M03] **Moore G. E.,** No Exponential is Forever … but We Can Delay 'Forever', International Solid State Circuits Conference, 2003

[M04] **Moussa R.** Contribution à la Concetion et l'Implantation de la Structure de Données Distribuée & Scalable à Haute Disponibiulité LH* RS . Rapport de Thése . 4/10/2004

[M65] **Moore G. E.,** Cramming More Components Onto Integrated Circuits, Electronics, Vol. 38, Num. 8, April 1965

[MS97] **Mac W., Sloane F. J.** The Theory of Error Correcting Codes. Elsevier/North Holland, Amsterdam (1997).

[OAE+10] **Ousterhout J., Agrawal P., Erickson D., Kozyrakis C., Leverich J., Mazières D., Mitra S., Narayanan A., Parulkar G., Rosenblum M., Rumble S.M., Stratmann E., and Stutsman R. 2010.** The case for RAMClouds: scalable high-performance storage entirely in DRAM. SIGOPS Oper. Syst. Rev. 43, 4 (January 2010), 92-105. DOI=10.1145/1713254.1713276 http://doi.acm.org/10.1145/1713254.1713276

[PGK88] **Patterson D-A., Gibson G., and Katz R-H.. 1988.** A case for redundant arrays of inexpensive disks (RAID). SIGMOD Rec. 17, 3 (June 1988), 109-116. DOI=10.1145/971701.50214 http://doi.acm.org/10.1145/971701.50214

[RGRK04] **Rhea S., Geels D., Roscoe D., and Kubiatowicz J.. 2004.** Handling churn in a DHT. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04). USENIX Association, Berkeley, CA, USA, 0-10.

[S01] **Schollmeier R. 2001. [16]** A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In Proceedings of the First International Conference on Peer-to-Peer Computing (P2P '01). IEEE Computer Society, Washington, DC, USA, 101-.

[SG00] **Steven D, Gribble, Eric A, Brewer, Joseph M, Hellerstein, and Culler D**. Scalable, Distributed Data Structures for Internet Service Construction, Proceedings of the Fourth Symposium on Operating Systems Design and Implementation.(2000)

[SGG02] **Saroiu S., Gummadi P. K., and Gribble S. D.** A measurement study of peer-to-peer file sharing systems. In Proc. MMCN, Jan. 2002.

[SMK+01] **Stoica, Morris, Karger, Kaashoek, Balakrishman,** Chord : A Scalable Peer to Peer Lookup Service for Internet Appication, SIGCOMM'0, August 27-31, 2001, San Diego, California USA

[SR06] **Stutzbach D. and Rejaie R.. 2006.** Understanding churn in peer-to-peer networks. In Proceedings of the 6th ACM SIGCOMM conference on Internet measurement (IMC '06). ACM, New York, NY, USA, 189-202. DOI=10.1145/1177080.1177105 http://doi.acm.org/10.1145/1177080.1177105

[SW02] **Subhabrata Sen and Jia Wang. 2002.** Analyzing peer-to-peer traffic across large networks. In Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement (IMW '02). ACM, New York, NY, USA, 137-150. DOI=10.1145/637201.637222 http://doi.acm.org/10.1145/637201.637222

[T85] **Teradata Corporation.** DBC/ 1012 data base computer concepts and facilities. Technical Report Teradata Document C10-0001-02, Release 2.0, November, 1985

[VBW94] **Vingralek R., Breitbart Y., and Weikum G. 1994.** Distributed file organization with scalable cost/performance. In Proceedings of the 1994 ACM SIGMOD international conference on Management of data (SIGMOD '94), Richard Thomas Snodgrass and Marianne Winslett (Eds.). ACM, New York, NY, USA, 253-264. DOI=10.1145/191839.191889 http://doi.acm.org/10.1145/191839.191889

[VMW] http://www.vmware.com/products/application-platform/vfabric-gemfire

[WK02] **Weatherspoon H. and Kubiatowicz J. 2002.** Erasure Coding Vs. Replication: A Quantitative Comparison. In Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS '01), Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron

(Eds.). Springer-Verlag, London, UK, UK, 328-338.

XM03] **XIN, Q., Miller, E., Schwarz, T., Brandt, S., Long, D., Litwin, W.** (2003). Reliability mechanisms for very large storage systems. 20th IEEE mass storage systems and technologies (MSST 2003), p. 146-156. San Diego, CA. 2003.

[Y08] **Yakouben H. 2008.** Scalable and Distributed Linear Hashing for P2P Environment. 3rd International Conference for Internet Technology and Secured Transactions (ICTST), Dublin. June 22nd 2008

[YLS07] **Yakouben H., Litwin W., Schwarz T. 2007.** LH*$_{RS}^{P2P}$: A Scalable Distributed Data Structure for the P2P Environment. 3ème journée francophones sur les Entrepôts de Données et l'Analyse en ligne. EDA June 2007, Poitier. Extended Abstract.

[YLS08] **Yakouben H, Litwin W, Schwarz T**. **2008**. LH*$_{RS}^{P2P}$: a scalable distributed data structure for P2P environment. Proc of the 8th international conference on New technologies in distributed systems (NOTERE' 08). Lyon, France

[YS10] **Yakouben H. and Sahri S. 2010.** LH*$_{RS}^{P2P}$; a fast and high churn resistant scalable distributed data structure for P2P systems. Int. J. Internet Technol. Secur. Syst. 2, 1/2 (February 2010), 5-31. DOI=10.1504/IJITST.2010.031470 http://dx.doi.org/10.1504/IJITST.2010.031470

[Z04] **Zegour D.E.** Scalable distributed compact trie hashing (CTH*). Information & Software Technology 46(14): 923-935 (2004)